
FreeBSD Device Drivers

From First Steps to Kernel Mastery

Edson Brandi

Version 2.0 - April, 22th 2026

Contents

1	Introduction - From Curiosity to Contribution	2
1.1	My Journey: From Curiosity to Career	2
1.2	FreeBSD in Context	3
1.2.1	A Brief History	3
1.2.2	Why FreeBSD Is Special	4
1.2.3	Kernel Insight	5
1.2.4	Clearing Misconceptions	5
1.2.5	Real-World Uses	5
1.2.6	Why FreeBSD for Driver Development?	5
1.3	Drivers and the Kernel - A First Glimpse	6
1.3.1	Why Drivers Matter	6
1.3.2	Everyday Technology, Powered by Drivers	6
1.3.3	Beginner's Tip	6
1.3.4	A Teaser of FreeBSD's Kernel World	7
1.4	Your Path Through This Book	7
1.4.1	Who This Book Is For	7
1.4.2	Who This Book Isn't For	7
1.4.3	What You'll Learn and Gain	8
1.4.4	The Learning Path Ahead	8
1.5	How to Read This Book	8
1.5.1	Time and Commitment	8
1.5.2	Labs Are Strongly Recommended	9
1.5.3	A Suggested Cadence	9
1.5.4	Fast-Path Hints for Experienced Readers	9
1.5.5	What to Do When You Are Stuck	9
1.6	How to Get the Most from This Book	10
1.6.1	Build the Right Mindset	10
1.6.2	Approach Exercises Seriously	10
1.6.3	Expect Challenges and Learn from Mistakes	10
1.6.4	Beginner's Tip	11
1.6.5	Engage with the Community	11
1.6.6	Kernel Insight	11
1.6.7	Stay Motivated	11
1.7	Wrapping Up	11
2	Setting Up Your Lab	12
2.0.1	Reader Guidance: How to Use This Chapter	12
2.1	Why a Lab Environment Matters	13
2.1.1	Why Not Use Your Main Computer?	13
2.1.2	Virtual Machines: The Beginner's Best Friend	13
2.1.3	Bare Metal: When You Need the Real Thing	13
2.1.4	Real-World Example	14
2.1.5	Hands-On Lab: Preparing Your Lab Mindset	14
2.1.6	Wrapping Up	14

2.2	Choosing Your Setup: Virtual Machine or Bare Metal	14
2.2.1	Virtual Machines: Your Sandbox in a Box	15
2.2.2	Bare Metal: The Real Deal	15
2.2.3	Hybrid Strategy	16
2.2.4	Quick Comparison Table	16
2.2.5	Hands-On Lab: Deciding Your Path	16
2.2.6	Wrapping Up	16
2.3	Installing FreeBSD (VM and Bare Metal)	17
2.3.1	Downloading the FreeBSD ISO	17
2.3.2	Installing FreeBSD in VirtualBox (Step by Step)	17
2.3.3	Installing FreeBSD on Bare Metal	38
2.3.4	First Boot	41
2.3.5	Wrapping Up	42
2.4	First Boot and Initial Configuration	42
2.4.1	Logging In	43
2.4.2	Switching to the Root User	43
2.4.3	Setting the Hostname and Time	44
2.4.4	Networking Basics	45
2.4.5	Installing and Configuring sudo	45
2.4.6	Updating the System	46
2.4.7	Creating a Comfortable Environment	47
2.4.8	Basic Hardening for Your Lab	47
2.4.9	Wrapping Up	49
2.5	Preparing the System for Development	49
2.5.1	Installing Development Tools	49
2.5.2	Choosing an Editor	49
2.5.3	Accessing Documentation	51
2.5.4	Installing the FreeBSD Source Tree	51
2.5.5	Testing Your Toolkit: A “Hello Kernel Module”	52
2.5.6	Hands-On Lab: Verifying Your Development Setup	54
2.5.7	Wrapping Up	54
2.6	Using Snapshots and Backups	54
2.6.1	Snapshots in Virtual Machines	55
2.6.2	Snapshots on Bare Metal	57
2.6.3	Backing Up Your Work	58
2.6.4	Hands-On Lab: Breaking and Fixing	58
2.6.5	Wrapping Up	59
2.7	Setting Up Version Control	59
2.7.1	Why Version Control Matters	59
2.7.2	Installing Git	59
2.7.3	Configuring Git (Your Identity)	60
2.7.4	Creating Your First Repository	60
2.7.5	Best Practices for Commits	60
2.7.6	Using Remote Repositories (Optional)	61
2.7.7	Hands-On Lab: Version Control for Your Driver	61
2.7.8	Wrapping Up	61
2.8	Documenting Your Work	61
2.8.1	Why Documentation Matters	62
2.8.2	Writing a Simple README	62
2.8.3	Using Commit Messages as Documentation	62
2.8.4	Keeping a Lab Logbook	63

2.8.5	Tools to Help	63
2.8.6	Hands-On Lab: Documenting Your First Module	63
2.8.7	Wrapping Up	64
2.9	Wrapping Up	64
2.9.1	Exercises	64
2.9.2	Looking Ahead	65
3	A Gentle Introduction to UNIX	66
3.1	Reader Guidance: How to Use This Chapter	66
3.2	Introduction: Why UNIX Matters	66
3.2.1	Why Should You Learn UNIX Before Writing Drivers?	67
3.2.2	What You Will Learn in This Chapter	67
3.2.3	The Bridge to Device Drivers	67
3.2.4	Wrapping Up	68
3.3	What Is UNIX?	68
3.3.1	A Brief History of UNIX	68
3.3.2	The UNIX Philosophy	68
3.3.3	UNIX-like Systems Today	69
3.3.4	Key Concepts and Terms	69
3.3.5	How UNIX Differs from Windows	70
3.3.6	Everyday UNIX in Your Life	70
3.3.7	Hands-On Lab: Your First UNIX Commands	70
3.3.8	Wrapping Up	71
3.4	The Shell: Your Window Into FreeBSD	71
3.4.1	What Is a Shell?	72
3.4.2	How to Know Which Shell You're Using	72
3.4.3	The Structure of a Command	73
3.4.4	Essential Commands for Beginners	73
3.4.5	Tips and Shortcuts	76
3.4.6	Hands-On Lab: Navigating and Managing Files	78
3.4.7	Wrapping Up	78
3.5	The FreeBSD Filesystem Layout	79
3.5.1	Devices as Files: /dev	80
3.5.2	Absolute vs. Relative Paths	80
3.5.3	Hands-On Lab: Exploring the Filesystem	81
3.5.4	Wrapping Up	82
3.6	Users, Groups, and Permissions	82
3.6.1	Users and Groups	82
3.6.2	File Ownership	83
3.6.3	Permissions	83
3.6.4	Changing Permissions	84
3.6.5	Changing Ownership	84
3.6.6	Practical Scenario: Project Directory	85
3.6.7	Hands-On Lab: Permissions in Action	85
3.6.8	Wrapping Up	86
3.7	Processes and System Monitoring	86
3.7.1	What Is a Process?	86
3.7.2	Foreground vs. Background Processes	86
3.7.3	Viewing Processes	87
3.7.4	Stopping Processes	89
3.7.5	Monitoring System Resources	91
3.7.6	Hands-On Lab: Working with Processes	91

3.7.7	Wrapping Up	92
3.8	Installing and Managing Software	92
3.8.1	Binary Packages with pkg	92
3.8.2	The FreeBSD Ports Collection	93
3.8.3	Where Installed Software Goes	96
3.8.4	Practical Example: Installing vim and htop	96
3.8.5	Hands-On Lab: Managing Software	97
3.8.6	Wrapping Up	97
3.9	Keeping FreeBSD Up to Date	98
3.9.1	Why Updates Matter	98
3.9.2	The freebsd-update Tool	98
3.9.3	The Update Workflow	98
3.9.4	Example Session	99
3.9.5	Kernel Updates with freebsd-update	100
3.9.6	Upgrading to a New Release with freebsd-update	100
3.9.7	Hands-On Lab: Running Your First Update	101
3.9.8	Wrapping Up	101
3.10	Scheduling and Automation	101
3.10.1	Why Automate Tasks?	102
3.10.2	cron: The Automation Workhorse	102
3.10.3	Understanding the crontab Format	102
3.10.4	Editing and Managing Crontabs	103
3.10.5	Where Do Logs Go?	103
3.10.6	at: One-Time Scheduling	103
3.10.7	periodic: FreeBSD's Maintenance Helper	104
3.10.8	Hands-On Lab: Automating Tasks	107
3.10.9	Common Pitfalls for Beginners	108
3.10.10	Why This Matters for Driver Developers	108
3.10.11	Wrapping Up	108
3.11	Introduction to Shell Scripting	109
3.11.1	1) Your first script: shebang, make it executable, run it	109
3.11.2	2) Variables and quoting	110
3.11.3	3) Exit status and short circuit operators	110
3.11.4	4) Tests and conditions: if, [], files and numbers	111
3.11.5	5) Loops: for and while	112
3.11.6	6) Case statements for tidy branching	112
3.11.7	7) Functions to organize your script	113
3.11.8	8) A practical example: a tiny backup script	114
3.11.9	9) Working with temporary files safely	115
3.11.10	10) Debugging your scripts	116
3.11.11	11) Putting it together: organize downloads by type	116
3.11.12	Hands-on Lab: three mini tasks	118
3.11.13	Common beginner pitfalls and how to avoid them	118
3.11.14	Wrapping up	118
3.12	Shell Portability: Handling Edge Cases and bash vs sh	118
3.12.1	The Problem: Filenames with Special Characters	119
3.12.2	A Naïve Approach That Breaks	119
3.12.3	The bash Solution: Using Null Delimiters	120
3.12.4	The POSIX-Compliant Alternative	121
3.12.5	Understanding the Trade-Off	122
3.12.6	When to Choose bash	122

3.12.7	A Third Option: <code>find -exec</code>	122
3.12.8	Practical Advice	123
3.12.9	Wrapping Up	123
3.13	Seeking Help and Documentation in FreeBSD	124
3.13.1	The Power of man Pages	124
3.13.2	The FreeBSD Handbook	126
3.13.3	Community and Support	126
3.13.4	Hands-On Lab: Exploring Documentation	128
3.13.5	Wrapping Up	128
3.14	Peeking into the Kernel and System State	128
3.14.1	<code>dmesg</code> : Reading the Kernel's Diary	129
3.14.2	<code>sysctl</code> : The Kernel's Control Panel	130
3.14.3	<code>/dev</code> : Where Devices Come to Life	131
3.14.4	Hands-On Lab: Your First Peek Inside	132
3.14.5	From Shell to Hardware: The Big Picture	132
3.14.6	Common Beginner Pitfalls	133
3.14.7	Wrapping Up	134
3.15	Wrapping Up	135
3.15.1	Practice Ground	135
3.15.2	Filesystem and Navigation (8 exercises)	136
3.15.3	Users, Groups, and Permissions (6 exercises)	136
3.15.4	Processes and System Monitoring (7 exercises)	136
3.15.5	Installing and Managing Software (<code>pkg</code> and Ports) (6 exercises)	136
3.15.6	Automation and Scheduling (<code>cron</code> , <code>at</code> , <code>periodic</code>) (6 exercises)	137
3.15.7	Shell Scripting (<code>/bin/sh</code>) (7 exercises)	137
3.15.8	Peeking into the Kernel (<code>dmesg</code> , <code>sysctl</code> , <code>/dev</code>) (6 exercises)	137
3.15.9	Wrapping Up	137
3.15.10	Looking Ahead	138
4	A First Look at the C Programming Language	139
4.1	Reader Guidance: How to Use This Chapter	139
4.1.1	How to Get the Most Out of This Chapter	139
4.2	Introduction	140
4.2.1	What is C?	140
4.2.2	Why Should I Learn C for FreeBSD?	140
4.2.3	What If I've Never Programmed Before?	141
4.2.4	How Is This Chapter Organized?	141
4.3	Setting Up Your Environment	142
4.3.1	Installing a C Compiler on FreeBSD	142
4.3.2	Behind the Scenes: The Compilation Pipeline	143
4.3.3	Using Makefiles	144
4.3.4	Installing the FreeBSD Source Code	145
4.3.5	Summary	145
4.4	Anatomy of a C Program	146
4.4.1	The Basic Structure	146
4.4.2	<code>#include</code> Directives: Adding Libraries	146
4.4.3	The <code>main()</code> Function: Where Execution Begins	148
4.4.4	Statements and Function Calls	148
4.4.5	Return Values	149
4.4.6	Bonus Learning Point About Return Values	149
4.4.7	Putting It All Together	150
4.4.8	A First Glimpse at Good Practices in C	151

4.4.9	Summary	152
4.5	Variables and Data Types	152
4.5.1	What Is a Variable?	153
4.5.2	Declaring Variables	153
4.5.3	Common C Data Types	154
4.5.4	Type Qualifiers	154
4.5.5	Constant Values and <code>#define</code>	154
4.5.6	Why Use <code>#define</code> for Constants?	155
4.5.7	Real Example From FreeBSD	155
4.5.8	Watch Out: There Is No Type Checking	155
4.5.9	Best Practices for <code>#define</code> Constants in Kernel Development	155
4.5.10	Best Practices for Variables	156
4.5.11	Summary	156
4.6	Operators and Expressions	157
4.6.1	What Is an Expression?	157
4.6.2	Arithmetic Operators	157
4.6.3	Comparison Operators	158
4.6.4	Logical Operators	158
4.6.5	Assignment and Compound Assignment	159
4.6.6	Bitwise Operators	159
4.6.7	Real Example from FreeBSD: <code>sys/kern/tty_info.c</code>	159
4.6.8	Summary	161
4.7	Control Flow	162
4.7.1	Understanding the <code>if</code> , <code>else</code> , and <code>else if</code>	162
4.7.2	Understanding the <code>switch</code> and <code>case</code>	164
4.7.3	Understanding the <code>for</code> Loops	165
4.7.4	Understanding the <code>while</code> Loop	166
4.7.5	Understanding <code>do...while</code> Loops	172
4.7.6	Understanding <code>break</code> and <code>continue</code>	173
4.7.7	Pro Tip: Always Use Braces <code>{}</code>	177
4.7.8	Summary	177
4.8	Functions	178
4.8.1	Defining and Declaring Functions	179
4.8.2	Calling Functions	184
4.8.3	Functions with No Return Value: <code>void</code>	187
4.8.4	Function Declarations (Prototypes)	189
4.8.5	Why prototypes matter	190
4.8.6	Prototypes in header files	190
4.8.7	FreeBSD driver perspective	191
4.8.8	Try It Yourself: Moving a Function Below <code>main()</code>	192
4.8.9	FreeBSD Driver Connection	193
4.8.10	Tip for Kernel Code	197
4.8.11	Real Example from the FreeBSD 14.3 Source Tree: <code>device_printf()</code>	198
4.8.12	Variable Scope in Functions	202
4.8.13	Variable Storage Duration	206
4.8.14	The Three Main Storage Durations in C	206
4.8.15	Bridging Theory and Practice	209
4.8.16	Real Example from FreeBSD 14.3	209
4.8.17	Why Storage Duration Matters in FreeBSD Drivers	212
4.8.18	Common Beginner Mistakes	212
4.8.19	Wrapping Up	213

4.8.20	Variable Linkage (Visibility Across Files)	213
4.8.21	The three kinds of linkage in C	213
4.8.22	A tiny two-file illustration	214
4.8.23	Real FreeBSD 14.3 Example: External vs. Internal vs. No Linkage	215
4.8.24	Why this matters for drivers	218
4.8.25	Common beginner pitfalls (and how to sidestep them)	218
4.8.26	Wrapping up	219
4.9	Parameters Are Copies	219
4.9.1	A Simple Example: Modifying a Copy	219
4.9.2	A Real Example from FreeBSD 14.3	220
4.9.3	Why This Matters in FreeBSD Driver Development	222
4.9.4	Common Beginner Mistakes	222
4.9.5	Wrapping Up	223
4.10	Arrays and Strings in C	223
4.10.1	Declaring and Using Arrays	223
4.10.2	Strings in C	224
4.10.3	Common String Functions (<string.h>)	224
4.10.4	Why This Matters in FreeBSD Drivers	225
4.10.5	Mini Lab: The Danger of a Missing '\0'	228
4.10.6	Real Example from FreeBSD 14.3 Source Code	229
4.10.7	Common Beginner Pitfalls	231
4.10.8	Hands-On Lab 1: Arrays in Practice	232
4.10.9	Hands-On Lab 2: Strings and the Null Terminator	233
4.10.10	Wrapping Up	235
4.10.11	Recap Quiz - Arrays and Strings	235
4.10.12	Challenge Exercises	235
4.10.13	Looking Ahead	235
4.11	Pointers and Memory	236
4.11.1	What Is a Pointer?	236
4.11.2	Declaring and Using Pointers	242
4.11.3	Pointers and Functions	247
4.11.4	Pointers and Arrays: A Natural Pair	250
4.11.5	Pointer Arithmetic and Boundaries	254
4.11.6	Pointers to Structs	262
4.11.7	Arrays of Pointers and Pointer Arrays	273
4.12	Dynamic Memory Allocation	278
4.12.1	User Space vs Kernel Space	278
4.12.2	Visualising Memory in User Space	279
4.12.3	Kernel-Space Allocation with malloc(9)	279
4.12.4	Kernel malloc(9) Workflow	280
4.12.5	Memory Types and Flags	281
4.12.6	Hands-On Lab 1: Allocating and Freeing a Buffer	281
4.12.7	Hands-On Lab 2: Allocating an Array of Structures	283
4.12.8	Real FreeBSD Example: Building the corefile path in kern_sig.c	283
4.12.9	Why This Matters in FreeBSD Device Drivers	287
4.12.10	Common Beginner Pitfalls	288
4.12.11	Challenge Questions	289
4.12.12	Wrapping Up	289
4.13	Memory Safety in Kernel Code	289
4.13.1	What Can Go Wrong?	289
4.13.2	Best Practices for Safer Kernel Code	290

4.13.3	A Real Example from FreeBSD 14.3	291
4.13.4	Dangerous Code to Avoid	291
4.13.5	The Correct Pattern	291
4.13.6	Why this pattern matters	293
4.13.7	If your context do not allow sleep	293
4.13.8	Hands-On Lab 1: Crashing with an Uninitialized Pointer	293
4.13.9	Hands-On Lab 2: Memory Leak and the Forgotten Free	296
4.13.10	Detecting Memory Leaks with AddressSanitizer	298
4.13.11	Mini-Lab 3: Memory Allocation in a Kernel Module	299
4.13.12	Common Beginner Pitfalls	301
4.13.13	Golden Rules for Kernel Memory	302
4.13.14	Pointers Recap: The Lifecycle of Memory in C	302
4.13.15	Pointers Recap Quiz	303
4.13.16	Wrapping Up	304
4.14	Structures and typedef in C	304
4.14.1	What is a struct?	305
4.14.2	Accessing Structure Members	306
4.14.3	Making Code Cleaner with typedef	307
4.14.4	Typedefs in FreeBSD	308
4.14.5	Why this matters in driver development	308
4.14.6	Real Example from FreeBSD 14.3: Creating a USB Character Device	309
4.14.7	Why this example matters	310
4.14.8	Takeaways	310
4.14.9	Hands-On Lab 1: Building a TTY Device Struct	311
4.14.10	Hands-On Lab 2: Using typedef for Cleaner Code	312
4.14.11	Extra Hands-On Lab: Managing Multiple TTY Devices	313
4.14.12	What did you learn in these labs?	316
4.14.13	Common Beginner Pitfalls with Structs and typedefs	316
4.14.14	Recap Quiz: Structs and typedef Pitfalls	318
4.14.15	Wrapping Up	318
4.15	Header Files and Modular Code	319
4.15.1	What Is a Header File?	319
4.15.2	Header Guards: Why They Exist and How They Work	319
4.15.3	Why the Name Matters	320
4.15.4	A Real FreeBSD Example	320
4.15.5	Hands-On Lab 1: Seeing Header Guards in Action	321
4.15.6	Why Use Modular Code?	322
4.15.7	A Simple Multi-File Example	323
4.15.8	How FreeBSD Uses Headers (Example from FreeBSD 14.3)	324
4.15.9	Common Beginner Pitfalls with Headers	327
4.15.10	Hands-On Lab 2: Triggering Pitfalls on Purpose	328
4.15.11	Recap Quiz: Header Pitfalls	331
4.15.12	Hands-On Lab 3: Your First Modular Program	331
4.15.13	Why This Matters for Driver Development	333
4.15.14	Wrapping Up	334
4.16	Compiling, Linking, and Debugging C Programs	334
4.16.1	The compilation pipeline, in plain words	334
4.16.2	Reading compiler messages with care	335
4.16.3	Multi-file programs and linker errors	336
4.16.4	Why make matters in real projects	337
4.16.5	Hands-On Lab 1: When make misses a change	338

4.16.6	Recap Quiz: Why make Matters	340
4.16.7	Debugging with GDB: more than just “fixing bugs”	341
4.16.8	Hands-On Lab 2: Fixing a Logic Bug with GDB	342
4.16.9	Hands-On Lab 3: Chasing a Segmentation Fault with GDB	344
4.16.10	Recap Quiz: Debugging Segmentation Faults with GDB	345
4.16.11	A peek into real FreeBSD build rules	346
4.16.12	Common beginner pitfalls	351
4.16.13	Why this matters for FreeBSD driver development	356
4.16.14	Wrapping up	356
4.17	The C Preprocessor: Directives Before Compilation	356
4.17.1	What the Preprocessor Does	357
4.17.2	A Quick Example	357
4.17.3	Real Example from FreeBSD 14.3 Source	358
4.17.4	Common Preprocessor Directives	359
4.17.5	Hands-On Lab 1: Protecting Headers with Include Guards	362
4.17.6	Hands-On Lab 2: Enforcing Build Requirements with <code>#error</code>	363
4.17.7	Common Beginner Pitfalls with the Preprocessor	364
4.17.8	Why This Matters for FreeBSD Driver Development	365
4.17.9	Recap Quiz: The C Preprocessor	366
4.17.10	Wrapping Up	367
4.18	Good Practices for C Programming	367
4.18.1	Readability First: Indentation, Names, and Comments	367
4.18.2	Small, Focused Functions and Early Returns	371
4.18.3	Check Return Values and Propagate Errors	373
4.18.4	Mini Lab: Check, Log, Clean Up, Propagate	375
4.18.5	<code>const</code> Signals Intent	382
4.18.6	Mini Lab: Making Interfaces Safer with <code>const</code>	383
4.18.7	Common Beginner Pitfalls	385
4.18.8	Mini Lab: Spot the Pitfalls	387
4.18.9	Keep Whitespace and Braces Consistent	390
4.18.10	Clarity Beats Cleverness	392
4.18.11	FreeBSD Style, Linters, and Editor Helpers	392
4.18.12	Hands-On Lab 1: Make It KNF-Clean	392
4.18.13	Real-Code Tour: KNF in the FreeBSD 14.3 Tree	393
4.18.14	Hands-On Lab 2: Style Hunt in the Kernel Tree	394
4.18.15	Hands-On Lab 3: Refactor and Harden an Attach Path	397
4.18.16	Hands-On Lab 4: Putting It All Together	399
4.18.17	Why This Matters for FreeBSD Drivers	400
4.18.18	Wrapping Up	401
4.18.19	Style Hunt Worksheet: From Observation to Practice	401
4.18.20	Style Hunt Worksheet	401
4.19	Final Practice Labs	404
4.19.1	Lab 1 (Easy): Bit Flags and Enums - “Device State Inspector”	404
4.19.2	Lab 2 (Easy → Medium): Preprocessor Hygiene - “Feature-Gated Logging”	406
4.19.3	Lab 3 (Medium): Safe Strings and Arrays - “Bounded Device Names”	408
4.19.4	Lab 4 (Medium → Hard): Function Pointer Dispatch - “Mini devsw”	411
4.19.5	Lab 5 (Hard): Fixed-Size Circular Buffer - “Single-Producer Single-Consumer Queue”	413
4.19.6	Wrapping Up	416
4.20	Final Knowledge Check	417
4.20.1	Questions	417
4.21	Wrapping Up	419

5	Understanding C for FreeBSD Kernel Programming	420
5.1	Reader Guidance: How to Use This Chapter	420
5.1.1	How to Get the Most Out of This Chapter	421
5.2	Introduction	421
5.2.1	Why Kernel C Is Different	421
5.2.2	The Mental Shift	422
5.2.3	What You'll Learn	422
5.3	Kernel-Specific Data Types	422
5.3.1	Why Standard C Types Aren't Enough	422
5.3.2	Fixed-Size Integer Types	423
5.3.3	System-Specific Size Types	423
5.3.4	Pointer and Address Types	424
5.3.5	Time and Timing Types	425
5.3.6	Device and Resource Types	425
5.3.7	Boolean and Status Types	425
5.3.8	Hands-On Lab: Exploring Kernel Types	426
5.3.9	Common Type Mistakes to Avoid	427
5.3.10	Summary	428
5.4	Memory in Kernel Space	428
5.4.1	The Kernel Memory Landscape	429
5.4.2	malloc(9): The Kernel's Memory Allocator	429
5.4.3	Memory Types: Organizing Allocations	430
5.4.4	Allocation Flags: Controlling Behavior	430
5.4.5	The Critical Difference: Sleep vs. No-Sleep Contexts	431
5.4.6	Memory Zones: High-Performance Allocation	431
5.4.7	Stack Considerations: The Kernel's Precious Resource	432
5.4.8	Memory Barriers and Cache Coherency	433
5.4.9	Hands-On Lab: Kernel Memory Management	434
5.4.10	Memory Debugging and Leak Detection	436
5.4.11	Safe String and Memory Operations in Kernel Space	436
5.4.12	Summary	438
5.5	Error Handling Patterns in Kernel C	438
5.5.1	Return Values: Zero Means Success	438
5.5.2	Using goto for Cleanup Paths	439
5.5.3	Defensive Strategy	440
5.5.4	Summary	440
5.6	Assertions and Diagnostics in the Kernel	440
5.6.1	KASSERT() - Enforcing Invariants	440
5.6.2	panic() - The Last Resort	441
5.6.3	printf() and Friends	441
5.6.4	Tracing with CTRn() and SDT_PROBE()	441
5.6.5	Summary	442
5.7	Strings and Buffers in the Kernel	442
5.7.1	Why Standard String Functions Don't Work	442
5.7.2	Safe String Copying: strncpy() and strlcat()	442
5.7.3	String Length and Validation: strlen() and strnlen()	443
5.7.4	Memory Operations: memcpy(), memset(), and memcmp()	444
5.7.5	User Space Data Access: copyin() and copyout()	444
5.7.6	String Formatting: sprintf() vs snprintf()	445
5.7.7	Buffer Management: mbuf Chains	446
5.7.8	Hands-On Lab: Safe String Handling	446

5.7.9	String Handling Best Practices	448
5.7.10	Common String Pitfalls	449
5.7.11	Summary	449
5.8	Functions and Return Conventions	449
5.8.1	The Kernel's Function Signature Patterns	450
5.8.2	Error Return Conventions	450
5.8.3	Parameter Patterns and Conventions	451
5.8.4	Output Parameters and Return Values	452
5.8.5	Function Naming Conventions	453
5.8.6	Static Functions vs. External Functions	453
5.8.7	Inline Functions vs. Macros	454
5.8.8	Function Documentation and Commenting	454
5.8.9	Hands-On Lab: Function Design Patterns	455
5.8.10	Function Design Best Practices	459
5.8.11	Summary	459
5.9	Restrictions and Pitfalls of Kernel C	460
5.9.1	The Floating-Point Restriction	460
5.9.2	Stack Size Limitations	461
5.9.3	Sleep Restrictions: Atomic vs. Preemptible Context	462
5.9.4	Recursion Limitations	464
5.9.5	Global Variables and Thread Safety	465
5.9.6	Memory Allocation Context Awareness	466
5.9.7	Performance Considerations	467
5.9.8	Hands-On Lab: Understanding Restrictions	467
5.9.9	Summary	470
5.9.10	Atomic Operations and Inline Functions	470
5.9.11	Why Atomic Operations Matter	470
5.9.12	FreeBSD's Atomic Operations	471
5.9.13	Memory Barriers and Ordering	472
5.9.14	Compare-and-Swap: The Building Block	473
5.9.15	Inline Functions for Performance	474
5.9.16	When to Use Inline Functions	475
5.9.17	Combining Atomics and Inlines	475
5.9.18	Hands-On Lab: Atomic Operations and Performance	476
5.9.19	Performance Considerations	479
5.9.20	Summary	479
5.10	Coding Idioms and Style in Kernel Development	479
5.10.1	FreeBSD Kernel Normal Form (KNF)	479
5.10.2	Error Handling Patterns	481
5.10.3	Resource Management Patterns	482
5.10.4	Locking Patterns	483
5.10.5	Assertion and Debugging Patterns	484
5.10.6	Memory Allocation Patterns	485
5.10.7	Function Naming and Organization	486
5.10.8	Hands-On Lab: Implementing Kernel Coding Patterns	486
5.10.9	Key Takeaways from Kernel Coding Style	492
5.10.10	Summary	493
5.11	Defensive C in the Kernel	493
5.11.1	The Paranoid Mindset	493
5.11.2	Input Validation: Trust No One	495
5.11.3	Integer Overflow Prevention	496

5.11.4	Buffer Management and Bounds Checking	497
5.11.5	Error Propagation Patterns	498
5.11.6	Race Condition Prevention	499
5.11.7	Resource Leak Prevention	499
5.11.8	Assertions for Development	501
5.11.9	Hands-On Lab: Building Defensive Kernel Code	502
5.11.10	Summary of Defensive Programming Principles	509
5.11.11	Kernel Attributes and Error Handling Idioms	509
5.11.12	Compiler Attributes for Kernel Safety	509
5.11.13	Error Code Conventions and Patterns	511
5.11.14	Debugging and Diagnostic Idioms	514
5.11.15	Performance Monitoring Idioms	516
5.11.16	Hands-On Lab: Error Handling and Diagnostics	517
5.11.17	Summary	524
5.12	Real-World Kernel Code Walkthrough	524
5.12.1	A Simple Character Device Driver: <code>/dev/null</code>	524
5.12.2	Memory Allocation in Action: <code>malloc(9)</code> Implementation	525
5.12.3	Network Packet Processing: IP Input	526
5.12.4	Device Driver Initialization: PCI Bus Driver	527
5.12.5	Synchronization in Practice: Reference Counting	528
5.12.6	What We've Learned from Real Code	529
5.12.7	Summary	529
5.13	Hands-On Labs (Beginner Kernel C)	530
5.13.1	Lab Prerequisites	530
5.13.2	Lab 1: Safe Memory Allocation and Cleanup	530
5.13.3	Lab 2: User-Kernel Data Exchange	535
5.13.4	Lab 3: Driver-Safe Logging and Device Context	541
5.13.5	Lab 4: Error Handling and Graceful Failures	548
5.13.6	Lab Summary: Mastering the Kernel C Dialect	558
5.14	Wrapping Up	559
5.14.1	What You've Accomplished	560
5.14.2	The Dialect Mastery	560
5.14.3	From Dialect to Fluency	561
5.14.4	A Personal Reflection	561
5.14.5	The Next Chapter: From Language to Structure	561
5.14.6	The Foundation Is Complete	562
5.15	Challenge Exercises: Practising the Kernel C Mindset	562
5.15.1	Challenge 1: Trace the Data Type Origins	562
5.15.2	Challenge 2: Memory Allocation Scenarios	562
5.15.3	Challenge 3: Error-Handling Discipline	563
5.15.4	Challenge 4: Safe String Operations	563
5.15.5	Challenge 5: Diagnostics and Assertions	563
5.15.6	What You'll Gain	563
5.16	Summary Reference: User Space vs. Kernel Space Equivalents	563
5.16.1	Key Takeaways	565
6	The Anatomy of a FreeBSD Driver	566
6.1	Introduction	566
6.1.1	What This Chapter <i>IS</i>	566
6.1.2	What This Chapter <i>Is Not</i>	567
6.1.3	Where This Chapter Fits	567

6.2	Reader Guidance: How to Use This Chapter	568
6.2.1	Estimated Time Investment	568
6.2.2	What to Have Ready	568
6.2.3	Pacing and Approach	568
6.2.4	Managing Your Curiosity	569
6.2.5	Success Criteria	569
6.3	How to Get the Most Out of This Chapter	570
6.3.1	Keep <code>/usr/src</code> Nearby	570
6.3.2	Type the Micro-Snippets Yourself	571
6.3.3	Treat Labs as Checkpoints	571
6.3.4	Defer the Deep Mechanics	572
6.3.5	Use <code>man 9</code> as Your Reference	572
6.3.6	Skim Code Before Reading Explanations	573
6.3.7	Visualize as You Read	573
6.3.8	Study Patterns Across Multiple Drivers	573
6.3.9	Set Realistic Expectations	574
6.3.10	When to Take Breaks	574
6.3.11	You’re Building a Foundation	574
6.4	The Big Picture: How FreeBSD Sees Devices and Drivers	574
6.4.1	One-Screen Driver Lifecycle	574
6.4.2	Devices, Drivers, and Devclasses	575
6.4.3	The Bus Hierarchy and Newbus (One Page)	576
6.4.4	From Kernel to <code>/dev</code> : What <code>devfs</code> Presents	577
6.4.5	Your Manual-Page Map (Read, Don’t Memorize)	578
6.5	Driver Families: Choosing the Right Shape	580
6.5.1	Character Devices	580
6.5.2	Storage via GEOM (Why “Block Devices” Are Different Here)	581
6.5.3	Network via <code>ifnet</code>	583
6.5.4	Pseudo and Clone Devices (Safe, Small, Instructive)	584
6.5.5	Decision Checklist: Which Shape Fits?	585
6.5.6	Mini Exercise: Classify Live Drivers	585
6.6	The Minimal Driver Skeleton	587
6.6.1	Core Types: <code>device_t</code> and the <code>softc</code>	587
6.6.2	Method Tables and <code>kobj</code> - Why Callbacks Look “Magical”	589
6.6.3	Registration Macros You’ll Always Meet	590
6.6.4	Finding Your State and Speaking Clearly	592
6.6.5	Build & Load a Stub Safely (Preview Only)	593
6.7	The Newbus Lifecycle: From Discovery to Goodbye	595
6.7.1	Where Enumeration Comes From	595
6.7.2	<code>probe</code> : “Am I Your Driver?”	596
6.7.3	<code>attach</code> : “Prepare to Operate”	599
6.7.4	<code>detach</code> and shutdown: “Leave No Footprints”	601
6.7.5	The Failure-Unwinding Pattern	604
6.7.6	Watching the Lifecycle in Logs	605
6.8	Character Device Entry Points: Your I/O Surface	607
6.8.1	<code>cdev</code> and <code>cdevsw</code> : The Routing Table	607
6.8.2	<code>open/close</code> : Sessions and Per-open State	608
6.8.3	<code>read/write</code> : Moving Bytes Safely	610
6.8.4	<code>ioctl</code> : Control Paths	612
6.8.5	<code>poll/kqfilter</code> : Readiness Notifications	615
6.8.6	<code>mmap</code> : When Mapping Makes Sense	616

6.8.7	Back-pointers (si_drv1, etc.)	617
6.8.8	Permissions and Ownership	617
6.9	Alternative Surfaces: Network and Storage (Fast Orientation)	619
6.9.1	A First Look at ifnet	619
6.9.2	A First Look at GEOM	621
6.9.3	Mixed Models (tun as a Bridge)	623
6.9.4	What Comes Later	624
6.10	Resources and Registers: A Safe Preview	625
6.10.1	Claiming Resources (bus_alloc_resource_*)	625
6.10.2	Talking to Hardware with bus_space	626
6.10.3	Interrupts in Two Sentences	628
6.10.4	DMA in Two Sentences	628
6.10.5	Concurrency Note	629
6.11	Creating and Removing Device Nodes	630
6.11.1	make_dev/make_dev_s: Creating /dev/foo	630
6.11.2	Minors and Naming Conventions	631
6.11.3	destroy_dev: Cleaning Up	632
6.11.4	devctl/devmatch: Runtime Events	633
6.12	Module Packaging & Lifecycle (Load, Init, Unload)	634
6.12.1	What a Kernel Module (.ko) Is	634
6.12.2	The Module Event Handler	635
6.12.3	Declaring Dependencies & Versions	638
6.12.4	kldload/kldunload Flow and Logs	639
6.12.5	Troubleshooting Loads	640
6.13	Logging, Errors, and User-Facing Behaviour	642
6.13.1	Logging Etiquette (device_printf, rate-limiting hints)	642
6.13.2	Return Codes and Conventions	645
6.13.3	Lightweight Observability with sysctl	647
6.14	Read-Only Tour of Tiny Real Drivers (FreeBSD 14.3)	649
6.14.1	Tour 1 - The canonical character trio /dev/null, /dev/zero, and /dev/full	649
6.14.2	Tour 2 - A tiny write-only control surface with timers: led(4)	677
6.14.3	Tour 3 - A pseudo-NIC that is also a character device: tun(4)/tap(4):	734
6.14.4	Tour 4 - The PCI glue: uart(4)	785
6.15	From Four Drivers to One Mental Model	821
6.15.1	The Progression You've Completed	821
6.15.2	Patterns That Emerged	821
6.15.3	What You Can Now Recognize	823
6.15.4	The Blueprint Ahead	824
6.16	Driver Anatomy Blueprint (FreeBSD 14.3)	824
6.16.1	Core Skeleton: What Every Driver Needs	824
6.16.2	Character Device Blueprint	825
6.16.3	Network Pseudo-Interface Blueprint	825
6.16.4	PCI Glue Blueprint	826
6.16.5	Locking & Concurrency Cheatsheet	827
6.16.6	Data Movement Patterns	827
6.16.7	Common Patterns From the Tours	828
6.16.8	Errors, Edge Cases, and User Experience	829
6.16.9	Minimal Templates	829
6.16.10	Pre-Lab Self-Check (2 Minutes)	832
6.16.11	After-Lab Reflection (5 minutes)	833
6.16.12	Common Pitfalls and How to Avoid Them	833

6.16.13	Debugging Pitfall Patterns	851
6.16.14	Prevention Checklist	852
6.16.15	When Things Go Wrong	853
6.16.16	You're Ready: From Patterns to Practice	853
6.17	Hands-On Labs: From Reading to Building (Beginner-Safe)	854
6.17.1	Lab Design Philosophy	854
6.17.2	Prerequisites for All Labs	854
6.17.3	Time Commitment	854
6.18	Lab 1: Explore the Driver Map (Read-Only Scavenger Hunt)	855
6.18.1	Goal	855
6.18.2	What You'll Learn	855
6.18.3	Prerequisites	855
6.18.4	Time Estimate	855
6.18.5	Instructions	855
6.18.6	Check Your Answers	858
6.18.7	Success Criteria	859
6.18.8	What You Learned	859
6.18.9	Lab Logbook Entry Template	860
6.19	Lab 2: Minimal Module with Logs Only	860
6.19.1	Goal	860
6.19.2	What You'll Learn	860
6.19.3	Prerequisites	860
6.19.4	Time Estimate	861
6.19.5	Instructions	861
6.19.6	Behind the Scenes: What Just Happened?	868
6.19.7	Troubleshooting Guide	868
6.19.8	Success Criteria	870
6.19.9	What You Learned	870
6.19.10	Lab Logbook Entry Template	871
6.19.11	Optional Experiment: Module Load Order	871
6.20	Lab 3: Create and Remove a Device Node	872
6.20.1	Goal	872
6.20.2	What You'll Learn	872
6.20.3	Prerequisites	872
6.20.4	Time Estimate	872
6.20.5	Instructions	872
6.20.6	Behind the Scenes: The Complete Path	887
6.20.7	Concept Deep-Dive: Why uiomove()?	890
6.20.8	Success Criteria	891
6.20.9	What You Learned	891
6.20.10	Common Mistakes and How to Avoid Them	892
6.20.11	Lab Logbook Entry Template	893
6.21	Lab 4: Error Handling and Defensive Programming	894
6.21.1	Goal	894
6.21.2	What You'll Learn	894
6.21.3	Prerequisites	894
6.21.4	Time Estimate	894
6.21.5	Important Safety Note	895
6.21.6	Part 1: The Resource Leak Bug	895
6.21.7	Part 2: The Wrong Order Bug	897
6.21.8	Part 3: The NULL Pointer Bug	899

6.21.9	Part 4: The Allocation Failure Bug	901
6.21.10	Part 5: Complete Example with Full Error Handling	903
6.21.11	Debugging Checklist: Finding Driver Bugs	904
6.21.12	Success Criteria	905
6.21.13	What You Learned	906
6.21.14	Lab Logbook Entry Template	907
6.22	Labs Summary and Next Steps	907
6.22.1	Lab Progression Summary	907
6.22.2	Key Concepts Mastered	908
6.22.3	Your Driver Development Toolkit	908
6.22.4	Celebrate Your Achievement!	909
6.23	Wrapping Up	909
6.23.1	What You Now Know	909
6.23.2	What You Can Now Do	909
6.23.3	Mindset Shift	910
6.23.4	Final Thoughts	910
6.23.5	Before You Move On	910
6.23.6	Looking Ahead	910
6.24	Part 1 Checkpoint	911
6.25	Challenge Exercises (Optional)	912
6.25.1	Challenge 1: Trace a Lifecycle in dmesg	912
6.25.2	Challenge 2: Map the Entry Points	912
6.25.3	Challenge 3: Classification Drill	913
6.25.4	Challenge 4: Error Code Audit	914
6.25.5	Challenge 5: Dependency Detective	914
6.26	Summary Reference Table - Driver Building Blocks at a Glance	915
6.26.1	Quick Lookup by Task	917
6.26.2	Probe/Attach/Detach Quick Reference	918
7	Writing Your First Driver	919
7.1	Reader Guidance & Outcomes	919
7.1.1	What You'll Build	919
7.1.2	What You'll Not Build (Yet)	919
7.1.3	Estimated Time Investment	920
7.1.4	Prerequisites	920
7.1.5	Learning Outcomes	920
7.1.6	Success Criteria	921
7.1.7	Where This Chapter Fits	921
7.1.8	A Note on "Hello World" Versus "Hello Production"	921
7.1.9	How to Use This Chapter	921
7.1.10	A Word About Mistakes	922
7.1.11	Let's Begin	922
7.2	Project Scaffold (KLD skeleton)	922
7.2.1	Directory Layout	922
7.2.2	The Minimal Makefile	923
7.2.3	The Bare-Bones <code>myfirst.c</code>	924
7.2.4	Build and Test the Scaffold	926
7.2.5	What Just Happened?	927
7.2.6	Troubleshooting Common Scaffold Issues	928
7.2.7	Clean Build Practices	928
7.2.8	Version Control Checkpoint	933
7.2.9	What's Next?	933

7.3	Newbus: Just Enough to Attach	933
7.3.1	Why Newbus?	934
7.3.2	The Minimal Newbus Pattern	934
7.3.3	Step 1: Include Newbus Headers	934
7.3.4	Step 2: Define Your Softc	935
7.3.5	Step 3: Implement Probe	935
7.3.6	Step 4: Implement Attach	937
7.3.7	Step 5: Implement Detach	938
7.3.8	Step 6: Implement Identify	939
7.3.9	Step 7: Create the Method Table	940
7.3.10	Step 8: Declare the Driver	940
7.3.11	Step 9: Register with DRIVER_MODULE	941
7.3.12	Step 10: Remove the Old Module Loader	942
7.3.13	Step 11: Adjust the Makefile	942
7.3.14	Build and Test the Newbus Driver	942
7.3.15	What Changed?	943
7.3.16	Common Newbus Mistakes (and How to Avoid Them)	944
7.3.17	Newbus Lifecycle Timing Diagram	945
7.3.18	Quick Self-Check	946
7.4	softc and Lifecycle State	946
7.4.1	What Is the softc?	946
7.4.2	What Belongs in the softc?	947
7.4.3	Our myfirst Softc (Minimal Example)	948
7.4.4	Initializing the softc in attach()	948
7.4.5	Destroying the softc in detach()	950
7.4.6	Accessing the softc from Other Methods	951
7.4.7	Using Locks to Protect the softc	951
7.4.8	Softc Best Practices	952
7.4.9	Debugging softc Issues	953
7.4.10	Quick Self-Check	954
7.5	Logging Etiquette & dmesg Hygiene	954
7.5.1	Why Logging Matters	955
7.5.2	The Golden Rules of Driver Logging	955
7.5.3	Logging Patterns for Common Events	957
7.5.4	Rate-Limited Logging Macro (Advanced Preview)	958
7.5.5	What to Log During Development vs Production	959
7.5.6	Using Sysctls for Debug Logging	960
7.5.7	Inspecting Logs	960
7.5.8	Example: Logging in myfirst	961
7.5.9	Common Logging Mistakes	962
7.5.10	Quick Self-Check	963
7.6	A Temporary User Surface: /dev (Preview Only)	964
7.6.1	What Is a Character Device Switch (cdevsw)?	964
7.6.2	Defining the cdevsw for myfirst	964
7.6.3	Implementing open()	965
7.6.4	Implementing close()	966
7.6.5	Stubbing read() and write()	967
7.6.6	Creating the Device Node in attach()	968
7.6.7	Destroying the Device Node in detach()	970
7.6.8	Build, Test, and Verify	970
7.6.9	What Just Happened?	971

7.6.10	Common Device Node Mistakes	971
7.6.11	Quick Self-Check	973
7.7	A Tiny Control Plane: Read-Only sysctl	973
7.7.1	Why Sysctls Matter	974
7.7.2	The FreeBSD Sysctl Tree	974
7.7.3	Static vs Dynamic Sysctls	974
7.7.4	Adding a Sysctl Context to the softc	975
7.7.5	Creating the Sysctl Tree in attach()	975
7.7.6	Destroying the Sysctl Tree in detach()	977
7.7.7	Build, Load, and Test Sysctls	977
7.7.8	Making Sysctls More Useful	978
7.7.9	Read-Only vs Read-Write Sysctls	979
7.7.10	Sysctl Best Practices	980
7.7.11	Common Sysctl Mistakes	981
7.7.12	Quick Self-Check	982
7.8	Error Paths & Clean Unwind	982
7.8.1	Why Error Handling Matters	982
7.8.2	The Single-Label Unwind Pattern	983
7.8.3	Applying the Pattern to myfirst	984
7.8.4	Label Naming Convention	986
7.8.5	Testing Error Paths	987
7.8.6	Common Error Handling Mistakes	988
7.8.7	The Full Picture: Attach and Detach	989
7.8.8	Defensive Programming Checklist	990
7.8.9	Quick Self-Check	990
7.9	Look-in-Tree Anchors (Reference Only)	991
7.9.1	Why Look at Real Drivers?	991
7.9.2	Anchor 1: /usr/src/sys/dev/null/null.c	991
7.9.3	Anchor 2: /usr/src/sys/dev/led/led.c	992
7.9.4	Anchor 3: /usr/src/sys/net/if_tuntap.c	992
7.9.5	Anchor 4: /usr/src/sys/dev/uart/uart_bus_pci.c	993
7.9.6	Anchor 5: DRIVER_MODULE and MODULE_VERSION Patterns	993
7.9.7	How to Use These Anchors	994
7.9.8	Quick Tour: Common Patterns Across Drivers	994
7.9.9	What to Skip (For Now)	994
7.9.10	Self-Study Exercise	995
7.9.11	Quick Self-Check	995
7.10	Hands-On Labs	995
7.10.1	Lab Overview	996
7.10.2	Lab 7.1: Source Code Scavenger Hunt	996
7.10.3	Lab 7.2: Build, Load, and Verify Lifecycle	998
7.10.4	Lab 7.3: Test Device Node Open/Close	1001
7.10.5	Lab 7.4: Simulate Attach Failure and Verify Unwinding	1003
7.10.6	Labs Complete!	1006
7.11	Short Exercises	1006
7.11.1	Exercise 7.1: Add a Sysctl Flag	1007
7.11.2	Exercise 7.2: Log First and Last Open	1007
7.11.3	Exercise 7.3: Add a “Reset Stats” Sysctl	1008
7.11.4	Exercise 7.4: Test Load/Unload 100 Times	1009
7.11.5	Exercise 7.5: Compare Your Driver to null.c	1009

7.12	Optional Challenges	1010
7.12.1	Challenge 7.1: Implement a Simple Read Buffer	1010
7.12.2	Challenge 7.2: Allow Multiple Openers	1011
7.12.3	Challenge 7.3: Add a Write Counter	1012
7.12.4	Challenge 7.4: Create a Second Device (myfirst1)	1013
7.12.5	Challenge 7.5: Implement Rate-Limited Logging	1013
7.13	Pitfalls & Troubleshooting Decision Tree	1014
7.13.1	Symptom: Driver Won't Load	1014
7.13.2	Symptom: Device Node Doesn't Appear	1015
7.13.3	Symptom: Kernel Panic on Load	1015
7.13.4	Symptom: Kernel Panic on Unload	1016
7.13.5	Symptom: "Device Busy" on Unload	1016
7.13.6	Symptom: Sysctls Don't Appear	1017
7.13.7	Symptom: Open/Close Not Logging	1017
7.13.8	Symptom: Module Loads But Does Nothing	1017
7.13.9	General Debugging Tips	1018
7.14	Self-Assessment Rubric	1018
7.14.1	Core Knowledge	1019
7.14.2	Practical Skills	1019
7.14.3	Code Reading	1019
7.14.4	Conceptual Understanding	1020
7.14.5	Overall Assessment	1020
7.15	Wrapping Up & Pointers Forward	1021
7.15.1	What You Built	1021
7.15.2	What You Learned	1021
7.15.3	What's Still Missing (And Why)	1021
7.15.4	What's Next	1022
7.15.5	Your Next Steps	1022
7.15.6	A Final Word	1022
7.15.7	See You in Chapter 8	1023
8	Working with Device Files	1024
8.1	Reader Guidance & Outcomes	1024
8.1.1	Why This Chapter Earns Its Own Place	1024
8.1.2	Where Chapter 7 Left the Driver	1024
8.1.3	What You Will Learn	1025
8.1.4	What You Will Build	1025
8.1.5	What This Chapter Does Not Cover	1025
8.1.6	Estimated Time Investment	1026
8.1.7	Prerequisites	1026
8.1.8	How to Get the Most Out of This Chapter	1026
8.1.9	Roadmap Through the Chapter	1027
8.2	What a Device File Actually Is	1027
8.2.1	A Short History, in Pieces Worth Knowing	1027
8.2.2	Why the File Abstraction Earned Its Keep	1028
8.2.3	What a Device File Is Not	1029
8.2.4	The Variety You Will See Under /dev	1029
8.2.5	Character Devices Are the Common Case	1030
8.2.6	Reading <code>ls -l</code> for a Device Node	1030
8.3	devfs: Where /dev Comes From	1031
8.3.1	Why This Model Replaced <code>mknod</code>	1032
8.3.2	What devfs Is and Is Not	1032

8.3.3	devfs in the Family of Synthetic Filesystems	1032
8.3.4	Mount Options Worth Knowing	1033
8.3.5	devfs Inside Jails	1033
8.3.6	chroot and devfs	1034
8.3.7	/dev/fd and the Standard Symlinks	1034
8.3.8	Why the “Live Mirror” Property Matters to Drivers	1034
8.4	cdev, vnode, and File Descriptor	1035
8.4.1	Tracing a Single read(2) Through the Stack	1036
8.4.2	Why This Matters for Beginners	1036
8.4.3	A Closer Look at struct cdev	1036
8.4.4	struct cdevsw: The Dispatch Table	1038
8.4.5	The D_VERSION Stamp and Why It Exists	1040
8.4.6	Reference Counting at the cdev Level	1040
8.4.7	One More Turn on Lifetimes	1041
8.5	Permissions, Ownership, and Mode	1041
8.5.1	Where the Mode Comes From	1041
8.5.2	Named Groups You Will Meet	1042
8.5.3	A Worked Example	1042
8.5.4	Case Studies from the Tree	1043
8.5.5	Least Privilege Applied to Device Files	1044
8.5.6	Transitioning a Loose Mode to a Tight One	1044
8.5.7	What the uid and gid Constants Actually Are	1044
8.5.8	Three-Layer Policy: Driver, devfs.conf, devfs.rules	1045
8.6	Naming, Unit Numbers, and Subdirectories	1045
8.6.1	Names in the Real FreeBSD Tree	1046
8.6.2	Multiple Nodes Per Device	1046
8.6.3	The make_dev Family in Depth	1047
8.6.4	Which Form Should You Use?	1048
8.6.5	The MAKEDEV_* Flags	1049
8.6.6	The cdevsw d_flags	1049
8.6.7	Name Length and Name Characters	1050
8.6.8	Unit Numbers: Where They Come From	1050
8.6.9	Naming Conventions in the Tree	1051
8.6.10	Handling Multiple Units	1051
8.6.11	A Small Detour: devtoname and Friends	1052
8.7	Aliases: One cdev, More Than One Name	1052
8.7.1	Aliases vs link in devfs.conf	1053
8.7.2	Comparison Table: Three Ways to Give a Node Two Names	1053
8.7.3	The make_dev_alias_p Variant	1054
8.7.4	The make_dev_physpath_alias Variant	1054
8.7.5	Reading Uses of make_dev_alias in the Tree	1054
8.7.6	Alias Lifetimes and Destruction Order	1055
8.7.7	When Aliases Start to Smell	1055
8.8	Per-Open State with devfs_set_cdevpriv	1055
8.8.1	A Minimal Per-Open Counter for myfirst	1056
8.8.2	What the Destructor Guarantees	1058
8.8.3	When devfs_set_cdevpriv Returns EBUSY	1059
8.8.4	When Not to Use devfs_set_cdevpriv	1059
8.8.5	Inside devfs_set_cdevpriv: How the Machinery Works	1059
8.8.6	Interaction with fork, dup, and SCM_RIGHTS	1061
8.8.7	A Tour of Real Uses in the Tree	1061

8.8.8	Patterns for What to Put in the Per-Open Structure	1062
8.8.9	A Warning Against Cross-Referencing From the <code>softc</code> to Per-Open Records	1063
8.9	Destroying <code>cdevs</code> Safely	1063
8.9.1	The Draining Model	1063
8.9.2	The Four Destruction Functions	1064
8.9.3	The Race That <code>destroy_dev_drain</code> Exists to Prevent	1064
8.9.4	The Order of Operations in <code>detach</code>	1065
8.9.5	Unload Under Load	1065
8.9.6	When <code>destroy_dev</code> Refuses to Proceed Quickly	1066
8.9.7	A Minimal <code>d_purge</code> Example	1066
8.9.8	Summary	1067
8.10	Persistent Policy: <code>devfs.conf</code> and <code>devfs.rules</code>	1067
8.10.1	<code>devfs.conf</code> : one-time, per-path adjustments	1067
8.10.2	<code>devfs.rules</code> : rule-based, used by jails	1068
8.10.3	The Full <code>devfs.conf</code> Syntax	1068
8.10.4	Reading the Example <code>devfs.conf</code>	1069
8.10.5	Worked Examples That Recur in Practice	1069
8.10.6	<code>devfs.rules</code> in Depth	1070
8.10.7	Reading the Default Rulesets	1070
8.10.8	A Complete Jail Example End to End	1072
8.10.9	Debugging Rule Mismatches	1073
8.10.10	Runtime Manipulation With <code>devfs(8)</code>	1074
8.10.11	A Caveat About <code>devfs.conf</code> Timing	1074
8.11	Exercising Your Device From Userland	1074
8.11.1	A Second Probe: Inspecting With <code>stat(2)</code>	1076
8.11.2	A Third Probe: Parallel Opens	1077
8.11.3	A Fourth Probe: Stress Testing	1078
8.11.4	Observing <code>devd</code> Events	1079
8.11.5	Handling <code>errno</code> From Device Operations	1080
8.11.6	Driving Devices From Shell Scripts	1080
8.12	Reading Real FreeBSD Drivers Through a Device-File Lens	1081
8.12.1	Walkthrough 1: <code>/usr/src/sys/dev/null/null.c</code>	1081
8.12.2	Walkthrough 2: <code>/usr/src/sys/dev/led/led.c</code>	1083
8.12.3	Walkthrough 3: <code>/usr/src/sys/dev/md/md.c</code>	1084
8.12.4	A Brief Word About Cloning	1085
8.12.5	Walkthrough 4: <code>/usr/src/sys/net/bpf.c</code> (Device-File Surface Only)	1085
8.12.6	A Synthesis Across All Four Drivers	1087
8.12.7	What Four Drivers Taught Us	1087
8.13	Common Scenarios and Recipes	1088
8.13.1	Recipe 1: A Driver That Must Never Be Opened Twice	1088
8.13.2	Recipe 2: A Driver With Per-User Read Offsets	1088
8.13.3	Recipe 3: A Device With a Privileged Control Node	1089
8.13.4	Recipe 4: A Node That Appears Only When a Condition Is Met	1089
8.13.5	Recipe 5: A Node Whose Ownership Should Change Based on a Runtime Knob	1089
8.13.6	Recipe 6: A Node That Should Be Visible Only Inside a Jail	1090
8.13.7	Recipe 7: A Node That Can Survive Unexpected Process Deaths	1090
8.13.8	Recipe 8: A Device That Must Support Polling	1090
8.13.9	Recipe 9: A Device That Needs to Be Mapped Into User Memory	1091
8.13.10	Recipe 10: A Device That Exposes a Log	1091
8.13.11	When a Recipe Does Not Fit	1091

8.14	Practical Workflows for the Device-File Surface	1092
8.14.1	The Inner Loop	1092
8.14.2	Reading dmesg Well	1092
8.14.3	Watching fstat	1093
8.14.4	The sysctl Partner of Every Driver	1093
8.14.5	A Quick Checklist for Every Code Change	1094
8.14.6	A Quick Checklist Before a Release	1094
8.14.7	A Workflow for Adding a New Node	1094
8.14.8	A Workflow for Diagnosing a Missing Node	1095
8.14.9	A Workflow for Reviewing Someone Else's Driver	1096
8.14.10	Keeping a Lab Logbook	1096
8.14.11	Common Design Questions and How to Think About Them	1097
8.14.12	A Decision Tree for Common Design Choices	1098
8.14.13	A Warning About Over-Engineering	1099
8.15	Hands-On Labs	1100
8.15.1	Lab 8.1: Structured Name and Tighter Permissions	1100
8.15.2	Lab 8.2: Add an Alias	1101
8.15.3	Lab 8.3: Per-Open State	1101
8.15.4	Lab 8.4: devfs.conf Persistence	1102
8.15.5	Lab 8.5: Two-Node Driver (Data and Control)	1102
8.15.6	Lab 8.6: Parallel Probe Verification	1103
8.15.7	Lab 8.7: devfs.rules for a Jail	1104
8.15.8	Lab 8.8: Destroy-Dev Drain	1105
8.16	Challenge Exercises	1106
8.16.1	Challenge 1: Use the Alias	1106
8.16.2	Challenge 2: Observe Per-Open Cleanup	1106
8.16.3	Challenge 3: Experiment with devfs.rules	1107
8.16.4	Challenge 4: Read Two More Drivers	1107
8.16.5	Challenge 5: devd Configuration	1107
8.16.6	Challenge 6: Add a Status Node	1107
8.17	Error Codes for Device File Operations	1108
8.17.1	The Short List	1108
8.17.2	Which Value for Which Situation?	1109
8.17.3	Returning Errors From d_close	1110
8.17.4	Mapping Your errno to User Messages	1110
8.17.5	A Quick Checklist Before You Pick an errno	1111
8.17.6	A Short Narrative: errno Chosen Three Times	1111
8.17.7	Using err(3) and warn(3) to Exercise errno Values	1112
8.17.8	errno Values Your Driver Should Almost Never Return	1112
8.18	Tools for Inspecting /dev	1113
8.18.1	ls -l for Permissions and Existence	1113
8.18.2	stat and stat(1)	1113
8.18.3	fstat(1): Who Has It Open?	1113
8.18.4	procstat -f: A Process-First View	1114
8.18.5	devinfo(8): The Newbus Side	1114
8.18.6	devfs(8): Rulesets and Rules	1115
8.18.7	sysctl dev.* and Other Hierarchies	1115
8.18.8	kldstat: Is the Module Loaded?	1115
8.18.9	dmesg: The Log of What Happened	1115
8.18.10	Troubleshooting Walkthrough 1: The Node Is Missing	1116
8.18.11	Troubleshooting Walkthrough 2: kldunload Returns EBUSY	1116

8.18.12	A Note on Habits	1116
8.19	Pitfalls and Things Worth Watching	1116
8.19.1	Pitfalls That Are Really About Lifecycle	1118
8.19.2	Pitfalls in Permission and Policy	1118
8.19.3	Pitfalls in Observability	1119
8.20	A Final Study Plan	1119
8.21	Companion Tree Quick Reference	1120
8.21.1	Driver Stages	1120
8.21.2	Userland Probes	1120
8.21.3	Configuration Samples	1120
8.21.4	How the Stages Differ	1120
8.21.5	On Re-using This Tree Later	1121
8.22	A Closing Reflection on Interfaces	1121
8.23	Wrapping Up	1122
8.23.1	A Short Self-Check	1122
8.23.2	Recap Organised by Topic	1123
8.23.3	Looking Ahead to Chapter 9	1124
8.24	Reference: <code>make_dev_s</code> and <code>cdevsw</code> at a Glance	1124
8.24.1	Canonical <code>make_dev_s</code> Skeleton	1125
8.24.2	Canonical <code>cdevsw</code> Skeleton	1125
8.24.3	The <code>make_dev_args</code> Structure	1125
8.24.4	The <code>MAKEDEV</code> Flag Word	1126
8.24.5	The <code>cdevsw d_flags</code> Field	1126
8.24.6	Common UID and GID Constants	1127
8.24.7	Destruction Functions	1127
8.24.8	Per-Open State Functions	1127
8.24.9	Alias Functions	1128
8.24.10	Reference-Count Helpers	1128
8.24.11	Where to Read More	1128
8.24.12	A Condensed Pattern Catalogue	1128
8.24.13	Common <code>errno</code> Values by Operation	1129
8.24.14	A Short Glossary of Terms Used in the Chapter	1129
8.25	Consolidation and Review	1130
8.25.1	The Three Ideas That Matter Most	1130
8.25.2	The Shape of the Driver You End the Chapter With	1130
8.25.3	What to Practice Before Starting Chapter 9	1131
9	Reading and Writing to Devices	1132
9.1	Reader Guidance & Outcomes	1132
9.1.1	Why This Chapter Earns Its Own Place	1132
9.1.2	Where Chapter 8 Left the Driver	1132
9.1.3	What You Will Learn	1133
9.1.4	What You Will Build	1133
9.1.5	What This Chapter Does Not Cover	1134
9.1.6	Estimated Time Investment	1134
9.1.7	Prerequisites	1134
9.1.8	How to Get the Most Out of This Chapter	1135
9.1.9	Roadmap Through the Chapter	1135
9.2	A Visual Map of the I/O Path	1136
9.3	Devices in UNIX: A Quick Refresher	1139
9.3.1	What Makes a Device Different from a File?	1139
9.3.2	How UNIX Treats Devices as Streams of Data	1139

9.3.3	Sequential vs. Random Access	1140
9.3.4	The Role of <code>read()</code> and <code>write()</code> in Device Drivers	1140
9.3.5	Character Devices vs. Block Devices Revisited	1141
9.3.6	Exercise: Classifying Real Devices on Your FreeBSD System	1141
9.4	The Anatomy of <code>d_read()</code>	1141
9.4.1	What <code>d_read</code> Is Asked to Do	1142
9.4.2	What <code>d_read</code> Is Not Asked to Do	1143
9.4.3	A First Real <code>d_read</code>	1143
9.4.4	A Variant: <code>uiomove_frombuf</code>	1144
9.4.5	The Signature in Use: <code>myfirst_read</code> Stage 1	1145
9.4.6	Reading <code>d_read</code> in the Tree	1146
9.5	The Anatomy of <code>d_write()</code>	1147
9.5.1	What <code>d_write</code> Is Asked to Do	1147
9.5.2	What <code>d_write</code> Is Not Asked to Do	1147
9.5.3	A First Real <code>d_write</code>	1147
9.5.4	A Write That Actually Stores Data	1148
9.5.5	Symmetry With <code>d_read</code>	1149
9.5.6	Reading <code>d_write</code> in the Tree	1150
9.5.7	Reading an Unfamiliar <code>d_read</code> or <code>d_write</code> in the Wild	1150
9.5.8	What If You Leave <code>d_read</code> or <code>d_write</code> Unset?	1151
9.5.9	A Second Real Driver: How <code>mem(4)</code> Uses One Handler for Both Directions	1152
9.5.10	Why This Walkthrough Was Worth the Detour	1153
9.6	Understanding the <code>ioflag</code> Argument	1154
9.6.1	Where <code>ioflag</code> Comes From	1154
9.6.2	The Flag Bits That Matter	1154
9.6.3	A Handler That Checks <code>ioflag</code>	1155
9.6.4	What the Chapter 9 Stages Do With <code>ioflag</code>	1156
9.6.5	A Small Debugging Aid	1156
9.6.6	<code>ioflag</code> in the Tree	1156
9.7	Understanding <code>struct uio</code> Deeply	1156
9.7.1	The Declaration	1156
9.7.2	<code>uio_iov</code> and <code>uio_iovcnt</code> : The Scatter-Gather List	1157
9.7.3	<code>uio_offset</code> : The Offset Into the Target	1157
9.7.4	<code>uio_resid</code> : The Remaining Bytes	1158
9.7.5	<code>uio_segflg</code> : Where the Buffer Lives	1158
9.7.6	<code>uio_rw</code> : The Direction	1158
9.7.7	<code>uio_td</code> : The Owning Thread	1159
9.7.8	An Illustration: What Happens to <code>uio</code> During a <code>read(fd, buf, 1024)</code> Call	1159
9.7.9	How <code>readv(2)</code> Differs	1160
9.7.10	Re-entering a Handler With a Partly Consumed <code>uio</code>	1160
9.7.11	What Your Driver Needs to Read and What It Should Leave Alone	1160
9.7.12	The <code>uio</code> Fields in Practice	1161
9.7.13	The Life of a Single <code>uio</code> : Three Snapshots	1161
9.7.14	What This Mental Model Buys You	1162
9.8	Safe Data Transfer: <code>uiomove</code> , <code>copyin</code> , <code>copyout</code>	1163
9.8.1	Why Direct Memory Access Is Unsafe	1163
9.8.2	What <code>copyin(9)</code> and <code>copyout(9)</code> Do	1163
9.8.3	What <code>uiomove(9)</code> Does	1164
9.8.4	When to Use Which	1164
9.8.5	A Table for Quick Reference	1165
9.8.6	Why Not a Straight <code>memcpy</code> ?	1165

9.8.7	What Happens Under a Fault	1165
9.8.8	Alignment and Type Safety	1166
9.8.9	A Small Worked Example	1166
9.8.10	A Mini Case Study: When a Struct Round-Trip Goes Wrong	1167
9.8.11	Summary of This Section	1168
9.9	Managing Internal Buffers in Your Driver	1168
9.9.1	Why You Need Buffers	1168
9.9.2	Static vs Dynamic Allocation	1169
9.9.3	The <code>malloc(9)</code> Call	1169
9.9.4	The Matching <code>free(9)</code> Call	1170
9.9.5	Buffer Sizing	1170
9.9.6	Buffer Overflows	1171
9.9.7	Locking Considerations	1171
9.9.8	A Preview of Circular Buffers	1171
9.9.9	A Note on Thread Safety of the Per-Descriptor <code>fh</code>	1172
9.9.10	Kernel Helpers You Should Recognise	1172
9.10	Error Handling and Edge Cases	1173
9.10.1	The <code>Erno</code> Values That Matter for I/O	1173
9.10.2	End of File in a Read	1174
9.10.3	Zero-Length Reads and Writes	1174
9.10.4	Short Transfers	1174
9.10.5	Handling <code>EFAULT</code> From <code>uiomove</code>	1174
9.10.6	Defensive Programming for User Input	1175
9.10.7	Logging Errors vs Silent Failures	1175
9.10.8	Predictability and User-Friendliness	1176
9.10.9	A Short Table of Conventions	1176
9.11	Evolving Your Driver: The Three Stages	1177
9.11.1	Stage 1: A Static-Message Reader	1177
9.11.2	Stage 2: A Write-Once / Read-Many Buffer	1181
9.11.3	Stage 3: A First-In / First-Out Echo Driver	1184
9.12	Tracing a <code>read(2)</code> From User Space to Your Handler	1187
9.12.1	Step 1: The User Program Calls <code>read(2)</code>	1187
9.12.2	Step 2: The Kernel Looks Up the File Descriptor	1188
9.12.3	Step 3: The Generic File-Operation Vector Dispatches	1188
9.12.4	Step 4: VFS Calls Into <code>devfs</code>	1188
9.12.5	Step 5: Your <code>d_read</code> Handler Runs	1188
9.12.6	Step 6: The Kernel Unwinds and Reports	1189
9.12.7	Step 7: The Reference Counts Unwind	1189
9.12.8	Why This Trace Matters to You	1189
9.12.9	The Mirror: Tracing a <code>write(2)</code>	1189
9.13	Practical Workflow: Testing Your Driver From the Shell	1190
9.13.1	<code>cat(1)</code> : the first check	1190
9.13.2	<code>echo(1)</code> and <code>printf(1)</code> : simple writes	1191
9.13.3	<code>dd(1)</code> : the precision tool	1191
9.13.4	<code>od(1)</code> and <code>hexdump(1)</code> : byte-level inspection	1192
9.13.5	<code>sysctl(8)</code> and <code>dmesg(8)</code> : the kernel's voice	1192
9.13.6	<code>fstat(1)</code> : who has the descriptor open?	1192
9.13.7	<code>truss(1)</code> and <code>ktrace(1)</code> : watching the syscalls	1193
9.13.8	A Suggested Test Rhythm	1193
9.13.9	A Concrete <code>truss</code> Walkthrough	1193
9.13.10	A Quick Note on <code>ktrace</code>	1194

9.13.11	Watching Kernel Memory With <code>vmstat -m</code>	1195
9.14	Observability: Making Your Driver Legible	1195
9.14.1	Three Surfaces: <code>sysctl</code> , <code>dmesg</code> , <code>userland</code>	1196
9.14.2	<code>Sysctl</code> : Counters vs State	1196
9.14.3	<code>dmesg</code> : Events Worth Seeing	1196
9.14.4	The Trap of Over-Logging	1197
9.14.5	Counters You Will Add Later	1197
9.14.6	What Your Driver Looks Like Under Light Load	1197
9.15	Signed, Unsigned, and the Perils of Off-by-One	1198
9.15.1	<code>ssize_t</code> vs <code>size_t</code>	1198
9.15.2	<code>uio_resid</code> Can Be Compared Against Unsigned	1199
9.15.3	Off-by-One in Counters	1199
9.15.4	The <code>uio_offset</code> - before Idiom	1200
9.16	Additional Troubleshooting: The Edge Cases	1200
9.16.1	“The second read on the same descriptor returns zero”	1200
9.16.2	“The driver returns zero bytes immediately when the buffer has data”	1200
9.16.3	“My <code>uiomove_frombuf</code> always returns zero regardless of the buffer”	1201
9.16.4	“The buffer overflows into adjacent memory”	1201
9.16.5	“A process reading from the device hangs forever”	1201
9.16.6	“ <code>kldunload</code> says <code>kldunload: can't unload file: Device busy</code> ”	1201
9.16.7	“I modified the driver and make compiles but <code>kldload</code> fails with version mismatch”	1201
9.16.8	“I see every byte written through the device printed twice in <code>dmesg</code> ”	1202
9.16.9	“My <code>d_read</code> is called but my <code>d_write</code> is not”	1202
9.17	Design Notes: Why Each Stage Stops Where It Does	1202
9.17.1	Why Stage 1 Exists	1202
9.17.2	Why Stage 2 Exists	1203
9.17.3	Why Stage 3 Exists	1203
9.17.4	Why No Ring Buffer Here	1203
9.17.5	Why No Blocking	1203
9.17.6	What the Stages Are Not Trying to Be	1204
9.18	Hands-On Labs	1204
9.18.1	Lab 9.1: Build and Load Stage 1	1204
9.18.2	Lab 9.2: Exercise Stage 2 with Writes and Reads	1205
9.18.3	Lab 9.3: Stage 3 FIFO Behaviour	1207
9.18.4	Lab 9.4: Using <code>dd</code> to Measure Transfer Behaviour	1208
9.18.5	Lab 9.5: A Small Round-Trip C Program	1209
9.18.6	Lab 9.6: Inspecting Binary Round-Trips	1211
9.18.7	Lab 9.7: Observing a Running Driver End-to-End	1212
9.19	Challenge Exercises	1213
9.19.1	Challenge 9.1: Per-Descriptor Read Counters	1213
9.19.2	Challenge 9.2: A <code>readv(2)</code> -Aware Test	1214
9.19.3	Challenge 9.3: Short-Write Demonstration	1214
9.19.4	Challenge 9.4: An <code>ls -l</code> Sensor	1214
9.19.5	Challenge 9.5: A Reverse-Echo Driver	1214
9.19.6	Challenge 9.6: Binary Round Trip	1215
9.19.7	Challenge 9.7: A Hex-View Test Harness	1215
9.19.8	Challenge 9.8: Who Has the Descriptor Open?	1215
9.20	Troubleshooting Common Mistakes	1216
9.20.1	“My driver returns zero bytes even though I wrote data”	1216
9.20.2	“My reads return garbage”	1216
9.20.3	“The kernel panics with a pagefault on a user address”	1216

9.20.4	“The driver refuses to unload”	1216
9.20.5	“My write handler returns EFAULT”	1217
9.20.6	“My counters are wrong under concurrent writes”	1217
9.20.7	“sysctl counters do not reflect the real state”	1217
9.20.8	“The first read after a fresh load returns zero bytes”	1217
9.20.9	“The build fails with unknown reference to uiomove_frombuf”	1218
9.20.10	“My handler is called twice for one read(2)”	1218
9.21	Contrast Patterns: Correct vs. Buggy Handlers	1218
9.21.1	Contrast 1: Returning a Byte Count	1218
9.21.2	Contrast 2: Handling a Zero-Length Request	1219
9.21.3	Contrast 3: Buffer Capacity Arithmetic	1220
9.21.4	Contrast 4: Holding a Spin Lock Across uiomove	1220
9.21.5	Contrast 5: Resetting Shared State in d_open	1221
9.21.6	Contrast 6: Accounting Before Knowing the Outcome	1222
9.21.7	Contrast 7: Dereferencing a User Pointer Directly	1222
9.21.8	Contrast 8: Leaking a Per-Open Structure on d_open Failure	1223
9.21.9	How to Use This Contrast Table	1224
9.22	Self-Assessment Before Chapter 10	1224
9.23	Wrapping Up	1225
9.23.1	The Three Ideas That Matter Most	1226
9.23.2	The Shape of the Driver You End the Chapter With	1226
9.23.3	What You Should Practice Before Starting Chapter 10	1226
9.23.4	Looking Ahead to Chapter 10	1227
9.24	Reference: The Signatures and Helpers Used in This Chapter	1227
9.24.1	d_read and d_write Signatures	1227
9.24.2	The Canonical struct uio	1228
9.24.3	The uio_seg and uio_rw Enumerations	1228
9.24.4	uiomove Family	1228
9.24.5	copyin and copyout	1228
9.24.6	ioflag Bits That Matter for Character Devices	1229
9.24.7	Memory Allocation	1229
9.24.8	Per-Open State (Chapter 8 carryover, used here)	1229
9.24.9	Errno Values Used in This Chapter	1230
9.24.10	Helpful device_printf(9) Patterns	1230
9.24.11	The Three Stages at a Glance	1230
9.24.12	Consolidated File List for the Chapter	1231
9.25	Appendix A: A Closer Look at uiomove’s Internal Loop	1231
9.25.1	The Setup	1231
9.25.2	The Main Loop	1231
9.25.3	What You Should Take Away	1233
9.26	Appendix B: Why read(fd, buf, 0) Is Allowed	1233
9.27	Appendix C: A Short Tour of /dev/zero’s Read Path	1233
9.27.1	From User Space to Kernel Dispatch	1234
9.27.2	Inside zero_read	1234
9.27.3	Inside uiomove	1234
9.27.4	Back Up the Stack	1235
9.27.5	What This Tells You About Your Own Driver	1235
9.28	Appendix D: Common read(2) / write(2) Return Values at the User Side	1235
9.28.1	read(2)	1235
9.28.2	write(2)	1236
9.28.3	What This Means for Your Driver	1236

9.29	Appendix E: The One-Page Cheat Sheet	1236
9.30	Chapter Summary	1238
10	Handling Input and Output Efficiently	1239
10.1	Reader Guidance & Outcomes	1239
10.1.1	Why This Chapter Earns Its Own Place	1240
10.1.2	Where Chapter 9 Left the Driver	1240
10.1.3	What You Will Learn	1240
10.1.4	What You Will Build	1241
10.1.5	What This Chapter Does Not Cover	1242
10.1.6	Estimated Time Investment	1242
10.1.7	Prerequisites	1242
10.1.8	How to Get the Most Out of This Chapter	1243
10.1.9	Roadmap Through the Chapter	1243
10.2	Section 1: What Buffered I/O Is and Why It Matters	1244
10.2.1	A Plain-Language Definition	1244
10.2.2	Two Concrete Pictures	1244
10.2.3	Buckets vs. Pipelines	1245
10.2.4	Performance: System Calls and Context Switches	1245
10.2.5	Buffered Data Transfers in Device Drivers	1246
10.2.6	When to Use Buffered I/O in a Driver	1246
10.2.7	A First Look at the Cost of an Unbuffered Driver	1247
10.2.8	Where We Are Going	1247
10.2.9	Wrapping Up Section 1	1248
10.3	Section 2: Creating a Circular Buffer	1248
10.3.1	What a Circular Buffer Is	1248
10.3.2	Managing Read and Write Pointers	1249
10.3.3	Detecting Buffer Full and Buffer Empty	1250
10.3.4	Allocating a Fixed-Size Circular Buffer	1250
10.3.5	A Walkthrough of the Wrap-Around	1251
10.3.6	Avoiding Overwrites and Data Loss	1252
10.3.7	Implementing It in Userland	1252
10.3.8	A Tiny Test Program	1256
10.3.9	Why Build It in Userland First	1258
10.3.10	Things That Can Still Go Wrong	1259
10.3.11	Wrapping Up Section 2	1259
10.4	Section 3: Integrating a Circular Buffer into Your Driver	1260
10.4.1	Moving cbuf Into the Kernel	1260
10.4.2	What Changes in myfirst.c	1263
10.4.3	The Stage 2 Driver: Handlers Refactored	1264
10.4.4	Updating attach and detach	1268
10.4.5	Logging Buffer State for Debugging	1271
10.4.6	Why the cbuf Has Its Own Memory Tag	1272
10.4.7	A Quick Trace of an Aligned and a Wrapped Transfer	1272
10.4.8	What the User Sees	1273
10.4.9	Handling Detach with Live Data	1273
10.4.10	Wrapping Up Section 3	1274
10.5	Section 4: Improving the Driver's Behaviour with Partial Reads and Writes	1274
10.5.1	What "Partial" Means in UNIX	1274
10.5.2	Why Drivers Should Embrace Partial Transfers	1275
10.5.3	Reporting Accurate Byte Counts	1276
10.5.4	End-of-Data: When Should d_read Return Zero?	1276

10.5.5	Backpressure on the Write Side	1277
10.5.6	Zero-Length Reads and Writes	1277
10.5.7	A User-Space Walk Through the Loop	1277
10.5.8	What cat, dd, and Friends Actually Do	1279
10.5.9	Common Mistakes in Partial-Transfer Code	1279
10.5.10	A Concrete Example: Watching Partial Reads	1280
10.5.11	Wrapping Up Section 4	1282
10.6	Section 5: Implementing Non-Blocking I/O	1282
10.6.1	What Non-Blocking I/O Is	1282
10.6.2	When Non-Blocking Behaviour Is Useful	1283
10.6.3	The <code>IO_NDELAY</code> Flag: How It Flows to the Driver	1283
10.6.4	The <code>EAGAIN</code> Convention	1283
10.6.5	The Blocking Path: <code>mtx_sleep(9)</code> and <code>wakeup(9)</code>	1284
10.6.6	The Writer Side	1285
10.6.7	The Full Stage 3 Handlers	1286
10.6.8	Waking Sleepers on Detach	1289
10.6.9	Adding <code>d_poll</code> for <code>select(2)</code> and <code>poll(2)</code>	1290
10.6.10	How a Non-Blocking Caller Uses All of This	1292
10.6.11	A Note About <code>O_NONBLOCK</code> and <code>select/poll</code> Composition	1292
10.6.12	Observing the Blocking Path in Action	1293
10.6.13	Common Mistakes in the Blocking Path	1293
10.6.14	Wrapping Up Section 5	1294
10.7	Section 6: Testing Buffered I/O with User Programs	1294
10.7.1	Three Tools We Will Build	1294
10.7.2	Updating <code>rw_myfirst</code> for Larger Inputs	1295
10.7.3	Why a Round-Trip Test Matters	1297
10.7.4	Building <code>rw_myfirst_nb</code>	1298
10.7.5	Building <code>producer_consumer.c</code>	1300
10.7.6	Using <code>dd(1)</code> for Volume Tests	1303
10.7.7	Using <code>hexdump(1)</code> to Verify Content	1304
10.7.8	Using <code>truss(1)</code> to See Syscall Traffic	1304
10.7.9	Using <code>sysctl(8)</code> to Watch State Live	1305
10.7.10	Using <code>vmstat -m</code> to Watch Memory	1305
10.7.11	Building the Test Kit	1305
10.7.12	A Consolidated Test Plan	1306
10.7.13	Debugging When a Test Fails	1307
10.7.14	Wrapping Up Section 6	1307
10.8	Section 7: Refactoring and Preparing for Concurrency	1308
10.8.1	Identifying Potential Race Conditions	1308
10.8.2	The Refactor: Pulling Buffer Access into Helpers	1309
10.8.3	Documenting the Locking Strategy	1313
10.8.4	Splitting <code>cbuf</code> Out of <code>myfirst.c</code>	1313
10.8.5	Naming Conventions	1314
10.8.6	Defending Against Buffer-Size Surprises	1314
10.8.7	Goals for the Next Milestone	1315
10.8.8	A Reading Order for the Next Chapter	1315
10.8.9	Wrapping Up Section 7	1315
10.9	Section 8: Three Supplementary Topics	1316
10.9.1	Topic 1: <code>d_mmap(9)</code> and Mapping a Kernel Buffer	1316
10.9.2	Topic 2: Zero-Copy Considerations	1320
10.9.3	Topic 3: Readahead and Write Coalescing	1321

10.9.4 Wrapping Up Section 8	1322
10.10 Hands-On Labs	1322
10.10.1 Lab 1: The Standalone Circular Buffer	1323
10.10.2 Lab 2: The Stage 2 Driver (Circular Buffer Integration)	1323
10.10.3 Lab 3: The Stage 3 Driver (Blocking and Non-Blocking)	1324
10.10.4 Lab 4: The Stage 4 Driver (Poll Support and Refactor)	1325
10.10.5 Lab 5: Memory Mapping	1326
10.10.6 Lab 6: Stress and Long-Running Tests	1327
10.10.7 Lab 7: Deliberate Failures	1327
10.10.8 Lab 8: Reading Real FreeBSD Drivers	1328
10.11 Challenge Exercises	1328
10.11.1 Challenge 1: Add a Tunable Buffer Size	1329
10.11.2 Challenge 2: Implement Overwrite Semantics as an Optional Mode	1329
10.11.3 Challenge 3: Per-Reader Position	1329
10.11.4 Challenge 4: Implement <code>d_kqfilter</code>	1330
10.11.5 Challenge 5: Per-CPU Counters	1330
10.11.6 Challenge 6: A Hardware-Style Interrupt Simulator	1330
10.11.7 Challenge 7: A Logging Buffer with <code>dmesg</code> -Style Behaviour	1331
10.11.8 Challenge 8: Performance Measurement	1331
10.11.9 Challenge 9: Cross-Reading Real Drivers	1331
10.11.10 Challenge 10: Document Your Driver	1332
10.12 Troubleshooting and Common Mistakes	1332
10.12.1 Symptom: <code>cat /dev/myfirst</code> blocks forever, even after <code>echo</code> writes data	1332
10.12.2 Symptom: Data corruption under heavy load	1332
10.12.3 Symptom: Kernel panic with “sleeping with mutex held”	1333
10.12.4 Symptom: <code>EAGAIN</code> is returned even when data is available	1333
10.12.5 Symptom: A write succeeds but a subsequent read gets fewer bytes	1333
10.12.6 Symptom: <code>kldunload</code> returns <code>EBUSY</code>	1333
10.12.7 Symptom: Memory growth in <code>vmstat -m grep cbuf</code>	1334
10.12.8 Symptom: <code>select(2)</code> or <code>poll(2)</code> does not wake up	1334
10.12.9 Symptom: <code>truss</code> shows <code>EINVAL</code> for zero-byte reads	1334
10.12.10 Symptom: <code>ps</code> shows the reader stuck in a sleep state called something different	1334
10.12.11 Symptom: A signal does not interrupt a blocked read	1335
10.12.12 Symptom: Compiler warns about a missing prototype for <code>cbuf_read</code>	1335
10.12.13 Symptom: <code>make</code> complains about missing <code>bus_if.h</code> or <code>device_if.h</code>	1335
10.12.14 Symptom: <code>kldload</code> fails with <code>Exec format error</code>	1335
10.12.15 Symptom: The driver reports correct data but loses it after several runs	1335
10.12.16 Symptom: <code>producer_consumer</code> reports a small number of mismatches under heavy load	1336
10.12.17 General Advice for Debugging	1336
10.13 Quick Reference: Patterns and Primitives	1336
10.13.1 The Circular Buffer API	1336
10.13.2 The Driver-Level Helpers	1337
10.13.3 The Read Handler Spine	1337
10.13.4 The Write Handler Spine	1338
10.13.5 The Sleep Pattern	1339
10.13.6 The Wake Pattern	1339
10.13.7 The <code>d_poll</code> Handler	1339
10.13.8 The <code>d_mmap</code> Handler	1340
10.13.9 The <code>cdevsw</code>	1340
10.13.10 Errno Values for I/O	1341
10.13.11 <code>lioflg</code> Bits	1341

10.13.12	poll(2) Events	1341
10.13.13	Memory Allocator Reference	1342
10.13.14	Sleep / Wake Reference	1342
10.13.15	Lock Reference	1342
10.13.16	Test Tools Reference	1343
10.13.17	Driver Lifecycle Summary	1343
10.13.18	File Layout Summary	1344
10.13.19	A Mental Model in One Paragraph	1345
10.14	Appendix: A Source-Reading Walkthrough of <code>evdev/cdev.c</code>	1345
10.14.1	What <code>evdev</code> Is	1345
10.14.2	The Per-Client State	1346
10.14.3	The Read Handler	1346
10.14.4	The Wakeup	1348
10.14.5	The Poll	1349
10.14.6	The <code>kqfilter</code>	1350
10.14.7	What the Walkthrough Confirms	1351
10.14.8	A Short Reading Plan	1351
10.15	Chapter Summary	1352
10.16	Wrapping Up	1352
10.17	Part 2 Checkpoint	1353
10.18	Looking Ahead: Bridge to Chapter 11	1354
11	Concurrency in Drivers	1356
11.1	Reader Guidance & Outcomes	1356
11.1.1	Why This Chapter Earns Its Own Place	1356
11.1.2	Where Chapter 10 Left the Driver	1357
11.1.3	What You Will Learn	1357
11.1.4	What This Chapter Does Not Cover	1358
11.1.5	Estimated Time Investment	1359
11.1.6	Prerequisites	1359
11.1.7	How to Get the Most Out of This Chapter	1359
11.1.8	Roadmap Through the Chapter	1360
11.2	Section 1: Why Concurrency Matters	1360
11.2.1	What Concurrency Means in the Kernel	1361
11.2.2	Where the Streams Come From: a Concrete Picture	1361
11.2.3	Concurrency vs Parallelism	1362
11.2.4	What Can Go Wrong: The Four Failure Modes	1363
11.2.5	A Thought Experiment	1364
11.2.6	A Preview of the Driver Under Load	1365
11.2.7	The FreeBSD Scheduler in One Page	1366
11.2.8	Kinds of Concurrency by Pedagogy Level	1366
11.2.9	Some Concurrency Is Invisible	1367
11.2.10	Wrapping Up Section 1	1367
11.3	Section 2: Race Conditions and Data Corruption	1368
11.3.1	A Precise Definition	1368
11.3.2	The Anatomy of a Race	1368
11.3.3	How Races Appear in the Circular Buffer	1369
11.3.4	A Deliberately Unsafe Driver	1370
11.3.5	Running the Race	1372
11.3.6	Observing the Damage with Logs	1373
11.3.7	Cleaning Up	1373
11.3.8	The Lesson	1374

11.3.9	A Second Race: Attach Versus I/O	1374
11.3.10	A Third Race: sysctl Versus I/O	1375
11.3.11	The Race Taxonomy, Revisited	1375
11.3.12	Wrapping Up Section 2	1375
11.4	Section 3: Analyzing Unsafe Code in Your Driver	1376
11.4.1	What “Shared” Means	1376
11.4.2	The Shared State of myfirst	1377
11.4.3	The Per-Descriptor State	1379
11.4.4	Classifying the Critical Sections	1380
11.4.5	Annotating the Source	1380
11.4.6	A Locking-Strategy Document	1381
11.4.7	A Practical Audit Procedure	1382
11.4.8	Common Audit Discoveries	1382
11.4.9	A Second Worked Audit: A Hypothetical Multi-Queue Driver	1383
11.4.10	How Audits Catch Bugs	1385
11.4.11	Audits as Change Gates	1385
11.4.12	Wrapping Up Section 3	1385
11.5	Section 4: Introduction to Atomic Operations	1386
11.5.1	What “Atomic” Means at the Hardware Level	1386
11.5.2	The FreeBSD atomic(9) API	1387
11.5.3	A Gentle Introduction to Memory Ordering	1387
11.5.4	When Atomics Suffice	1388
11.5.5	Per-CPU Counters: the counter(9) API	1389
11.5.6	Migrating the Driver’s Counters	1389
11.5.7	A Simple Atomic Flag Example	1391
11.5.8	The Compare-and-Swap Pattern	1391
11.5.9	When Atomics Are Not Enough	1392
11.5.10	A Worked Example: a “driver version” Counter	1392
11.5.11	Visibility, Ordering, and the Everyday Driver	1393
11.5.12	Memory Ordering on a Multi-Core Machine	1394
11.5.13	Atomic Swaps and More Flavours	1395
11.5.14	A Practical Example: Atomic Reference Counting	1395
11.5.15	counter(9) in More Depth	1396
11.5.16	Lock-Free Data Structures in Context	1396
11.5.17	When to Reach for an Atomic (Decision Guide)	1397
11.5.18	Wrapping Up Section 4	1397
11.6	Section 5: Introducing Locks (Mutexes)	1397
11.6.1	What a Mutex Is	1398
11.6.2	FreeBSD’s Two Mutex Shapes	1398
11.6.3	The Life Cycle of a Mutex	1398
11.6.4	Acquiring and Releasing	1399
11.6.5	The Other Operations	1400
11.6.6	Re-Examining the Chapter 10 Design	1400
11.6.7	The Sleep-with-Mutex Rule	1401
11.6.8	Priority Inversion and Priority Propagation	1401
11.6.9	Lock Order and Deadlocks	1402
11.6.10	WITNESS: What It Catches	1402
11.6.11	Running the Driver Under WITNESS	1403
11.6.12	When a Mutex Is the Wrong Answer	1404
11.6.13	A Mini-Walkthrough: Tracing a Lock	1404
11.6.14	Reading a Lock Contention Story	1405

11.6.15	Applying What We Have Learned	1405
11.6.16	Spin Mutexes in More Detail	1405
11.6.17	The MTX_RECURSE Flag and Lock Recursion	1406
11.6.18	Adaptive Spinning	1407
11.6.19	A Closer Look at mtx_sleep's Atomicity	1408
11.6.20	A Guided Tour Through kern_mutex.c	1408
11.6.21	Comparing Mutex to Semaphore, Monitor, and Binary Flag	1408
11.6.22	Mutex Lifetime Rules	1409
11.6.23	The MTX_NEW Option	1409
11.6.24	Nested Locks and Deadlock Patterns	1409
11.6.25	Wrapping Up Section 5	1410
11.7	Section 6: Testing Multi-threaded Access	1410
11.7.1	The Testing Ladder	1410
11.7.2	Multiple Threads in One Process	1411
11.7.3	Many Processes in Parallel	1413
11.7.4	Extended-Duration Tests	1417
11.7.5	Observing Without Disturbing	1417
11.7.6	A Testing Workflow	1417
11.7.7	A Latency-Measuring Tester	1418
11.7.8	A Tester That Probes for Torn Reads	1420
11.7.9	Multi-Core Scaling	1420
11.7.10	Coverage of the Failure Modes	1421
11.7.11	Wrapping Up Section 6	1421
11.8	Section 7: Debugging and Verifying Thread Safety	1421
11.8.1	Typical Symptoms	1422
11.8.2	INVARIANTS and KASSERT	1422
11.8.3	WITNESS in Action	1423
11.8.4	Reading a Kernel Panic Backtrace	1423
11.8.5	dtrace(1) as a Silent Observer	1424
11.8.6	A Debugging Checklist	1424
11.8.7	A Walkthrough: Diagnosing a Missing Wakeup	1425
11.8.8	Patterns for Adding Debug Traces	1428
11.8.9	Debugging the Hard Cases	1428
11.8.10	Wrapping Up Section 7	1429
11.9	Section 8: Refactoring and Versioning Your Concurrent Driver	1429
11.9.1	8.1: Organizing the Code for Clarity	1429
11.9.2	8.2: Versioning the Driver	1430
11.9.3	8.3: README and Comment Pass	1431
11.9.4	8.4: Static Analysis	1432
11.9.5	8.5: Regression Testing	1433
11.9.6	A Complete LOCKING.md Template	1434
11.9.7	A Pre-Commit Checklist	1436
11.9.8	Versioning Strategy: Why 0.x?	1436
11.9.9	A Note on Git Hygiene	1437
11.9.10	Wrapping Up Section 8	1437
11.10	Hands-on Labs	1438
11.10.1	Pre-Lab Setup Checklist	1438
11.10.2	Lab 11.1: Observe a Race Condition	1438
11.10.3	Lab 11.2: Verify the Locking Discipline with INVARIANTS	1439
11.10.4	Lab 11.3: Migrate to counter(9)	1439
11.10.5	Lab 11.4: Build a Multi-threaded Tester	1439

11.10.6 Lab 11.5: Deliberately Cause a WITNESS Warning	1440
11.10.7 Lab 11.6: Add KASSERTs Throughout	1440
11.10.8 Lab 11.7: Long-Running Stress	1441
11.10.9 Lab 11.8: Run clang -analyze	1441
11.11 Challenge Exercises	1442
11.11.1 Challenge 1: A Lock-Free Producer/Consumer	1442
11.11.2 Challenge 2: Multi-Mutex Design	1442
11.11.3 Challenge 3: WITNESS Under Pressure	1442
11.11.4 Challenge 4: Instrument with dtrace	1442
11.11.5 Challenge 5: Bounded Blocking	1443
11.11.6 Challenge 6: Write a Lock-Contention Benchmark	1443
11.11.7 Challenge 7: Port to a 32-bit Platform	1443
11.11.8 Challenge 8: Add a Per-Descriptor Mutex	1443
11.12 Troubleshooting	1443
11.12.1 Symptom: mtx_assert panic on a driver that used to work	1444
11.12.2 Symptom: sleepable after non-sleepable acquired warning	1444
11.12.3 Symptom: Lock-order reversal	1444
11.12.4 Symptom: Deadlock (hang)	1444
11.12.5 Symptom: Torn value read from a uint64_t	1444
11.12.6 Symptom: Double-free panic on the cbuf	1444
11.12.7 Symptom: Slow performance under many-core load	1445
11.12.8 Symptom: producer_consumer occasionally hangs	1445
11.12.9 Symptom: WITNESS reports “unowned lock released”	1445
11.12.10 Symptom: Compiles cleanly, passes single-threaded tests, fails mp_stress	1445
11.12.11 Symptom: kldunload hangs with descriptors open	1445
11.12.12 Symptom: Kernel panic with a clean backtrace in mtx_lock	1445
11.12.13 Symptom: WITNESS never warns, even when you expect it to	1446
11.12.14 Symptom: dmesg is silent but the test still fails	1446
11.13 Wrapping Up	1446
11.14 Chapter 11 Reference Material	1447
11.15 Reference: Building and Booting a Debug Kernel	1447
11.15.1 Step 1: Prepare a Custom Configuration	1447
11.15.2 Step 2: Build the Kernel	1448
11.15.3 Step 3: Install and Reboot	1448
11.15.4 Step 4: Confirm the Boot	1448
11.15.5 Step 5: Rebuild the Driver	1448
11.15.6 Reverting to the Production Kernel	1449
11.16 Reference: Further Reading on FreeBSD Concurrency	1449
11.16.1 Manual Pages	1449
11.16.2 Source Files	1449
11.16.3 External Material	1450
11.17 Reference: A Chapter 11 Self-Assessment	1450
11.17.1 Conceptual Questions	1450
11.17.2 Code Reading Questions	1451
11.17.3 Hands-On Questions	1451
11.18 Reference: A Closer Look at Sleep Queues	1451
11.18.1 The Sleep Queue	1451
11.18.2 The Atomicity of mtx_sleep	1452
11.18.3 The wakeup Path	1452
11.18.4 Priority Propagation Inside mtx_sleep	1453
11.18.5 Adaptive Spinning Revisited	1453

11.18.6 The Priority Argument	1453
11.18.7 Wait Message (wmesg) Conventions	1454
11.18.8 Sleep with Timeouts	1454
11.19 Reference: Reading kern_mutex.c	1454
11.19.1 The mutex Structure	1455
11.19.2 The Fast Path of mtx_lock	1455
11.19.3 The Slow Path	1456
11.19.4 The Unlock Path	1456
11.19.5 The Takeaway	1457
11.20 Reference: A Mini-Glossary of Concurrency Vocabulary	1457
11.21 Reference: Concurrency Patterns in Real FreeBSD Drivers	1458
11.21.1 Pattern: Per-Driver Softc Mutex	1458
11.21.2 Pattern: Sleep Channel on a softc Field	1458
11.21.3 Pattern: selinfo for poll() Support	1459
11.21.4 Pattern: Bounce Buffer with uiomove	1459
11.21.5 Pattern: Counter(9) for Statistics	1459
11.21.6 Pattern: Detach with Outstanding Users	1459
11.21.7 Pattern: WITNESS-Documented Lock Order	1460
11.21.8 Pattern: Assertions Everywhere	1460
11.21.9 What Real Drivers Do That Ours Does Not	1460
11.21.10 The Takeaway	1461
11.22 Reference: Common Mutex Idioms	1461
11.22.1 Basic Critical Section	1461
11.22.2 Assert Held	1461
11.22.3 Try-Lock	1461
11.22.4 Sleep Waiting for a Condition	1462
11.22.5 Wake Sleepers	1462
11.22.6 Sleep with Timeout	1462
11.22.7 Drop and Reacquire Around a Potentially-Sleeping Call	1463
11.22.8 Atomic-Instead-of-Lock for Simple Counters	1463
11.22.9 Check and Set Atomically	1463
11.23 Looking Ahead: Bridge to Chapter 12	1463
12 Synchronization Mechanisms	1465
12.1 Reader Guidance & Outcomes	1465
12.1.1 Why This Chapter Earns Its Own Place	1465
12.1.2 Where Chapter 11 Left the Driver	1466
12.1.3 What You Will Learn	1466
12.1.4 What This Chapter Does Not Cover	1467
12.1.5 Estimated Time Investment	1468
12.1.6 Prerequisites	1468
12.1.7 How to Get the Most Out of This Chapter	1468
12.1.8 Roadmap Through the Chapter	1469
12.2 Section 1: What Is Synchronization in the Kernel?	1469
12.2.1 What Synchronization Means	1470
12.2.2 Synchronization Versus Locking	1470
12.2.3 Blocking Versus Spinning	1470
12.2.4 A Small Map of FreeBSD Primitives	1471
12.2.5 Atomics in the Same Map	1472
12.2.6 A First Decision Tree	1472
12.2.7 A Worked Decision: Where Each myfirst State Sits	1472
12.2.8 Real-World Analogy: Doors, Hallways, and Whiteboards	1473

12.2.9	Comparing Primitives Side by Side	1473
12.2.10	A Note on Semaphores	1474
12.2.11	Wrapping Up Section 1	1474
12.3	Section 2: Condition Variables and Sleep/Wakeup	1475
12.3.1	Why Mutex Plus Wakeup Is Not Enough	1475
12.3.2	What a Condition Variable Is	1476
12.3.3	A Worked Refactor: Adding Two Condition Variables to myfirst	1477
12.3.4	When to Signal Versus Broadcast	1480
12.3.5	A Subtle Trap: cv_signal Without the Mutex	1481
12.3.6	Verifying the Refactor	1481
12.3.7	A Mental Model: How a cv_wait Plays Out	1482
12.3.8	A Look at kern_condvar.c	1483
12.3.9	Wrapping Up Section 2	1483
12.4	Section 3: Handling Timeouts and Interruptible Sleep	1483
12.4.1	What Goes Wrong With Indefinite Sleeps	1484
12.4.2	The Three Wait Outcomes	1484
12.4.3	EINTR, ERESTART, and the Restart Question	1485
12.4.4	Adding a Timeout to the Read Path	1485
12.4.5	Verifying the Timeout	1487
12.4.6	Choosing Between EAGAIN and ETIMEDOUT	1488
12.4.7	A Note on Fairness Under Timeouts	1489
12.4.8	When to Use a Timeout	1489
12.4.9	Sub-Tick Precision With sbintime_t	1489
12.4.10	What Happens Inside cv_timedwait_sig	1490
12.4.11	Wrapping Up Section 3	1491
12.5	Section 4: The sx(9) Lock: Shared and Exclusive Access	1491
12.5.1	What Shared and Exclusive Mean	1491
12.5.2	Why Shared/Exclusive Beats a Plain Mutex Here	1492
12.5.3	The sx(9) API	1492
12.5.4	A Quick Worked Example	1494
12.5.5	Try-Upgrade and Downgrade	1495
12.5.6	Comparing sx(9) to rw(9)	1496
12.5.7	The Sleepability Rule, Revisited	1497
12.5.8	Lock Order and sx	1497
12.5.9	A Look at kern_sx.c	1497
12.5.10	A Brief Tour of rw(9)	1498
12.5.11	A Worked Example with rw(9)	1499
12.5.12	Wrapping Up Section 4	1500
12.6	Section 5: Implementing a Safe Multi-reader, Single-writer Scenario	1500
12.6.1	The Configuration Subsystem	1500
12.6.2	The Lock Order Decision	1501
12.6.3	Reading the Configuration on the Data Path	1502
12.6.4	Reading the Soft Byte Limit	1503
12.6.5	Updating the Configuration: a Sysctl Writer	1504
12.6.6	Updating the Soft Byte Limit	1505
12.6.7	A Single Operation Holding Both Locks	1506
12.6.8	Verifying Against WITNESS	1508
12.6.9	What Goes Wrong if You Get the Order Backwards	1509
12.6.10	A Slightly Larger Pattern: Snapshot-and-Apply	1509
12.6.11	A Walk-Through of myfirst_sysctl_reset	1510
12.6.12	Wrapping Up Section 5	1511

12.7	Section 6: Debugging Synchronization Issues	1511
12.7.1	A Catalogue of Synchronization Bugs	1511
12.7.2	Reading a WITNESS Warning Carefully	1512
12.7.3	Using DDB to Inspect a Hung System	1513
12.7.4	Recognising Lost Wakeups	1513
12.7.5	Recognising Spurious Wakeups	1514
12.7.6	Recognising Lock Order Reversal	1515
12.7.7	Sleeping With a Non-Sleepable Lock	1515
12.7.8	Premature Destroy	1516
12.7.9	Helpful Asserts to Add	1517
12.7.10	Tracing Lock Activity With dtrace and lockstat	1517
12.7.11	A Worked Walk-Through: A Lost Wakeup Bug	1518
12.7.12	A Worked Walk-Through: a Lock Order Reversal	1520
12.7.13	Common Mistakes That Hide From WITNESS	1521
12.7.14	Wrapping Up Section 6	1521
12.8	Section 7: Stress Testing with Realistic I/O Patterns	1522
12.8.1	What Realistic Looks Like	1522
12.8.2	A Composite Workload	1522
12.8.3	Long-Duration Variant	1523
12.8.4	Latency Under Mixed Load	1524
12.8.5	Reading lockstat Output	1524
12.8.6	Tracing With dtrace	1524
12.8.7	Watching With vmstat and top	1525
12.8.8	Observing the Sysctls	1526
12.8.9	Stress Test Acceptance Criteria	1526
12.8.10	Interpreting lockstat Output in Detail	1526
12.8.11	Tracing Specific Code Paths With dtrace	1527
12.8.12	Wrapping Up Section 7	1528
12.9	Section 8: Refactoring and Versioning Your Synchronized Driver	1529
12.9.1	Cleaning Up the Source	1529
12.9.2	Updating LOCKING.md	1529
12.9.3	Bumping the Version	1533
12.9.4	Updating the README	1534
12.9.5	Running Static Analysis	1535
12.9.6	Running the Regression Suite	1535
12.9.7	Pre-Commit Checklist	1537
12.9.8	Wrapping Up Section 8	1537
12.10	Hands-on Labs	1538
12.10.1	Pre-Lab Setup Checklist	1538
12.10.2	Lab 12.1: Replace Anonymous Wakeup Channels With Condition Variables	1538
12.10.3	Lab 12.2: Add Bounded Reads	1539
12.10.4	Lab 12.3: Add an sx-Protected Configuration Subsystem	1539
12.10.5	Lab 12.4: Inspect Held Locks With DDB	1540
12.10.6	Lab 12.5: Detect a Deliberate Lock Order Reversal	1540
12.10.7	Lab 12.6: Long-Running Composite Stress	1541
12.10.8	Lab 12.7: Verify the Snapshot-and-Apply Pattern Holds Under Contention	1541
12.10.9	Lab 12.8: Profile With lockstat	1542
12.11	Challenge Exercises	1542
12.11.1	Challenge 1: Use sx_downgrade for a Config Refresh	1542
12.11.2	Challenge 2: Implement a Drain Operation Using cv_broadcast	1543
12.11.3	Challenge 3: A dtrace Script for cv_wait Latency	1543

12.11.4 Challenge 4: Replace cv With Anonymous Channels	1543
12.11.5 Challenge 5: Add Per-Descriptor read_timeout_ms	1543
12.11.6 Challenge 6: Use rw(9) Instead of sx(9)	1543
12.11.7 Challenge 7: Implement a Multi-CV Drain	1543
12.11.8 Challenge 8: Lock-Order Visualization	1543
12.11.9 Challenge 9: Sleep-Channel Comparison	1544
12.11.10 Challenge 10: Bound Configuration Writes	1544
12.12 Troubleshooting	1544
12.12.1 Symptom: Reader hangs forever despite data being written	1544
12.12.2 Symptom: WITNESS warns “lock order reversal” between sc->mtx and sc->cfg_sx	1544
12.12.3 Symptom: cv_timedwait_sig returns EWOULDBLOCK immediately	1544
12.12.4 Symptom: detach hangs	1545
12.12.5 Symptom: sysctl write hangs	1545
12.12.6 Symptom: sx_destroy panics with “lock still held”	1545
12.12.7 Symptom: cv_signal or cv_broadcast wakes nothing visible	1545
12.12.8 Symptom: read_timeout_ms set to 100 produces 200 ms latency	1545
12.12.9 Symptom: A deliberately-broken lock order does not produce a WITNESS warning	1545
12.12.10 Symptom: lockstat shows enormous wait times on the data mutex	1546
12.12.1 Symptom: mp_stress reports byte-count mismatch after Chapter 12 changes	1546
12.12.1 Symptom: timeout_tester shows latency longer than the configured timeout	1546
12.12.1 Symptom: kldunload reports busy when no descriptor is open	1546
12.12.1 Symptom: cv_wait_sig wakes immediately and returns 0	1546
12.12.1 Symptom: Two threads waiting on the same cv get woken in unexpected order	1546
12.12.1 Symptom: sx_xlock under heavy reader load takes seconds to acquire	1547
12.12.1 Symptom: Tests pass on a non-WITNESS kernel but fail on WITNESS	1547
12.12.1 Symptom: Locking macros expand to nothing on the non-debug kernel	1547
12.12.1 Symptom: Sysctl handler blocks the entire system	1547
12.12.2 Symptom: A reader gets EAGAIN even with read_timeout_ms=0	1547
12.12.2 Symptom: kldunload after a successful test still hangs briefly	1548
12.12.2 Symptom: Two separate driver instances both report WITNESS warnings about the same lock name	1548
12.12.2 Symptom: A cv with many waiters takes a long time to broadcast	1548
12.13 Wrapping Up	1548
12.13.1 What You Should Now Be Able to Do	1549
12.13.2 Three Closing Reminders	1549
12.14 Reference: The Driver’s Stage Progression	1549
12.14.1 Stage 1: cv-channels	1550
12.14.2 Stage 2: bounded-read	1550
12.14.3 Stage 3: sx-config	1550
12.14.4 Stage 4: final	1550
12.15 Reference: Migrating From mtx_sleep to cv	1551
12.15.1 Step 1: Identify Each Logical Wait Channel	1551
12.15.2 Step 2: Add cv Fields to the Softc	1551
12.15.3 Step 3: Replace mtx_sleep with cv_wait_sig	1551
12.15.4 Step 4: Replace wakeup with cv_signal or cv_broadcast	1552
12.15.5 Step 5: Update the Detach Path	1552
12.15.6 Step 6: Update LOCKING.md	1552
12.15.7 Step 7: Re-run the Stress Kit	1552
12.15.8 When Migration Is and Is Not Worth It	1553
12.16 Reference: Pre-Production Audit Checklist	1553
12.16.1 Lock Inventory	1553

12.16.2 Lock Order	1553
12.16.3 cv Inventory	1553
12.16.4 Wait Helpers	1554
12.16.5 Signal Sites	1554
12.16.6 Detach Path	1554
12.16.7 Static Analysis	1554
12.16.8 Documentation	1554
12.17 Reference: Sleep Channel Hygiene	1554
12.17.1 One Channel Per Logical Condition	1555
12.17.2 Channel Pointers Must Be Stable	1555
12.17.3 Channel Pointers Must Be Unique to Their Purpose	1555
12.17.4 Wakers Should Hold the Interlock When the State Could Change	1555
12.17.5 A Signal With No Waiter Is Free	1555
12.17.6 A Wait With No Signaller Is a Bug	1555
12.18 Reference: Common cv Idioms	1556
12.18.1 Wait for a Condition	1556
12.18.2 Signal One Waiter	1556
12.18.3 Broadcast a State Change	1556
12.18.4 Wait With a Timeout	1556
12.18.5 Wait With Detach Awareness	1557
12.18.6 Drain Waiters Before Destroy	1557
12.19 Reference: Common sx Idioms	1558
12.19.1 Read-Mostly Field	1558
12.19.2 Update With Validation	1558
12.19.3 Snapshot-and-Apply Across Two Locks	1558
12.19.4 Try-Upgrade Pattern	1559
12.19.5 Assert Held	1559
12.20 Reference: Decision Table for Synchronization Primitives	1559
12.21 Reference: Reading kern_condvar.c and kern_sx.c	1560
12.22 Reference: Common Mistakes With cv and sx	1561
12.22.1 cv Mistakes	1561
12.22.2 sx Mistakes	1561
12.22.3 Mistakes Specific to Combining Both	1561
12.23 Reference: Time Primitives	1562
12.24 Reference: Common WITNESS Warnings, Decoded	1563
12.24.1 “lock order reversal”	1563
12.24.2 “duplicate lock of same name”	1563
12.24.3 “sleeping thread (pid N) owns a non-sleepable lock”	1563
12.24.4 “exclusive sleep mutex foo not owned at”	1563
12.24.5 “lock list reversal”	1563
12.24.6 “sleepable acquired while holding non-sleepable”	1564
12.24.7 Acting on a Warning	1564
12.25 Reference: Lock Class Quick Reference	1564
12.26 Reference: Multi-Primitive Driver Design Patterns	1565
12.26.1 Pattern: One Mutex, One Configuration sx	1565
12.26.2 Pattern: Per-Queue Lock With a Configuration Lock	1565
12.26.3 Pattern: Per-Object Lock With a Container Lock	1565
12.26.4 Pattern: Per-CPU Counters With Mutex-Protected Tail	1566
12.26.5 Pattern: Snapshot-and-Apply Across Two Lock Classes	1566
12.27 Reference: Pre-Conditions for Each Primitive	1566
12.27.1 mtx(9) (MTX_DEF)	1566

12.27.2	cv(9)	1567
12.27.3	sx(9)	1567
12.27.4	rw(9)	1567
12.28	Reference: A Chapter 12 Self-Assessment	1567
12.28.1	Conceptual Questions	1568
12.28.2	Code Reading Questions	1568
12.28.3	Hands-On Questions	1568
12.29	Reference: Further Reading on Synchronization	1569
12.29.1	Manual Pages	1569
12.29.2	Source Files	1569
12.29.3	External Material	1570
12.30	Looking Ahead: Bridge to Chapter 13	1570
13	Timers and Delayed Work	1572
13.1	Reader Guidance & Outcomes	1572
13.1.1	Why This Chapter Earns Its Own Place	1572
13.1.2	Where Chapter 12 Left the Driver	1573
13.1.3	What You Will Learn	1573
13.1.4	What This Chapter Does Not Cover	1574
13.1.5	Estimated Time Investment	1574
13.1.6	Prerequisites	1575
13.1.7	How to Get the Most Out of This Chapter	1575
13.1.8	Roadmap Through the Chapter	1576
13.2	Section 1: Why Use Timers in a Driver?	1576
13.2.1	The Three Shapes of “Later”	1577
13.2.2	Real-World Patterns	1577
13.2.3	Why Not Just Use a Kernel Thread?	1578
13.2.4	Why Not Just Use cv_timedwait In a Loop?	1578
13.2.5	A Brief Note on Time Itself	1579
13.2.6	When a Callout Is the Wrong Tool	1579
13.2.7	A Mental Model: The Callout Wheel	1579
13.2.8	What “Now” Means in the Kernel	1580
13.2.9	Wrapping Up Section 1	1581
13.3	Section 2: Introduction to FreeBSD’s callout(9) API	1581
13.3.1	The Callout Structure	1581
13.3.2	The Callback Function Signature	1582
13.3.3	The Four Initialization Variants	1582
13.3.4	The flags Argument	1583
13.3.5	The Five Lifecycle Stages	1583
13.3.6	A First Look at the API	1584
13.3.7	Reading the Source	1585
13.3.8	A Worked Lifecycle Walkthrough	1585
13.3.9	A Word About “active” Versus “pending”	1586
13.3.10	Wrapping Up Section 2	1587
13.4	Section 3: Scheduling One-shot and Repeating Events	1587
13.4.1	The One-Shot Pattern	1587
13.4.2	The Periodic Pattern	1588
13.4.3	callout_schedule for Re-Arming Without Repeating Arguments	1589
13.4.4	Sub-Tick Precision With callout_reset_sbt	1589
13.4.5	Cancellation: callout_stop	1590
13.4.6	Cancellation: callout_drain	1590
13.4.7	callout_pending and callout_active	1591

13.4.8	A Worked Example: A Two-Stage Schedule	1592
13.4.9	Wrapping Up Section 3	1593
13.5	Section 4: Integrating Timers into Your Driver	1593
13.5.1	Adding a Heartbeat Callout	1593
13.5.2	The Heartbeat Callback	1594
13.5.3	Starting the Heartbeat From a Sysctl	1596
13.5.4	Verifying the Refactor	1597
13.5.5	A Note on the Heartbeat's Overhead	1598
13.5.6	A Mental Model: How a Heartbeat Plays Out	1599
13.5.7	Visualizing the Timer With dtrace	1599
13.5.8	Wrapping Up Section 4	1600
13.6	Section 5: Handling Locking and Context in Timers	1600
13.6.1	What the Kernel Does Before Your Callback Runs	1600
13.6.2	Why the Lock Is Acquired By the Kernel	1601
13.6.3	What You May Do Inside a Callback	1602
13.6.4	What You May Not Do Inside a Callback	1602
13.6.5	What If You Do Need To Sleep?	1602
13.6.6	The CALLOUT_RETURNUNLOCKED Flag	1603
13.6.7	The CALLOUT_SHAREDLOCK Flag	1603
13.6.8	The "Direct Execution" Mode	1603
13.6.9	A Worked Example: Lock Contract in the Heartbeat	1604
13.6.10	What Happens If Two Callouts Share a Lock	1605
13.6.11	What If the Callback Is Currently Firing When You Call <code>callout_reset</code> ?	1605
13.6.12	What If the Callback Is Currently Firing When You Call <code>callout_stop</code> ?	1606
13.6.13	<code>callout_stop</code> From a Context Without the Lock	1606
13.6.14	A Pattern: Conditional Re-Arm	1606
13.6.15	Wrapping Up Section 5	1607
13.7	Section 6: Timer Cleanup and Resource Management	1607
13.7.1	The Unload Race	1608
13.7.2	<code>callout_stop</code> vs <code>callout_drain</code>	1608
13.7.3	Two Critical Rules for <code>callout_drain</code>	1609
13.7.4	The <code>is_attached</code> Pattern, Revisited	1609
13.7.5	Why Not <code>callout_stop</code> Instead?	1610
13.7.6	<code>callout_async_drain</code>	1610
13.7.7	The Standard Detach Pattern with Timers	1610
13.7.8	A Note on Kernel-Module Unload	1612
13.7.9	Initializing After Drain	1613
13.7.10	A Worked Walkthrough: Catching the Unload Race in DDB	1613
13.7.11	What Happens If You Forget Drain on a Production Kernel	1615
13.7.12	What <code>callout_drain</code> Returns	1615
13.7.13	Detach Order with Multiple Callouts	1615
13.7.14	The Same Bug, Different Primitive: A Taskqueue Sketch	1615
13.7.15	Wrapping Up Section 6	1617
13.8	Section 7: Use Cases and Extensions for Timed Work	1617
13.8.1	Pattern 1: A Watchdog Timer	1617
13.8.2	Pattern 2: A Tick Source for Synthetic Events	1620
13.8.3	Pattern 3: A Debounce Shape	1622
13.8.4	Pattern 4: A Retry With Exponential Backoff	1623
13.8.5	Pattern 5: A Deferred Reaper	1624
13.8.6	Pattern 6: A Polling Loop Replacement	1626
13.8.7	Pattern 7: A Statistics Window	1626

13.8.8	Pattern 8: A Timed Status Refresh	1628
13.8.9	Pattern 9: A Periodic Reset	1628
13.8.10	Combining Patterns	1629
13.8.11	Wrapping Up Section 7	1629
13.9	Section 8: Refactoring and Versioning Your Timer-Enhanced Driver	1630
13.9.1	Cleaning Up the Source	1630
13.9.2	Updating LOCKING.md	1632
13.9.3	Bumping the Version	1633
13.9.4	Updating the README	1634
13.9.5	Running Static Analysis	1634
13.9.6	Running the Regression Suite	1635
13.9.7	Pre-Commit Checklist	1637
13.9.8	A Note on Backward Compatibility	1637
13.9.9	A Note on Sysctl Naming	1638
13.9.10	Wrapping Up Section 8	1638
13.10	Hands-on Labs	1638
13.10.1	Pre-Lab Setup Checklist	1638
13.10.2	Lab 13.1: Add a Heartbeat Callout	1639
13.10.3	Lab 13.2: Add a Watchdog Callout	1639
13.10.4	Lab 13.3: Add a Tick Source	1640
13.10.5	Lab 13.4: Verify Detach With Active Callouts	1640
13.10.6	Lab 13.5: Build a Debounce Timer	1641
13.10.7	Lab 13.6: Detect a Deliberate Race	1641
13.10.8	Lab 13.7: Long-Running Composite Stress with Timers	1641
13.10.9	Lab 13.8: Profile Callout Activity With dtrace	1642
13.10.10	Lab 13.9: Cancel a Watchdog Inline	1642
13.10.11	Lab 13.10: Schedule From Inside a Sysctl Handler	1643
13.11	Challenge Exercises	1643
13.11.1	Challenge 1: Sub-Millisecond Tick Source	1643
13.11.2	Challenge 2: Watchdog with Adaptive Intervals	1643
13.11.3	Challenge 3: Defer the Selwakeup to a Taskqueue	1643
13.11.4	Challenge 4: Multi-CPU Callout Distribution	1643
13.11.5	Challenge 5: Bound the Maximum Interval	1644
13.11.6	Challenge 6: Callout-Based Read Timeout	1644
13.11.7	Challenge 7: A Statistics Rollover	1644
13.11.8	Challenge 8: Drain Without Hold	1644
13.11.9	Challenge 9: Reuse a Callout Structure	1644
13.11.10	Challenge 10: Callout-Based Hello-World Module	1644
13.11.11	Challenge 11: Verify Lock Serialization	1644
13.11.12	Challenge 12: Coalesce Latency	1644
13.11.13	Challenge 13: Callout Wheel Inspection	1645
13.11.14	Challenge 14: Callout Function Pointer Replacement	1645
13.11.15	Challenge 15: Adaptive Heartbeat	1645
13.12	Troubleshooting	1645
13.12.1	Symptom: Callout never fires	1645
13.12.2	Symptom: kldunload panics shortly after	1645
13.12.3	Symptom: WITNESS warns “sleeping thread (pid X) owns a non-sleepable lock”	1645
13.12.4	Symptom: Heartbeat fires once and then never again	1646
13.12.5	Symptom: Callout fires more often than the configured interval	1646
13.12.6	Symptom: Detach hangs	1646
13.12.7	Symptom: WITNESS warns about lock-order issues with the callout’s lock	1646

13.12.8 Symptom: Callout-Drain Sleeps Forever	1646
13.12.9 Symptom: Callback runs but data is stale	1646
13.12.10 Symptom: <code>procstat -kk</code> shows no thread waiting on the callout	1647
13.12.11 Symptom: <code>callout_reset</code> returns 1 unexpectedly	1647
13.12.12 Symptom: Sysctl handler reports EINVAL for valid input	1647
13.12.13 Symptom: Two callbacks for different callouts run concurrently and deadlock	1647
13.12.14 Symptom: <code>tick_source</code> produces the wrong byte	1647
13.12.15 Symptom: <code>lockstat</code> shows the data mutex held for unusually long during heartbeats	1647
13.12.16 Symptom: Heartbeat continues after sysctl is set to 0	1648
13.12.17 Symptom: WITNESS warns about acquiring the callout's lock during init	1648
13.12.18 Symptom: A single fast callout causes high CPU usage	1648
13.12.19 Symptom: <code>dtrace</code> cannot find the callback function	1648
13.12.20 Symptom: <code>heartbeat_interval_ms</code> reads back as 0 after setting it	1648
13.12.21 Symptom: WITNESS warns "callout_init: lock has sleepable lock_class"	1648
13.12.22 Symptom: Detach takes seconds even when callouts seem idle	1649
13.12.23 Symptom: <code>callout_pending</code> returns true after <code>callout_stop</code>	1649
13.12.24 Symptom: A callout function appears in <code>dmesg</code> long after the driver was unloaded	1649
13.12.25 Symptom: Multiple callouts fire all at once after a long pause	1649
13.12.26 Symptom: A periodic callout drifts: each firing is slightly later than the previous	1649
13.12.27 Symptom: Callout never fires even though <code>callout_pending</code> returns true	1650
13.12.28 Symptom: Stress test causes intermittent panic in <code>callout_process</code>	1650
13.12.29 Symptom: <code>kern.callout.busy</code> counter grows under load	1650
13.12.30 Symptom: Driver logs show "callout_drain detected migration" or similar	1650
13.12.31 Symptom: <code>callout_reset_sbt</code> gives unexpected timing	1650
13.12.32 Symptom: A callout that was working stops firing after a power-management event	1650
13.12.33 Symptom: All callouts in the driver fire on the same CPU	1651
13.12.34 Symptom: <code>callout_drain</code> returns but the next syscall sees stale state	1651
13.12.35 Symptom: Re-arm in callback fails silently	1651
13.12.36 Symptom: Callout fires but <code>device_printf</code> is silent	1651
13.13 Reference: The Driver's Stage Progression	1651
13.13.1 Stage 1: heartbeat	1651
13.13.2 Stage 2: watchdog	1652
13.13.3 Stage 3: tick-source	1652
13.13.4 Stage 4: final	1652
13.14 Reference: Anatomy of a Real Watchdog	1652
13.14.1 Per-Request Tracking	1652
13.14.2 Threshold-Based Action	1653
13.14.3 Statistics	1653
13.14.4 Configurable Thresholds	1653
13.14.5 Recovery Logging	1653
13.14.6 Coordination With Other Subsystems	1654
13.15 Reference: Periodic vs Event-Driven Driver Architecture	1654
13.15.1 Event-Driven	1654
13.15.2 Polling-Driven	1654
13.15.3 Hybrid	1654
13.16 Wrapping Up	1655
13.16.1 What You Should Now Be Able to Do	1655
13.16.2 A Note On the Companion Examples	1656
13.17 Reference: <code>callout(9)</code> Quick Reference	1656
13.17.1 Initialization	1656
13.17.2 Scheduling	1657

13.17.3 Cancellation	1657
13.17.4 Inspection	1657
13.17.5 Common Flags	1657
13.18 Reference: Standard Detach Pattern	1657
13.19 Reference: When to Use Each Timing Primitive	1659
13.20 Reference: Callout Mistakes to Avoid	1659
13.21 Reference: Reading kern_timeout.c	1660
13.22 Reference: The c_iflags and c_flags Fields	1660
13.23 Reference: A Chapter 13 Self-Assessment	1660
13.23.1 Conceptual Questions	1661
13.23.2 Code Reading Questions	1662
13.23.3 Hands-On Questions	1662
13.24 Reference: Callouts in Real FreeBSD Drivers	1663
13.24.1 /usr/src/sys/dev/led/led.c	1663
13.24.2 /usr/src/sys/dev/uart/uart_core.c	1663
13.24.3 Network Driver Watchdogs	1663
13.24.4 Storage Driver I/O Timeouts	1664
13.24.5 USB Hub Polling	1664
13.24.6 What These Drivers Do Differently	1664
13.25 Reference: Comparing callout to Other Time Primitives	1664
13.25.1 callout(9) vs cv_timedwait_sig	1664
13.25.2 callout(9) vs taskqueue(9)	1664
13.25.3 callout(9) vs Kernel Threads	1665
13.25.4 callout(9) vs Periodic SIGALRM in User Space	1665
13.25.5 callout(9) vs Hardware Timers	1665
13.26 Reference: Common Callout Vocabulary	1665
13.27 Reference: Timer Anti-Patterns to Avoid	1666
13.28 Reference: Tracing Callouts With dtrace	1667
13.28.1 Count Firings of a Specific Callback	1667
13.28.2 Histogram Time Spent In Callback	1667
13.28.3 Trace All Callout Resets	1667
13.28.4 Trace Callout Drains	1668
13.28.5 Per-CPU Callout Activity	1668
13.28.6 Identify Slow Callouts	1668
13.28.7 A Combined Diagnostic Script	1668
13.29 Reference: Inspecting Callout State From DDB	1669
13.29.1 show callout <addr>	1669
13.29.2 show callout_stat	1669
13.29.3 ps	1669
13.29.4 bt <thread>	1669
13.29.5 show all locks	1669
13.29.6 Combined: Inspecting a Hung Callout	1670
13.30 Reference: Comparing Tick-Based and SBT-Based APIs	1670
13.30.1 Tick-Based API	1670
13.30.2 SBT-Based API	1670
13.30.3 Conversion Helpers	1671
13.31 Reference: Pre-Production Callout Audit	1671
13.31.1 Callout Inventory	1671
13.31.2 Initialization	1671
13.31.3 Scheduling	1671
13.31.4 Callback Hygiene	1672

13.31.5 Cancellation	1672
13.31.6 Detach	1672
13.31.7 Documentation	1672
13.31.8 Testing	1672
13.32 Reference: Standardising Timers Across a Driver	1672
13.32.1 One Naming Convention	1673
13.32.2 One Init/Drain Pattern	1673
13.32.3 One Sysctl Handler Pattern	1673
13.32.4 One Callback Pattern	1674
13.32.5 One Documentation Pattern	1675
13.32.6 Why Standardise	1675
13.33 Reference: Further Reading on Timers	1675
13.33.1 Manual Pages	1675
13.33.2 Source Files	1676
13.33.3 Manual Pages To Read in Order	1676
13.33.4 External Material	1676
13.34 Reference: Callout Cost Analysis	1676
13.34.1 Cost At Rest	1677
13.34.2 Cost Per Firing	1677
13.34.3 Cost At Cancel/Drain	1677
13.34.4 Practical Implications	1677
13.34.5 Comparison To Other Approaches	1678
13.34.6 When To Worry About Cost	1678
13.35 Looking Ahead: Bridge to Chapter 14	1678
13.35.1 A Final Aside on Time	1679
14 Taskqueues and Deferred Work	1681
14.1 Reader Guidance & Outcomes	1681
14.1.1 Why This Chapter Earns Its Own Place	1681
14.1.2 Where Chapter 13 Left the Driver	1682
14.1.3 What You Will Learn	1682
14.1.4 What This Chapter Does Not Cover	1683
14.1.5 Estimated Time Investment	1684
14.1.6 Prerequisites	1684
14.1.7 How to Get the Most Out of This Chapter	1684
14.1.8 Roadmap Through the Chapter	1685
14.2 Section 1: Why Use Deferred Work in a Driver?	1685
14.2.1 The Callout Contract, Re-Read	1686
14.2.2 Constrained Contexts, Not Just Callouts	1686
14.2.3 Real-World Reasons to Defer	1687
14.2.4 Polling Versus Deferred Execution	1688
14.2.5 FreeBSD's Solutions	1688
14.2.6 When Deferral Is the Wrong Answer	1689
14.2.7 A Worked Example: Why the Tick Source Cannot Wake Pollers	1689
14.2.8 A Small Mental Model	1690
14.2.9 Wrapping Up Section 1	1691
14.3 Section 2: Introduction to taskqueue(9)	1691
14.3.1 The Task Structure	1691
14.3.2 Initialising a Task	1692
14.3.3 The Taskqueue Structure, From a Driver's Point of View	1693
14.3.4 The API, One Function at a Time	1694
14.3.5 The Predefined Taskqueues	1696

14.3.6	Comparing Taskqueues and Callouts	1697
14.3.7	Comparing Taskqueues and a Private Kernel Thread	1698
14.3.8	The Enqueue-Already-Pending Rule	1698
14.3.9	A Minimal Example, End to End	1699
14.3.10	Wrapping Up Section 2	1700
14.4	Section 3: Deferring Work from a Timer or Simulated Interrupt	1701
14.4.1	The Change in One Sentence	1701
14.4.2	The Softc Addition	1701
14.4.3	The Task Callback	1702
14.4.4	The <code>tick_source</code> Edit	1702
14.4.5	The Attach Change	1704
14.4.6	The Detach Change	1705
14.4.7	The Makefile	1706
14.4.8	Building and Loading	1706
14.4.9	Observing the Fix	1707
14.4.10	Unloading	1708
14.4.11	What We Just Did	1709
14.4.12	Common Mistakes to Avoid	1709
14.4.13	Wrapping Up Section 3	1709
14.5	Section 4: Taskqueue Setup and Cleanup	1710
14.5.1	Why a Private Taskqueue	1710
14.5.2	The Softc Addition	1710
14.5.3	Creating the Taskqueue in Attach	1711
14.5.4	Updating the Enqueue Call Sites	1711
14.5.5	The Teardown Sequence	1711
14.5.6	The Attach Failure Path	1712
14.5.7	Thread Naming Conventions	1714
14.5.8	Choosing Thread Count	1714
14.5.9	Choosing a Thread Priority	1715
14.5.10	A Worked Source Excerpt: <code>ale(4)</code>	1715
14.5.11	Regressing Chapter 13 Behaviour	1716
14.5.12	Observing the Private Taskqueue	1716
14.5.13	The Pre-Production Checklist for Stage 2	1717
14.5.14	Common Mistakes in Stage 2	1717
14.5.15	Wrapping Up Section 4	1717
14.6	Section 5: Prioritization and Coalescing of Work	1718
14.6.1	The Priority Ordering Rule	1718
14.6.2	The Coalescing Rule, Restated	1718
14.6.3	A Deliberate Batching Pattern: Stage 3	1719
14.6.4	The Coalescing in Action	1720
14.6.5	Using <code>pending</code> for Adaptive Batching	1721
14.6.6	The Enqueue Flags	1722
14.6.7	The <code>timeout_task</code> Variant	1722
14.6.8	A Stage 3 Timeout Task: Delayed Reset	1723
14.6.9	Observing the Delayed Reset	1725
14.6.10	When to Use <code>timeout_task</code> vs. Callout Plus Task	1726
14.6.11	Priority Ordering Across Task Kinds	1727
14.6.12	A Note on Fairness	1727
14.6.13	Wrapping Up Section 5	1727
14.7	Section 6: Real-World Patterns Using Taskqueues	1727
14.7.1	Pattern 1: Deferred Logging or Notification From an Edge Context	1728

14.7.2	Pattern 2: Delayed Reset or Reconfiguration	1729
14.7.3	Pattern 3: Post-Interrupt Processing (the Filter + Task Split)	1729
14.7.4	Pattern 4: Async copyin/copyout After Hardware Completion	1730
14.7.5	Pattern 5: Retry-With-Backoff After Transient Failure	1731
14.7.6	Pattern 6: Deferred Teardown	1732
14.7.7	Pattern 7: Scheduled Statistics Rollover	1733
14.7.8	Pattern 8: taskqueue_block During Delicate Configuration	1734
14.7.9	Pattern 9: taskqueue_drain_all at a Subsystem Boundary	1735
14.7.10	Pattern 10: Simulation-Grade Synthetic Event Generation	1735
14.7.11	A Selection From Real Drivers	1735
14.7.12	Wrapping Up Section 6	1736
14.8	Section 7: Debugging Taskqueues	1736
14.8.1	The Tools	1736
14.8.2	Common Bug 1: The Task Never Runs	1737
14.8.3	Common Bug 2: The Task Runs Too Often	1737
14.8.4	Common Bug 3: Use-After-Free In the Task Callback	1738
14.8.5	Common Bug 4: Deadlock Between Task and Driver Mutex	1738
14.8.6	Common Bug 5: Drain Hangs Forever	1738
14.8.7	The Break-and-Fix Exercise	1739
14.8.8	A DTrace One-Liner	1741
14.8.9	Diagnostic Sysctls to Add	1741
14.8.10	The Debug-Kernel Obligation	1742
14.8.11	Wrapping Up Section 7	1742
14.9	Section 8: Refactoring and Versioning Your Taskqueue Driver	1743
14.9.1	File Organisation	1743
14.9.2	The LOCKING.md Update	1744
14.9.3	The Version Bump	1745
14.9.4	The Final Regression Pass	1745
14.9.5	Keeping Documentation in Sync	1746
14.9.6	A Final Audit	1746
14.9.7	Wrapping Up Section 8	1747
14.10	Additional Topics: epoch(9), Grouptaskqueues, and Per-CPU Taskqueues	1747
14.10.1	epoch(9) in One Page	1747
14.10.2	Grouptaskqueues in One Page	1748
14.10.3	Per-CPU Taskqueues in One Page	1749
14.10.4	When to Use Which	1750
14.10.5	Wrapping Up Additional Topics	1750
14.11	Hands-On Labs	1751
14.11.1	Lab 1: Observing a Taskqueue Worker Thread	1751
14.11.2	Lab 2: Measuring Coalescing Under Load	1751
14.11.3	Lab 3: Verify the Detach Ordering	1752
14.11.4	Lab 4: Coalescing vs. Adaptive Batching	1753
14.12	Challenge Exercises	1754
14.12.1	Challenge 1: Per-File-Handle Task	1754
14.12.2	Challenge 2: Two-Tier Task Pipeline	1754
14.12.3	Challenge 3: Priority-Driven Task Ordering	1754
14.12.4	Challenge 4: Blocking Reconfiguration	1754
14.12.5	Challenge 5: Multi-Threaded Taskqueue	1755
14.12.6	Challenge 6: Implementing a Watchdog Using a Timeout Task	1755
14.12.7	Challenge 7: Load a Real Driver and Read Its Code	1755

14.13	Troubleshooting Reference	1756
14.13.1	The task never runs	1756
14.13.2	The task runs twice when you expected once	1756
14.13.3	The task runs once when you expected twice or more	1756
14.13.4	Kernel panic in the task callback	1756
14.13.5	Detach hangs	1757
14.13.6	kldunload returns EBUSY immediately	1757
14.13.7	Coalescing count stays at zero	1757
14.13.8	The private taskqueue's thread does not appear in <code>procstat -t</code>	1757
14.13.9	Deadlock between task and driver mutex	1757
14.13.10	<code>taskqueue_enqueue</code> fails with <code>EEXIST</code>	1757
14.13.11	<code>taskqueue_enqueue_timeout</code> does not seem to fire	1757
14.13.12	Re-arming a <code>timeout_task</code> is not replacing the deadline	1758
14.13.13	WITNESS complains about a lock order involving <code>tq_mutex</code>	1758
14.13.14	<code>procstat -kk</code> shows the taskqueue thread sleeping on a lock	1758
14.13.15	Task callbacks are slow	1758
14.13.16	Taskqueue deadlock during boot	1758
14.14	Wrapping Up	1758
14.14.1	A Reflection Before Chapter 15	1759
14.14.2	A Second Reflection: The Discipline	1760
14.14.3	A Short Note on Simplicity	1760
14.14.4	What to Do If You Are Stuck	1760
14.15	Bridge to Chapter 15	1760
14.15.1	A Final Aside on the Shape of the Kernel	1762
14.16	Reference: Pre-Production Taskqueue Audit	1762
14.16.1	Task Inventory	1762
14.16.2	Taskqueue Inventory	1762
14.16.3	Initialisation	1763
14.16.4	Enqueue Sites	1763
14.16.5	Callback Hygiene	1763
14.16.6	Cancellation	1763
14.16.7	Detach	1763
14.16.8	Documentation	1763
14.16.9	Testing	1764
14.17	Reference: Standardising Tasks Across a Driver	1764
14.17.1	One Naming Convention	1764
14.17.2	One Init/Drain Pattern	1764
14.17.3	One Callback Pattern	1765
14.17.4	One Documentation Pattern	1765
14.17.5	Why Standardise	1766
14.18	Reference: Further Reading on Taskqueues	1766
14.18.1	Manual Pages	1766
14.18.2	Source Files	1766
14.18.3	Manual Pages To Read in Order	1766
14.18.4	External Material	1767
14.19	Reference: Taskqueue Cost Analysis	1767
14.19.1	Cost At Rest	1767
14.19.2	Cost Per Enqueue	1767
14.19.3	Cost Per Dispatch	1768
14.19.4	Cost At Cancel/Drain	1768
14.19.5	Practical Implications	1768

14.19.6 Comparison To Other Approaches	1769
14.19.7 When To Worry About Cost	1769
14.20 Reference: The Task Coalescing Semantics, Precisely	1769
14.20.1 The Rule	1769
14.20.2 Worked Examples	1770
14.20.3 Design Implications	1771
14.21 Reference: The Taskqueue State Diagram	1771
14.22 Reference: Observability Cheat Sheet	1772
14.22.1 List Every Taskqueue Thread	1772
14.22.2 List Every Task Submission Rate With DTrace	1772
14.22.3 Measure Dispatch Latency	1773
14.22.4 Measure Callback Duration	1773
14.22.5 Stack Trace of the Taskqueue Thread When Stuck	1773
14.22.6 Active Tasks Inside ddb	1773
14.22.7 Sysctl Knobs Your Driver Should Provide	1774
14.23 Reference: A Minimal Working Task Template	1774
14.24 Reference: Comparison With Linux Workqueues	1776
14.24.1 Naming	1776
14.24.2 Semantic Differences	1777
14.25 Reference: When Not To Use a Taskqueue	1777
14.26 Reference: A Worked Reading of <code>subr_taskqueue.c</code>	1777
14.27 A Final Tour: Five Common Shapes	1778
14.27.1 Shape A: The Solo Task	1778
14.27.2 Shape B: The Filter-Plus-Task Split	1779
14.27.3 Shape C: The Callout-Driven Periodic Task	1779
14.27.4 Shape D: The Timeout Task	1779
14.27.5 Shape E: The Self-Re-Enqueuing Task	1780
14.28 Closing Summary: What The Chapter Delivered	1780
14.29 Reference: Reading <code>taskqueue_run_locked</code> Line By Line	1781
14.30 Reference: A Walkthrough of <code>taskqueue_drain</code>	1784
14.31 Reference: The Lifecycle As Seen From the Softc	1785
14.32 Reference: A Glossary of Terms	1785
15 More Synchronization: Conditions, Semaphores, and Coordination	1787
15.1 Reader Guidance & Outcomes	1787
15.1.1 Why This Chapter Earns Its Own Place	1787
15.1.2 Where Chapter 14 Left the Driver	1788
15.1.3 What You Will Learn	1788
15.1.4 What This Chapter Does Not Cover	1789
15.1.5 Estimated Time Investment	1790
15.1.6 Prerequisites	1790
15.1.7 How to Get the Most Out of This Chapter	1790
15.1.8 Roadmap Through the Chapter	1791
15.2 Section 1: When Mutexes and Condition Variables Are Not Enough	1791
15.2.1 The Shape of the Mismatch	1792
15.2.2 A Concrete Motivating Example: Too Many Writers	1793
15.2.3 The Rest of the Section Previews the Chapter	1794
15.2.4 Wrapping Up Section 1	1795
15.3 Section 2: Using Semaphores in the FreeBSD Kernel	1795
15.3.1 The Counting Semaphore, Precisely	1795
15.3.2 The Data Structure	1795
15.3.3 The API	1796

15.3.4	What a Semaphore Is Not	1797
15.3.5	A Real-World Example: Hyper-V storvsc	1797
15.3.6	The Writer-Cap Refactor: Stage 1	1798
15.3.7	The Softc Addition	1798
15.3.8	Initialising and Destroying the Semaphore	1798
15.3.9	Using the Semaphore in the Write Path	1800
15.3.10	The Sysctl Handler for the Limit	1801
15.3.11	Observing the Effect	1802
15.3.12	Common Mistakes	1803
15.3.13	A Note on When Not to Use Semaphores	1804
15.3.14	Wrapping Up Section 2	1804
15.4	Section 3: Read-Mostly Scenarios and Shared Access	1804
15.4.1	The Read-Mostly Cache Problem	1804
15.4.2	<code>sx_try_upgrade</code> and <code>sx_downgrade</code>	1805
15.4.3	A Worked Application: Stage 2 Driver	1806
15.4.4	The Cache Refresh Function	1806
15.4.5	The Sysctl Handler	1807
15.4.6	Attach and Detach	1807
15.4.7	Observing the Effect	1808
15.4.8	The <code>sx</code> Macro Vocabulary	1808
15.4.9	Trade-offs and Cautions	1809
15.4.10	Wrapping Up Section 3	1809
15.5	Section 4: Condition Variables with Timeout and Interruption	1809
15.5.1	What the Return Values Mean	1810
15.5.2	The Partial-Progress Convention	1810
15.5.3	The Refactored Read Path	1810
15.5.4	Handling at the Caller	1812
15.5.5	The Write Path Gets the Same Treatment	1813
15.5.6	The Interruptible Wait Pattern	1813
15.5.7	Common Mistakes	1814
15.5.8	Wrapping Up Section 4	1814
15.6	Section 5: Synchronization Between Modules or Subsystems	1815
15.6.1	The Shutdown-Flag Problem	1815
15.6.2	The Atomic API in One Page	1815
15.6.3	Why Not Just a Mutex-Protected Flag?	1816
15.6.4	Applying the Atomic Flag	1816
15.6.5	The Callout-to-Task Handshake	1817
15.6.6	The Stage 4 Softc	1818
15.6.7	Memory Ordering in One Subsection	1819
15.6.8	Wrapping Up Section 5	1819
15.7	Section 6: Synchronization and Modular Design	1819
15.7.1	Why Encapsulate	1820
15.7.2	The Shape of <code>myfirst_sync.h</code>	1820
15.7.3	The Implementation	1821
15.7.4	How the Source Changes	1823
15.7.5	Naming Conventions	1823
15.7.6	What Not to Put in the Header	1824
15.7.7	A Related Discipline: Assertions	1824
15.7.8	A Short WITNESS Walkthrough: Sleeping Under a Non-Sleepable Lock	1824
15.7.9	Wrapping Up Section 6	1826

15.8	Section 7: Testing Advanced Synchronization	1826
15.8.1	Why Stress Tests Matter	1826
15.8.2	Test 1: Writer-Cap Correctness	1826
15.8.3	Test 2: Stats Cache Concurrency	1828
15.8.4	Test 3: Detach Under Load	1828
15.8.5	Observing With DTrace	1829
15.8.6	WITNESS, INVARIANTS, and the Debug Kernel	1830
15.8.7	Regression Discipline	1830
15.8.8	Wrapping Up Section 7	1830
15.9	Section 8: Refactoring and Versioning Your Coordinated Driver	1831
15.9.1	File Organisation	1831
15.9.2	The LOCKING.md Update	1831
15.9.3	The Version Bump	1834
15.9.4	The Final Regression Pass	1834
15.9.5	Documentation Audit	1835
15.9.6	A Final Audit Checklist	1835
15.9.7	Wrapping Up Section 8	1835
15.10	Additional Topics: Atomic Operations, epoch(9) Revisited, and Memory Ordering	1835
15.10.1	Atomic Operations in More Depth	1836
15.10.2	epoch(9) In One Page	1836
15.10.3	Memory Ordering, A Little Deeper	1837
15.10.4	When to Reach for Each Tool	1837
15.10.5	Wrapping Up Additional Topics	1837
15.11	Hands-On Labs	1838
15.11.1	Lab 1: Observe Writer-Cap Enforcement	1838
15.11.2	Lab 2: Stats Cache Contention	1838
15.11.3	Lab 3: Signal Handling and Partial Progress	1839
15.11.4	Lab 4: Detach Under Maximum Load	1840
15.12	Challenge Exercises	1840
15.12.1	Challenge 1: Replace sema with Interruptible Wait	1840
15.12.2	Challenge 2: Read-Mostly with a Background Refresher	1841
15.12.3	Challenge 3: Multiple Writer-Cap Semaphores	1841
15.12.4	Challenge 4: Atomic-Based Recovery Flag Elimination	1841
15.12.5	Challenge 5: Epoch for a Hypothetical Read Path	1841
15.12.6	Challenge 6: Stress Test a Specific Race	1841
15.12.7	Challenge 7: Read a Real Driver's Synchronization Vocabulary	1842
15.13	Troubleshooting Reference	1842
15.13.1	Semaphore Deadlock or Leak	1842
15.13.2	Sx Lock Issues	1842
15.13.3	Signal-Handling Issues	1842
15.13.4	Atomic and Memory Ordering	1843
15.13.5	Coordination Bugs	1843
15.13.6	Testing Issues	1843
15.13.7	Detach Issues	1843
15.14	Wrapping Up Part 3	1843
15.14.1	The Mutex (Chapter 11)	1844
15.14.2	The Condition Variable (Chapter 12)	1844
15.14.3	The Shared/Exclusive Lock (Chapter 12)	1844
15.14.4	The Callout (Chapter 13)	1844
15.14.5	The Taskqueue (Chapter 14)	1844
15.14.6	The Semaphore (Chapter 15)	1844

15.14.7 Atomic Operations (Chapter 15)	1844
15.14.8 Encapsulation (Chapter 15)	1844
15.14.9 What the Driver Can Now Do	1845
15.14.10 What Part 3 Did Not Cover	1845
15.14.11 A Reflection	1845
15.15 Part 3 Checkpoint	1846
15.16 Bridge to Chapter 16	1847
15.17 Reference: Pre-Production Synchronisation Audit	1848
15.17.1 Mutex Audit	1848
15.17.2 Condition Variable Audit	1848
15.17.3 Sx Audit	1848
15.17.4 Semaphore Audit	1848
15.17.5 Callout Audit	1849
15.17.6 Task Audit	1849
15.17.7 Atomic Audit	1849
15.17.8 Cross-Component Audit	1849
15.17.9 Documentation Audit	1849
15.17.10 Testing Audit	1849
15.18 Reference: Synchronisation Primitive Cheat Sheet	1850
15.18.1 When to Use Which	1850
15.18.2 API Quick Reference	1850
15.18.3 Context Rules	1851
15.18.4 Partial Progress Convention	1851
15.18.5 Detach Ordering	1851
15.19 Reference: Further Reading	1851
15.19.1 Manual Pages	1851
15.19.2 Source Files	1852
15.19.3 Books and External Material	1852
15.19.4 Suggested Reading Order	1852
15.20 Reference: A Glossary of Chapter 15 Terms	1852
15.21 Reference: Reading kern_sema.c Line by Line	1853
15.21.1 sema_init	1853
15.21.2 sema_destroy	1854
15.21.3 _sema_post	1854
15.21.4 _sema_wait	1855
15.21.5 _sema_timedwait	1856
15.21.6 _sema_trywait	1857
15.21.7 sema_value	1857
15.21.8 Observations	1857
15.22 Reference: Standardising Synchronization Primitives Across a Driver	1858
15.22.1 One Naming Convention	1858
15.22.2 One Init/Destroy Pattern	1858
15.22.3 One Encapsulation Pattern	1858
15.22.4 One LOCKING.md Template	1859
15.22.5 Why Standardise	1859
15.23 Reference: When Each Primitive Is Wrong	1859
15.23.1 Mutex Misuses	1859
15.23.2 Cv Misuses	1859
15.23.3 Sx Misuses	1859
15.23.4 Sema Misuses	1860
15.23.5 Atomic Misuses	1860

15.23.6 Pattern Misuses	1860
15.24Reference: Observability Cheat Sheet	1860
15.24.1 Sysctl Knobs Your Driver Should Expose	1860
15.24.2 DTrace Probes	1861
15.24.3 procstat	1862
15.24.4 ps	1862
15.24.5 ddb	1862
15.25Reference: A Worked Stage-by-Stage Diff Summary	1862
15.25.1 Stage 1 Diff (v0.8 -> v0.8+writers_sema)	1862
15.25.2 Stage 2 Diff (+stats_cache_sx)	1863
15.25.3 Stage 3 Diff (EINTR/ERESTART + partial-progress)	1863
15.25.4 Stage 4 Diff (coordination + encapsulation)	1863
15.25.5 Total Lines Added	1864
15.26Reference: The Chapter 15 Driver Lifecycle	1864
15.26.1 Attach Sequence	1864
15.26.2 Runtime	1865
15.26.3 Detach Sequence	1865
15.26.4 Notes on the Sequence	1865
15.27Reference: Costs and Comparisons	1866
15.28Reference: A Worked Synchronization Walk	1866
15.29Reference: A Minimal Working Template	1868
15.30Reference: Comparison With POSIX User-Space Synchronization	1870
15.31Reference: A Worked Pattern Catalog	1871
15.31.1 Pattern 1: Producer/Consumer With Bounded Queue	1871
15.31.2 Pattern 2: Completion Semaphore	1871
15.31.3 Pattern 3: Admission Control	1871
15.31.4 Pattern 4: Upgrade-Promote-Downgrade Read Cache	1871
15.31.5 Pattern 5: Atomic Flag Coordination	1871
15.31.6 Pattern 6: At-Most-One State Flag	1871
15.31.7 Pattern 7: Signal-Interruptible Bounded Wait	1872
15.31.8 Pattern 8: Periodic Refresh via Callout	1872
15.31.9 Pattern 9: Deferred Teardown via Reference Count	1872
15.31.10Pattern 10: Cross-Subsystem Handshake	1872
15.32Reference: A Glossary of Part 3 Terms	1872
15.33Reference: A Worked Debugging Scenario	1873
15.33.1 Step 1: Reproduce	1874
15.33.2 Step 2: Capture the Stack	1874
15.33.3 Step 3: Identify the Race	1875
15.33.4 Step 4: Apply the Fix	1875
15.33.5 Step 5: Verify	1875
15.33.6 Step 6: Document	1876
15.33.7 Takeaways	1876
15.34Reference: Reading a Real Driver's Synchronization	1876
15.34.1 Primitives It Uses	1876
15.34.2 Patterns It Demonstrates	1877
15.34.3 What It Does Not Demonstrate	1877
15.34.4 Takeaways	1877
15.35Reference: The Full myfirst_sync.h Design	1877
15.36Reference: Extended Lab: The "Break a Primitive" Exercise	1881
15.36.1 Break the Writer-Cap	1882
15.36.2 Break the Sx Upgrade	1882

15.36.3 Break the Partial-Progress Handling	1882
15.36.4 Break the Atomic Read	1882
15.36.5 Break the Detach Ordering	1883
15.36.6 Break the Sema Destroy Lifetime	1883
15.36.7 Why This Matters	1883
15.37 Reference: When To Split A Driver	1883
15.38 Reference: A Final Summary of Part 3	1884
15.39 Reference: Chapter 15 Deliverables Checklist	1884
15.39.1 Chapter Content	1884
15.39.2 Examples	1884
15.39.3 Documentation	1885
15.39.4 Testing	1885
15.40 Reference: An Invitation to Experiment	1885
15.41 Reference: cv_signal vs cv_broadcast, Precisely	1886
15.41.1 The Semantic Difference	1886
15.41.2 When cv_signal Is Correct	1886
15.41.3 When cv_broadcast Is Correct	1886
15.41.4 A Subtle Case: The Reset Sysctl	1887
15.41.5 Cost Consideration	1887
15.41.6 Rules of Thumb	1887
15.42 Reference: The Rare Case Where You Write Your Own Semaphore	1887
15.43 Reference: A Parting Observation on Kernel Synchronization	1889
15.44 Reference: A Final Note on Testing Discipline	1889
16 Accessing Hardware	1891
16.1 Reader Guidance & Outcomes	1891
16.1.1 Why bus_space(9) Earns a Chapter of Its Own	1892
16.1.2 Where Chapter 15 Left the Driver	1892
16.1.3 What You Will Learn	1893
16.1.4 What This Chapter Does Not Cover	1893
16.1.5 Estimated Time Investment	1894
16.1.6 Prerequisites	1894
16.1.7 How to Get the Most Out of This Chapter	1894
16.1.8 Roadmap Through the Chapter	1895
16.2 Section 1: What Is Hardware I/O?	1896
16.2.1 The Device as a Cooperating Partner	1896
16.2.2 What “Accessing Hardware” Really Means	1897
16.2.3 Why a Raw Pointer Cast Is Not How Drivers Reach Hardware	1897
16.2.4 Categories of Resources a Driver Cares About	1898
16.2.5 The Register, Closer Up	1899
16.2.6 A First Mental Model: The Control Panel	1899
16.2.7 Why Simulating Hardware Is a Good First Practice	1900
16.2.8 A Glance at Real Drivers That Use Hardware I/O	1900
16.2.9 What Comes Next in This Chapter	1901
16.2.10 Wrapping Up Section 1	1901
16.3 Section 2: Understanding Memory-Mapped I/O (MMIO)	1901
16.3.1 Physical Addresses and Device Memory	1902
16.3.2 The Virtual Mapping	1902
16.3.3 Cache Attributes Matter	1903
16.3.4 Alignment	1903
16.3.5 Endianness	1904
16.3.6 Read and Write Side Effects	1905

16.3.7	A Concrete Picture: A Device's Register Block	1905
16.3.8	The Shape of a Register Access	1906
16.3.9	What MMIO Is Not	1907
16.3.10	Wrapping Up Section 2	1907
16.4	Section 3: Introduction to <code>bus_space(9)</code>	1907
16.4.1	The Two Types: <code>bus_space_tag_t</code> and <code>bus_space_handle_t</code>	1908
16.4.2	Offsets	1908
16.4.3	The Read Functions	1909
16.4.4	The Write Functions	1910
16.4.5	The Multi and Region Helpers	1911
16.4.6	The Barrier Function	1912
16.4.7	The <code>bus_*</code> Shorthand over a <code>struct resource *</code>	1913
16.4.8	Setup and Teardown	1914
16.4.9	A First Stand-Alone Example	1915
16.4.10	A Look at a Real Driver's <code>bus_space</code> Usage	1916
16.4.11	Wrapping Up Section 3	1916
16.5	Section 4: Simulating Hardware for Testing	1917
16.5.1	What "Simulating Hardware" Means Here	1917
16.5.2	The Register Map for the Simulated Device	1917
16.5.3	The Register Offsets Header	1918
16.5.4	Allocating the Register Block	1920
16.5.5	The First Accessor Helpers	1921
16.5.6	Exposing the Registers Through Sysctls	1923
16.5.7	A Writeable Sysctl for <code>CTRL</code> and <code>DATA_IN</code>	1924
16.5.8	The Observer Pattern: Coupling Registers to Driver State	1925
16.5.9	What Stage 1 Accomplished	1927
16.5.10	A Note on What Stage 1 Is Not	1927
16.5.11	Wrapping Up Section 4	1928
16.6	Section 5: Using <code>bus_space</code> in a Real Driver Context	1928
16.6.1	The Pedagogical Trick: Using <code>bus_space</code> on Kernel Memory	1928
16.6.2	Stage 2: Setting Up the Simulated Tag and Handle	1928
16.6.3	Replacing the Accessors	1929
16.6.4	Exposing <code>DATA_IN</code> Through the Write Path	1930
16.6.5	Exposing <code>DATA_OUT</code> Through the Read Path	1932
16.6.6	Driving Register State From a Task	1933
16.6.7	The Full Stage 2 Sysctl Tree	1935
16.6.8	A Note on Reads From <code>STATUS</code> While the Driver Is Running	1935
16.6.9	The Detach Path, Updated	1936
16.6.10	Updating <code>LOCKING.md</code> (Now <code>HARDWARE.md</code> Too)	1936
16.6.11	Stage 2 Complete	1937
16.6.12	A Look at a Real Pattern: The <code>em</code> Attach Path	1938
16.6.13	Wrapping Up Section 5	1938
16.7	Section 6: Safety and Synchronization with MMIO	1939
16.7.1	Why a Register Access Can Be Unsafe Without a Lock	1939
16.7.2	Adding the Lock	1939
16.7.3	Why Barriers Matter Even On x86	1942
16.7.4	Partitioning Register Access by Context	1943
16.7.5	Avoiding Busy-Wait Loops	1944
16.7.6	Interrupts and MMIO: A Forward Pointer	1946
16.7.7	Stage 3 Complete	1946
16.7.8	Wrapping Up Section 6	1946

16.8	Section 7: Debugging and Tracing Hardware Access	1947
16.8.1	What You Want to Observe	1947
16.8.2	A Simple Access Log	1947
16.8.3	Kernel Printf: A Controlled Flood	1950
16.8.4	DTrace Probes	1950
16.8.5	Specialised DTrace Probes: <code>sdt(9)</code>	1951
16.8.6	The Heartbeat Log From Stage 1 Onwards	1952
16.8.7	Using <code>kgdb</code> on a Core Dump	1953
16.8.8	DDB Extensions	1953
16.8.9	What to Do When a Register Read Returns Garbage	1954
16.8.10	What to Do When a Register Write Has No Effect	1954
16.8.11	A Lab: The Access Log Does Its Job	1955
16.8.12	Wrapping Up Section 7	1955
16.9	Section 8: Refactoring and Versioning Your MMIO-Ready Driver	1955
16.9.1	The File Split	1956
16.9.2	The Hardware State Structure	1956
16.9.3	The CSR Macros	1958
16.9.4	Moving <code>Sysctls</code>	1959
16.9.5	Final <code>HARDWARE.md</code>	1959
16.9.6	The Version Bump	1961
16.9.7	The Final Regression Pass	1961
16.9.8	Running the Final Stage	1962
16.9.9	A Small Rule for Chapter 16's Refactor	1962
16.9.10	What Stage 4 Accomplished	1962
16.9.11	Wrapping Up Section 8	1963
16.10	Hands-On Labs	1963
16.10.1	Lab 1: Observe the Register Dance	1963
16.10.2	Lab 2: Trigger a Lock Violation (Debug Kernel)	1964
16.10.3	Lab 3: Simulate a Concurrent Writer Race	1964
16.10.4	Lab 4: Watch the Heartbeat Register Log	1964
16.10.5	Lab 5: Add a New Register	1965
16.10.6	Lab 6: Inject a Bogus Access (Debug Kernel)	1965
16.10.7	Lab 7: Trace With DTrace	1965
16.10.8	Lab 8: The Watchdog-Meets-Register Scenario	1966
16.11	Challenge Exercises	1966
16.11.1	Challenge 1: Per-File-Handle Register Snapshot	1967
16.11.2	Challenge 2: Register Diff Logging	1967
16.11.3	Challenge 3: Loopback Mode	1967
16.11.4	Challenge 4: Read-to-Clear on <code>INTR_STATUS</code>	1967
16.11.5	Challenge 5: A Barrier Correctness Stress Test	1967
16.11.6	Challenge 6: A Register-Aware Lockstat Run	1968
16.11.7	Challenge 7: Read a Real Driver's Register Interface	1968
16.12	Troubleshooting Reference	1968
16.12.1	Driver fails to load	1968
16.12.2	Driver loads but <code>dmesg</code> shows <code>myfirst: cannot allocate register block</code>	1968
16.12.3	<code>sysctl dev.myfirst.0.reg_ctrl</code> returns <code>ENOENT</code>	1969
16.12.4	<code>sysctl dev.myfirst.0.reg_ctrl</code> returns a plausible value, but changes never happen	1969
16.12.5	Kernel panic on first register write: "page fault in kernel mode"	1969
16.12.6	Kernel panic: "myfirst: register read past end of register block"	1969
16.12.7	<code>WITNESS</code> warning: "acquiring duplicate lock"	1969
16.12.8	<code>WITNESS</code> warning: "lock order reversal"	1969

16.12.9 Ticker does not fire	1969
16.12.10 Access log returns empty	1969
16.12.11 dmesg shows no output from myfirst_ctrl_update	1970
16.12.12 kldunload myfirst returns EBUSY	1970
16.12.13 kldunload myfirst hangs	1970
16.12.14 The stress test complains about WITNESS warnings	1970
16.13 Wrapping Up	1970
16.13.1 A Reflection Before Chapter 17	1970
16.13.2 What to Do If You Are Stuck	1971
16.14 Bridge to Chapter 17	1971
16.15 Reference: bus_space(9) Cheat Sheet	1972
16.15.1 Types	1972
16.15.2 Reads	1972
16.15.3 Writes	1973
16.15.4 Multi Accesses (same offset, different buffer positions)	1973
16.15.5 Region Accesses (advancing offset and buffer)	1973
16.15.6 Barrier	1973
16.15.7 Resource Shorthand (/usr/src/sys/sys/bus.h)	1974
16.15.8 Allocation	1974
16.15.9 Types of Resource	1974
16.15.10 Flags	1974
16.16 Reference: Further Reading	1974
16.16.1 Manual Pages	1974
16.16.2 Source Files	1975
16.16.3 Reading Order	1975
16.17 Reference: A Glossary of Chapter 16 Terms	1975
16.17.1 Terms Introduced in This Chapter	1975
16.17.2 Terms Previously Introduced (Reminders)	1976
16.18 Reference: The Chapter 16 Driver Diff Summary	1976
16.18.1 Stage 1 (Section 4)	1977
16.18.2 Stage 2 (Section 5)	1977
16.18.3 Stage 3 (Section 6)	1977
16.18.4 Stage 4 (Section 8)	1977
16.18.5 Lines of Code	1977
16.19 Reference: A Comparison with Linux Device Register Access	1978
16.19.1 Linux: ioremap + readl / writel	1978
16.19.2 FreeBSD: bus_alloc_resource + bus_read_* / bus_write_*	1978
16.19.3 What Translates	1978
16.19.4 What Differs	1978
16.20 Reference: A Worked Example: The Full myfirst_hw.h	1979
16.21 Reference: A Worked Example: The myfirst_hw.c Accessor Functions	1981
16.22 Reference: A Minimal Stand-Alone Test Module	1983
16.23 Reference: Why volatile Matters in bus_space	1986
16.24 Reference: A Short Comparison of Access Patterns in Real FreeBSD Drivers	1986
16.25 Reference: The Road Ahead in Part 4	1987
16.26 Reference: How to Read a Datasheet	1988
16.26.1 The Shape of a Datasheet	1988
16.26.2 Reading the Register Reference	1988
16.26.3 Reading the Programming Model	1989
16.26.4 Extracting the C Header	1989
16.26.5 A Pattern for Each Type of Register	1989

16.26.6 An Exercise: Pretend the Chapter 16 Device Has a Datasheet	1990
16.27 Reference: A Case Study in Missing Barriers	1990
16.28 Reference: Reading <code>if_ale.c</code> Step By Step	1991
16.28.1 Step 1: The Attach Entry Point	1991
16.28.2 Step 2: Early Locking Setup	1992
16.28.3 Step 3: PCI Bus-Mastering and Resource Allocation	1992
16.28.4 Step 4: Reading the First Register	1993
16.28.5 Step 5: Reading More Registers	1993
16.28.6 Step 6: IRQ Setup	1993
16.28.7 Step 7: Reading the Whole File	1994
16.29 Reference: An Honest Accounting of Chapter 16's Simplifications	1994
16.29.1 The Simulated Tag	1994
16.29.2 The Register Protocol	1994
16.29.3 Locking Granularity	1994
16.29.4 Endianness	1995
16.29.5 Cache Attributes	1995
16.29.6 Error Handling	1995
16.29.7 Interrupts	1995
16.29.8 DMA	1995
16.29.9 Summary	1996
16.30 Reference: A Quick Reference for Common MMIO Bugs	1996
16.30.1 1. Off-by-One in the Register Map	1996
16.30.2 2. Wrong Access Width	1996
16.30.3 3. Missing Volatile Qualifier in a Hand-Rolled Accessor	1996
16.30.4 4. Lost Update in Read-Modify-Write	1997
16.30.5 5. Missing Barrier Before a Doorbell	1997
16.30.6 6. Reading a Write-Only Register	1997
16.30.7 7. Unexpected Side Effect on Read	1997
16.30.8 8. Dangling Tag or Handle	1997
16.30.9 9. Sysctl Handler Without Lock	1998
16.30.10 0. Detach Races	1998
17 Simulating Hardware	1999
17.1 Reader Guidance & Outcomes	1999
17.1.1 What Chapter 17 Is About	1999
17.1.2 Why Autonomous Behaviour Deserves a Chapter of Its Own	2000
17.1.3 Where Chapter 16 Left the Driver	2001
17.1.4 What You Will Learn	2001
17.1.5 What This Chapter Does Not Cover	2002
17.1.6 Estimated Time Investment	2002
17.1.7 Prerequisites	2003
17.1.8 How to Get the Most Out of This Chapter	2003
17.1.9 Roadmap Through the Chapter	2003
17.2 Section 1: Why Simulate Hardware?	2004
17.2.1 The Working Driver Author's Toolkit	2004
17.2.2 Simulation Goals: Behaviour, Timing, Side Effects	2005
17.2.3 What a Good Simulation Teaches and What It Does Not	2006
17.2.4 No Hardware Dependency: The Practical Benefit	2006
17.2.5 Full Control: The Pedagogical Benefit	2006
17.2.6 Safe Experiments: The Risk-Reduction Benefit	2007
17.2.7 What We Will Simulate in Chapter 17	2007
17.2.8 The Simulation's Place in the Book	2008

17.2.9	Wrapping Up Section 1	2008
17.3	Section 2: Designing the Register Map	2009
17.3.1	What Is a Register Map?	2009
17.3.2	The Chapter 16 Map as a Starting Point	2009
17.3.3	Register Access Semantics	2010
17.3.4	Chapter 17 Additions	2011
17.3.5	Header Additions for the New Registers	2012
17.3.6	Writing a Register Map Table for Your Own Device	2013
17.3.7	Example Layout and Design Choices	2013
17.3.8	Naming Conventions for Constants	2014
17.3.9	Validating the Map Before Writing Code	2015
17.3.10	Documenting the Map	2015
17.3.11	Common Mistakes in Register Map Design	2016
17.3.12	Wrapping Up Section 2	2017
17.4	Section 3: Implementing the Simulated Hardware Backend	2017
17.4.1	The First Simulation File	2017
17.4.2	The Simulation State Structure	2019
17.4.3	The Sensor Callout: The First Autonomous Update	2020
17.4.4	Initialising the Sensor Callout	2021
17.4.5	Enabling and Disabling the Simulation	2022
17.4.6	Wiring the Simulation into Attach and Detach	2024
17.4.7	A First Test: The Sensor Updates Visibly	2026
17.4.8	A Second Callout: The Command Completion	2027
17.4.9	Intercepting CTRL.GO	2029
17.4.10	A Second Test: The Command Path	2029
17.4.11	Protecting Against the Overlapping Command	2030
17.4.12	Sysctls for the Simulation Controls	2030
17.4.13	What Stage 1 Accomplished	2031
17.4.14	What Stage 1 Is Not	2032
17.4.15	Wrapping Up Section 3	2032
17.5	Section 4: Exposing the Fake Device to the Driver	2033
17.5.1	The Tag and the Handle, Still	2033
17.5.2	Why the Abstraction Matters More Now	2033
17.5.3	The Command-Path Integration	2034
17.5.4	Where the Command Integrates with the Write Syscall	2037
17.5.5	Stage 2 Test	2038
17.5.6	What Does Not Change	2038
17.5.7	What Still Needs Attention	2039
17.5.8	Wrapping Up Section 4	2039
17.6	Section 5: Testing Device Behavior from the Driver	2039
17.6.1	Configurable Timeouts	2040
17.6.2	Read-Path Integration	2041
17.6.3	A First Load Test	2042
17.6.4	Error-Recovery Refinement	2043
17.6.5	Stats Infrastructure	2045
17.6.6	Observing the Driver Under Load	2047
17.6.7	Hands-On Behaviour Test	2047
17.6.8	What Stage 2 Accomplished	2048
17.6.9	Wrapping Up Section 5	2049
17.7	Section 6: Adding Timing and Delay	2049
17.7.1	The Three Timing Primitives	2049

17.7.2	A Time-Scale Decision Table	2050
17.7.3	Where the Chapter 17 Driver Already Uses Each	2050
17.7.4	A Good Use of <code>DELAY</code>	2050
17.7.5	A Good Use of <code>pause_sbt</code> (With the Lock Dropped)	2051
17.7.6	A Good Use of <code>callout_reset_sbt</code>	2052
17.7.7	A Realistic Example: Simulating a Slow Device	2053
17.7.8	Timing for the Simulation Itself	2054
17.7.9	Realistic Constraints: Ready After 500 ms	2054
17.7.10	Handling Timeouts Gracefully	2055
17.7.11	A Debugging Technique: Artificial Slowdown	2056
17.7.12	Common Timing Mistakes	2056
17.7.13	Wrapping Up Section 6	2057
17.8	Section 7: Simulating Errors and Fault Conditions	2057
17.8.1	The Fault-Injection Philosophy	2057
17.8.2	The Fault Modes	2058
17.8.3	The Fault-Probability Field	2059
17.8.4	Implementing the Timeout Fault	2059
17.8.5	Implementing the Error Fault	2061
17.8.6	Implementing the Stuck-Busy Fault	2062
17.8.7	The Driver's Error-Handling Paths	2064
17.8.8	Testing the Faults One at a Time	2065
17.8.9	Integration: Random Faults Under Load	2066
17.8.10	A Common Pitfall: Recovery That Never Clears Everything	2067
17.8.11	Combining Faults	2068
17.8.12	Observability for Fault Testing	2069
17.8.13	Safe Fault Injection	2070
17.8.14	Hands-On: The Fault-Injection Checklist	2071
17.8.15	A Reflection: What Fault Injection Taught Us	2071
17.8.16	Wrapping Up Section 7	2071
17.9	Section 8: Refactoring and Versioning Your Simulated Hardware Driver	2072
17.9.1	The Final File Layout	2072
17.9.2	The Simulation-Specific Fields Move into the <code>Sim Struct</code>	2072
17.9.3	The <code>SIMULATION.md</code> Document	2073
17.9.4	The Updated <code>HARDWARE.md</code>	2074
17.9.5	The Updated <code>LOCKING.md</code>	2074
17.9.6	The Version Bump	2075
17.9.7	The Regression Pass	2075
17.9.8	Running the Final Stage	2076
17.9.9	A Small Rule for Chapter 17's Refactor	2076
17.9.10	What Stage 5 Accomplished	2077
17.9.11	Real FreeBSD Drivers That Use the Same Patterns	2077
17.9.12	Wrapping Up Section 8	2078
17.10	Hands-On Labs	2078
17.10.1	Lab 1: Watch the Sensor Breathe	2078
17.10.2	Lab 2: Run a Single Command	2079
17.10.3	Lab 3: Stress the Command Path	2080
17.10.4	Lab 4: Inject a Timeout Fault	2080
17.10.5	Lab 5: Inject an Error Fault	2081
17.10.6	Lab 6: Inject Stuck-Busy and Watch the Driver Wait	2081
17.10.7	Lab 7: Mixed Faults Under Load	2082
17.10.8	Lab 8: Observe Sensor Updates During Heavy Command Load	2083

17.10.9 Lab 9: Build and Run the hwsim2 Module	2083
17.10.10 Lab 10: Inject a Memory-Corruption Attack (Debug Kernel)	2084
17.11 Challenge Exercises	2084
17.11.1 Challenge 1: Read-to-Clear INTR_STATUS	2084
17.11.2 Challenge 2: Add a Queue of Commands	2084
17.11.3 Challenge 3: Simulate a Sampling Rate Drift	2085
17.11.4 Challenge 4: A Write-One-to-Clear INTR_STATUS	2085
17.11.5 Challenge 5: Error Recovery Without Reset	2085
17.11.6 Challenge 6: A User-Space Debugger	2085
17.11.7 Challenge 7: A DTrace-Driven Fault Injection	2085
17.11.8 Challenge 8: Simulate Two Devices	2086
17.12 Troubleshooting Reference	2086
17.12.1 The sensor does not change	2086
17.12.2 Commands always time out	2086
17.12.3 WITNESS warns about lock order	2086
17.12.4 Kernel panic on kldunload	2086
17.12.5 Memory usage grows over time	2087
17.12.6 The access log shows unexpected entries	2087
17.12.7 Fault injection probability does not match observations	2087
17.12.8 sim_running toggles ignored	2087
17.12.9 Commands succeed but take much longer than expected	2087
17.12.10 Load tests are slower than expected	2087
17.13 Wrapping Up	2087
17.13.1 A Reflection Before Chapter 18	2088
17.13.2 What to Do If You Are Stuck	2088
17.14 Bridge to Chapter 18	2089
17.15 Reference: The Chapter 17 Simulation Cheat Sheet	2090
17.15.1 Simulation API (in myfirst_sim.h)	2090
17.15.2 Register Additions	2090
17.15.3 CTRL Additions	2090
17.15.4 Fault Modes	2091
17.15.5 New Sysctls	2091
17.16 Reference: Timing Primitive Cheat Sheet	2091
17.17 Reference: Anatomy of a Simulation Callout	2092
17.17.1 Step 1: Declare the callout in the sim state	2092
17.17.2 Step 2: Initialise the callout at attach	2092
17.17.3 Step 3: Write the callback	2092
17.17.4 Step 4: Start the callout at enable	2093
17.17.5 Step 5: Stop it at disable, drain at detach	2093
17.17.6 Step 6: Document in SIMULATION.md	2093
17.18 Reference: Test Scripts Inventory	2094
17.19 Reference: An Honest Accounting of Chapter 17's Simplifications	2094
17.19.1 The Callout's Mutex Interaction	2094
17.19.2 The Fault-Injection Framework	2095
17.19.3 The Random Number Source	2095
17.19.4 The Command Queue	2095
17.19.5 The Sensor Model	2095
17.19.6 The Error Taxonomy	2096
17.19.7 The Lack of Interrupts	2096
17.19.8 The Timing Precision	2096
17.19.9 Summary	2096

17.20	Reference: A Glossary of Chapter 17 Terms	2096
17.21	Reference: The Chapter 17 Driver Diff Summary	2097
17.21.1	New files	2097
17.21.2	Modified files	2098
17.21.3	Line count evolution	2098
17.21.4	Behavioural additions	2098
17.21.5	Test additions	2098
17.22	Reference: Reading a Callout-Driven Driver	2099
17.22.1	The LED Driver's Structure	2099
17.22.2	The Callout Callback	2099
17.22.3	The Pattern	2100
17.22.4	Lessons for Chapter 17	2100
17.22.5	An Exercise	2101
17.23	Reference: The Difference Between "In Simulation" and "In Reality"	2101
17.23.1	What the simulation and reality have in common	2101
17.23.2	What differs	2101
17.23.3	The Chapter 18 Change	2102
17.24	Reference: When Simulation Is Not Enough	2102
17.24.1	When the bug is in the real-silicon timing	2102
17.24.2	When the bug is in the platform-specific code	2102
17.24.3	When the bug depends on actual hardware resources	2102
17.24.4	When the bug is in the test itself	2102
17.24.5	What to do about it	2103
17.25	Reference: Further Reading	2103
17.25.1	FreeBSD manual pages	2103
17.25.2	Source files worth reading	2103
17.25.3	Real drivers with callouts similar to Chapter 17's	2103
17.25.4	Related reading on hardware simulation	2104
17.26	Reference: A Worked Example: The Full myfirst_sim.h	2104
17.27	Reference: A Comparison with Chapter 16 Patterns	2106
17.28	Reference: A Closing Note on Simulation Philosophy	2107
18	Writing a PCI Driver	2108
18.1	Reader Guidance & Outcomes	2108
18.1.1	Why the PCI Subsystem Earns a Chapter of Its Own	2109
18.1.2	Where Chapter 17 Left the Driver	2109
18.1.3	What You Will Learn	2110
18.1.4	What This Chapter Does Not Cover	2111
18.1.5	Estimated Time Investment	2112
18.1.6	Prerequisites	2112
18.1.7	How to Get the Most Out of This Chapter	2112
18.1.8	Roadmap Through the Chapter	2113
18.2	Section 1: What Is PCI and Why It Matters	2114
18.2.1	A Short History, and Why It Matters	2114
18.2.2	Where PCI Devices Live on a Modern Machine	2115
18.2.3	The Bus-Device-Function Tuple	2115
18.2.4	Configuration Space and What Lives There	2116
18.2.5	Vendor ID and Device ID: How Matches Happen	2117
18.2.6	How FreeBSD Enumerates PCI Devices	2117
18.2.7	The pci(4) Subsystem from the Driver's Perspective	2118
18.2.8	Common PCI Devices in the Real World	2119
18.2.9	Simulated PCI Devices: bhyve and QEMU	2119

18.2.10	Placing Chapter 18 in the Driver's Evolution	2120
18.2.11	Exercise: Read Your Own PCI Topology	2120
18.2.12	Wrapping Up Section 1	2122
18.3	Section 2: Probing and Attaching a PCI Device	2122
18.3.1	The Probe-Attach-Detach Contract	2122
18.3.2	The Device Method Table	2123
18.3.3	The Driver Structure	2124
18.3.4	DRIVER_MODULE and MODULE_DEPEND	2124
18.3.5	Matching by Vendor and Device ID	2125
18.3.6	The Probe Priority Tiers	2126
18.3.7	Writing a Minimal PCI Driver	2126
18.3.8	What Happens When the Driver Loads	2129
18.3.9	device_printf and Why It Matters	2130
18.3.10	The Softc and device_get_softc	2131
18.3.11	Probe-Attach-Detach Order in the Face of Unload	2131
18.3.12	Stage 1 Output: What Success Looks Like	2131
18.3.13	Common Mistakes at This Stage	2132
18.3.14	Checkpoint: Stage 1 Working	2133
18.3.15	Wrapping Up Section 2	2133
18.4	Section 3: Understanding and Claiming PCI Resources	2133
18.4.1	What a BAR Is, Precisely	2134
18.4.2	How a BAR Gets an Address	2134
18.4.3	The rid Argument and PCIR_BAR	2135
18.4.4	The Resource Type: SYS_RES_MEMORY vs SYS_RES_IOPORT	2135
18.4.5	The RF_ACTIVE Flag	2135
18.4.6	bus_alloc_resource_any in Detail	2136
18.4.7	From Resource to Tag and Handle	2136
18.4.8	rman_get_size and rman_get_start	2137
18.4.9	Releasing the BAR	2137
18.4.10	Partial Failure in Attach	2138
18.4.11	What Lives in the Softc	2139
18.4.12	The Stage 2 Attach	2139
18.4.13	The Stage 2 Detach	2140
18.4.14	What Stage 2 Does Not Yet Do	2140
18.4.15	Verifying Stage 2	2141
18.4.16	Common Mistakes in BAR Allocation	2141
18.4.17	A Note on 64-bit BARs	2142
18.4.18	Prefetchable vs Non-Prefetchable BARs	2142
18.4.19	Wrapping Up Section 3	2143
18.5	Section 4: Accessing Device Registers via bus_space(9)	2143
18.5.1	Revisiting the Chapter 16 Accessors	2143
18.5.2	The Two Origins of a Tag and Handle	2144
18.5.3	Extending myfirst_hw_attach	2145
18.5.4	The First Real Register Read	2147
18.5.5	The Full Accessor Family	2147
18.5.6	bus_space_read_multi vs bus_space_read_region	2148
18.5.7	When Barriers Matter	2148
18.5.8	The Access Log on a Real BAR	2149
18.5.9	A Small Subtlety: the CSR Macros Do Not Know About PCI	2149
18.5.10	Stage 2 to Stage 3: Wiring It Together	2150
18.5.11	Common Mistakes in the bus_space-on-PCI Transition	2151

18.5.12	Wrapping Up Section 4	2152
18.6	Section 5: Driver Attach-Time Initialisation	2152
18.6.1	The Attach-Time Checklist	2153
18.6.2	pci_enable_busmaster and the Command Register	2153
18.6.3	Reading Configuration-Space Fields	2154
18.6.4	pci_write_config and the Side-Effect Contract	2154
18.6.5	pci_find_cap: Walking the Capability List	2155
18.6.6	pci_find_extcap: PCIe Extended Capabilities	2156
18.6.7	PCIe AER: An Introduction	2156
18.6.8	Composing the Simulation with the Real PCI Backend	2157
18.6.9	Why the Simulation Callouts Stay Silent on Real PCI	2158
18.6.10	Creating the cdev in a PCI Driver	2158
18.6.11	A Full Stage 3 Attach	2159
18.6.12	Verifying Stage 3	2160
18.6.13	Wrapping Up Section 5	2161
18.7	Section 6: Supporting Detach and Resource Cleanup	2161
18.7.1	The Core Rule: Reverse Order	2162
18.7.2	The Chapter 18 Detach Ordering	2162
18.7.3	myfirst_is_busy: When to Refuse Detach	2163
18.7.4	Quiescing Callouts and Tasks	2164
18.7.5	Releasing the BAR After the Hardware Layer	2165
18.7.6	Failures During Detach	2165
18.7.7	Detach vs Module Unload vs Device Removal	2166
18.7.8	Partial-Attach Failure and the Detach Path	2166
18.7.9	A Concrete Walk Through the Cascade	2167
18.7.10	Detach vs Resume: A Preview	2167
18.7.11	Wrapping Up Section 6	2167
18.8	Section 7: Testing PCI Driver Behaviour	2168
18.8.1	The Test Environment	2168
18.8.2	Preparing the bhyve Guest	2168
18.8.3	Verifying the Guest Sees the Device	2169
18.8.4	Preparing the Guest for myfirst	2170
18.8.5	Loading myfirst and Watching the Attach	2171
18.8.6	Exercising the cdev	2172
18.8.7	Reading and Writing Configuration Space from User Space	2173
18.8.8	devinfo -v and What It Tells You	2174
18.8.9	dmesg as a Diagnostic Tool	2174
18.8.10	Using devctl to Simulate Hotplug	2175
18.8.11	A Simple Regression Script	2176
18.8.12	Reading Configuration Space for Diagnostic Purposes	2177
18.8.13	What Section 7 Does Not Test	2177
18.8.14	Wrapping Up Section 7	2178
18.9	Section 8: Refactoring and Versioning Your PCI Driver	2178
18.9.1	The Final File Layout	2178
18.9.2	The Final Makefile	2179
18.9.3	The Version String	2180
18.9.4	The PCI.md Document	2180
18.9.5	Updating LOCKING.md	2182
18.9.6	Updating HARDWARE.md	2182
18.9.7	The Regression Pass	2182
18.9.8	What the Refactor Accomplished	2183

18.9.9 A Short Note on Symbol Visibility	2183
18.9.10 Wrapping Up Section 8	2184
18.10 Reading a Real Driver Together: <code>uart_bus_pci.c</code>	2184
18.10.1 The Top of the File	2184
18.10.2 Method Table and Driver Structure	2185
18.10.3 The ID Table	2185
18.10.4 The Probe Routine	2186
18.10.5 The Attach Routine	2188
18.10.6 The Detach Routine	2188
18.10.7 The <code>DRIVER_MODULE</code> Line	2189
18.10.8 What This Walkthrough Teaches	2189
18.11 A Deeper Look at the PCI Capability List	2189
18.11.1 Structure of the Legacy Capability List	2190
18.11.2 Structure of the Extended Capability List	2190
18.11.3 Why the Walk Matters	2191
18.11.4 Walking a Specific Capability's Fields	2192
18.12 A Deeper Look at Configuration Space	2192
18.12.1 Configuration Space Layout	2192
18.12.2 Header Type	2193
18.12.3 Commands and Status	2194
18.12.4 Reading Wider Than Available Width	2194
18.12.5 Alignment in Configuration-Space Reads	2194
18.12.6 Writing Configuration Space: Caveats	2195
18.13 A Deeper Look at the <code>bus_space</code> Abstraction	2195
18.13.1 What <code>bus_space_tag_t</code> Is	2195
18.13.2 What <code>bus_space_handle_t</code> Is	2196
18.13.3 What Happens Inside <code>bus_space_read_4</code>	2196
18.13.4 Why the Abstraction Matters	2196
18.13.5 The <code>bus_read</code> vs <code>bus_space_read</code> Naming	2197
18.14 Hands-On Labs	2197
18.14.1 Lab 1: Explore Your PCI Topology	2197
18.14.2 Lab 2: Write the Probe Skeleton on Paper	2198
18.14.3 Lab 3: Load the Stage 1 Driver in a bhyve Guest	2199
18.14.4 Lab 4: Claim the BAR and Read a Register	2200
18.14.5 Lab 5: Exercise the <code>cdev</code> and Verify Detach Cleanup	2200
18.14.6 Lab Summary	2201
18.15 Challenge Exercises	2201
18.15.1 Challenge 1: Support a Second Vendor and Device ID	2201
18.15.2 Challenge 2: Print the Full Capability Chain	2202
18.15.3 Challenge 3: Implement a Simple <code>ioctl</code> for Configuration-Space Access	2202
18.15.4 Challenge 4: Refuse to Attach if the BAR Is Too Small	2202
18.15.5 Challenge 5: Split the Driver into Two Modules	2202
18.15.6 Challenge 6: Reimplement probe Using Class and Subclass Matching	2203
18.15.7 Challenge 7: Add a Read-Only <code>sysctl</code> That Reports the Driver's PCI State	2203
18.15.8 Challenge 8: Simulate a Failed Attach	2203
18.15.9 Challenge Summary	2203
18.16 Troubleshooting and Common Mistakes	2203
18.16.1 "Driver does not attach; no <code>dmesg</code> banner"	2203
18.16.2 " <code>kldload</code> panics the kernel"	2204
18.16.3 "BAR allocation fails with <code>NULL</code> "	2204
18.16.4 " <code>CSR_READ_4</code> returns <code>0xffffffff</code> "	2205

18.16.5 “kldunload returns Device busy”	2205
18.16.6 “dmesg says ‘cleanup failed in detach’ ”	2205
18.16.7 “ioctl or read returns an unexpected error”	2205
18.16.8 “vmstat -m shows leaked allocations”	2206
18.16.9 “pci_find_cap returns ENOENT for a capability I know the device has”	2206
18.16.10 “Module unloads, but dmesg shows a page fault during unload”	2206
18.16.11 “devinfo -v shows the driver attached but the cdev does not appear”	2207
18.16.12 “Attach takes a long time to return”	2207
18.16.13 “Driver attaches on boot but never produces output for the first few seconds”	2207
18.16.14 “A second attempt to attach after a failed first attempt succeeds”	2208
18.16.15 “vmstat -m InUse grows after each load-unload cycle”	2208
18.16.16 Troubleshooting Summary	2208
18.16.17 A Debug Kernel Checklist	2209
18.17 Wrapping Up	2209
18.17.1 A Reflection Before Chapter 19	2210
18.17.2 What to Do If You Are Stuck	2210
18.18 Bridge to Chapter 19	2211
18.19 Reference: PCI Header Offsets the Chapter Used	2212
18.19.1 Command Register Bits	2213
18.19.2 Capability IDs (legacy)	2213
18.19.3 Extended Capability IDs (PCIe)	2214
18.20 Reference: A Comparison with Chapter 16 and Chapter 17 Patterns	2214
18.21 Reference: Patterns From Real FreeBSD PCI Drivers	2215
18.21.1 Pattern: Walking BARs by Type	2215
18.21.2 Pattern: Matching on Class, Subclass, and progIF	2215
18.21.3 Pattern: Conditional MSI Allocation	2216
18.21.4 Pattern: Detach-Time IRQ Release	2216
18.21.5 Pattern: Reading Vendor-Specific Configuration Fields	2217
18.21.6 Pattern: A Power-Aware Resume Handler	2218
18.21.7 Pattern: Responding to a Device-Specific Error State	2218
18.21.8 Pattern: Using sysctl to Expose Driver Configuration	2219
18.21.9 Pattern: Recording Supported Features in a Capabilities Structure	2219
18.21.10 Pattern Summary	2219
18.22 Reference: A Closing Note on PCI Driver Philosophy	2220
18.23 Reference: Chapter 18 Quick-Reference Card	2220
18.23.1 Vocabulary	2220
18.23.2 Essential APIs	2221
18.23.3 Essential Macros	2221
18.23.4 Common Procedures	2221
18.23.5 Useful Commands	2222
18.23.6 Files to Keep Bookmarked	2222
18.24 Reference: Glossary of Chapter 18 Terms	2223
19 Handling Interrupts	2225
19.1 Reader Guidance & Outcomes	2225
19.1.1 Why Interrupt Handling Earns a Chapter of Its Own	2226
19.1.2 Where Chapter 18 Left the Driver	2226
19.1.3 What You Will Learn	2227
19.1.4 What This Chapter Does Not Cover	2228
19.1.5 Estimated Time Investment	2228
19.1.6 Prerequisites	2229
19.1.7 How to Get the Most Out of This Chapter	2229

19.1.8	Roadmap Through the Chapter	2230
19.2	Section 1: What Are Interrupts?	2230
19.2.1	The Problem Interrupts Solve	2231
19.2.2	What a Hardware Interrupt Actually Is	2231
19.2.3	Edge-Triggered vs Level-Triggered	2232
19.2.4	The CPU's Interrupt Handling Flow, Simplified	2233
19.2.5	What a Driver Must Do When an Interrupt Arrives	2234
19.2.6	Real-World Examples	2234
19.2.7	A Quick Exercise: Find Interrupt-Driven Devices on Your Lab Host	2235
19.2.8	A Brief History of Interrupts	2236
19.2.9	Wrapping Up Section 1	2236
19.3	Section 2: Interrupts in FreeBSD	2236
19.3.1	The FreeBSD Interrupt Model in One Picture	2237
19.3.2	IRQ Assignment and Routing	2237
19.3.3	SYS_RES_IRQ: The Interrupt Resource	2238
19.3.4	Filter Handlers vs ithread Handlers	2238
19.3.5	When to Use Filter Only	2239
19.3.6	When to Use Filter Plus ithread	2239
19.3.7	When to Use ithread Only	2240
19.3.8	INTR_MPSAFE: What the Flag Promises	2240
19.3.9	INTR_TYPE_* Flags	2240
19.3.10	Shared vs Exclusive Interrupts	2241
19.3.11	What vmstat -i Shows	2241
19.3.12	What devinfo -v Shows About Interrupts	2242
19.3.13	A Simple Interrupt Path Diagram	2242
19.3.14	Constraints on What Handlers May Do	2243
19.3.15	A Note on Per-CPU vs Shared ithreads	2244
19.3.16	Wrapping Up Section 2	2244
19.4	Section 3: Registering an Interrupt Handler	2245
19.4.1	What Stage 1 Produces	2245
19.4.2	The IRQ Resource Allocation	2245
19.4.3	Storing the Resource in the Softc	2246
19.4.4	The Filter Handler's Signature	2247
19.4.5	Registering the Handler With bus_setup_intr	2247
19.4.6	Describing the Handler to devinfo	2248
19.4.7	The Extended Attach Cascade	2248
19.4.8	The Extended Detach	2251
19.4.9	The sysctl for the Interrupt Counter	2252
19.4.10	What Stage 1 Proves	2253
19.4.11	What Stage 1 Does Not Do	2253
19.4.12	Common Mistakes at This Stage	2253
19.4.13	Checkpoint: Stage 1 Working	2254
19.4.14	Wrapping Up Section 3	2254
19.5	Section 4: Writing a Real Interrupt Handler	2255
19.5.1	The Register Picture	2255
19.5.2	The Filter Handler at Stage 2	2255
19.5.3	Why a Taskqueue, Not an ithread?	2257
19.5.4	The Deferred-Work Task	2258
19.5.5	Declaring and Initialising the Task	2259
19.5.6	Enabling the Interrupts at the Device	2260
19.5.7	Disabling the Interrupts at Detach	2260

19.5.8	A Worked Flow	2261
19.5.9	Urgent Work Inline vs Deferred	2261
19.5.10	FILTER_SCHEDULE_THREAD vs Taskqueue	2261
19.5.11	When the Taskqueue Itself Is the Wrong Answer	2262
19.5.12	Common Mistakes at This Stage	2262
19.5.13	Stage 2 Output: What Success Looks Like	2263
19.5.14	Wrapping Up Section 4	2263
19.6	Section 5: Using Simulated Interrupts for Testing	2263
19.6.1	Why Simulation Deserves a Section	2264
19.6.2	The Simulated-Interrupt sysctl	2264
19.6.3	Exercising the Simulation	2265
19.6.4	Why This Is Not a Toy	2266
19.6.5	Locking in the Simulated Path	2267
19.6.6	Using the Chapter 17 Simulation to Produce Interrupts	2267
19.6.7	Extending the sysctl to Schedule at a Rate	2267
19.6.8	What the Simulation Does Not Capture	2268
19.6.9	Observing the Task Running	2269
19.6.10	Integration With Chapter 17's Fault Framework	2269
19.6.11	Wrapping Up Section 5	2269
19.7	Section 6: Handling Shared Interrupts	2270
19.7.1	Why Share an IRQ at All?	2270
19.7.2	The Flow on a Shared IRQ	2270
19.7.3	The INTR_STATUS Test	2270
19.7.4	What "Is It Mine?" Looks Like on Real Hardware	2271
19.7.5	Returning FILTER_STRAY Correctly	2271
19.7.6	What Happens If Multiple Drivers Share an IRQ	2272
19.7.7	What Happens If a Driver Gets It Wrong	2273
19.7.8	Coexisting With Non-Sharing Drivers	2273
19.7.9	The bhyve Virtio IRQ Topology	2273
19.7.10	The Starvation Concern	2274
19.7.11	False Positives and Defensive Handling	2274
19.7.12	When the Filter Should Handle and the Task Should Not Run	2274
19.7.13	Wrapping Up Section 6	2275
19.8	Section 7: Cleaning Up Interrupt Resources	2275
19.8.1	The Required Order	2275
19.8.2	Why Disable at the Device Before bus_teardown_intr	2276
19.8.3	Why bus_teardown_intr Before Releasing the Resource	2276
19.8.4	Why Drain the Taskqueue Before Freeing the Softc	2276
19.8.5	Why bus_teardown_intr Before taskqueue_drain	2277
19.8.6	The Cleanup Code	2277
19.8.7	Handling Partial-Attach Failure	2278
19.8.8	Verifying the Teardown	2279
19.8.9	Handling the "Handler Fired During Detach" Case	2279
19.8.10	What Goes Wrong When Teardown Is Skipped	2280
19.8.11	Sanity Testing the Teardown	2281
19.8.12	Wrapping Up Section 7	2281
19.9	Section 8: Refactoring and Versioning Your Interrupt-Ready Driver	2281
19.9.1	The Final File Layout	2282
19.9.2	The Final Makefile	2282
19.9.3	The Version String	2283
19.9.4	The myfirst_intr.h Header	2283

19.9.5	The myfirst_intr.c File	2284
19.9.6	The Refactored PCI Attach	2287
19.9.7	The Refactored Detach	2288
19.9.8	The INTERRUPTS.md Document	2289
19.9.9	The Regression Pass	2291
19.9.10	What the Refactor Accomplished	2291
19.9.11	Wrapping Up Section 8	2291
19.10	Reading a Real Driver Together: mgb(4)'s Interrupt Path	2292
19.10.1	The Legacy Filter	2292
19.10.2	What the mgb Filter Teaches	2293
19.10.3	Interrupt Setup in mgb	2294
19.10.4	Interrupt Teardown in mgb	2294
19.10.5	What the Walkthrough Teaches	2294
19.11	A Deeper Look at Interrupt Context	2294
19.11.1	The Stack Situation	2294
19.11.2	Why No Sleep Locks	2295
19.11.3	Why No Sleeping malloc	2295
19.11.4	Why No Condition Variables	2295
19.11.5	Why the Filter Cannot Re-enter Itself	2296
19.11.6	Why Atomic Operations Are Safe	2296
19.11.7	Why the ithread Gets More Rope	2296
19.11.8	Why taskqueue Threads Have Still More Rope	2296
19.11.9	How Interrupt Context Interacts With Locking	2297
19.11.10	The "Giant" Footnote	2297
19.12	A Deeper Look at CPU Affinity for Interrupts	2297
19.12.1	What Affinity Means	2297
19.12.2	Default Behaviour	2298
19.12.3	When Affinity Matters	2298
19.12.4	The bus_bind_intr API	2298
19.12.5	The Kernel's CPU-Set Abstraction	2299
19.12.6	Observing Affinity	2299
19.13	A Deeper Look at Interrupt Storm Detection	2299
19.13.1	The hw.intr_storm_threshold Sysctl	2299
19.13.2	When to Enable Detection	2299
19.13.3	What a Storm Looks Like	2300
19.13.4	Driver-Side Storm Mitigation	2300
19.13.5	The Relationship to Shared IRQs	2300
19.14	A Mental Model for Filter vs ithread Selection	2300
19.14.1	Four Questions	2301
19.14.2	Applying the Framework	2301
19.14.3	Worked Example: A Hypothetical Storage Driver	2301
19.14.4	Worked Example: A Hypothetical Network Driver	2302
19.14.5	Worked Example: The Chapter 19 Driver	2302
19.14.6	When to Revisit the Decision	2302
19.15	Lock Ordering and the Interrupt Path	2302
19.15.1	The Filter Takes No Locks	2302
19.15.2	The Task Acquires sc->mtx	2303
19.15.3	The Simulated-Interrupt sysctl Acquires and Releases sc->mtx	2303
19.15.4	The Attach and Detach Paths	2303
19.15.5	A Subtle Ordering Concern: bus_teardown_intr Can Block	2303
19.15.6	WITNESS and the Interrupt Path	2303

19.15.7 What to Document in LOCKING.md	2303
19.16 Observability: What Chapter 19 Exposes to Operators	2304
19.16.1 The Counter Suite	2304
19.16.2 The Writable Sysctls	2304
19.16.3 The Kernel-Level View	2304
19.16.4 Correlating the Views	2304
19.16.5 DTrace	2305
19.16.6 ktrace and kgdb	2305
19.16.7 Lab 1: Explore Interrupt Sources on Your System	2306
19.16.8 Lab 2: Stage 1, Register and Fire the Handler	2306
19.16.9 Lab 3: Stage 2, Real Filter and Deferred Task	2307
19.16.10 Lab 4: Stage 3, Simulated Interrupts via sysctl	2308
19.16.11 Lab 5: Stage 4, Refactor, Regression, Version	2309
19.17 Challenge Exercises	2309
19.17.1 Challenge 1: Add a Filter-Plus-ithread Handler	2309
19.17.2 Challenge 2: Implement Driver-Side Storm Mitigation	2310
19.17.3 Challenge 3: Bind the Interrupt to a Specific CPU	2310
19.17.4 Challenge 4: Extend the Simulated Interrupts With Per-Type Rates	2310
19.17.5 Challenge 5: Add a Rate-Limited Stray Log	2310
19.17.6 Challenge 6: Implement MSI Allocation (Preview of Chapter 20)	2310
19.17.7 Challenge 7: Write a Latency Test	2310
19.17.8 Challenge 8: Share the IRQ Deliberately	2311
19.18 Troubleshooting and Common Mistakes	2311
19.18.1 “Driver loads but no interrupts are counted”	2311
19.18.2 “GIANT-LOCKED warning in dmesg”	2311
19.18.3 “Kernel panic in the filter”	2311
19.18.4 “Task runs but accesses freed state”	2312
19.18.5 “INTR_STATUS bits keep firing; storm detected”	2312
19.18.6 “Simulated interrupt does not run the task”	2312
19.18.7 “kldunload fails with Device busy”	2312
19.18.8 “vmstat -m shows live allocations after unload”	2312
19.18.9 “Handler fires on the wrong CPU”	2313
19.18.10 “INTR_MASK write has unintended side effects”	2313
19.18.11 “Stray interrupt messages in dmesg”	2313
19.18.12 “Handler is called concurrently on multiple CPUs”	2313
19.18.13 “bus_setup_intr returns EINVAL”	2314
19.18.14 “bus_setup_intr returns EEXIST”	2314
19.18.15 “Debug kernel panics on taskqueue_drain”	2314
19.18.16 “Filter is called but INTR_STATUS reads as 0xffffffff”	2314
19.18.17 “Interrupts are counted but the device makes no progress”	2315
19.18.18 “kldunload hangs”	2315
19.18.19 “Unaligned memory access in the filter”	2315
19.18.20 “device_printf from the filter slows the system”	2316
19.18.21 “Driver passes all tests but misbehaves under load”	2316
19.18.22 “vmstat -i shows many strays on a line I don’t own”	2316
19.19 Advanced Observability: Integrating With DTrace	2316
19.19.1 Counting Filter Invocations	2317
19.19.2 Measuring Filter Latency	2317
19.19.3 Observing the Task Queue	2317
19.19.4 Correlating the Filter and Task	2317
19.19.5 Tracing Taskqueue Scheduling Decisions	2318

19.19.6 DTrace and the Simulated-Interrupt Path	2318
19.20 Detailed Walkthrough: Stage 2 End to End	2319
19.20.1 The Sequence	2319
19.20.2 What the Counters Show	2319
19.20.3 What dmesg Shows	2320
19.20.4 What vmstat -i Shows	2320
19.20.5 Tracing With Print Statements	2320
19.21 Patterns From Real FreeBSD Drivers	2320
19.21.1 Pattern: Fast-Filter-With-Slow-Task	2321
19.21.2 Pattern: ithread-Only Handler	2321
19.21.3 Pattern: INTR_EXCL for Exclusive Access	2322
19.21.4 Pattern: Short Debug Log	2322
19.21.5 Pattern: Bind to a Specific CPU	2322
19.21.6 Pattern: Describe the Handler for Diagnostics	2322
19.21.7 Pattern: Quiesce Before Detach	2323
19.22 Reference: Common Mistakes Cheat Sheet	2323
19.23 Reference: ithread and Taskqueue Priorities	2324
19.24 Reference: A Short Tour of /usr/src/sys/kern/kern_intr.c	2324
19.24.1 Key Functions in kern_intr.c	2325
19.25 Wrapping Up	2325
19.25.1 A Reflection Before Chapter 20	2326
19.25.2 What to Do If You Are Stuck	2326
19.26 Bridge to Chapter 20	2327
19.27 Reference: Chapter 19 Quick-Reference Card	2328
19.27.1 Vocabulary	2328
19.27.2 Essential APIs	2328
19.27.3 Essential Macros	2329
19.27.4 Common Procedures	2329
19.27.5 Useful Commands	2329
19.27.6 Files to Keep Bookmarked	2329
19.28 Reference: A Comparison Table Across Part 4	2330
19.29 Reference: FreeBSD Manual Pages for Chapter 19	2330
19.29.1 Kernel API Manual Pages	2330
19.29.2 Device Subsystem Manual Pages	2331
19.30 Reference: Driver Memorable Phrases	2331
19.31 Reference: Glossary of Chapter 19 Terms	2331
19.32 Reference: A Closing Note on Interrupt-Handling Philosophy	2332
20 Advanced Interrupt Handling	2334
20.1 Reader Guidance & Outcomes	2334
20.1.1 Why MSI-X Earns a Chapter of Its Own	2335
20.1.2 Where Chapter 19 Left the Driver	2335
20.1.3 What You Will Learn	2336
20.1.4 What This Chapter Does Not Cover	2337
20.1.5 Estimated Time Investment	2337
20.1.6 Prerequisites	2338
20.1.7 How to Get the Most Out of This Chapter	2338
20.1.8 Roadmap Through the Chapter	2339
20.2 Section 1: What Are MSI and MSI-X?	2339
20.2.1 The Problem With Legacy INTx	2339
20.2.2 How an MSI-Triggered Interrupt Actually Happens	2340
20.2.3 MSI: The Simpler of the Two	2341

20.2.4	MSI-X: The Fuller Mechanism	2341
20.2.5	How FreeBSD Abstracts the Difference	2342
20.2.6	The Fallback Ladder	2342
20.2.7	Real-World Examples	2343
20.2.8	Why MSI-X and Not MSI	2343
20.2.9	A Diagram of the MSI-X Flow	2344
20.2.10	Exercise: Finding MSI-Capable Devices on Your System	2344
20.2.11	Wrapping Up Section 1	2345
20.3	Section 2: Enabling MSI in Your Driver	2345
20.3.1	What Stage 1 Produces	2346
20.3.2	The MSI Count Query	2346
20.3.3	The MSI Allocation Call	2346
20.3.4	The Per-Vector Resource Allocation	2347
20.3.5	The Filter Handler on an MSI Vector	2347
20.3.6	Building the Fallback	2348
20.3.7	The <code>intr_mode</code> Field and the <code>dmesg</code> Summary	2350
20.3.8	What the Teardown Needs to Change	2351
20.3.9	Verifying Stage 1	2352
20.3.10	What Stage 1 Does Not Do	2353
20.3.11	Common Mistakes at This Stage	2353
20.3.12	Checkpoint: Stage 1 Working	2353
20.3.13	Wrapping Up Section 2	2354
20.4	Section 3: Managing Multiple Interrupt Vectors	2354
20.4.1	The Per-Vector Design	2354
20.4.2	The Filter Functions	2356
20.4.3	The Per-Vector Task	2358
20.4.4	Allocating Multiple Vectors	2358
20.4.5	The Full Teardown Path	2362
20.4.6	Simulating Interrupts Per Vector	2363
20.4.7	What Happens on Legacy Fallback	2364
20.4.8	The Stage 2 <code>dmesg</code> Banner	2365
20.4.9	Common Mistakes in Stage 2	2365
20.4.10	Wrapping Up Section 3	2366
20.5	Section 4: Designing Interrupt-Safe Data Structures	2366
20.5.1	The New Concurrency Picture	2366
20.5.2	Per-Vector State: The Default	2367
20.5.3	Shared Counters: Atomic Operations	2367
20.5.4	Shared Device State: The Mutex Discipline	2367
20.5.5	The Per-Vector Lock: When You Need It	2368
20.5.6	Per-CPU Data: The Advanced Option	2368
20.5.7	Lock Ordering and Multi-Vector Complications	2368
20.5.8	The Memory Model: Why Atomics Matter	2369
20.5.9	Observability: Per-Vector Counters in <code>sysctl</code>	2369
20.5.10	Wrapping Up Section 4	2370
20.6	Section 5: Using MSI-X for High Flexibility	2370
20.6.1	The MSI-X Count and Allocation API	2370
20.6.2	The Fallback Ladder Extended	2371
20.6.3	Vector Binding With <code>bus_bind_intr</code>	2372
20.6.4	The MSI-X <code>dmesg</code> Summary	2373
20.6.5	Per-Vector <code>bus_describe_intr</code>	2374
20.6.6	MSI-X Table and BAR Considerations	2374

20.6.7	Allocating Fewer Vectors Than Requested	2375
20.6.8	Testing MSI-X on bhyve vs QEMU	2375
20.6.9	Common Mistakes in MSI-X Setup	2376
20.6.10	Wrapping Up Section 5	2376
20.7	Section 6: Handling Vector-Specific Events	2376
20.7.1	The Admin Vector	2377
20.7.2	The RX Vector	2377
20.7.3	The TX Vector	2378
20.7.4	Vector-to-Event Mapping	2379
20.7.5	Simulated Per-Vector Events	2379
20.7.6	Dynamic Vector Assignment	2380
20.7.7	A Pattern From nvme(4)	2381
20.7.8	Observability: Per-Vector Rate	2381
20.7.9	Wrapping Up Section 6	2382
20.8	Section 7: Teardown and Cleanup With MSI/MSI-X	2382
20.8.1	The Required Order	2382
20.8.2	Why Reverse Order Per Vector	2382
20.8.3	The Per-Vector Teardown Code	2383
20.8.4	The Full Teardown	2383
20.8.5	Partial-Attach Failure Unwind	2384
20.8.6	Observability: Verifying a Clean Teardown	2385
20.8.7	MSI Resource Leak Across Load-Unload Cycles	2385
20.8.8	Common Mistakes in Teardown	2385
20.8.9	Wrapping Up Section 7	2386
20.9	Section 8: Refactoring and Versioning Your Multi-Vector Driver	2386
20.9.1	The Final File Layout	2386
20.9.2	The myfirst_msix.h Header	2387
20.9.3	The Full Makefile	2388
20.9.4	The Version String	2388
20.9.5	The MSIX.md Document	2388
20.9.6	The Regression Pass	2390
20.9.7	What the Refactor Accomplished	2391
20.9.8	Wrapping Up Section 8	2391
20.10	Reading a Real Driver Together: virtio_pci.c	2391
20.10.1	The Allocation Ladder	2391
20.10.2	The MSI-X Allocation Helper	2392
20.10.3	The Per-Vector Resource Allocation	2393
20.10.4	The Teardown	2394
20.10.5	What the Virtio Walkthrough Teaches	2394
20.11	A Deeper Look at Vector-to-CPU Placement	2395
20.11.1	The NUMA Picture	2395
20.11.2	Querying NUMA Locality	2395
20.11.3	The Operator View	2397
20.11.4	Measuring the Effect	2397
20.11.5	When Vector Placement Matters	2397
20.11.6	When Vector Placement Doesn't Matter	2398
20.12	A Deeper Look at Vector Assignment Strategies	2398
20.12.1	One-Vector-Per-Queue	2398
20.12.2	Coalesced RX+TX Vector	2398
20.12.3	One Vector For All Queues	2399
20.12.4	Dynamic Per-CPU Assignment	2399

20.12.5 Chapter 20's Strategy	2399
20.13 A Deeper Look at Interrupt Moderation and Coalescing	2399
20.13.1 Why Moderation Matters	2399
20.13.2 How Drivers Control Moderation	2400
20.13.3 Chapter 20's Driver Does Not Moderate	2400
20.14 Patterns From Real FreeBSD Drivers	2400
20.14.1 Pattern: nvme(4) Admin + I/O Vector Split	2400
20.14.2 Pattern: ixgbe's Queue-Pair Vector	2401
20.14.3 Pattern: virtio_pci's Per-Virtqueue Vector	2401
20.14.4 Pattern: ahci's Per-Port Vector	2402
20.14.5 Pattern: iflib's Hidden Vector Management	2402
20.14.6 What the Patterns Teach	2403
20.15 A Performance Observation: Measuring the MSI-X Benefit	2403
20.15.1 The Test Setup	2403
20.15.2 Measuring With DTrace	2403
20.15.3 What the Results Show	2404
20.15.4 The General Lesson	2404
20.16 A Deeper Look at sysctl Tree Design for Multi-Vector Drivers	2404
20.16.1 The Flat vs Tree Trade-off	2405
20.16.2 The Tree-Building Code	2405
20.16.3 Querying the Tree	2406
20.16.4 When to Use the Tree	2407
20.16.5 Common Mistakes	2407
20.17 A Deeper Look at Error Paths in MSI-X Setup	2407
20.17.1 Failure Mode 1: pci_msix_count Returns 0	2407
20.17.2 Failure Mode 2: pci_alloc_msix Returns EINVAL	2407
20.17.3 Failure Mode 3: pci_alloc_msix Returns Fewer Vectors Than Requested	2407
20.17.4 Failure Mode 4: bus_alloc_resource_any Returns NULL for an MSI-X Vector	2408
20.17.5 Failure Mode 5: bus_setup_intr Returns EINVAL for a Per-Vector Handler	2408
20.17.6 Failure Mode 6: bus_bind_intr Returns an Error	2408
20.17.7 Failure Mode 7: vmstat -i Shows Vectors but Counters Don't Increment	2408
20.17.8 Failure Mode 8: Second kldload Falls Back to Lower Tier	2408
20.17.9 Failure Mode 9: WITNESS Panics on Multi-Vector Setup	2408
20.17.10 Failure Mode 10: Partial Setup Doesn't Clean Up Properly	2409
20.18 Additional Troubleshooting	2409
20.18.1 "QEMU guest does not expose MSI-X"	2409
20.18.2 "intr_simulate_rx increments fire_count but task never runs"	2409
20.18.3 "Per-vector counters increment but the stray count goes up proportionally"	2409
20.18.4 "cpuset -g -x \$irq reports mask 0 on all vectors"	2409
20.18.5 "Driver load succeeds but dmesg shows no attach banner"	2410
20.18.6 "Detach hangs after multi-vector setup"	2410
20.18.7 "pci_alloc_msix succeeds but only some vectors fire"	2410
20.19 Worked Example: Tracing an Event Through Three Tiers	2410
20.19.1 Tier 3: Legacy INTx	2410
20.19.2 Tier 2: MSI	2410
20.19.3 Tier 1: MSI-X	2411
20.19.4 The Lesson	2411
20.20 A Practical Lab: Regression Testing Across All Three Tiers	2411
20.20.1 Setup	2411
20.20.2 Procedure	2412
20.20.3 Expected Observations	2412

20.20.4 What to Do if a Tier Fails	2413
20.21 Extended Challenge: Building a Production-Quality Driver	2413
20.21.1 The Goal	2413
20.21.2 Implementation Outline	2413
20.21.3 Testing	2413
20.21.4 What This Exercises	2414
20.22 Reference: Priority Values for Interrupt and Task Work	2414
20.23 Reference: Useful DTrace One-Liners for MSI-X Drivers	2414
20.23.1 Count filter invocations per CPU	2414
20.23.2 Time spent in each filter	2415
20.23.3 Rate of simulated vs real interrupts	2415
20.23.4 Task latency	2415
20.24 Reference: A Closing Note Before Part 4's End	2415
20.25 Hands-On Labs	2416
20.25.1 Lab 1: Discover MSI and MSI-X Capabilities	2416
20.25.2 Lab 2: Stage 1, MSI Fallback Ladder	2416
20.25.3 Lab 3: Stage 2, Multi-Vector Allocation (MSI)	2417
20.25.4 Lab 4: Stage 3, MSI-X With CPU Binding	2418
20.25.5 Lab 5: Stage 4, Refactor, Regression, Version	2419
20.26 Challenge Exercises	2419
20.26.1 Challenge 1: Dynamic Vector-Count Adaptation	2419
20.26.2 Challenge 2: NUMA-Aware CPU Binding	2420
20.26.3 Challenge 3: Per-Vector Taskqueues	2420
20.26.4 Challenge 4: Per-Vector MSI-X Mask Control	2420
20.26.5 Challenge 5: Implement Interrupt Moderation	2420
20.26.6 Challenge 6: Vector Reassignment at Runtime	2420
20.26.7 Challenge 7: Per-Queue Sysctl Tree	2420
20.26.8 Challenge 8: Dtrace Instrumentation	2420
20.27 Troubleshooting and Common Mistakes	2421
20.27.1 "pci_alloc_msix returns EBUSY or ENXIO"	2421
20.27.2 "vmstat -i shows only one line on MSI-X guest"	2421
20.27.3 "Filter fires but vec->fire_count stays at zero"	2421
20.27.4 "Driver panics on kldunload after multiple load/unload cycles"	2421
20.27.5 "Different vectors all fire on the same CPU"	2421
20.27.6 "MSI-X allocation succeeds but vmstat -i shows no events"	2421
20.27.7 "Stray interrupts accumulate on the MSI-X admin vector"	2421
20.27.8 "Shared-IRQ behaviour on legacy fallback differs from MSI-X"	2421
20.27.9 "Stage 2 compiles but Stage 3 fails at bus_get_cpus link error"	2422
20.27.10 QEMU guest does not expose MSI-X despite using virtio-rng-pci"	2422
20.28 Wrapping Up	2422
20.28.1 A Reflection Before Chapter 21	2422
20.28.2 What to Do If You Are Stuck	2423
20.29 Bridge to Chapter 21	2423
20.30 Reference: Chapter 20 Quick-Reference Card	2424
20.30.1 Vocabulary	2424
20.30.2 Essential APIs	2424
20.30.3 Essential Macros	2425
20.30.4 Common Procedures	2425
20.30.5 Useful Commands	2425
20.30.6 Files to Keep Bookmarked	2425
20.31 Reference: Glossary of Chapter 20 Terms	2426

20.32	Reference: The Complete Stage 4 myfirst_msix.c Walkthrough	2426
20.32.1	The Top of the File	2427
20.32.2	The Per-Vector Helpers	2427
20.32.3	The Per-Vector Filter Functions	2428
20.32.4	The RX Task	2429
20.32.5	The Main Setup Function	2430
20.32.6	The Teardown Function	2433
20.32.7	The Bind Function	2434
20.32.8	The sysctl Function	2434
20.32.9	Lines of Code	2435
20.33	Reference: A Closing Note on Multi-Vector Philosophy	2435
21	DMA and High-Speed Data Transfer	2437
21.1	Reader Guidance & Outcomes	2437
21.1.1	Why bus_dma(9) Earns a Chapter of Its Own	2438
21.1.2	Where Chapter 20 Left the Driver	2439
21.1.3	What You Will Learn	2439
21.1.4	What This Chapter Does Not Cover	2440
21.1.5	Estimated Time Investment	2441
21.1.6	Prerequisites	2441
21.1.7	How to Get the Most Out of This Chapter	2442
21.1.8	Roadmap Through the Chapter	2443
21.2	Section 1: What Is DMA and Why Use It?	2443
21.2.1	The Problem With MMIO-Only Data Movement	2444
21.2.2	What DMA Means at the Hardware Level	2444
21.2.3	What the Driver Gains	2445
21.2.4	Real-World Examples	2445
21.2.5	Why Direct Memory Access Cannot Be Unrestricted	2446
21.2.6	Why bus_dma(9) Exists	2447
21.2.7	The Concrete Flow Chapter 21 Will Build	2448
21.2.8	Exercise: Identify DMA-Capable Devices on Your System	2449
21.2.9	Wrapping Up Section 1	2450
21.3	Section 2: Understanding FreeBSD's bus_dma(9) Interface	2450
21.3.1	The Four Pieces of the API	2450
21.3.2	Static Versus Dynamic Transactions	2451
21.3.3	The Sync Discipline	2452
21.3.4	The Load Callback Pattern	2453
21.3.5	Tag Hierarchies and Inheritance	2454
21.3.6	A Simple Hardware-to-Software Timeline	2455
21.3.7	Exercise: Walk Through a Sample DMA Setup Without a Real Device	2456
21.3.8	Wrapping Up Section 2	2457
21.4	Section 3: Allocating and Mapping DMA Memory	2457
21.4.1	New Softc Fields	2457
21.4.2	The Single-Segment Callback	2458
21.4.3	The Setup Function	2458
21.4.4	The Teardown Function	2461
21.4.5	Integrating Setup and Teardown Into Attach and Detach	2462
21.4.6	Running Stage 1	2463
21.4.7	What Stage 1 Does and Does Not Do	2463
21.4.8	Common Mistakes at This Stage	2464
21.4.9	A Word on the bus_dma Template API	2464
21.4.10	Wrapping Up Section 3	2465

21.5	Section 4: Synchronising and Using DMA Buffers	2465
21.5.1	The Ownership Model	2465
21.5.2	The Four Sync Flags in Depth	2466
21.5.3	The Four Common Transfer Patterns	2467
21.5.4	The Simulated Transfer's Sync Sequence	2468
21.5.5	What Coherence Flags Mean and Do Not Mean	2469
21.5.6	What Happens If a Sync Is Omitted	2469
21.5.7	Lock Context and Sync Calls	2470
21.5.8	The First Sync: When to Do It	2470
21.5.9	Wrapping Up Section 4	2471
21.6	Section 5: Building a Simulated DMA Engine	2471
21.6.1	The DMA Register Set	2471
21.6.2	Extending the Simulation State	2472
21.6.3	The Register Access Hook	2474
21.6.4	The Callout Handler	2476
21.6.5	The Driver's Side: Programming the Engine	2477
21.6.6	Exercising the Engine From User Space	2479
21.6.7	Observability: Per-Transfer Counters	2481
21.6.8	Verifying Stage 2	2481
21.6.9	What Stage 2 Does and Does Not Do	2482
21.6.10	Common Mistakes in Stage 2	2482
21.6.11	Wrapping Up Section 5	2483
21.7	Section 6: Handling DMA Completion in the Driver	2483
21.7.1	Why Interrupt-Driven Is the Right Design	2484
21.7.2	The Interrupt Path Modifications	2484
21.7.3	The New Interrupt-Driven Transfer Helper	2486
21.7.4	The MYFIRST_INTR_COMPLETE Bit Goes Live	2488
21.7.5	Verifying Stage 3	2488
21.7.6	Stopping In-Flight Transfers at Detach	2489
21.7.7	What Stage 3 Produces	2490
21.7.8	Common Mistakes in Stage 3	2490
21.7.9	Wrapping Up Section 6	2491
21.8	Section 7: DMA Error Handling and Edge Cases	2491
21.8.1	Failure Mode 1: bus_dma_tag_create Returns ENOMEM	2491
21.8.2	Failure Mode 2: bus_dmamem_alloc Returns ENOMEM	2492
21.8.3	Failure Mode 3: bus_dmamap_load Returns EINVAL	2492
21.8.4	Failure Mode 4: bus_dmamap_load Returns EINPROGRESS	2493
21.8.5	Failure Mode 5: Engine Reports Partial Transfer	2493
21.8.6	Failure Mode 6: Engine Never Signals Completion (Timeout)	2494
21.8.7	Failure Mode 7: Detach While a Transfer Is In Flight	2495
21.8.8	Failure Mode 8: Bounce Buffer Exhaustion	2496
21.8.9	Failure Mode 9: Unaligned Buffer	2497
21.8.10	Failure Mode 10: The Load Callback Runs with Non-Zero error	2497
21.8.11	Failure Mode 11: A Sync Call Is Skipped	2498
21.8.12	Failure Mode 12: The Callback Sets *addr to Zero and the Driver Proceeds	2498
21.8.13	A Pattern: The dma_setup / dma_tear down Pair	2499
21.8.14	Wrapping Up Section 7	2499
21.9	Section 8: Refactoring and Versioning Your DMA-Capable Driver	2499
21.9.1	The Final File Layout	2500
21.9.2	The myfirst_dma.h Header	2500
21.9.3	The myfirst_dma.c File	2502

21.9.4	The Updated Makefile	2503
21.9.5	The Version String in <code>kldstat</code>	2504
21.9.6	The <code>DMA.md</code> Document	2504
21.9.7	The Regression Pass	2506
21.9.8	What the Refactor Accomplished	2507
21.9.9	Exercise: Create a Utility That Verifies DMA Data	2507
21.9.10	Wrapping Up Section 8	2507
21.10	Reading a Real Driver Together: <code>if_re.c</code>	2508
21.10.1	The Parent Tag	2508
21.10.2	The Per-Buffer Child Tags	2509
21.10.3	The Ring Allocation	2509
21.10.4	The Per-Transfer Pattern	2510
21.10.5	The Cleanup	2511
21.10.6	What the Walkthrough Teaches	2511
21.11	A Deeper Look at Bounce Buffers	2511
21.11.1	What Bounce Buffers Are	2511
21.11.2	When Bounce Buffers Matter	2512
21.11.3	The Performance Cost	2512
21.11.4	How to Avoid Bouncing	2512
21.11.5	Observability	2513
21.12	A Deeper Look at 32-Bit-Capable Devices	2513
21.12.1	The Setting	2513
21.12.2	The Tag Setting	2513
21.12.3	Double-Addressing-Cycle (DAC)	2514
21.12.4	The Impact on Bounce Traffic	2514
21.13	A Deeper Look at IOMMU Integration	2514
21.13.1	What the IOMMU Does	2514
21.13.2	How the Kernel Integrates	2515
21.13.3	Detecting the IOMMU	2515
21.13.4	Performance Implications	2515
21.14	A Deeper Look at <code>BUS_DMA_COHERENT</code> vs Explicit Sync	2516
21.14.1	The Relationship	2516
21.14.2	Why the Sync Is Still Required	2516
21.14.3	When to Use the Flag	2516
21.15	A Deeper Look at Parent and Child Tag Inheritance	2516
21.15.1	Inheritance Semantics	2517
21.15.2	Why Drivers Create Explicit Parent Tags	2517
21.15.3	When to Skip the Explicit Parent	2517
21.16	A Deeper Look at Descriptor Rings as a Future Topic	2517
21.17	A Mental Model for Chapter 21's Sync Discipline	2518
21.18	Patterns From Real FreeBSD Drivers	2518
21.18.1	Pattern: <code>bus_dmamem_alloc</code> for Static Descriptor Rings	2518
21.18.2	Pattern: <code>bus_dmamap_load_mbuf_sg</code> for Per-Packet Mappings	2519
21.18.3	Pattern: Parent-Child Tag Hierarchy	2519
21.18.4	Pattern: <code>BUS_DMASYNC_PREREAD</code> <code>BUS_DMASYNC_PREWRITE</code> for Shared Rings	2519
21.18.5	Pattern: Single-Segment Callback as a Universal Helper	2519
21.18.6	Pattern: Coherent Tag for Rings, Non-Coherent for Payloads	2519
21.19	Hands-On Labs	2519
21.19.1	Lab 1: Identify DMA-Capable Devices on Your System	2520
21.19.2	Lab 2: Stage 1, Allocate and Map a DMA Buffer	2520
21.19.3	Lab 3: Stage 2, Simulated DMA Engine with Polling	2520

21.19.4 Lab 4: Stage 3, Interrupt-Driven Completion	2521
21.19.5 Lab 5: Stage 4, Refactor and Regression	2521
21.20 Challenge Exercises	2522
21.20.1 Challenge 1: Multi-Buffer Rotation	2522
21.20.2 Challenge 2: Scatter-Gather Load	2522
21.20.3 Challenge 3: Descriptor Ring Sketch	2522
21.20.4 Challenge 4: Partial Transfer Reporting	2522
21.20.5 Challenge 5: IOMMU Awareness	2523
21.20.6 Challenge 6: Bounce Buffer Observability	2523
21.20.7 Challenge 7: Tag Template Refactor	2523
21.20.8 Challenge 8: DTrace-Based Transfer Profiling	2523
21.21 Troubleshooting and Common Mistakes	2523
21.21.1 “bus_dma_tag_create fails with EINVAL”	2523
21.21.2 “bus_dmamap_load callback runs with error != 0”	2524
21.21.3 “dma_bus_addr is zero after bus_dmamap_load returned zero”	2524
21.21.4 “Transfers succeed but the buffer contents are wrong”	2524
21.21.5 “bus_dmamap_unload panics with ‘map not loaded’ ”	2524
21.21.6 “bus_dmamem_free panics with ‘no allocation’ ”	2524
21.21.7 “dma_complete_intrs is zero even though transfers succeed”	2525
21.21.8 “cv_timedwait always returns EWOULDBLOCK”	2525
21.21.9 “Driver hangs on kldunload”	2525
21.21.10 “WITNESS warns about lock order reversal”	2525
21.21.11 “Counters climb but the buffer is not populated”	2525
21.21.12 “Load-unload cycle leaks memory”	2526
21.21.13 “Transfers are slow compared to expected”	2526
21.22 Worked Example: Tracing a DMA Transfer End to End	2526
21.22.1 The Starting State	2526
21.22.2 Event 1: The sysctl Handler Runs	2526
21.22.3 Event 2: The Helper Takes the Softc Lock	2527
21.22.4 Event 3: The PRE Sync	2527
21.22.5 Event 4: The Device Programming	2527
21.22.6 Event 5: The Simulation Arms the Callout	2527
21.22.7 Event 6: The Helper Waits	2527
21.22.8 Event 7: The Callout Fires	2528
21.22.9 Event 8: The Simulation Raises the Completion Bit	2528
21.22.10 Event 9: The Filter Runs	2528
21.22.11 Event 10: The Task Runs	2528
21.22.12 Event 11: The POST Sync	2528
21.22.13 Event 12: The Task Releases the Lock	2529
21.22.14 Event 13: The Helper Wakes	2529
21.22.15 Event 14: The Helper Examines the Status	2529
21.22.16 Event 15: The Sysctl Handler Returns	2529
21.22.17 Event 16: The User Sees the Result	2529
21.22.18 The Total Time	2529
21.22.19 The Lesson	2529
21.23 A Deeper Look at Locking and DMA	2530
21.23.1 The Softc Mutex Covers the DMA State	2530
21.23.2 The Tag’s Lockfunc	2530
21.23.3 No Locks in the Sync Path	2530
21.23.4 The Detach Race	2531

21.24A Deeper Look at Observability for DMA Drivers	2531
21.24.1 The Per-Transfer Counters	2531
21.24.2 The Ring Buffer of Recent Transfers	2531
21.24.3 DTrace Probes	2531
21.24.4 vmstat -i and systat -vmstat	2532
21.24.5 procstat -kke	2532
21.24.6 sysctl hw.busdma	2532
21.25A Deeper Look at Performance of DMA Transfers	2533
21.25.1 The Latency Budget	2533
21.25.2 The Throughput Ceiling	2533
21.25.3 Measuring the Chapter's Driver	2533
21.25.4 What the Measurement Tells Us	2534
21.26 Additional Troubleshooting: Ten More Failure Patterns	2534
21.26.1 "Kernel panics on first transfer with 'null pointer dereference'"	2534
21.26.2 "First transfer works, subsequent transfers hang"	2534
21.26.3 "Transfers succeed but dma_complete_intrs is zero"	2534
21.26.4 "Buffer contents are correct but verification fails"	2534
21.26.5 "dma_errors counter climbs but the driver does not propagate the error"	2535
21.26.6 "taskqueue_drain hangs on kldunload"	2535
21.26.7 "bus_dmamem_alloc returns a buffer above 4 GB despite lowaddr = MAXADDR_32BIT"	2535
21.26.8 "Test script runs fine once, fails after module reload"	2535
21.26.9 "WITNESS warns about 'DMA map used while not loaded' or similar"	2535
21.26.10 "Transfers work on my amd64 test VM but fail on the arm64 build server"	2535
21.27 Reference: The Complete Stage 4 myfirst_dma.c Walkthrough	2536
21.27.1 The Includes and Macros	2536
21.27.2 The Single-Segment Callback	2536
21.27.3 The Setup Function	2536
21.27.4 The Teardown Function	2536
21.27.5 The Polling Transfer Helper	2536
21.27.6 The Interrupt-Driven Transfer Helper	2537
21.27.7 The Completion Handler	2537
21.27.8 The Sysctl Registration	2537
21.27.9 The Sysctl Handlers	2538
21.27.10 Lines of Code	2539
21.28 Reference: FreeBSD Manual Pages for Chapter 21	2539
21.29A Deeper Look at How DMA and Interrupts Fit Together	2540
21.29.1 The Three-Piece Architecture	2540
21.29.2 The Common Flow	2540
21.29.3 The Scale Story	2541
21.29.4 The Interaction Patterns	2541
21.30A Deeper Look at Comparisons with Other Kernel DMA APIs	2541
21.30.1 Linux's DMA API	2542
21.30.2 NetBSD's bus_dma	2542
21.30.3 Windows's DMA Abstractions	2542
21.31 Reference: A Short Tour of /usr/src/sys/kern/subr_bus_dma.c	2543
21.32 Reference: Driver Memorable Phrases	2543
21.33 Reference: A Comparison Table Across Part 4	2544
21.34 Reference: What the Reader Should Now Be Able to Do	2544
21.35 Reference: When to Reach for Each API Variant	2545
21.36 Wrapping Up	2545
21.36.1 A Reflection Before Chapter 22	2546

21.36.2 What to Do If You Are Stuck	2546
21.37 Bridge to Chapter 22	2547
21.38 Reference: Chapter 21 Quick-Reference Card	2547
21.38.1 Vocabulary	2548
21.38.2 Essential APIs	2548
21.38.3 Essential Flags	2548
21.38.4 Essential Sync Operations	2548
21.38.5 Common Procedures	2549
21.38.6 Useful Commands	2549
21.38.7 Files to Keep Bookmarked	2550
21.39 Reference: Glossary of Chapter 21 Terms	2550
21.40 Reference: A Closing Note on DMA Philosophy	2550
22 Power Management	2552
22.1 Reader Guidance & Outcomes	2552
22.1.1 Why Suspend and Resume Earn a Chapter of Their Own	2553
22.1.2 Where Chapter 21 Left the Driver	2554
22.1.3 What You Will Learn	2554
22.1.4 What This Chapter Does Not Cover	2555
22.1.5 Estimated Time Investment	2556
22.1.6 Prerequisites	2556
22.1.7 How to Get the Most Out of This Chapter	2557
22.1.8 Roadmap Through the Chapter	2557
22.2 Section 1: What Is Power Management in Device Drivers?	2558
22.2.1 Why the Driver Must Care About Power	2559
22.2.2 System-Level Versus Device-Level Power Management	2559
22.2.3 ACPI System Sleep States (S0 to S5)	2560
22.2.4 PCI and PCIe Device Power States (D0 to D3cold)	2561
22.2.5 PCIe Link States and Active-State Power Management (ASPM)	2562
22.2.6 The Anatomy of a Suspend-Resume Cycle	2563
22.2.7 Wake Sources	2564
22.2.8 Real-World Examples: Wi-Fi, NVMe, USB	2564
22.2.9 What the Reader Has Gained	2565
22.2.10 Wrapping Up Section 1	2566
22.3 Section 2: FreeBSD's Power Management Interface	2566
22.3.1 The Four Kobj Methods	2566
22.3.2 Adding the Methods to the Driver's Method Table	2567
22.3.3 Prototypes and Return Values	2568
22.3.4 The Order of Event Delivery	2568
22.3.5 bus_generic_suspend, bus_generic_resume, and the PCI Bus	2569
22.3.6 pci_save_state and pci_restore_state: When the Driver Calls Them	2570
22.3.7 The pci_has_pm Helper	2571
22.3.8 PME#: Enabling, Disabling, and Clearing	2571
22.3.9 A First Skeleton: Stage 1	2572
22.3.10 What the Skeleton Proves	2574
22.3.11 A Note on Detach, Quiesce, and Suspend	2574
22.3.12 The Detach Path as Inspiration	2575
22.3.13 Observing the Skeleton with WITNESS	2575
22.3.14 Wrapping Up Section 2	2576
22.4 Section 3: Quiescing a Device Safely	2576
22.4.1 What Quiesce Really Means	2576
22.4.2 The Order Matters	2577

22.4.3	Helpers, Not Inline Code	2577
22.4.4	Implementing myfirst_mask_interrupts	2579
22.4.5	Implementing myfirst_drain_dma	2580
22.4.6	Implementing myfirst_drain_workers	2581
22.4.7	Updating the Suspend Method	2582
22.4.8	Saving Runtime State the Hardware Loses	2583
22.4.9	The suspended Flag as a User-Facing Invariant	2583
22.4.10	Handling the Case Where DMA Is Not in Flight	2584
22.4.11	Testing Stage 2	2584
22.4.12	A Cautious Note on Locking	2585
22.4.13	Integrating with the Shutdown Method	2586
22.4.14	Wrapping Up Section 3	2586
22.5	Section 4: Restoring State on Resume	2586
22.5.1	What the PCI Layer Has Already Done	2587
22.5.2	The Resume Discipline	2587
22.5.3	Why pci_enable_busmaster Matters	2589
22.5.4	Restoring Device-Specific State	2589
22.5.5	Validating Device Identity After Resume	2590
22.5.6	Detecting and Recovering from a Device Reset	2591
22.5.7	Restoring DMA State	2591
22.5.8	Re-Arming Interrupts	2592
22.5.9	Wake Source Cleanup	2592
22.5.10	Updating the Stage 3 Driver	2593
22.5.11	Interaction with the Chapter 20 MSI-X Setup	2594
22.5.12	Handling a Failed Resume Gracefully	2594
22.5.13	A Short Detour: pci_save_state / pci_restore_state in Runtime PM	2595
22.5.14	A Reality Check Against a Real Driver	2596
22.5.15	Wrapping Up Section 4	2596
22.6	Section 5: Handling Runtime Power Management	2596
22.6.1	What Runtime PM Means in FreeBSD	2597
22.6.2	The Runtime-PM State Machine	2597
22.6.3	Implementation	2598
22.6.4	The Runtime-Suspend and Runtime-Resume Helpers	2600
22.6.5	Interaction Between Runtime PM and System PM	2602
22.6.6	Exposing Runtime-PM Controls Through Sysctl	2602
22.6.7	Tradeoffs	2603
22.6.8	A Word on Platform Runtime PM	2603
22.6.9	Wrapping Up Section 5	2604
22.7	Section 6: Interacting With the Power Framework	2604
22.7.1	The Four User-Space Entry Points	2604
22.7.2	Using acpiconf to Suspend the System	2605
22.7.3	Using zzz	2606
22.7.4	Using devctl for Per-Device Suspend	2606
22.7.5	Using devinfo to Inspect Device State	2607
22.7.6	Inspecting Power States Through sysctl	2607
22.7.7	A Regression Script	2608
22.7.8	Running the Stress Test	2610
22.7.9	Combining Runtime PM and User-Space Testing	2610
22.7.10	Interpreting Failure Modes	2611
22.7.11	A Minimal Troubleshooting Flow	2611
22.7.12	Wrapping Up Section 6	2612

22.8	Section 7: Debugging Power Management Issues	2612
22.8.1	Symptom 1: Frozen Device After Resume	2612
22.8.2	Symptom 2: Lost Interrupts	2613
22.8.3	Symptom 3: Bad DMA After Resume	2613
22.8.4	Symptom 4: Lost PME# Wake Events	2615
22.8.5	Symptom 5: WITNESS Complaints During Suspend	2616
22.8.6	Symptom 6: Counters That Do Not Match	2616
22.8.7	Symptom 7: Hangs During Suspend	2617
22.8.8	Using DTrace to Trace Suspend and Resume	2617
22.8.9	A Note on Logging Discipline	2618
22.8.10	Using the INVARIANTS Kernel for Assertion Coverage	2619
22.8.11	A Debugging Case Study	2619
22.8.12	Wrapping Up Section 7	2621
22.9	Section 8: Refactoring and Versioning Your Power-Aware Driver	2622
22.9.1	The Target Layout	2622
22.9.2	The myfirst_power.c File	2623
22.9.3	Updating myfirst_pci.c	2623
22.9.4	Updating the Makefile	2625
22.9.5	Writing POWER.md	2625
22.9.6	Regression Script	2626
22.9.7	Integration With the Existing Regression Tests	2627
22.9.8	Version History	2627
22.9.9	A Final Integration Test on Real Hardware	2627
22.9.10	The Chapter 22 Code in One Place	2628
22.9.11	Wrapping Up Section 8	2628
22.10	Deep Look: Power Management in /usr/src/sys/dev/re/if_re.c	2629
22.10.1	The Method Table	2629
22.10.2	re_suspend	2629
22.10.3	re_resume	2630
22.10.4	re_shutdown	2632
22.10.5	re_setwol	2632
22.10.6	What the Deep Look Shows	2634
22.11	Deep Look: Simpler Patterns in if_xl.c and virtio_blk.c	2634
22.11.1	if_xl.c: Shutdown Calls Suspend	2634
22.11.2	virtio_blk.c: Minimal Quiesce	2635
22.11.3	What the Comparison Shows	2636
22.12	Deep Look: ACPI Sleep States in More Detail	2637
22.12.1	S0: Working	2637
22.12.2	S1: Standby	2637
22.12.3	S2: Reserved	2637
22.12.4	S3: Suspend to RAM	2637
22.12.5	S4: Suspend to Disk (Hibernate)	2638
22.12.6	S5: Soft Off	2638
22.12.7	Observing Which States the Platform Supports	2639
22.12.8	What to Test	2639
22.12.9	Mapping Sleep States to the Driver's Methods	2639
22.13	Deep Look: PCIe Link States and ASPM in Action	2640
22.13.1	Why the Link Has Its Own States	2640
22.13.2	L0: Active	2640
22.13.3	L0s: Transmitter Standby	2640
22.13.4	L1: Both Sides Standby	2641

22.13.5 L1.1 and L1.2: Deeper L1 Sub-States	2641
22.13.6 L2: Near-Off	2641
22.13.7 Who Controls ASPM	2641
22.13.8 When ASPM Matters to the Driver	2642
22.13.9 Observing Link States	2642
22.13.10 Link State and Power Consumption	2642
22.14 Deep Look: Wake Sources Explained	2643
22.14.1 PME# on PCIe	2643
22.14.2 USB Remote Wakeup	2644
22.14.3 GPIO-Based Wake on Embedded Platforms	2644
22.14.4 Wake on LAN (WoL)	2644
22.14.5 Wake on Lid, Power Button, etc.	2645
22.14.6 When the Driver Must Enable Wake	2645
22.14.7 Testing Wake Sources	2645
22.15 Deep Look: The <code>hw.pci.do_power_suspend</code> Tunable	2646
22.15.1 What the Default Does	2646
22.15.2 What <code>hw.pci.do_power_suspend=0</code> Does	2646
22.15.3 When to Change It	2646
22.15.4 Verifying the Tunable Is in Effect	2646
22.15.5 Interaction with <code>pci_save_state</code>	2647
22.15.6 Does the Tunable Affect System Suspend or <code>devctl suspend</code> ?	2647
22.15.7 A Concrete Debugging Scenario	2647
22.16 Deep Look: <code>DEVICE_QUIESCE</code> and When You Need It	2648
22.16.1 What <code>DEVICE_QUIESCE</code> Is For	2648
22.16.2 What the Default Does	2648
22.16.3 When You Would Implement It	2648
22.16.4 An Example from <code>bce(4)</code>	2649
22.16.5 Relation to <code>DEVICE_SUSPEND</code>	2649
22.16.6 An Optional Addition to <code>myfirst</code>	2649
22.17 Hands-On Labs	2650
22.17.1 Lab 1: Single-Cycle Suspend-Resume	2650
22.17.2 Lab 2: One-Hundred-Cycle Stress	2651
22.17.3 Lab 3: Transfer Across a Cycle	2652
22.17.4 Lab 4: Runtime PM (Optional)	2653
22.17.5 Lab Notes	2653
22.18 Challenge Exercises	2654
22.18.1 Challenge 1: Implement a Wake-on-Sysctl Mechanism	2654
22.18.2 Challenge 2: Save and Restore a Descriptor Ring	2654
22.18.3 Challenge 3: Implement a Veto Policy	2654
22.18.4 Challenge 4: Add a Post-Resume Self-Test	2655
22.18.5 Challenge 5: Implement Manual <code>pci_save_state</code> / <code>pci_restore_state</code>	2655
22.19 Troubleshooting Reference	2656
22.19.1 “ <code>devctl: DEV_SUSPEND failed: Operation not supported</code> ”	2656
22.19.2 “ <code>devctl: DEV_SUSPEND failed: Device busy</code> ”	2656
22.19.3 “ <code>devctl: DEV_RESUME failed</code> ”	2656
22.19.4 Device is suspended but <code>dev.myfirst.0.suspended</code> reads 0	2656
22.19.5 <code>power_suspend_count != power_resume_count</code>	2656
22.19.6 DMA transfers fail after resume	2656
22.19.7 WITNESS complains about a lock held during suspend	2657
22.19.8 System does not wake from S3	2657
22.19.9 Runtime PM never fires	2657

22.19.10	Kernel panic during suspend	2657
22.20	Wrapping Up	2657
22.20.1	A Reflection Before Chapter 23	2658
22.20.2	What to Do If You Are Stuck	2658
22.21	Part 4 Checkpoint	2659
22.22	Bridge to Chapter 23	2660
22.23	Reference: Chapter 22 Quick-Reference Card	2661
22.23.1	Vocabulary	2661
22.23.2	Essential Kobj Methods	2661
22.23.3	Essential PCI APIs	2661
22.23.4	Essential Bus Helpers	2662
22.23.5	Essential Sysctls	2662
22.23.6	Useful Commands	2662
22.23.7	Common Procedures	2662
22.23.8	Files to Keep Bookmarked	2663
22.24	Reference: Glossary of Chapter 22 Terms	2664
22.25	Reference: A Closing Note on Power-Management Philosophy	2664
23	Debugging and Tracing	2666
23.1	Reader Guidance & Outcomes	2666
23.1.1	Why Kernel Debugging Earns a Chapter of Its Own	2667
23.1.2	Where Chapter 22 Left the Driver	2667
23.1.3	What You Will Learn	2668
23.1.4	What This Chapter Does Not Cover	2669
23.1.5	Estimated Time Investment	2669
23.1.6	Prerequisites	2670
23.1.7	How to Get the Most Out of This Chapter	2670
23.1.8	Roadmap Through the Chapter	2671
23.2	Section 1: Why Kernel Debugging Is Challenging	2672
23.2.1	The Nature of Kernel Code	2672
23.2.2	Risks of Debugging in Kernel Space	2673
23.2.3	Differences From Userland Debugging	2674
23.2.4	Importance of Controlled, Incremental Changes	2675
23.2.5	The Mental Shift	2675
23.2.6	Wrapping Up Section 1	2676
23.3	Section 2: Logging With printf() and device_printf()	2676
23.3.1	Basics of printf() in Kernel Context	2676
23.3.2	Preferred Use: device_printf() for Structured, Device-Aware Logging	2677
23.3.3	device_log() and Log Priority Levels	2678
23.3.4	What to Log and What to Leave Out	2678
23.3.5	Tracing Function Flow Without Spamming	2680
23.3.6	Redirecting Logs to dmesg and /var/log/messages	2680
23.3.7	A Concrete Walkthrough: Logging in the myfirst Driver	2681
23.3.8	A Note on uprintf and tprintf	2682
23.3.9	Formatting Conventions	2682
23.3.10	Exercise: Add device_printf() at Key Points in Your Driver	2683
23.3.11	Wrapping Up Section 2	2683
23.4	Section 3: Using dmesg and Syslog for Diagnostics	2683
23.4.1	The Kernel Message Buffer	2684
23.4.2	dmesg in Practice	2684
23.4.3	Permanent Logs in /var/log/messages	2685
23.4.4	Console Output Versus Buffered Output	2686

23.4.5	Reviewing Logs in Practice	2687
23.4.6	Filtering Output	2688
23.4.7	Permanent Configuration: <code>/etc/syslog.conf</code> and <code>/etc/newsyslog.conf</code>	2688
23.4.8	Correlating Messages Across a Boot Cycle	2689
23.4.9	Exercise: Generate a Known Driver Message and Confirm It Appears	2690
23.4.10	Wrapping Up Section 3	2690
23.5	Section 4: Kernel Debug Builds and Options	2690
23.5.1	The Core Debug Options	2691
23.5.2	Building a Debug Kernel	2692
23.5.3	When to Use <code>INVARIANTS</code>	2694
23.5.4	When to Use <code>WITNESS</code> and <code>DEBUG_LOCKS</code>	2695
23.5.5	When to Use <code>DEBUG_MEMGUARD</code> and <code>MEMGUARD</code>	2695
23.5.6	Trade-Offs: Performance vs. Debug Visibility	2696
23.5.7	A Survey of <code>ddb</code> Commands	2696
23.5.8	A Brief Introduction to <code>kgdb</code>	2698
23.5.9	Exercise: Build a Debug Kernel and Confirm Symbols Are Present With <code>kgdb</code>	2699
23.5.10	Wrapping Up Section 4	2700
23.6	Section 5: Using DTrace for Live Kernel Inspection	2700
23.6.1	What Is DTrace?	2700
23.6.2	Enabling DTrace Support	2701
23.6.3	Writing Simple DTrace Scripts	2702
23.6.4	Measuring Function Latency	2703
23.6.5	Using the <code>syscall</code> Provider	2704
23.6.6	Using the <code>sched</code> Provider	2705
23.6.7	Using the <code>io</code> Provider	2705
23.6.8	Statically Defined Tracing (SDT)	2705
23.6.9	Useful Providers for Driver Work	2707
23.6.10	Common DTrace Pitfalls	2707
23.6.11	Exercise: Use DTrace to Measure How Long a Driver Read Operation Takes	2708
23.6.12	Wrapping Up Section 5	2709
23.7	Section 6: Tracing Kernel Activity With <code>ktrace</code> and <code>kdump</code>	2709
23.7.1	What Is <code>ktrace</code> ?	2709
23.7.2	Starting a Trace	2710
23.7.3	Reading Output With <code>kdump</code>	2711
23.7.4	Useful <code>kdump</code> Filters	2712
23.7.5	Finding Driver-Related Syscalls	2712
23.7.6	Comparing <code>ktrace</code> and DTrace	2712
23.7.7	A Note on KTR vs. <code>ktrace</code>	2713
23.7.8	Exercise: Trace a Simple User Program Accessing Your Driver	2713
23.7.9	Wrapping Up Section 6	2715
23.8	Section 7: Diagnosing Common Driver Bugs	2715
23.8.1	7.1 Memory Leaks: the Quiet Growth	2715
23.8.2	7.2 Race Conditions: the Rare, Destructive Bug	2717
23.8.3	7.3 <code>bus_space</code> and <code>bus_dma</code> Misuse	2717
23.8.4	7.4 Use-After-Detach	2718
23.8.5	7.5 Interrupt Handler Mistakes	2719
23.8.6	7.6 Lifecycle Sequencing Errors	2720
23.8.7	7.7 Copy-To/From-User Mistakes	2720
23.8.8	7.8 Module-Level Bugs	2721
23.8.9	7.9 A Debug Checklist	2721
23.8.10	Wrapping Up Section 7	2721

23.9	Section 8: Refactoring and Versioning With Trace Points	2722
23.9.1	8.1 Why a Debug Header	2722
23.9.2	8.2 Declaring Debug Classes	2723
23.9.3	8.3 The DPRINTF Macro	2723
23.9.4	8.4 Adding <code>sc_debug</code> to the softc	2724
23.9.5	8.5 The Entry / Exit / Error Pattern	2725
23.9.6	8.6 Adding SDT Probes	2726
23.9.7	8.7 Writing <code>DEBUG.md</code>	2727
23.9.8	8.8 Bumping the Version	2728
23.9.9	8.9 Wrapping Section 8	2729
23.10	Hands-On Labs	2729
23.10.1	Lab 23.1: A First DDB Session	2729
23.10.2	Lab 23.2: Measuring the Driver with DTrace	2730
23.10.3	Lab 23.3: User-Side Tracing with <code>ktrace</code>	2731
23.10.4	Lab 23.4: Finding a Memory Leak with <code>vmstat -m</code>	2732
23.10.5	Lab 23.5: Installing the 1.6-debug Refactor	2732
23.11	Challenge Exercises	2734
23.11.1	Challenge 23.1: An SDT Probe in a Real Driver	2734
23.11.2	Challenge 23.2: A Custom DTrace Script	2734
23.11.3	Challenge 23.3: A WITNESS Experiment	2735
23.11.4	Challenge 23.4: Extended Error Categorisation	2735
23.11.5	Challenge 23.5: Boot-Time Debug	2735
23.12	Troubleshooting and Common Mistakes	2735
23.13	Appendix: Real-World Debug Patterns in the FreeBSD Tree	2737
23.14	Appendix: Worked Case Studies in Driver Debugging	2737
23.14.1	Case Study 1: The Missing Message	2738
23.14.2	Case Study 2: The Intermittent Panic	2739
23.14.3	Case Study 3: The Slow Interrupt	2740
23.14.4	Case Study 4: The Vanishing Device	2742
23.14.5	Case Study 5: The Corrupted Output	2743
23.14.6	Summary of the Five Cases	2745
23.15	Appendix: Extended DTrace Techniques for Driver Work	2746
23.15.1	Predicates and Selective Tracing	2746
23.15.2	Aggregations That Tell a Story	2746
23.15.3	Thread-Local Variables	2747
23.15.4	Arrays and Associative Access	2747
23.15.5	Speculative Tracing	2748
23.15.6	Predicate Chains for Narrow Tracing	2748
23.15.7	Profiling the Kernel	2749
23.15.8	Pretty-Printing Structures	2749
23.15.9	Ring-Buffered Output	2749
23.15.10	Exiting Cleanly	2750
23.15.11	Summary	2750
23.16	Appendix: WITNESS Trace Interpretation	2750
23.16.1	The General Format	2750
23.16.2	The Header	2751
23.16.3	The Lock Descriptions	2751
23.16.4	The Backtrace	2752
23.16.5	Reading a Real Example	2752
23.16.6	Asserting Lock Ownership	2752
23.16.7	When WITNESS Is Too Noisy	2752

23.17	Appendix: Building a Driver-Specific Debug Toolkit	2753
23.17.1	Debug-Only Sysctls	2753
23.17.2	A Debug Ioctl	2753
23.17.3	Counters for Key Events	2754
23.17.4	A Self-Test Ioctl	2754
23.17.5	Integration with rc Scripts	2754
23.17.6	Summary	2755
23.18	Appendix: A Lab Reference for Chapter 23	2755
23.19	Appendix: Complete Annotated Listing of myfirst_debug.h and myfirst_debug.c Changes	2756
23.19.1	myfirst_debug.h	2756
23.19.2	The SDT Definitions in myfirst_debug.c	2758
23.19.3	Firing the Probes in the Driver	2759
23.19.4	Registering the Sysctl in myfirst_attach	2761
23.19.5	The Version Bump	2762
23.19.6	A Sample dmesg With All Classes Enabled	2762
23.19.7	A Sample DTrace Session With the Probes	2763
23.19.8	Summary of the Refactor	2763
23.20	Appendix: What the Kernel Does When a Driver Panics	2763
23.20.1	The Panic Sequence	2764
23.20.2	Driver Responsibilities	2764
23.20.3	Reading a Crash Dump	2764
23.20.4	Combining Kernel and Driver Symbols	2765
23.20.5	Crash Dump Size	2765
23.20.6	Summary	2765
23.21	Appendix: Chapter 23 at a Glance	2765
23.21.1	Tools in order of first use	2766
23.21.2	Reflexes to build	2766
23.21.3	The 1.6-debug refactor	2766
23.21.4	Recommended next steps	2766
23.22	Wrapping Up	2767
23.22.1	What the reader can now do	2767
23.22.2	What lies ahead	2767
23.23	Bridge to Chapter 24	2768
23.24	Reference: Chapter 23 Quick-Reference Card	2768
23.24.1	Logging	2768
23.24.2	Syslog priority levels	2768
23.24.3	Kernel options for debugging	2769
23.24.4	Common DTrace one-liners	2769
23.24.5	ktrace / kdump workflow	2770
23.24.6	Debugging checklist (abbreviated)	2770
23.25	Reference: Glossary of Chapter 23 Terms	2770
24	Integrating with the Kernel	2772
24.1	Reader Guidance & Outcomes	2772
24.1.1	Why devfs, ioctl, and sysctl Integration Earn a Chapter of Their Own	2773
24.1.2	Where Chapter 23 Left the Driver	2774
24.1.3	What You Will Learn	2774
24.1.4	What This Chapter Does Not Cover	2775
24.1.5	Estimated Time Investment	2776
24.1.6	Prerequisites	2777
24.1.7	How to Get the Most Out of This Chapter	2777
24.1.8	Roadmap Through the Chapter	2778

24.2	Section 1: Why Integration Matters	2779
24.2.1	From Standalone Module to System Component	2780
24.2.2	Common Integration Targets	2780
24.2.3	Benefits of Proper Integration	2781
24.2.4	A Small Tour of System Tools That Depend on Integration	2782
24.2.5	What “Optional” Means in This Chapter	2783
24.2.6	Pitfalls on the Way to Integration	2783
24.2.7	A Mental Model for the Chapter	2784
24.2.8	Wrapping Up Section 1	2785
24.3	Section 2: Working With devfs and the Device Tree	2785
24.3.1	What devfs Is	2785
24.3.2	The Lifecycle of a cdev	2786
24.3.3	The make_dev Family	2786
24.3.4	The cdevsw Structure	2788
24.3.5	Per-Cdev State Through si_drv1	2790
24.3.6	The null(4) Reference	2791
24.3.7	Multiple Nodes and Clonable Nodes	2792
24.3.8	Permissions, Ownership, and devfs.rules	2794
24.3.9	Putting It All Together: The Stage 1 Driver	2794
24.3.10	A Concrete Walkthrough: Loading and Inspecting the Stage 1 Driver	2798
24.3.11	Common Mistakes With devfs	2798
24.3.12	Pitfalls Unique to Multi-Instance Drivers	2799
24.3.13	Wrapping Up Section 2	2800
24.4	Section 3: Implementing ioctl() Support	2800
24.4.1	What ioctl(2) Is, and Why Drivers Need It	2800
24.4.2	How ioctl Numbers Are Encoded	2801
24.4.3	Choosing a Group Letter	2802
24.4.4	The d_ioctl Callback Signature	2803
24.4.5	How the Kernel Performs Copyin and Copyout	2803
24.4.6	Designing a Public Header	2804
24.4.7	Implementing the Dispatcher	2806
24.4.8	The Softc Additions	2809
24.4.9	Wiring the Dispatcher Into cdevsw	2810
24.4.10	Building the Stage 2 Driver	2810
24.4.11	The myfirstctl User-Space Companion	2811
24.4.12	Common Pitfalls With ioctl	2813
24.4.13	Wrapping Up Section 3	2814
24.5	Section 4: Exposing Metrics Through sysctl()	2815
24.5.1	What sysctl Is, and Why Drivers Use It	2815
24.5.2	The Sysctl Namespace	2815
24.5.3	Static OIDs Versus Dynamic OIDs	2816
24.5.4	Handler-Backed OIDs	2818
24.5.5	The myfirst Sysctl Tree	2819
24.5.6	Wiring Sysctl Into Attach and Detach	2822
24.5.7	Boot-Time Tunables Through /boot/loader.conf	2823
24.5.8	Combining Sysctl With the Debug Mask	2823
24.5.9	Building the Stage 3 Driver	2824
24.5.10	Common Pitfalls With Sysctl	2825
24.5.11	Wrapping Up Section 4	2826
24.6	Section 5: Networking Integration (Optional)	2826
24.6.1	Why This Section Is Optional	2826

24.6.2	What an ifnet Is	2827
24.6.3	A Walkthrough Using <code>disc(4)</code>	2827
24.6.4	How an Operator Sees the Result	2829
24.6.5	What This Pattern Generalises To	2829
24.6.6	Wrapping Up Section 5	2830
24.7	Section 6: CAM Storage Integration (Optional)	2830
24.7.1	Why This Section Is Optional	2830
24.7.2	What CAM Is	2830
24.7.3	The Registration Lifecycle	2831
24.7.4	The Action Callback	2832
24.7.5	How an Operator Sees the Result	2834
24.7.6	Pattern Recognition	2834
24.7.7	Wrapping Up Section 6	2835
24.8	Section 7: Registration, Teardown, and Cleanup Discipline	2835
24.8.1	The Cardinal Rule	2835
24.8.2	Why Reverse Order	2835
24.8.3	The <code>goto err1</code> Pattern in Attach	2836
24.8.4	The Detach Mirror	2838
24.8.5	Module Event Handlers	2839
24.8.6	<code>EVENTHANDLER</code> for System Events	2840
24.8.7	<code>SYSINIT</code> for One-Shot Kernel Init	2840
24.8.8	<code>bus_generic_detach</code> and <code>device_delete_children</code>	2841
24.8.9	Putting It All Together	2842
24.8.10	Wrapping Up Section 7	2842
24.9	Section 8: Refactoring and Versioning	2843
24.9.1	A Driver Has a History	2843
24.9.2	From One File to Several	2843
24.9.3	Version Strings, Version Numbers, and the API Version	2844
24.9.4	When to Bump Each	2845
24.9.5	Compatibility Shims	2845
24.9.6	A Worked Refactor: Splitting <code>myfirst.c</code>	2846
24.9.7	Three In-Tree Drivers Compared	2846
24.9.8	Wrapping Up Section 8	2847
24.9.9	Putting It All Together: The Final Attach and Detach	2847
24.10	Hands-on Labs	2850
24.10.1	Lab 1: Build and Load the Stage 1 Driver	2850
24.10.2	Lab 2: Add the <code>ioctl</code> Interface	2852
24.10.3	Lab 3: Add the <code>Sysctl</code> Tree	2854
24.10.4	Lab 4: Walk the Lifecycle by Injecting a Failure	2856
24.10.5	Lab 5: Trace the Integration Surfaces with <code>DTrace</code>	2857
24.10.6	Lab 6: Integration Smoke Test	2858
24.10.7	Lab 7: Reload Without Restarting User-Space Programs	2861
24.10.8	Wrapping Up the Labs	2862
24.11	Challenge Exercises	2862
24.11.1	Challenge 1: Add a Variable-Length <code>ioctl</code>	2862
24.11.2	Challenge 2: Add a Per-Open Counter	2863
24.11.3	Challenge 3: Implement Soft Detach With <code>dev_ref</code>	2863
24.11.4	Challenge 4: Replace the Static Magic Letter	2864
24.11.5	Challenge 5: Add an <code>EVENTHANDLER</code> for Shutdown	2864
24.11.6	Challenge 6: A Second Per-Driver <code>Sysctl</code> Subtree	2864
24.11.7	Challenge 7: Cross-Module Dependency	2864

24.11.8 Closing the Challenges	2865
24.12 Troubleshooting	2865
24.12.1 /dev/myfirst0 Does Not Appear After kldload	2865
24.12.2 cat /dev/myfirst0 Returns “Permission denied”	2865
24.12.3 ioctl Returns “Inappropriate ioctl for device”	2866
24.12.4 ioctl Returns “Bad file descriptor”	2866
24.12.5 sysctl dev.myfirst.0 Shows the Tree but Reads Return “operation not supported”	2866
24.12.6 sysctl -w dev.myfirst.0.foo=value Fails With “permission denied”	2866
24.12.7 sysctl Hangs or Causes a Deadlock	2866
24.12.8 kldunload myfirst Fails With “Device busy”	2867
24.12.9 kldunload myfirst Causes a Kernel Panic	2867
24.12.10 Trace Probes Are Not Visible	2867
24.12.11 sysctl Tree Survives Unload But Hangs the Next Sysctl	2867
24.12.12 General Diagnostic Checklist	2868
24.12.13 Frequently Asked Questions	2868
24.13 Wrapping Up	2870
24.14 Bridge to Chapter 25	2872
24.15 Reference Card and Glossary	2872
24.15.1 Quick Reference: cdev Construction	2873
24.15.2 Quick Reference: cdevsw Flags	2873
24.15.3 Quick Reference: make_dev Flags (MAKEDEV_*)	2873
24.15.4 Quick Reference: ioctl Encoding Macros	2874
24.15.5 Quick Reference: d_ioctl_t Signature	2874
24.15.6 Quick Reference: sysctl OID Macros	2875
24.15.7 Quick Reference: sysctl Flag Bits	2875
24.15.8 Quick Reference: sysctl Type Bits	2875
24.15.9 Quick Reference: sysctl Handler Format Strings	2876
24.15.10 Quick Reference: sysctl Handler Boilerplate	2876
24.15.11 Quick Reference: per-Device Sysctl Context	2877
24.15.12 Quick Reference: Loader Tunables	2877
24.15.13 Quick Reference: ifnet Lifecycle	2877
24.15.14 Quick Reference: CAM SIM Lifecycle	2878
24.15.15 Quick Reference: Module Lifecycle	2878
24.15.16 Quick Reference: Event Handler	2879
24.15.17 Quick Reference: Errno Conventions	2879
24.15.18 Quick Reference: Debug Class Bits (from Chapter 23)	2879
24.15.19 Glossary of Integration Vocabulary	2880
24.15.20 Companion File Inventory	2882
24.15.21 Where the Chapter’s Source Tree Sits in Real FreeBSD	2883
24.15.22 What Did Not Make It Into the Chapter	2884
24.15.23 Reader Self-Check	2885
24.15.24 Final Word	2885
25 Advanced Topics and Practical Tips	2886
25.1 Reader Guidance & Outcomes	2886
25.1.1 Why Maintenance Discipline Earns a Chapter of Its Own	2887
25.1.2 Where Chapter 24 Left the Driver	2888
25.1.3 What You Will Learn	2888
25.1.4 What This Chapter Does Not Cover	2889
25.1.5 Estimated Time Investment	2890
25.1.6 Prerequisites	2890
25.1.7 How to Get the Most Out of This Chapter	2891

25.1.8	Roadmap Through the Chapter	2891
25.2	Section 1: Rate Limiting and Logging Etiquette	2893
25.2.1	The Problem With Unrestricted Logging	2893
25.2.2	Three Categories of Driver Log Message	2894
25.2.3	Introducing <code>ratecheck</code> and <code>ppsratecheck</code>	2894
25.2.4	A Simple Rate-Limited Log Macro	2895
25.2.5	Cooperating With the Chapter 23 Debug Mask	2897
25.2.6	<code>log(9)</code> Priority Levels	2897
25.2.7	The Kernel Message Buffer and What It Costs	2898
25.2.8	A Worked Example: The <code>myfirst</code> Read Path	2898
25.2.9	Common Mistakes and How to Avoid Them	2900
25.2.10	When Not to Rate-Limit	2900
25.2.11	What the <code>myfirst</code> Driver Now Contains	2900
25.2.12	Wrapping Up Section 1	2901
25.3	Section 2: Error Reporting and Return Conventions	2901
25.3.1	Why <code>Errno</code> Discipline Matters	2901
25.3.2	The Small Vocabulary	2902
25.3.3	The <code>ioctl</code> Dispatcher Audit	2904
25.3.4	The Read and Write Path Audit	2905
25.3.5	The Open and Close Path Audit	2905
25.3.6	The Attach and Detach Path Audit	2905
25.3.7	The <code>Sysctl</code> Handler Audit	2906
25.3.8	Error Path Messages	2906
25.3.9	Module Event Handler Conventions	2907
25.3.10	An <code>Errno</code> Checklist	2908
25.3.11	Wrapping Up Section 2	2909
25.4	Section 3: Driver Configuration via Loader Tunables and <code>sysctl</code>	2909
25.4.1	The Difference Between a Tunable and a <code>Sysctl</code>	2909
25.4.2	The <code>TUNABLE_*_FETCH</code> Family	2910
25.4.3	Per-Driver Versus Per-Instance Tunables	2911
25.4.4	The <code>CTLFLAG_TUN</code> Flag	2912
25.4.5	Declaring a Tunable as a Static <code>Sysctl</code>	2912
25.4.6	A Small Side Note on <code>SYSCTL_INT</code> Versus <code>SYSCTL_ADD_INT</code>	2913
25.4.7	Documenting a Tunable	2913
25.4.8	A Worked Example: <code>hw.myfirst.timeout_sec</code>	2913
25.4.9	Range Checks and Validation	2914
25.4.10	When to Expose a Tunable and When to Leave It Internal	2915
25.4.11	A Cautionary Note on <code>CTLFLAG_RWTUN</code> for Strings	2916
25.4.12	Tunables Versus Kernel Modules: Where They Live	2916
25.4.13	Wrapping Up Section 3	2916
25.5	Section 4: Versioning and Compatibility Strategies	2917
25.5.1	The Three Version Numbers	2917
25.5.2	Why Users Need to Query the Version	2918
25.5.3	Adding a New <code>ioctl</code> Without Breaking Older Callers	2918
25.5.4	Retiring a Deprecated <code>ioctl</code>	2919
25.5.5	The Capability Bitmask Pattern	2919
25.5.6	<code>Sysctl</code> Deprecation	2920
25.5.7	User-Visible Behaviour Changes	2921
25.5.8	<code>MODULE_DEPEND</code> and Inter-Module Compatibility	2921
25.5.9	A Worked Example: The 1.7 to 1.8 Transition	2922
25.5.10	Common Mistakes in Versioning	2923

25.5.11	Wrapping Up Section 4	2923
25.6	Section 5: Managing Resources in Failure Paths	2923
25.6.1	The Problem: Nested <code>if</code> Paths Do Not Scale	2924
25.6.2	The <code>goto fail;</code> Pattern	2925
25.6.3	Why Linear Unwinding Is the Right Shape	2926
25.6.4	Label Naming Conventions	2927
25.6.5	The Rule of Falling Through	2927
25.6.6	What the Success Path Looks Like	2928
25.6.7	When an Acquisition Cannot Fail	2929
25.6.8	Helpers to Reduce Duplication	2929
25.6.9	Detach as the Mirror of Attach	2930
25.6.10	Deliberate Failure Injection for Testing	2930
25.6.11	A Complete <code>myfirst_attach</code> for Chapter 25	2931
25.6.12	Common Mistakes in Failure Paths	2933
25.6.13	Wrapping Up Section 5	2934
25.7	Section 6: Modularization and Separation of Concerns	2935
25.7.1	Why Split Files	2935
25.7.2	A File Layout for <code>myfirst</code>	2935
25.7.3	The Single-Responsibility Rule	2936
25.7.4	Public vs Private Headers	2936
25.7.5	The Anatomy of <code>myfirst.c</code> After the Split	2938
25.7.6	<code>myfirst_cdev.c</code> : The Character Device Callbacks	2940
25.7.7	<code>myfirst_ioctl.c</code> : The Command Switch	2941
25.7.8	<code>myfirst_log.h</code> and <code>myfirst_log.c</code> : Rate-Limited Logging	2942
25.7.9	The Updated Makefile	2943
25.7.10	Where to Draw Each File Boundary	2944
25.7.11	The Include Graph and Build Ordering	2945
25.7.12	The Cost of Splitting	2945
25.7.13	A Practical Procedure for Splitting a Real Driver	2945
25.7.14	Common Mistakes When Splitting	2946
25.7.15	Wrapping Up Section 6	2946
25.8	Section 7: Preparing for Production Use	2947
25.8.1	The Production Readiness Mindset	2947
25.8.2	Declaring Module Dependencies	2947
25.8.3	Publishing PNP Information	2948
25.8.4	The <code>MOD_QUIESCE</code> Event	2948
25.8.5	Emitting <code>devctl_notify</code> Events	2948
25.8.6	Writing a <code>MAINTENANCE.md</code>	2949
25.8.7	A <code>devd</code> Rule Set	2951
25.8.8	Logs: Friend of the Support Engineer	2952
25.8.9	Handling Bus Attach/Detach Gracefully	2952
25.8.10	Handling Unexpected Operator Actions	2952
25.8.11	A Production-Readiness Checklist	2953
25.8.12	What the <code>myfirst</code> Driver Covers	2954
25.8.13	Wrapping Up Section 7	2954
25.9	Section 8: <code>SYSINIT</code> , <code>SYSUNINIT</code> , and <code>EVENTHANDLER</code>	2955
25.9.1	Why the Kernel Needs Boot-Time Ordering	2955
25.9.2	The Subsystem ID Space	2955
25.9.3	When a Driver Needs <code>SYSINIT</code>	2956
25.9.4	The Shape of a <code>SYSINIT</code> Declaration	2956
25.9.5	A Worked <code>SYSINIT</code> Example for <code>myfirst</code>	2957

25.9.6	The Ordering Between SYSINIT and DRIVER_MODULE	2958
25.9.7	EVENTHANDLER: Reacting to Runtime Events	2958
25.9.8	The Shape of an EVENTHANDLER Registration	2959
25.9.9	EVENTHANDLER in the Attach Chain	2960
25.9.10	Priority: SHUTDOWN_PRI_*	2960
25.9.11	When to Use vm_lowmem	2961
25.9.12	power_suspend_early and power_resume	2961
25.9.13	The Module Event Handler	2962
25.9.14	Deregistration Discipline	2963
25.9.15	A Complete Lifecycle Example	2963
25.9.16	Deciding What to Register For	2963
25.9.17	Common Mistakes with SYSINIT and EVENTHANDLER	2964
25.9.18	Wrapping Up Section 8	2964
25.10	Hands-on Labs	2965
25.10.1	Lab 1: Reproducing a Log Flood	2965
25.10.2	Lab 2: errno Audit with truss	2966
25.10.3	Lab 3: Tunable Reboot Behaviour	2967
25.10.4	Lab 4: Deliberate Attach Failure Injection	2968
25.10.5	Lab 5: shutdown_pre_sync Handler	2969
25.10.6	Lab 6: The 100-Cycle Regression Script	2970
25.10.7	Lab 7: Capability Discovery in User Space	2971
25.10.8	Lab 8: Sysctl Range Validation	2972
25.10.9	Lab 9: Log Message Audit Across the Driver	2974
25.10.10	Lab 10: Multi-Version Compatibility Matrix	2975
25.11	Challenge Exercises	2977
25.11.1	Challenge 1: Per-Class Rate Limits	2977
25.11.2	Challenge 2: A Writable String sysctl	2978
25.11.3	Challenge 3: A MOD_QUIESCE Handler Separate from Detach	2978
25.11.4	Challenge 4: A Drain-Based Detach Instead of EBUSY	2979
25.11.5	Challenge 5: A Sysctl-Driven Version Bump Check	2979
25.11.6	Challenge 6: Add a Sysctl That Enumerates Open File Descriptors	2980
25.11.7	Challenge 7: A Second EVENTHANDLER for vm_lowmem	2980
25.11.8	Challenge 8: A MYFIRSTIOC_GETSTATS ioctl Returning a Structured Payload	2980
25.11.9	Challenge 9: Devctl-Based Live Monitoring	2981
25.12	Troubleshooting Guide	2981
25.12.1	Symptom: kldload returns “Exec format error”	2981
25.12.2	Symptom: kldload returns “No such file or directory” for an obvious-looking file	2982
25.12.3	Symptom: kldload succeeds but kldstat does not show the module	2982
25.12.4	Symptom: kldload reports “module busy”	2982
25.12.5	Symptom: sysctl dev.myfirst.0.debug.mask=0x4 returns “Operation not permitted”	2982
25.12.6	Symptom: A new sysctl does not appear in the tree	2982
25.12.7	Symptom: A tunable seems to be ignored	2983
25.12.8	Symptom: A log message should fire but does not appear in dmesg	2983
25.12.9	Symptom: A log message appears once and is then silent forever	2983
25.12.10	Symptom: Attach fails and the cleanup chain does not release a resource	2983
25.12.11	Symptom: Detach panics with a “witness” warning	2983
25.12.12	Symptom: The driver panics at module unload with a dangling pointer	2984
25.12.13	Symptom: MYFIRSTIOC_GETVER returns an unexpected value	2984
25.12.14	Symptom: devctl_notify events never appear in devd.log	2984
25.12.15	Symptom: The 100-cycle regression script grows the kernel’s memory footprint	2984
25.12.16	Symptom: Two concurrent MYFIRSTIOC_SETMSG calls interleave their writes	2984

25.12.1	Symptom: A sysctl value resets on every module load	2985
25.12.1	Symptom: MYFIRSTIOC_GETCAPS returns a value without the bits you just added	2985
25.12.1	Symptom: A SYSINIT fires before the kernel's allocator is ready	2985
25.12.2	Symptom: A handler registered with EVENTHANDLER_PRI_FIRST still runs late	2985
25.12.2	Symptom: dmesg does not show any output from the driver	2985
25.13	Quick Reference	2985
25.13.1	Rate-limited logging	2986
25.13.2	Errno vocabulary	2986
25.13.3	Tunable families	2987
25.13.4	Sysctl flag summary	2987
25.13.5	Version identifiers	2987
25.13.6	Capability bits	2987
25.13.7	Labelled-cleanup skeleton	2988
25.13.8	File layout for a modular driver	2988
25.13.9	Production checklist	2989
25.13.10	SYSINIT subsystem IDs	2989
25.13.11	Shutdown event priorities	2990
25.13.12	EVENTHANDLER skeleton	2990
25.13.13	Tunable name ladders	2990
25.13.14	ioctl command ladder	2991
25.14	Real-World Driver Walkthroughs	2991
25.14.1	mmsd(4): Rate-Limited Error Logging on a Hot Path	2991
25.14.2	uftdi(4): Module Dependencies and PNP Metadata	2992
25.14.3	iscsi(4): Shutdown Handler Registered in attach	2993
25.14.4	ufs_dirhash: Cache Eviction on vm_lowmem	2993
25.14.5	tcp_subr: vm_lowmem Without a Softc	2994
25.14.6	if_vtnet: Rate-Limited Logging Around a Specific Fault	2994
25.14.7	vt_core: Multiple EVENTHANDLER Registrations in Sequence	2995
25.14.8	FFS (ffs_alloc.c): Per-Condition Rate-Limit Buckets	2995
25.14.9	Reading More Drivers	2996
25.14.10	What You Will Not See	2997
25.14.11	Wrapping Up the Walkthroughs	2997
25.15	Wrapping Up	2998
25.16	Part 5 Checkpoint	2999
25.17	Bridge to Chapter 26	3000
25.18	Glossary	3000
26	USB and Serial Drivers	3003
26.1	Introduction	3003
26.1.1	Where Chapter 25 Left the Driver	3005
26.1.2	What You Will Learn	3005
26.1.3	What This Chapter Does Not Cover	3006
26.1.4	Where We Are in Part 6	3007
26.1.5	A Small Note on Difficulty	3008
26.2	Reader Guidance: How to Use This Chapter	3008
26.2.1	Recommended Pacing	3009
26.2.2	What a Good Study Session Looks Like	3009
26.2.3	If You Do Not Have a USB Device to Test With	3009
26.3	How to Get the Most Out of This Chapter	3009
26.3.1	What to Do When Something Does Not Work	3010
26.3.2	Roadmap Through the Chapter	3011

26.4	Section 1: Understanding USB and Serial Device Fundamentals	3012
26.4.1	What a Transport Is, and Why It Matters Here	3012
26.4.2	The USB Mental Model	3013
26.4.3	The Serial Mental Model	3020
26.4.4	Exercise: Use <code>usbconfig</code> and <code>dmesg</code> to Explore USB Devices on Your System	3025
26.4.5	Wrapping Up Section 1	3027
26.5	Section 2: Writing a USB Device Driver	3027
26.5.1	Moving from Concepts to Code	3027
26.5.2	Where a USB Driver Sits in the FreeBSD Tree	3028
26.5.3	The Match Table: Telling the Kernel Which Devices Are Yours	3029
26.5.4	The <code>probe</code> Method	3030
26.5.5	The <code>attach</code> Method	3031
26.5.6	The Softc: Per-Device State	3033
26.5.7	The <code>detach</code> Method	3034
26.5.8	Registration Macros	3035
26.5.9	The Hot-Plug Lifecycle, Revisited in Code	3037
26.5.10	Wrapping Up Section 2	3037
26.5.11	Reading <code>uled.c</code> As a Complete Example	3038
26.6	Section 3: Performing USB Data Transfers	3039
26.6.1	The Transfer Configuration Array	3039
26.6.2	Setting Up and Tearing Down Transfers	3041
26.6.3	The Callback State Machine	3042
26.6.4	The Write Callback	3044
26.6.5	Control Transfers	3045
26.6.6	Interrupt Transfers	3046
26.6.7	Frame-Level Operations: What the Framework Gives You	3048
26.6.8	The <code>usbd_copy_in</code> and <code>usbd_copy_out</code> Helpers	3048
26.6.9	Starting, Stopping, and Querying Transfers	3048
26.6.10	A Worked Example: Echo-Loop over USB	3049
26.6.11	Putting the Whole Echo-Loop Driver Together	3050
26.6.12	Choosing Between <code>usb_fifo</code> and a Custom <code>cdevsw</code>	3055
26.6.13	Error Handling and Retry Policy	3055
26.6.14	Timeouts and Their Consequences	3055
26.6.15	What Goes Wrong When Transfer Setup Fails	3056
26.6.16	A Detail About <code>pipe_bof</code>	3057
26.6.17	<code>short_xfer_ok</code> and Data-Length Semantics	3057
26.6.18	Locking Rules Around Transfers	3058
26.6.19	Wrapping Up Section 3	3058
26.7	Section 4: Writing a Serial UART Driver	3059
26.7.1	From USB to UART: A Shift of Landscape	3059
26.7.2	Where the <code>uart(4)</code> Framework Lives	3060
26.7.3	The <code>uart_class</code> Structure	3060
26.7.4	The <code>uart_ops</code> Structure	3061
26.7.5	A Closer Look at <code>ns8250</code>	3062
26.7.6	The <code>uart_softc</code> and How the Framework Uses It	3064
26.7.7	Integration with the TTY Layer	3064
26.7.8	The <code>termios</code> Interface in Practice	3065
26.7.9	Flow Control at the Register Level	3065
26.7.10	The Interrupt Handler Path in Detail	3066
26.7.11	The TX and RX Ring Buffers	3066
26.7.12	Debugging Serial Drivers	3067

26.7.13	Writing a UART Driver Yourself	3068
26.7.14	Looking at <code>myfirst_uart.c</code> in Skeleton Form	3068
26.7.15	The <code>ucom(4)</code> Framework: USB-to-Serial Bridges	3070
26.7.16	Reading <code>uftdi.c</code> As a Complete Example	3071
26.7.17	Choosing Between <code>uart(4)</code> and <code>ucom(4)</code>	3073
26.7.18	Differences Between Chip Variants	3073
26.7.19	The Console Path	3074
26.7.20	Comparing UART Drivers Across Architectures	3074
26.7.21	What About the USB CDC ACM Standard?	3075
26.7.22	Wrapping Up Section 4	3075
26.8	Section 5: Testing USB and Serial Drivers Without Real Hardware	3076
26.8.1	Why This Section Exists	3076
26.8.2	The <code>nmdm(4)</code> Null-Modem Driver	3076
26.8.3	The <code>cu(1)</code> , <code>tip(1)</code> , and <code>stty(1)</code> Toolbox	3077
26.8.4	The <code>comcontrol(8)</code> Utility	3077
26.8.5	The <code>usbconfig(8)</code> Utility	3078
26.8.6	Testing USB Drivers with QEMU	3078
26.8.7	FreeBSD USB Gadget Mode	3079
26.8.8	Techniques That Do Not Require Special Tooling	3079
26.8.9	A Walkthrough of QEMU USB Redirection	3080
26.8.10	Using <code>devd(8)</code> During Development	3081
26.8.11	Writing a Simple Userland Test Harness	3082
26.8.12	Logging Patterns for Development	3083
26.8.13	A Test-Driven Workflow for Chapter 26	3084
26.8.14	Wrapping Up Section 5	3084
26.9	Common Patterns Across Transport Drivers	3084
26.9.1	Pattern 1: Match Table, Probe, Attach, Detach	3084
26.9.2	Pattern 2: Softc As The Single Source of Per-Device State	3085
26.9.3	Pattern 3: Labelled-Goto Cleanup Chain	3085
26.9.4	Pattern 4: Frameworks Hide Transport Details	3085
26.9.5	Pattern 5: Callbacks with State Machines	3086
26.9.6	Pattern 6: Mutexes and Wakeups	3086
26.9.7	Pattern 7: Errno-Returning Helpers	3086
26.9.8	Pattern 8: Version Declarations and Module Dependencies	3086
26.9.9	Pattern 9: Cross-Framework Layering	3087
26.9.10	Pattern 10: Early Attach Deferral	3087
26.9.11	Pattern 11: Separate Data Path and Control Path	3087
26.9.12	Pattern 12: Reference Drivers	3088
26.10	Hands-On Labs	3088
26.10.1	Lab 1: Exploring a USB Device with <code>usbconfig</code>	3088
26.10.2	Lab 2: Building and Loading the USB Driver Skeleton	3089
26.10.3	Lab 3: A Bulk Loopback Test	3090
26.10.4	Lab 4: A Simulated Serial Driver with <code>nmdm(4)</code>	3091
26.10.5	Lab 5: Talking to a Real USB-to-Serial Adapter	3092
26.10.6	Lab 6: Observing Hot-Plug Lifecycle	3093
26.10.7	Lab 7: Building a <code>ucom(4)</code> Skeleton from Scratch	3094
26.10.8	Lab 8: Troubleshooting a Hung TTY Session	3096
26.10.9	Lab 9: Monitoring USB Transfer Statistics	3097
26.11	Challenge Exercises	3097
26.11.1	Challenge 1: Add a Third USB Endpoint Type	3098
26.11.2	Challenge 2: Write a Minimal USB Gadget Driver	3098

26.11.3 Challenge 3: A Custom <code>termios</code> Flag Handler	3098
26.11.4 Challenge 4: Parsing a Small USB Protocol	3099
26.11.5 Challenge 5: Robustness Under Load	3099
26.11.6 Challenge 6: Implement Suspend/Resume Properly	3099
26.11.7 Challenge 7: Adding <code>poll(2)</code> Support	3100
26.11.8 Challenge 8: Writing a Character-Counter <code>ioctl</code>	3100
26.12 Troubleshooting Guide	3100
26.12.1 The Module Will Not Load	3100
26.12.2 The Driver Loads but Never Attaches	3101
26.12.3 The Driver Attaches but <code>/dev</code> Node Does Not Appear	3101
26.12.4 The Driver Panics on Detach	3101
26.12.5 Transfers Stall	3102
26.12.6 Serial Characters Garbled	3102
26.12.7 A Process is Stuck in <code>read(2)</code> and Will Not Exit	3102
26.12.8 High CPU Usage When Idle	3102
26.12.9 <code>usbconfig</code> Shows the Device but <code>dmesg</code> Is Silent	3103
26.12.10 Debugging a Deadlock with <code>WITNESS</code>	3103
26.12.11A Driver That Appears Twice for the Same Device	3103
26.12.12USB Serial Baud Rate Does Not Take Effect	3104
26.12.13A Kernel Panic with “Spin lock held too long”	3104
26.12.14A Device Never Completes Enumeration	3104
26.12.15Diagnostic Checklist When You Are Stuck	3104
26.13 Reading the FreeBSD USB Source Tree: A Guided Tour	3105
26.13.1 Tour 1: <code>uled.c</code> , the Simplest USB Driver	3105
26.13.2 Tour 2: <code>ugold.c</code> , Adding Interrupt Transfers	3106
26.13.3 Tour 3: <code>udbp.c</code> , Bidirectional Bulk Transfers	3106
26.13.4 Tour 4: <code>uplcom.c</code> , a USB-to-Serial Bridge	3107
26.13.5 Tour 5: <code>uhid.c</code> , the Human Interface Device Driver	3107
26.13.6 How to Study a Driver You Have Never Seen	3108
26.13.7 Where to Go Beyond the Tour	3108
26.14 Performance Considerations for Transport Drivers	3109
26.14.1 The USB Bus as a Shared Resource	3109
26.14.2 Latency: What Costs Microseconds, What Costs Milliseconds	3109
26.14.3 When to Rearm: The Classic USB Tradeoff	3110
26.14.4 Buffer Sizing	3110
26.14.5 UART Performance	3110
26.14.6 Concurrency and Lock Hold Times	3111
26.14.7 Measuring, Not Guessing	3111
26.15 Common Mistakes When Writing Your First Transport Driver	3111
26.15.1 Mistake 1: Taking the Framework Mutex Explicitly in a Callback	3111
26.15.2 Mistake 2: Calling Framework Primitives Without the Mutex Held	3112
26.15.3 Mistake 3: Forgetting <code>USB_ERR_CANCELLED</code> Handling	3113
26.15.4 Mistake 4: Submitting to a Channel That Has Not Been Started	3113
26.15.5 Mistake 5: Assuming <code>USB_GET_STATE</code> Returns the Real Hardware State	3113
26.15.6 Mistake 6: Using <code>M_WAITOK</code> in a Callback	3113
26.15.7 Mistake 7: Forgetting <code>MODULE_DEPEND</code> on <code>usb</code>	3114
26.15.8 Mistake 8: Mutable State in a Read-Only Path	3114
26.15.9 Mistake 9: Stale Pointers After <code>usbd_transfer_unsetup</code>	3115
26.15.10 Mistake 10: Not Zeroing the Softc’s detaching Flag at Attach Time	3115
26.15.11 Mistake 11: Forgetting to Destroy the Device Node on Detach	3115
26.15.12 Mistake 12: Racing on the Character Device Open	3115

26.15.13	Mistake 13: Overlooking the TTY Layer’s Own Locking	3116
26.15.14	Mistake 14: Not Handling Zero-Length Data in Read or Write	3116
26.15.15	Mistake 15: Copying Data Before Checking the Transfer Status	3116
26.15.16	Mistake 16: Assuming <code>termios</code> Values Are in Standard Units	3117
26.15.17	Mistake 17: Missing <code>device_set_desc</code> or <code>device_set_desc_copy</code>	3117
26.15.18	Mistake 18: <code>device_printf</code> in the Data Path Without Rate Limiting	3117
26.15.19	Mistake 19: Not Waking Readers on Device Removal	3118
26.15.20	Mistake 20: Thinking “It Works on My Machine” Is Enough	3118
26.15.21	What To Do After You Make One of These Mistakes	3118
26.16	Wrapping Up	3119
26.17	Bridge to Chapter 27	3120
26.18	Quick Reference	3120
26.18.1	USB Driver APIs	3120
26.18.2	USB Transfer Types (<code>usb.h</code>)	3121
26.18.3	USB Transfer Direction	3121
26.18.4	USB Callback States (<code>usbdi.h</code>)	3122
26.18.5	USB Error Codes (<code>usbdi.h</code>)	3122
26.18.6	Registration Macros	3122
26.18.7	UART Framework APIs	3122
26.18.8	<code>uart_ops</code> Methods	3123
26.18.9	<code>ucom_callback</code> Methods	3123
26.18.10	Key Source Files	3123
26.18.11	Userland Diagnostic Commands	3124
26.18.12	Standard Development Flags	3124
26.19	Glossary	3124
26.19.1	A Closing Note on Terminology Precision	3129
27	Working with Storage Devices and the VFS Layer	3130
27.1	Introduction	3130
27.2	How Part 6 Differs from Parts 1 Through 5	3131
27.3	Reader Guidance: How to Use This Chapter	3132
27.3.1	Work Section by Section	3133
27.3.2	Type the Code	3133
27.3.3	Open the FreeBSD Source Tree	3133
27.3.4	Use Your Lab Logbook	3133
27.3.5	Pace Yourself	3133
27.4	How to Get the Most Out of This Chapter	3133
27.4.1	Work in Sections	3134
27.4.2	Keep the Driver Running	3134
27.4.3	Consult Manual Pages	3134
27.4.4	Type the Code, Then Mutate It	3134
27.4.5	Trust the Tooling	3134
27.4.6	Take Breaks	3134
27.5	Section 1: What Is the Virtual File System Layer?	3135
27.5.1	The Problem VFS Solves	3135
27.5.2	The VFS Object Model	3135
27.5.3	Where Storage Drivers Fit	3136
27.5.4	VFS in the Kernel Source	3137
27.5.5	What This Means for Our Driver	3137
27.5.6	The VFS Call Chain in Practice	3137
27.5.7	Wrapping Up Section 1	3138

27.6	Section 2: devfs vs VFS	3138
27.6.1	What devfs Is	3138
27.6.2	What VFS Is	3139
27.6.3	Why Storage Drivers Live on Both Sides	3139
27.6.4	A Closer Look at the Buffer Cache	3139
27.6.5	The Read and Write Paths	3140
27.6.6	Why This Distinction Matters	3141
27.6.7	A Concrete Example: /dev/null and /dev/ada0	3141
27.6.8	How We Will Proceed	3142
27.6.9	Wrapping Up Section 2	3142
27.7	Section 3: Registering a Pseudo Block Device	3142
27.7.1	The g_disk API	3142
27.7.2	A Minimum Skeleton	3144
27.7.3	The Makefile	3147
27.7.4	Loading and Inspecting the Skeleton	3148
27.7.5	Why the Failures Are Expected	3149
27.7.6	How Things Fit Together	3150
27.7.7	Wrapping Up Section 3	3150
27.8	Section 4: Exposing a GEOM-Backed Provider	3150
27.8.1	GEOM in One Page	3151
27.8.2	A Concrete Example of Composition	3151
27.8.3	Providers and Consumers in Code	3151
27.8.4	Access Rights	3152
27.8.5	The BIO Object and Its Life Cycle	3152
27.8.6	The GEOM Topology Lock	3153
27.8.7	The GEOM Event Queue	3153
27.8.8	How g_disk Wraps All of This	3153
27.8.9	Where to Read Source	3154
27.8.10	Comparing md(4) and g_zero	3154
27.8.11	Why Our Driver Uses g_disk	3155
27.8.12	A Walk Through g_class	3155
27.8.13	The Life of a BIO, In Detail	3156
27.8.14	Error Propagation	3156
27.8.15	What g_disk Does That You Do Not See	3157
27.8.16	Inspecting Our Provider	3157
27.8.17	Wrapping Up Section 4	3158
27.9	Section 5: Implementing Basic Read and Write	3158
27.9.1	The Backing Store	3159
27.9.2	The Strategy Function	3160
27.9.3	The Role of bio_resid	3163
27.9.4	The Lock	3163
27.9.5	Rebuilding and Reloading	3163
27.9.6	A Quick Look Under Load	3164
27.9.7	Refining the Driver with ioctl Support	3164
27.9.8	Refining the Driver with getattr Support	3165
27.9.9	Partial Writes and Short Reads	3166
27.9.10	Common Mistakes in Strategy Functions	3166
27.9.11	Chained BIOs and BIO Hierarchies	3166
27.9.12	The Thread Context of the Strategy Function	3167
27.9.13	A Worked Example: Reading Across a Boundary	3167
27.9.14	The Devstat Side Effect	3168

27.9.15 Wrapping Up Section 5	3168
27.10 Section 6: Mounting a Filesystem on the Device	3168
27.10.1 The Plan	3169
27.10.2 Increasing the Media Size	3169
27.10.3 Formatting with newfs_ufs	3170
27.10.4 Mounting the Filesystem	3170
27.10.5 Creating and Reading Files	3171
27.10.6 Watching the Traffic	3171
27.10.7 Unmounting	3172
27.10.8 Attempting to Unload While Mounted	3173
27.10.9 A Brief Anatomy of UFS on Top of Our Driver	3173
27.10.10 How Mount Actually Works	3174
27.10.11 The GEOM Character Interface	3174
27.10.12 What Filesystems Need from a Block Driver	3175
27.10.13 Raw Access Versus Filesystem Access, Visualised	3175
27.10.14 Watching the Request Path with DTrace	3176
27.10.15 Wrapping Up Section 6	3176
27.11 Section 7: Persistence and In-Memory Backing Stores	3177
27.11.1 Why Persistence Is Hard	3177
27.11.2 The Options	3177
27.11.3 Saving and Restoring the Buffer	3178
27.11.4 The Contract With Upper Layers	3178
27.11.5 What md(4) Does Differently	3179
27.11.6 A Note on Swap-Backed Memory	3179
27.11.7 A Note on Preloaded Images	3180
27.11.8 A Small Extension: Persisting Across Module Reloads Only	3180
27.11.9 Sketching a Vnode-Backed Strategy Function	3181
27.11.10 Wrapping Up Section 7	3184
27.12 Section 8: Safe Unmount and Cleanup	3184
27.12.1 The Expected Teardown Sequence	3184
27.12.2 Step 1: Unmount	3185
27.12.3 Step 2: Close Raw Access	3185
27.12.4 Step 3: kldunload	3185
27.12.5 Step 4: disk_destroy	3185
27.12.6 Step 5 to 7: Withering	3186
27.12.7 Step 8 to 9: Free Resources	3186
27.12.8 Step 10: Module Unload	3187
27.12.9 What Can Go Wrong	3187
27.12.10 The d_gone Callback	3187
27.12.11 Testing the Teardown	3188
27.12.12 The Idempotency Principle	3188
27.12.13 Ordering and Reverse Order	3189
27.12.14 The MOD_QUIESCE Event	3189
27.12.15 Coordinating With Future Worker Threads	3190
27.12.16 Cleanup in the Face of Errors	3190
27.12.17 Wrapping Up Section 8	3191
27.13 Section 9: Refactoring and Versioning	3191
27.13.1 Multi-Unit Support	3191
27.13.2 Ioctl Surface for Runtime Configuration	3193
27.13.3 Splitting the Source File	3193
27.13.4 Versioning	3194

27.13.5 Debugging and Observability Helpers	3194
27.13.6 Naming Stability	3195
27.13.7 Keeping the Teaching Driver Simple	3195
27.13.8 A Short Design Patterns Recap	3195
27.13.9 Wrapping Up Section 9	3196
27.14 Section 10: Observability and Measuring Your Driver	3196
27.14.1 gstat	3197
27.14.2 iostat	3197
27.14.3 diskinfo	3198
27.14.4 sysctl	3199
27.14.5 vmstat	3199
27.14.6 procstat	3199
27.14.7 DTrace for the Block Layer	3200
27.14.8 kgdb and Crash Dumps	3201
27.14.9 ktrace	3201
27.14.10 mesg and the Kernel Log	3201
27.14.11 A Simple Measurement Recipe	3202
27.14.12 Comparing Against md(4)	3202
27.14.13 Wrapping Up Section 10	3202
27.15 Case Studies in Real FreeBSD Storage Code	3203
27.15.1 Case Study 1: g_zero.c	3203
27.15.2 Case Study 2: md.c, the Malloc Strategy Function	3204
27.15.3 Case Study 3: md.c, the Module Load Path	3205
27.15.4 Case Study 4: The newbus Side of a Real Storage Driver	3206
27.15.5 Takeaways from the Case Studies	3207
27.16 Hands-On Labs	3208
27.16.1 Lab 1: Explore GEOM on a Running System	3208
27.16.2 Lab 2: Build the Skeleton Driver	3209
27.16.3 Lab 3: Implement the BIO Handler	3209
27.16.4 Lab 4: Increase Size and Mount UFS	3210
27.16.5 Lab 5: Observing Real Cross-Reload Persistence with md(4)	3211
27.16.6 Lab 6: Safe Unmount Under Load	3212
27.16.7 Lab 7: Observing BIO Traffic with DTrace	3212
27.16.8 Lab 8: Adding a getattr Attribute	3213
27.16.9 Lab 9: Exploring md(4) for Comparison	3213
27.16.10 Lab 10: Break It On Purpose	3214
27.16.11 Lab 11: Measure Under Varying Block Sizes	3214
27.16.12 Lab 12: Race Test With Two Processes	3215
27.16.13 A Word on Lab Discipline	3215
27.17 Challenge Exercises	3215
27.17.1 Challenge 1: Expose a Read-Only Mode	3216
27.17.2 Challenge 2: Implement a Second Unit	3216
27.17.3 Challenge 3: Honour BIO_DELETE with a Sysctl Counter	3216
27.17.4 Challenge 4: Respond to BIO_GETATTR for rotation_rate	3216
27.17.5 Challenge 5: Resizing the Device	3216
27.17.6 Challenge 6: A Write Counter and Rate Display	3217
27.17.7 Challenge 7: A Fixed Pattern Backing Store	3217
27.17.8 Challenge 8: Write a mdconfig-Like Control Utility	3217
27.17.9 Challenge 9: Survive a Simulated Crash	3217
27.17.10 Challenge 10: Understand md(4) Well Enough to Teach It	3217
27.17.11 When to Attempt Challenges	3218

27.18	Troubleshooting	3218
27.18.1	kldload Succeeds but No /dev Node Appears	3218
27.18.2	kldload Fails with “module version mismatch”	3218
27.18.3	diskinfo Prints the Wrong Size	3219
27.18.4	newfs_ufs Fails with “Device not configured”	3219
27.18.5	newfs_ufs Hangs	3219
27.18.6	mount Fails with “bad superblock”	3219
27.18.7	kldunload Hangs	3220
27.18.8	kldunload Returns “Device busy”	3220
27.18.9	Kernel Panic with “freeing free memory”	3220
27.18.10	Kernel Panic with “vm_fault: kernel mode”	3220
27.18.11	lsstat Shows No Activity	3221
27.18.12	Operation not supported” for DELETE	3221
27.18.13	dev/myblk0 Does Not Appear Until Several Seconds After kldload	3221
27.18.14	Data Written Is Readable but Garbled	3221
27.18.15	Backing Store Not Freed, Memory Grows With Every Reload	3222
27.18.16	lsstat Shows Very High Queue Length	3222
27.18.17	Strategy Function Called With NULL bio_disk	3222
27.18.18	Mysterious I/O Errors After Reload	3223
27.18.19	Access Count Stuck at Non-Zero After Unmount	3223
27.18.20	Driver Loaded But Not Visible in geom -t	3223
27.18.21	Panic on Second Load	3223
27.18.22	newfs_ufs Aborts with “File system too small”	3224
27.18.23	mount -o trim Does Not Trigger BIO_DELETE	3224
27.18.24	UFS Complains About “Fragment out of bounds”	3224
27.18.25	Kernel Printing “interrupt storm” Messages	3225
27.18.26	Reboot Hangs on Unmount During Shutdown	3225
27.18.27	General Advice	3225
27.19	Wrapping Up	3225
27.19.1	A Summary of the Key Moves	3226
27.19.2	A Reminder About Raw Access	3227
27.19.3	A Reminder About Safety	3227
27.19.4	Where to Look Next in the FreeBSD Tree	3227
27.20	Bridge to the Next Chapter	3227
27.21	Quick Reference	3228
27.21.1	Key Headers	3228
27.21.2	Key Structures	3228
27.21.3	Common BIO Commands	3229
27.21.4	Common GEOM Tools	3229
27.21.5	Key Callback Typedefs	3230
27.21.6	File and Path Reference	3230
27.21.7	Common Disk Flags	3231
27.21.8	Patterns for d_strategy	3231
27.21.9	Minimum Registration Sequence	3232
27.22	Glossary	3233
27.23	Frequently Asked Questions	3237
27.23.1	Do I need to support BIO_ORDERED?	3237
27.23.2	What is the relationship between d_maxsize and MAXPHYS?	3237
27.23.3	Can my driver issue BIOs to itself?	3237
27.23.4	Why do some fields in struct disk use u_int and others off_t?	3237
27.23.5	Is disk_alloc safe to call at any time?	3237

27.23.6	What happens if I call <code>disk_create</code> twice with the same name?	3237
27.23.7	Can the strategy function sleep?	3237
27.23.8	How do I know when all BIOs have finished for a given filesystem?	3238
27.23.9	Can I pass pointers between threads through <code>bio_caller1</code> or similar fields?	3238
27.23.10	My driver works on my laptop but not on the server. Why?	3238
27.23.11	Where does the <code>/dev</code> node's name actually come from?	3238
27.23.12	What is the maximum number of units I can create?	3238
27.23.13	Can I change <code>d_mediasize</code> after <code>disk_create</code> ?	3238
27.23.14	What happens if I forget <code>MODULE_DEPEND</code> ?	3238
27.23.15	Should I use <code>biodone</code> or <code>g_io_deliver</code> in my driver?	3239
27.23.16	How does <code>BIO_DELETE</code> relate to TRIM and UNMAP?	3239
27.23.17	Why does my strategy function sometimes receive a BIO with <code>bio_length</code> of zero?	3239
27.23.18	What is the difference between <code>d_flags</code> and <code>bio_flags</code> ?	3239
27.23.19	Can my driver present itself as removable media?	3239
27.23.20	What thread actually calls my strategy function?	3239
27.23.21	How do I know which process caused a given BIO?	3239
27.23.22	Can two different filesystems be mounted simultaneously on two unit numbers of my driver?	3240
27.23.23	Should my driver handle power management events?	3240
27.23.24	What is the practical impact of choosing 512 versus 4096 as <code>d_sectorsize</code> ?	3240
27.23.25	If I reload my driver many times, does memory leak?	3240
27.23.26	Why does <code>mount</code> sometimes succeed on my raw device but <code>newfs_ufs</code> fail?	3240
27.23.27	How do I verify that my <code>BIO_FLUSH</code> implementation actually makes data durable?	3240
27.23.28	What are the correct locking rules inside <code>d_strategy</code> ?	3241
27.23.29	Why is <code>BIO_FLUSH</code> not a percent-of-capacity barrier like write barriers on Linux?	3241
27.23.30	Are there any tools that generate BIO traffic to help me test?	3241
27.24	What This Chapter Did Not Cover	3241
27.25	Final Reflection	3242
27.26	Further Reading	3242
28	Writing a Network Driver	3244
28.1	Introduction	3244
28.2	Reader Guidance: How to Use This Chapter	3245
28.2.1	Work Section by Section	3246
28.2.2	Type the Code	3246
28.2.3	Open the FreeBSD Source Tree	3246
28.2.4	Use Your Lab Logbook	3246
28.2.5	Pace Yourself	3247
28.3	How to Get the Most Out of This Chapter	3247
28.3.1	Work in Sections	3247
28.3.2	Keep the Driver Running	3247
28.3.3	Consult Manual Pages	3247
28.3.4	Type the Code, Then Mutate It	3247
28.3.5	Trust the Tooling	3248
28.3.6	Take Breaks	3248
28.4	Section 1: What a Network Driver Does	3248
28.4.1	A Packet, in the Kernel	3248
28.4.2	A Transport	3249
28.4.3	The Network Stack	3249
28.4.4	How a Network Driver Differs From a Character Driver	3250
28.4.5	How a Network Driver Differs From a Storage Driver	3250
28.4.6	The Role of <code>ifconfig</code> , <code>netstat</code> , and <code>tcpdump</code>	3250

28.4.7	A Useful Picture	3251
28.4.8	Tracing a Packet Through the Stack	3252
28.4.9	The Queue Disciplines Above You	3253
28.4.10	A Note on Packet-Tap Visibility	3253
28.4.11	Wrapping Up Section 1	3254
28.5	Section 2: Introducing <code>ifnet</code>	3254
28.5.1	Where <code>ifnet</code> Lives	3254
28.5.2	The Minimum Fields You Care About	3254
28.5.3	The <code>ifnet</code> Life Cycle	3255
28.5.4	Why <code>ifnet</code> Is Not a <code>cdevsw</code>	3256
28.5.5	Pseudo-Interfaces vs Real NIC Interfaces	3256
28.5.6	A Closer Look at Key <code>ifnet</code> Fields	3257
28.5.7	The Accessor API in More Detail	3258
28.5.8	A First Glimpse of the Code	3259
28.5.9	Wrapping Up Section 2	3260
28.5.10	A Guided Tour of <code>if_disc.c</code>	3260
28.6	Section 3: Creating and Registering a Network Interface	3262
28.6.1	Project Layout	3262
28.6.2	The Makefile	3262
28.6.3	Preliminary Includes and Module Glue	3263
28.6.4	The Softc	3264
28.6.5	The Skeleton of <code>mynet_create</code>	3265
28.6.6	The Skeleton of <code>mynet_destroy</code>	3267
28.6.7	Registering the Cloner	3268
28.6.8	The Module Handler	3269
28.6.9	A Closer Look at Interface Cloners	3269
28.6.10	What Happens Inside <code>ether_ifattach</code>	3270
28.6.11	Build, Load, and Verify	3271
28.6.12	What the Stack Now Knows About Us	3271
28.6.13	Common Mistakes	3272
28.6.14	Wrapping Up Section 3	3272
28.7	Section 4: Handling Packet Transmission	3272
28.7.1	How the Stack Reaches Us	3273
28.7.2	The Transmit Callback Signature	3273
28.7.3	The NULL Check	3274
28.7.4	<code>M_ASSERTPKTHDR</code>	3274
28.7.5	MTU Validation	3275
28.7.6	State Validation	3275
28.7.7	BPF Tap	3275
28.7.8	Counter Updates	3275
28.7.9	The Final Drop	3276
28.7.10	The Queue-Flush Callback	3276
28.7.11	The <code>mynet_init</code> Callback	3276
28.7.12	A Close Look at Ordering and Locking	3277
28.7.13	Packet Surgery with <code>m_pullup</code>	3278
28.7.14	A Deeper Look at mbuf Structure	3278
28.7.15	mbuf Chains and Scatter-Gather	3279
28.7.16	mbuf Types and What They Mean	3280
28.7.17	Packet Header Fields	3280
28.7.18	Reference-Counted External Buffers	3280
28.7.19	The Receive Allocation Pattern	3281

28.7.20 Common mbuf-Related Mistakes	3281
28.7.21 Multi-Queue Transmit in Real Drivers	3282
28.7.22 A Digression on the <code>drbr(9)</code> Helper	3282
28.7.23 Testing the Transmit Path	3283
28.7.24 Pitfalls	3283
28.7.25 Wrapping Up Section 4	3284
28.8 Section 5: Handling Packet Reception	3284
28.8.1 The Callback Direction	3284
28.8.2 The Simulated Arrival	3285
28.8.3 <code>MGETHDR</code> : Allocating the Head of the Chain	3287
28.8.4 Setting Packet Header Fields	3287
28.8.5 Writing the Ethernet Header	3287
28.8.6 Writing the ARP Body	3287
28.8.7 Handing It to BPF	3288
28.8.8 Incrementing Input Counters	3288
28.8.9 Calling <code>if_input</code>	3288
28.8.10 Verifying the Receive Path	3288
28.8.11 Ownership: A Diagram	3289
28.8.12 Keeping Receive Safe Under Contention	3289
28.8.13 Alternative: Using <code>if_epoch</code>	3290
28.8.14 Receive Paths in Real NIC Drivers	3290
28.8.15 Deferred Receive Processing with Taskqueues	3290
28.8.16 Understanding <code>net_epoch</code>	3291
28.8.17 Receive Backpressure	3292
28.8.18 Common Mistakes	3292
28.8.19 Wrapping Up Section 5	3293
28.9 Section 6: Media State, Flags, and Link Events	3293
28.9.1 Interface Flags: <code>IFF_</code> and <code>IFF_DRV_</code>	3293
28.9.2 The <code>SIOCSIFFLAGS</code> <code>Ioctl</code>	3294
28.9.3 Writing <code>mynet_stop</code>	3294
28.9.4 Link State: <code>if_link_state_change</code>	3295
28.9.5 Media Descriptors	3295
28.9.6 Handling MTU Changes	3297
28.9.7 Handling Multicast Group Changes	3297
28.9.8 Putting the <code>Ioctl</code> Handler Together	3298
28.9.9 Flag Coherence Rules	3299
28.9.10 Interface Capabilities in Depth	3299
28.9.11 The <code>ether_ioctl</code> Common Handler	3300
28.9.12 Hardware Multicast Filtering	3301
28.9.13 Wrapping Up Section 6	3302
28.10 Section 7: Testing the Driver With Standard Networking Tools	3302
28.10.1 Load, Create, Configure	3302
28.10.2 Inspecting With <code>netstat</code>	3303
28.10.3 Capturing With <code>tcpdump</code>	3303
28.10.4 Generating Traffic With <code>ping</code>	3303
28.10.5 <code>arp -an</code>	3304
28.10.6 <code>sysctl net.link</code> and <code>sysctl net.inet</code>	3304
28.10.7 Monitoring Link Events With <code>ifstatusd</code> or <code>devd</code>	3304
28.10.8 Testing MTU Changes	3304
28.10.9 Testing Media Commands	3305
28.10.10 Comparing Against <code>if_disc</code>	3305

28.10.1	Stress Test With <code>iperf3</code>	3305
28.10.1	Scripted Test Runs	3306
28.10.13	What to Watch For	3307
28.10.14	Deeper Observability with DTrace	3307
28.10.1	Kernel Debugger Tips for Drivers	3308
28.10.16	<code>systat -if</code> for Live Counter Views	3308
28.10.17	Wrapping Up Section 7	3309
28.11	Section 8: Cleanup, Detach, and Refactoring of the Network Driver	3309
28.11.1	The Full Teardown Sequence	3309
28.11.2	The Cloner Destroy Path	3310
28.11.3	Module Unload	3310
28.11.4	Memory Accounting	3311
28.11.5	Dealing With Stuck Callouts	3311
28.11.6	VNET Considerations	3312
28.11.7	Refactoring Advice	3312
28.11.8	Feature Flags and Capabilities	3313
28.11.9	Writing a Run Script	3313
28.11.10	Tidying Up the File	3313
28.11.11	Handling Partial Initialisation Failure	3313
28.11.12	Synchronising With In-Flight Callbacks	3314
28.11.13A	A Worked Error Scenario	3315
28.11.14	Wrapping Up Section 8	3315
28.12	Hands-On Labs	3315
28.12.1	Lab 1: Build and Load the Skeleton	3316
28.12.2	Lab 2: Exercise the Transmit Path	3316
28.12.3	Lab 3: Exercise the Receive Path	3317
28.12.4	Lab 4: Media and Link State	3318
28.12.5	Lab 5: <code>tcpdump</code> and BPF	3319
28.12.6	Lab 6: Clean Detach	3319
28.12.7	Lab 7: Reading the Real Tree	3320
28.13	Challenge Exercises	3321
28.13.1	Challenge 1: Shared Queue Between Paired Interfaces	3321
28.13.2	Challenge 2: Link Flap Simulation	3321
28.13.3	Challenge 3: Error Injection	3322
28.13.4	Challenge 4: <code>iperf3</code> Stress	3322
28.13.5	Challenge 5: Per-Interface <code>sysctl</code> Tree	3322
28.13.6	Challenge 6: Netgraph Node	3323
28.14	Troubleshooting and Common Mistakes	3323
28.14.1	Symptom: <code>ifconfig mynet create</code> Returns “Operation not supported”	3323
28.14.2	Symptom: <code>ifconfig mynet0 up</code> Hangs or Panics	3323
28.14.3	Symptom: <code>tcpdump -i mynet0</code> Sees No Packets	3323
28.14.4	Symptom: <code>ping</code> Shows 100% Loss (Expected) but <code>Opkts</code> Stays Zero	3324
28.14.5	Symptom: <code>kldunload</code> Panics with “destroying locked mutex”	3324
28.14.6	Symptom: <code>netstat -in -I mynet0</code> Shows <code>Opkts</code> Higher Than <code>Opkts</code> in <code>systat -if</code>	3324
28.14.7	Symptom: Module Loads But <code>ifconfig mynet create</code> Produces a Kernel Warning	3324
28.14.8	Symptom: <code>kldunload</code> Returns But <code>ifconfig -a</code> Still Shows <code>mynet0</code>	3324
28.14.9	Symptom: Intermittent Panics Under <code>iperf3</code> Load	3324
28.14.10	Symptom: <code>ifconfig mynet0 mtu 9000</code> Succeeds But Jumbo Frames Fail	3325
28.14.11	Symptom: <code>dmesg</code> Shows “acquiring a destroyed mutex”	3325
28.14.12	Symptom: <code>WITNESS</code> Reports a Lock-Order Reversal	3325
28.14.13	Symptom: Packet Loss Under Moderate Load	3325

28.14.1	Symptom: <code>ifconfig mynet0 inet6 2001:db8::1/64</code> Has No Effect	3325
28.14.1	Symptom: Module Loads But <code>ifconfig mynet create</code> Produces No Interface and No Error	3325
28.14.1	Symptom: <code>sysctl net.link.generic</code> Returns Unexpected Results	3326
28.15	Quick Reference Tables	3326
28.15.1	Lifecycle Functions	3326
28.15.2	Data-Path Functions	3327
28.15.3	Common <code>IFF_</code> Flags	3327
28.15.4	Common <code>IFCAP_</code> Capabilities	3328
28.15.5	Common <code>IFCOUNTER_</code> Counters	3329
28.15.6	Common Interface <code>Ioctls</code>	3329
28.16	Reading Real Network Drivers	3329
28.16.1	Reading <code>/usr/src/sys/net/if_tuntap.c</code>	3330
28.16.2	Reading <code>/usr/src/sys/net/if_bridge.c</code>	3330
28.16.3	Reading <code>/usr/src/sys/dev/e1000/if_em.c</code>	3330
28.16.4	Reading <code>/usr/src/sys/dev/virtio/network/if_vtnet.c</code>	3331
28.16.5	Reading <code>/usr/src/sys/net/if_lagg.c</code>	3331
28.16.6	Suggested Reading Order	3331
28.16.7	Reading as a Habit	3332
28.17	Production Considerations	3332
28.17.1	Performance	3332
28.17.2	Reliability	3332
28.17.3	Observability	3333
28.17.4	Backward Compatibility	3333
28.17.5	Licensing and Upstreaming	3333
28.18	Wrapping Up	3333
28.18.1	What This Chapter Did Not Cover	3334
28.18.2	Final Reflection	3335
28.18.3	Further Reading	3335
28.19	Field Guide to Related <code>ifnet</code> Subsystems	3336
28.19.1	<code>arp(4)</code> and Neighbour Discovery	3336
28.19.2	<code>bpf(4)</code>	3336
28.19.3	<code>route(4)</code>	3336
28.19.4	<code>if_clone(4)</code>	3336
28.19.5	<code>pf(4)</code>	3337
28.19.6	<code>netmap(4)</code>	3337
28.19.7	<code>netgraph(4)</code>	3337
28.19.8	<code>iflib(9)</code>	3337
28.19.9	Summary of the Landscape	3337
28.20	Debugging Scenarios: A Worked Example	3338
28.20.1	The Problem	3338
28.20.2	First Hypothesis: the <code>transmit</code> function is not being called.	3338
28.20.3	Investigating Why	3338
28.20.4	Second Problem: counter disagreement	3338
28.20.5	Third Problem: the receive path seems intermittent	3339
28.20.6	What This Scenario Teaches	3339
28.21	A Note on Testing Discipline	3339
28.21.1	Unit-Level Thinking	3340
28.21.2	Lifecycle Tests	3340
28.21.3	Error-Path Tests	3340
28.21.4	Regression Tests	3340

28.21.5 Watching for Latent Issues	3340
28.22A Complete Walkthrough of mynet .c	3341
28.22.1 File-Level Preamble	3341
28.22.2 Forward Declarations	3341
28.22.3 Cloner Dispatch	3342
28.22.4 Per-Unit Creation	3342
28.22.5 Destruction	3344
28.22.6 Init and Stop	3344
28.22.7 The Data Path	3345
28.22.8 Ioctl and Media Callbacks	3347
28.22.9 Module Glue and Cloner Registration	3349
28.22.10 Line Count and Density	3350
28.23A Complete Lifecycle Trace	3350
28.23.1 Trace 1: From kldload to ifconfig up	3350
28.23.2 Trace 2: From ifconfig down to kldunload	3352
28.24 Common Misconceptions About Network Drivers	3352
28.25 How Packets Actually Reach and Leave Your Driver	3353
28.25.1 The outbound path	3354
28.25.2 The inbound path	3354
28.25.3 Why this geography matters for driver authors	3355
28.26 Things This Chapter Did Not Cover	3355
28.27 Historical Context: Why ifnet Looks the Way It Does	3356
28.28 Self-Assessment: Do You Really Know This Material?	3357
28.29 Further Reading and Source Code Study	3358
28.30 Frequently Asked Questions	3359
28.31 A Short Closing Note on Craft	3361
28.32 Mini-Glossary of Network Driver Terms	3362
28.33 Part 6 Checkpoint	3363
28.34 Looking Ahead: Bridge to Chapter 29	3364
29 Portability and Driver Abstraction	3366
29.1 Introduction	3366
29.2 Reader Guidance: How to Use This Chapter	3367
29.2.1 Work Section by Section	3368
29.2.2 A Gentle Warning About Over-Engineering	3368
29.2.3 Keep the Reference Driver Close	3368
29.2.4 Open the FreeBSD Source Tree	3369
29.2.5 Keep the Lab Logbook	3369
29.2.6 Pace Yourself	3369
29.3 How to Get the Most Out of This Chapter	3369
29.3.1 Refactor in Small Steps	3369
29.3.2 Read the Compiler Output	3370
29.3.3 Use Version Control as a Safety Net	3370
29.3.4 Compare Against Real FreeBSD Drivers	3370
29.3.5 Take Notes on Patterns, Not Syntax	3370
29.3.6 Trust the Build System	3370
29.3.7 Take Breaks	3370
29.4 Section 1: What Does “Portable” Mean for a Device Driver?	3371
29.4.1 A Working Definition	3371
29.4.2 Portability Across Architectures	3371
29.4.3 Portability Across Buses	3372
29.4.4 Portability Across Systems	3372

29.4.5	Portability Across Time	3372
29.4.6	Portability Across Deployments	3373
29.4.7	Why Portability Matters	3373
29.4.8	Code Duplication vs Modular Design	3373
29.4.9	A Tale of Two Drivers	3374
29.4.10	The Cost of Deferring Portability	3375
29.4.11	Why FreeBSD Makes This Easier Than Most	3375
29.4.12	Wrapping Up Section 1	3376
29.5	Section 2: Isolating Hardware-Dependent Code	3376
29.5.1	What Counts as Hardware-Dependent?	3376
29.5.2	A Small Thought Experiment	3377
29.5.3	A Closer Look at the uart Driver	3377
29.5.4	The bus_space(9) Abstraction	3378
29.5.5	Wrapping bus_space for Clarity	3379
29.5.6	Conditional Register Access by Bus	3379
29.5.7	The bus_dma(9) Abstraction	3380
29.5.8	A Practical Header-Per-Backend Pattern	3381
29.5.9	Common Mistakes When Isolating Hardware Code	3381
29.5.10	Subregions, Barriers, and Burst Operations	3382
29.5.11	Layered Wrapping: From Primitive to Domain	3382
29.5.12	Accessor Layering Inside the FreeBSD Tree	3383
29.5.13	When the Line Between HW-Dependent and HW-Independent Blurs	3384
29.5.14	Wrapping Up Section 2	3385
29.6	Section 3: Abstracting Device Behavior	3385
29.6.1	Why a Formal Interface Beats If-Else Chains	3385
29.6.2	Setting Up the Backend at Attach Time	3387
29.6.3	How the Real UART Driver Does This	3388
29.6.4	Choosing the Set of Operations	3388
29.6.5	Avoiding If-Else Creep	3389
29.6.6	Backend Discovery and Registration	3389
29.6.7	Common Mistakes When Designing an Interface	3389
29.6.8	A Closer Look at kobj(9)	3390
29.6.9	Stacked Backends and Delegation	3390
29.6.10	Making the Interface Contract Explicit	3391
29.6.11	Minimal vs. Rich Interfaces	3392
29.6.12	The “Soft” Contract of a Backend	3393
29.6.13	Wrapping Up Section 3	3393
29.7	Section 4: Splitting Code Into Logical Modules	3393
29.7.1	The Canonical Layout	3394
29.7.2	Responsibilities Per File	3394
29.7.3	A Minimal Core File	3395
29.7.4	A Minimal Backend File	3397
29.7.5	Header Organisation	3400
29.7.6	The Makefile	3402
29.7.7	Compare with /usr/src/sys/modules/uart/Makefile	3403
29.7.8	Makefile Idioms for Portable Drivers	3404
29.7.9	When Multiple Modules Share Code	3405
29.7.10	Benefits of the Split	3405
29.7.11	Common Mistakes When Splitting a Driver	3406
29.7.12	Header Hygiene and Include Graphs	3406
29.7.13	Managing CFLAGS and Symbol Visibility	3407

29.7.14	A Dependency Graph on Paper	3407
29.7.15	Static Analysis and Continuous Reviews	3408
29.7.16	Wrapping Up Section 4	3409
29.8	Section 5: Supporting Multiple Architectures	3409
29.8.1	How <code>bus_space</code> Varies Across Platforms	3409
29.8.2	Endianness in Three Questions	3410
29.8.3	The Endian Helpers in Detail	3410
29.8.4	A Simple Mental Rule	3411
29.8.5	Alignment and the <code>NO_STRICT_ALIGNMENT</code> Macro	3411
29.8.6	Word Size: 32 vs 64 Bits	3412
29.8.7	Bus Address Types	3413
29.8.8	Testing on Emulated Architectures	3413
29.8.9	Case Study: A DMA Descriptor Ring	3413
29.8.10	A Register Layout Worth Doing Carefully	3415
29.8.11	Using the Newer <code>bus_read_*/bus_write_*</code> Helpers	3416
29.8.12	Common Mistakes When Supporting Multiple Architectures	3416
29.8.13	Memory Barriers and Ordering	3417
29.8.14	Cache Coherency and <code>bus_dmamap_sync</code>	3417
29.8.15	Bounce Buffers and Address Limits	3418
29.8.16	Architectural Smoke Tests	3418
29.8.17	The Realities of Non-x86 FreeBSD	3419
29.8.18	Aside: CHERI-BSD and Morello	3419
29.8.19	Wrapping Up Section 5	3421
29.9	Section 6: Conditional Compilation and Build-time Flexibility	3421
29.9.1	The Three Levels of Conditional Compilation	3422
29.9.2	The <code>__FreeBSD_version</code> Macro	3422
29.9.3	The Kernel Options Mechanism	3423
29.9.4	Simulated Mode as a Build Option	3424
29.9.5	Debug Builds	3425
29.9.6	Feature Flags Through <code>sysctl</code> and <code>dmesg</code>	3425
29.9.7	Avoiding <code>#ifdef</code> Sprawl	3426
29.9.8	Architecture-Specific Conditionals	3426
29.9.9	Optional Subsystem Support	3427
29.9.10	Avoiding Runtime Feature Checks That Should Be Compile-Time	3427
29.9.11	Options Files and Kernel Configuration	3427
29.9.12	Module Load Hints and <code>device.hints</code>	3428
29.9.13	Feature Flags Through <code>MODULE_PNP_INFO</code>	3429
29.9.14	The Limits of <code>__FreeBSD_version</code>	3429
29.9.15	Organising Compatibility Shims	3430
29.9.16	Build-Time Telemetry	3430
29.9.17	Wrapping Up Section 6	3431
29.10	Section 7: Adapting for Cross-BSD Compatibility	3431
29.10.1	Where the BSDs Agree	3431
29.10.2	Where the BSDs Differ	3432
29.10.3	The Cross-BSD-Friendly Style	3432
29.10.4	Compatibility Wrappers	3433
29.10.5	Listing the FreeBSD-Specific APIs You Use	3434
29.10.6	When to Commit to Cross-BSD Support	3434
29.10.7	A Concrete Wrapping Example	3435
29.10.8	What “Porting” Actually Involves	3436
29.10.9	When Cross-BSD Means Cross-Platform Generally	3436

29.10.10	NetBSD and OpenBSD Pointers for the Curious	3436
29.10.11	Wrapping Up Section 7	3437
29.11	Section 8: Refactoring and Versioning for Portability	3437
29.11.1	A Short README.portability.md	3437
29.11.2	Versioning the Driver	3439
29.11.3	Document Supported Build Configurations	3439
29.11.4	Validating the Build Before a Release	3440
29.11.5	When to Version Bump	3441
29.11.6	Refactoring for Portability as an Ongoing Discipline	3441
29.11.7	The Final Shape of a Portable Driver	3442
29.11.8	ABI vs. API Stability	3442
29.11.9	Upgrade and Downgrade Paths	3443
29.11.10	Telemetry and Observability	3443
29.11.11	A Post-Refactor Checklist	3443
29.11.12	Rollouts and Rollbacks	3444
29.11.13	Wrapping Up Section 8	3444
29.12	Hands-On Labs	3445
29.12.1	Setting Up the Lab Environment	3445
29.12.2	Lab 1: Audit the Monolithic Driver	3446
29.12.3	Lab 2: Introduce Register Accessors	3447
29.12.4	Lab 3: Extract the Backend Interface	3448
29.12.5	Lab 4: Split the Core and the Backend into Separate Files	3449
29.12.6	Lab 5: Add a Simulation Backend	3450
29.12.7	Lab 6: Conditional Compilation of Backends	3452
29.12.8	Lab 7: Endian-Safe Register Access	3454
29.12.9	Lab 8: Build Matrix Script	3454
29.12.10	Lab 9: Exercising the Simulation Backend from Userland	3455
29.12.11	Lab 10: Add a Basic sysctl Tree	3457
29.12.12	Lab 11: Runtime Backend Selection	3458
29.12.13	Lab 12: Write a Minimal CI Harness	3459
29.13	Challenge Exercises	3460
29.13.1	Challenge 1: Add a Third Backend	3460
29.13.2	Challenge 2: Expose Version Metadata via sysctl	3460
29.13.3	Challenge 3: Simulate a Big-Endian Host	3460
29.13.4	Challenge 4: Cross-Build for arm64	3461
29.13.5	Challenge 5: Write a README.portability.md	3461
29.13.6	Challenge 6: Create a Versioned Backend Interface	3461
29.13.7	Challenge 7: Audit a Real FreeBSD Driver	3461
29.13.8	Challenge 8: Measure the Cost of Backend Dispatch	3462
29.13.9	Challenge 9: Add a Second FreeBSD Version Target	3462
29.13.10	Challenge 10: Design a New Driver from Scratch	3462
29.14	Troubleshooting and Common Mistakes	3463
29.14.1	Symptom: Build Fails with “Undefined Reference” After Splitting a File	3463
29.14.2	Symptom: Build Fails with “Implicit Function Declaration” After Splitting	3463
29.14.3	Symptom: Build Succeeds, but Load Fails with a Dependency Error	3463
29.14.4	Symptom: The Driver Loads, but dmesg Shows Only One Backend	3463
29.14.5	Symptom: Register Reads Return Zero on ARM but Work on x86	3464
29.14.6	Symptom: The Simulation Backend Works but the PCI Backend Hangs at Attach	3464
29.14.7	Symptom: Build Fails on a Different FreeBSD Release	3464
29.14.8	Symptom: The Driver Panics with “Locking Assertion Failed” After a Backend Split	3464
29.14.9	Symptom: The Driver Builds but make clean Leaves Stale Object Files	3464

29.14.1	Symptom: Cross-Architecture Build Fails with “Cannot Find <machine/bus.h>” . . .	3465
29.14.1	Symptom: A <code>sysctl</code> You Just Added Does Not Appear	3465
29.14.1	Symptom: The Driver Works on Real Hardware but Not in <code>bhyve</code>	3465
29.14.1	Symptom: Performance Varies Wildly Between Architectures	3465
29.14.1	Symptom: <code>kldunload</code> Hangs	3465
29.14.1	Symptom: Different Behaviour Between <code>make PORTDRV_WITH_SIM=yes</code> and <code>make PORTDRV_WITH_SIM=yes PORTDRV_DEBUG=yes</code>	3465
29.14.1	Symptom: Warning about “Incompatible Pointer Types” After Moving a Function . . .	3466
29.14.1	Symptom: <code>bus_dma</code> Fails with <code>ENOMEM</code> on <code>amd64</code> but Works on <code>arm64</code>	3466
29.14.1	Symptom: Userland Program Sees Different Data Than the Kernel Log Shows	3466
29.14.1	Symptom: The Module Loads but <code>kldstat</code> Shows It in a Different Position Than Expected	3466
29.14.2	Symptom: A Debug Build Consumes Much More Memory Than a Release Build . . .	3466
29.14.2	Symptom: Tests Pass on the Development Machine but Fail in CI	3467
29.14.2	Symptom: A <code>Sysctl</code> Can Be Read but Not Written, Even Though <code>CTLFLAG_RW</code> Was Set	3467
29.14.2	Symptom: The Driver Panics on <code>kldunload</code> After Running for a Long Time	3467
29.14.2	Symptom: Cross-Compiled Module Refuses to Load on the Target	3467
29.14.2	Symptom: A Backend’s <code>attach</code> Method Is Called Twice on the Same Softc	3467
29.14.2	Symptom: Compiler Warns about an Unused Variable in a Conditional Block	3467
29.14.2	Symptom: The Build Succeeds but <code>kldxref</code> Complains About Missing Metadata . . .	3468
29.14.2	Symptom: A Backend That Worked Yesterday Fails Today After a Kernel Update . . .	3468
29.14.2	Symptom: The Driver Rejects a Device That Should Work	3468
29.14.3	Symptom: A USB Backend Works with One Device but Not Another of the Same Type	3468
29.14.3	Symptom: After a File Split, Some <code>static</code> Functions Can No Longer Be Inlined . . .	3468
29.14.3	Symptom: <code>make</code> Recomiles Everything Even When Only One File Changed	3469
29.14.3	Generic Guidance	3469
29.15	A Retrospective: The Patterns at a Glance	3469
29.16	Portability Antipatterns: What to Unlearn	3470
29.16.1	The “It Works on My Machine” Driver	3471
29.16.2	The Silent <code>#ifdef</code>	3471
29.16.3	The Casual Cast	3471
29.16.4	The Optimistic Register	3471
29.16.5	The Ad-Hoc Version Check	3472
29.16.6	The “Temporary” Hack	3472
29.16.7	The Duplicated Constant	3472
29.16.8	The Growing <code>struct softc</code>	3472
29.16.9	The Uneven Test Coverage	3472
29.16.10	The Gratuitous Abstraction	3472
29.16.11	The Monolithic Commit	3473
29.17	Portability in the Real World: Three Short Stories	3473
29.17.1	Story 1: The Endian Bug That Slept for Five Years	3473
29.17.2	Story 2: The Conditional Block Nobody Remembered	3474
29.17.3	Story 3: The Simulation Backend That Saved a Release	3474
29.18	A Short Closing Note on Discipline	3474
29.18.1	What Success Looks Like	3475
29.18.2	A Note on the Path Ahead	3475
29.19	Mini-Glossary of Portability Terms	3476
29.20	Frequently Asked Questions	3478
29.21	Where the Chapter’s Patterns Appear in the Real Tree	3484
29.22	A Self-Study Schedule for the Patterns	3485
29.23	A Note on Collaboration and Code Review	3486
29.23.1	Code Review Checklists	3486

29.24A Short Section on Pragmatism	3487
29.25 Questions to Ask Yourself Before Calling a Driver Done	3487
29.26 Resources for Further Study	3488
29.27 Ten Principles to Carry Forward	3489
29.28 Wrapping Up	3490
29.29 Looking Ahead: Bridge to Chapter 30	3491
29.30A Final Word: The Long Arc	3492
30 Virtualisation and Containerization	3493
30.1 Introduction	3493
30.2 Reader Guidance: How to Use This Chapter	3494
30.2.1 Prerequisites	3495
30.2.2 What This Chapter Does Not Cover	3495
30.2.3 Structure and Pacing	3496
30.2.4 Work Section by Section	3496
30.2.5 Keep the Reference Driver Close	3496
30.2.6 Open the FreeBSD Source Tree	3497
30.2.7 Keep a Lab Logbook	3497
30.2.8 Pace Yourself	3497
30.3 How to Get the Most Out of This Chapter	3497
30.3.1 Read With the Source Tree Open	3497
30.3.2 Type the Labs	3498
30.3.3 Run What You Read	3498
30.3.4 Treat dmesg as Part of the Manuscript	3498
30.3.5 Break Things Deliberately	3498
30.3.6 Work in Pairs When You Can	3498
30.3.7 Trust the Iteration, Not the Memorisation	3498
30.3.8 Take Breaks	3499
30.4 Section 1: What Virtualisation and Containerization Mean for Driver Authors	3499
30.4.1 Two Families, Not One	3499
30.4.2 Virtual Machines as an Environment for Drivers	3500
30.4.3 Containers as an Environment for Drivers	3501
30.4.4 Why This Matters for Driver Authors	3501
30.4.5 Virtualisation Is Not Just Hardware But Slower	3502
30.4.6 A Quick Taxonomy	3502
30.4.7 Wrapping Up	3503
30.5 Section 2: Guest Drivers, Emulated Devices, and Paravirtualised Devices	3503
30.5.1 Emulated Devices in Detail	3503
30.5.2 What Emulated Devices Imply for Driver Code	3503
30.5.3 Why Emulation Exists at All	3504
30.5.4 Passthrough as Seen by the Guest	3504
30.5.5 Firmware and Platform Dependencies Under Passthrough	3505
30.5.6 When to Prefer Which Device Style	3505
30.5.7 Paravirtual Devices in Detail	3506
30.5.8 Paravirtual Devices and the Driver Author	3506
30.5.9 Passthrough Devices in Detail	3507
30.5.10 Passthrough and the IOMMU	3507
30.5.11 The Driver Author's Mental Model	3507
30.5.12 A Note on Hybrid Models	3508
30.5.13 Wrapping Up	3508
30.6 Section 3: VirtIO Fundamentals and virtqueue(9)	3508
30.6.1 The Ingredients of a VirtIO Device	3509

30.6.2	Virtqueues and the Shared-Ring Model	3509
30.6.3	Feature Negotiation in Practice	3510
30.6.4	A Walkthrough of <code>virtio_random</code>	3511
30.6.5	What the <code>virtqueue(9)</code> API Buys You	3514
30.6.6	Indirect Descriptors	3515
30.6.7	A Deeper Look at the Ring Layout	3515
30.6.8	Event Indexes: A Performance Detail	3516
30.6.9	Legacy Versus Modern VirtIO	3517
30.6.10	Feature Bits Worth Knowing	3517
30.6.11	A Second Example: Structure of <code>virtio_net</code>	3518
30.6.12	A Third Example for Intuition: <code>virtio_blk</code>	3519
30.6.13	Common Mistakes When Reading VirtIO Code	3519
30.6.14	Wrapping Up	3520
30.7	Section 4: Hypervisor Detection and Environment-Aware Driver Behaviour	3520
30.7.1	The <code>vm_guest</code> Global	3520
30.7.2	How the Detection Works	3521
30.7.3	When to Consult <code>vm_guest</code>	3521
30.7.4	A Usage Example	3522
30.7.5	The <code>Sysctl</code> Path	3522
30.7.6	Interaction with Subsystems	3523
30.7.7	Detecting Within a VirtIO Driver	3523
30.7.8	Detecting From the Host Side	3523
30.7.9	Wrapping Up	3524
30.8	Section 5: <code>bhyve</code> , PCI Passthrough, and Host-Side Considerations	3524
30.8.1	<code>bhyve</code> From the Driver Author's Perspective	3524
30.8.2	<code>vmm(4)</code> : The Kernel Side of <code>bhyve</code>	3525
30.8.3	PCI Passthrough: Giving a Guest a Real Device	3525
30.8.4	How Passthrough Looks to the Guest Driver	3526
30.8.5	Host-Side Driver Attach Under <code>bhyve</code>	3526
30.8.6	IOMMU Groups and the Reality of Partial Isolation	3527
30.8.7	Host-Only Considerations: Memory, CPU, and the Hypervisor	3527
30.8.8	Cross-Platform Notes: <code>bhyve</code> on <code>arm64</code>	3528
30.8.9	Inspecting <code>bhyve</code> From the Host	3528
30.8.10	A Typical <code>bhyve</code> Command Line	3528
30.8.11	Inside the <code>vmm(4)</code> Module	3529
30.8.12	Nested Virtualisation	3529
30.8.13	An Example: Debugging a PCI Passthrough Driver	3530
30.8.14	Wrapping Up	3530
30.9	Section 6: Jails, <code>devfs</code> , VNET, and Device Visibility	3531
30.9.1	What a Jail Is, and What It Is Not	3531
30.9.2	The Kernel's View of Jails	3531
30.9.3	<code>devfs</code> : The Filesystem Through Which Devices Are Exposed	3532
30.9.4	<code>devfs</code> Rulesets: Declaring What a Jail Can See	3532
30.9.5	Creating a Custom <code>devfs</code> Ruleset	3533
30.9.6	What <code>devfs</code> Rulesets Do Not Do	3534
30.9.7	A Driver Author's View of <code>devfs</code> Rulesets	3534
30.9.8	Wrapping Up the Jail and <code>devfs</code> Side	3534
30.9.9	The VNET Framework: One Kernel, Many Network Stacks	3535
30.9.10	Declaring VNET State: <code>VNET_DEFINE</code> and <code>CURVNET_SET</code>	3535
30.9.11	<code>if_vmove</code> : Moving an Interface Between VNETs	3536
30.9.12	VNET Lifecycle Hooks	3537

30.9.13 A Concrete VNET Pattern	3537
30.9.14 Reading Real VNET Code	3538
30.9.15 Hierarchical Jails	3539
30.9.16 Putting It All Together	3539
30.9.17 Single-Stack Jails and the In-Between	3539
30.9.18 Wrapping Up	3539
30.10 Section 7: Resource Limits, Security Boundaries, and Host-Versus-Jail Access	3540
30.10.1 The Privilege Framework and <code>prison_priv_check</code>	3540
30.10.2 Named Privileges Most Relevant to Drivers	3541
30.10.3 <code>rctl(8)</code> : Per-Jail and Per-Process Resource Limits	3542
30.10.4 Enforcement Actions and What They Mean for Drivers	3542
30.10.5 Host-Side Drivers versus Jail-Side Processes	3543
30.10.6 How Drivers Cooperate with Jails in Practice	3543
30.10.7 The Container Frameworks: <code>ocijail</code> and <code>pot</code>	3544
30.10.8 Wrapping Up	3544
30.11 Section 8: Testing and Refactoring Drivers for Virtualised and Containerised Environments	3545
30.11.1 Using a VM as Your Development Host	3545
30.11.2 Using <code>VirtIO</code> as a Test Substrate	3545
30.11.3 Using Jails for Integration Testing	3546
30.11.4 Using VNET Jails for Network Driver Testing	3546
30.11.5 Passthrough Testing	3547
30.11.6 Hypervisor Detection in Tests	3547
30.11.7 Automating the Test Suite	3547
30.11.8 Refactoring Tips Revisited	3548
30.11.9 A Development Order That Works	3548
30.11.10 An End-to-End Example: From Bare Metal to Passthrough	3549
30.11.11 Measuring Virtualisation Overhead	3550
30.11.12 Wrapping Up	3550
30.12 Section 9: Timekeeping, Memory, and Interrupt Handling Under Virtualisation	3551
30.12.1 Why Time Is Different Inside a VM	3551
30.12.2 What This Means for Drivers	3552
30.12.3 Callouts and Timers Across Migration	3553
30.12.4 Memory Pressure, Ballooning, and Pinned Buffers	3554
30.12.5 Memory Hotplug	3554
30.12.6 DMA and the IOMMU Path	3554
30.12.7 Interrupt Delivery in Virtualised Environments	3555
30.12.8 Interrupt Coalescing and Notification Suppression	3556
30.12.9 Putting the Pieces Together	3557
30.13 Section 10: Virtualisation on Other Architectures	3557
30.13.1 <code>arm64</code> Virtualisation	3558
30.13.2 <code>riscv64</code> Virtualisation	3558
30.13.3 Cross-Architecture Considerations for Driver Authors	3558
30.13.4 When Architecture Matters for Real	3559
30.14 Hands-on Labs	3559
30.14.1 Lab 1: Exploring a <code>VirtIO</code> Guest	3559
30.14.2 Lab 2: Using Hypervisor Detection in a Kernel Module	3560
30.14.3 Lab 3: A Minimal Character Device Driver Inside a Jail	3561
30.14.4 Lab 4: A Network Driver Inside a VNET Jail	3562
30.14.5 Lab 5: PCI Passthrough Simulation (Optional)	3563
30.14.6 Lab 6: Building and Loading the <code>vtedu</code> Driver	3563
30.14.7 Lab 7: Measuring <code>VirtIO</code> Overhead	3564

30.14.8 A Note on Labs You Cannot Complete Today	3565
30.15 Challenge Exercises	3565
30.15.1 Challenge 1: Extend the Detect Module	3566
30.15.2 Challenge 2: A Simulation Backend for a Real VirtIO Driver	3566
30.15.3 Challenge 3: A VNET-Aware Driver Skeleton	3566
30.15.4 Challenge 4: A Jail-Visible Status Device	3566
30.15.5 Challenge 5: A bhyve Emulated Device in User Space	3567
30.15.6 Challenge 6: Container Orchestration for Driver Testing	3567
30.15.7 How to Approach These Challenges	3567
30.16 Troubleshooting and Common Mistakes	3568
30.16.1 VirtIO Device Not Detected in the Guest	3568
30.16.2 virtqueue_enqueue Fails With ENOBUFS	3568
30.16.3 Passthrough Device Not Available in the Guest	3568
30.16.4 Kernel Panic at Module Load Inside a Guest	3569
30.16.5 Jail Cannot See a Device Even After Unhiding	3569
30.16.6 Privilege Denied Inside a Jail That Should Work	3569
30.16.7 VNET Jail Cannot Send or Receive Packets	3570
30.16.8 PCI Passthrough Driver Fails to Attach in the Guest	3570
30.16.9 Host Becomes Unresponsive When a Guest Starts	3570
30.16.10 kldunload Hangs	3571
30.16.11 Timing-Related Bugs Only in VMs	3571
30.16.12 VirtIO Negotiation Fails or Returns Unexpected Features	3571
30.16.13 bus_alloc_resource Fails Inside a Guest	3572
30.16.14 kldload Succeeds but No Device Attaches	3572
30.16.15 Module Is Loaded but /dev Node Does Not Appear	3572
30.16.16 Virtqueue Interrupts Never Fire	3573
30.16.17 Interrupt Storm Under Virtualisation	3573
30.16.18 DMA Failures Only Under Passthrough	3573
30.16.19 A VirtIO Device Appears but Has the Wrong Type	3574
30.16.20 Jail-Specific ioctl Fails With ENOTTY	3574
30.16.21 VNET Move Leaks Per-VNET State	3574
30.16.22 Guest Kernel Panics with “Fatal Trap 12” on First I/O	3575
30.16.23 Live Migration Fails or Causes Guest Hang	3575
30.16.24 Unexpected devfs Entries Appear Inside a Jail	3575
30.16.25 Two Drivers Fight Over the Same Device	3576
30.16.26 m_guest Shows “no” Inside an Obvious VM	3576
30.16.27 General Diagnostic Approach	3576
30.17 Wrapping Up	3577
30.18 Looking Ahead: Security and Privilege	3578
30.19 Case Study: Designing a Pedagogical VirtIO Driver	3579
30.19.1 What vtedu Is For	3579
30.19.2 Choosing the Device Identifier	3579
30.19.3 Defining the Features	3579
30.19.4 Softc Layout	3579
30.19.5 Transport Registration	3580
30.19.6 Probe and Attach	3580
30.19.7 Feature Negotiation	3582
30.19.8 Queue Allocation	3582
30.19.9 The Character Device Interface	3582
30.19.10 The Interrupt Handler	3584
30.19.11 The Read Path	3584

30.19.1	Detach	3585
30.19.1	Putting vtedu Together	3585
30.19.14	Using vtedu for Teaching	3586
30.19.15	What vtedu Does Not Do	3586
30.19.16A	Teaching Driver Versus a Real Driver	3586
30.20	Appendix: Quick Reference	3586
30.20.1	VirtIO Core API Functions	3587
30.20.2	virtqueue(9) Functions	3587
30.20.3	vm_guest Values	3587
30.20.4	Default devfs Rulesets	3588
30.20.5	Common Privilege Constants	3588
30.20.6	VNET Macros	3588
30.20.7	bhyve and Passthrough Tools	3589
30.20.8	Manual Pages Worth Bookmarking	3589
30.20.9	Top Five Things a Driver Author Should Do	3589
30.21	Appendix: Common Code Patterns	3590
30.21.1	Pattern: Environment-aware default	3590
30.21.2	Pattern: Privilege-gated ioctl	3590
30.21.3	Pattern: VNET-aware callout	3591
30.21.4	Pattern: VirtIO attach skeleton	3591
30.21.5	Pattern: Clean detach	3592
30.21.6	Pattern: make_dev with sensible defaults	3593
30.22	Appendix: Glossary	3593
30.23	Appendix: Observing VirtIO with DTrace	3594
30.23.1	A First Probe: Counting virtqueue Enqueues	3594
30.23.2	Separating Queues by Virtqueue	3594
30.23.3	Measuring Time Spent in Enqueue	3595
30.23.4	Watching for VM Exits	3595
30.23.5	Tracing a Specific Driver	3596
30.23.6	Tracing a Custom Driver	3596
30.23.7	Observing the Guest-Host Boundary	3596
30.23.8	Saving and Reusing Probes	3596
30.23.9	When DTrace Cannot Help	3597
30.23.10A	Closing Word	3597
30.24	Appendix: A Complete bhyve Configuration Walkthrough	3597
30.24.1	The Host Side	3597
30.24.2	Install vm-bhyve	3598
30.24.3	Create and Install a Guest	3599
30.24.4	Configure Networking	3599
30.24.5	Start the Guest	3599
30.24.6	Troubleshooting the Host Side	3600
30.24.7	Using This Setup for the Labs	3600
30.25	Appendix: VirtIO Feature Bits Reference	3600
30.25.1	Device-Independent Features	3600
30.25.2	Block Device Features (virtio_blk)	3601
30.25.3	Network Device Features (virtio_net)	3601
30.25.4	How to Read a Feature Word	3602
30.25.5	Common Pitfalls	3602
30.25.6	A Useful Habit	3602
30.26	Appendix: Further Reading	3603
30.26.1	FreeBSD source tree	3603

30.26.2 Manual pages	3603
30.26.3 External standards	3603
30.26.4 FreeBSD Handbook	3603
30.27 Appendix: Anti-Patterns in Virtualised Drivers	3604
30.27.1 Busy-Waiting on a Status Register	3604
30.27.2 Caching Physical Addresses	3604
30.27.3 Ignoring <code>si_drv1</code>	3604
30.27.4 Assuming <code>INTx</code>	3605
30.27.5 Hard-Coding Feature Bits	3605
30.27.6 Sleep-With-Spin-Lock Held	3605
30.27.7 Returning <code>ENOTTY</code> for Unsupported Operations	3605
30.27.8 Forgetting the Detach Path	3606
30.27.9 Assuming <code>device_printf</code> Works Without a Device	3606
30.27.10 Not Cleaning Up VNET State	3606
30.27.11 Using <code>getnanouptime</code> for Timing	3607
30.27.12 Probing Hardware in the Probe Method	3607
30.27.13 Storing Kernel Pointers Through <code>ioctl</code>	3607
30.27.14 Summary	3608
30.28 Appendix: A Checklist for a Virtualisation-Ready Driver	3608
30.28.1 Device Binding and Probe	3608
30.28.2 Resources	3608
30.28.3 DMA	3608
30.28.4 Interrupts	3609
30.28.5 Character Device Interface	3609
30.28.6 Locking	3609
30.28.7 Timing	3609
30.28.8 Privilege	3609
30.28.9 Detach and Unload	3609
30.28.10 VNET (if applicable)	3610
30.28.11 Testing	3610
30.28.12 Documentation	3610
30.29 Appendix: Sketching a bhyve Backend for <code>vtedu</code>	3610
30.29.1 Where the Code Lives	3610
30.29.2 The Framework Handles the Protocol	3611
30.29.3 Sketching the <code>vtedu</code> Backend	3611
30.29.4 Connecting to the Device Table	3613
30.29.5 What the Backend Must Verify	3613
30.29.6 Testing the End-to-End Loop	3613
30.29.7 Why This Exercise Is Worth the Effort	3614
30.30 Appendix: Running the Chapter's Techniques in CI	3614
30.30.1 Why CI Benefits from Virtualisation	3614
30.30.2 A Minimal CI Flow for a VirtIO Driver	3615
30.30.3 Practical Tools	3615
30.30.4 A Minimal Test Harness	3615
30.30.5 Scaling to Multiple Configurations	3616
30.30.6 A Note on Hardware-in-the-Loop CI	3617
30.30.7 The Payoff	3617
30.31 Appendix: Commands Cheat Sheet	3617
30.31.1 Host-Side Virtualisation	3617
30.31.2 Guest-Side Inspection	3618
30.31.3 PCI Passthrough	3618

30.31.4 Jails	3618
30.31.5 VNET	3619
30.31.6 Resource Limits	3619
30.31.7 Observability	3619
30.31.8 Module Lifecycle	3620
31 Security Best Practices	3621
31.1 Introduction	3621
31.2 Reader Guidance: How to Use This Chapter	3622
31.2.1 Prerequisites	3623
31.2.2 What This Chapter Does Not Cover	3624
31.2.3 Structure and Pacing	3624
31.2.4 Work Section by Section	3624
31.2.5 Keep the Reference Driver Close	3625
31.2.6 Open the FreeBSD Source Tree	3625
31.2.7 Keep a Lab Logbook	3625
31.2.8 Pace Yourself	3625
31.3 How to Get the Most Out of This Chapter	3625
31.3.1 Read With a Hostile Mind	3626
31.3.2 Run What You Read	3626
31.3.3 Type the Labs	3626
31.3.4 Treat dmesg as Part of the Manuscript	3626
31.3.5 Break Things Deliberately	3626
31.3.6 Work in Pairs When You Can	3626
31.3.7 Trust the Iteration	3627
31.3.8 Take Breaks	3627
31.4 Section 1: Why Driver Security Matters	3627
31.4.1 What the Kernel Trusts	3627
31.4.2 A Kernel Bug Changes the Trust Model	3628
31.4.3 A Partial Catalogue of What Is at Risk	3628
31.4.4 Real-World Incidents	3629
31.4.5 The Security Mindset	3630
31.4.6 Even Root Is Not Trusted	3630
31.4.7 Where the Boundaries Are	3630
31.4.8 Where This Chapter Fits	3631
31.4.9 Wrapping Up Section 1	3631
31.5 Section 2: Avoiding Buffer Overflows and Memory Corruption	3632
31.5.1 A Short Refresher on Buffers	3632
31.5.2 How Overflows Happen in Kernel Code	3632
31.5.3 Bound Everything	3633
31.5.4 The sizeof(buf) Trap	3634
31.5.5 Bounded String Functions	3635
31.5.6 Arithmetic and Overflow	3635
31.5.7 Integer Types Matter	3636
31.5.8 Stack Buffers Are Cheap but Limited	3636
31.5.9 Heap Buffers and Their Lifetimes	3637
31.5.10 A Worked Example: Safe and Unsafe Copy Routines	3638
31.5.11 A Second Worked Example: The Descriptor Length	3639
31.5.12 Buffer Overflow as a Class of Bug	3640
31.5.13 Memory Corruption Beyond Overflows	3640
31.5.14 How Compiler Protections Help, and Where They Stop	3641
31.5.15 Wrapping Up Section 2	3642

31.6	Section 3: Handling User Input Safely	3642
31.6.1	The User-Kernel Boundary	3643
31.6.2	copyin(9) and copyout(9)	3643
31.6.3	copyinstr(9) for Strings	3644
31.6.4	uiomove(9) for Streams	3645
31.6.5	Validate Every Field of Every Structure	3646
31.6.6	Validate Structures That Come In Multiple Parts	3648
31.6.7	The “frozen” Problem	3649
31.6.8	Kernel Addresses Do Not Leak Into User Pointers	3649
31.6.9	compat32 and Structure Sizes	3649
31.6.10	A Larger Example: A Complete ioctl Handler	3650
31.6.11	Common Mistakes in User-Input Handling	3651
31.6.12	A Detailed Walkthrough: Designing a Safe ioctl from Scratch	3652
31.6.13	Wrapping Up Section 3	3655
31.7	Section 4: Secure Use of Memory Allocation	3655
31.7.1	malloc(9) in the Kernel	3655
31.7.2	When to Use M_WAITOK and When to Use M_NOWAIT	3656
31.7.3	M_ZERO Is Your Friend	3657
31.7.4	uma_zone for High-Frequency Allocations	3657
31.7.5	Use-After-Free: What It Is and Why It Matters	3658
31.7.6	Double-Free: What It Is and Why It Matters	3659
31.7.7	Memory Leaks Are a Security Problem	3659
31.7.8	explicit_bzero and zfree for Sensitive Data	3660
31.7.9	malloc_type Tags and Accountability	3660
31.7.10	The Lifetime of Softc	3661
31.7.11	A Complete Allocation/Deallocation Pattern	3662
31.7.12	A Closer Look at UMA Zones	3663
31.7.13	Reference Counting for Shared Objects	3665
31.7.14	Wrapping Up Section 4	3665
31.8	Section 5: Preventing Race Conditions and TOCTOU Bugs	3665
31.8.1	How Races Arise in Drivers	3666
31.8.2	The TOCTOU Pattern	3666
31.8.3	Real-World Example: ioctl with a Path	3667
31.8.4	Shared State and Locking	3668
31.8.5	A Closer Look: Fixing a Racy Driver	3668
31.8.6	Lifetime Races at Detach	3670
31.8.7	Atomics and Lock-Free Code	3671
31.8.8	Refcounts as a Lifetime Tool	3671
31.8.9	Ordering and Memory Barriers	3671
31.8.10	Signal and Sleep Safety	3672
31.8.11	Rate Limiting and Resource Exhaustion	3672
31.8.12	Epoch-Based Reclamation: A Lock-Free Reader Idiom	3672
31.8.13	Wrapping Up Section 5	3673
31.9	Section 6: Access Control and Privilege Enforcement	3673
31.9.1	The Caller’s Credential: struct ucred	3674
31.9.2	priv_check and priv_check_cred	3674
31.9.3	Jail Awareness	3675
31.9.4	Securelevel	3675
31.9.5	Layering Checks	3676
31.9.6	The Credential on Open, ioctl, and Other Paths	3677
31.9.7	Device File Permissions	3678

31.9.8	Devfs Rules	3678
31.9.9	The MAC Framework	3678
31.9.10	Capsicum and Capability Mode	3679
31.9.11	Sysctls With Security Flags	3680
31.9.12	A Complete Privilege-Gated Ioctl	3680
31.9.13	Wrapping Up Section 6	3681
31.10	Section 7: Protecting Against Information Leaks	3682
31.10.1	How Information Leaks Happen	3682
31.10.2	The Padding Problem	3682
31.10.3	The Heap Allocation Case	3683
31.10.4	Oversized Replies	3684
31.10.5	Strings and the NUL Terminator	3684
31.10.6	Sensitive Data: Explicit Zeroing Before Freeing	3685
31.10.7	Never Leak Kernel Pointers	3686
31.10.8	Sysctl Output	3686
31.10.9	copyout Errors	3686
31.10.10	Deliberate Disclosure	3686
31.10.11	Kernel Pointer Hashing	3687
31.10.12	Wrapping Up Section 7	3687
31.11	Section 8: Safe Logging and Debugging	3687
31.11.1	The Logging Primitives	3687
31.11.2	What to Log and What Not to Log	3688
31.11.3	Rate Limiting	3688
31.11.4	Log Levels in Practice	3689
31.11.5	Debug Logging and Production	3689
31.11.6	Leaving Nothing Behind	3690
31.11.7	dmesg and the Kernel Message Buffer	3690
31.11.8	KTR and Tracing	3691
31.11.9	Logging Privileged Operations	3691
31.11.10	The audit(4) Framework	3691
31.11.11	Using dtrace with Your Driver	3692
31.11.12	Wrapping Up Section 8	3692
31.12	Section 9: Secure Defaults and Fail-Safe Design	3692
31.12.1	Fail Closed	3693
31.12.2	Whitelist, Don't Blacklist	3694
31.12.3	Smallest Useful Interface	3694
31.12.4	Least Privilege on Open	3694
31.12.5	Conservative Default Values	3695
31.12.6	Defense in Depth	3695
31.12.7	Timeouts and Watchdogs	3695
31.12.8	Bounded Resource Use	3696
31.12.9	Safe Module Load and Unload	3696
31.12.10	Safe Concurrent Entries	3696
31.12.11	Error Paths Are Normal Paths	3697
31.12.12	Don't Trust Yourself Either	3698
31.12.13	Graceful Degradation vs Full Refusal	3698
31.12.14	Case Study: Real-World Secure Defaults in /dev/null	3699
31.12.15	Wrapping Up Section 9	3699
31.13	Section 10: Testing and Hardening Your Driver	3700
31.13.1	A Walkthrough: Finding a Bug with KASAN	3700
31.13.2	Build With Kernel Sanitizers	3701

31.13.3 Stress Testing	3702
31.13.4 Fuzzing	3702
31.13.5 ASLR, PIE, and Stack Protection	3702
31.13.6 Reading Your Diff's	3703
31.13.7 Static Analysis	3703
31.13.8 Code Review	3704
31.13.9 Testing After a Change	3704
31.13.10 Continuous Integration	3704
31.13.11 Treat Bug Reports Seriously	3704
31.13.12 Hardening Over Time	3705
31.13.13 Post-Incident: What To Do When a Bug Becomes a CVE	3705
31.13.14 Wrapping Up Section 10	3706
31.14 Hands-On Labs	3706
31.14.1 Lab 1: The Unsafe <code>secdev</code>	3707
31.14.2 Lab 2: Bounds-Check the Buffer	3708
31.14.3 Lab 3: Zero Before <code>copyout</code>	3709
31.14.4 Lab 4: Add Privilege Checks	3710
31.14.5 Lab 5: Rate-Limited Logging	3711
31.14.6 Lab 6: Safe Detach	3712
31.14.7 Lab 7: Secure Defaults Everywhere	3714
31.15 Challenge Exercises	3715
31.15.1 Challenge 1: Add a Multi-Step <code>ioctl</code>	3715
31.15.2 Challenge 2: Write a <code>syzkaller</code> Description	3715
31.15.3 Challenge 3: Detect Double Free in Your Own Code	3715
31.15.4 Challenge 4: Threat-Model an Existing Driver	3716
31.15.5 Challenge 5: Compare <code>/dev/null</code> and <code>/dev/mem</code>	3716
31.16 Troubleshooting and Common Mistakes	3716
31.16.1 "Sometimes it works, sometimes it doesn't"	3716
31.16.2 "The driver crashes on unload"	3716
31.16.3 "The <code>ioctl</code> works from root but not from my service account"	3717
31.16.4 "dmesg is flooding"	3717
31.16.5 "The structure comes back with garbage bytes"	3717
31.16.6 "We leak memory but I don't see where"	3717
31.16.7 "kldload succeeds but my device doesn't show up in <code>/dev</code> "	3717
31.16.8 "A simple C test passes, but a shell script that does the same thing in a loop fails"	3718
31.16.9 "KASAN says use-after-free but my <code>malloc/free</code> are balanced"	3718
31.16.10 "WITNESS complains about lock order"	3718
31.16.11 "vmstat -m shows a negative free count"	3718
31.16.12 "The driver works on amd64 but panics on arm64"	3719
31.16.13 "The driver compiles without errors but dmesg shows kernel build warnings"	3719
31.16.14 "The driver is intermittently slower than expected"	3719
31.17 Security Checklist for Driver Code Review	3719
31.17.1 Structural Checks	3719
31.17.2 Input and Boundary Checks	3720
31.17.3 Memory Management	3720
31.17.4 Privilege and Access Control	3720
31.17.5 Concurrency	3721
31.17.6 Information Hygiene	3721
31.17.7 Failure Modes	3721
31.17.8 Testing	3721
31.17.9 Evolution	3722

31.18A Closer Look at Real-World Vulnerability Patterns	3722
31.18.1 The Incomplete Copy	3722
31.18.2 The Sign Confusion	3723
31.18.3 The Incomplete Validation	3723
31.18.4 The Skipped Check on the Error Path	3724
31.18.5 The Double-Lookup	3724
31.18.6 The Buffer That Grew	3724
31.18.7 The Unchecked Pointer from a Structure	3725
31.18.8 The Forgotten copyout	3725
31.18.9 The Accidental Race	3726
31.18.10 The Silent Truncation	3726
31.18.11 The Over-Logged Event	3727
31.18.12 The Invisible Bug	3727
31.18.13 Wrapping Up the Pattern Catalogue	3727
31.19 Appendix: Headers and APIs Used in This Chapter	3727
31.19.1 Memory and Copy Operations	3728
31.19.2 Privilege and Access Control	3728
31.19.3 Locking and Concurrency	3728
31.19.4 Device Files and Dev Infrastructure	3728
31.19.5 Timing, Rate Limiting, Callouts	3728
31.19.6 Logging and Diagnostics	3728
31.19.7 Sysctls	3729
31.19.8 Network (when applicable)	3729
31.19.9 Epoch and Lock-Free	3729
31.19.10 Tracing and Observability	3729
31.20 Appendix: Further Reading	3730
31.21 Wrapping Up	3731
31.21.1 One More Reflection: Security as Professional Identity	3732
31.22 Looking Ahead: Device Tree and Embedded Development	3732
31.23A Final Note on Habits	3733
31.24A Note on Evolving Threats	3734
31.25 Closing Words	3734
32 Device Tree and Embedded Development	3736
32.1 Introduction	3736
32.2 Reader Guidance: How to Use This Chapter	3738
32.2.1 Prerequisites	3738
32.2.2 Structure and Pacing	3739
32.2.3 Work Section by Section	3739
32.2.4 Keep the Lab Driver Close	3739
32.2.5 Open the FreeBSD Source Tree	3739
32.2.6 Keep a Lab Logbook	3740
32.2.7 Pace Yourself	3740
32.3 How to Get the Most Out of This Chapter	3740
32.3.1 Read Real Source	3740
32.3.2 Run What You Build	3740
32.3.3 Type the Labs	3741
32.3.4 Follow the Nodes	3741
32.3.5 Treat dmesg as Part of the Manuscript	3741
32.3.6 Break Things on Purpose	3741
32.3.7 Work in Pairs When You Can	3741
32.3.8 Trust the Iteration	3741

32.3.9	Take Breaks	3742
32.4	Section 1: Introduction to Embedded FreeBSD Systems	3742
32.4.1	What Embedded FreeBSD Looks Like	3742
32.4.2	Why Embedded Platforms Rely on Device Trees	3743
32.4.3	Architecture Overview: FreeBSD on ARM, RISC-V, and MIPS	3744
32.4.4	Typical Limitations: No ACPI, Limited Buses, Power Constraints	3744
32.4.5	Boot Flow: From Firmware to Driver Attachment	3745
32.4.6	Where to See It for Yourself	3746
32.4.7	Wrapping Up This Section	3746
32.5	Section 2: What Is the Device Tree?	3746
32.5.1	A Minimal Device Tree	3746
32.5.2	The Tree Structure	3747
32.5.3	Reading the Minimal Example	3748
32.5.4	Source vs Binary: .dts, .dtsi, and .dtb	3749
32.5.5	Nodes, Properties, Addresses, and the Phandle	3750
32.5.6	A More Realistic Example	3751
32.5.7	How the DTB Is Loaded	3752
32.5.8	Wrapping Up This Section	3752
32.6	Section 3: FreeBSD's Device Tree Support	3753
32.6.1	The fdt(4) Framework Overview	3753
32.6.2	OF_*: The Raw Property Readers	3753
32.6.3	ofw_bus: The Compatibility Helpers	3754
32.6.4	simplebus: The Default Enumerator	3755
32.6.5	Mapping the Pieces to a Probe Call	3756
32.6.6	Registering a Driver with ofw_bus	3756
32.6.7	Writing a Compatibility Table	3757
32.6.8	Reading Properties	3758
32.6.9	Acquiring Resources	3758
32.6.10	Peripheral Frameworks: Clock, Regulator, Reset, Pinctrl, GPIO	3759
32.6.11	Interrupt Routing: A Quick Look at interrupt-parent	3759
32.6.12	A Brief Look at Overlays	3760
32.6.13	ACPI vs FDT on arm64	3760
32.6.14	Wrapping Up This Section	3760
32.7	Section 4: Writing a Driver for an FDT-Based System	3760
32.7.1	The Header Includes	3761
32.7.2	The Softc	3762
32.7.3	The Compatibility Table	3762
32.7.4	The Probe Routine	3763
32.7.5	The Attach Routine	3763
32.7.6	The Detach Routine	3766
32.7.7	Interrupt Handler	3767
32.7.8	Module Registration	3767
32.7.9	The Full Skeleton Assembled	3768
32.7.10	What the Kernel Does Next	3770
32.7.11	Checking Your Work	3770
32.7.12	A Note on Naming and Vendor Prefixes	3771
32.7.13	Wrapping Up This Section	3771
32.8	5. Creating and Modifying Device Tree Blobs	3771
32.8.1	The Source-to-Binary Pipeline	3771
32.8.2	Installing the Tools	3772
32.8.3	Writing a Standalone .dts File	3773

32.8.4	Compiling a .dts File	3774
32.8.5	The Role of .dtsi Include Files	3775
32.8.6	Understanding Overlays	3775
32.8.7	Compiling and Deploying Overlays	3776
32.8.8	A Walkthrough: Adding a Node to a Raspberry Pi 4 Tree	3777
32.8.9	Inspecting the Result	3778
32.8.10	Troubleshooting Build Failures	3779
32.8.11	The Kernel's dtb Loading Process	3779
32.8.12	Wrapping Up This Section	3779
32.9	6. Testing and Debugging FDT Drivers	3780
32.9.1	The ofwdump(8) Tool	3780
32.9.2	Reading the Raw Blob Through sysctl	3781
32.9.3	Confirming FDT Mode on arm64	3781
32.9.4	Debugging a Probe That Does Not Fire	3781
32.9.5	Adding Temporary Debug Output	3782
32.9.6	Observing Attach Order and Dependencies	3783
32.9.7	Debugging Detach and Unload	3784
32.9.8	Using DTrace for Deeper Visibility	3784
32.9.9	Testing on QEMU	3784
32.9.10	When to Stop Debugging and Rebuild	3785
32.9.11	Wrapping Up This Section	3785
32.10.7	Practical Example: GPIO Driver for an Embedded Board	3785
32.10.1	What You Need	3786
32.10.2	The Overall File Layout	3786
32.10.3	The Softc	3786
32.10.4	Headers	3787
32.10.5	The Compatibility Table	3787
32.10.6	The Probe	3788
32.10.7	The Attach	3788
32.10.8	The Sysctl Handler	3790
32.10.9	The Detach	3790
32.10.10	The Method Table and Driver Registration	3791
32.10.11	The Full Source	3791
32.10.12	The Makefile	3795
32.10.13	The Overlay Source	3795
32.10.14	Building and Loading	3796
32.10.15	Inspecting the Resulting Device	3796
32.10.16	Pitfalls Worth Calling Out	3797
32.10.17	Wrapping Up This Section	3797
32.11	8. Refactoring and Finalizing Your Embedded Driver	3797
32.11.1	What Refactoring Means Here	3798
32.11.2	Pass One: Tighten the Attach Error Paths	3798
32.11.3	Pass Two: Add Proper Locking	3799
32.11.4	Pass Three: Handle the Power Rail Explicitly	3800
32.11.5	Pass Four: Pinmux Setup	3802
32.11.6	Pass Five: Style and Naming Audit	3803
32.11.7	Pass Six: Static Analysis	3803
32.11.8	Pass Seven: Documentation	3804
32.11.9	Packaging and Distribution	3804
32.11.10	Reviewing Against Real FreeBSD Drivers	3804
32.11.11	Wrapping Up This Section	3805

32.129. Reading Real FreeBSD FDT Drivers	3805
32.12.1 gpioled_fdt.c: A Close Relative of edled	3805
32.12.2 DRIVER_MODULE Registration	3807
32.12.3 What to Take Away from gpioled_fdt.c	3807
32.12.4 bcm2835_gpio.c: A Bus Provider	3808
32.12.5 ofw_iicbus.c: A Bus That Is Both Parent and Child	3808
32.12.6 ofw_bus_subr.c: The Helpers Themselves	3809
32.12.7 simplebus.c: The Driver That Runs Your Driver	3809
32.12.8 A Survey of Drivers Worth Reading	3810
32.12.9 Using grep as a Study Tool	3810
32.12.10 Wrapping Up This Section	3811
32.1310. Interrupt Plumbing in FDT-Based Systems	3811
32.13.1 The Interrupt Tree	3811
32.13.2 Interrupt-Parent Chaining	3812
32.13.3 The intrng Framework	3813
32.13.4 Requesting an IRQ in Practice	3813
32.13.5 Filter Versus Thread Handlers	3815
32.13.6 Edge Versus Level Triggering	3815
32.13.7 Debugging Interrupts That Do Not Fire	3816
32.13.8 Real Example: gpiokeys.c	3816
32.13.9 Wrapping Up This Section	3816
32.14 Hands-on Labs	3816
32.14.1 Lab 1: Build and Load the fdthello Skeleton	3817
32.14.2 Lab 2: Build and Deploy an Overlay	3818
32.14.3 Lab 3: Debug a Broken Device Tree	3820
32.14.4 Lab 4: Build the edled Driver End to End	3822
32.14.5 Lab 5: Extend edled to Consume a GPIO Interrupt	3823
32.14.6 After the Labs	3826
32.15 Challenge Exercises	3826
32.15.1 Challenge 1: Multiple LEDs Per Node	3826
32.15.2 Challenge 2: Support a Blink Timer	3826
32.15.3 Challenge 3: Generalize to an Arbitrary GPIO Output	3827
32.15.4 Challenge 4: Consume an Interrupt	3827
32.15.5 Challenge 5: Produce a Custom Device Tree for QEMU	3827
32.15.6 Challenge 6: Port a Real Driver's DT Binding to a New Board	3827
32.15.7 Challenge 7: Hand-Write a Full DT for an Embedded Target Under QEMU	3828
32.15.8 Challenge 8: Implement a Simple MMIO Peripheral Driver	3828
32.15.9 After the Challenges	3828
32.16 Common Mistakes and Troubleshooting	3828
32.16.1 Module Loads but No Device Attaches	3828
32.16.2 Overlay Fails to Apply at Boot	3829
32.16.3 Cannot Allocate Resources	3829
32.16.4 GPIO Acquisition Fails	3829
32.16.5 Interrupt Handler Does Not Fire	3830
32.16.6 Unload Returns EBUSY	3830
32.16.7 Panic During Boot	3830
32.16.8 DT Compile Errors	3831
32.16.9 The Kernel Sees the Wrong Hardware	3831
32.16.10 You Changed the Driver But Nothing Changed	3831
32.16.11 Quick Reference: Most-Used OFW Calls	3831
32.16.12 Quick Reference: Most-Used Peripheral Calls	3832

32.16.1 Summary Checklist	3834
32.17 Glossary of Device Tree and Embedded Terms	3834
32.18 Frequently Asked Questions	3836
32.19 Wrapping Up	3838
32.19.1 What You Should Be Able to Do Now	3839
32.19.2 What This Chapter Leaves Implicit	3839
32.19.3 Key Takeaways	3840
32.19.4 Before You Move On	3840
32.19.5 A Note on Sustained Practice	3841
32.19.6 Connecting Back to the Rest of the Book	3841
32.19.7 Looking Ahead	3841
32.19.8 A Parting Word	3842
33 Performance Tuning and Profiling	3843
33.1 Introduction	3843
33.2 Reader Guidance: How to Use This Chapter	3844
33.2.1 Prerequisites	3845
33.2.2 Structure and Pacing	3845
33.2.3 Work Section by Section	3846
33.2.4 Keep the Lab Driver Close	3846
33.2.5 Open the FreeBSD Source Tree	3846
33.2.6 Keep a Lab Logbook	3847
33.2.7 Pace Yourself	3847
33.3 How to Get the Most Out of This Chapter	3847
33.3.1 Measure Before You Change	3847
33.3.2 State the Goal in Numbers	3847
33.3.3 Prefer the Simplest Tool That Answers the Question	3847
33.3.4 Never Instrument the Hot Path at Production Cost	3847
33.3.5 Read the Manual Pages You Use	3848
33.3.6 Type the Lab Code	3848
33.3.7 Tag Every Measurement with Its Context	3848
33.3.8 Do Not Tune What Is Not Slow	3848
33.3.9 Follow the Numbers Between Sections	3848
33.4 Section 1: Why Performance Matters in Device Drivers	3848
33.4.1 The Four Axes of Driver Performance	3849
33.4.2 A Worked Example: Where Do These Axes Show Up?	3849
33.4.3 When Optimisation Is Worth Doing	3850
33.4.4 When Optimisation Is Premature	3850
33.4.5 Setting Measurable Goals	3851
33.4.6 A Note on “Fast” and “Correct”	3851
33.4.7 Exercise 33.1: Define Performance Goals for a Driver You Have	3852
33.4.8 Wrapping Up Section 1	3852
33.5 Section 2: Measuring Performance in the Kernel	3852
33.5.1 Kernel Time Functions	3853
33.5.2 Time Stamp Counters and Other High-Precision Sources	3854
33.5.3 Composition: Timing, Counting, and Aggregating Together	3855
33.5.4 A Complete Instrumented Read Path	3857
33.5.5 Counters: Simple, Atomic, Per-CPU	3858
33.5.6 Exposing Metrics via sysctl	3859
33.5.7 The Measurement Tools FreeBSD Gives You	3860
33.5.8 The Heisenberg Effect, Practically	3860
33.5.9 Instrumenting Without Contamination	3861

33.5.10	Exercise 33.2: Time the Read Path	3862
33.5.11	Wrapping Up Section 2	3862
33.6	Section 3: Using DTrace to Analyze Driver Behavior	3862
33.6.1	What DTrace Is	3862
33.6.2	Enabling DTrace on FreeBSD	3863
33.6.3	The Probe Name Format	3863
33.6.4	Your First DTrace One-Liner	3864
33.6.5	Aggregations: the DTrace Superpower	3865
33.6.6	Useful DTrace Patterns for Driver Work	3865
33.6.7	Profile Sampling with the <code>profile</code> Provider	3866
33.6.8	Static Probes with SDT	3867
33.6.9	The <code>lockstat</code> Provider	3868
33.6.10	A Longer DTrace Script	3869
33.6.11	The <code>sched</code> Provider for Scheduler Events	3870
33.6.12	The <code>io</code> Provider for Storage Drivers	3871
33.6.13	Combining Multiple Providers in One Script	3872
33.6.14	Putting It All Together: Driver Trace Session	3873
33.6.15	A Note on DTrace's Limitations	3873
33.6.16	Exercise 33.3: DTrace Your Driver	3874
33.6.17	Wrapping Up Section 3	3874
33.7	Section 4: Using <code>pmcstat</code> and CPU Counters	3874
33.7.1	Hardware Performance Counters in One Page	3875
33.7.2	<code>hwpmc(4)</code> and <code>pmcstat(8)</code>	3875
33.7.3	A First Sampling Session	3876
33.7.4	Sampling Event Choice	3876
33.7.5	Callgraphs and Flame Graphs	3877
33.7.6	Interpreting a PMC Result	3877
33.7.7	Reading a Callgraph in Detail	3878
33.7.8	Cross-Referencing <code>pmcstat</code> and DTrace	3878
33.7.9	A Top-Down Performance Analysis Method	3879
33.7.10	Counting Mode	3879
33.7.11	Process-Attached vs System-Wide	3880
33.7.12	<code>pmcstat</code> and Virtualisation	3880
33.7.13	Exercise 33.4: <code>pmcstat perfdemo</code>	3881
33.7.14	Wrapping Up Section 4	3881
33.8	Section 5: Buffering and Memory Optimization	3881
33.8.1	Cache Lines and False Sharing	3881
33.8.2	Alignment for DMA	3882
33.8.3	Pre-Allocated Buffers	3883
33.8.4	UMA Zones	3884
33.8.5	A Tour of UMA Internals	3885
33.8.6	Bounce Buffers and DMA Memory Mapping	3886
33.8.7	Per-CPU Counters Revisited	3887
33.8.8	Per-CPU Data Patterns Beyond Counters	3888
33.8.9	When Not to Align, Allocate, or Cache	3890
33.8.10	Exercise 33.5: Cache-Line and Per-CPU Tuning	3890
33.8.11	Wrapping Up Section 5	3891
33.9	Section 6: Interrupt and Taskqueue Optimization	3891
33.9.1	Measuring Interrupt Behaviour	3891
33.9.2	Filter Handlers vs Ithread Handlers	3892
33.9.3	Moving Work to Taskqueues	3893

33.9.4	Interrupt Coalescing	3894
33.9.5	Debouncing	3895
33.9.6	Dedicated Threads for Long Work	3895
33.9.7	MSI and MSI-X: Per-Vector Interrupts	3895
33.9.8	Polling as an Alternative to Interrupts	3897
33.9.9	NAPI-Style Batching in Practice	3897
33.9.10	Budget Allocation Across Stages	3898
33.9.11	Exercise 33.6: Interrupt vs Taskqueue Latency	3899
33.9.12	Wrapping Up Section 6	3899
33.10	Section 7: Using sysctl and Logging for Runtime Metrics	3899
33.10.1	Designing a sysctl Tree	3900
33.10.2	Exposing counter(9) Values via sysctl	3901
33.10.3	What to Expose	3901
33.10.4	Read-Only, Read-Write, and Tunable	3902
33.10.5	Rate-Limited Logging with log(9)	3903
33.10.6	device_printf(9) and Its Friends	3903
33.10.7	Debug Modes	3904
33.10.8	Tracking Behaviour Over Time	3905
33.10.9	Exposing Histograms via sysctl	3905
33.10.10	Per-CPU Sysctl Presentation	3906
33.10.11	Stable Interfaces for Operators	3907
33.10.12	Handling Sysctl Writes Safely	3907
33.10.13	Exercise 33.7: A Reporting sysctl Tree	3908
33.10.14	Wrapping Up Section 7	3909
33.11	Section 8: Refactoring and Versioning After Tuning	3909
33.11.1	Removing Temporary Measurement Code	3909
33.11.2	What to Keep	3910
33.11.3	Benchmarking the Final Driver	3910
33.11.4	Documenting Tuning Knobs	3911
33.11.5	Versioning	3911
33.11.6	Reviewing the Diff	3912
33.11.7	The Performance Report	3912
33.11.8	A/B Testing Tuning Changes	3913
33.11.9	The Benchmark Harness	3914
33.11.10	Sharing Performance Numbers With the Project	3915
33.11.11	Exercise 33.8: Ship perfdemo v2.3	3916
33.11.12	Wrapping Up Section 8	3916
33.12	Putting It All Together: A Complete Tuning Session	3916
33.13	Hands-On Labs	3919
33.13.1	Lab 1: Baseline perfdemo, Counters, and sysctl	3919
33.13.2	Lab 2: DTrace perfdemo	3920
33.13.3	Lab 3: pmcstat perfdemo (optional, requires hwpmc)	3922
33.13.4	Lab 4: Cache Alignment and Per-CPU Counters	3924
33.13.5	Lab 5: Interrupt and Taskqueue Split	3927
33.13.6	Lab 6: Ship v2.3-optimized	3928
33.13.7	Wrapping Up the Labs	3930
33.13.8	A Lab Summary Table	3930
33.13.9	Lab Dependencies	3931
33.14	Challenge Exercises	3931
33.14.1	Challenge 1: Latency Budget for a Real Driver	3931
33.14.2	Challenge 2: Compare Three Allocators	3932

33.14.3 Challenge 3: Write a Useful DTrace Script	3932
33.14.4 Challenge 4: Instrument a Read Path Without Contaminating It	3933
33.14.5 Challenge 5: Expose a Live Histogram	3933
33.14.6 Challenge 6: Tune for Two Workloads	3934
33.14.7 Challenge 7: Write the Manual Page	3935
33.14.8 Challenge 8: Profile an Unfamiliar Driver	3936
33.14.9 Wrapping Up the Challenges	3936
33.15 Performance Patterns Reference	3936
33.15.1 Pattern: You Need a Hot-Path Counter	3937
33.15.2 Pattern: You Need a Hot-Path Allocation	3937
33.15.3 Pattern: You Need a Pre-Allocated Pool	3937
33.15.4 Pattern: You Need to Avoid False Sharing	3938
33.15.5 Pattern: You Need to Measure Latency Without Polluting It	3938
33.15.6 Pattern: You Need to Expose Driver State	3938
33.15.7 Pattern: You Need to Split an Interrupt Handler	3939
33.15.8 Pattern: You Need to Rate-Limit a Log Message	3939
33.15.9 Pattern: You Need a Stable Operator Interface	3939
33.15.10 Pattern: You Need to Ship After Tuning	3940
33.15.11 A Short Tool Reference Table	3940
33.15.12 A Short Primitive Reference Table	3941
33.15.13 Wrapping Up the Patterns Reference	3942
33.16 Troubleshooting and Common Mistakes	3942
33.16.1 DTrace Reports Zero Samples	3942
33.16.2 DTrace Drops Events	3943
33.16.3 pmcstat Reports <i>No Device</i> or <i>No PMCs</i>	3943
33.16.4 pmcstat Sampling Misses Hot Functions	3943
33.16.5 Counter Values Look Impossible	3943
33.16.6 sysctl Handler Crashes	3944
33.16.7 The Driver Slows Down Under Measurement	3944
33.16.8 Taskqueue Thread Starvation	3944
33.16.9 Interrupt Storm Detected	3945
33.16.10 Cache-Line Alignment Has No Effect	3945
33.16.11 UMA Zone Leaks	3945
33.16.12 The Benchmark Is Noisy	3945
33.16.13 You Measured the Wrong Thing	3946
33.16.14 Numbers Get Worse After a Tuning Change	3946
33.16.15 Throughput Climbs but P99 Latency Gets Worse	3947
33.16.16 Results Differ Between Hardware Configurations	3947
33.16.17 KASSERT Fires Only Under Load	3947
33.16.18 Measurement Disagrees With User Perception	3948
33.16.19 DTrace Script Hangs or Takes a Long Time to Exit	3948
33.16.20 Sysctl Reports Different Numbers on Successive Reads	3948
33.16.21 A Driver Runs Fast Alone, Slow in Production	3949
33.16.22 Wrapping Up the Troubleshooting	3949
33.17 Wrapping Up	3949
33.17.1 What You Should Be Able to Do Now	3951
33.17.2 What This Chapter Leaves Implicit	3951
33.17.3 Key Takeaways	3951
33.17.4 Before You Move On	3952
33.17.5 A Note on Sustained Practice	3952
33.17.6 Connecting Back to the Rest of the Book	3952

33.17.7 Looking Ahead	3953
33.17.8 Glossary of Terms Introduced in This Chapter	3953
33.17.9 A Parting Word	3956
34 Advanced Debugging Techniques	3957
34.1 Introduction	3957
34.2 Reader Guidance: How to Use This Chapter	3958
34.2.1 Reading This Chapter Alongside Chapter 23	3958
34.3 How to Get the Most Out of This Chapter	3959
34.4 1. When and Why You Need Advanced Debugging	3959
34.4.1 Bugs That Need More Than a printf	3959
34.4.2 The Cost of Advanced Tooling	3960
34.4.3 A Decision Framework	3961
34.4.4 Meeting the bugdemo Driver	3961
34.5 2. Using KASSERT, panic, and Related Macros	3961
34.5.1 KASSERT: A Check That Vanishes in Production	3962
34.5.2 INVARIANTS and INVARIANT_SUPPORT	3963
34.5.3 MPASS: KASSERT With a Default Message	3963
34.5.4 CTASSERT: Compile-Time Assertions	3963
34.5.5 panic: The Unconditional Stop	3964
34.5.6 __predict_false and __predict_true	3964
34.5.7 Where Assertions Belong in a Driver	3965
34.5.8 When Not to Use Assertions	3966
34.5.9 A Small Real-World Example	3966
34.5.10 A Quick Example: Adding Assertions to bugdemo	3967
34.6 3. Analyzing Panics and Crash Dumps	3968
34.6.1 What Happens During a Panic	3968
34.6.2 Reading a Panic Message	3968
34.6.3 Configuring the Dump Device	3969
34.6.4 Enabling Dumps in the Kernel	3970
34.6.5 Retrieving the Dump with savecore	3971
34.6.6 Opening a Dump with kgdb	3972
34.6.7 Useful kgdb Commands	3973
34.6.8 Walking struct thread in a Dump	3974
34.6.9 Walking struct proc in a Dump	3975
34.6.10 Using kgdb Against a Live Kernel	3976
34.6.11 A Note on Symbols and Modules	3977
34.6.12 Closing Thoughts on Dumps	3977
34.7 4. Building a Debug-Friendly Kernel Environment	3978
34.7.1 Building GENERIC-DEBUG	3978
34.7.2 What GENERIC-DEBUG Enables	3978
34.7.3 makeoptions DEBUG=-g	3979
34.7.4 INVARIANTS and INVARIANT_SUPPORT	3979
34.7.5 WITNESS: Lock Order Verifier	3980
34.7.6 A Worked Race-Condition Walkthrough: Lock-Order Bug in bugdemo	3981
34.7.7 The Same Bug Seen Through lockstat(1)	3982
34.7.8 Reading WITNESS and lockstat Together	3984
34.7.9 MALLOC_DEBUG_MAXZONES	3984
34.7.10 ALT_BREAK_TO_DEBUGGER and BREAK_TO_DEBUGGER	3984
34.7.11 DEADLKRES: The Deadlock Detector	3985
34.7.12 BUF_TRACKING	3985
34.7.13 QUEUE_MACRO_DEBUG_TRASH	3985

34.7.14	Memory Debuggers: DEBUG_MEMGUARD and DEBUG_REDZONE	3985
34.7.15	KDB, DDB, and GDB Together	3986
34.7.16	KDB_UNATTENDED	3986
34.7.17	CTF and Debug Info Paths	3986
34.7.18	Confirming Your Debug Kernel	3987
34.7.19	Running on Top of the Debug Kernel	3987
34.8	5. Tracing Driver Behavior: DTrace, ktrace, and ktr(9)	3988
34.8.1	DTrace for Correctness Debugging	3988
34.8.2	The fbt Provider	3988
34.8.3	SDT Probes: Statically Defined Tracing	3989
34.8.4	Following a Bug With DTrace	3990
34.8.5	Understanding ktrace(1)	3991
34.8.6	ktr(9): Lightweight In-Kernel Tracing	3992
34.8.7	When to Use Which	3993
34.8.8	Tracing and Production	3994
34.9	6. Hunting Memory Leaks and Invalid Memory Accesses	3994
34.9.1	Understanding the Kernel Memory Allocator	3994
34.9.2	sysctl vm and kern.malloc: Observing Allocation State	3995
34.9.3	memguard(9): Hunting Use-After-Free	3995
34.9.4	A memguard Session in Action	3996
34.9.5	redzone: Buffer Overrun Detection	3996
34.9.6	Guard Pages in the VM Layer	3997
34.9.7	Leak Detection in Practice	3997
34.9.8	When Memory Bugs Hide	3998
34.9.9	Working With DDB on Memory Bugs	3999
34.9.10	Reality Check on Memory Debuggers	3999
34.9.11	Wrapping Up the Memory Section	3999
34.10	7. Safe Debugging Practices	4000
34.10.1	The Development vs Production Divide	4000
34.10.2	What Is Safe in Production	4000
34.10.3	What Is Unsafe in Production	4001
34.10.4	Capturing Evidence on Production Systems	4001
34.10.5	Using log(9) Over printf for Diagnostics	4002
34.10.6	Do Not Mix Debug and Release Kernels in One Fleet	4002
34.10.7	Working With a Recovery Plan	4003
34.10.8	Keep a Debug Journal	4003
34.10.9	Practicing on Your Own Drivers	4003
34.10.10	Knowing When to Stop	4003
34.11	8. Refactoring After a Debugging Session: Recovery and Resilience	4004
34.11.1	Reading the Bug as a Message	4004
34.11.2	Strengthening Invariants	4004
34.11.3	Documenting the Ownership Model	4005
34.11.4	Narrowing the API Surface	4005
34.11.5	Hardening the Unload Path	4005
34.11.6	The Shape of a Resilient Driver	4006
34.11.7	Revisiting the bugdemo Driver	4006
34.11.8	Closing the Refactoring Loop	4006
34.12	Hands-on Labs	4007
34.12.1	Lab 1: Adding Assertions to bugdemo	4007
34.12.2	Lab 2: Capturing and Analyzing a Panic With kgdb	4009
34.12.3	Lab 3: Building GENERIC-DEBUG and Confirming Options Active	4010

34.12.4 Lab 4: Tracing bugdemo With DTrace and ktrace	4012
34.12.5 Lab 5: Catching a Use-After-Free With memguard	4014
34.12.6 Lab 6: Remote Debugging With the GDB Stub	4015
34.13 Challenge Exercises	4016
34.13.1 Challenge 1: Find the Silent Bug	4017
34.13.2 Challenge 2: Hunt the Leak	4017
34.13.3 Challenge 3: Diagnose the Deadlock	4017
34.13.4 Challenge 4: Read a Real Panic	4017
34.13.5 Challenge 5: Build Your Own bugdemo Variant	4017
34.14 Troubleshooting Common Issues	4018
34.14.1 The Dump Is Not Being Captured	4018
34.14.2 kgdb Says “No Symbols”	4018
34.14.3 DDB Freezes the System	4018
34.14.4 A Module Refuses to Unload	4019
34.14.5 memguard Does Nothing	4019
34.14.6 DTrace Says “Probe Does Not Exist”	4019
34.14.7 Kernel Builds Fail With Symbol Conflicts	4019
34.14.8 The System Boots Into the Old Kernel	4020
34.14.9 WITNESS Reports False Positives	4020
34.15 Putting It All Together: A Debugging Session Walkthrough	4020
34.15.1 The Symptom	4020
34.15.2 Step 1: Collect Evidence	4021
34.15.3 Step 2: Form a Hypothesis	4021
34.15.4 Step 3: Test the Hypothesis With DTrace	4021
34.15.5 Step 4: Confirm With SDT Probes	4022
34.15.6 Step 5: Identify the Fix	4022
34.15.7 Step 6: Implement and Verify	4022
34.15.8 Lessons From This Session	4023
34.16 Additional Techniques Worth Knowing	4023
34.16.1 witness_checkorder With Manual Lists	4023
34.16.2 sysctl debug.ktr	4023
34.16.3 DDB Commands Beyond bt	4023
34.16.4 Kernel Option KDB_TRACE	4024
34.16.5 EKCD: Encrypted Kernel Crash Dumps	4024
34.16.6 Lightweight Debug Output: bootverbose	4024
34.17 A Closer Look at DDB	4024
34.17.1 Entering DDB	4024
34.17.2 DDB Prompts and Commands	4025
34.17.3 Walking a Thread	4025
34.17.4 Inspecting Structures	4025
34.17.5 Setting Breakpoints	4026
34.17.6 Scripting DDB	4026
34.17.7 Exiting DDB	4026
34.17.8 DDB vs kgdb: When to Use Each	4026
34.18 Practical Worked Example: Following a Null Pointer	4027
34.18.1 Step 1: Capture the Dump	4027
34.18.2 Step 2: Read the Backtrace	4027
34.18.3 Step 3: Inspect the Variables	4027
34.18.4 Step 4: Walk Back	4028
34.18.5 Step 5: Form a Hypothesis	4028
34.18.6 Step 6: Add an Assertion	4028

34.18.7 Step 7: Fix the Bug	4028
34.18.8 Step 8: Verify	4029
34.18.9 Takeaway	4029
34.19 Making Drivers Observable From Day One	4029
34.19.1 Name Every Allocator Type	4029
34.19.2 Count Important Events	4029
34.19.3 Declare SDT Probes	4029
34.19.4 Use Consistent Log Messages	4029
34.19.5 Provide Useful Sysctls	4030
34.19.6 Write Assertions As You Go	4030
34.19.7 Expose the State of the State Machine	4030
34.19.8 Test the Unload Path	4030
34.20 Real-World Reading List	4030
34.21 Wrapping Up	4031
34.22 Bridge to Chapter 35: Asynchronous I/O and Event Handling	4032
35 Asynchronous I/O and Event Handling	4033
35.1 Introduction	4033
35.2 Reader Guidance: How to Use This Chapter	4034
35.3 How to Get the Most Out of This Chapter	4035
35.4 1. Synchronous vs. Asynchronous I/O in Device Drivers	4035
35.4.1 The World Does Not Wait for read()	4036
35.4.2 Why Busy Waiting Is a Bad Answer	4036
35.4.3 Asynchronous I/O Is Not Just Non-blocking Read	4036
35.4.4 Real-World Patterns: Event Sources That Want Asynchronous I/O	4037
35.4.5 When Not to Use Asynchronous I/O	4038
35.4.6 A Mental Model for the Rest of the Chapter	4038
35.5 2. Introducing poll() and select()	4039
35.5.1 What Userland Expects From poll() and select()	4039
35.5.2 The d_poll Callback	4041
35.5.3 Registering the d_poll Method on the cdevsw	4042
35.5.4 What Real Drivers Look Like	4043
35.5.5 What the User Sees	4044
35.5.6 Wrapping Up Section 2	4044
35.6 3. Using selwakeup() and selrecord()	4045
35.6.1 struct selinfo	4045
35.6.2 What selrecord() Does	4046
35.6.3 What selwakeup() Does	4046
35.6.4 The Missed Wakeup Race	4047
35.6.5 Thinking About the Lock	4048
35.6.6 Common Mistakes	4048
35.6.7 Diagnostic Techniques	4049
35.6.8 Closing the Loop: evdemo With Poll Support	4049
35.6.9 Wrapping Up Section 3	4051
35.7 4. Supporting kqueue and EVFILT_READ/EVFILT_WRITE	4051
35.7.1 What kqueue Looks Like to Userland	4051
35.7.2 The Data Structures	4052
35.7.3 Initializing a knlist	4053
35.7.4 The d_kqfilter Callback	4054
35.7.5 The filterops Implementation	4054
35.7.6 Delivering Events Through KNOTE_LOCKED	4056
35.7.7 Detach: Cleaning Up knlist	4056

35.7.8	A Fuller Example: Pipes	4057
35.7.9	The Anatomy of struct knote	4058
35.7.10	Inside struct knlist: How the Driver's Waiting Room Works	4059
35.7.11	The knlist Lifecycle in Detail	4060
35.7.12	The kqfilter Callback Contract	4062
35.7.13	Worked Example: The /dev/klog Driver	4062
35.7.14	Worked Example: TTY Read and Write Filters	4064
35.7.15	Worked Example: evdev Detach Discipline	4066
35.7.16	The hint Parameter: What It Is and Why It Exists	4069
35.7.17	Delivering Events: KNOTE_LOCKED vs KNOTE_UNLOCKED, in Depth	4069
35.7.18	Common Pitfalls in Driver kqfilter Implementations	4070
35.7.19	A Mental Model for the kqueue Framework	4071
35.7.20	Reading kern_event.c: A Guide for the Curious	4071
35.7.21	Driver Patterns We Have Not Yet Used	4072
35.7.22	A Note on EV_CLEAR, EV_ONESHOT, and Edge-Triggered Behaviour	4073
35.7.23	Wrapping Up Section 4	4073
35.8	5. Asynchronous Signals With SIGIO and FIOASYNC	4074
35.8.1	How Signal-Driven I/O Works From Userland	4074
35.8.2	The FIOASYNC, FIOSETOWN, and FIOGETOWN Ioctl's	4076
35.8.3	fsetown, fgetown, and funsetown	4076
35.8.4	Implementing SIGIO in evdemo	4077
35.8.5	Caveats: Signal Semantics in Multi-Threaded Programs	4079
35.8.6	What Real Drivers Look Like	4079
35.8.7	Testing SIGIO From Shell	4080
35.8.8	A Note on POSIX AIO	4080
35.8.9	How AIO Dispatches: fo_aio_queue and aio_queue_file	4080
35.8.10	The Block Device Path: aio_qbio and Bio Callbacks	4081
35.8.11	The Worker Thread Path: aio_process_rw	4082
35.8.12	Completion: aio_complete, aio_cancel, and aio_return	4082
35.8.13	EVFILT_AIO: How AIO Uses kqueue	4083
35.8.14	Why kqueue Is the Right Answer for Most Drivers	4083
35.8.15	Reading vfs_aio.c: A Guide	4084
35.8.16	A Driver-Side Checklist	4084
35.8.17	Wrapping Up Section 5	4085
35.9	6. Internal Event Queues and Message Passing	4085
35.9.1	Designing the Event Record	4086
35.9.2	The Ring Buffer	4086
35.9.3	The Read Path	4088
35.9.4	Integrating the Producer Path	4089
35.9.5	The Poll and Kqueue Integration	4089
35.9.6	Exposing Queue Metrics Through sysctl	4090
35.9.7	Overflow Policies: A Design Discussion	4091
35.9.8	Avoiding Partial Reads	4092
35.9.9	Handling Event Coalescing	4092
35.9.10	Wrapping Up Section 6	4092
35.10	7. Combining Asynchronous Techniques	4092
35.10.1	When to Use Each Mechanism	4093
35.10.2	The Producer Path Template	4093
35.10.3	Lock Ordering	4094
35.10.4	The Consumer Paths	4095
35.10.5	Common Pitfalls	4095

35.10.6 Testing the Combined Design	4096
35.10.7 Application Compatibility	4096
35.10.8 Wrapping Up Section 7	4097
35.11 8. Final Refactor for Asynchronous Support	4097
35.11.1 Thread-Safety Review	4097
35.11.2 The Complete Attach Sequence	4098
35.11.3 The Complete Detach Sequence	4099
35.11.4 Exposing Event Metrics	4100
35.11.5 Versioning the Driver	4101
35.11.6 Wrapping Up Section 8	4101
35.12 Hands-On Labs	4101
35.12.1 Lab 1: Synchronous Baseline	4102
35.12.2 Lab 2: Adding poll() Support	4103
35.12.3 Lab 3: Adding kqueue Support	4104
35.12.4 Lab 4: Adding SIGIO Support	4106
35.12.5 Lab 5: The Event Queue	4108
35.12.6 Lab 6: The Combined v2.5-async Driver	4109
35.12.7 Lab 7: Unload Stress Test	4110
35.13 Challenge Exercises	4110
35.13.1 Challenge 1: Dual Mechanism Shootout	4111
35.13.2 Challenge 2: Multi-Reader Stress	4111
35.13.3 Challenge 3: Observe the Missed Wakeup Race	4111
35.13.4 Challenge 4: Event Coalescing	4112
35.13.5 Challenge 5: POLLHUP and POLLERR	4112
35.13.6 Challenge 6: evdev-Style Compatibility	4112
35.13.7 Challenge 7: Trace a kqueue Registration End to End	4112
35.13.8 Challenge 8: Observe the knlist Lock Discipline	4113
35.14 Troubleshooting Common Mistakes	4113
35.14.1 Symptom: poll() never returns	4113
35.14.2 Symptom: kqueue event fires but read() returns no data	4114
35.14.3 Symptom: SIGIO is delivered but the handler is not called	4114
35.14.4 Symptom: Kernel panic on module unload	4114
35.14.5 Symptom: Memory leak on repeated load/unload	4115
35.14.6 Symptom: Slow-to-wake poll() under load	4115
35.14.7 Symptom: Events are dropped under load	4115
35.14.8 Symptom: Only one reader wakes up even when multiple are waiting	4115
35.14.9 Symptom: The device hangs during detach	4115
35.14.10 Symptom: Reader wakes but finds nothing to do	4116
35.14.11 Symptom: Panic on module unload with “knlist not empty”	4116
35.14.12 Symptom: Panic in f_event with a stale kn_hook	4117
35.14.13 Symptom: KNOTE_LOCKED panics with a lock-not-held assertion	4117
35.14.14 Symptom: kqueue events arrive but kn_data is always zero	4117
35.14.15 Symptom: poll() works but kqueue never fires	4118
35.14.16 General Advice	4118
35.15 Wrapping Up	4118
35.16 Bridge to Chapter 36: Creating Drivers Without Documentation	4119
36 Creating Drivers Without Documentation (Reverse Engineering)	4121
36.1 Introduction	4121
36.2 Reader Guidance: How to Use This Chapter	4122
36.3 How to Get the Most Out of This Chapter	4123

36.4	1. Why Reverse Engineering May Be Necessary	4124
36.4.1	1.1 Legacy Hardware With No Vendor Support	4125
36.4.2	1.2 Devices With Binary-Only Drivers on Other Operating Systems	4125
36.4.3	1.3 Custom Embedded Systems With Little or No Documentation	4126
36.4.4	1.4 The Common Thread	4126
36.4.5	1.5 Legal and Ethical Considerations	4127
36.4.6	1.6 Wrapping Up Section 1	4128
36.5	2. Preparing for the Reverse Engineering Process	4128
36.5.1	2.1 The Mental Model: What the Lab Is For	4128
36.5.2	2.2 The Software Tool Kit From the FreeBSD Base System	4129
36.5.3	2.3 Optional Hardware Tools	4131
36.5.4	2.4 The Observation Rig	4132
36.5.5	2.5 Bringing Up the Vendor Driver Under Another OS	4133
36.5.6	2.6 The Lab Notebook	4134
36.5.7	2.7 A Sample Lab Layout	4134
36.5.8	2.8 Wrapping Up Section 2	4135
36.6	3. Observing Device Behavior in a Controlled Environment	4135
36.6.1	3.1 Static Descriptors and Identity Information	4135
36.6.2	3.2 The First Capture: Initialization	4137
36.6.3	3.3 Functional Captures	4138
36.6.4	3.4 The Diff: Isolating Bits of Meaning	4139
36.6.5	3.5 Wireshark and the USB Dissector	4140
36.6.6	3.6 Observation Patterns to Recognise	4140
36.6.7	3.7 Wrapping Up Section 3	4141
36.7	4. Creating a Hardware Register Map by Experimentation	4141
36.7.1	4.1 Mapping the Address Space	4142
36.7.2	4.2 The Read-First Principle	4143
36.7.3	4.3 The Probing Module's Skeleton	4143
36.7.4	4.4 Inferring Register Purpose From Behavior	4145
36.7.5	4.5 Comparing Captured Writes Against Hypotheses	4146
36.7.6	4.6 The Discipline of One Hypothesis at a Time	4146
36.7.7	4.7 Common Register-Layout Patterns	4147
36.7.8	4.8 What Not to Do Yet	4147
36.7.9	4.9 Wrapping Up Section 4	4148
36.8	5. Identifying Data Buffers and Command Interfaces	4148
36.8.1	5.1 The Big Three: Linear, Ring, and Descriptor	4148
36.8.2	5.2 Telling Buffers From Each Other	4149
36.8.3	5.3 Command Queues and Their Sequencing	4149
36.8.4	5.4 DMA and Bus Mastering	4150
36.8.5	5.5 Reversing Packet Formats	4150
36.8.6	5.6 Identifying Which Operations Are Read-Like and Which Are	4151
36.8.7	5.7 Wrapping Up Section 5	4151
36.9	6. Re-Implementing Minimal Device Functionality	4152
36.9.1	6.1 Start With Reset, Always	4152
36.9.2	6.2 The Skeleton of a Minimal Driver	4152
36.9.3	6.3 Implementing Reset	4154
36.9.4	6.4 Verifying the Identifier	4155
36.9.5	6.5 Adding Logging	4156
36.9.6	6.6 Comparing Logged Output Against Captures	4157
36.9.7	6.7 Writing the Module Boilerplate	4157
36.9.8	6.8 The First Successful Load	4158

36.9.9	6.9 Unloading Cleanly	4159
36.9.10	6.10 Wrapping Up Section 6	4159
36.107.	Iteratively Expanding the Driver and Confirming Behavior	4160
36.10.1	7.1 The One-Feature-at-a-Time Rule	4160
36.10.2	7.2 The Verification Loop	4160
36.10.3	7.3 The Compare-Against-Vendor Pattern	4161
36.10.4	7.4 Documenting Each Discovered Register	4161
36.10.5	7.5 When the Vendor's Driver Disagrees With Itself	4162
36.10.6	7.6 Branching the Driver as Hypotheses Diverge	4162
36.10.7	7.7 Wrapping Up Section 7	4162
36.118.	Validating Assumptions With Simulation Tools	4163
36.11.1	8.1 The Kernel-Space Mock Device	4163
36.11.2	8.2 Trapping Driver Accesses	4164
36.11.3	8.3 USB Template Drivers	4164
36.11.4	8.4 bhyve and Passthrough	4165
36.11.5	8.5 Loop With Known Return States	4166
36.11.6	8.6 Limits of Simulation	4166
36.11.7	8.7 Wrapping Up Section 8	4167
36.129.	Community Collaboration in Reverse Engineering	4167
36.12.1	9.1 Finding Prior Work	4167
36.12.2	9.2 Evaluating Trustworthiness	4168
36.12.3	9.3 The Clean-Room Discipline	4169
36.12.4	9.4 Sharing Your Findings	4169
36.12.5	9.5 Maintaining a Pseudo-Datasheet in Markdown or Git	4170
36.12.6	9.6 Wrapping Up Section 9	4170
36.1310.	Avoiding Bricking or Hardware Damage	4170
36.13.1	10.1 The General Principle	4171
36.13.2	10.2 Operations That Deserve Particular Caution	4171
36.13.3	10.3 Soft Resets and Watchdog Timers	4172
36.13.4	10.4 Backing Up Firmware Before Doing Anything Risky	4174
36.13.5	10.5 Read-Only Probes Where Possible	4174
36.13.6	10.6 Safe Probing Wrappers	4175
36.13.7	10.7 Recognising When to Stop	4175
36.13.8	10.8 Wrapping Up Section 10	4175
36.1411.	Refactoring and Documenting the Reversed Device	4176
36.14.1	11.1 The Driver Cleanup	4176
36.14.2	11.2 Driver Documentation	4176
36.14.3	11.3 The Pseudo-Datasheet Structure	4177
36.14.4	11.4 The Provenance Section in Detail	4179
36.14.5	11.5 Registers as Tables	4179
36.14.6	11.6 Pseudo-Datasheet as a Living Document	4180
36.14.7	11.7 Versioning the Driver	4180
36.14.8	11.8 Wrapping Up Section 11	4180
36.1512.	Legal and Ethical Guardrails in Practice	4181
36.15.1	12.1 Why a Second Legal Section	4181
36.15.2	12.2 Licence Compatibility	4181
36.15.3	12.3 Contractual Restrictions	4182
36.15.4	12.4 Clean-Room Practice	4183
36.15.5	12.5 Safe Harbour	4183
36.15.6	12.6 Wrapping Up Section 12	4184

36.16	13. Case Studies From the FreeBSD Tree	4184
36.16.1	13.1 How to Read These Case Studies	4185
36.16.2	13.2 Case Study 1: The umcs Driver and Its Undocumented GPIO	4185
36.16.3	13.3 Case Study 2: The axe Driver and the Missing IPG Initialisation	4186
36.16.4	13.4 Shared Lessons	4188
36.16.5	13.5 Wrapping Up Section 13	4188
36.17	Hands-On Labs	4189
36.17.1	Lab 1: Identifying a Device and Dumping Descriptors	4189
36.17.2	Lab 2: Capturing a USB Initialization Sequence	4190
36.17.3	Lab 3: Building a Safe Register Probing Module	4190
36.17.4	Lab 4: Writing and Driving a Mock Device	4191
36.17.5	Lab 5: Building a Safe Probing Wrapper	4191
36.17.6	Lab 6: Writing a Pseudo-Datasheet	4191
36.18	Challenge Exercises	4192
36.18.1	Challenge 1: Triage an Unknown Capture	4192
36.18.2	Challenge 2: Extend the Mock Device	4192
36.18.3	Challenge 3: Detect Device Identity From Behaviour Alone	4193
36.18.4	Challenge 4: Document a Device You Own	4193
36.19	Hands-On Exercise: Your Own Observation	4193
36.19.1	Before You Start: The Ethical Gate	4194
36.19.2	Step 1: Initial Identification With <code>usbconfig(8)</code>	4194
36.19.3	Step 2: Packet-Level Observation With <code>usbdump(8)</code>	4195
36.19.4	Step 3: Map Descriptors to Interface and Endpoint Structures	4196
36.19.5	Step 4: Sketch a Newbus-Style Driver Skeleton	4196
36.19.6	Step 5: Write Down What You Learned	4198
36.19.7	A Closing Reminder on Scope	4199
36.20	Common Mistakes and Troubleshooting	4199
36.20.1	Mistake 1: Believing the First Hypothesis	4199
36.20.2	Mistake 2: Skipping the Notebook	4199
36.20.3	Mistake 3: Writing to Unknown Registers	4200
36.20.4	Mistake 4: Treating the Vendor Driver as Complete	4200
36.20.5	Mistake 5: Not Testing the Detach Path	4200
36.20.6	Mistake 6: Not Saving Captures	4200
36.20.7	Mistake 7: Working Alone	4200
36.20.8	Troubleshooting: When the Driver Silently Does Nothing	4201
36.20.9	Troubleshooting: When the Driver Panics on Unload	4201
36.20.10	Troubleshooting: When Behaviour Differs Between Runs	4202
36.21	Wrapping Up	4202
36.22	Bridge to Chapter 37: Submitting Your Driver to the FreeBSD Project	4203
37	Submitting Your Driver to the FreeBSD Project	4205
37.1	Introduction	4205
37.2	Reader Guidance: How to Use This Chapter	4206
37.3	How to Get the Most Out of This Chapter	4208
37.4	Section 1: Understanding the FreeBSD Contribution Process	4209
37.4.1	What the FreeBSD Project Actually Is	4209
37.4.2	The Project's Decision-Making Structure	4209
37.4.3	Contributor Versus Committer	4210
37.4.4	How src Work Is Organised	4210
37.4.5	Where Device Drivers Live in the Source Tree	4211
37.4.6	The Second Half of the Driver: Kernel Build Integration	4211
37.4.7	The Review Platforms	4212

37.4.8	Exercise: Browse the Source Tree and Identify Similar Drivers	4212
37.4.9	Wrapping Up Section 1	4213
37.5	Section 2: Getting Your Driver Ready for Submission	4213
37.5.1	The Gap Between a Working Driver and a Submission-Ready Driver	4213
37.5.2	Where the Files Go	4213
37.5.3	What Goes in mydev.c	4214
37.5.4	A Minimal Driver File	4214
37.5.5	The Module Makefile	4217
37.5.6	The Header File	4217
37.5.7	Following style(9): The Short Summary	4218
37.5.8	Using tools/build/checkstyle9.pl	4220
37.5.9	Using indent(1) Carefully	4221
37.5.10	Commit Messages	4221
37.5.11	Signed-off-by and the Developer Certificate of Origin	4222
37.5.12	What a Complete Submission-Ready Tree Looks Like	4222
37.5.13	Common Mistakes in Section-2 Preparation	4223
37.5.14	Wrapping Up Section 2	4224
37.6	Section 3: Licensing and Legal Considerations	4224
37.6.1	Why Licensing Matters Up Front	4224
37.6.2	What Licences FreeBSD Accepts	4225
37.6.3	What Licences FreeBSD Does Not Accept	4225
37.6.4	The Copyright Header in Detail	4226
37.6.5	Reading Existing Headers to Build Intuition	4227
37.6.6	Derivative Works and External Code	4227
37.6.7	Adapted Code from Vendor Sources	4228
37.6.8	Code You Did Not Write but Are Submitting	4228
37.6.9	Adding Licence Headers to an Existing Driver	4229
37.6.10	Validating the Header	4229
37.6.11	Wrapping Up Section 3	4229
37.7	Section 4: Writing a Manual Page for Your Driver	4230
37.7.1	Why the Manual Page Matters	4230
37.7.2	Manual Page Sections: A Quick Orientation	4230
37.7.3	The Structure of a Section-4 Manual Page	4231
37.7.4	A Minimum Working Manual Page	4231
37.7.5	The Header Block	4234
37.7.6	The NAME Section	4234
37.7.7	The SYNOPSIS Section	4234
37.7.8	The DESCRIPTION Section	4234
37.7.9	The HARDWARE Section	4235
37.7.10	The FILES Section	4235
37.7.11	The SEE ALSO Section	4235
37.7.12	The HISTORY Section	4236
37.7.13	The AUTHORS Section	4236
37.7.14	Validating the Manual Page	4236
37.7.15	The One-Sentence-Per-Line Rule	4237
37.7.16	Common Manual-Page Mistakes	4238
37.7.17	Looking at Real Manual Pages	4238
37.7.18	Wrapping Up Section 4	4238
37.8	Section 5: Testing Your Driver Before Submission	4239
37.8.1	The Tests That Matter	4239
37.8.2	Code-Style Tests	4239

37.8.3	Manual-Page Tests	4240
37.8.4	Local Build Tests	4240
37.8.5	Runtime Tests	4241
37.8.6	Cross-Architecture Build Tests	4241
37.8.7	tinderbox: The Failure-Tracking Variant of universe	4243
37.8.8	Running Kernel Lint Tools	4243
37.8.9	Testing in a Virtual Machine	4243
37.8.10	Testing Against HEAD	4244
37.8.11	A Shell Script for the Whole Pipeline	4244
37.8.12	What Testing Does Not Catch	4245
37.8.13	Wrapping Up Section 5	4245
37.9	Section 6: Submitting a Patch for Review	4245
37.9.1	What a Patch Is, in the FreeBSD Sense	4245
37.9.2	Preparing the Commit	4246
37.9.3	Generating a Patch for Review	4246
37.9.4	Which Route to Take	4247
37.9.5	Writing a Good Review Description	4247
37.9.6	The Draft Email for the Mailing List	4248
37.9.7	What Happens After You Submit	4249
37.9.8	Response Time and Patience	4250
37.9.9	Iteration and Patch Updates	4250
37.9.10	Common Mistakes in Submission	4251
37.9.11	Wrapping Up Section 6	4251
37.10	Section 7: Working With a Mentor or Committer	4252
37.10.1	Why the Human Side Matters	4252
37.10.2	The Role of a Mentor	4252
37.10.3	Sponsorship: The Commit Pathway	4253
37.10.4	Receiving Feedback Gracefully	4253
37.10.5	Iteration and Patience	4254
37.10.6	Handling Disagreements	4254
37.10.7	Building a Long-Term Relationship	4255
37.10.8	Identifying Existing Maintainers	4255
37.10.9	Exercise: Identify a Similar Driver's Maintainer	4256
37.10.10	Wrapping Up Section 7	4256
37.11	Section 8: Maintaining and Supporting Your Driver Post-Merge	4256
37.11.1	The Merge Is Not the End	4256
37.11.2	Bugzilla Watching	4256
37.11.3	Responding to Community Feedback	4257
37.11.4	Kernel API Drift	4257
37.11.5	The UPDATING File	4258
37.11.6	Tree-Wide Refactors	4258
37.11.7	Participating in Future Releases	4259
37.11.8	Keeping Code Updated: A Rhythm	4259
37.11.9	Exercise: Create a Monthly Maintenance Checklist	4259
37.11.10	When You Cannot Maintain Anymore	4260
37.11.11	Wrapping Up Section 8	4260
37.12	Hands-On Labs	4260
37.12.1	Lab 1: Prepare the File Layout	4260
37.12.2	Lab 2: Audit the Code Style	4261
37.12.3	Lab 3: Add the Copyright Header	4262
37.12.4	Lab 4: Draft the Manual Page	4262

37.12.5 Lab 5: Build and Load Automation	4263
37.12.6 Lab 6: Generate a Submission Patch	4263
37.12.7 Lab 7: Draft a Review Cover Letter	4264
37.12.8 Lab 8: Dry-Run a Review Cycle	4264
37.13 Challenge Exercises	4265
37.13.1 Challenge 1: Audit a Historical Driver	4265
37.13.2 Challenge 2: Cross-Architecture Debug	4265
37.13.3 Challenge 3: Manual Page Depth	4266
37.13.4 Challenge 4: Run a Mock Review	4266
37.13.5 Challenge 5: Trace the Life of a Real Commit	4267
37.14 Troubleshooting and Common Mistakes	4267
37.14.1 Patch Rejected Because of Style	4267
37.14.2 Patch Rejected Because of Manual-Page Issues	4267
37.14.3 Patch Does Not Apply Cleanly	4268
37.14.4 Kernel Panic on Load	4268
37.14.5 Kernel Panic on Unload	4268
37.14.6 Driver Builds But Does Not Probe	4269
37.14.7 Manual Page Does Not Render	4269
37.14.8 Reviewer Is Unresponsive	4269
37.14.9 Patch Is Approved But Not Committed	4269
37.14.10 Patch Committed But You Were Not Credited	4270
37.14.11 Your Driver Was Accepted but Your Interface Choice Was Wrong	4270
37.15 Wrapping Up	4270
37.16 Bridge to Chapter 38: Final Thoughts and Next Steps	4271
38 Final Thoughts and Next Steps	4273
38.1 Introduction	4273
38.2 Reader Guidance: How to Use This Chapter	4275
38.3 How to Get the Most Out of This Chapter	4276
38.4 1. Celebrating What You Have Accomplished	4278
38.4.1 1.1 A Recap of What We Have Covered	4278
38.4.2 1.2 Skills You Have Developed	4279
38.4.3 1.3 Reflection: What Has Actually Changed	4280
38.4.4 1.4 Why Celebrating Progress Matters	4281
38.4.5 1.5 Exercise: Write a Personal Reflection	4282
38.4.6 1.6 Wrapping Up Section 1	4282
38.5 2. Understanding Where You Stand Now	4282
38.5.1 2.1 You Are Now Capable Of	4283
38.5.2 2.2 Confidence With Core Technologies	4283
38.5.3 2.3 Review Exercise: Map the APIs You Used	4284
38.5.4 2.4 The Difference Between Having Followed the Book and Being Independent	4285
38.5.5 2.5 Identifying Your Strongest and Weakest Areas	4286
38.5.6 2.6 Calibrating Against Real FreeBSD Developers	4286
38.5.7 2.7 The Shape of Competence You Have Built	4287
38.5.8 2.8 Wrapping Up Section 2	4287
38.6 3. Exploring Advanced Topics for Continued Learning	4287
38.6.1 3.1 Filesystems and Deeper VFS Work	4288
38.6.2 3.2 Network Stack Internals	4288
38.6.3 3.3 USB Composite Devices	4289
38.6.4 3.4 PCI Hotplug and Runtime Power Management	4289
38.6.5 3.5 SMP and NUMA-Aware Drivers	4290
38.6.6 3.6 Additional Areas Worth Naming	4290

38.6.7	3.7 Exercise: Pick One, Read One Manpage, Write One Paragraph	4291
38.6.8	3.8 Wrapping Up Section 3	4292
38.7	4. Building Your Own Development Toolkit	4292
38.7.1	4.1 Setting Up a Reusable Driver Template	4292
38.7.2	4.2 Building a Reusable Virtual Lab	4293
38.7.3	4.3 Loopback Testing With User-Mode Helpers	4294
38.7.4	4.4 Creating Regression Tests for Your Driver	4294
38.7.5	4.5 Lightweight Continuous Integration	4295
38.7.6	4.6 Package Your Driver With Documentation and Test Scripts	4295
38.7.7	4.7 Exercise: Package a Driver End-to-End	4296
38.7.8	4.8 Wrapping Up Section 4	4296
38.8	5. Giving Back to the FreeBSD Community	4296
38.8.1	5.1 Why Community Engagement Matters	4296
38.8.2	5.2 Participating in Mailing Lists	4297
38.8.3	5.3 Using Bugzilla	4298
38.8.4	5.4 Contributing to Documentation	4298
38.8.5	5.5 Translating Documentation	4299
38.8.6	5.6 Mentoring Other Beginners	4299
38.8.7	5.7 Contributing Driver Fixes	4300
38.8.8	5.8 Translating Knowledge to Other BSDs	4300
38.8.9	5.9 Exercise: Write a Thank-You Message	4301
38.8.10	5.10 The Relationship Between Contribution and Growth	4301
38.8.11	5.11 Wrapping Up Section 5	4302
38.9	6. Staying Up to Date With FreeBSD Kernel Development	4302
38.9.1	6.1 Where to Follow FreeBSD Development	4302
38.9.2	6.2 Developer Summits and BSD Conferences	4303
38.9.3	6.3 Tracking API and Driver Model Changes Across Releases	4304
38.9.4	6.4 Tools for Comparing Kernel Trees	4304
38.9.5	6.5 A Monthly Rhythm for Staying Current	4305
38.9.6	6.6 Exercise: Subscribe and Read	4305
38.9.7	6.7 Watching for Deprecation Notices	4305
38.9.8	6.8 Engaging When You See a Change That Affects You	4306
38.9.9	6.9 A Curated Reading List for Continued Study	4306
38.9.10	6.10 Wrapping Up Section 6	4308
38.10	7. Hands-On Reflection Labs	4308
38.10.1	7.1 Lab 1: Complete a Self-Assessment Worksheet	4308
38.10.2	7.2 Lab 2: Draft a Personal Learning Roadmap	4309
38.10.3	7.3 Lab 3: Set Up Your Driver Template	4310
38.10.4	7.4 Lab 4: Subscribe to a Mailing List	4310
38.10.5	7.5 Lab 5: Review a Manual Page You Have Not Read	4311
38.10.6	7.6 Lab 6: Build a Regression Test for a Driver You Wrote	4312
38.10.7	7.7 Lab 7: File or Triage a Bugzilla PR	4312
38.10.8	7.8 Lab 8: Set Up Your Virtual Lab	4313
38.10.9	7.9 Wrapping Up the Reflection Labs	4314
38.11	8. Challenge Exercises	4314
38.11.1	8.1 Challenge 1: Write a Real Driver for Real Hardware	4314
38.11.2	8.2 Challenge 2: Fix a Real Bug in a Real Driver	4315
38.11.3	8.3 Challenge 3: Port a Linux or NetBSD Driver to FreeBSD	4315
38.11.4	8.4 Challenge 4: Write a Deep Technical Blog Post	4316
38.11.5	8.5 Challenge 5: Become a Reviewer	4316
38.11.6	8.6 Challenge 6: Run a Study Group	4317

38.11.7	8.7 Challenge 7: Contribute a Non-Code Artifact	4317
38.11.8	8.8 Challenge 8: Sustain a Personal Practice for Six Months	4318
38.11.9	8.9 Wrapping Up the Challenges	4318
38.12.9	Personal Planning and Checklists	4318
38.12.1	9.1 A Contribution Checklist	4319
38.12.2	9.2 A Stay-Current Checklist	4319
38.12.3	9.3 A Self-Assessment Worksheet	4319
38.12.4	9.4 A Learning Roadmap Template	4320
38.12.5	9.5 A Code Review Checklist	4321
38.12.6	9.6 A Debugging Journal Entry Template	4322
38.12.7	9.7 A Driver Handoff Checklist	4323
38.12.8	9.8 A Quarterly Review Template	4324
38.12.9	9.9 Where to Keep These Artifacts	4325
38.12.10	10.10 Wrapping Up Section 9	4326
38.13	Wrapping Up	4326
38.14	Part 7 Checkpoint	4327
38.15	Final Closing Words for the Book	4328
39	Appendix A: FreeBSD Kernel API Reference	4331
39.1	How to Use This Appendix	4331
39.1.1	How the Entries Are Organised	4331
39.1.2	What This Appendix Is Not	4332
39.2	Reader Guidance	4332
39.3	Memory Allocation	4333
39.3.1	malloc(9) / free(9) / realloc(9)	4333
39.3.2	uma(9) Zones	4334
39.3.3	contigmalloc(9) / contigfree(9)	4334
39.3.4	Allocation Flag Cheat Sheet	4335
39.4	Synchronisation Primitives	4335
39.4.1	mtx(9): Mutexes	4336
39.4.2	sx(9): Sleepable Shared-Exclusive Locks	4336
39.4.3	rmlock(9): Read-Mostly Locks	4337
39.4.4	cv(9) / condvar(9): Condition Variables	4338
39.4.5	sema(9): Counting Semaphores	4338
39.4.6	atomic(9): Atomic Operations	4339
39.4.7	epoch(9): Read-Mostly Lock-Free Sections	4339
39.4.8	Lock Decision Cheat Sheet	4340
39.5	Deferred Execution and Timers	4340
39.5.1	callout(9): Kernel Timers	4340
39.5.2	taskqueue(9): Deferred Work in a Worker Thread	4341
39.5.3	kthread(9) and kproc(9): Kernel Threads and Processes	4342
39.5.4	Deferred Work Decision Cheat Sheet	4342
39.6	Bus and Resource Management	4343
39.6.1	Newbus: DRIVER_MODULE, DEVMETHOD, and Related Macros	4343
39.6.2	devclass(9) and Device Accessors	4343
39.6.3	rman(9): Resource Manager	4344
39.6.4	bus_space(9): Portable Register Access	4345
39.6.5	bus_dma(9): Portable DMA	4345
39.6.6	Interrupt Setup	4346
39.7	Device Nodes and Character-Device I/O	4347
39.7.1	make_dev_s(9) and destroy_dev(9)	4347
39.7.2	cdevsw: Character-Device Switch Table	4348

39.7.3	<code>ioctl(9)</code> Dispatch	4348
39.7.4	<code>devfs_set_cdevpriv(9)</code> : Per-Open State	4349
39.8	Process and User-Space Interaction	4349
39.8.1	<code>copyin(9)</code> , <code>copyout(9)</code> , <code>copyinstr(9)</code>	4349
39.8.2	<code>uio(9)</code> : I/O Descriptor for Read and Write	4350
39.8.3	<code>proc(9)</code> and Thread Context for Drivers	4350
39.9	Observability and Notification	4351
39.9.1	<code>sysctl(9)</code> : Read-and-Write Configuration Nodes	4351
39.9.2	<code>eventhandler(9)</code> : In-Kernel Publish-Subscribe	4351
39.9.3	<code>poll(2)</code> and <code>kqueue(2)</code> : Readiness Notification	4352
39.10	Diagnostics, Logging, and Tracing	4352
39.10.1	<code>log(9)</code> and <code>printf(9)</code>	4353
39.10.2	<code>KASSERT(9)</code> : Kernel Assertions	4353
39.10.3	<code>WITNESS</code> : Lock-Order Verifier	4354
39.10.4	<code>ktr(9)</code> : Kernel Trace Facility	4354
39.10.5	DTrace Static Probes and Major Providers	4354
39.11	Cross-Reference by Driver Lifecycle Phase	4355
39.11.1	Module Load	4355
39.11.2	Probe	4355
39.11.3	Attach	4355
39.11.4	Normal Operation	4356
39.11.5	Detach	4356
39.11.6	Module Unload	4356
39.12	Quick-Reference Checklists	4356
39.12.1	Locking Discipline Checklist	4356
39.12.2	Resource Lifetime Checklist	4357
39.12.3	User-Space Safety Checklist	4357
39.12.4	Diagnostic Coverage Checklist	4357
39.13	Wrapping Up	4357
40	Appendix B: Algorithms and Logic for Systems Programming	4359
40.1	How to Use This Appendix	4359
40.1.1	What You Will Find Here	4359
40.1.2	What This Appendix Is Not	4360
40.2	Reader Guidance	4360
40.3	Data Structures in the Kernel	4360
40.3.1	The <code><sys/queue.h></code> Family	4361
40.3.2	Red-Black Trees via <code><sys/tree.h></code>	4362
40.3.3	Radix Trees via <code><net/radix.h></code>	4363
40.3.4	Bitmaps and Bit-Strings via <code><sys/bitstring.h></code>	4363
40.3.5	Hash Tables with <code>hashinit(9)</code>	4364
40.4	Circular and Ring Buffers	4365
40.4.1	Single-Producer Single-Consumer Ring Buffers	4365
40.4.2	<code>buf_ring(9)</code> for MPMC Rings	4366
40.4.3	Wrap-Around Arithmetic and the Full-vs-Empty Problem	4367
40.5	Sorting and Searching	4367
40.5.1	In-Kernel <code>qsort</code> and <code>bsearch</code>	4367
40.5.2	Binary Search Over Device ID Tables	4368
40.5.3	When Linear Search Wins	4368
40.6	State Machines and Protocol Handling	4368
40.6.1	Explicit State Enum vs Implicit Flag Bits	4369
40.6.2	Switch-Based FSMs	4369

40.6.3	Table-Driven FSMs	4369
40.6.4	Function-Pointer Dispatch	4370
40.6.5	Reentrancy and Partial Completion	4370
40.7	Error-Handling Patterns	4371
40.7.1	The goto out Idiom	4371
40.7.2	Return-Code Conventions	4373
40.7.3	Resource Acquisition and Release Ordering	4373
40.8	Concurrency Patterns	4374
40.8.1	Producer-Consumer with Condition Variables	4374
40.8.2	Reader-Writer: rmlck(9) vs sx(9)	4375
40.8.3	Reference Counting with refcount(9)	4375
40.9	Decision Aids	4376
40.9.1	Collection Shape	4376
40.9.2	Ring Buffer Shape	4376
40.9.3	State Representation	4377
40.9.4	Cleanup Strategy	4377
40.9.5	Concurrency Pattern	4377
40.10	Wrapping Up: How to Recognise These Patterns in Real Code	4378
41	Appendix C: Hardware Concepts for Driver Developers	4379
41.1	How to Use This Appendix	4379
41.1.1	What You Will Find Here	4379
41.1.2	What This Appendix Is Not	4379
41.2	Reader Guidance	4380
41.3	Memory in the Kernel	4380
41.3.1	Physical vs Virtual Memory	4381
41.3.2	The MMU, Page Tables, and Device Mappings	4381
41.3.3	Why Hardware Cannot Simply See Kernel Pointers	4382
41.3.4	Kernel Memory Allocation in One Page	4383
41.3.5	IOMMUs and Bus Addresses	4383
41.3.6	Memory Barriers in One Paragraph	4384
41.4	Buses and Interfaces	4384
41.4.1	PCI and PCIe at a Driver-Relevant Level	4385
41.4.2	USB at a Driver-Relevant Level	4385
41.4.3	I2C at a Driver-Relevant Level	4386
41.4.4	SPI at a Driver-Relevant Level	4387
41.4.5	A Compact Comparison of the Four Buses	4387
41.5	Interrupts	4388
41.5.1	What an Interrupt Actually Is	4388
41.5.2	Edge-Triggered vs Level-Triggered	4389
41.5.3	MSI and MSI-X Without the Acronyms	4390
41.5.4	Why Interrupt Discipline Matters	4390
41.6	DMA	4391
41.6.1	What DMA Is and Why It Exists	4391
41.6.2	The Mental Model: Tags, Maps, Segments, Sync	4392
41.6.3	Why Synchronisation and Mapping Matter	4393
41.6.4	DMA Buffers vs Control Buffers	4393
41.7	Quick-Reference Tables	4394
41.7.1	Address Spaces at a Glance	4394
41.7.2	When to Use Which Allocator	4394
41.7.3	Interrupt vs Polling	4394
41.7.4	Bus at a Glance (Driver-Writer View)	4395

41.7.5 DMA Sync Quick Check	4395
41.8 Wrapping Up: How to Keep Relating Hardware Concepts to Driver Code	4395
42 Appendix D: Operating System Concepts	4397
42.1 How to Use This Appendix	4397
42.1.1 What You Will Find Here	4397
42.1.2 What This Appendix Is Not	4397
42.2 Reader Guidance	4398
42.3 Section 1: What Is a Kernel, and What Does It Do?	4398
42.3.1 Kernel Responsibilities in One Page	4398
42.3.2 Kernel Space vs User Space	4399
42.3.3 System Calls: The Door Between the Two Sides	4400
42.3.4 Why Kernel Mistakes Have System-Wide Consequences	4400
42.3.5 How Drivers Participate in Kernel Responsibilities	4401
42.4 Section 2: What Is a Device Driver?	4401
42.4.1 Driver Types: Character, Block, Network, Pseudo	4402
42.4.2 The Role of a Driver in Hardware Communication	4403
42.4.3 How Drivers Are Loaded and Bound to Devices	4403
42.4.4 Connecting Hardware, Kernel Subsystems, and Userland	4404
42.5 Section 3: Kernel Data Structures You Will See Often	4405
42.5.1 Lists and Queues from <code><sys/queue.h></code>	4405
42.5.2 Kernel Buffers: <code>mbuf</code> , <code>buf</code> , and <code>bio</code>	4406
42.5.3 Locking: Mutex, Spin Mutex, Sleep Lock, Condition Variable	4406
42.5.4 <code>softc</code> : The Per-Device Driver Structure	4407
42.5.5 Why These Structures Keep Reappearing	4408
42.6 Section 4: Kernel Compilation and Booting	4408
42.6.1 The Boot Process Overview	4408
42.6.2 From Power-On to <code>init</code> : A Compact Timeline	4409
42.6.3 Boot Loaders and Kernel Modules at Boot Time	4410
42.6.4 Kernel Modules and Driver Attachment	4410
42.6.5 <code>init_main</code> and the <code>SYSINIT</code> Chain	4411
42.6.6 Kernel Compilation in One Paragraph	4411
42.7 Quick-Reference Tables	4412
42.7.1 Kernel Space vs User Space at a Glance	4412
42.7.2 Driver Type Comparison	4412
42.7.3 Locking Primitive Selection	4412
42.7.4 <code>softc</code> Lifecycle Checkpoint	4413
42.7.5 Boot-to-Init Quick Map	4413
42.8 Wrapping Up: How These OS Concepts Support Driver Development	4413
43 Appendix E: Navigating FreeBSD Kernel Internals	4415
43.1 How to Use This Appendix	4415
43.1.1 What You Will Find Here	4415
43.1.2 What This Appendix Is Not	4415
43.2 Reader Guidance	4416
43.3 How This Appendix Differs From Appendix A	4417
43.4 A Map of the Major Subsystems	4417
43.5 Process and Thread Subsystem	4418
43.5.1 What the subsystem is for	4418
43.5.2 Why a driver author should care	4418
43.5.3 Key structures, interfaces, or concepts	4418
43.5.4 Typical driver touchpoints	4419

43.5.5	Where to look in /usr/src	4419
43.5.6	Manual pages and files to read next	4419
43.5.7	Common beginner confusion	4419
43.5.8	Where the book teaches this	4420
43.5.9	Further reading	4420
43.6	Memory Subsystem	4420
43.6.1	What the subsystem is for	4420
43.6.2	Why a driver author should care	4420
43.6.3	Key structures, interfaces, or concepts	4420
43.6.4	Typical driver touchpoints	4421
43.6.5	Where to look in /usr/src	4421
43.6.6	Manual pages and files to read next	4421
43.6.7	Common beginner confusion	4422
43.6.8	Where the book teaches this	4422
43.6.9	Further reading	4422
43.7	File and VFS Subsystem	4422
43.7.1	What the subsystem is for	4422
43.7.2	Why a driver author should care	4422
43.7.3	Key structures, interfaces, or concepts	4422
43.7.4	Typical driver touchpoints	4423
43.7.5	Where to look in /usr/src	4423
43.7.6	Manual pages and files to read next	4424
43.7.7	Common beginner confusion	4424
43.7.8	Where the book teaches this	4424
43.7.9	Further reading	4424
43.8	Network Subsystem	4424
43.8.1	What the subsystem is for	4424
43.8.2	Why a driver author should care	4424
43.8.3	Key structures, interfaces, or concepts	4425
43.8.4	Typical driver touchpoints	4425
43.8.5	Where to look in /usr/src	4425
43.8.6	Manual pages and files to read next	4426
43.8.7	Common beginner confusion	4426
43.8.8	Where the book teaches this	4426
43.8.9	Further reading	4426
43.9	Driver Infrastructure (Newbus)	4426
43.9.1	What the subsystem is for	4426
43.9.2	Why a driver author should care	4426
43.9.3	Key structures, interfaces, or concepts	4427
43.9.4	Typical driver touchpoints	4427
43.9.5	Where to look in /usr/src	4427
43.9.6	Manual pages and files to read next	4428
43.9.7	Common beginner confusion	4428
43.9.8	Where the book teaches this	4428
43.9.9	Further reading	4428
43.10	Boot and Module System	4428
43.10.1	What the subsystem is for	4428
43.10.2	Why a driver author should care	4428
43.10.3	Key structures, interfaces, or concepts	4429
43.10.4	Typical driver touchpoints	4429
43.10.5	Where to look in /usr/src	4429

43.10.6 Manual pages and files to read next	4430
43.10.7 Common beginner confusion	4430
43.10.8 Where the book teaches this	4430
43.10.9 Further reading	4430
43.11 Kernel Services	4430
43.11.1 What the subsystem is for	4430
43.11.2 Why a driver author should care	4430
43.11.3 Key structures, interfaces, or concepts	4431
43.11.4 Typical driver touchpoints	4431
43.11.5 Where to look in /usr/src	4431
43.11.6 Manual pages and files to read next	4432
43.11.7 Common beginner confusion	4432
43.11.8 Where the book teaches this	4432
43.11.9 Further reading	4432
43.12 Cross-References: Structures and Their Subsystems	4432
43.13 Source-Tree Navigation Checklists	4433
43.13.1 When you have a structure name	4434
43.13.2 When you have a function name	4434
43.13.3 When you have a macro name	4434
43.13.4 When you have a subsystem question	4434
43.14 Manual Pages and Source Reading Itinerary	4435
43.15 Wrapping Up: How to Keep Exploring the Kernel Safely	4435
44 Appendix F: Benchmark Harness and Results	4437
44.1 How to Use This Appendix	4437
44.1.1 How this appendix is organised	4437
44.2 Hardware and Software Setup	4438
44.2.1 Caveats that apply to every section	4438
44.3 Timecounter Read Cost	4439
44.3.1 What is being measured	4439
44.3.2 The harness	4439
44.3.3 How to reproduce	4439
44.3.4 Representative result	4440
44.3.5 Hardware envelope	4440
44.4 Synchronization Primitive Latencies	4440
44.4.1 What is being measured	4440
44.4.2 The harness	4441
44.4.3 How to reproduce	4441
44.4.4 Representative result	4441
44.4.5 Hardware envelope	4442
44.5 iflib Driver Code-Size Reduction	4442
44.5.1 What is being measured	4442
44.5.2 The harness	4442
44.5.3 How to reproduce	4443
44.5.4 Representative result	4443
44.5.5 Pinned per-driver measurement	4444
44.5.6 Hardware envelope	4444
44.6 DTrace, INVARIANTS, and WITNESS Overhead	4444
44.6.1 What is being measured	4444
44.6.2 The harness	4445
44.6.3 How to reproduce	4445
44.6.4 Representative result	4446

44.6.5 Hardware envelope 4446

44.7 Scheduler Wakeup Latency 4446

44.8 Wrapping Up: Using the Harness Without Being Misled 4447

“What begins as curiosity becomes skill, and what becomes skill empowers the next generation.” - Edson Brandi

1 Introduction - From Curiosity to Contribution

1.1 My Journey: From Curiosity to Career

Every book begins with a reason to exist. For this one, the reason is deeply personal. My own path into technology was anything but conventional. I did not start as a computer science student or grow up surrounded by computers. My early academic journey was in chemistry, not computing. I spent my days with experiments, formulas, and laboratory equipment, tools that, at first glance, seemed far removed from operating systems and device drivers.

From Chemistry to Technology: A True Story About Curiosity and Change

In 1995, I was a chemistry student at Unicamp (State University of Campinas), one of Brazil's top universities. I had no plan to work with infrastructure, software, or systems. But I had questions. I wanted to know how computers really worked, not just how to use them, but how the pieces fit together under the hood.

That search for answers led me to FreeBSD. What fascinated me most was not only the way the system behaved but that I had access to its source code. This was not just an operating system to use; it was one I could study, line by line. That possibility became the fuel for my growing passion.

At first, I could only explore FreeBSD on the university's machines. During the day, my time was consumed by chemistry studies, but I longed for more hours to study it more closely. I couldn't let this curiosity interfere with my degree. Coming from a humble background, I was the first in my family to reach university, and I carried that responsibility with pride.

In those early years, I had no computer of my own. Living on a limited budget, I barely managed the basics of being away from home. For nearly two years, my studies of FreeBSD took place in the chemistry institute's library, where I devoured every document I could find about Unix and system programming. But without my own machine, I couldn't put theory into practice for many of the things I was learning.

Finally, in 1997, I managed to assemble my first computer: a 386DX running at 40 MHz, with 8 MB of RAM and a 240 MB hard drive. The parts came from a generous friend I had met in the FreeBSD community, Adriano Martins Pereira. Over the years, he became such a close friend that he would later stand beside me as one of my groomsmen at my wedding.

I will never forget the moment I brought home the more than 20 floppy disks to install **FreeBSD 2.1.7**. It might seem trivial now, but for me it was transformative. For the first time, I could experiment with FreeBSD at night and on weekends, testing everything I had been reading about for two years. Looking back, I doubt I would be where I am today without the determination to nurture that spark of curiosity.

When I first touched FreeBSD, I understood very little of what I was seeing. The installation messages felt cryptic, the commands unfamiliar. Yet each small success, booting into a prompt, configuring a network card, compiling a kernel, felt like unlocking a new part of a hidden world. That thrill of discovery never fully goes away.

Over time, I wanted to share what I was learning. I created **FreeBSD Primeiros Passos** (FreeBSD First Steps), a small website in Portuguese that introduced newcomers to the system. It was simple, but it helped many people. That effort grew into the **Brazilian FreeBSD User Group (FUG-BR)**, which brought enthusiasts together to learn, exchange knowledge, and even meet in person at conferences. Speaking at early editions of **FISL**, Brazil's International Free Software Forum, was one of my proudest moments. Standing in front of an audience to share something that had changed my life was unforgettable.

Around that time, I also co-created the **FreeBSD LiveCD Tool Set**. Back in the late 1990s, installing an operating system like FreeBSD could feel daunting. Our tool allowed users to boot a complete FreeBSD environment directly from a CD without touching their hard drive. It was a simple idea, but it lowered the barrier for countless new users.

By 2002, I became one of the founders of **FreeBSD Brasil** (<https://www.freebsdbrasil.com.br>), a company dedicated to training, consulting, and supporting businesses with FreeBSD. It was an opportunity to bridge the gap between open source ideals and professional, real-world applications. Although I'm no longer involved, FreeBSD Brasil continues to this day, helping companies across Brazil adopt FreeBSD in their operations.

Eventually, my work led me deeper into the FreeBSD Project itself. I became a **committer**, focusing on documentation and translations. Today, I am part of the **FreeBSD Documentation Engineering Team (DocEng)**, helping maintain the systems that keep FreeBSD's documentation alive, updated, and accessible worldwide.

All of this, every project, every friendship, every opportunity, grew from that first decision to explore FreeBSD out of sheer curiosity.

Even though my formal studies were in chemistry, my career shifted toward technology. I worked in infrastructure engineering, managing on-premise datacenters, then building systems in the cloud, and eventually leading teams in software and product development across multiple industries in Brazil. Today, I serve as **IT Director at Teya**, a fintech company in London, helping design the systems that power businesses across Europe.

And it all began with one question: **How does this work?**

I share this story because I want you to see what is possible. You don't need a degree in computer science to become a great developer, sysadmin, or technology leader. What you need is curiosity, persistence, and the patience to keep learning when things get hard.

FreeBSD helped me unlock my potential. This book is here to help you unlock yours.

1.2 FreeBSD in Context

Before we start on writing drivers, let's take a moment to appreciate the stage on which all this happens.

1.2.1 A Brief History

FreeBSD traces its roots back to the **Berkeley Software Distribution (BSD)**, a derivative of the original UNIX developed at AT&T in the early 1970s. At the University of California, Berkeley, students and researchers contributed foundational work that shaped later UNIX-like systems, including the first widely adopted open **TCP/IP networking stack**, the Fast File System, and sockets, all of which remain recognizable in FreeBSD and many other systems today.

While many commercial UNIX variants rose and fell over the decades, FreeBSD endured. The project was officially founded in 1993, built on the work of the Berkeley CSRG (Computer Systems Research Group), and guided by a commitment to openness, technical excellence, and long-term stability. Over thirty years later, it is still actively developed and maintained, a rare achievement in the fast-moving world of technology.

FreeBSD has powered **research labs, universities, internet service providers, and enterprise products**. It has also been quietly present behind some of the busiest websites and networks in the world, chosen for its reliability and performance when failure is not an option.

More than a piece of software, FreeBSD represents a **continuum of knowledge**. By studying it, you are not only learning a modern operating system but also connecting with decades of accumulated engineering wisdom that continues to influence computing today.

1.2.2 Why FreeBSD Is Special

FreeBSD is developed as a complete UNIX-like operating system, with the kernel and the base userland built, versioned, and released together. Several characteristics follow directly from that approach:

- **Stability:** FreeBSD systems are famous for their ability to run for months or even years without rebooting. Internet service providers, data centers, and research institutions rely on this predictability for mission-critical workloads.
- **Performance:** The operating system consistently demonstrates excellent performance under demanding conditions. From high-throughput networking to complex storage systems, FreeBSD has been chosen in environments where efficiency is not optional but essential.
- **Clarity of design:** Unlike many other UNIX-like systems, FreeBSD is developed as a single, cohesive whole rather than a collection of components from different sources. Its codebase has a reputation for being well-structured and approachable, making it not just a platform to run but also a platform to learn from. For someone interested in systems programming, the source code is as valuable as the binaries.
- **Culture of documentation:** The project has always placed a high value on documentation. The FreeBSD Handbook and the many manual pages are written and maintained with the same care as the code itself. This reflects a core principle of the community: knowledge should be as accessible as the software.

Beyond these qualities, FreeBSD also stands out for its **license and philosophy**. The BSD license is permissive, giving both individuals and companies the freedom to use, adapt, and even commercialize their work without the obligation to open-source every change. This balance has encouraged widespread adoption in industry while keeping the project open and community-driven.

FreeBSD also has a **strong sense of stewardship**. Developers in the project are not just writing code for today's needs; they are maintaining a system that has been evolving for decades, with careful attention to long-term stability and clean design. This mindset makes it an excellent environment for learning, because decisions are not made hastily but with a view of how they will shape the system years into the future.

For beginners, this means FreeBSD is a system you can both use and study. Exploring its tools, its source code, and its documentation provides concrete lessons in how operating systems are built and how a long-running open source community sustains itself.

1.2.3 Kernel Insight

Did you know? Apple’s macOS and iOS draw heavily from BSD code, including parts of FreeBSD. When you browse the web on an iPhone or MacBook, you’re relying on decades of BSD engineering that has been tested and trusted in countless environments.

This lineage highlights an important truth: BSD is not a relic of the past. It is living technology, still shaping the systems people use every day. FreeBSD, in particular, has remained fully open and community-driven, offering the same foundation of reliability that large companies use, but without hiding it behind closed doors. When you study FreeBSD, you are looking at the same DNA that runs through some of the world’s most advanced operating systems.

1.2.4 Clearing Misconceptions

One common misunderstanding is to assume that FreeBSD is “*just another Linux*”. It is not. While both share UNIX roots, FreeBSD takes a very different approach.

Linux is a kernel combined with userland tools assembled from many independent projects. FreeBSD, on the other hand, is developed as a **complete operating system**, where the kernel, libraries, compiler toolchain, and core utilities are maintained together under a single project. This unified design makes FreeBSD feel consistent and cohesive, with fewer surprises when moving between components.

Another misconception is that FreeBSD is “*only for servers*”. While it is indeed trusted in server environments, FreeBSD also runs on desktops, laptops, and embedded systems. Its flexibility is part of what has allowed it to survive and evolve for decades while other UNIX variants have disappeared.

1.2.5 Real-World Uses

FreeBSD is everywhere, though often invisibly. When you stream a show from Netflix, there is a good chance that FreeBSD is delivering the content to you through Netflix’s global content delivery network. Networking companies like Juniper and storage providers like NetApp build products on top of FreeBSD, trusting its stability for their customers.

Even closer to home, FreeBSD powers firewalls, NAS devices, and appliances you might have in your office or living room. Projects like pfSense and FreeNAS (now TrueNAS) are based on FreeBSD, bringing enterprise-grade networking and storage into homes and small businesses worldwide.

And of course, FreeBSD has a long tradition in **research and education**. Universities use it in computer science curricula and labs, where having open access to the complete source code of a production-grade operating system is invaluable. Whether you realize it or not, you’ve probably already depended on FreeBSD in your daily life.

1.2.6 Why FreeBSD for Driver Development?

For someone learning about device drivers, FreeBSD offers a rare balance. Its codebase is modern and production-ready, but it is also clean and approachable compared to many alternatives. Developers often describe FreeBSD’s kernel as “readable,” a trait that matters greatly when you are just starting out.

The project also has a strong tradition of documentation. The FreeBSD Handbook, man pages, and developer guides provide a level of guidance that is hard to find in other open source kernels. This means you won't be left guessing how things fit together.

For professionals, FreeBSD is more than a teaching tool. It is widely respected in areas where performance, networking, and storage are critical. Learning how to write drivers here prepares you for work that extends far beyond FreeBSD itself; it builds skills in systems programming, debugging, and hardware interaction that are transferable to many platforms.

Perhaps most importantly, FreeBSD encourages learning by participation. Its open, collaborative culture welcomes contributions from beginners and experts alike. Starting here, you're not just learning in isolation; you are stepping into a community that values clarity, quality, and curiosity.

1.3 Drivers and the Kernel - A First Glimpse

Now that you know why FreeBSD matters, let's peek at the world you are about to explore.

At the heart of every operating system lies the **kernel**, the core that never sleeps. It orchestrates memory, manages processes, and directs communication with hardware. Most users never notice it, yet every keystroke, network packet, and disk read depends on its decisions.

1.3.1 Why Drivers Matter

A kernel without drivers would be like a conductor without musicians. Drivers are the interpreters that let hardware and software communicate with each other. Your keyboard, network card, and graphics adapter are just pieces of silicon until a driver tells the kernel how to address their registers, handle their interrupts, and move data to and from them. Without drivers, even sophisticated hardware is inert circuitry as far as the operating system is concerned.

1.3.2 Everyday Technology, Powered by Drivers

Drivers are everywhere, working quietly in the background. When you plug in a USB stick and see it appear on your desktop, that's a driver at work. When you connect to Wi-Fi, adjust your screen brightness, or listen to sound through your headphones, all of these actions depend on drivers. They are invisible to most users, but they are what make computers feel alive and responsive.

Think about it: every modern convenience of computing, from cloud servers processing millions of requests per second to the phone in your pocket, depends on the unseen labour of device drivers.

1.3.3 Beginner's Tip

If terms like *kernel*, *driver*, or *module* feel abstract right now, don't worry. This chapter is just the trailer, a preview of the whole story. In the coming chapters, we'll break these ideas down step by step until the pieces start to click together.

1.3.4 A Teaser of FreeBSD's Kernel World

One of FreeBSD's strengths is its **modular design**. Drivers can be loaded and unloaded dynamically as kernel modules, giving you the freedom to experiment without rebuilding the entire system. This flexibility is a gift for learners: you can try out code, test it, and remove it when you are done.

In this book, you will begin with the simplest form of drivers, character drivers, and progressively move toward more complex subsystems like PCI devices, USB peripherals, and even high-performance features such as DMA.

For now, hold on to this simple truth: **drivers are the bridge between possibility and reality in computing**. They turn abstract code into working hardware, and by learning how to write them, you are learning how to connect ideas with the physical world.

1.4 Your Path Through This Book

This book is not just a reference; it is a guided course, designed to take you step by step from the very basics to advanced concepts in FreeBSD driver development. You won't just read, you'll practice, experiment, and build real code along the way.

1.4.1 Who This Book Is For

This book was written with inclusivity in mind, especially for readers who may feel that systems programming is beyond their reach. It is for:

- **Beginners** who may know little about C, UNIX, or kernels, but are willing to learn.
- **Developers** who are curious about how operating systems work under the hood.
- **Professionals** who already use FreeBSD (or similar systems) and want to deepen their knowledge by learning how drivers are actually built.

If you bring curiosity and persistence, you will find a path here that starts where you are and builds your confidence chapter by chapter.

1.4.2 Who This Book Isn't For

Not every book fits every reader, and that is intentional. This book may not be the right fit if:

- You are looking for a **quick copy-and-paste manual**. This book emphasizes understanding and practice, not shortcuts.
- You are already a **seasoned kernel developer**. The pace starts from the ground up, introducing fundamentals carefully before moving on to complex topics.
- You expect a **comprehensive hardware reference manual**. This is not an encyclopedic listing of every device or bus specification. Instead, it focuses on practical, real-world driver development in FreeBSD.

1.4.3 What You'll Learn and Gain

This book gives you a structured, hands-on path into FreeBSD driver development. Along the way, you will:

- Begin with the foundations: installing FreeBSD, learning UNIX tools, and writing C programs.
- Progress into building and loading your own drivers.
- Explore concurrency, synchronization, and direct interaction with hardware.
- Learn to debug, test, and integrate your drivers into the FreeBSD ecosystem.

By the end, you will not only know how FreeBSD drivers are written, but also have the confidence to keep exploring and perhaps even contribute your own work back to the community.

1.4.4 The Learning Path Ahead

This book is organized as a sequence of parts that build on each other. You will start with the essentials, learning the FreeBSD environment, basic UNIX tools, and C programming fundamentals before gradually stepping into kernel space and driver development. From there, the path expands into more advanced areas of practice:

- **Part 1:** Foundations: FreeBSD, C, and the Kernel
- **Part 2:** Building Your First Driver
- **Part 3:** Concurrency and Synchronization
- **Part 4:** Hardware and Platform-Level Integration
- **Part 5:** Debugging, Tools, and Real-World Practices
- **Part 6:** Writing Transport-Specific Drivers
- **Part 7:** Mastery Topics: Special Scenarios and Edge Cases
- **Appendices:** Quick references, extra exercises, and resources

Along this path, you will reach important milestones. You will write and load your own kernel modules, build character drivers, and explore how FreeBSD handles PCI, USB, and networking. You will also learn how to debug and profile your code, and how to submit your work upstream to the FreeBSD Project.

When I first started writing drivers, I felt intimidated. “*Kernel programming*” sounded like something reserved for experts in dark rooms full of servers. The truth is more straightforward. It is still programming, only with more explicit rules, greater responsibility, and a bit more power. Once you understand that, the fear gives way to excitement. That is the spirit in which this learning path was designed: approachable, progressive, and rewarding.

1.5 How to Read This Book

Before the learning begins in earnest, a short word on pace, expectations, and what to do when something does not land on the first pass.

1.5.1 Time and Commitment

The book is long because the material is cumulative. A careful reader who works through the exercises should expect to spend around **two hundred hours** with it from start to finish: roughly **one hundred hours of reading**

and an additional **one hundred hours on the labs and challenge exercises**. A reader who only reads can finish in about one hundred hours but will leave with correct mental models and little muscle memory.

Part 1 alone accounts for about forty-five hours of reader time. That is not an accident. Part 1 lays the foundations on which the rest of the book rests, and the later Parts move faster because they lean on that foundation. Later Parts are typically shorter, depending on whether the subsystem they cover is new to you.

1.5.2 Labs Are Strongly Recommended

Every chapter includes hands-on labs, and most chapters include challenge exercises that push beyond the labs. **The labs are strongly recommended.** They are where prose turns into reflex. Kernel programming rewards muscle memory in a way few disciplines do: the same attach pattern, the same cleanup chain, the same locking shape appears in chapter after chapter and in driver after driver. Typing those patterns, compiling them, loading them into a running kernel, and watching them fail on purpose is the single most effective way to internalise them.

Learning is self-paced and ultimately under your control. If a chapter is informational and the material is already familiar, skimming is a fair choice. But when a chapter asks you to load a module and observe its behaviour, resist the urge to read around the exercise. The difference between a reader who ran the labs and one who did not tends to show up several chapters later, as either smooth progress or quiet confusion.

1.5.3 A Suggested Cadence

For a reader with about **five hours per week** available for study, **one chapter per week** is a sustainable pace. That cadence puts the whole book within reach across a typical academic or professional year. More hours per week allow a brisker pace; fewer hours call for patience rather than skipping. What matters most is continuity: short, regular sessions beat occasional marathons.

Some chapters are naturally longer than others. Chapter 4 on C is the longest in Part 1 and may span two or three weeks at five hours a week. Chapters in Parts 3, 4, and 5 are substantial because the technical material is layered. Allow extra time for Part 6 and Part 7 if the subsystems they cover are new to you.

1.5.4 Fast-Path Hints for Experienced Readers

If you already know C, UNIX, and the general shape of an operating-system kernel, not every section will be new territory. **Chapter 4 includes an “If You Already Know C” sidebar near its opening** that names the sections still worth a careful read and the ones you can skim. A similar principle applies to Chapters 2 and 3: if FreeBSD is familiar and the UNIX toolchain is second nature, read the opening pages to absorb this book’s vocabulary, then move on. Parts 3 through 7 introduce disciplines that reward careful reading regardless of prior experience.

1.5.5 What to Do When You Are Stuck

You will get stuck. Every reader does, and the places where you get stuck are often where the real learning happens. Three tactics work well.

First, **re-read the section slowly**. Kernel writing is dense, and a second pass often reveals a sentence you skimmed the first time. A slow re-read is almost always more productive than rushing onward.

Second, **run the labs that led up to the confusion**. A concept that is blurry in prose often becomes clear when you type the code, compile it, load it, and watch the kernel's response in `dmesg`. If a lab is already behind you, repeat it and vary one piece at a time. Observation on a running module teaches things no paragraph can.

Third, **file an issue**. The book's repository welcomes questions and corrections. If a passage seems wrong or a lab fails in an unexpected way, an issue helps more than you might expect; every such report is a chance to make the next reader's path smoother.

Above all, be patient with yourself. The material compounds. A concept that feels opaque in Chapter 5 may feel obvious by Chapter 11, not because you re-read Chapter 5 but because the vocabulary has had time to settle. Trust that, and keep going.

One final note on commitment. Taken together with the labs, the book is on the order of two hundred hours of work: call it a six-month evening project at roughly five hours a week, or a focused two-month sprint at twice that pace. A reader who skips the labs can finish in about half that time, with correct ideas but less muscle memory to show for it. You do not need to decide any of that today. Pick a pace that feels honest, begin with the next chapter, and let the rhythm settle as you go.

1.6 How to Get the Most from This Book

Learning kernel programming is not just about reading; it's about patience, practice, and persistence. The following principles will help you make the most of this book.

1.6.1 Build the Right Mindset

At first, kernels and drivers may feel overwhelming. That's normal. The secret is to take things one step at a time. Don't rush through chapters. Let each concept settle, and give yourself room to experiment and make mistakes.

1.6.2 Approach Exercises Seriously

This is a hands-on book. Every chapter includes labs and tutorials designed to turn abstract ideas into real experience. The only way to truly learn kernel programming is by doing it yourself, typing the code, running it, breaking it, and fixing it again.

1.6.3 Expect Challenges and Learn from Mistakes

You will face errors, failed compilations, and perhaps even a kernel panic or two. That is not failure; it is part of the process. Some mistakes will be small, others frustrating, but each one is an opportunity to refine your understanding and grow stronger. The most successful developers are not those who avoid mistakes, but those who persist, turning setbacks into stepping stones.

1.6.4 Beginner's Tip

Don't be afraid of mistakes; treat them as milestones. Every failed driver load or kernel panic is proof that you're experimenting and learning. With practice, these mistakes will become your best teachers.

1.6.5 Engage with the Community

FreeBSD is built by a community of volunteers and professionals who share their time and expertise. Don't try to learn in isolation. Use the mailing lists, forums, and chat channels. Ask questions, share your progress, and contribute when you can. Becoming part of the community is one of the most rewarding ways to learn.

1.6.6 Kernel Insight

One reason FreeBSD is an excellent learning platform is its modular driver system. You can write a small driver, load it into the kernel, test it, and unload it, all without rebooting the machine. This makes experimentation safer and faster than you might expect, and it lowers the barrier to trying out new ideas.

1.6.7 Stay Motivated

Remember: you are not just learning to write code, you are learning to shape how an entire operating system interacts with hardware. That is a rare and empowering skill. When progress feels slow, remind yourself that every small step is moving you closer to understanding and influencing the core of a modern operating system.

1.7 Wrapping Up

This first chapter has been about setting the stage. You've walked through my story, seen why FreeBSD is worth your attention, and caught a first glimpse of the role device drivers play in modern systems. The road ahead will bring challenges, but also the satisfaction of overcoming them step by step.

This book is, in many ways, the guide I wish I had when I was starting out. If it saves you even a fraction of the confusion I faced and gives you the same spark of excitement that kept me going, then it has already achieved its purpose.

So take a deep breath. You're about to move from inspiration to action. In the next chapter, we'll roll up our sleeves and set up your FreeBSD lab, the environment where all your learning will take place.

In chemistry, I learned, the lab was where theory met practice. In this journey, your computer is the lab, and it's time to prepare it.

2 Setting Up Your Lab

Before we can begin writing code or exploring FreeBSD's internals, we need a place where it's safe to experiment, make mistakes, and learn. That place is your **lab environment**. In this chapter, we'll create the foundation you'll use throughout the rest of the book: a FreeBSD system configured for driver development.

Think of this chapter as preparing your **workshop**. Just as a carpenter needs the right bench, tools, and safety gear before building furniture, you need a reliable FreeBSD installation, the right development utilities, and a way to recover quickly when things go wrong. Kernel programming is unforgiving; a tiny mistake in your driver can crash the whole system. Having a dedicated lab means those crashes become part of the learning process, not disasters.

After finishing this chapter you will be able to:

- Understand the importance of isolating your experiments from your main computer.
- Choose between using a virtual machine or bare-metal installation.
- Install FreeBSD 14.3 step by step.
- Configure your system with the tools and source code needed for driver development.
- Learn how to take snapshots, manage backups, and use version control so your progress is never lost.

Along the way, we'll pause for **hands-on labs** so you don't just read about setting things up, you'll actually do it. When you finish, you'll have a FreeBSD lab that is safe, repeatable, and ready for everything we'll build together in the coming chapters.

2.0.1 Reader Guidance: How to Use This Chapter

This chapter is more practical than theoretical. Think of it as a step-by-step manual to set up your FreeBSD lab before you begin the real experiments. You'll be asked to make choices (virtual machine or bare metal), follow installation steps, and configure your FreeBSD system.

The best way to use this chapter is to **do the steps as you read them**. Don't just skim through, actually install FreeBSD, take the snapshots, write down your choices in your lab logbook, and try the exercises. Each section builds on the last, so by the end you'll have a complete environment that matches the examples in the rest of this book.

If you already know how to install and configure FreeBSD, you may skim or skip parts of this chapter, but don't skip the labs; they ensure your setup matches what we'll use throughout the book.

Above all, remember: mistakes here are not failures, they are part of the process. This is your safe place to experiment and learn.

Estimated time to complete this chapter: 1-2 hours, depending on your choice of virtual machine or bare-metal installation, and whether you already have experience installing operating systems.

2.1 Why a Lab Environment Matters

Before we start typing commands and writing our first pieces of code, we need to pause for a moment and think about *where* we are going to do all this work. Kernel programming and device driver development are not like writing a simple script or a web page. When you experiment with the kernel, you are experimenting with the **heart of the operating system**. A small mistake in your code can cause your machine to freeze, reboot unexpectedly, or even corrupt data if you're not careful.

This doesn't mean driver development is dangerous, it means we need to take precautions and set up a **safe environment** where mistakes are expected, recoverable, and even encouraged as part of the learning process. That environment is what we'll call your **lab**.

Your lab should be a **dedicated, isolated environment**. Just like a chemist wouldn't run an experiment on the family dining table without protective equipment, you shouldn't run unfinished kernel code on the same machine where you keep your personal photos, work documents, or important school projects. You need a space designed for exploration and failure, because failure is how you'll learn.

2.1.1 Why Not Use Your Main Computer?

It's tempting to think: *"I already have a computer running FreeBSD (or Linux, or Windows), why not just use that?"* The short answer: because your main computer is for productivity, not experiments. If you accidentally cause a kernel panic while testing your driver, you don't want to lose unsaved work, crash your network connection during an online meeting, or even damage a filesystem with corrupted data.

Your lab setup gives you freedom: you can break things, restart, and recover in minutes without stress. This freedom is essential for learning.

2.1.2 Virtual Machines: The Beginner's Best Friend

Most beginners (and even many experienced developers) start with **virtual machines (VMs)**. A VM is like a sandboxed computer running inside your real computer. It acts just like a physical machine, but if something goes wrong, you can reset it, take a snapshot, or reinstall FreeBSD in minutes. You don't need a spare laptop or server to start developing drivers; your current computer can host your lab.

We'll cover virtualization in more detail in the next section, but here are the highlights:

- **Safe experiments:** If your driver crashes the kernel, only the VM goes down, not your host computer.
- **Easy recovery:** Snapshots let you save the VM's state and roll back instantly if you break something.
- **Low cost:** No need for dedicated hardware.
- **Portable:** You can move your VM between computers.

2.1.3 Bare Metal: When You Need the Real Thing

There are times when only real hardware will do, for example, if you want to develop a driver for a PCIe card or a USB device that requires direct access to the machine's bus. In those cases, testing in a VM might not be enough because not all virtualization solutions can pass through hardware reliably.

If you have an old spare PC, installing FreeBSD there can give you the closest-to-reality environment for testing. But remember: bare metal setups don't have the same safety net as VMs. If you crash the kernel, your machine will reboot, and you'll need to recover manually. That's why I recommend starting in a VM, even if you eventually move to bare metal for specific hardware projects.

2.1.4 Real-World Example

To make this more concrete: imagine you're writing a simple driver that mistakenly dereferences a NULL pointer (don't worry if that sounds technical now, you'll learn all about it later). On a VM, your system might freeze, but with a reset and a snapshot rollback, you're back in action within minutes. On bare metal, the same mistake could cause filesystem corruption, requiring a lengthy recovery process. That's why a safe lab environment is so valuable.

2.1.5 Hands-On Lab: Preparing Your Lab Mindset

Before we even install FreeBSD, let's do a simple exercise to get into the right mindset.

1. Grab a notebook (physical or digital). This will be your **lab logbook**.
2. Write down:
 - Today's date
 - The machine you're going to use for your FreeBSD lab (VM or physical)
 - Why you chose that option (safety, convenience, access to real hardware, etc.)
3. Make a first entry: *"Lab setup started. Goal: build a safe environment for FreeBSD driver experiments."*

This may feel unnecessary, but keeping a **lab log** will help you track your progress, repeat successful setups later, and debug when something goes wrong. In professional driver development, engineers keep very detailed notes; starting this habit now will make you think and work like a real systems developer.

2.1.6 Wrapping Up

We've introduced the idea of your **lab environment** and why it's so important for safe driver development. Whether you choose a virtual machine or a spare physical computer, the key is having a dedicated place where it's okay to make mistakes.

In the next section, we'll look more closely at the **pros and cons of virtual machines versus bare metal setups**. By the end of that section, you'll know exactly which setup makes sense for you to begin your FreeBSD work.

2.2 Choosing Your Setup: Virtual Machine or Bare Metal

Now that you understand why a dedicated lab environment is important, the next question is: **where should you build it?** FreeBSD can run in two main ways for your experiments:

1. **Inside a virtual machine (VM)**, running on top of your existing operating system.
2. **Directly on physical hardware** (often called *bare metal*).

Both options work, and both are widely used in real-world FreeBSD development. The right choice depends on your goals, your hardware, and your comfort level. Let's compare them side by side.

2.2.1 Virtual Machines: Your Sandbox in a Box

A **virtual machine** is a piece of software that lets you run FreeBSD as if it were a separate computer, but inside your existing computer. Popular VM solutions include:

- **VirtualBox** (free and cross-platform, great for beginners).
- **VMware Workstation / Fusion** (commercial, polished, widely used).
- **bhyve** (FreeBSD's native hypervisor, ideal if you want to run FreeBSD *on* FreeBSD).

Why developers love VMs for kernel work:

- **Snapshots save the day:** Before trying risky code, you take a snapshot. If the system panics or breaks, you restore in seconds.
- **Multiple labs on one machine:** You can create several FreeBSD instances, each for a different project.
- **Easy to share:** You can export a VM image and share it with a teammate.

When to prefer VMs:

- You're just starting out and want the safest environment possible.
- You don't have spare hardware to dedicate.
- You expect to crash the kernel often while learning (and you will).

2.2.2 Bare Metal: The Real Deal

Running FreeBSD **directly on hardware** is as close as you can get to "the real thing". This means FreeBSD boots as the only operating system on the machine, talking directly to the CPU, memory, storage, and peripherals.

Advantages:

- **True hardware testing:** Essential when developing drivers for PCIe, USB, or other physical devices.
- **Performance:** No VM overhead. You get full access to your system's resources.
- **Accuracy:** Some bugs only appear on bare metal, especially timing-related issues.

Disadvantages:

- **No safety net:** If the kernel crashes, your whole machine goes down.
- **Recovery takes time:** If you corrupt the OS, you may need to reinstall FreeBSD.
- **Dedicated hardware needed:** You'll need a spare PC or laptop you can dedicate fully to experiments.

When to prefer bare metal:

- You plan to develop a driver for hardware that doesn't work well in VMs.
- You already have a spare machine you can dedicate entirely to FreeBSD.
- You want maximum realism, even if it means more risk.

2.2.3 Hybrid Strategy

Many professional developers use **both**. They do most of their experimentation and prototyping in a VM, where it's safe and fast, and only move to bare metal once their driver is stable enough to test against real hardware. You don't have to commit to one forever, you can start with a VM today and add a bare-metal machine later if you need it.

2.2.4 Quick Comparison Table

Feature	Virtual Machine	Bare Metal
Safety	Very high (snapshots, rollback)	Low (manual recovery needed)
Performance	Slightly lower (overhead)	Full system performance
Hardware access	Limited / emulated devices	Full, real hardware
Setup difficulty	Easy and fast	Moderate (full install)
Cost	None (runs on your PC)	Requires dedicated machine
Best for	Beginners, safe learning	Advanced hardware testing

2.2.5 Hands-On Lab: Deciding Your Path

1. Look at your current resources. Do you have a spare laptop or desktop that you can dedicate to FreeBSD experiments?
 - If yes -> Bare metal is an option for you.
 - If no -> A VM is the perfect starting point.
2. In your **lab logbook**, write down:
 - Which option you will use (VM or bare metal).
 - Why you chose it.
 - Any limitations you expect (for example, "Using a VM, may not test USB passthrough yet").
3. If you choose VM, note which hypervisor you'll use (VirtualBox, VMware, bhyve, etc).

This decision doesn't lock you in forever. You can always add a second environment later. The goal now is to start with a safe, reliable setup.

2.2.6 Wrapping Up

We've compared virtual machines and bare metal setups and seen the strengths and trade-offs of each. For most beginners, starting with a VM is the best balance of safety, convenience, and flexibility. If you later need to interact with real hardware, you can add a bare-metal system to your toolkit.

In the next section, we'll roll up our sleeves and perform the actual **FreeBSD 14.3 installation**, first in a VM, and then we'll cover the key points for bare-metal installs. This is where your lab truly begins to take shape.

2.3 Installing FreeBSD (VM and Bare Metal)

At this point, you've chosen whether to set up FreeBSD in a **virtual machine** or on **bare metal**. Now it's time to actually install the operating system that will serve as the foundation for all our experiments. We'll focus on **FreeBSD 14.3**, the latest stable release at the time of writing, so that everything you do matches the examples in this book.

FreeBSD's installer is text-based, but don't let that intimidate you, it's straightforward, and in less than 20 minutes you'll have a working system ready for development.

2.3.1 Downloading the FreeBSD ISO

1. Visit the official FreeBSD download page: <https://www.freebsd.org/where>
2. Choose the **14.3-RELEASE** image.
 - If you are installing in a VM, download the **amd64 Disk1 ISO** (FreeBSD-14.3-RELEASE-amd64-disc1.iso).
 - If you're installing on real hardware, the same ISO works, though you might also consider the **memstick image** if you want to write it to a USB stick.

2.3.2 Installing FreeBSD in VirtualBox (Step by Step)

If you don't already have **VirtualBox** installed, you'll need to set it up before creating your FreeBSD VM. VirtualBox is available for Windows, macOS, Linux, and even Solaris hosts. Download the latest version from the official website:

<https://www.virtualbox.org/wiki/Downloads>

Choose the package that matches your host operating system (for example, Windows hosts or macOS hosts), download it, and follow the installer. The installation is straightforward and only takes a few minutes. Once it's complete, launch VirtualBox, and you'll then be ready to create your first FreeBSD virtual machine.

Now that you are ready, let's go through the VirtualBox process, since that's the easiest entry point for most readers. The steps are similar in VMware or bhyve.

To get started execute VirtualBox App in your computer and in the main screen select **Home** in the left column, and then click in **New**, and follow the steps below:

1. **Create a new VM** in VirtualBox:
 - VM Name: FreeBSD Lab
 - VM Folder: Choose a directory to hold your FreeBSD VM
 - ISO Image: Choose the FreeBSD Iso file that you downloaded above
 - OS Edition: Leave in blank
 - OS: Choose BSD
 - OS Distribution: Choose FreeBSD

- OS Version: Choose FreeBSD (64-bit)

Click in **Next** to continue

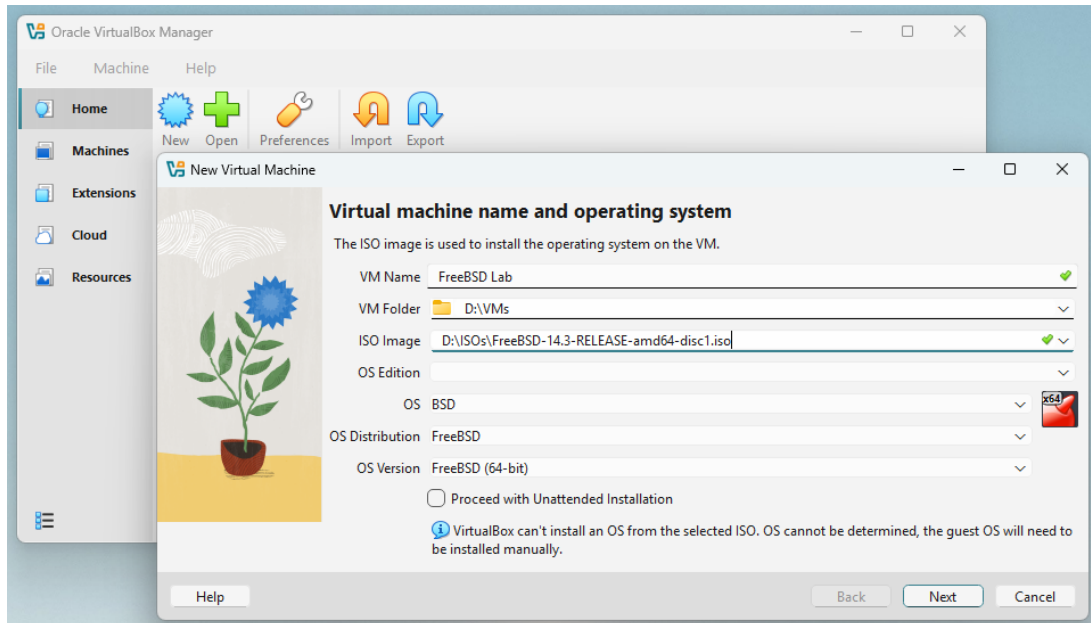


Figure 2.1: image-20250823183742036

2. Allocate resources:

- Base Memory: at least 2 GB (4 GB recommended).
- Number of CPUs: 2 or more if available.
- Disk Size: 30 GB or more.

Click in **Next** to continue

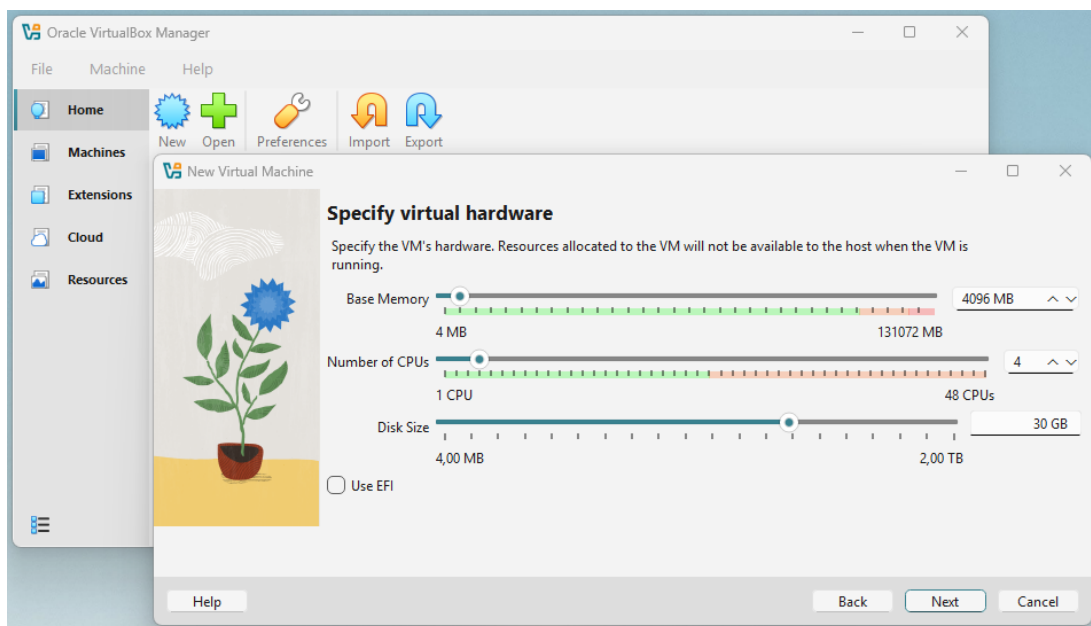


Figure 2.2: image-20250823183937505

3. **Review your options:** If you are happy with the summary, click in **Finish** to create the VM.

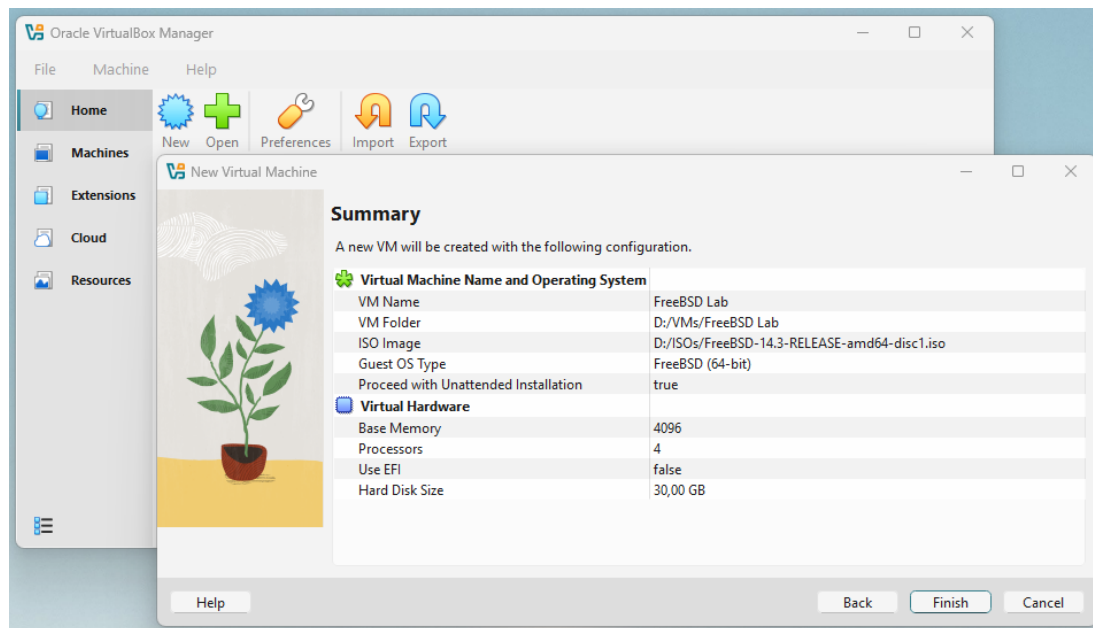


Figure 2.3: image-20250823184925505

4. **Start the virtual machine:** Click in the green **Start** button. When starting the VM, it will boot using the FreeBSD installation disk that you specified when creating it.

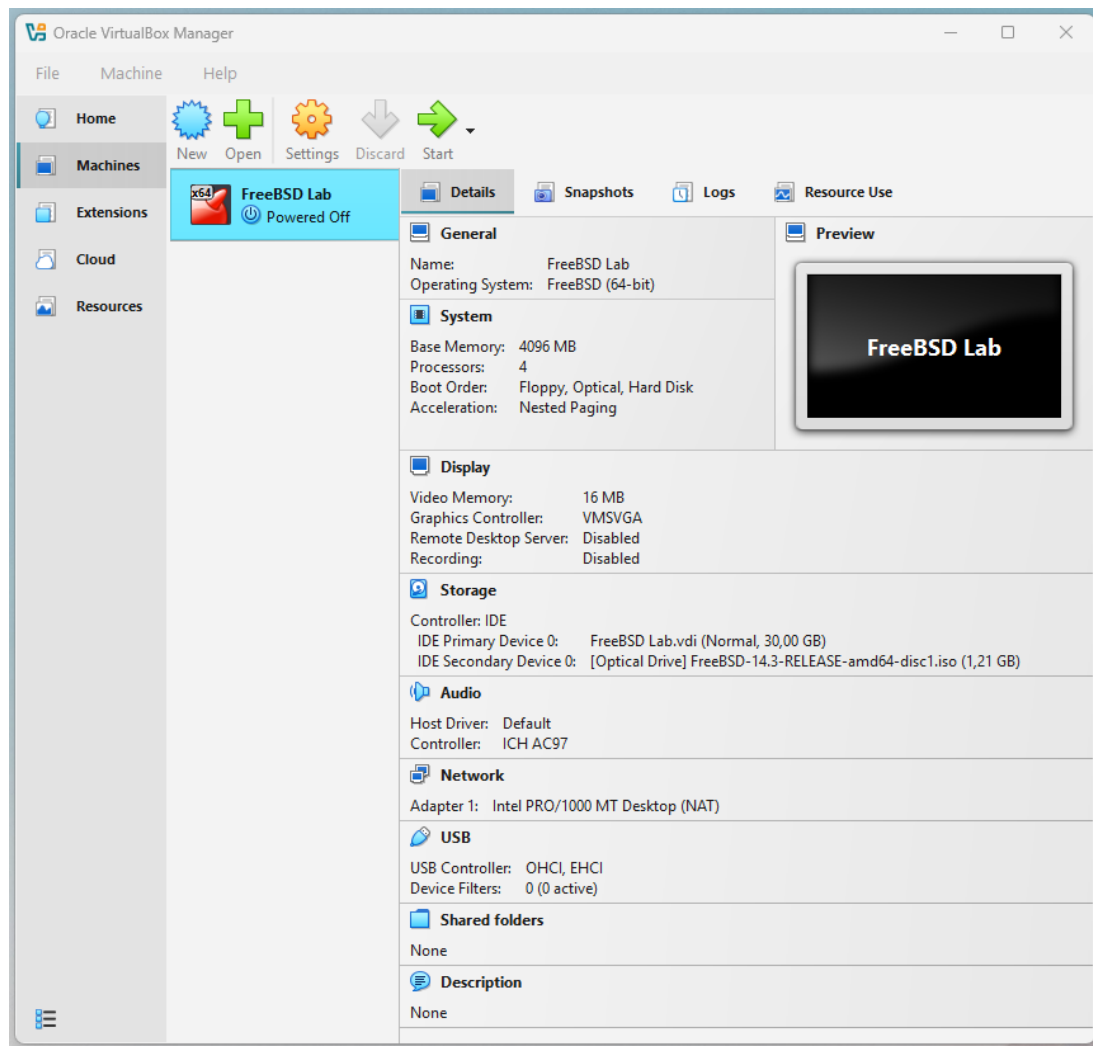


Figure 2.4: image-20250823185259010

5. **Boot the VM:** The VM will show the FreeBSD boot loader, press **1** to continue the boot into the FreeBSD.

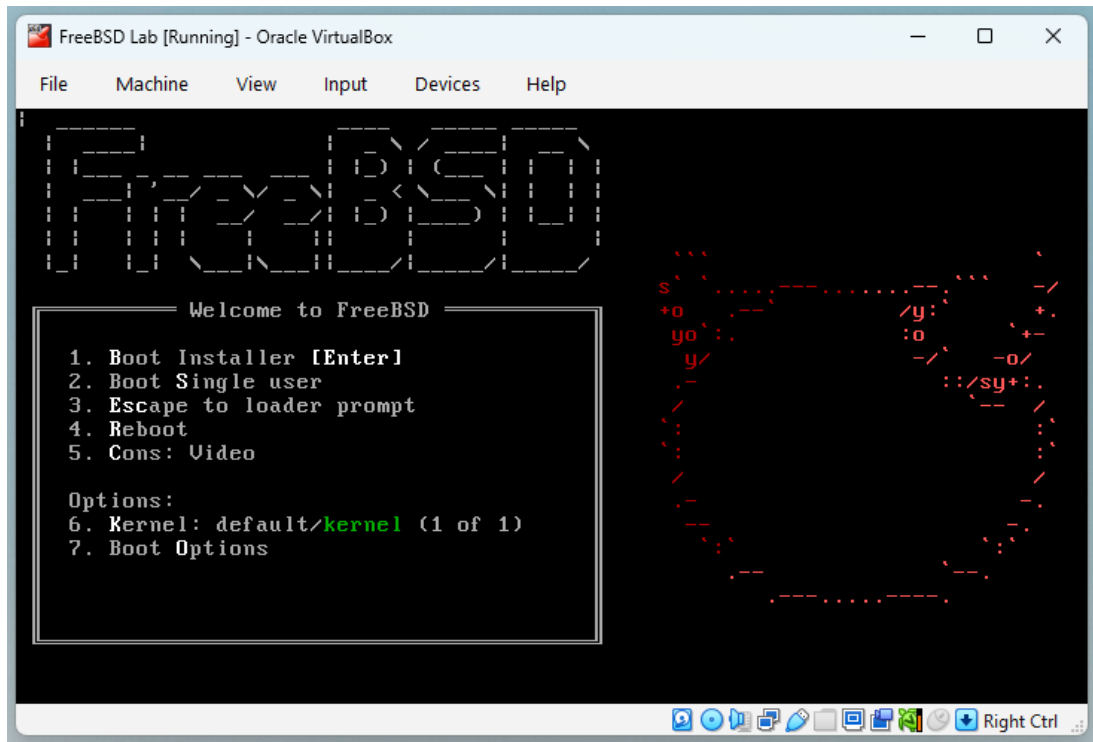


Figure 2.5: image-20250823185756980

6. **Run the installer:** The boot process, the installer will run automatically, choose [**Install**] to continue.

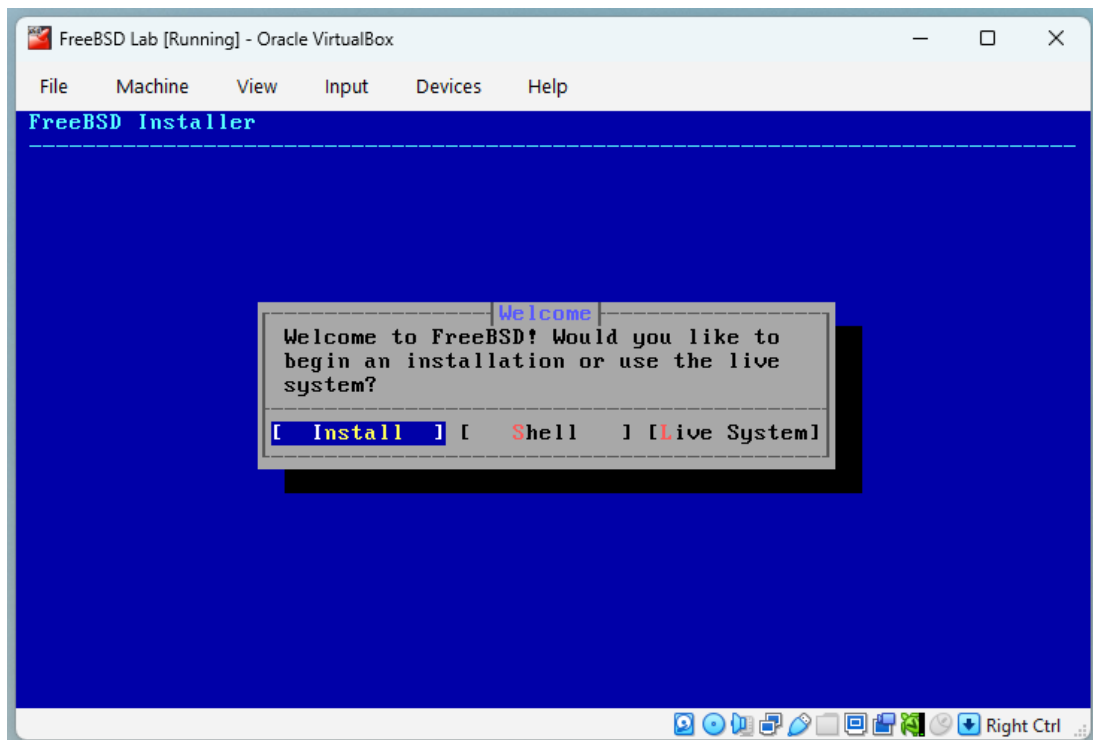


Figure 2.6: image-20250823190016799

7. **Keyboard layout:** Choose your preferred language/keyboard, the default is US layout. Press **Enter** to continue.

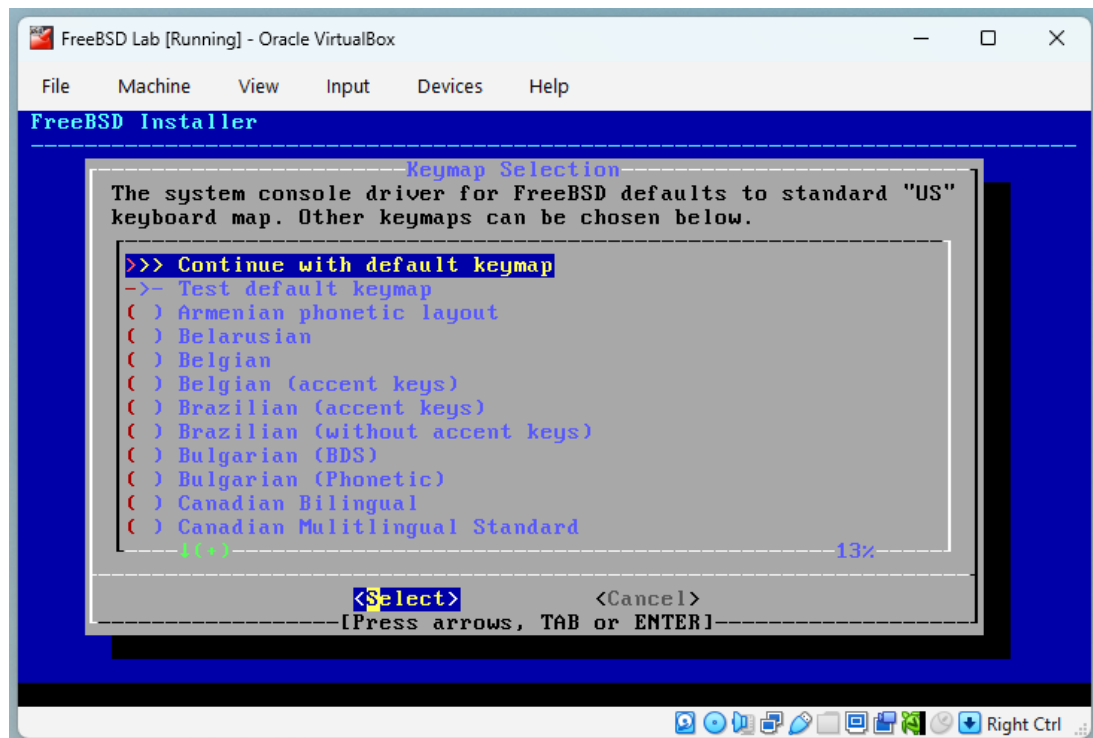


Figure 2.7: image-20250823190046619

8. **Hostname:** Type the hostname for your lab, in the example I choose fbsd-lab. Press **Enter** to continue.

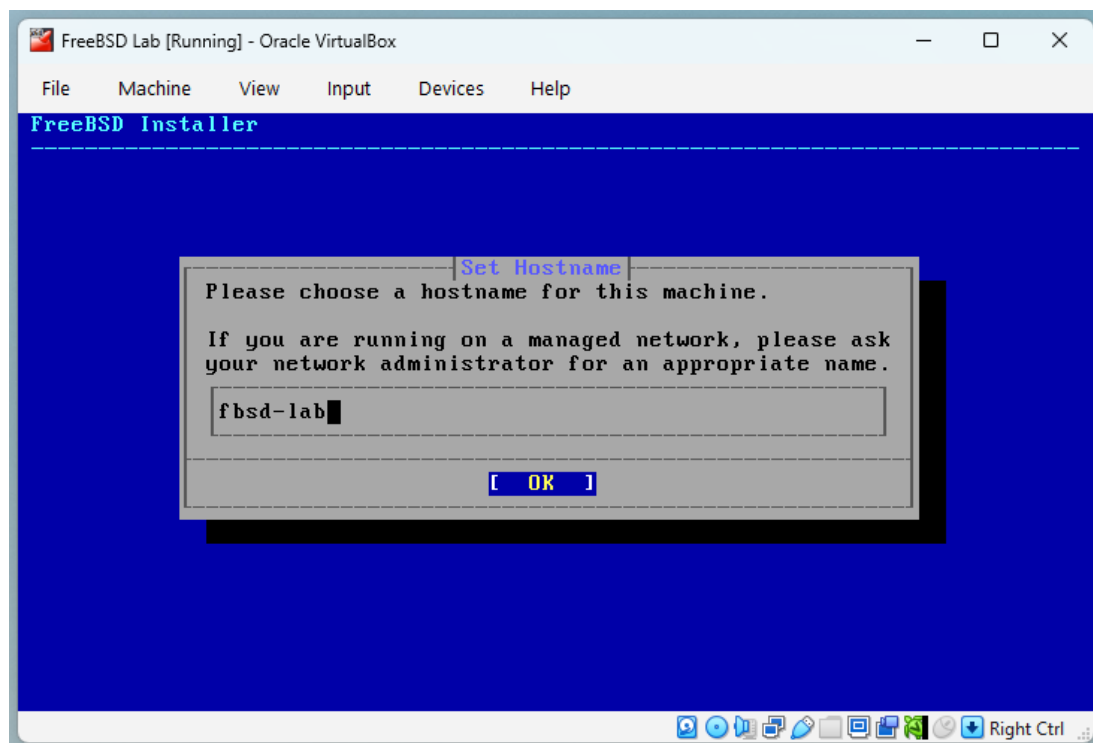


Figure 2.8: image-20250823190129010

9. **Distribution Select:** Leave defaults (base system, kernel). Press **Enter** to continue.

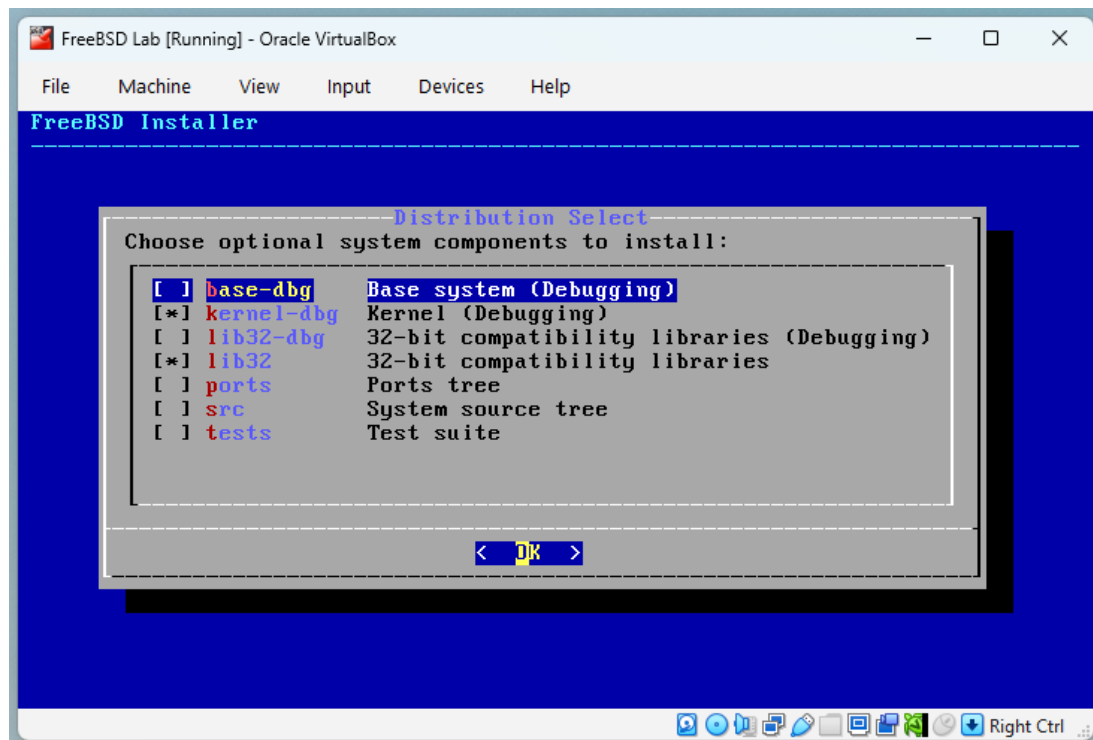


Figure 2.9: image-20250823190234155

10. **Partitioning:** Choose *Auto (UFS)* unless you want to learn ZFS later. Press **Enter** to continue.

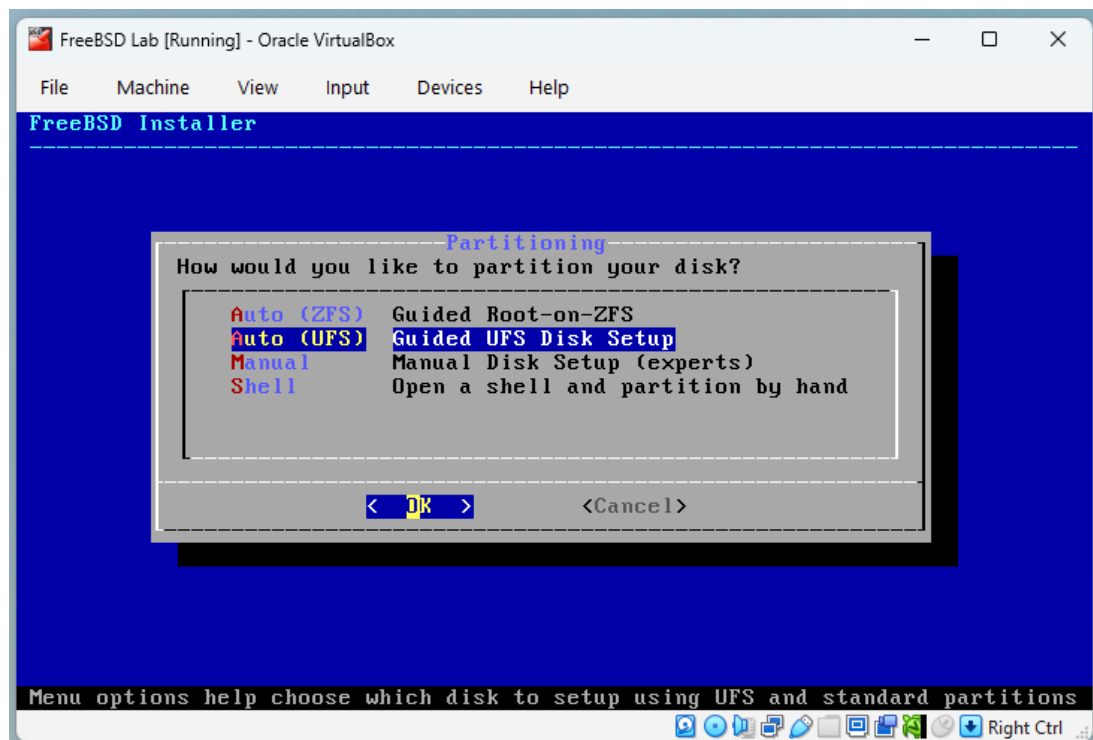


Figure 2.10: image-20250823190350815

11. **Partition:** Choose [**Entire Disk**]. Press **Enter** to continue.

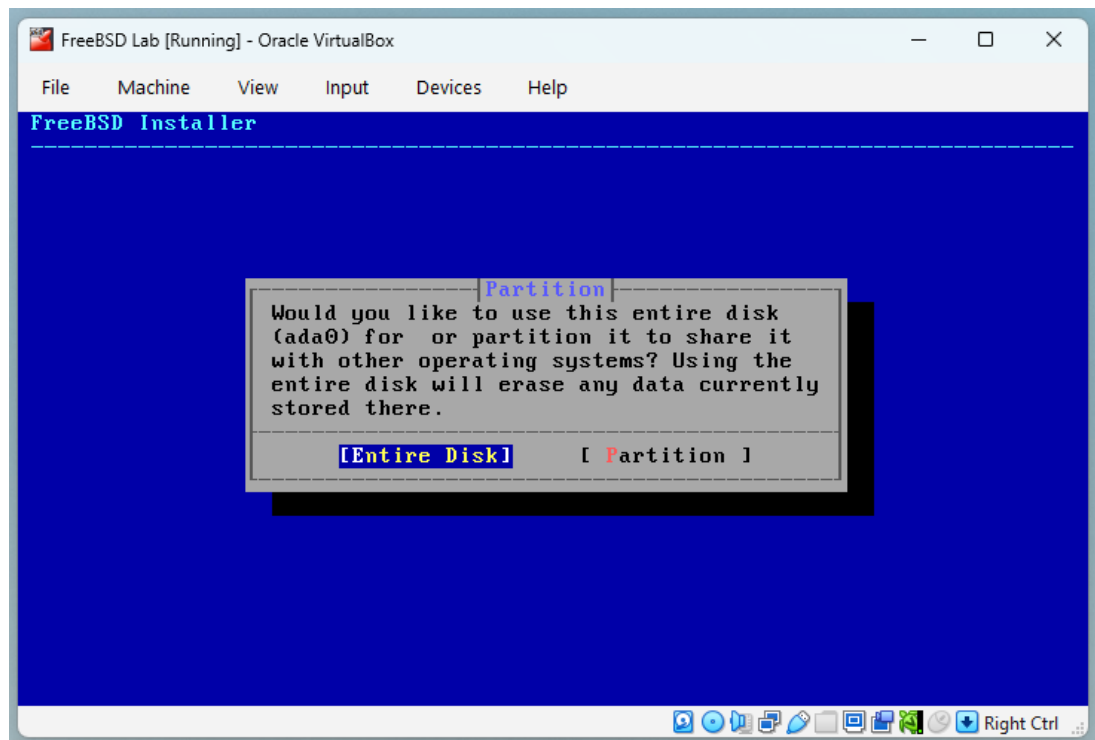


Figure 2.11: image-20250823190450571

12. **Partition Scheme:** Choose **GPT GUID Partition Table**. Press **Enter** to continue.

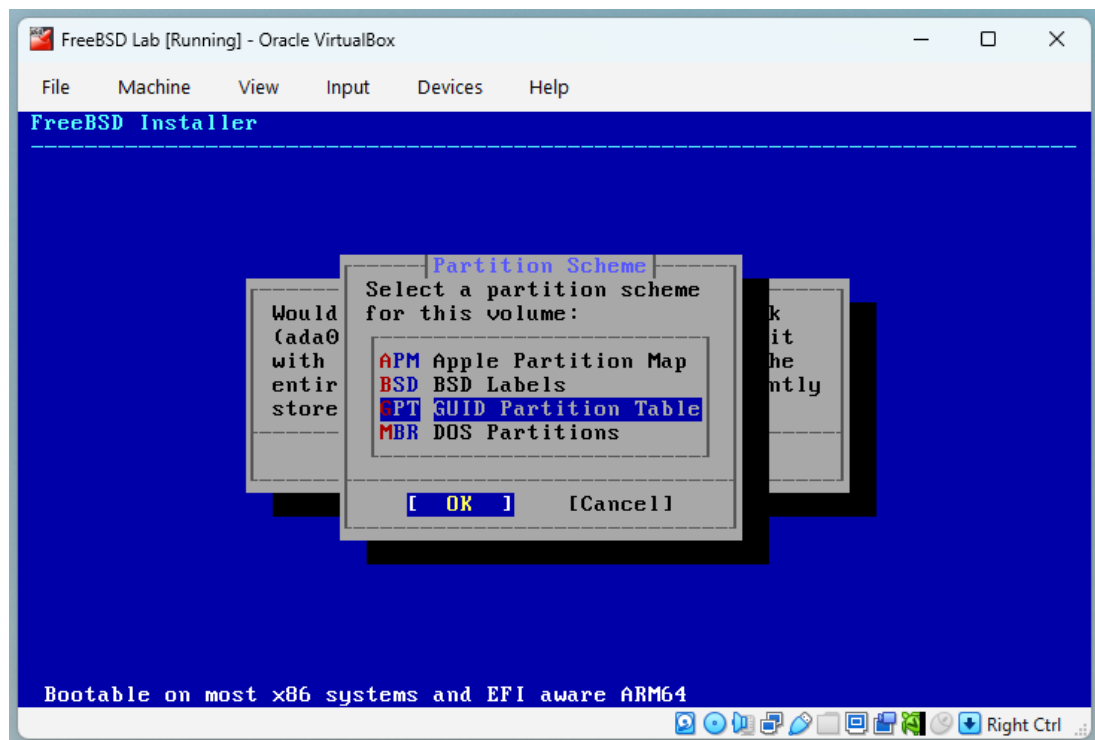


Figure 2.12: image-20250823190622981

13. **Partition Editor:** Accept the default, and choose **[Finish]**. Press **Enter** to continue.

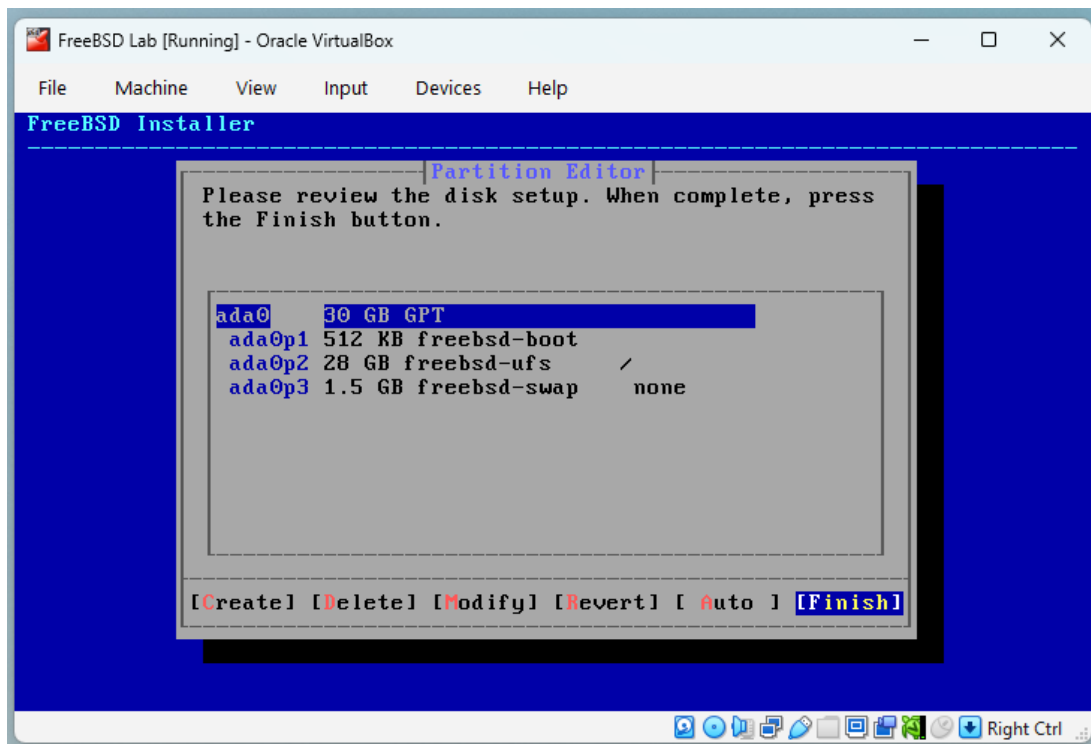


Figure 2.13: image-20250823190742861

14. **Confirmation:** In this screen you will confirm that you want to proceed with the FreeBSD Installation, after this confirmation the installer will start writing data to your hard disk. To proceed with the installation choose **[Commit]** and press **Enter** to continue.

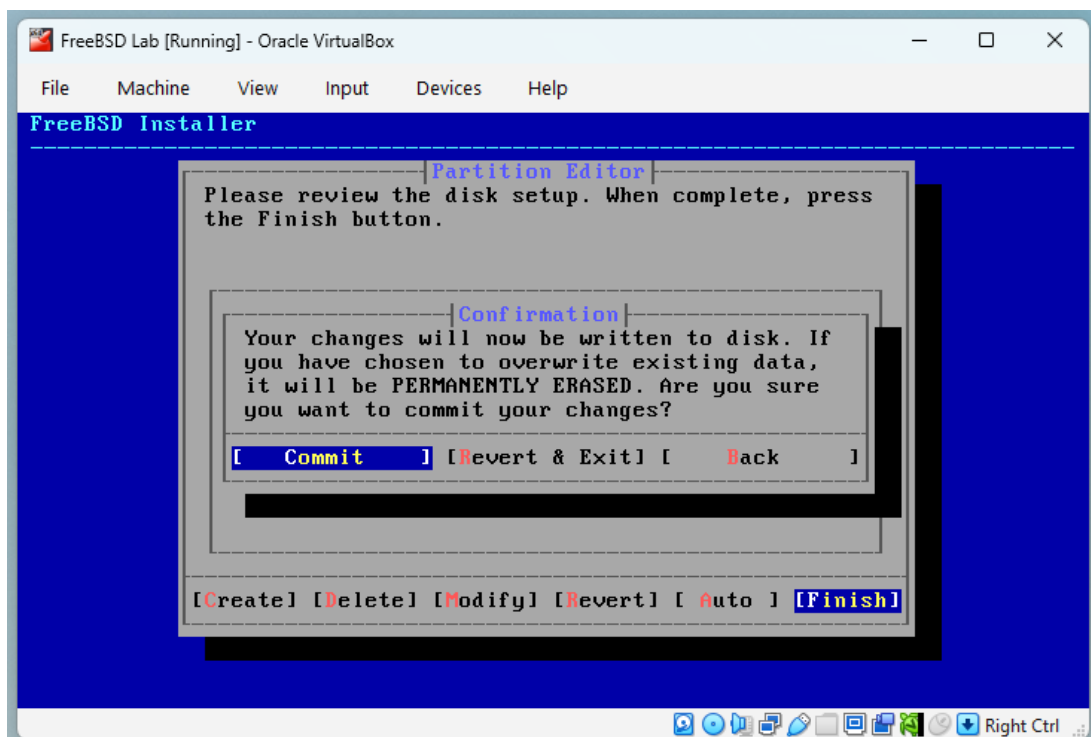


Figure 2.14: image-20250823190903913

15. **Checksum Verification:** At beginning of the process FreeBSD installer will check the integrity of the installation files.

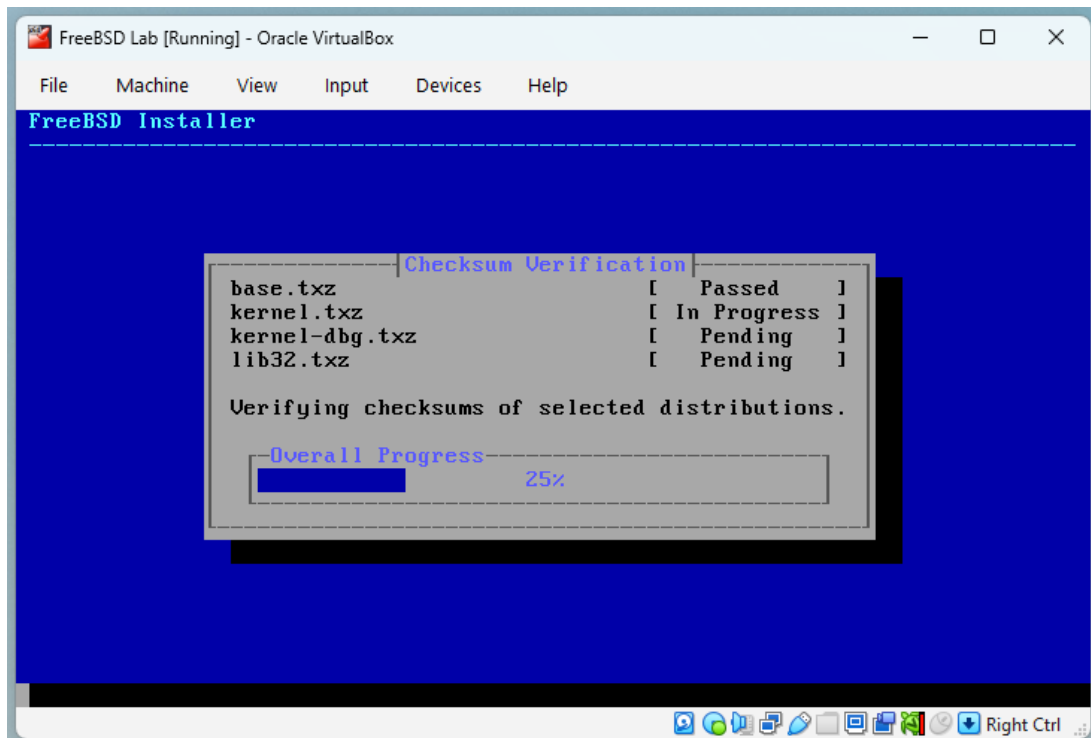


Figure 2.15: image-20250823191020839

16. **Archive Extraction:** After the files were validated, the installer will extract the files to your hard disk

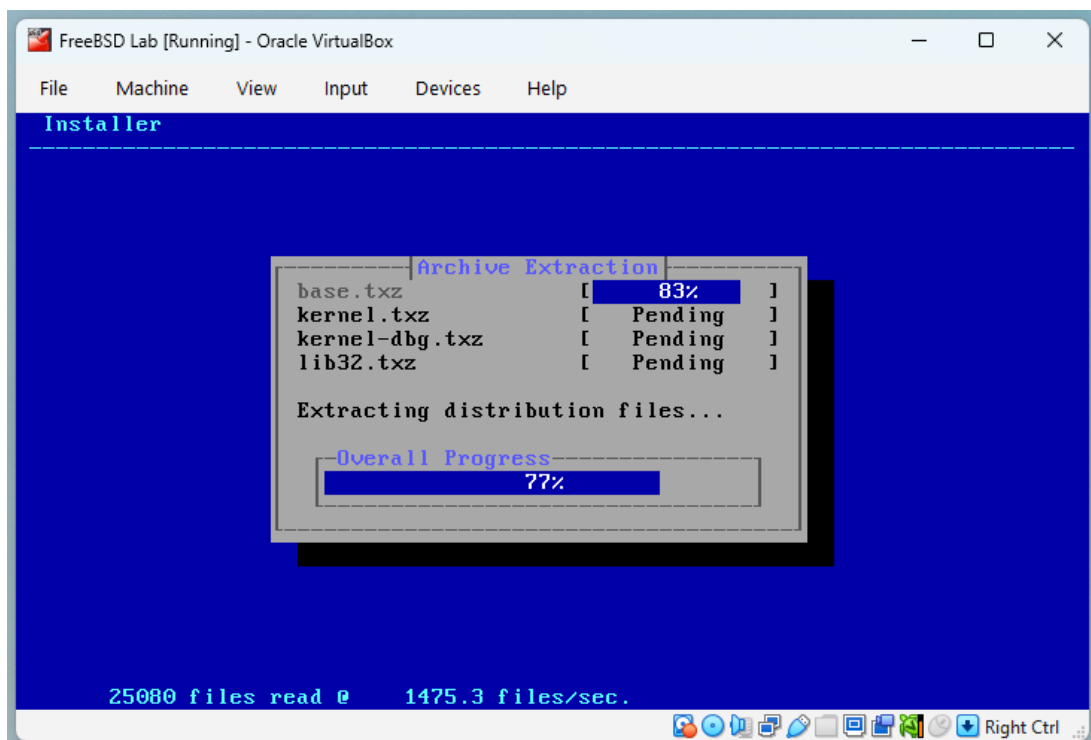


Figure 2.16: image-20250823191053163

17. **Root Password:** When the installer finish extracting the files, you will need to choose a password for your root access. Choose one that you will remember. Press **Enter** to continue.

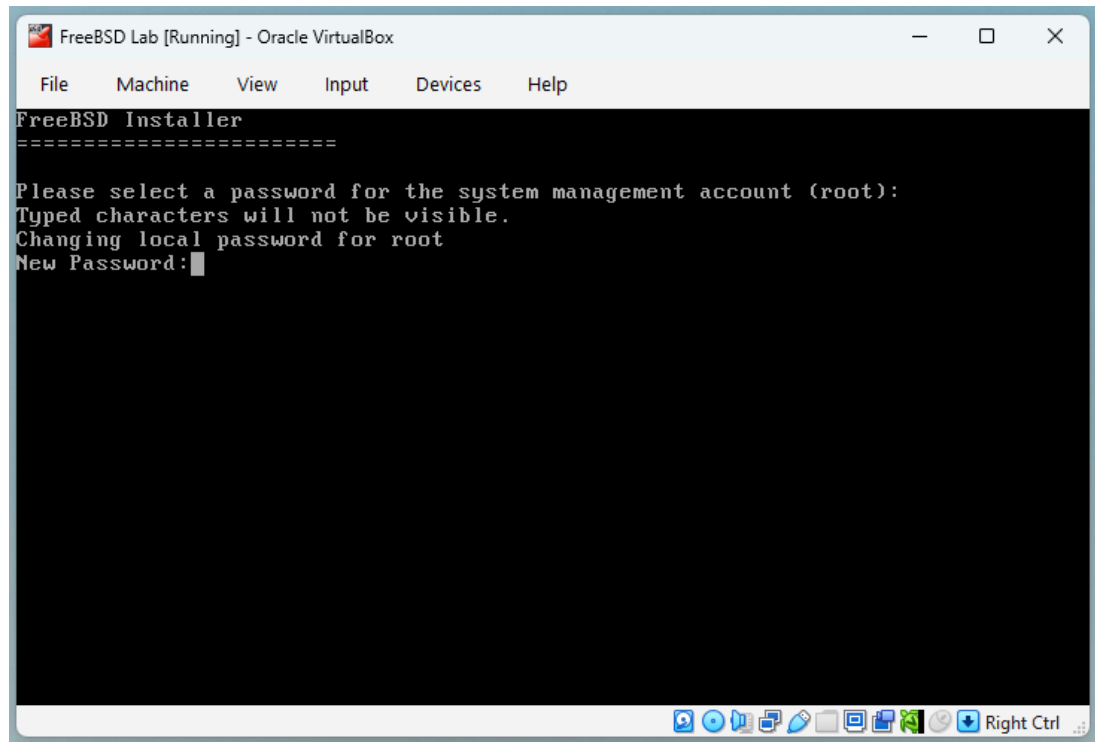


Figure 2.17: image-20250823191405000

18. **Network Configuration:** Choose the network interface (**em0**) that you want to use and press **Enter** to continue.

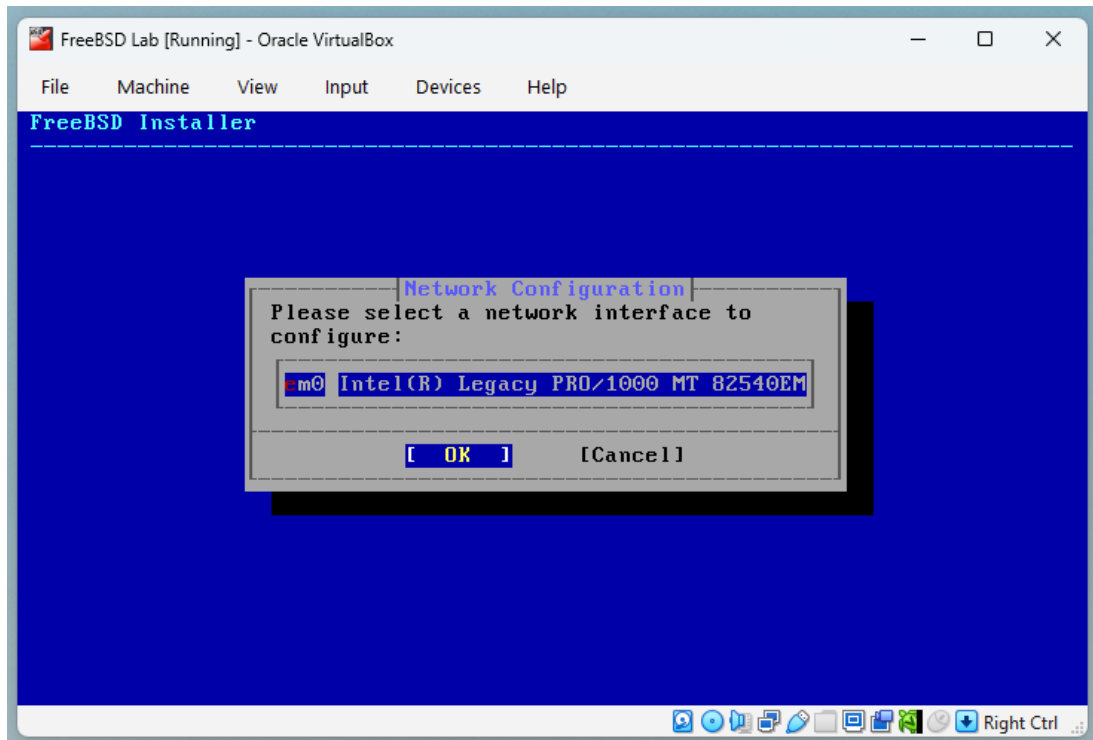


Figure 2.18: image-20250823191520068

19. **Network Configuration:** Choose [Yes] to enable IPv4 in your network interface and press **Enter** to continue.

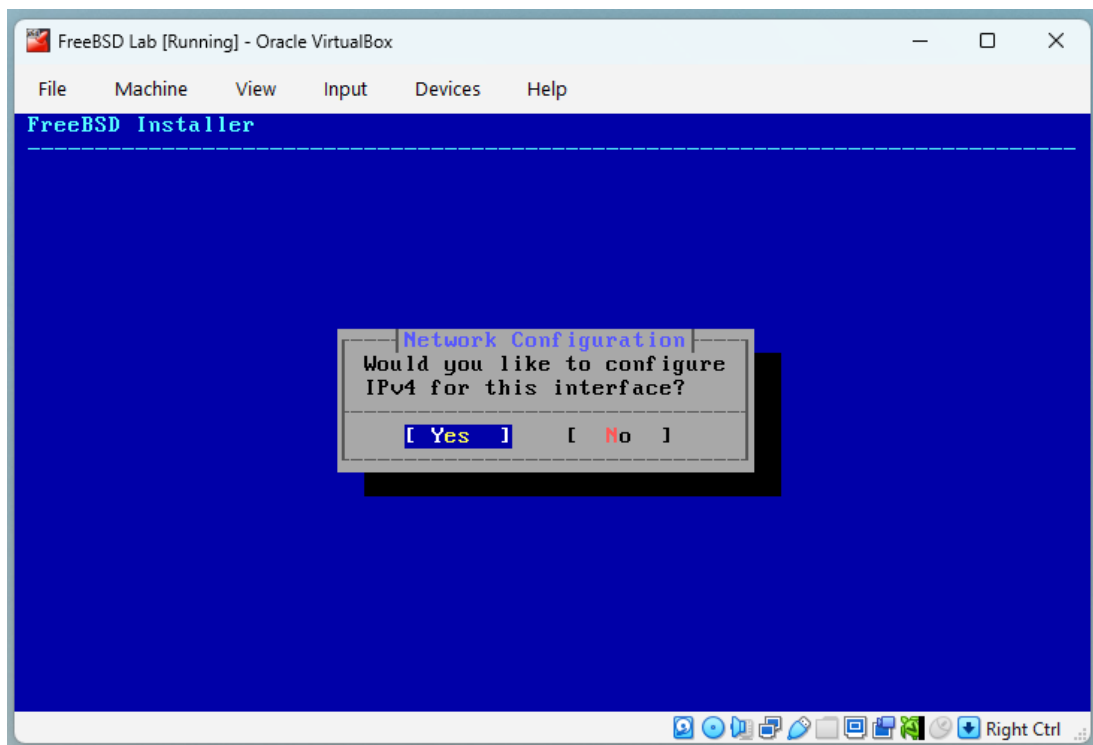


Figure 2.19: image-20250823191559429

20. **Network Configuration:** Choose [Yes] to enable DHCP in your network interface ,if you prefer to use

a static IP address, choose [**No**]. Press **Enter** to continue.

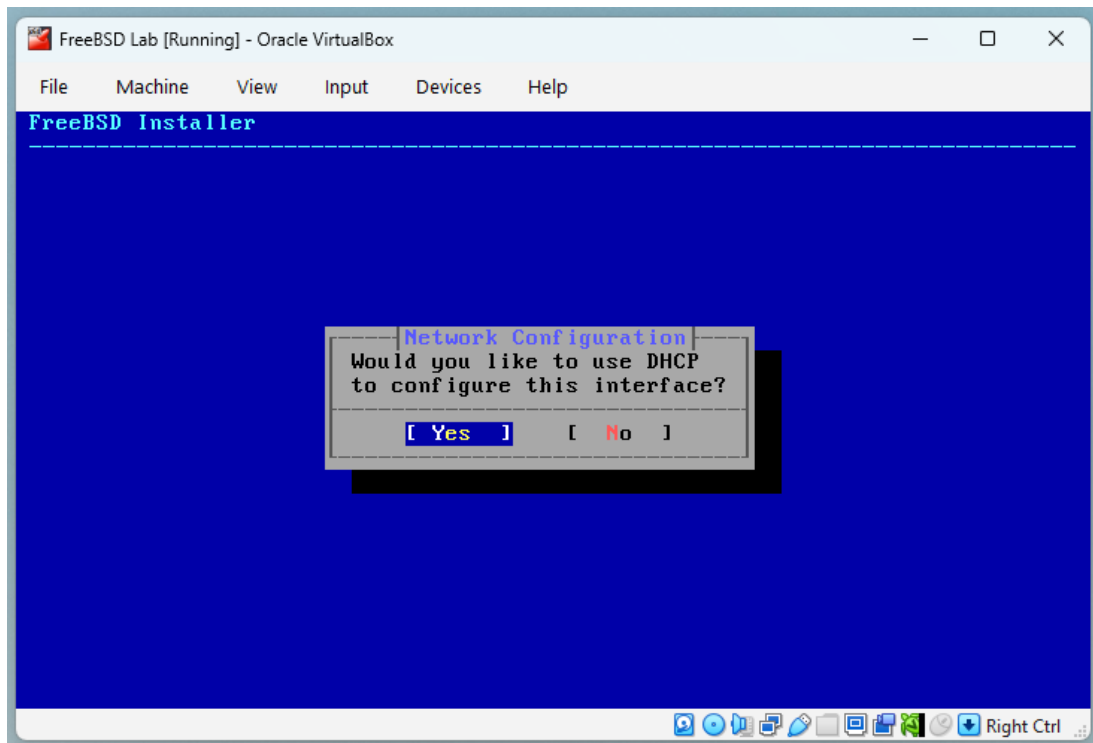


Figure 2.20: image-20250823191626027

21. **Network Configuration:** Choose [**No**] to disable IPv6 in your network interface and press **Enter** to continue.

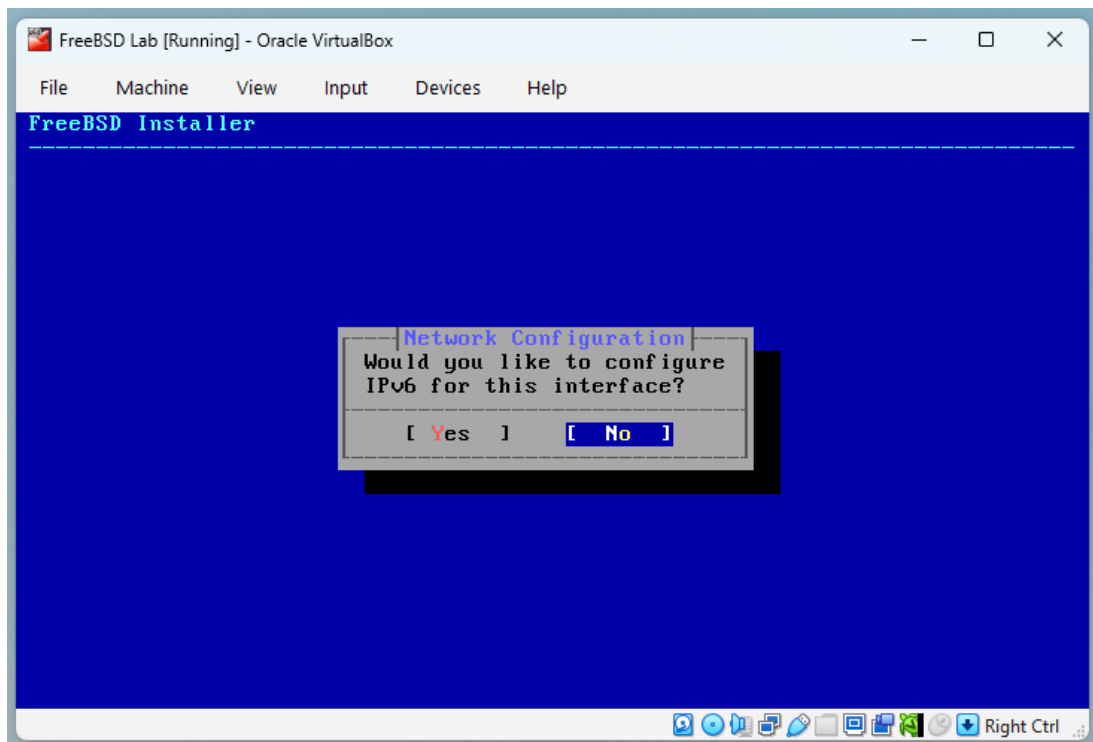


Figure 2.21: image-20250823191705347

22. **Network Configuration:** Type the IP Address of your preferred DNS servers, in the example I'm using the Google DNS. Press **Enter** to continue.

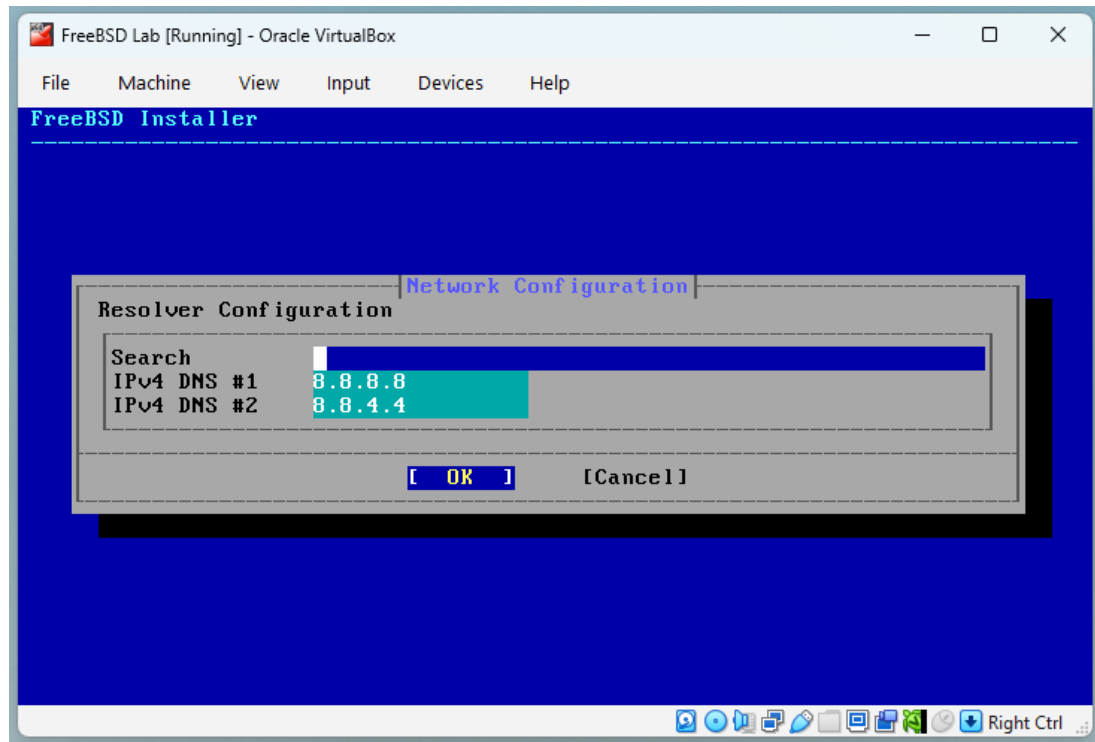


Figure 2.22: image-20250823191748088

23. **Time Zone Selector:** Choose the desired Time Zone for your FreeBSD System, for this example I'm using **UTC**. Press **Enter** to continue.

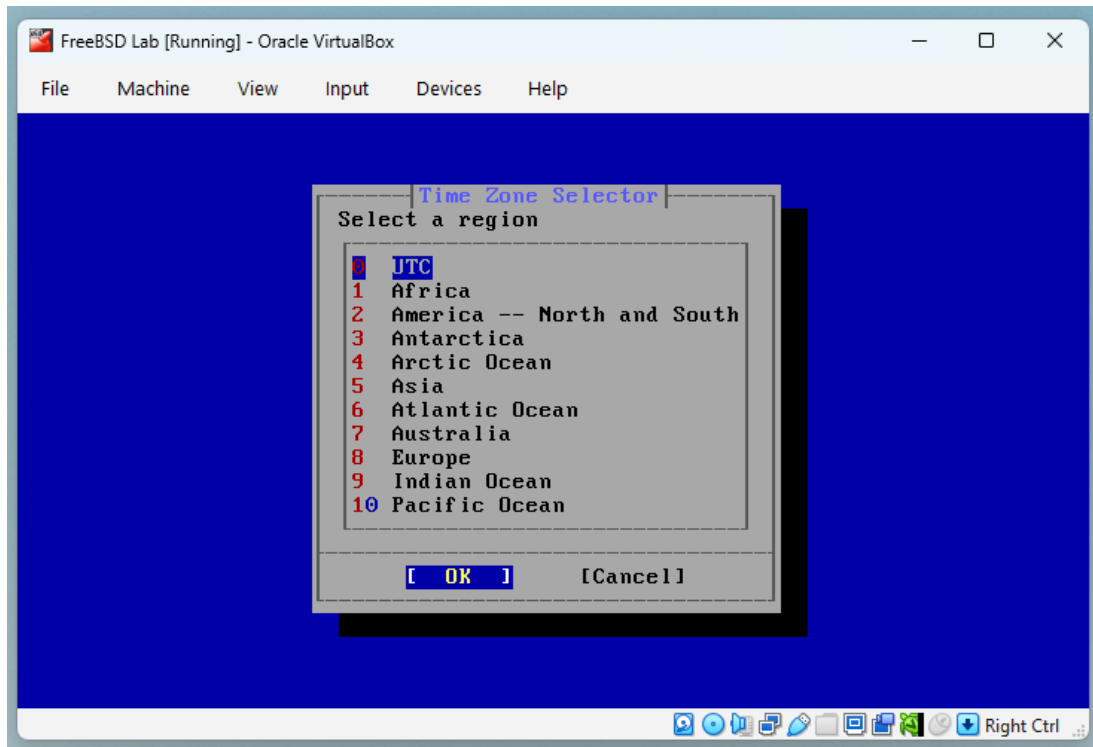


Figure 2.23: image-20250823191820859

24. **Confirm Time Zoner:** Confirm the Time Zone you want to use. Choose [YES] and press **Enter** to continue.

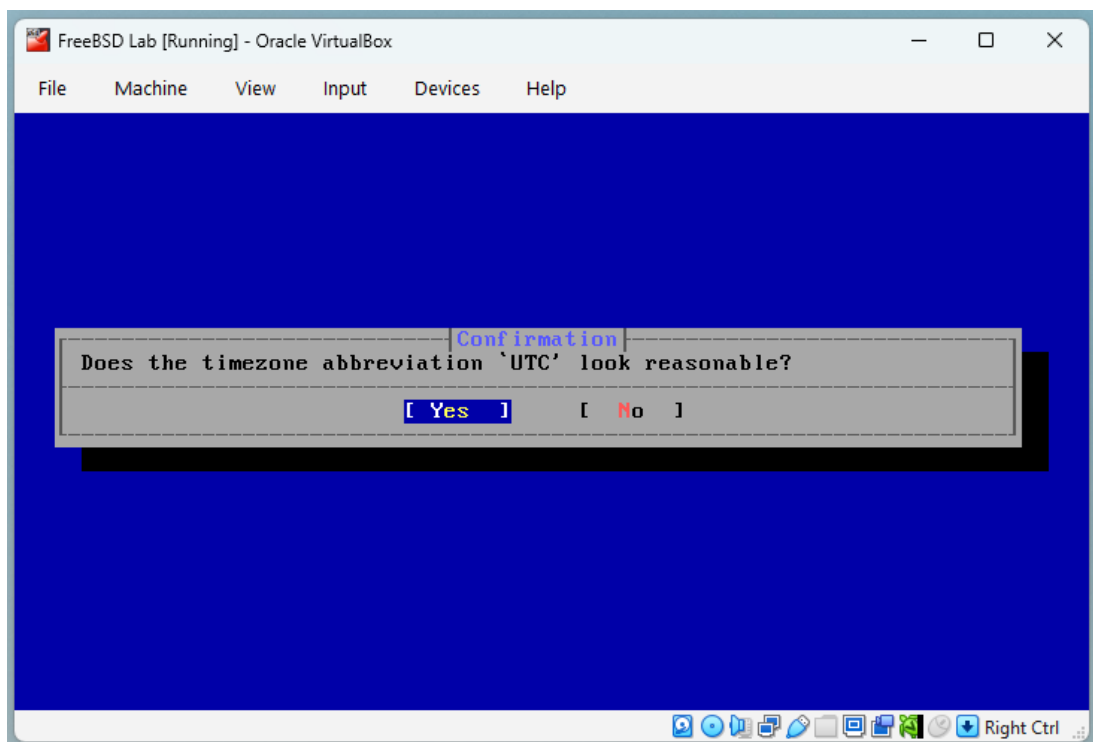


Figure 2.24: image-20250823191849469

25. **Time & Date:** The installer will give you the chance to adjust manually the Date and Time. Normally its

safe to choose [**Skip**]. Press **Enter** to continue.

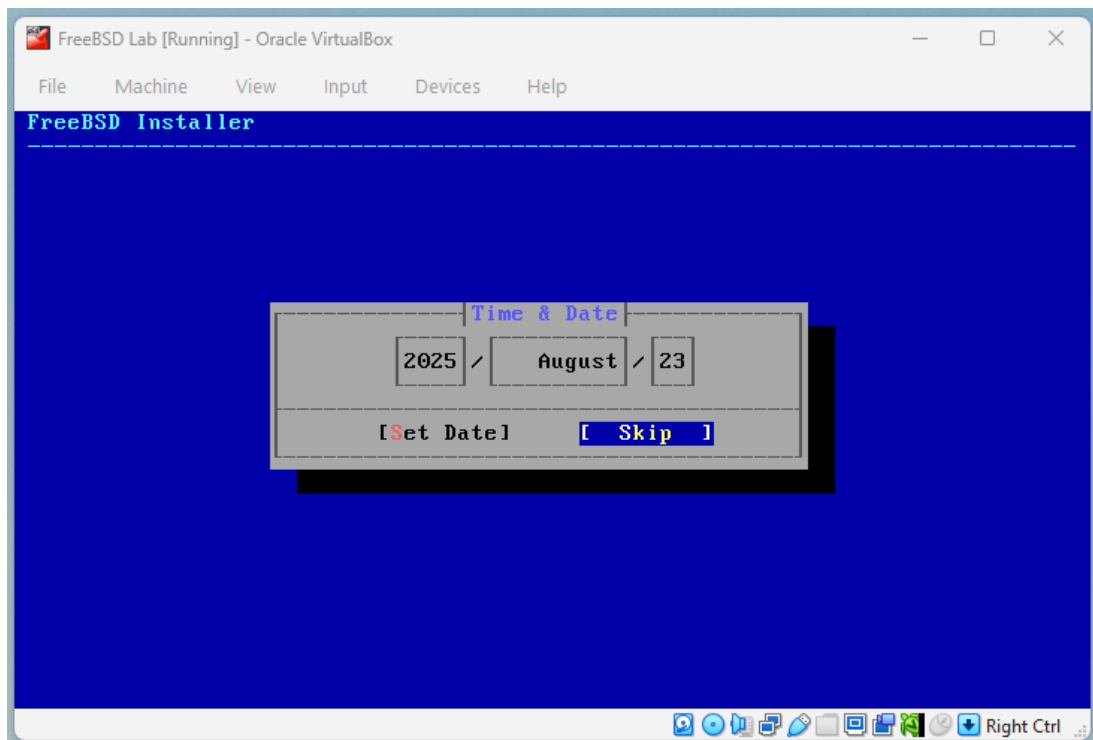


Figure 2.25: image-20250823191926758

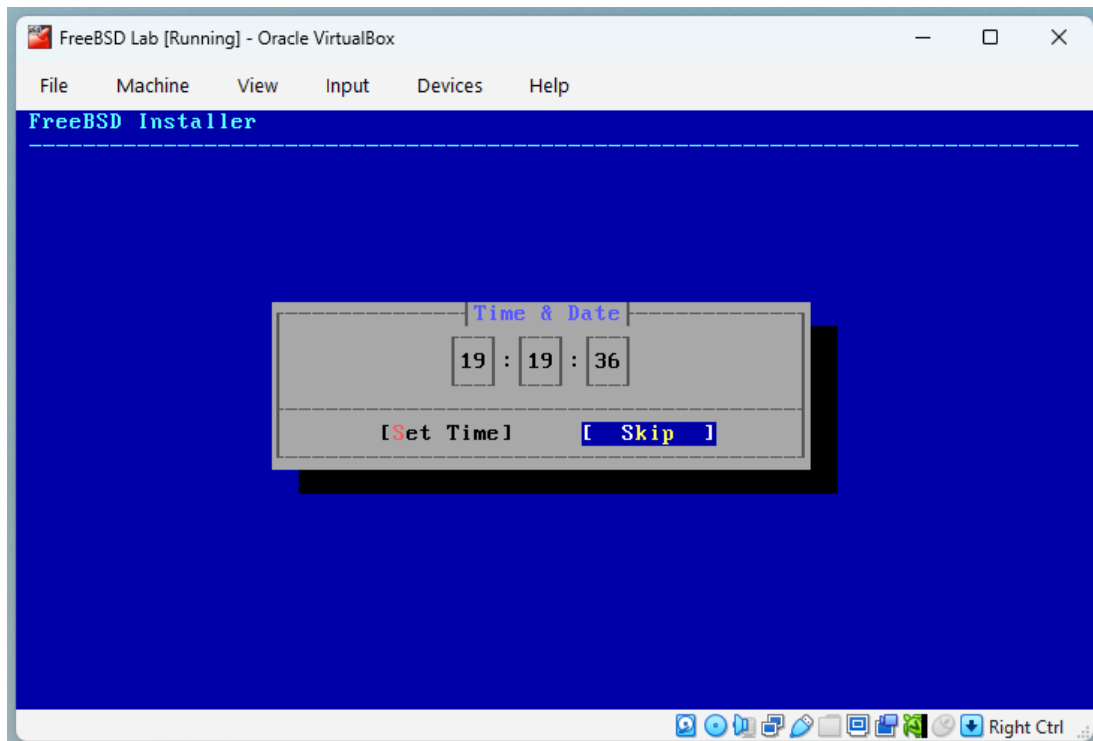


Figure 2.26: image-20250823191957558

26. **System Configuration:** The installer will give you the chance to choose some services to start at boot, select **ntpd** and press **Enter** to continue.

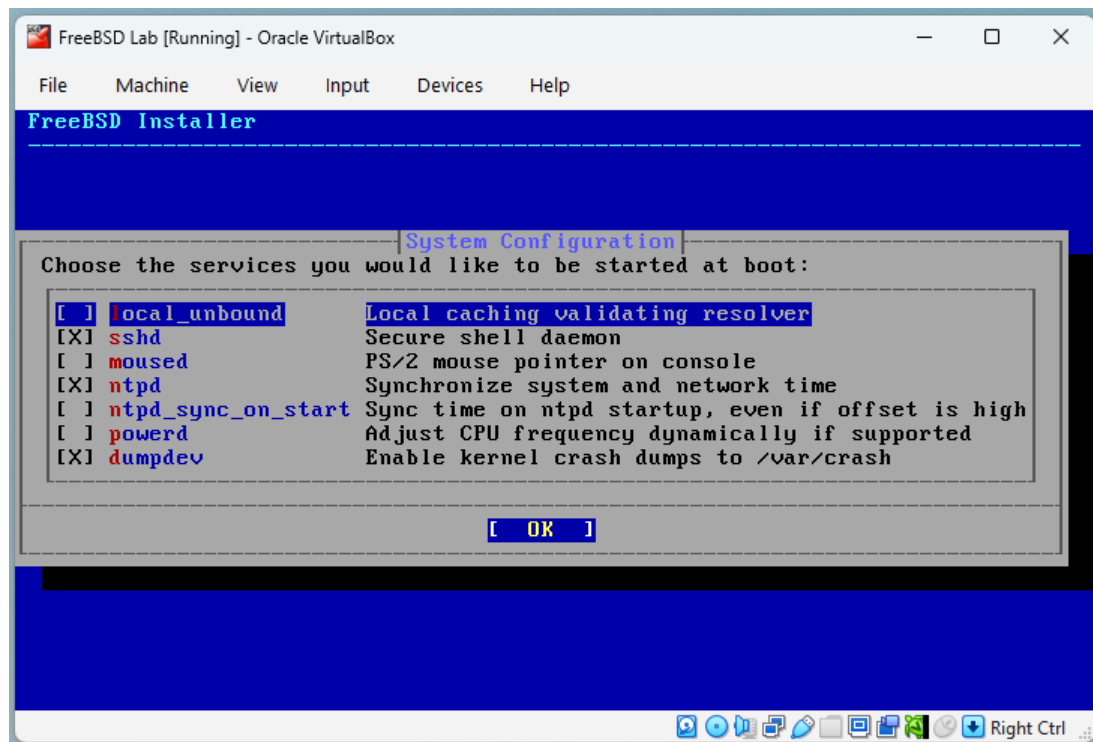


Figure 2.27: image-20250823192055299

27. **System hardening:** The installer will give you the chance to enable some security hardening to be applied at boot, accept the default for now and press **Enter** to continue.

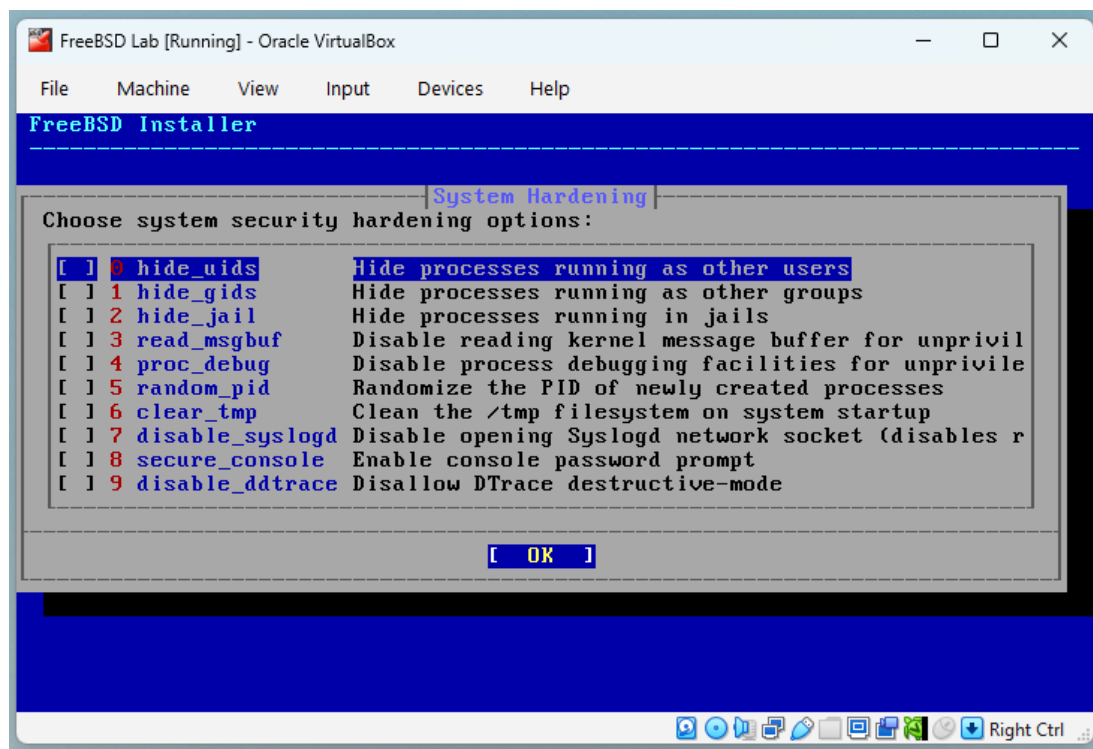


Figure 2.28: image-20250823192128039

28. **Firmware Check:** The installer will verify if your hardware components need any specific firmware to

work properly and will install it if needed. Press **Enter** to continue.

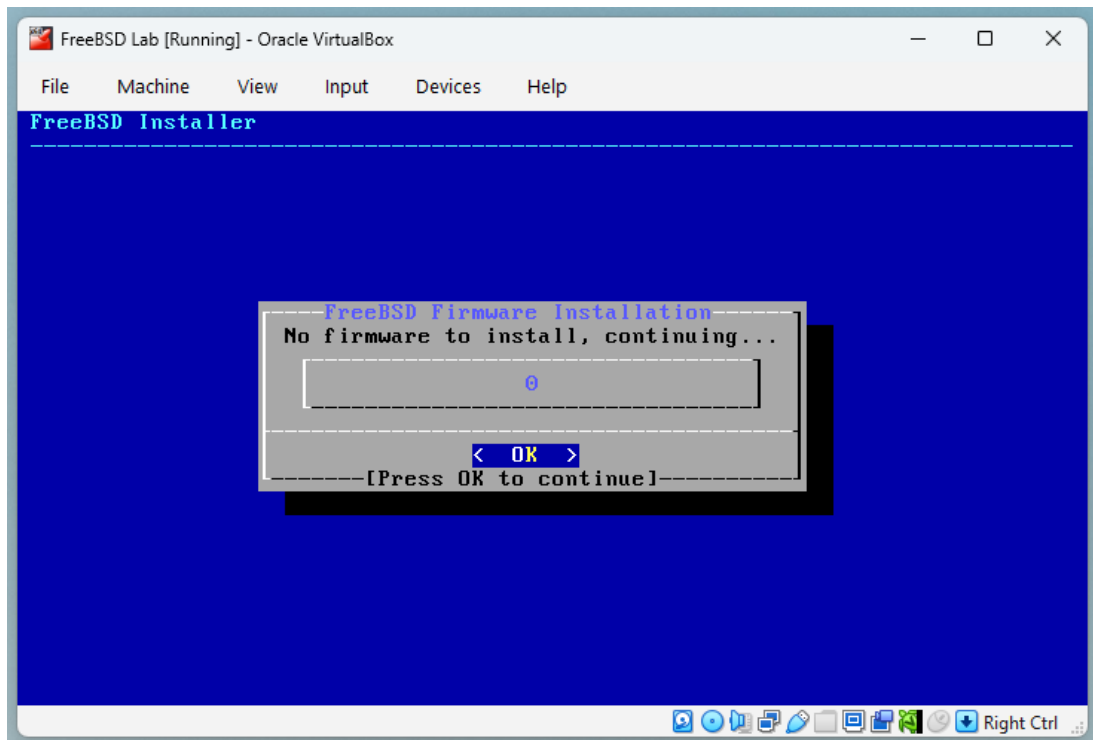


Figure 2.29: image-20250823192211024

29. **Add User Accounts:** The installer will give you the opportunity to add a normal user to your system. Choose [**Yes**] and press **Enter** to continue.

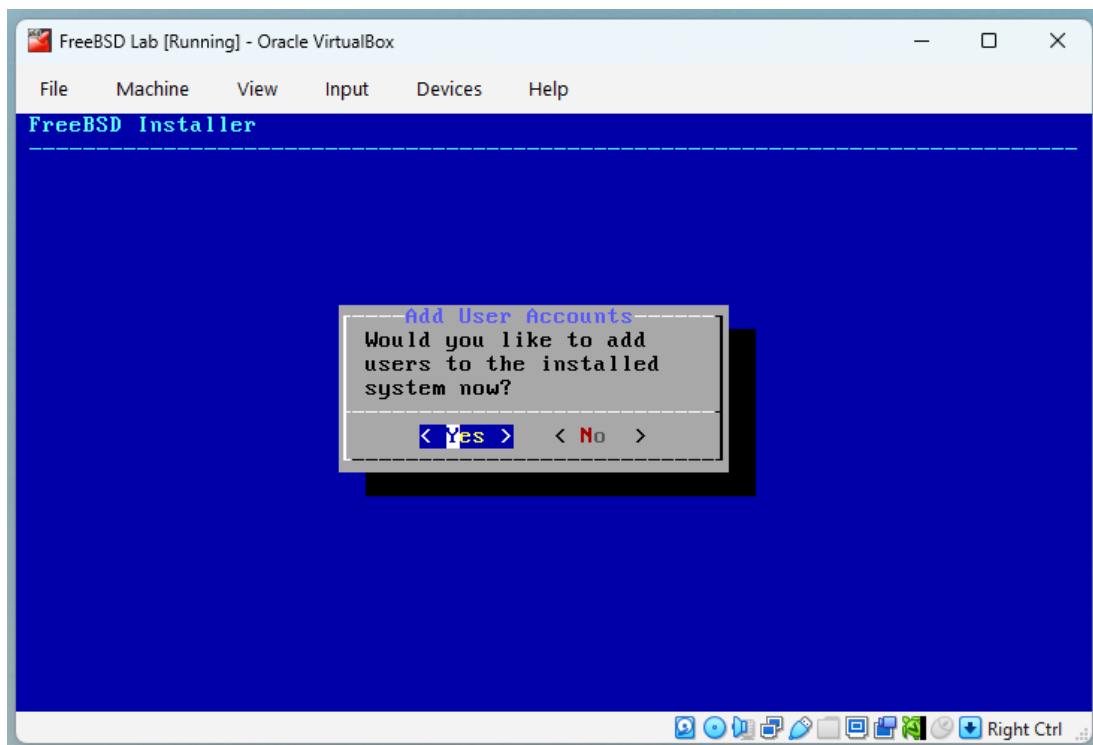


Figure 2.30: image-20250823192233281

30. **Create a User:** The installer will ask you to type the user information and to answer some basic questions, you should choose your desired **username** and **password**, you can **accept the default answers** for all questions except for the question “*Invite USER into other groups?*”, to this question you need answer “**wheel**”, this is the group in FreeBSD that will allow you to use the command `sudo` to become root during a normal session.

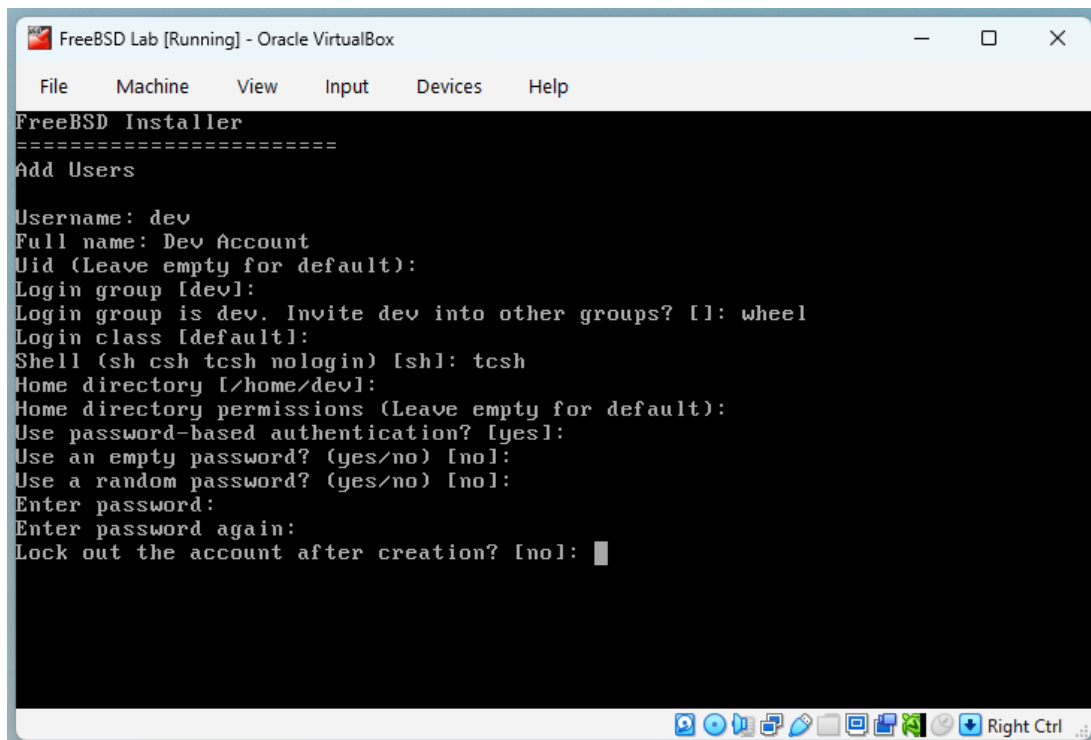


Figure 2.31: image-20250823192452683

31. **Create a User:** After you have answered all questions and have created your user, the FreeBSD installer will ask if you want to add another user. Just press **Enter** to accept the default answer (no) to go to the Final Configuration menu.

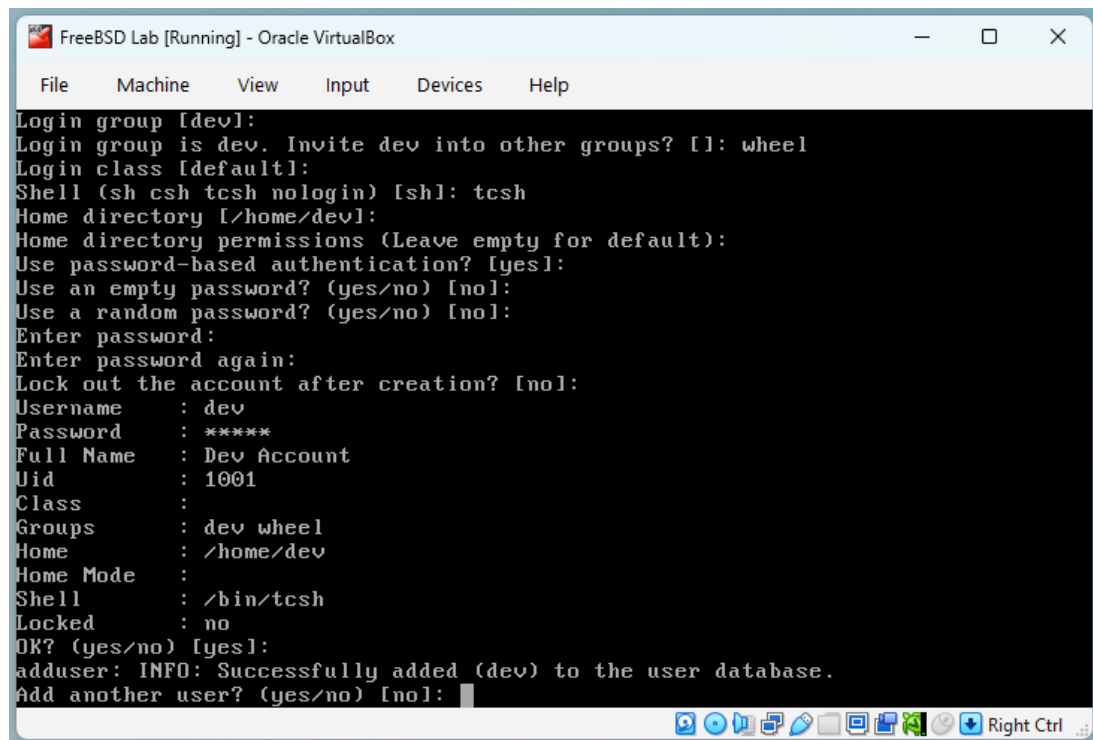


Figure 2.32: image-20250823192600794

32. **Final Configuration:** At this point you have already finished the FreeBSD installation. This final menu allow you to review and change the options that you have made in the previous steps. Select **Exit** to leave the installer and press **Enter**.

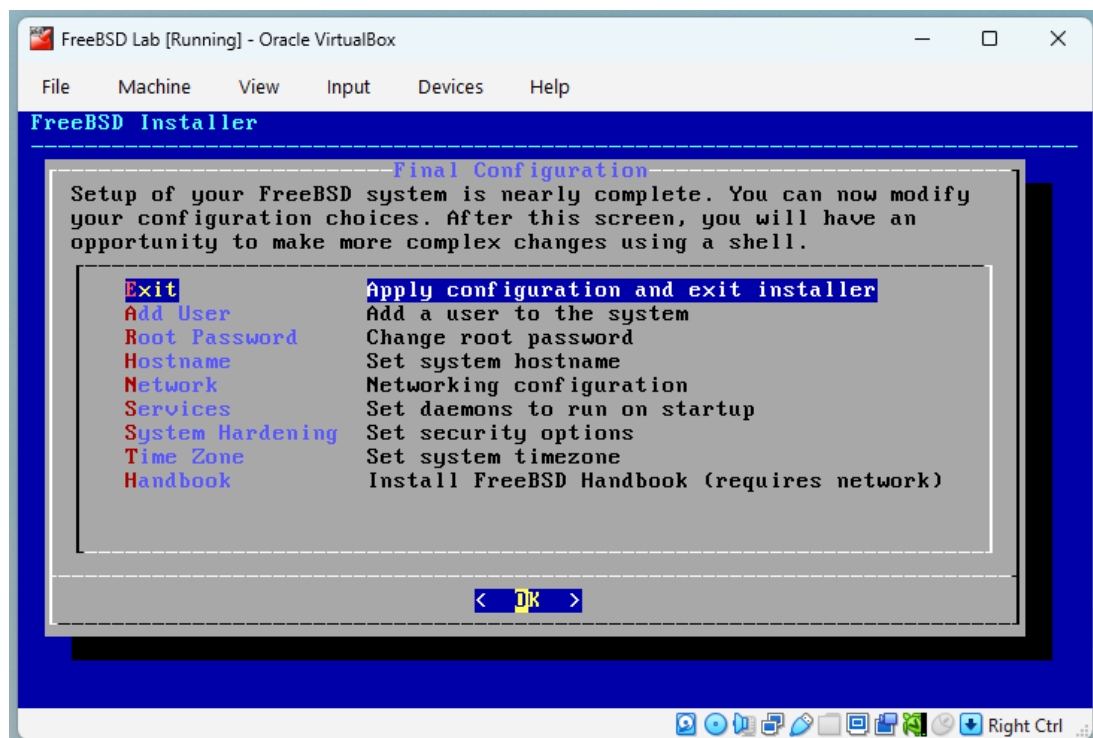


Figure 2.33: image-20250823192642433

33. **Manual Configuration:** The installer will ask you if you want to open a shell to do manual configurations in your fresh installed system. Choose [No] and press **Enter**.

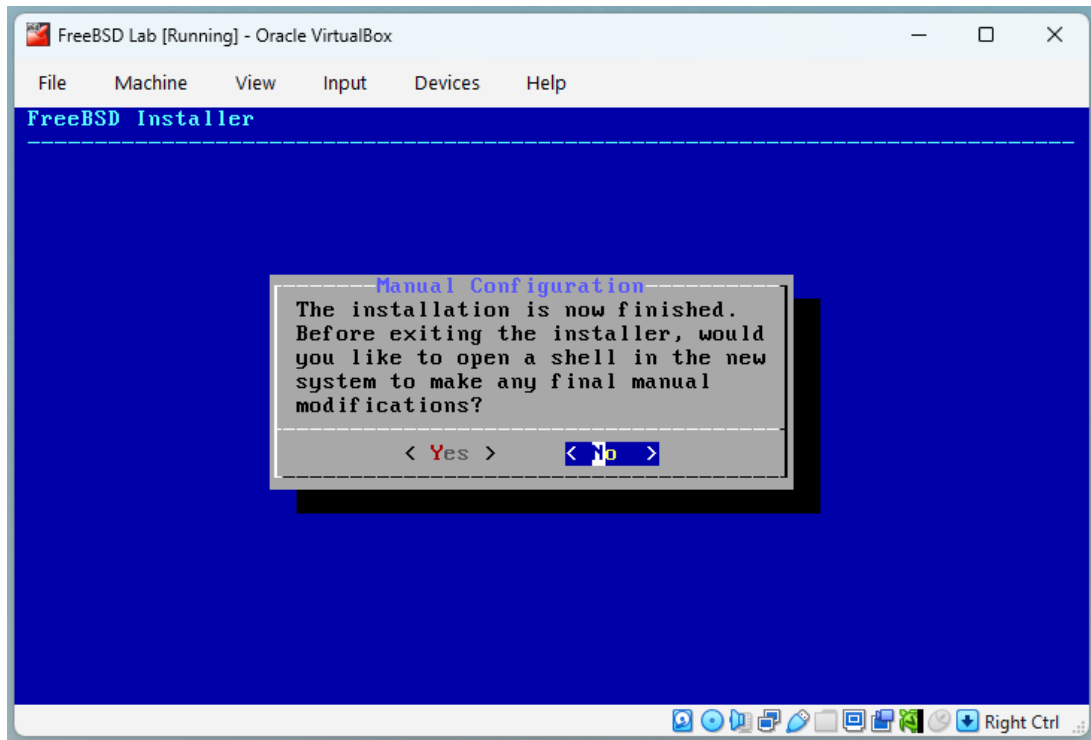


Figure 2.34: image-20250823192704460

34. **Eject the Install Disc:** Before we reboot the VM, we need to eject the virtual disc that we used for the installation. To do that, left click with your mouse in the CD/DVD icon in the lower status bar of your VirtualBox VM window, and right click in the menu “Remove Disk From Virtual Drive”.

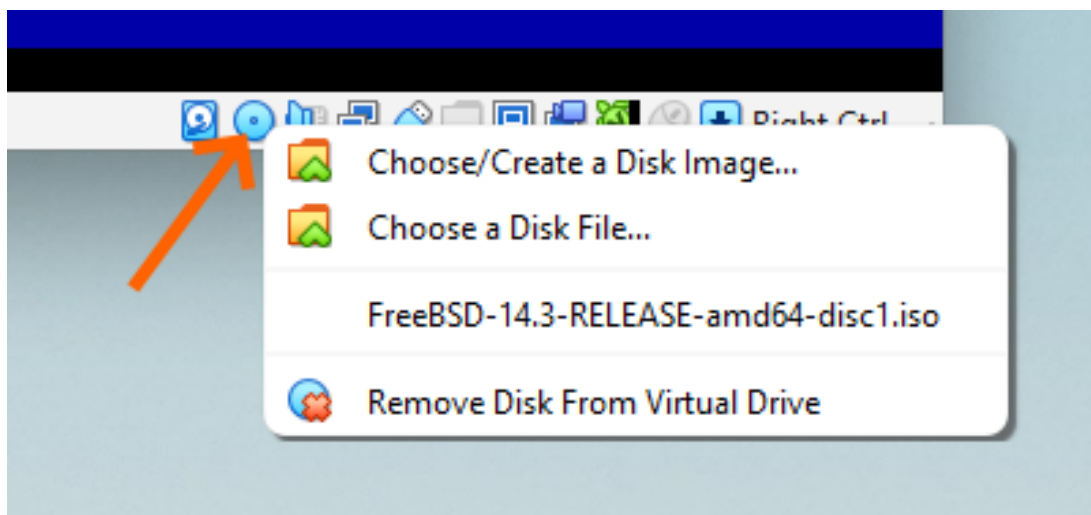


Figure 2.35: image-20250823193213602

If for some reason you receive a message telling that the virtual optical disk is in use and can't be ejected, click in the button “Force Unmount”, after that you can proceed to reboot.

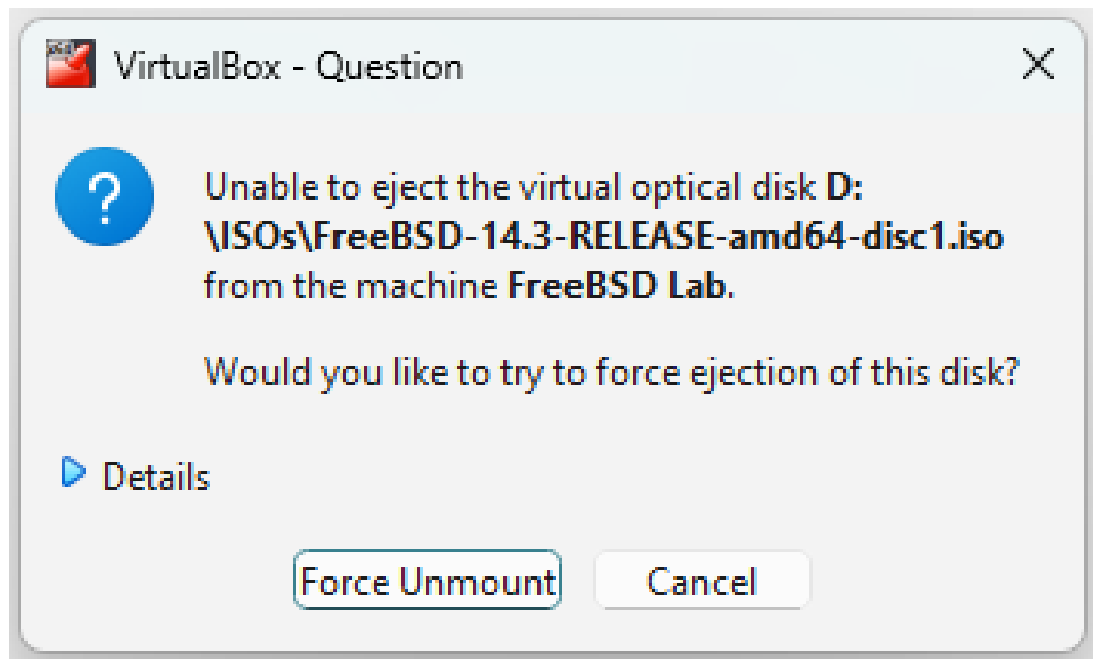


Figure 2.36: image-20250823193252804

35. **Reboot your VM:** Press **Enter** in this menu to reboot your FreeBSD VM.

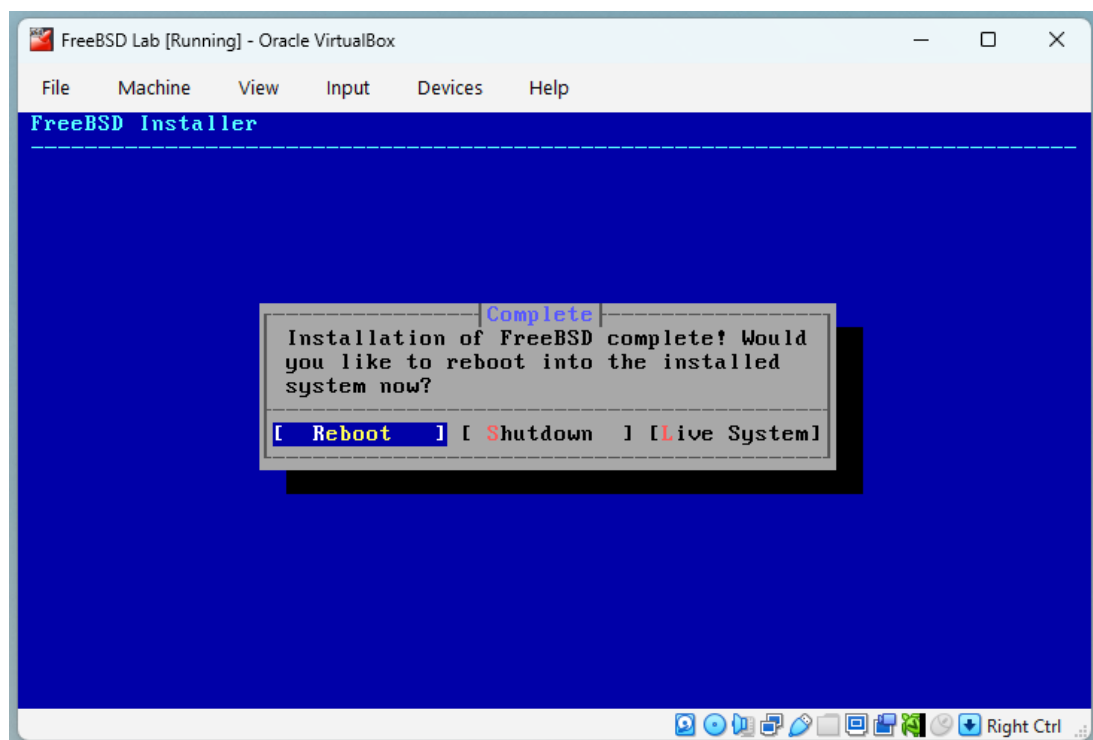


Figure 2.37: image-20250823192732830

2.3.3 Installing FreeBSD on Bare Metal

If you're using a spare PC or laptop, you'll need to install FreeBSD directly from a bootable USB flash drive. Here's how:

2.3.3.1 Step 1: Prepare a USB Stick

- You'll need a USB stick with at least **2 GB capacity**.
- Make sure to back up any data on it; the process will erase everything.

2.3.3.2 Step 2: Download the Right Image

- For USB installs, download the **memstick image** (FreeBSD-14.3-RELEASE-amd64-memstick.img).

2.3.3.3 Step 3: Create the Bootable USB (Windows Instructions)

On Windows, the easiest tool is **Rufus**:

1. Download Rufus from <https://rufus.ie>.
2. Insert your USB stick.
3. Open Rufus and select:
 - **Device**: your USB stick.
 - **Boot selection**: the FreeBSD memstick .img file you downloaded.
 - **Partition scheme**: MBR
 - **Target System**: BIOS (or UEFI-CSM)
 - **File system**: leave default.
4. Click *Start*. Rufus will warn you that all data will be destroyed, accept it.
5. Wait until the process finishes. Your USB stick is now bootable.

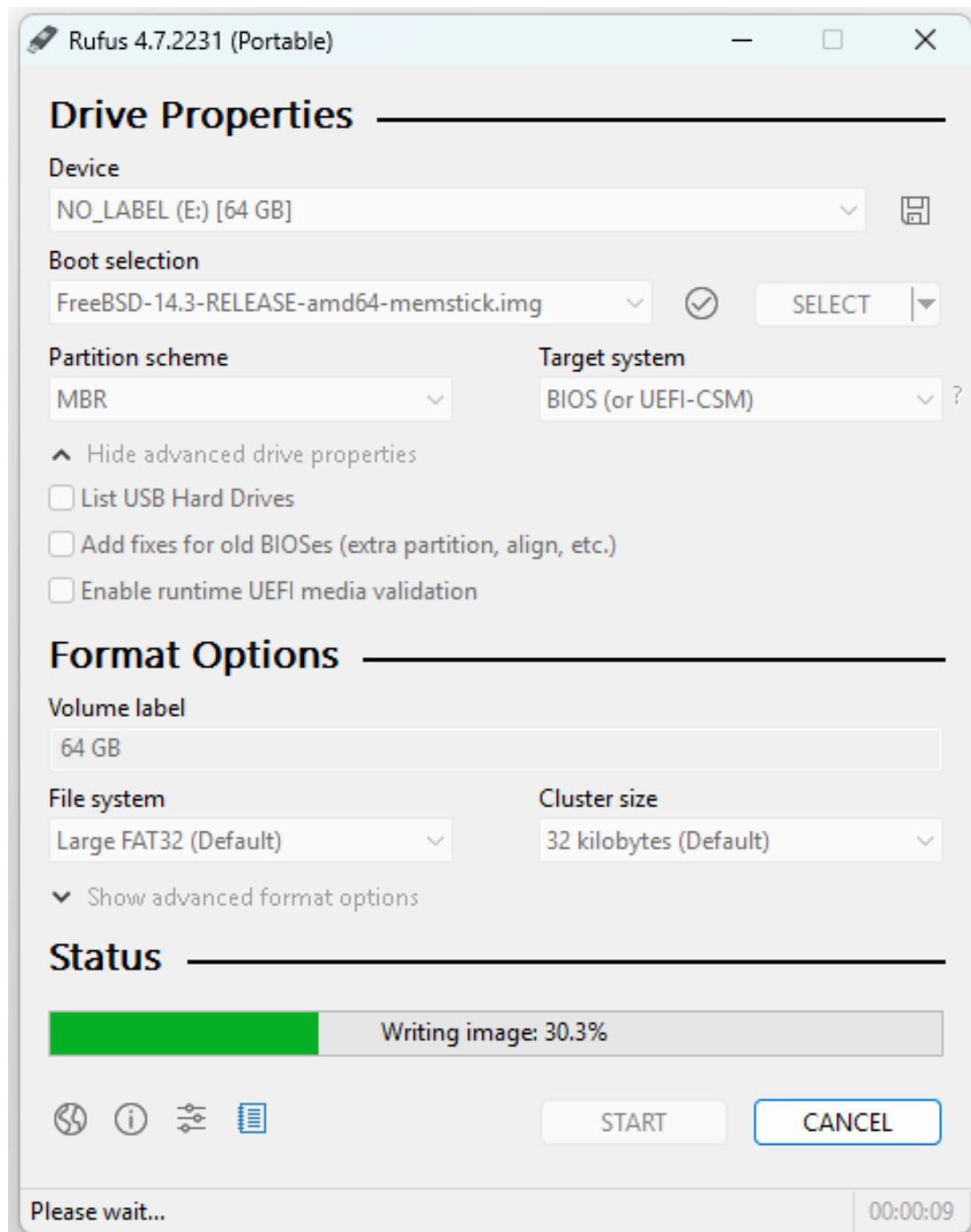


Figure 2.38: image-20250823210622431

If you already have a UNIX-like system, you can create the USB from the terminal using the `dd` command:

```
% sudo dd if=FreeBSD-14.3-RELEASE-amd64-memstick.img of=/dev/da0 bs=1M
```

Replace `/dev/da0` with your USB device path.

2.3.3.4 Step 4: Boot from USB

1. Insert the USB stick into your target machine.
2. Enter the BIOS/UEFI boot menu (usually by pressing F12, Esc, or Del during startup).
3. Select the USB drive as the boot device.

2.3.3.5 Step 5: Run the Installer

Once FreeBSD boots, follow the same installer steps described above, where we choose the Keyboard layout, Hostname, Distribution, etc.

After installation completes, remove the USB stick and reboot. FreeBSD will now boot from the hard drive.

2.3.4 First Boot

After installation, you'll see the FreeBSD boot menu:

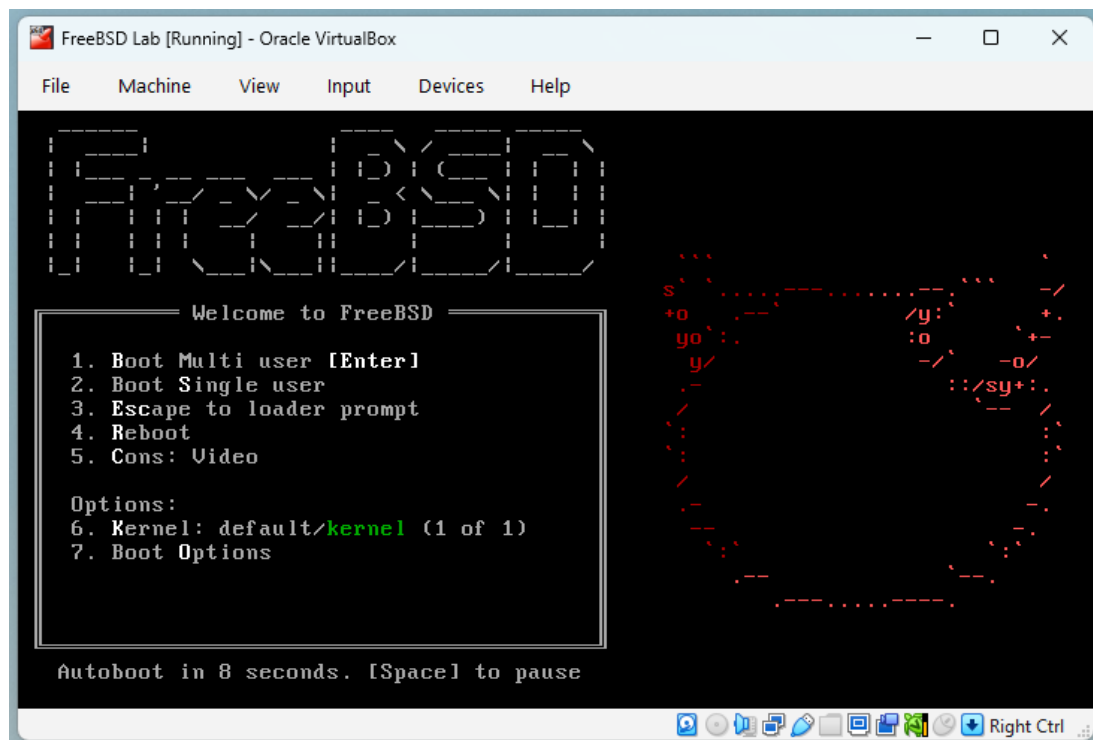


Figure 2.39: image-20250823213050882

Followed by the login prompt:

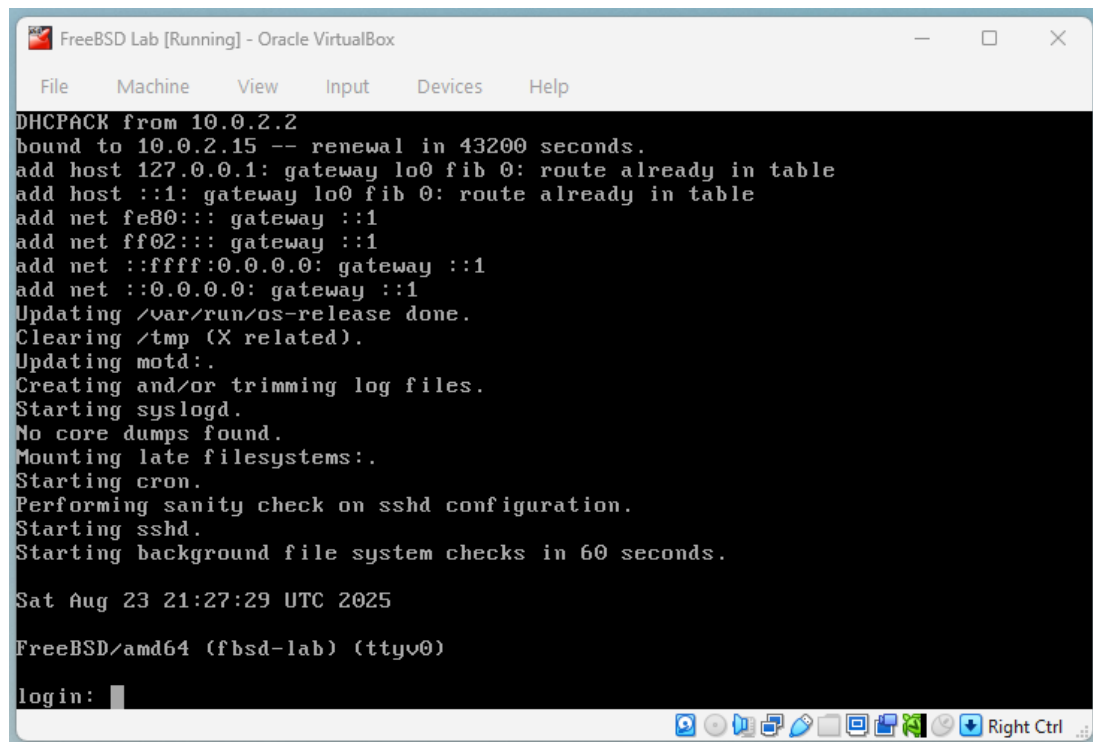


Figure 2.40: image-20250823212856938

Congratulations! Your FreeBSD lab machine is now alive and ready for configuration.

2.3.5 Wrapping Up

You’ve just completed one of the most important milestones: installing FreeBSD 14.3 on your dedicated lab environment. Whether in a VM or on bare metal, you now have a clean system that you can safely break, fix, and rebuild as you learn.

In the next section, we’ll walk through the **initial configuration** you should do right after installation: setting up networking, enabling essential services, and preparing the system for development work.

2.4 First Boot and Initial Configuration

When your FreeBSD system finishes its first reboot after installation, you’re greeted with something that feels very different from Windows or macOS. There’s no flashy desktop, no icons, no “getting started” wizard. Instead, you’re dropped at a **login prompt**.

Don’t worry, this is normal and intentional. FreeBSD is a UNIX-like system designed for stability and flexibility, not flashy first impressions. The default environment is deliberately minimal so that you, the administrator, stay in full control. Think of this as the first time you sit down at a freshly provisioned lab machine: the shell is empty, the tools are not yet installed, but the system is ready to be shaped into exactly what your work requires.

In this section, we’ll perform the **essential first steps** to make your FreeBSD lab comfortable, secure, and ready for driver development.

2.4.1 Logging In

At the login prompt:

- Enter the username you created during installation.
- Type your password (remember that UNIX systems do not display * when typing passwords).

You're now inside FreeBSD as a regular user.

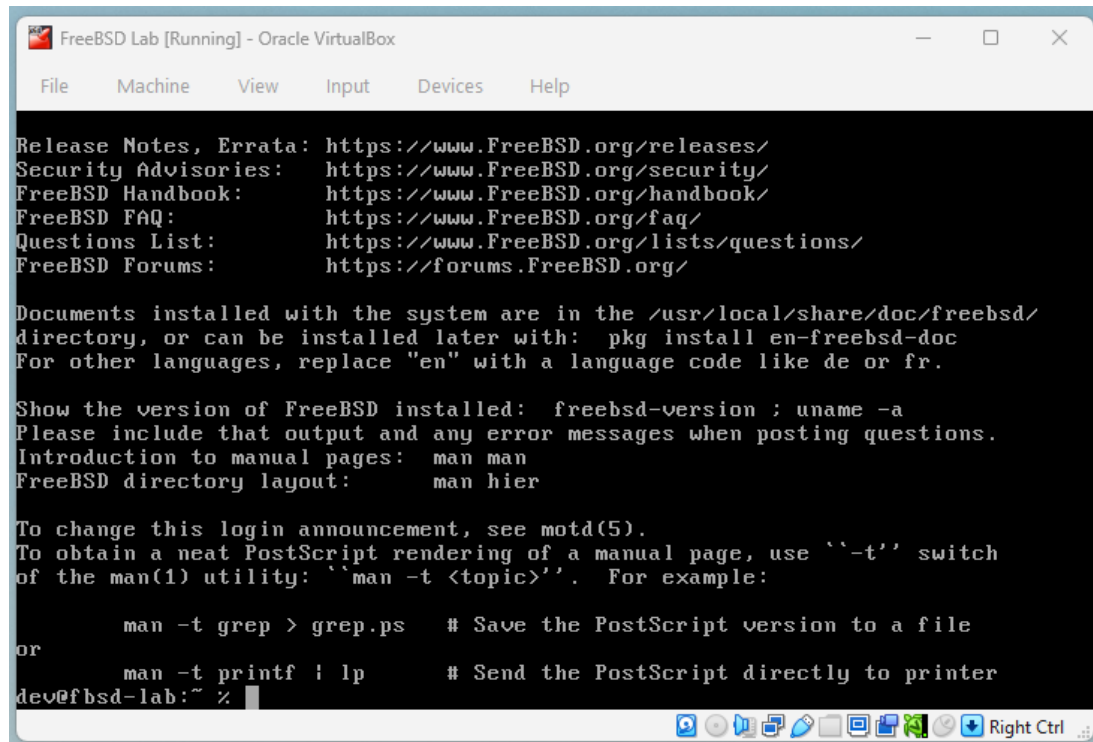


Figure 2.41: image-20250823212710535

2.4.2 Switching to the Root User

Some tasks, like installing software or editing system files, require **root privileges**. You should avoid staying logged in as root all the time (it's too risky if you mistype a command), but it's good practice to temporarily switch to root when needed:

```
% su -  
Password:
```

Enter the root password you set during installation. The prompt will change from % to #, which means you are now root.

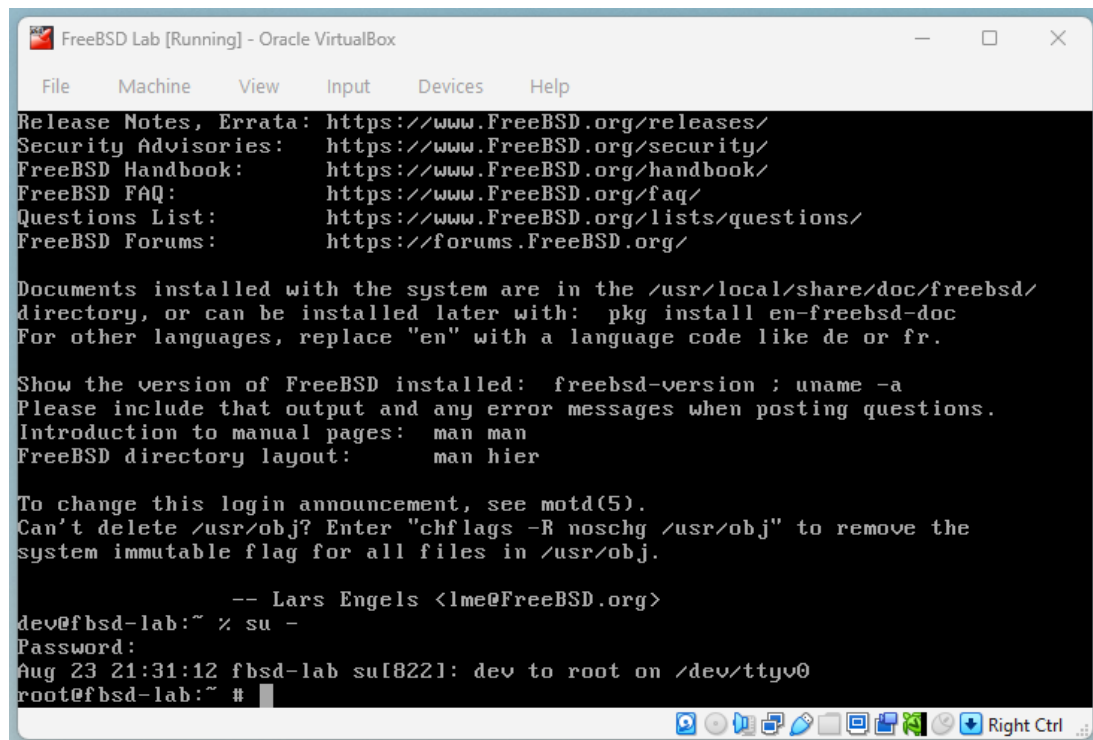


Figure 2.42: image-20250823213238499

2.4.3 Setting the Hostname and Time

Your system needs a name and correct time settings.

- To check the hostname:

```
% hostname
```

If you want to change it, edit `/etc/rc.conf`:

```
# ee /etc/rc.conf
```

Add or adjust this line:

```
hostname="fbsd-lab"
```

- To synchronise time, make sure NTP is enabled (it usually is if you selected it during installation). You can test with:

```
% date
```

If the time is wrong, correct it manually for now:

```
# date 202508231530
```

(This sets date/time to 23 Aug 2025, 15:30 - format is `YYYYMMDDhhmm`).

2.4.4 Networking Basics

Most installations with DHCP “just work.” To verify:

```
% ifconfig
```

You should see an interface (like `em0`, `re0`, or `vtnet0` in VMs) with an IP address. If you don’t, you may need to enable DHCP in `/etc/rc.conf`:

```
ifconfig_em0="DHCP"
```

Replace `em0` with your actual interface name from `ifconfig`.

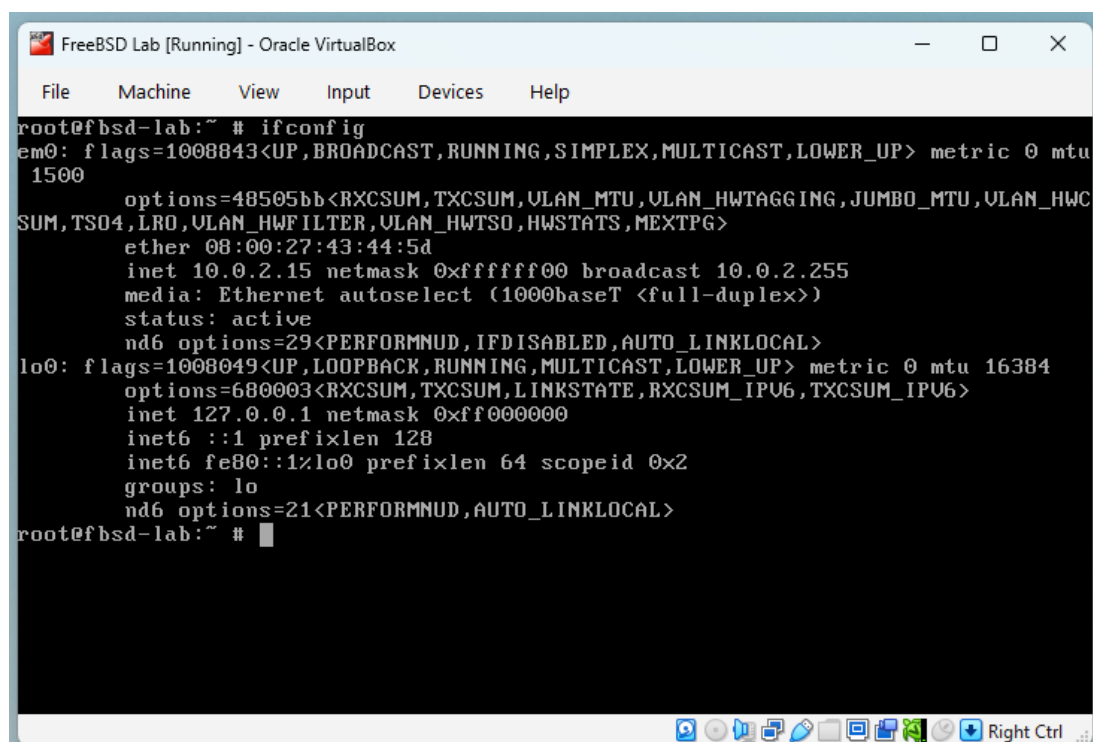


Figure 2.43: image-20250823213433266

2.4.5 Installing and Configuring sudo

As a best practice, you should use `sudo` instead of switching to `root` for every privileged command.

1. Install `sudo`:

```
# pkg install sudo
```

2. Add your user to the `wheel` group (if you haven’t done that when you created it):

```
# pw groupmod wheel -m yourusername
```

3. Now, lets enable the group wheel to use sudo.

Execute the command `visudo` and search for these lines in the file editor that will open:

```
##  
## User privilege specification  
##  
root ALL=(ALL:ALL) ALL  
  
## Uncomment to allow members of group wheel to execute any command  
# %wheel ALL=(ALL:ALL) ALL  
  
## Same thing without a password  
##%wheel ALL=(ALL:ALL) NOPASSWD: ALL
```

Delete the # from the line `##%wheel ALL=(ALL:ALL) NOPASSWD: ALL` , place the cursor in the character you want to delete using your arrow keys and press **x** , to save the file and exit the editor pressing **ESC** and than type **:wq** and pressing **Enter**.

4. to verify that it is working as expected, log out and log back in, then run:

```
% sudo whoami  
root
```

Now your user can safely perform admin tasks without staying logged in as root.

2.4.6 Updating the System

Before installing development tools, bring your system up to date:

```
# freebsd-update fetch install  
# pkg update  
# pkg upgrade
```

This ensures you're running the latest security patches.

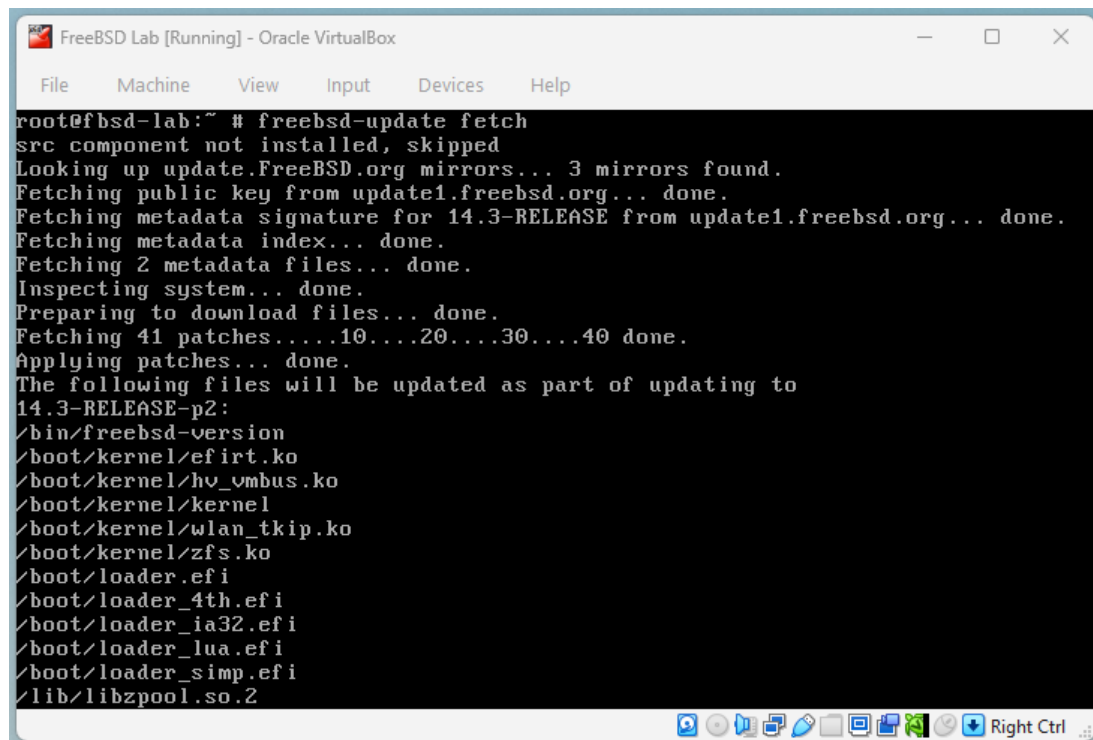


Figure 2.44: image-20250823215034288

2.4.7 Creating a Comfortable Environment

Even small tweaks make your daily work smoother:

- **Enable command history and completion** (if using `tcsh`, the default for users, this is already included).
- **Edit `.cshrc`** in your home directory to add useful aliases:

```
alias ll 'ls -lh'
alias cls 'clear'
```

- **Install a friendlier editor** (optional):

```
# pkg install nano
```

2.4.8 Basic Hardening for Your Lab

Even though this is a **lab environment**, it's important to add a few layers of safety. This is especially true if you enable **SSH**, whether you're running FreeBSD inside a VM on your laptop or on a spare physical machine. Once SSH is on, your system accepts remote logins and that means you should take some precautions.

You have two simple approaches. Pick the one you prefer; both are fine for a lab.

2.4.8.1 Option A: Minimal `pf` rules (block everything inbound except SSH)

1. Enable `pf` and create a small ruleset:


```
# sysrc pf_enable="YES"
# nano /etc/pf.conf
```

Put this in /etc/pf.conf (replace vtnet0/em0 with your interface):

```
set skip on lo

ext_if = "em0"           # VM often uses vtnet0; on bare metal you may see
                        ↪ em0/re0/igb0, etc.
tcp_services = "{ ssh }"

block in all
pass out all keep state
pass in on $ext_if proto tcp to (self) port $tcp_services keep state
```

2. Start pf (and it will persist across reboots):

```
# service pf start
```

VM note: If your VM uses NAT, you may also need to configure **port forwarding** in your hypervisor (e.g., VirtualBox: Host Port 2222 -> Guest Port 22) and then SSH to localhost -p 2222. The pf rule above still applies **inside** the guest.

2.4.8.2 Option B: Use the built-in ipfw presets (very beginner-friendly)

1. Enable ipfw with the workstation preset and open SSH:

```
# sysrc firewall_enable="YES"
# sysrc firewall_type="workstation"
# sysrc firewall_myservices="22/tcp"
# sysrc firewall_logdeny="YES"
# service ipfw start
```

- workstation provides a stateful ruleset that “protects this machine” and is easy to start with.
- firewall_myservices lists inbound services you want to allow; here we allow SSH on TCP/22.
- You can switch to other presets later (e.g., client, simple) as your needs evolve.

Tip: Choose **either** pf **or** ipfw, not both. For a first lab, the ipfw preset is the quickest path; the tiny pf ruleset is equally fine and very explicit.

2.4.8.3 Keep it patched

Run these regularly to stay current:

```
% sudo freebsd-update fetch install
% sudo pkg update && pkg upgrade
```

Why bother in a VM? Because a VM is still a real machine on your network. Good habits here prepare you for production later.

2.4.9 Wrapping Up

Your FreeBSD system is no longer a bare skeleton, it now has a hostname, working networking, an updated base system, and a user account with sudo access. You’ve also applied a small but meaningful layer of hardening: a simple firewall that still permits SSH, and regular updates. These are not just optional tweaks, they’re the kind of habits that make you a responsible systems developer.

In the next section, we’ll install the **development tools** needed for driver programming, including compilers, debuggers, editors, and the FreeBSD source tree itself. This is where your lab transforms from a blank canvas into a real development workstation.

2.5 Preparing the System for Development

Now that your FreeBSD lab is installed, updated, and lightly hardened, it’s time to turn it into a proper **driver development environment**. This step adds the pieces needed to build, debug, and version-control kernel code: the compiler, the debugger, a version control system, and the FreeBSD source tree. Without these, you won’t be able to build or test the code we’ll write in later chapters.

The good news is that FreeBSD already includes most of what we need. In this section, we’ll install the missing pieces, verify everything works, and run a tiny “hello module” test to prove that your lab is ready for driver development.

2.5.1 Installing Development Tools

FreeBSD comes with **Clang/LLVM** in the base system. To confirm:

```
% cc --version
FreeBSD clang version 19.1.7 (...)
```

If you see a version string like the above, you’re ready to compile C code.

Still, you’ll need some additional tools:

```
# pkg install git gmake gdb
```

- **git**: version control system.
- **gmake**: GNU make (some projects require it in addition to FreeBSD’s own make).
- **gdb**: the GNU debugger.

2.5.2 Choosing an Editor

Every developer has their favourite editor. FreeBSD includes **vi** by default, capable but with a steep learning curve. If you’re brand new, you can safely start with **ee (Easy Editor)**, which guides you with on-screen help, or install **nano**, which has simpler shortcuts like Ctrl+O to save and Ctrl+X to quit:

```
% sudo pkg install nano
```

But sooner or later, you'll want to learn **vim**, the improved version of **vi**. It's fast, highly configurable, and widely used in FreeBSD development. One of its big advantages is **syntax highlighting**, which makes C code much easier to read.

2.5.2.1 Configuring Vim for Syntax Highlighting

1. Install vim:

```
# pkg install vim
```

2. Create a configuration file in your home directory:

```
% ee ~/.vimrc
```

3. Add these lines:

```
syntax on
set number
set tabstop=8
set shiftwidth=8
set autoindent
set background=dark
```

- `syntax on` -> enables syntax highlighting.
- `set number` -> shows line numbers.
- The `tab/indent` settings follow **FreeBSD coding style** (8-space tabs, not 4).
- `set background=dark` -> makes colours readable on a dark terminal.

4. Save the file and open a C program:

```
% vim hello.c
```

You should now see coloured keywords, strings, and comments.

2.5.2.2 Nano Syntax Highlighting

If you prefer nano, it also supports syntax highlighting. The configuration is stored in `/usr/local/share/nano/`. To enable it for C:

```
% cp /usr/local/share/nano/c.nanorc ~/.nanorc
```

Now open a `.c` file with nano and you'll see basic highlighting.

2.5.2.3 Easy Editor (ee)

`ee` is the simplest option, no highlighting, just plain text. It's safe for beginners and great for editing configuration files quickly, but you'll probably outgrow it for driver development.

2.5.3 Accessing Documentation

The **man pages** are your built-in reference library. Try this:

```
% man 9 malloc
```

This brings up the manual page for the `malloc(9)` kernel function. The (9) section number indicates it's part of the **kernel interfaces**, which is where we'll spend most of our time later.

Other useful commands:

- `man 1 ls` -> user command documentation.
- `man 5 rc.conf` -> configuration file format.
- `man 9 intro` -> overview of kernel programming interfaces.

2.5.4 Installing the FreeBSD Source Tree

Most driver development requires access to the FreeBSD kernel source code. You'll store it in `/usr/src`.

From this point on, whenever this book cites a file such as `/usr/src/sys/kern/kern_module.c`, it refers to a real file inside the source tree you are about to clone. `/usr/src` is the conventional location for the FreeBSD source tree on a FreeBSD system, and every path of the form `/usr/src/...` in later chapters maps directly to a file under the `src` checkout below. Later chapters will not re-explain this convention; they will simply quote the path and expect you to find it at that location.

Clone it with Git:

```
% sudo git clone --branch releng/14.3 --depth 1 https://git.FreeBSD.org/src.git src
↪ /usr/src
```

This will take a few minutes and download several gigabytes. When finished, you'll have the full kernel source tree available.

Verify with:

```
% ls /usr/src/sys
```

You should see directories like `dev`, `kern`, `net`, and `vm`. These are where the FreeBSD kernel lives.

2.5.4.1 Warning: Match your headers to your running kernel.

FreeBSD is very strict about building loadable kernel modules against the exact header set that matches the kernel you are running. If your kernel was built from 14.3-RELEASE but `/usr/src` points to a different branch or version, you can hit confusing compile or load errors. To avoid trouble in the exercises presented in this book, please make sure you have the **FreeBSD 14.3** source tree installed in `/usr/src` and that it matches your running kernel. A quick check is `freebsd-version -k` which should print `14.3-RELEASE`, and your `/usr/src` should be on the `releng/14.3` branch as instructed above.

Tip: if /usr/src already exists and points elsewhere, you can retarget it:

```
% sudo git -C /usr/src fetch --all --tags
% sudo git -C /usr/src checkout releng/14.3
% sudo git -C /usr/src pull --ff-only
```

With the kernel and headers aligned, your example module will build and load reliably.

2.5.5 Testing Your Toolkit: A “Hello Kernel Module”

To confirm that everything works, let’s compile and load a tiny kernel module. This is not yet a driver, but it proves your lab can build and interact with the kernel.

1. Create a file called `hello_world.c`:

```
/*
 * hello_world.c - Simple FreeBSD kernel module
 * Prints messages when loaded and unloaded
 */

#include <sys/param.h>
#include <sys/kernel.h>
#include <sys/module.h>
#include <sys/systm.h>

/*
 * Load handler - called when the module is loaded
 */
static int
hello_world_load(module_t mod, int cmd, void *arg)
{
    int error = 0;

    switch (cmd) {
    case MOD_LOAD:
        printf("Hello World! Kernel module loaded.\n");
        break;
    case MOD_UNLOAD:
        printf("Goodbye World! Kernel module unloaded.\n");
        break;
    default:
        error = EOPNOTSUPP;
        break;
    }
}
```

```
    return (error);
}

/*
 * Module declaration
 */
static moduledata_t hello_world_mod = {
    "hello_world",    /* module name */
    hello_world_load, /* event handler */
    NULL              /* extra data */
};

/*
 * Register the module with the kernel
 * DECLARE_MODULE(name, data, sub-system, order)
 */
DECLARE_MODULE(hello_world, hello_world_mod, SI_SUB_DRIVERS, SI_ORDER_MIDDLE);
MODULE_VERSION(hello_world, 1);
```

1. Create a Makefile:

```
# Makefile for hello_world kernel module

KMOD=    hello_world
SRCS=    hello_world.c

.include <bsd.kmod.mk>
```

1. Build the module:

```
# make
```

This should create a file `hello.ko`.

1. Load the module:

```
# kldload ./hello_world.ko
```

Check the message in the system log:

```
% dmesg | tail -n 5
```

You should see:

```
Hello World! Kernel module loaded..
```

1. Unload the module:

```
# kldunload hello_world.ko
```

Check again:

```
% dmesg | tail -n 5
```

You should see:

```
Goodbye World! Kernel module unloaded.
```

2.5.6 Hands-On Lab: Verifying Your Development Setup

1. Install `git`, `gmake`, and `gdb`.
2. Verify Clang is working with `% cc --version`.
3. Install and configure `vim` with syntax highlighting, or set up `nano` if you prefer.
4. Clone the FreeBSD 14.3 source tree to `/usr/src`.
5. Write, compile, and load the `hello_world` kernel module.
6. Record the results (did you see the “Hello, kernel world!” message?) in your **lab logbook**.

2.5.7 Wrapping Up

You’ve now stocked your FreeBSD workshop with the essential tools: compiler, debugger, version control, documentation, and the kernel source itself. You’ve even built and loaded your very first kernel module, proving your setup works end-to-end.

In the next section, we’ll look at **using snapshots and backups** so that you can experiment freely without fear of losing your progress. This will give you the confidence to take bigger risks and recover quickly when things break.

2.6 Using Snapshots and Backups

One of the biggest advantages of setting up a **lab environment** is that you can experiment without fear. When you’re writing kernel code, mistakes are inevitable: a wrong pointer, an infinite loop, or a bad unload routine can crash the entire operating system. Instead of worrying, you can treat crashes as part of the learning process, *if* you have a way to recover quickly.

That’s where **snapshots and backups** come in. Snapshots let you “freeze” your FreeBSD lab at a safe point, then roll back instantly if something goes wrong. Backups protect your important files, like your code or lab notes, in case you need to reinstall the system.

In this section, we’ll explore both.

2.6.1 Snapshots in Virtual Machines

If you're running FreeBSD in a VM (VirtualBox, VMware, bhyve), you have a huge safety net: **snapshots**.

- In **VirtualBox** or **VMware**, snapshots are managed from the GUI, you can save, restore, and delete them with a few clicks.
- In **bhyve**, snapshots are managed through the **storage backend**, usually ZFS. You snapshot the dataset that holds the VM's disk image and roll it back when needed.

2.6.1.1 Example workflow for VirtualBox

1. Shut down your FreeBSD VM after finishing initial setup.
2. In the VirtualBox Manager, select your VM -> **Snapshots** -> click **Take**.
3. Name it: Clean FreeBSD 14.3 Install.

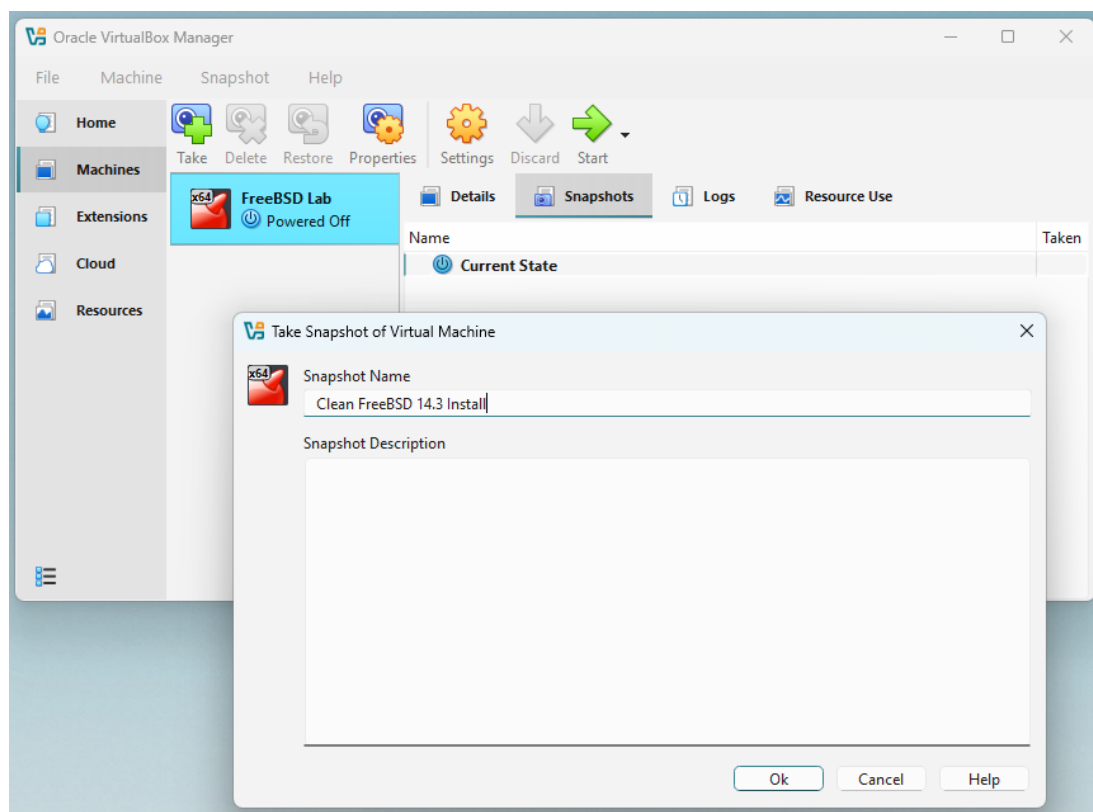


Figure 2.45: image-20250823231838089

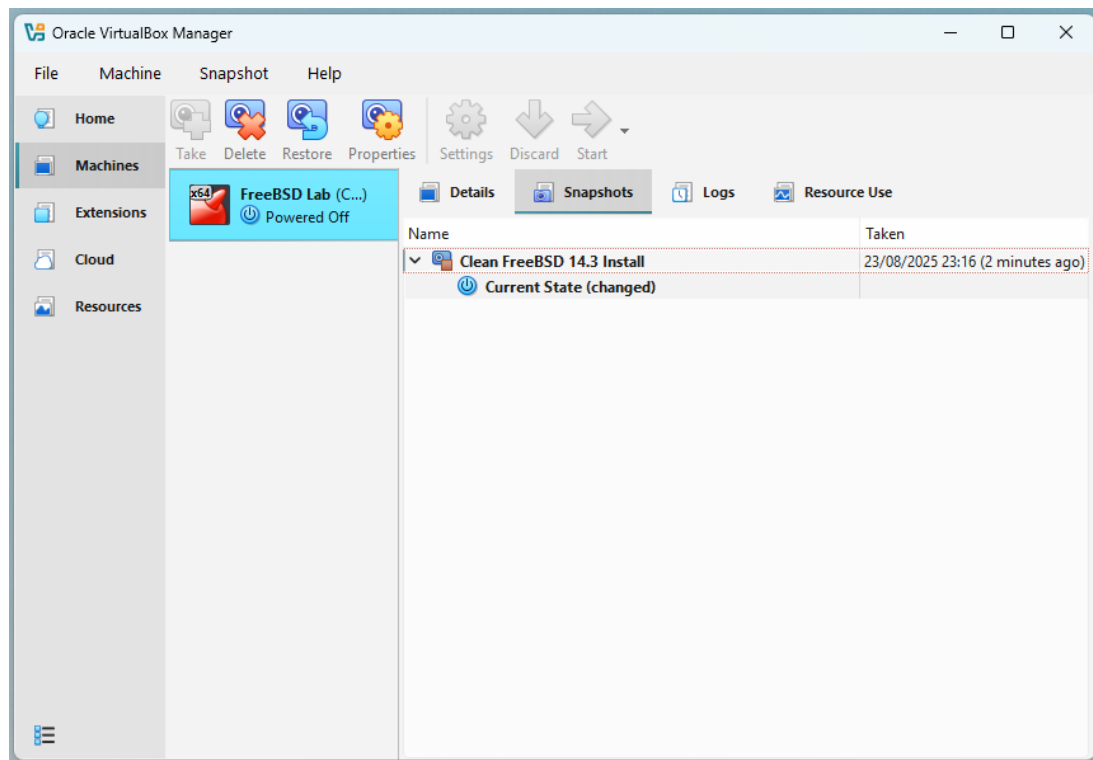


Figure 2.46: image-20250823231940246

- Later, before testing risky kernel code, take another snapshot: Before Hello Driver.

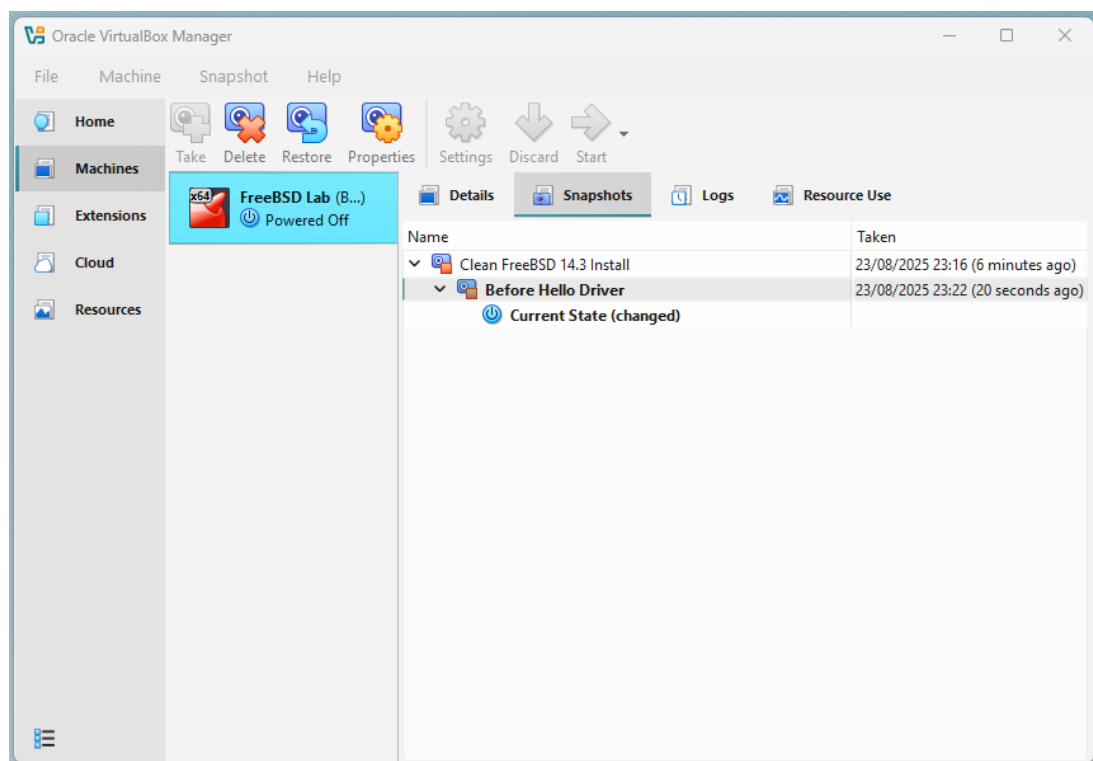


Figure 2.47: image-20250823232320392

- If the system crashes or you break networking, just restore the snapshot.

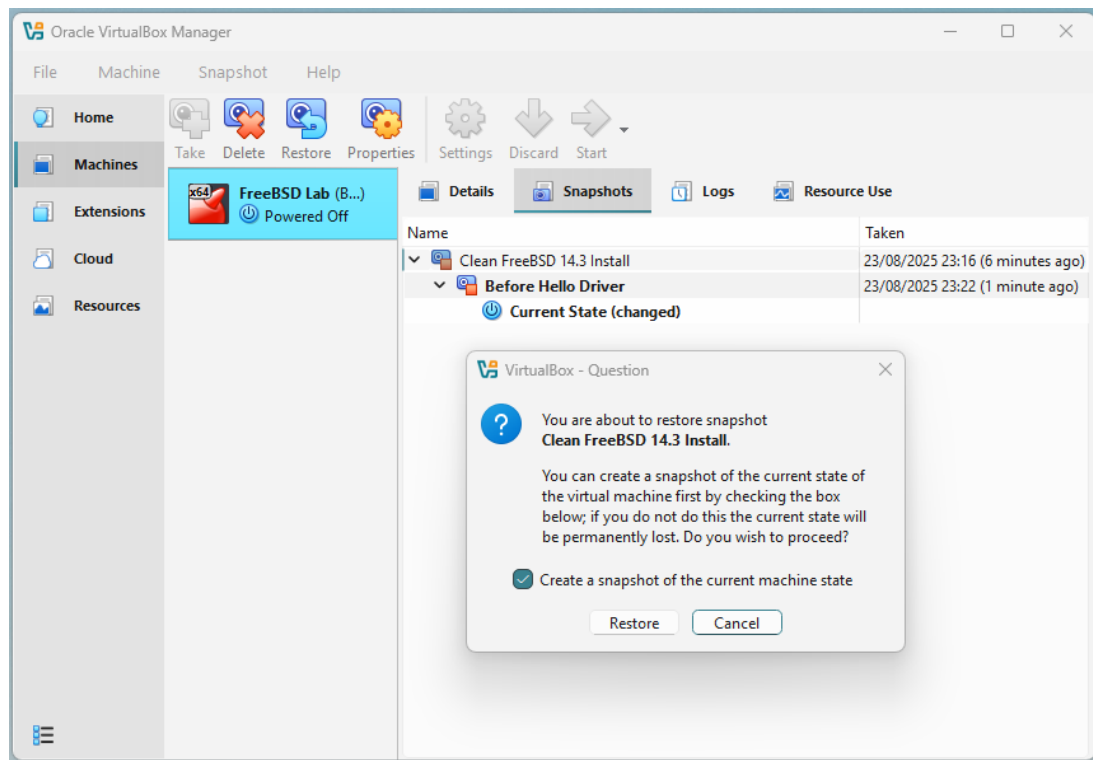


Figure 2.48: image-20250823232420760

2.6.1.2 Example Workflow in bhyve (with ZFS)

If your VM disk is stored on a ZFS dataset, for example `/zroot/vm/freebsd.img`:

1. Create a snapshot before experiments:

```
# zfs snapshot zroot/vm@clean-install
```

2. Make changes, test code, or even crash the kernel.
3. Roll back instantly:

```
# zfs rollback zroot/vm@clean-install
```

2.6.2 Snapshots on Bare Metal

If you're running FreeBSD directly on hardware, you don't have the luxury of GUI snapshots. But if you installed FreeBSD with **ZFS**, you still have access to the same snapshot tools.

With ZFS:

```
# zfs snapshot -r zroot@clean-install
```

- This creates a snapshot of your root filesystem.
- If something goes wrong, you can roll back:

```
# zfs rollback -r zroot@clean-install
```

ZFS snapshots are instantaneous and don't duplicate data, they only track changes. For serious bare-metal labs, ZFS is highly recommended.

If you installed with **UFS** instead of ZFS, you won't have snapshots. In that case, rely on **backups** (see below) and perhaps consider reinstalling with ZFS later if you want this safety net.

2.6.3 Backing Up Your Work

Snapshots protect the **system state**, but you also need to protect your **work**, your driver code, notes, and Git repositories.

Simple strategies:

- **Git:** If you're using Git (and you should), push your code to a remote service like GitHub or GitLab. This is the best backup.
- **Tarballs:** Create an archive of your project:

```
% tar czf mydriver-backup.tar.gz mydriver/
```
- **Copy to host:** If using a VM, copy files from guest to host (VirtualBox shared folders, or scp over SSH).

Note: Think of your VM as disposable, but your **code is precious**. Always back it up before testing dangerous changes.

2.6.4 Hands-On Lab: Breaking and Fixing

1. If you're in VirtualBox/VMware:

- Create a snapshot named `Before Break`.
- As root, run something harmless but destructive (e.g., delete `/tmp/*`).
- Restore the snapshot and confirm `/tmp` is back to normal.

2. If you're in bhyve with ZFS-backed storage:

- Snapshot your VM dataset.
- Delete a test file inside the guest.
- Roll back the ZFS snapshot.

3. If you're on bare metal with ZFS:

- Take a snapshot `zroot@before-break`.
- Delete a test file.
- Roll back with `zfs rollback` and confirm the file is restored.

4. Back up your `hello_world` kernel module source with:

```
% tar czf hello-backup.tar.gz hello_world/
```

Record in your **lab logbook**: which method you used, how long it took, and how confident you feel now about experimenting.

2.6.5 Wrapping Up

By learning to use **snapshots and backups**, you’ve added one of the most important safety nets to your lab. Now you can crash, break, or misconfigure FreeBSD and recover in minutes. This freedom is what makes a lab so valuable, it lets you focus on learning, not on being afraid of mistakes.

In the next section, we’ll set up **version control with Git** so you can track your progress, manage your experiments, and share your drivers.

2.7 Setting Up Version Control

Up to now, you’ve prepared your FreeBSD lab, installed the tools, and even built your first kernel module. But imagine this: you make a change to your driver, test it, and suddenly nothing works. You wish you could go back to the last working version. Or maybe you want to keep two different experiments without mixing them up.

This is exactly why developers use **version control systems**, tools that record the history of your work, allow you to roll back to previous states, and make it easy to share code with others. In the FreeBSD world (and in most open-source projects), the standard is **Git**.

In this section, you’ll learn how to use Git to manage your drivers from day one.

2.7.1 Why Version Control Matters

- **Track your changes:** Every experiment, every fix, every mistake is saved.
- **Undo safely:** If your code stops working, you can roll back to a known good version.
- **Organize experiments:** You can work on new ideas in “branches” without breaking your main code.
- **Share your work:** If you want feedback from others or publish your drivers, Git makes it easy.
- **Professional habit:** Every serious software project (including FreeBSD itself) uses version control.

Think of Git as the **lab notebook for your code**, only smarter: it not only records what you did, but can also restore your code to any past point in time.

2.7.2 Installing Git

If you haven’t already installed Git in section 2.5, do it now:

```
# pkg install git
```

Check the version:

```
% git --version
git version 2.45.2
```

2.7.3 Configuring Git (Your Identity)

Before using Git, configure your identity so your commits are labeled correctly:

```
% git config --global user.name "Your Name"
% git config --global user.email "you@example.com"
```

This doesn't need to be your real name or email if you're just experimenting locally, but if you ever share code publicly, it helps to use something consistent.

You can check your settings with:

```
% git config --list
```

2.7.4 Creating Your First Repository

Let's put your `hello_world` kernel module under version control.

1. Navigate to the directory where you created `hello_world.c` and the `Makefile`.
2. Initialize a Git repository:

```
% git init
```

This creates a hidden `.git` directory where Git stores its history.

3. Add your files:

```
% git add hello_world.c Makefile
```

4. Make your first commit:

```
% git commit -m "Initial commit: hello_world kernel module"
```

5. Check the history:

```
% git log
```

You should see your commit listed.

2.7.5 Best Practices for Commits

- **Write clear commit messages:** Describe what changed and why.
 - Bad: `fix stuff`
 - Good: `Fix null pointer dereference in hello_loader()`
- **Commit often:** Small commits are easier to understand and roll back.
- **Keep experiments separate:** If you try a new idea, make a branch:

```
% git checkout -b experiment-null-fix
```

Even if you never share your code, these habits will help you debug and learn faster.

2.7.6 Using Remote Repositories (Optional)

For now, you can keep everything local. But if you want to sync your code between machines or share it publicly, you can push it to a remote service like **GitHub** or **GitLab**.

Basic workflow:

```
% git remote add origin git@github.com:yourname/mydriver.git
% git push -u origin main
```

This is optional in the lab, but very useful if you want to back up your work in the cloud.

2.7.7 Hands-On Lab: Version Control for Your Driver

1. Initialize a Git repository in your `hello` module directory.
2. Make your first commit.
3. Edit `hello_world.c` (e.g., change the message text).
4. Run:

```
% git diff
```

to see exactly what changed.

5. Commit the change with a clear message.
6. Record in your **lab logbook**:
 - How many commits you made.
 - What each commit did.
 - How you would roll back if something broke.

2.7.8 Wrapping Up

You've now taken the first steps with Git, one of the most important tools in your developer toolkit. From now on, every driver you write in this book should live in its own Git repository. This way, you'll never lose your progress, and you'll always have a record of your experiments.

In the next section, we'll discuss **documenting your work** another key habit of professional developers. A well-written README or commit message can be the difference between code you understand a year later and code you have to rewrite from scratch.

2.8 Documenting Your Work

Software development isn't just about writing code it's also about making sure that *you* (and sometimes others) can understand that code later. When working on FreeBSD drivers, you'll often come back to a project weeks or months later and ask yourself: *"Why did I write this? What was I testing? What did I change?"*

Without documentation, you'll waste hours rediscovering your own thought process. With good notes, you can pick up exactly where you left off.

Think of documentation as the **memory of your lab**. Just like scientists keep detailed lab notebooks, developers should keep clear notes, READMEs, and commit messages.

2.8.1 Why Documentation Matters

- **Future you will thank you:** The details that seem obvious today will be forgotten in a month.
- **Debugging becomes easier:** When something breaks, notes help you understand what changed.
- **Sharing is smoother:** If you publish your driver, others can learn from your README.
- **Professional habit:** FreeBSD itself is famous for its high-quality documentation, following this tradition makes your work fit naturally into the ecosystem.

2.8.2 Writing a Simple README

Every project should start with a `README.md` file. At a minimum, include:

1. **Project name:**

```
Hello Kernel Module
```

2. **Description:**

```
A simple "Hello, kernel world!" module for FreeBSD 14.3.
```

3. **How to build:**

```
% make
```

4. **How to load/unload:**

```
# kldload ./hello_world.ko
# kldunload hello_world
```

5. **Notes:**

```
This was created as part of my driver development lab, Chapter 2.
```

2.8.3 Using Commit Messages as Documentation

Git commit messages are a form of documentation. Together, they tell the story of your project. Follow these tips:

- Write commit messages in the present tense (“Add feature”, not “Added feature”).
- Make the first line short (50 characters or less).
- If needed, add a blank line and then a longer explanation.

Example:

Fix panic when unloading hello module

The handler did not check for NULL before freeing resources, causing a panic when unloading. Added a guard condition.

2.8.4 Keeping a Lab Logbook

In Section 2.1, we suggested starting a lab logbook. Now is a good time to make it a habit. Keep a text file (for example, LABLOG.md) in the root of your Git repository. Each time you try something new, add a short entry:

2025-08-23

- Built hello module successfully.
- Confirmed "Hello, kernel world!" appears in dmesg.
- Tried unloading/reloading multiple times, no errors.
- Next step: experiment with passing parameters to the module.

This log doesn't need to be polished, it's just for you. Later, when debugging, these notes can be invaluable.

2.8.5 Tools to Help

- **Markdown:** Both README and lab logs can be written in Markdown (.md), which is easy to read in plain text and nicely formatted on GitHub/GitLab.
- **man pages:** Always note which man pages you used (e.g., `man 9 module`). This will remind you of your sources.
- **Screenshots/Logs:** If you're using a VM, take screenshots of important steps, or save command outputs into files with redirection (`dmesg > dmesg.log`).

2.8.6 Hands-On Lab: Documenting Your First Module

1. In your `hello_world` module directory, create a `README.md` describing what it does, how to build it, and how to load/unload it.
2. Add your `README.md` to Git and commit it:

```
% git add README.md
% git commit -m "Add README for hello_world module"
```

3. Start a `LABLOG.md` file and record today's activities.
4. Review your Git history with:

```
% git log --oneline
```

to see how your commits tell the story of your project.

2.8.7 Wrapping Up

You've now learned how to document your FreeBSD driver experiments so that you never lose track of what you did or why. With a **README**, meaningful commit messages, and a lab logbook, you're building habits that will make you a more professional and efficient developer.

In the next section, we'll wrap up this chapter, reviewing everything you've built: a safe FreeBSD lab with the right tools, backups, version control, and documentation, all ready for the deeper exploration of FreeBSD itself in Chapter 3.

2.9 Wrapping Up

Congratulations! You've built your FreeBSD lab!

In this chapter, you:

- Understood why a **safe lab environment** is critical for driver development.
- Chose the right setup for your situation, a **virtual machine** or **bare metal**.
- Installed **FreeBSD 14.3** step by step.
- Performed the **initial configuration**, including networking, users, and basic hardening.
- Installed the essential **development tools**: compiler, debugger, Git, and editors.
- Set up **syntax highlighting** in your editor, making C code easier to read.
- Cloned the **FreeBSD 14.3 source tree** to `/usr/src`.
- Compiled and tested your very first **kernel module**.
- Learned how to use **snapshots and backups** to recover quickly from mistakes.
- Started using **Git** for version control and added a **README** and **lab logbook** to document your work.

That's an impressive amount of progress for one chapter. You now have a complete workshop: a FreeBSD system where you can write, build, test, break, and recover as much as you need.

The most important takeaway is not just the tools you installed, but the **mindset**:

- Expect mistakes.
- Record your process.
- Use snapshots, backups, and Git to recover and learn.

2.9.1 Exercises

1. Snapshots

- Take a snapshot of your VM or a ZFS snapshot on bare metal.
- Deliberately make a change (e.g., remove `/tmp/testfile`).
- Roll back and verify the system is restored.

2. Version Control

- Make a small edit to your `hello_world.c` kernel module.
- Commit the change with Git.
- Use `git log` and `git diff` to review your history.

3. Documentation

- Add a new entry to your `LABLOG.md` describing today's work.
- Update your `README.md` with one new note (for example, mention the `uname -a` output).

4. Reflection

- In your lab logbook, answer: *What are the three most important safety nets I set up in Chapter 2?*

2.9.2 Looking Ahead

In the next chapter, we'll step inside your new FreeBSD lab and explore how to **use the system itself**. You'll learn the basics of UNIX commands, navigation, and file management. These skills will make you comfortable living inside FreeBSD, preparing you for the more advanced topics to come.

Your lab is ready. Now it's time to learn how to work in it.

3 A Gentle Introduction to UNIX

Now that your FreeBSD system is installed and running, it's time to get comfortable living inside it. FreeBSD isn't just an operating system, it's part of a long tradition that started with UNIX more than fifty years ago.

In this chapter, we'll take our first real tour of the system. You'll learn how to navigate the filesystem, run commands in the shell, manage processes, and install applications. Along the way, you'll see how FreeBSD inherits the UNIX philosophy of simplicity and consistency, and why that matters for us as future driver developers.

Think of this chapter as your **survival guide** for working inside FreeBSD. Before we start diving into C code and kernel internals, you'll need to be comfortable moving around the system, manipulating files, and using the tools that every developer relies on daily.

By the end of this chapter, you won't just know *what UNIX is*; you'll be using FreeBSD confidently as both a user and an aspiring systems programmer.

3.1 Reader Guidance: How to Use This Chapter

This chapter is not just something to skim, it's designed to be both a **reference** and a **hands-on bootcamp**. How long it takes depends on how you approach it:

- **Reading only:** About **2 hours** to go through the text and examples at a comfortable beginner's pace.
- **Reading + labs:** About **4 hours** if you pause to type and run each of the hands-on labs in your own FreeBSD system.
- **Reading + challenges:** About **6 hours or more** if you also complete the full set of 46 challenge exercises at the end.

Recommendation: Don't try to do everything in one sitting. Break the chapter into sections, and after each one, run the lab before moving on. Save the challenges for when you feel confident and want to test your mastery.

3.2 Introduction: Why UNIX Matters

Before we start writing device drivers for FreeBSD, we need to pause and talk about the foundation they stand on: **UNIX**.

Every driver you'll ever write for FreeBSD, every system call you'll explore, every kernel message you'll read, they all make sense only when you understand the operating system they live in. For a beginner, the world of UNIX can feel mysterious, filled with odd commands and a very different philosophy compared to Windows or macOS. But once you learn its logic, you'll see it's not only approachable but also elegant.

This chapter is about giving you a **gentle introduction** to UNIX as it appears in FreeBSD. By the end, you'll feel comfortable navigating the system, working with files, running commands, managing processes, installing

applications, and even writing small scripts to automate your tasks. These are everyday skills for any FreeBSD developer and absolutely essential before we start on kernel development.

3.2.1 Why Should You Learn UNIX Before Writing Drivers?

Think of it like this: if writing drivers is like building an engine, UNIX is the entire car around it. You need to know where the fuel goes, how the dashboard works, and what the controls do before you can safely swap parts under the hood.

Here are a few reasons why learning UNIX basics is essential:

- **Everything in UNIX is connected.** Files, devices, processes, they all follow consistent rules. Once you know these rules, the system becomes predictable.
- **FreeBSD is a direct descendant of UNIX.** The commands, filesystem layout, and overall philosophy are not add-ons; they are part of its DNA.
- **Drivers integrate with userland.** Even though your code will run in the kernel, it will interact with user programs, files, and processes. Understanding the userland environment helps you design drivers that feel natural and intuitive.
- **Debugging requires UNIX skills.** When your driver misbehaves, you'll rely on tools like `dmesg`, `sysctl`, and shell commands to figure out what's happening.

3.2.2 What You Will Learn in This Chapter

By the end of this chapter, you will:

- Understand what UNIX is and how FreeBSD fits into its family.
- Be able to use the shell to run commands and manage files.
- Navigate the FreeBSD filesystem and know where things live.
- Manage users, groups, and file permissions.
- Monitor processes and system resources.
- Install and remove applications using FreeBSD's package manager.
- Automate tasks with shell scripts.
- Peek into FreeBSD internals with tools like `dmesg` and `sysctl`.

All along the way, I'll give you **hands-on labs** so you can practice. Reading about UNIX is not enough; you need to **touch the system**. Each lab will involve real commands you'll run on a FreeBSD installation, so by the time you reach the end of this chapter, you won't just understand UNIX, you'll be using it confidently.

3.2.3 The Bridge to Device Drivers

Why are we spending an entire chapter on UNIX basics if this is a book about writing drivers? Because drivers don't exist in isolation. When you eventually load your own kernel module, you'll see it appear under `/dev`. When you test it, you'll use shell commands to read and write to it. When you debug it, you'll rely on system logs and monitoring tools.

So think of this chapter as laying down the **operating system literacy** you need before becoming a driver developer. Once you have it, everything else will feel less intimidating and far more logical.

3.2.4 Wrapping Up

In this opening section, we looked at why UNIX matters for anyone who wants to write FreeBSD drivers. Drivers don't live in isolation; they exist inside a larger operating system that follows rules, conventions, and a philosophy inherited from UNIX. Understanding this foundation is what makes everything else, from using the shell to debugging drivers, logical instead of mysterious.

With that motivation in mind, it's time to ask the natural next question: **what exactly is UNIX?** To move forward, we'll take a closer look at its history, its guiding principles, and the key concepts that still shape FreeBSD today.

3.3 What Is UNIX?

Before you can get comfortable using FreeBSD, it helps to understand what UNIX is and why it matters. UNIX isn't just a piece of software, it's a family of operating systems, a set of design choices, and even a philosophy that has shaped computing for more than fifty years. FreeBSD is one of its most important modern descendants, so learning UNIX is like studying the family tree to see where FreeBSD fits.

3.3.1 A Brief History of UNIX

UNIX was born in **1969** at Bell Labs, when Ken Thompson and Dennis Ritchie created a lightweight operating system for a PDP-7 minicomputer. At a time when mainframes were huge, costly, and complex, UNIX stood out because it was **small, elegant, and designed for experimentation**.

The **1973 rewrite in C** was the turning point. For the first time, an operating system was portable: you could move UNIX to different hardware by recompiling it, not by rewriting everything from scratch. This was unheard of in the 1970s and changed the trajectory of system design forever.

BSD at Berkeley is the part of the story that leads directly to FreeBSD. Graduate students and researchers at the University of California, Berkeley, took AT&T's UNIX source code and extended it with modern features:

- **Virtual memory** (so programs weren't limited by physical RAM).
- **Networking** (the TCP/IP stack that still powers the internet today).
- **The C shell** with scripting and job control.

In the **1990s**, after legal disputes over UNIX source code were resolved, the FreeBSD Project was launched. Its mission: to carry the BSD tradition forward, freely and openly, for anyone to use, modify, and share.

Today, FreeBSD is a direct continuation of that lineage. It is not a UNIX imitation; it is UNIX heritage alive and well.

You might be thinking, *"Why should I care?"*. You should because when you peek into `/usr/src` or type commands like `ls` and `ps`, you're not just using software, you're benefiting from decades of problem-solving and craftsmanship, the work of thousands of developers who built and polished these tools long before you.

3.3.2 The UNIX Philosophy

UNIX is not only a system; it's a **mindset**. Understanding its philosophy will make everything else, from basic commands to device drivers, feel more natural.

1. **Do one thing, and do it well.** Instead of giant, all-in-one programs, UNIX gives you focused tools.

Example: `grep` only searches text. It doesn't open files, edit them, or format results; it leaves that to other tools.

2. **Everything is a file.** Files are not just documents; they are the way you interact with almost everything: devices, processes, sockets, logs.

Analogy: Think of the entire system as a library. Every book, desk, and even the librarian's notebook is part of the same filing system.

3. **Build small tools, then combine them.** This is the genius of the **pipe operator (|)**. You take the output of one program and use it as the input for another.

Example:

```
ps -aux | grep ssh
```

Here, one program lists all processes, and another filters only the ones related to SSH. Neither program knows about the other, but the shell glues them together.

4. **Use plain text whenever possible.** Text files are easy to read, edit, share, and debug. FreeBSD's `/etc/rc.conf` (system configuration) is just a plain text file. No binary registries, no proprietary formats.

When you start writing device drivers, you'll see this philosophy everywhere: your driver will expose a **simple interface under /dev**, behave predictably, and integrate smoothly with other tools.

3.3.3 UNIX-like Systems Today

The word "UNIX" today refers less to a single operating system and more to a **family of UNIX-like systems**.

- **FreeBSD** - Your focus in this book. Used in servers, networking gear, firewalls, and embedded systems. Known for reliability and documentation. Many commercial appliances (routers, storage systems) silently run FreeBSD under the hood.
- **Linux** - Created in 1991, inspired by UNIX principles. Popular in data centers, embedded devices, and supercomputers. Unlike FreeBSD, Linux is not a direct UNIX descendant but shares the same interface and ideas.
- **macOS and iOS** - Built on Darwin, a BSD-based foundation. macOS is a UNIX-certified OS, meaning its command line tools behave like FreeBSD's. If you use a Mac, you already have a UNIX system.
- **Others** - Commercial variants like AIX, Solaris, or HP-UX still exist but are rare outside enterprise contexts.

Why this matters: Once you learn FreeBSD, you'll feel comfortable on almost any other UNIX-like system. The commands, filesystem layout, and philosophy all carry over.

3.3.4 Key Concepts and Terms

Here are some essential UNIX terms you'll see throughout this book:

- **Kernel** - The heart of the OS. It manages memory, CPU, devices, and processes. Your drivers will live here.
- **Shell** - The program that interprets your commands. It's your main tool for talking to the system.
- **Userland** - Everything outside the kernel: commands, libraries, daemons. It's where you'll spend most of your time as a user.
- **Daemon** - A background service (like `sshd` for remote logins or `cron` for scheduled tasks).
- **Process** - A running program. Each command creates a process.
- **File descriptor** - A numeric handle that the kernel gives programs to work with files or devices. For example, 0 = standard input, 1 = standard output, 2 = standard error.

Tip: Don't worry about memorizing these yet. Think of them as characters you'll meet again later in the story. By the time you write a driver, you'll know them like old friends.

3.3.5 How UNIX Differs from Windows

If you've mostly used Windows, the UNIX approach will feel different at first. Here are a few contrasts:

- **Drives vs. unified tree** Windows uses drive letters (`C:\`, `D:\`). UNIX has a single tree rooted at `/`. Disks and partitions are mounted into this tree.
- **Registry vs. text files** Windows centralizes settings in the Registry. UNIX uses plain-text configuration files under `/etc` and `/usr/local/etc`. You can open them with any text editor.
- **GUI focus vs. CLI focus** While Windows assumes a graphical interface, UNIX treats the command line as the primary tool. Graphical environments exist, but the shell is always available.
- **Permissions model** UNIX was multi-user from day one. Every file has permissions (read, write, execute) for the owner, group, and others. This makes security and sharing simpler and more consistent.

These differences explain why UNIX often feels "stricter" but also more transparent. Once you get used to it, the consistency becomes a huge advantage.

3.3.6 Everyday UNIX in Your Life

Even if you've never logged into a FreeBSD system before, UNIX is already around you:

- Your Wi-Fi router or NAS may run FreeBSD or Linux.
- Netflix uses FreeBSD servers to deliver streaming video.
- Sony's PlayStation uses a FreeBSD-based OS.
- macOS and iOS are direct descendants of BSD UNIX.
- Android phones run Linux, another UNIX-like system.

Learning FreeBSD is not just about writing drivers, it's about learning the **language of modern computing**.

3.3.7 Hands-On Lab: Your First UNIX Commands

Let's make this concrete. Open a terminal in the FreeBSD you installed in the previous chapter and try:

```
% uname -a
```

This prints system details: the OS, the system name, release version, kernel build, and machine type. On FreeBSD 14.x, you might see:

```
FreeBSD freebsd.edsonbrandi.com 14.3-RELEASE FreeBSD 14.3-RELEASE
↪ releng/14.3-n271432-8c9ce319fef7 GENERIC amd64
```

Now try the commands:

```
% date
% whoami
% hostname
```

- `date` - shows the current time and date.
- `whoami` - tells you which user account you're logged in as.
- `hostname` - shows the machine's network name.

Finally, a small experiment with UNIX's *"everything is a file"* idea:

```
% echo "Hello FreeBSD" > /tmp/testfile
% cat /tmp/testfile
```

You just created a file, wrote to it, and read it back. This is the same model you'll later use to talk to your own drivers.

3.3.8 Wrapping Up

In this section, you learned that UNIX is not just an operating system but a family of ideas and design principles that shaped modern computing. You saw how FreeBSD fits into this history as a direct descendant of BSD UNIX, why its philosophy of small tools and plain text makes it effective, and how many of the concepts you'll rely on as a driver developer, like processes, daemons, and file descriptors, have been part of UNIX since the beginning.

But knowing what UNIX is only gets us halfway. To really use FreeBSD, you need a way to **interact with it**. That's where the shell comes in, the command interpreter that lets you speak the system's language. In the next section, we'll start using the shell to run commands, explore the filesystem, and get hands-on experience with the tools that every FreeBSD developer depends on daily.

3.4 The Shell: Your Window Into FreeBSD

Now that you know what UNIX is and why it matters, it's time to start **talking to the system**. The way you do that in FreeBSD (and other UNIX-like systems) is through the **shell**.

Think of the shell as both an **interpreter** and a **translator**: you type a command in human-readable form, and the shell passes it to the operating system to execute. It's the window between you and the UNIX world.

3.4.1 What Is a Shell?

At its core, the shell is just a program, but a very special one. It listens to what you type, figures out what you mean, and asks the kernel to carry it out.

Some common shells include:

- **sh** - The original Bourne shell. Simple and reliable.
- **csh** / **tcsh** - The C shell and its enhanced version, with scripting features inspired by the C language. **tcsh** is FreeBSD's default for new users.
- **bash** - The Bourne Again Shell, very popular in Linux.
- **zsh** - A modern, user-friendly shell with many conveniences.

On FreeBSD 14.x, if you log in as a normal user, you'll probably be using **tcsh**. If you log in as the root administrator, you may see **sh**. Don't worry if you're not sure which shell you have, we'll cover how to check in just a moment.

Why this matters for driver developers: you'll use the shell constantly to compile, load, and test your drivers. Knowing how to navigate it is as important as knowing how to turn the ignition key in a car.

3.4.2 How to Know Which Shell You're Using

FreeBSD comes with more than one shell, and you might notice slight differences between them, for example, the prompt might look different, or certain shortcuts might behave differently. Don't worry: the **core UNIX commands work the same** no matter which shell you're in. Still, it's helpful to know which shell you're currently using, especially if you later decide to write scripts or customize your environment.

Type:

```
% echo $SHELL
```

You'll see something like:

```
/bin/tcsh
```

or

```
/bin/sh
```

This tells you your default shell. You don't need to change it now; just be aware that shells may look slightly different but share the same basic commands.

Hands-On Tip There's also a quick way to check which shell is running for your current process:

```
% echo $0
```

This may show `-tcsh`, `sh`, or something else. It's slightly different from `$SHELL`, because `$SHELL` tells you your **default shell** (the one you get when you log in), while `$0` tells you the **shell you're actually running right now**. If you ever start a different shell inside your session (for example by typing `sh` at the prompt), `$0` will reflect that.

3.4.3 The Structure of a Command

Every shell command follows the same simple pattern:

```
command [options] [arguments]
```

- **command** - The program you want to run.
- **options** - Flags that change how it behaves (usually starting with `-`).
- **arguments** - The targets of the command, like filenames or directories.

Example:

```
% ls -l /etc
```

- `ls` = list directory contents.
- `-l` = option for “long format.”
- `/etc` = argument (the directory to list).

This consistency is one of UNIX's strengths: once you learn the pattern, every command feels familiar.

3.4.4 Essential Commands for Beginners

We will walk through the core commands you'll use constantly.

3.4.4.1 Navigating Directories

- **pwd** - Print Working Directory Shows where you are in the filesystem.

```
% pwd
```

Output:

```
/home/dev
```

- **cd** - Change Directory Moves you into another directory.

```
% cd /etc
% pwd
```

Output:

```
/etc
```

- **ls** - List Shows the contents of a directory.

```
% ls
```

Output might include:

```
rc.conf  ssh/  resolv.conf
```

Tip: Try `ls -lh` for human-readable file sizes.

3.4.4.2 Managing Files and Directories

- **mkdir** - Make Directory

```
% mkdir projects
```

- **rmdir** - Remove Directory (only if empty)

```
% rmdir projects
```

- **cp** - Copy

```
% cp file1.txt file2.txt
```

- **mv** - Move (or rename)

```
% mv file2.txt notes.txt
```

- **rm** - Remove (delete)

```
% rm notes.txt
```

Warning: `rm` does not ask for confirmation. Once removed, the file is gone unless you have a backup. This is a common beginner pitfall.

3.4.4.3 Viewing File Contents

- **cat** - Concatenate and display file contents

```
% cat /etc/rc.conf
```

- **less** - View file contents with scrolling

```
% less /etc/rc.conf
```

Use arrow keys or spacebar, press `q` to quit.

- **head / tail** - Show the beginning or end of a file, `-n` parameter specifies the number of lines you want to see

```
% head -n 5 /etc/rc.conf
% tail -n 5 /etc/rc.conf
```

3.4.4.4 Editing Files

Sooner or later, you'll need to edit a configuration file or a source file. FreeBSD ships with a few editors, each with different strengths:

- **ee (Easy Editor)**

- Installed by default.
- Designed to be beginner-friendly, with visible menus at the top of the screen.
- To save, press **Esc**, then choose “*Leave editor*” → “*Save changes.*”
- Great choice if you've never used a UNIX editor before.

- **vi / vim**

- The traditional UNIX editor, always available.
- Extremely capable, but it has a steep learning curve.
- Beginners often get stuck because **vi** starts in *command mode* instead of insert mode.
- To start typing text: press **i**, write your text, then press **Esc** followed by **:wq** to save and quit.
- You don't need to master it now, but every sysadmin and developer eventually learns at least the basics of **vi**.

- **nano**

- Not part of the FreeBSD base system, but can be installed easily running the following command while logged as root:

```
# pkg install nano
```

- Very beginner-friendly, with shortcuts listed at the bottom of the screen.
- If you're coming from Linux distributions like Ubuntu, you may already know it.

Tip for Beginners Start with **ee** to get comfortable editing files on FreeBSD. Once you're ready, learn the basics of **vi**, it will always be there for you, even in rescue environments or minimal systems where nothing else is installed.

3.4.4.4.1 Hands-On Lab: Your First Edits

1. Create and edit a new file with **ee**:

```
% ee hello.txt
```

Write a short line of text, save, and exit.

2. Try the same with **vi**:

```
% vi hello.txt
```

Press **i** to insert, type something new, then press **Esc** and type **:wq** to save and quit.

3. If you installed nano:

```
% nano hello.txt
```

Notice how the bottom line shows commands like **^O** for save and **^X** for exit.

3.4.4.4.2 Common Beginner Pitfall: Stuck in vi Almost every UNIX beginner has faced this: you open a file with **vi**, start pressing keys, and nothing happens the way you expect. Worse, you can't figure out how to quit.

Here's what's happening:

- **vi** starts in **command mode**, not typing mode.
- To insert text, press **i** (insert).
- To return to command mode, press **Esc**.
- To save and quit: type **:wq** and press Enter.
- To quit without saving: type **:q!** and press Enter.

Tip: If you accidentally open **vi** and just want to escape, press **Esc**, type **:q!**, and hit Enter. That will exit without saving.

3.4.5 Tips and Shortcuts

Once you get comfortable typing commands, you'll quickly discover that the shell has many built-in features to save time and reduce mistakes. Learning these early will make you feel at home much faster.

Note about FreeBSD shells:

- The **default login shell** for new users is usually **/bin/tcsh**, which supports tab completion, history navigation with arrow keys, and many interactive shortcuts.
- The more minimal **/bin/sh** shell is excellent for scripting and system use, but it does not provide conveniences like tab completion or arrow-key history out of the box.
- So if some shortcuts below don't seem to work, check which shell you are using (`echo $SHELL`).

3.4.5.1 Tab completion (tcsh)

Start typing a command or filename and press **Tab**. The shell will try to complete it for you.

```
% cd /et<Tab>
```

Becomes:

```
% cd /etc/
```

If there's more than one match, press Tab twice to see a list of possibilities. This feature is not available in /bin/sh.

3.4.5.2 Command history (tcsh)

Press the **Up arrow** to recall the last command, and keep pressing it to walk further back in time. Press the **Down arrow** to move forward again.

```
% sysctl kern.hostname
```

You don't have to retype it, just hit the Up arrow and press Enter. In /bin/sh, you don't get arrow-key navigation (but you can still run commands again with !!).

3.4.5.3 Wildcards (globbing)

Works in *all* shells, including /bin/sh.

```
% ls *.conf
```

Lists all files that start with host and end with .conf.

```
% ls host?.conf
```

Matches files like host1.conf, hostA.conf, but not hosts.conf.

3.4.5.4 Editing on the command line (tcsh)

In tcsh you can move the cursor left and right with arrow keys, or use shortcuts:

- **Ctrl+A** → Move to the beginning of the line.
- **Ctrl+E** → Move to the end of the line.
- **Ctrl+U** → Erase everything from the cursor to the start of the line.
- **Repeating commands quickly (all shells)**

```
% !!
```

Re-executes your last command.

```
% !ls
```

Repeats the last command that started with ls.

Tip: If you want a friendlier interactive shell, stick with **/bin/tcsh** (the FreeBSD default for users). If you later want advanced customization, you can install shells like **bash** or **zsh** from packages or ports. But for scripting, always use **/bin/sh**, since it's guaranteed to be present and is the system's standard.

3.4.6 Hands-On Lab: Navigating and Managing Files

Let's practice:

1. Go to your home directory:

```
% cd ~
```

2. Create a new directory:

```
% mkdir unix_lab  
% cd unix_lab
```

3. Create a new file:

```
% echo "Hello FreeBSD" > hello.txt
```

4. View the file:

```
% cat hello.txt
```

5. Make a copy:

```
% cp hello.txt copy.txt  
% ls
```

6. Rename it:

```
% mv copy.txt renamed.txt
```

7. Delete the renamed file:

```
% rm renamed.txt
```

By completing these steps, you've just navigated the filesystem, created files, copied them, renamed them, and removed them, the daily bread and butter of UNIX work.

3.4.7 Wrapping Up

The shell is your **gateway into FreeBSD**. Every interaction with the system, whether running commands, compiling code, or testing a driver, flows through it. In this section, you learned what the shell is, how commands are structured, and how to perform basic navigation and file management.

Next, we'll explore **how FreeBSD organizes its filesystem**. Understanding the layout of directories like **/etc**, **/usr**, and **/dev** will give you a mental map of the system, which is especially important when we start dealing with device drivers that live under **/dev**.

3.5 The FreeBSD Filesystem Layout

In Windows, you may be used to drives like C:\ and D:\. In UNIX and FreeBSD, there are no drive letters. Instead, everything lives in a **single tree of directories** that starts at the root /.

This is called a **hierarchical filesystem**. At the very top is /, and everything else branches out beneath it like folders inside folders. Devices, configuration files, and user data are all organized inside this tree.

Here's a simplified map:

```
/
├─ bin      → Essential user commands (ls, cp, mv)
├─/sbin     → System administration commands (ifconfig, shutdown)
├─ etc      → Configuration files
├─ usr
│   ├─ bin  → Non-essential user commands
│   ├─/sbin → Non-essential system admin tools
│   ├─ local → Software installed by pkg or ports
│   └─ src   → FreeBSD source code
├─ var      → Logs, mail, spools, temp runtime data
├─ home     → User home directories
├─ dev      → Device files
└─ boot     → Kernel and boot loader
```

And here's a table with some of the most important directories you'll work with:

Directory	Purpose
/	Root of the entire system. Everything begins here.
/bin	Essential command-line tools (used during early boot).
/sbin	System binaries (like init, ifconfig).
/usr/bin	User command-line tools and programs.
/usr/sbin	System-level tools used by administrators.
/usr/src	FreeBSD source code (kernel, libraries, drivers).
/usr/local	Where packages and installed software go.
/boot	Kernel and boot loader files.
/dev	Device nodes, files that represent devices.
/etc	System configuration files.
/home	User home directories (like /home/dev).
/var	Log files, mail spools, runtime files.
/tmp	Temporary files, cleared on reboot.

Understanding this layout is critical for a driver developer because some directories, especially `/dev`, `/boot`, and `/usr/src`, are directly tied to the kernel and drivers. But even outside of those, knowing where things live helps you navigate confidently.

Base system vs local software: One key idea in FreeBSD is the separation between the base system and user-installed software. The base system: kernel, libraries, and essential tools live in `/bin`, `/sbin`, `/usr/bin`, and `/usr/sbin`. Everything you install later with `pkg` or `ports` goes into `/usr/local`. This separation keeps your core OS stable while letting you add and update software freely.

3.5.1 Devices as Files: `/dev`

One of UNIX's core ideas is that **devices appear as files** under `/dev`.

Examples:

- `/dev/null`: A “black hole” that discards anything you write to it.
- `/dev/zero`: Outputs an endless stream of zero bytes.
- `/dev/random`: Provides random data.
- `/dev/ada0`: Your first SATA disk.
- `/dev/da0`: A USB storage device.
- `/dev/tty`: Your terminal.

You can interact with these devices using the same tools you use for files:

```
% echo "test" > /dev/null
% head -c 10 /dev/zero | hexdump
```

Later in this book, when you create a driver, it will expose a file here, for example, `/dev/hello`. Writing to that file will actually run your driver's code inside the kernel.

3.5.2 Absolute vs. Relative Paths

When navigating the filesystem, paths can be:

- **Absolute** - Start at the root `/`. Example: `/etc/rc.conf`
- **Relative** - Start from your current location. Example: `../notes.txt`

Example:

```
% cd /etc      # absolute path
% cd ..        # relative path: move up one directory
```

Remember: `/` always means the root of the system, while `.` means “here” and `..` means “one level up.”

3.5.2.1 Example: Navigating with Absolute vs Relative Paths

Suppose your home directory contains this structure:

```
/home/dev/unix_lab/  
├─ docs/  
│   └─ notes.txt  
├─ code/  
│   └─ test.c  
└─ tmp/
```

- To open `notes.txt` with an **absolute path**:

```
% cat /home/dev/unix_lab/docs/notes.txt
```

- To open it with a **relative path** from inside `/home/dev/unix_lab`:

```
% cd /home/dev/unix_lab  
% cat docs/notes.txt
```

- Or, if you are already inside the `docs` directory:

```
% cd /home/dev/unix_lab/docs  
% cat ./notes.txt
```

Absolute paths always work, no matter where you are, while relative paths depend on your current directory. As a developer, you'll often prefer absolute paths in scripts (more predictable) and relative paths when working interactively (faster to type).

3.5.3 Hands-On Lab: Exploring the Filesystem

Let's practice exploring FreeBSD's layout:

1. Print your current location:

```
% pwd
```

2. Go to the root directory and list its contents:

```
% cd /  
% ls -lh
```

3. Peek into the `/etc` directory:

```
% ls /etc  
% head -n 5 /etc/rc.conf
```

4. Explore `/var/log` and view system logs:

```
% ls /var/log
% tail -n 10 /var/log/messages
```

5. Check devices under `/dev`:

```
% ls /dev | head
```

This lab gives you a “mental map” of the FreeBSD filesystem and shows how configuration files, logs, and devices are all organized in predictable places.

3.5.4 Wrapping Up

In this section, you’ve learned that FreeBSD uses a **single, hierarchical filesystem** starting at `/`, with key directories dedicated to system binaries, configuration, logs, user data, and devices. You also saw how `/dev` treats devices as files, a core concept that you’ll rely on when you write drivers.

But files and directories aren’t just about structure; they’re also about **who can access them**. UNIX is a multi-user system, and every file has an owner, a group, and permission bits that control what can be done with it. In the next section, we’ll explore **users, groups, and permissions**, and you’ll learn how FreeBSD keeps the system both secure and flexible.

3.6 Users, Groups, and Permissions

One of the biggest differences between UNIX and systems like early Windows is that UNIX was designed from the beginning as a **multi-user operating system**. That means it assumes multiple people (or services) can use the same machine at once, and it enforces rules about who can do what.

This design is essential for security, stability, and collaboration, and as a driver developer, you’ll need to understand it well, because permissions often control who can access your driver’s device file.

3.6.1 Users and Groups

Every person or service that uses FreeBSD does so under a **user account**.

- A **user** has a username, a numeric ID (UID), and a home directory.
- A **group** is a collection of users, identified by a group name and group ID (GID).

Each user belongs to at least one group, and permissions can be applied both to individuals and to groups.

You can see your current identity with:

```
% whoami
% id
```

Example output:

```
dev
uid=1001(dev) gid=1001(dev) groups=1001(dev), 0(wheel)
```

Here:

- Your username is dev.
- Your UID is 1001.
- Your primary group is dev.
- You also belong to the wheel group, which allows access to administrative privileges (via su or sudo).

3.6.2 File Ownership

In FreeBSD, every file and directory has an **owner** (a user) and a **group**.

Let's check with `ls -l`:

```
% ls -l hello.txt
```

Output:

```
-rw-r--r--  1 dev  dev  12 Aug 23 10:15 hello.txt
```

Breaking it down:

- `-rw-r--r--` = permissions (we'll cover in a moment).
- `1` = number of links (not important for now).
- `dev` = owner (the user who created the file).
- `dev` = group (the group associated with the file).
- `12` = file size in bytes.
- `Aug 23 10:15` = last modification time.
- `hello.txt` = filename.

So this file belongs to user dev and group dev.

3.6.3 Permissions

Permissions control what users can do with files and directories. There are three categories of users:

1. **Owner** - the user who owns the file.
2. **Group** - members of the file's group.
3. **Others** - everyone else.

And three kinds of permission bits:

- **r** = read (can view contents).
- **w** = write (can modify or delete).

- **x** = execute (for programs or, in directories, the ability to enter them).

Example:

```
-rw-r--r--
```

This means:

- **Owner** = read + write.
- **Group** = read only.
- **Others** = read only.

So the owner can modify the file, but everyone else can only look at it.

3.6.4 Changing Permissions

To modify permissions, you use the **chmod** command.

Two ways:

Symbolic mode

```
% chmod u+x script.sh
```

This adds execute permission (+x) for the user (u).

Octal mode

```
% chmod 755 script.sh
```

Here, numbers represent permissions:

- 7 = rwx
- 5 = r-x
- 0 = —

So 755 means: owner = rwx, group = r-x, others = r-x.

3.6.5 Changing Ownership

Sometimes you need to change who owns a file. Use **chown**:

```
% chown root:wheel hello.txt
```

Now the file is owned by root and the group wheel.

Note: Changing ownership usually requires administrator privileges.

3.6.6 Practical Scenario: Project Directory

Suppose you're working on a project with a teammate, and you both need access to the same files.

Here's how you'd set it up, run these commands as root:

1. Create a group called proj:

```
# pw groupadd proj
```

2. Add both users to the group:

```
# pw groupmod proj -m dev,teammate
```

3. Create a directory and assign it to the group:

```
# mkdir /home/projdir  
# sudo chown dev:proj /home/projdir
```

4. Set group permissions so members can write:

```
# chmod 770 /home/projdir
```

Now both users can work in /home/projdir, while others cannot access it.

This is exactly how UNIX systems enforce collaboration securely.

3.6.7 Hands-On Lab: Permissions in Action

Let's practice:

1. Create a new file:

```
% echo "secret" > secret.txt
```

2. Check its default permissions:

```
% ls -l secret.txt
```

3. Remove read access for others:

```
% chmod o-r secret.txt  
% ls -l secret.txt
```

4. Add execute permission for the user:

```
% chmod u+x secret.txt  
% ls -l secret.txt
```

5. Try to change ownership (will require root):

```
% sudo chown root secret.txt  
% ls -l secret.txt
```

Beware that `sudo` will ask for your password to execute the command shown above in the step 5.

With these commands, you've controlled access to files at a very fine level, a concept that applies directly when we create drivers, since drivers also have device files under `/dev` with ownership and permission rules.

3.6.8 Wrapping Up

In this section, you learned that FreeBSD is a **multi-user system** where every file has an owner, a group, and permission bits that control access. You saw how to inspect and change permissions, how to manage ownership, and how to set up collaboration safely with groups.

These rules may seem simple, but they are the backbone of FreeBSD's security model. Later, when you write drivers, your device files under `/dev` will also have ownership and permissions, controlling who can open and use them.

Next, we'll look at **processes**, the running programs that make the system alive. You'll learn how to see what's running, how to manage processes, and how FreeBSD keeps everything organized behind the scenes.

3.7 Processes and System Monitoring

So far, you've learned how to navigate the filesystem and manage files. But an operating system isn't just about files on disk; it's about **programs running in memory**. These running programs are called **processes**, and understanding them is essential for both daily use and driver development.

3.7.1 What Is a Process?

A process is a program in motion. When you run a command like `ls`, FreeBSD:

1. Loads the program into memory.
2. Assigns it a **process ID (PID)**.
3. Gives it resources like CPU time and memory.
4. Tracks it until it finishes or is stopped.

Processes are how FreeBSD manages everything that happens on your system. From the shell you type into, to daemons in the background, to your web browser, they're all processes.

For driver developers: When you write a driver, **processes in userland will talk to it**. Knowing how processes are created and managed helps you understand how drivers are used.

3.7.2 Foreground vs. Background Processes

Usually, when you run a command, it runs in the **foreground**, meaning you can't do anything else in that terminal until it finishes.

Example:

```
% sleep 10
```

This command pauses for 10 seconds. During that time, your terminal is “blocked.”

To run a process in the **background**, add an & at the end:

```
% sleep 10 &
```

Now you get your prompt back immediately, and the process runs in the background.

You can see background jobs with:

```
% jobs
```

And bring one back to the foreground:

```
% fg %1
```

(where %1 is the job number in the list you see with jobs).

3.7.3 Viewing Processes

To see which processes are running, use ps:

```
ps aux
```

Sample output:

USER	PID	%CPU	%MEM	VSZ	RSS	TT	STAT	STARTED	TIME	COMMAND
root	1	0.0	0.0	1328	640	-	Is	10:00AM	0:00.01	/sbin/init
dev	1024	0.0	0.1	4220	2012	-	S	10:05AM	0:00.02	-tcsh
dev	1055	0.0	0.0	1500	800	-	R	10:06AM	0:00.00	ps aux

Here:

- PID = process ID.
- USER = who started it.
- %CPU / %MEM = resources being used.
- COMMAND = the program running.

3.7.3.1 Watching Processes and System Load with top

While ps gives you a snapshot of processes at a single moment, sometimes you want a **live view** of what’s happening on your system. That’s where the top command comes in.


```
% top
```

This opens a continuously updating display of system activity. By default, it refreshes every 2 seconds. To quit, press **q**.

The top screen shows:

- **Load averages** (how busy your system is, averaged over 1, 5, and 15 minutes).
- **Uptime** (how long the system has been running).
- **CPU usage** (user, system, idle).
- **Memory and swap usage**.
- **A list of processes**, sorted by CPU usage, so you can see which programs are working hardest.

Example of top Output (simplified): .

```
last pid: 3124; load averages: 0.06, 0.12, 0.14
↵ up 0+20:43:11 11:45:09
17 processes: 1 running, 16 sleeping
CPU: 0.0% user, 0.0% nice, 0.0% system, 0.0% interrupt, 100% idle
Mem: 5480K Active, 1303M Inact, 290M Wired, 83M Buf, 387M Free
Swap: 1638M Total, 1638M Free
```

PID	USERNAME	THR	PRI	NICE	SIZE	RES	STATE	C	TIME	WCPU	COMMAND
3124	dev	1	20	0	15M	3440K	CPU3	3	0:00	0.03%	top
2780	dev	1	20	0	23M	11M	select	0	0:00	0.01%	sshd-session
639	root	1	20	0	14M	2732K	select	2	0:02	0.00%	syslogd
435	root	1	20	0	15M	4012K	select	2	0:04	0.00%	devd
730	root	1	20	0	14M	2612K	nanslp	0	0:00	0.00%	cron
697	root	2	20	0	18M	4388K	select	3	0:00	0.00%	qemu-ga
2778	root	1	20	0	23M	11M	select	1	0:00	0.00%	sshd-session
726	root	1	20	0	23M	9164K	select	3	0:00	0.00%	sshd
760	root	1	68	0	14M	2272K	ttyin	1	0:00	0.00%	getty

Here we can see:

- The system has been up for more than a day.
- Load averages are very low (system is idle).
- CPU is mostly idle.
- Memory is mostly free.
- The yes command (a test program that just outputs “y” endlessly) is using almost all the CPU.

3.7.3.1.1 Quick check with uptime If you don’t need the full detail of top, you can use:

```
% uptime
```

Which shows something like:

```
3:45PM up 2 days,  4:11,  2 users,  load averages:  0.32,  0.28,  0.25
```

This tells you:

- Current time.
- How long the system has been running.
- How many users are logged in.
- Load averages (1, 5, 15 minutes).

Tip: Load averages are a quick way to see if your system is overloaded. On a single-CPU system, a load average of 1.00 means the CPU is fully busy. On a 4-core system, 4.00 means all cores are fully loaded.

Hands-On Lab: Watching the System

1. Run `uptime` and note your system's load averages.
2. Open two terminals on your FreeBSD machine.
3. On the first terminal, start a busy process:

```
% yes > /dev/null &
```

4. On the second terminal run `top` to see how much CPU the `yes` process is using.
5. Stop the `yes` command with `kill %1` or `pkill yes`, or just by executing a `ctrl+c` on the first terminal
6. Run `uptime` again, notice how the load average is slightly higher than before, but it will drop back down over time.

3.7.4 Stopping Processes

Sometimes a process misbehaves or needs to be stopped. You can use:

- **kill** - Send a signal to a process.

```
% kill 1055
```

(replace 1055 with the actual PID).

- **kill -9** - Force a process to terminate immediately.

```
% kill -9 1055
```

Use `kill -9` only when necessary, because it doesn't give the program a chance to clean up.

When you use `kill`, you're not literally "*killing*" a process with brute force; you're sending it a **signal**. Signals are messages the kernel delivers to processes.

- By default, `kill` sends **SIGTERM (signal 15)**, which politely asks the process to terminate. Well-behaved programs clean up and exit.

- If a process refuses, you can send **SIGKILL (signal 9)** with `kill -9 PID`. This forces the process to stop immediately, without cleanup.
- Another useful one is **SIGHUP (signal 1)**, often used to tell daemons (background services) to reload their configuration.

Try this:

```
% sleep 100 &
% ps aux | grep sleep
% kill -15 <PID>  # try with SIGTERM first
% kill -9 <PID>   # if still running, use SIGKILL
```

As a future driver developer, this distinction matters. Your code might need to handle termination gracefully, cleaning up resources instead of leaving the kernel in an unstable state.

3.7.4.1 Process Hierarchy: Parents and Children

Every process in FreeBSD (and in UNIX systems in general) has a **parent** process that started it. For example, when you type a command in the shell, the shell process is the parent, and the command you run becomes its child.

You can view this relationship using `ps` with custom columns:

```
% ps -o pid,ppid,command | head -10
```

Example of the output (simplified):

PID	PPID	COMMAND
1	0	/sbin/init
534	1	/usr/sbin/cron
720	534	/bin/sh
721	720	sleep 100

Here you can see:

- Process **1** is `init`, the ancestor of all processes.
- `cron` was started by `init`.
- A `sh` shell process was started by `cron`.
- The `sleep 100` process was started by the shell.

Understanding process hierarchy matters when debugging: if a parent dies, its children may be **adopted by `init`**. Later, when you work on drivers, you'll see how system daemons and services create and manage child processes that interact with your code.

3.7.5 Monitoring System Resources

FreeBSD provides simple commands to check system health:

- **df -h** - Show disk usage.

```
% df -h
```

Example:

Filesystem	Size	Used	Avail	Capacity	Mounted on
/dev/ada0p2	50G	20G	28G	42%	/

- **du -sh** - Show size of a directory.

```
% du -sh /var/log
```

- **freebsd-version** - Show OS version.

```
% freebsd-version
```

- **sysctl** - Query system information.

```
% sysctl hw.model
```

```
% sysctl hw.ncpu
```

Output might show your CPU model and number of cores.

Later, when writing drivers, you'll often use `dmesg` and `sysctl` to monitor how your driver interacts with the system.

3.7.6 Hands-On Lab: Working with Processes

Let's practice:

1. Run a sleep command in the background:

```
% sleep 30 &
```

2. Check running jobs:

```
% jobs
```

3. List processes:

```
% ps aux | grep sleep
```

4. Stop the process:

```
% kill <PID>
```

5. Run `top` and watch system activity. Press `q` to quit.

6. Check system info:

```
% sysctl hw.model  
% sysctl hw.ncpu
```

3.7.7 Wrapping Up

In this section, you learned that processes are the living, running programs inside FreeBSD. You saw how to start them, move them between foreground and background, inspect them with `ps` and `top`, and stop them with `kill`. You also explored basic system monitoring commands to check disk, CPU, and memory usage.

Processes are essential because they make the system alive, and as a driver developer, the programs that use your driver will always run as processes.

But monitoring processes is only part of the story. To do real work, you'll need more tools than those included in the base system. FreeBSD provides a clean and flexible way to install and manage extra software, from simple utilities like `nano` to large applications like web servers. In the next section, we'll look at the **FreeBSD package system and the Ports Collection**, so you can extend your system with the software you need.

3.8 Installing and Managing Software

FreeBSD is designed as a lean and reliable operating system. Out of the box, you get a rock-solid **base system**, the kernel, system libraries, essential tools, and configuration files. Everything beyond that, editors, compilers, servers, monitoring tools, and even desktop environments, is considered **third-party software**, and FreeBSD provides two excellent ways to install it:

1. **pkg** - The binary package manager: fast, simple, and convenient.
2. **The Ports Collection** - A massive source-based build system that allows fine-tuned customization.

Together, they give FreeBSD one of the most flexible software ecosystems in the UNIX world.

3.8.1 Binary Packages with `pkg`

The `pkg` tool is FreeBSD's modern package manager. It gives you access to **tens of thousands of prebuilt applications** maintained by the FreeBSD ports team.

When you install a package with `pkg`, here's what happens:

- The tool fetches a **binary package** from the FreeBSD mirrors.
- Dependencies are downloaded automatically.
- Files are installed under `/usr/local`.
- The package database tracks what was installed, so you can update or remove it later.

3.8.1.1 Common Commands

- Update the package repository:

```
% sudo pkg update
```

- Search for software:

```
% sudo pkg search htop
```

- Install software:

```
% sudo pkg install htop
```

- Upgrade all packages:

```
% sudo pkg upgrade
```

- Remove software:

```
% sudo pkg delete htop
```

For beginners, pkg is the fastest and safest way to get software installed.

3.8.2 The FreeBSD Ports Collection

The **Ports Collection** is one of FreeBSD’s crown jewels. It is a **giant tree of build recipes** (called “ports”) located under `/usr/ports`. Each port contains:

- A **Makefile** describing how to fetch, patch, configure, and build the software.
- Checksums for verifying integrity.
- Metadata about dependencies and licensing.

When you build software from ports, FreeBSD downloads the source code from the original project’s site, applies FreeBSD-specific patches, and compiles it locally on your system.

3.8.2.1 Why Use Ports?

So why would anyone bother building from source when prebuilt packages are available?

- **Customization** - Many applications have optional features. With ports, you can choose exactly what to enable or disable during compilation.
- **Optimization** - Advanced users may want to compile with flags tuned for their hardware.
- **Bleeding-edge options** - Sometimes new features are available in ports before they make it into binary packages.
- **Consistency with pkg** - Ports and packages share the same underlying infrastructure. In fact, packages are built from ports by the FreeBSD build cluster.

3.8.2.2 Getting and Exploring the Ports Tree

The Ports Collection lives under `/usr/ports`, but on a fresh FreeBSD system this directory may not exist yet. Let’s check:

```
% ls /usr/ports
```

If you see categories such as `archivers`, `editors`, `net`, `security`, `sysutils`, and `www`, then Ports is installed. If the directory is missing, you'll need to fetch the Ports tree yourself.

3.8.2.3 Installing the Ports Collection with Git

The official and recommended way is to use **Git**:

1. Make sure `git` is installed:

```
% sudo pkg install git
```

2. Clone the official Ports repository:

```
% sudo git clone https://git.FreeBSD.org/ports.git /usr/ports
```

This will create `/usr/ports` and fill it with the entire Ports Collection. The initial clone can take some time, since it contains thousands of applications.

3. To update the ports tree later, just run:

```
% cd /usr/ports
% sudo git pull
```

There is also an older tool called `portsnap`, but **Git is the modern, recommended method** because it keeps your tree directly in sync with the FreeBSD project's repository.

3.8.2.4 Browsing the Ports

Once Ports is installed, explore it:

```
% cd /usr/ports
% ls
```

You'll see files and categories such as:

CHANGES	UIDs	comms	ftp	mail	
↪ portuguese	x11				
CONTRIBUTING.md	UPDATING	converters	games	math	print
↪ x11-clocks					
COPYRIGHT	accessibility	databases	german	misc	russian
↪ x11-drivers					
GIDs	arabic	deskutils	graphics	multimedia	science
↪ x11-fm					
Keywords	archivers	devel	hebrew	net	security
↪ x11-fonts					

MOVED	astro	dns	hungarian	net-im	shells
↳ x11-servers					
Makefile	audio	editors	irc	net-mgmt	sysutils
↳ x11-themes					
Mk	benchmarks	emulators	japanese	net-p2p	textproc
↳ x11-toolkits					
README	biology	filesystems	java	news	ukrainian
↳ x11-wm					
Templates	cad	finance	korean	polish	vietnamese
Tools	chinese	french	lang	ports-mgmt	www

Each category has subdirectories for specific applications. For example:

```
% cd /usr/ports/sysutils/memdump
% ls
```

Here you'll find files like `Makefile`, `distinfo`, `pkg-descr` and possibly a `files/` directory. These are the “ingredients” FreeBSD uses to build the application: the `Makefile` defines the process, `distinfo` ensures integrity, `pkg-descr` describes what this software does, and `files/` contains any FreeBSD-specific patches.

3.8.2.5 Building from Ports

Example: installing `memdump` from ports.

```
% cd /usr/ports/sysutils/memdump
% sudo make install clean
```

During the build, you may see a menu of options, such as enabling sensors or colours, installing documentation, etc. This is where ports shine, you control what features are compiled.

The `make install clean` process does three things:

- **install** - builds and installs the program.
- **clean** - removes temporary build files.

3.8.2.6 Mixing Ports and Packages

A common question: *Can I mix packages and ports?*

Yes, they're compatible, since both are built from the same source tree. However, if you rebuild something from ports with custom options, you should be careful not to accidentally overwrite it with a binary package update later.

Many users install most things with `pkg`, but use ports for specific applications where customization is important.

3.8.3 Where Installed Software Goes

Both pkg and ports install third-party software under `/usr/local`. This keeps them separate from the base system.

Typical locations:

- **Binaries** → `/usr/local/bin`
- **Libraries** → `/usr/local/lib`
- **Configuration** → `/usr/local/etc`
- **Man pages** → `/usr/local/man`

Try:

```
% which nano
```

Output:

```
/usr/local/bin/nano
```

This confirms that nano came from packages/ports, not from the base system.

3.8.4 Practical Example: Installing vim and htop

Let's try both methods.

3.8.4.1 Using pkg

```
% sudo pkg install vim htop
```

Run them:

```
% vim test.txt  
% htop
```

3.8.4.2 Using Ports

```
% cd /usr/ports/sysutils/htop  
% sudo make install clean
```

Run it:

```
% htop
```

Notice how the ports version may ask you about optional features during build, while pkg installs with defaults.

3.8.5 Hands-On Lab: Managing Software

1. Update your package repository:

```
% sudo pkg update
```

2. Install lynx with pkg:

```
% sudo pkg install lynx  
% lynx https://www.freebsd.org
```

3. Search for bsdinfo:

```
% pkg search bsdinfo
```

4. Install bsdinfo from ports:

```
% cd /usr/ports/sysutils/bsdinfo  
% sudo make install clean
```

5. Run bsdinfo to confirm it is now installed:

```
% bsdinfo
```

6. Remove nano:

```
% sudo pkg delete nano
```

You've now installed, run, and removed software with both pkg and ports, two complementary methods that give FreeBSD its flexibility.

3.8.6 Wrapping Up

In this section, you learned how FreeBSD handles third-party software:

- The **pkg system** gives you fast, easy binary installations.
- The **Ports Collection** offers source-based flexibility and customization.
- Both methods install under `/usr/local`, keeping the base system separate and clean.

Understanding this ecosystem is key to FreeBSD culture. Many administrators install common tools with pkg and turn to ports when they need fine-grained control. As a developer, you'll appreciate both approaches: pkg for convenience, and ports when you want to see exactly how software is built and integrated.

But applications are only part of the story. The FreeBSD **base system**, the kernel and core utilities also need regular updates to stay secure and reliable. In the next section, we'll learn how to use `freebsd-update` to keep the operating system itself current, so you always have a solid foundation to build on.

3.9 Keeping FreeBSD Up to Date

One of the most important habits you can develop as a FreeBSD user is keeping your system up to date. Updates fix security issues, squash bugs, and sometimes add support for new hardware. Unlike the command `pkg update && pkg upgrade`, which update your applications, **freebsd-update command is used to update the base operating system itself**, including the kernel and core utilities.

Keeping your system current ensures you're running FreeBSD safely and gives you the same solid foundation other developers are building on.

3.9.1 Why Updates Matter

- **Security:** Like any software, FreeBSD occasionally has security vulnerabilities. Updates patch these quickly.
- **Stability:** Bug fixes improve reliability, which is critical if you're developing drivers.
- **Compatibility:** Updates bring support for new CPUs, chipsets, and other hardware.

Don't think of updates as optional. They're part of responsible system administration.

3.9.2 The `freebsd-update` Tool

FreeBSD makes updating simple with the `freebsd-update` tool. It works by:

1. **Fetching** information about available updates.
2. **Applying** binary patches to your system.
3. If needed, **rebooting** into the updated kernel.

This is much easier than rebuilding the system from source (which we'll learn later, when we need that level of control).

3.9.3 The Update Workflow

Here's the standard process:

1. **Fetch available updates**

```
% sudo freebsd-update fetch
```

This contacts the FreeBSD update servers and downloads any security patches or bug fixes for your version.

2. **Review changes** After fetching, `freebsd-update` may show you a list of configuration files that would be modified. Example:

```
The following files will be updated as part of updating to 14.1-RELEASE-p3:  
/bin/ls  
/sbin/init  
/etc/rc.conf
```

Don't panic! This doesn't mean your system is broken, it just means some files will be updated.

- If system configuration files like `/etc/rc.conf` have changed in the base system, you'll be asked to review differences.
- `freebsd-update` uses a merge tool to show side-by-side changes.
- For beginners: if you're not sure, it's usually safe to **accept the default (keep your local version)**. You can always read `/var/db/freebsd-update` logs later.

Tip: If you're uncomfortable merging config files at this point, you can skip changes and check them manually later.

3. Install updates

```
% sudo freebsd-update install
```

This step applies the updates that were downloaded.

- If the update includes only userland programs (like `ls`, `cp`, libraries), you're done.
- If the update includes a **kernel patch**, you'll be asked to **reboot** after installation.

3.9.4 Example Session

Here's what a normal update might look like:

```
% sudo freebsd-update fetch  
Looking up update.FreeBSD.org mirrors... 3 mirrors found.  
Fetching metadata signature for 14.3-RELEASE from update1.FreeBSD.org... done.  
Fetching metadata index... done.  
Fetching 1 patches..... done.  
Applying patches... done.  
The following files will be updated as part of updating to 14.3-RELEASE-p1:  
  /bin/ls  
  /bin/ps  
  /sbin/init  
% sudo freebsd-update install  
Installing updates... done.
```

If the kernel was updated:

```
% sudo reboot
```

After reboot, your system is fully patched.

3.9.5 Kernel Updates with `freebsd-update`

One of the useful things about `freebsd-update` is that it can update the kernel itself. You don't have to rebuild it manually unless you want to run a custom kernel (we'll cover that later in the book).

This means that for most users, staying secure and current is just a matter of running `fetch + install` regularly.

3.9.6 Upgrading to a New Release with `freebsd-update`

In addition to applying security and bug fix patches, `freebsd-update` can also upgrade your system to a **new FreeBSD release**. For example, if you are running **FreeBSD 14.2** and want to upgrade to **14.3**, the process is straightforward.

The workflow has three steps:

1. **Fetch upgrade files**

```
% sudo freebsd-update upgrade -r 14.3-RELEASE
```

Replace `14.3-RELEASE` with the release you want to upgrade to.

2. **Install the new components**

```
% sudo freebsd-update install
```

This installs the first stage of updates. If the kernel was updated, you will need to reboot:

```
% sudo reboot
```

3. **Repeat install** After reboot, run the install step again to finish updating the rest of the system:

```
% sudo freebsd-update install
```

At the end, you'll be running the new release. You can confirm with:

```
% freebsd-version
```

Tip: Release upgrades can sometimes involve configuration file merges (just like security updates). If in doubt, keep your local versions, you can always compare with the new defaults stored under `/var/db/freebsd-update/`.

And remember, it's also a good idea to update your **packages** after a release upgrade, since they are built against the new system libraries:

```
% sudo pkg update
% sudo pkg upgrade
```

3.9.7 Hands-On Lab: Running Your First Update

1. Check your current FreeBSD version:

```
% freebsd-version -kru
```

- -k → kernel
- -r → running
- -u → userland

2. Run `freebsd-update fetch` to see if updates are available.
3. Carefully read any messages about configuration file merges. If unsure, choose to **keep your version**.
4. Run `freebsd-update install` to apply updates.
5. If the kernel was updated, reboot:

```
% sudo reboot
```

Common Beginner Pitfall: Fear of Config File Merges

When `freebsd-update` asks you to merge changes, it can look intimidating, with lots of text, plus/minus symbols, and prompts. Don't worry.

- If in doubt, keep your local version of files like `/etc/rc.conf` or `/etc/hosts`.
- The system will still work.
- You can always inspect the new default files later (they're stored in `/var/db/freebsd-update/`).

With time, you'll get comfortable resolving these merges, but in the beginning, **choosing to keep your configuration is the safe route**.

3.9.8 Wrapping Up

With just two commands, `freebsd-update fetch` and `freebsd-update install`, you now know how to keep your FreeBSD base system patched and secure. This process takes only a few minutes but ensures your environment is safe and reliable for development work.

Later, when we begin working on the kernel and writing drivers, we'll also learn how to build and install a custom kernel from source. But for now, you already have the essential knowledge to maintain your system like a pro.

And since checking for updates is something you may want to do regularly, wouldn't it be nice if the system could take care of some of these chores for you automatically? That's exactly what we'll look at next: **scheduling and automation** with tools like `cron`, `at`, and `periodic`.

3.10 Scheduling and Automation

One of UNIX's greatest strengths is that it was designed to let the computer handle repetitive tasks for you. Instead of waiting until midnight to run a backup or logging in every morning to start a monitoring script, you can tell FreeBSD:

“Run this command for me at this time, every day, forever.”

This not only saves time but also makes your system more reliable. In FreeBSD, the main tools for this are:

1. **cron** - for recurring tasks, like backups or monitoring.
2. **at** - for one-time jobs you want to schedule later.
3. **periodic** - FreeBSD’s built-in system for routine maintenance tasks.

3.10.1 Why Automate Tasks?

Automation matters because enhance our:

- **Consistency** - A job scheduled with cron will always run, even if you forget.
- **Efficiency** - Instead of manually repeating commands, you write them once.
- **Reliability** - Automation helps avoid mistakes. Computers don’t forget to rotate logs on Sunday night.
- **System maintenance** - FreeBSD itself relies heavily on cron and periodic to keep the system healthy (rotate logs, update databases, run security checks).

3.10.2 cron: The Automation Workhorse

The cron daemon runs continuously in the background. Every minute, it checks a list of scheduled jobs (stored in crontabs) and runs those that match the current time.

Each user has their own **crontab**, and the system has a global one. This means you can schedule personal tasks (like cleaning files in your home directory) without touching system jobs.

3.10.3 Understanding the crontab Format

The crontab format has **five fields** that describe *when* to run a job, followed by the command itself:

minute	hour	day	month	weekday	command

- **minute**: 0-59
- **hour**: 0-23 (24-hour clock)
- **day**: 1-31
- **month**: 1-12
- **weekday**: 0-6 (0 = Sunday, 6 = Saturday)

A mnemonic to help you remember: *“My Hungry Dog Must Wait.”* (Minute, Hour, Day, Month, Weekday)

3.10.3.1 Examples of cron jobs

- Run every day at midnight:

```
0 0 * * * /usr/bin/date >> /home/dev/midnight.log
```

- Run every 15 minutes:

```
* /15 * * * * /home/dev/scripts/check_disk.sh
```

- Run every Monday at 8 AM:

```
0 8 * * 1 echo "Weekly meeting" >> /home/dev/reminder.txt
```

- Run at 3:30 AM on the first of every month:

```
30 3 1 * * /usr/local/bin/backup.sh
```

3.10.4 Editing and Managing Crontabs

To edit your personal crontab:

```
crontab -e
```

This opens your crontab in the default editor (vi or ee).

To list your jobs:

```
crontab -l
```

To remove your crontab:

```
crontab -r
```

3.10.5 Where Do Logs Go?

When cron runs a job, its output (stdout and stderr) is sent by **email** to the user who owns the job. On FreeBSD, these emails are delivered locally and stored in `/var/mail/username`.

You can also redirect output to a log file to make things easier:

```
0 0 * * * /home/dev/backup.sh >> /home/dev/backup.log 2>&1
```

Here:

- `>>` appends output to `backup.log`.
- `2>&1` redirects error messages (stderr) into the same file.

This way, you always know what your cron jobs did, even if you don't check system mail.

3.10.6 at: One-Time Scheduling

Sometimes you don't want a recurring job, you just want something to run later, once. That's where **at** comes in.

Before a user can use the **at** the superuser must first append the user's username to the file `/var/at/at.allow`.


```
# echo "dev" >> /var/at/at.allow
```

Now the user can execute the `at` command. The usage is pretty simple, let's see some examples:

- Run a command 10 minutes from now:

```
% echo "echo Hello FreeBSD > /home/dev/hello.txt" | at now + 10 minutes
```

- Run a command tomorrow at 9 AM:

```
% echo "/usr/local/bin/htop" | at 9am tomorrow
```

Jobs scheduled with `at` are queued and run exactly once. You can list them with `atq` and remove them with `atrm`.

3.10.7 periodic: FreeBSD's Maintenance Helper

FreeBSD comes with a built-in housekeeping system called **periodic**. It's a framework of shell scripts that handle routine maintenance tasks for you, so you don't have to remember them manually.

These tasks run automatically at **daily, weekly, and monthly intervals**, thanks to entries already configured in the system-wide cron file `/etc/crontab`. This means a freshly installed FreeBSD system already takes care of many chores without you lifting a finger.

3.10.7.1 Where the Scripts Live

The scripts are organized in directories under `/etc/periodic`:

```
/etc/periodic/daily  
/etc/periodic/weekly  
/etc/periodic/monthly  
/etc/periodic/security
```

- **daily/** - jobs that run every day (log rotation, security checks, database updates).
- **weekly/** - jobs that run once a week (like updating the locate database).
- **monthly/** - jobs that run once a month (like monthly accounting reports).
- **security/** - additional checks focused on system security.

3.10.7.2 What periodic Does by Default

Some examples of the jobs included out of the box:

- **Security checks** - looks for setuid binaries, insecure file permissions, or known vulnerabilities.
- **Log rotation** - compresses and archives logs under `/var/log` so they don't grow forever.

- **Database updates** - rebuilds helper databases, like the one used by the `locate` command.
- **Temporary file cleanup** - removes leftovers in `/tmp` and other cache directories.

After they run, periodic scripts usually send a summary of their results to the **root user's mailbox** (read it by running `mail` as root).

Common Beginner Pitfall: “Nothing Happened!”

Many new FreeBSD users run their system for a few days, knowing periodic is supposed to run jobs daily, but they never see any output and assume it didn't work. In reality, periodic's reports are sent to the **root user's mail**, not displayed on screen.

To read them, log in as root and run:

```
# mail
```

Press Enter to open the mailbox and view the reports. You can quit the mail program by typing `q`.

Tip: If you prefer to receive these reports in your normal user's inbox, you can configure mail forwarding in `/etc/aliases` so that root's mail is redirected to your user account.

3.10.7.3 Running periodic Manually

You don't have to wait for cron to trigger them. You can run the full sets of jobs manually:

```
% sudo periodic daily
% sudo periodic weekly
% sudo periodic monthly
```

Or run just one script directly, for example:

```
% sudo /etc/periodic/security/100.chksetuid
```

3.10.7.4 Customizing periodic with `periodic.conf`

Periodic is not a black box. Its behavior is controlled through `/etc/periodic.conf` and `/etc/periodic.conf.local`.

Best practice: never edit the scripts themselves. Instead, override their behavior in `periodic.conf`, this keeps your changes safe when FreeBSD updates the base system.

Here are some common options you might use:

- **Enable or disable jobs**

```
daily_status_security_enable="YES"
daily_status_network_enable="NO"
```
- **Control log handling**

```
daily_clean_hoststat_enable="YES"
```

```
weekly_clean_pkg_enable="YES"
```

- **Enable locate database update**

```
weekly_locate_enable="YES"
```

- **Control tmp cleanup**

```
daily_clean_tmps_enable="YES"
```

```
daily_clean_tmps_days="3"
```

- **Security reports**

```
daily_status_security_inline="YES"
```

```
daily_status_security_output="mail"
```

To see all available options, use the command `man periodic.conf`

3.10.7.5 Discovering All Available Checks

By now you know `periodic` runs daily, weekly, and monthly jobs, but you may wonder: *what exactly are all these checks, and what do they do?*

There are several ways to explore them:

1. **List the scripts directly**

```
% ls /etc/periodic/daily
% ls /etc/periodic/weekly
% ls /etc/periodic/monthly
% ls /etc/periodic/security
```

You'll see files with names like `100.clean-disks` or `480.leapfile-ntpd`, the script names are descriptive and will give you an idea about what the script does. The numbers help control the order in which they run.

2. **Read the documentation**

The man pages `periodic(8)` and `periodic.conf(5)` explain many of the available scripts and their options. For example:

```
man periodic.conf
```

Gives you a summary of configuration variables and what they control.

3. **Check the script headers** Open any script in `/etc/periodic/*/` with `less` and read the first few comment lines. They usually contain a human-readable explanation of the script's purpose.

This means you never have to guess what `periodic` is doing; you can always inspect the scripts, preview their behaviour, or read the official documentation.

3.10.7.6 Why This Matters for Developers

For everyday users, periodic keeps the system tidy and secure without extra effort. But as a developer, you may later want to:

- Add a **custom periodic script** to test your driver or monitor its health once a day.
- Rotate or clean up custom log files created by your driver.
- Run automated integrity checks (e.g., verify your driver's device node exists and responds).

By hooking into periodic, you build on the same framework FreeBSD itself uses for its own housekeeping.

Hands-On Lab: Exploring and Customizing periodic

1. List the available daily scripts:

```
% ls /etc/periodic/daily
```

2. Run them manually:

```
% sudo periodic daily
```

3. Open `/etc/periodic.conf` (create it if it doesn't exist) and add:

```
weekly_locate_enable="YES"
```

4. Preview what the weekly jobs will do:

```
% sudo periodic weekly
```

5. Trigger the weekly jobs and then try:

```
% locate passwd
```

3.10.8 Hands-On Lab: Automating Tasks

1. Schedule a job to run every minute for testing:

```
% crontab -e
*/1 * * * * echo "Hello from cron: $(date)" >> /home/dev/cron_test.log
```

2. Wait a few minutes and check the file:

```
% tail -n 5 /home/dev/cron_test.log
```

3. Schedule a one-time job with at:

```
% echo "date >> /home/dev/at_test.log" | at now + 2 minutes
```

Check later:

```
% cat /home/dev/at_test.log
```

4. Run a periodic task manually:

```
% sudo periodic daily
```

You'll see reports on log files, security, and system status.

3.10.9 Common Pitfalls for Beginners

- Forgetting to set **full paths**. Cron jobs don't use the same environment as your shell, so always use full paths (/usr/bin/ls instead of just ls).
- Forgetting to redirect output. If you don't, results may be mailed to you silently.
- Overlapping jobs. Be careful not to schedule jobs that conflict or run too frequently.

3.10.10 Why This Matters for Driver Developers

You might wonder why we're spending time on cron jobs and scheduled tasks. The answer is that automation is **a developer's best friend**. When you begin writing device drivers, you'll often want to:

- Schedule automatic tests of your driver (for example, checking whether it loads and unloads cleanly every night).
- Rotate and archive kernel logs to keep track of driver behavior over time.
- Run periodic diagnostics that interact with your driver's /dev node and record results for analysis.

By mastering cron and periodic now, you'll already know how to set up these background routines later, saving yourself time and catching bugs early.

3.10.11 Wrapping Up

In this section, you learned how FreeBSD automates tasks using three main tools:

- **cron** for recurring jobs,
- **at** for one-time scheduling,
- **periodic** for built-in system maintenance.

You practiced creating jobs, checked their output, and learned how FreeBSD itself relies on automation to stay healthy.

Automation is useful, but sometimes you need to go beyond fixed schedules. You might want to chain commands together, use loops, or add logic to decide what happens. That's where **shell scripting** comes in. In the next section, we'll write your first scripts and see how to create custom automation tailored to your needs.

3.11 Introduction to Shell Scripting

You have learned to run commands one by one. Shell scripting lets you **save those commands into a reusable program**. On FreeBSD, the native and recommended shell for scripting is **/bin/sh**. This shell follows the POSIX standard and is available on every FreeBSD system.

Important note for Linux users On many Linux distributions, examples use **bash**. On FreeBSD, **bash is not part of the base system**. You can install it with `pkg install bash`, where it will live under `/usr/local/bin/bash`. For portable, dependency-free scripts on FreeBSD, use `#!/bin/sh`.

We will build this section progressively: shebang and execution, variables and quoting, conditions, loops, functions, working with files, return codes, and basic debugging. Every example script below is **fully commented** so a complete beginner can follow it.

3.11.1 1) Your first script: shebang, make it executable, run it

Create a file named `hello.sh`:

```
#!/bin/sh
# hello.sh  a first shell script using FreeBSD's native /bin/sh
# Print a friendly message with the current date and the active user.

# 'date' prints the current date and time
# 'whoami' prints the current user
echo "Hello from FreeBSD!"
echo "Date: $(date)"
echo "User: $(whoami)"
```

Tip: What Does `#!` (Shebang) Mean?

The first line of this script is:

```
#!/bin/sh
```

This is called the **shebang line**. The two characters `#!` tell the system *which program should interpret the script*.

- `#!/bin/sh` means: “run this script using the **sh** shell.”
- On other systems you might also see `#!/bin/tcsh`, `#!/usr/bin/env python3`, or `#!/usr/bin/env bash`.

When you make a script executable and run it, the system looks at this line to decide which interpreter to use. Without it, the script may fail or behave differently depending on your login shell.

Rule of thumb: Always include a shebang line at the top of your scripts. On FreeBSD, `#!/bin/sh` is the safest and most portable choice.

Now let’s make the script executable and run it:

```
% chmod +x hello.sh      # give the user execute permission
% ./hello.sh              # run it from the current directory
```

If you get “Permission denied”, you forgot `chmod +x`. If you get “Command not found”, you probably typed `hello.sh` without `./` and the current directory is not included in system `PATH`.

Tip: Don’t feel pressured to master all scripting features at once. Start small, write a 2-3 line script that prints your username and the date. Once you’re comfortable, add conditions (`if`), then loops, then functions. Shell scripting is like LEGO: build one block at a time.

3.11.2 2) Variables and quoting

Shell variables are untyped strings. Assign with `name=value` and reference with `$name`. There must be **no spaces** around `=`.

```
#!/bin/sh
# vars.sh  demonstrate variables and proper quoting

name="dev"
greeting="welcome"
# Double quotes preserve spaces and expand variables.
echo "$greeting, $name"
# Single quotes prevent expansion. This prints the literal characters.
echo '$greeting, $name'

# Command substitution captures output of a command.
today="$(date +%Y-%m-%d)"
echo "Today is $today"
```

Common beginner pitfalls:

- Using spaces around `=`: `name = dev` is an error.
- Forgetting quotes when variables may contain spaces. Use `"${var}"` as a habit.

3.11.3 3) Exit status and short circuit operators

Every command returns an **exit status**. Zero means success. Nonzero means error. The shell lets you chain commands using `&&` and `||`.

```
#!/bin/sh
# status.sh  show exit codes and conditional chaining

# Try to list a directory that exists. 'ls' should return 0.
ls /etc && echo "Listing /etc succeeded"
```

```
# Try something that fails. 'false' always returns nonzero.
false || echo "Previous command failed, so this message appears"

# You can test the last status explicitly using $?
echo "Last status was $?"
```

3.11.4 4) Tests and conditions: if, [], files and numbers

Use if with the test command or its bracket form [...]. There must be spaces inside the brackets.

```
#!/bin/sh
# ifs.sh demonstrate file and numeric tests

file="/etc/rc.conf"

# -f tests if a regular file exists
if [ -f "$file" ]; then
    echo "$file exists"
else
    echo "$file does not exist"
fi

num=5
if [ "$num" -gt 3 ]; then
    echo "$num is greater than 3"
fi

# String tests
user="$(whoami)"
if [ "$user" = "root" ]; then
    echo "You are root"
else
    echo "You are $user"
fi
```

Useful file tests:

- -e exists
- -f regular file
- -d directory
- -r readable
- -w writable
- -x executable

Numeric comparisons:

- -eq equal
- -ne not equal
- -gt greater than
- -ge greater or equal
- -lt less than
- -le less or equal

3.11.5 5) Loops: for and while

Loops let you repeat work over files or lines of input.

```
#!/bin/sh
# loops.sh   for and while loops in /bin/sh

# A 'for' loop over pathnames. Always quote expansions to handle spaces safely.
for f in /etc/*.conf; do
    echo "Found conf file: $f"
done

# A 'while' loop to read lines from a file safely.
# The 'IFS=' and 'read -r' avoid trimming spaces and backslash escapes.
count=0
while IFS= read -r line; do
    count=$((count + 1))
done < /etc/hosts
echo "The /etc/hosts file has $count lines"
```

Arithmetic in POSIX sh uses `$(( ... ))` for simple integer math.

3.11.6 6) Case statements for tidy branching

case is great when you have several patterns to match.

```
#!/bin/sh
# case.sh   handle options with a case statement

action="$1"   # first command line argument

case "$action" in
    start)
        echo "Starting service"
        ;;
```

```
stop)
    echo "Stopping service"
    ;;
restart)
    echo "Restarting service"
    ;;
*)
    echo "Usage: $0 {start|stop|restart}" >&2
    exit 2
    ;;
esac
```

3.11.7 7) Functions to organize your script

Functions keep code readable and reusable.

```
#!/bin/sh
# functions.sh - Demonstrates using functions and command-line arguments in a shell
↪ script.
#
# Usage:
# ./functions.sh NUM1 NUM2
# Example:
# ./functions.sh 5 7
# This will output: "[INFO] 5 + 7 = 12"
#
# A simple function to print informational messages
say() {
    # "$1" represents the first argument passed to the function
    echo "[INFO] $1"
}
#
# A function to sum two integers
sum() {
    # "$1" and "$2" are the first and second arguments
    local a="$1"
    local b="$2"
    # Perform arithmetic expansion to add them
    echo $((a + b))
}
# --- Main script execution starts here ---
```

```
# Make sure the user provided two arguments
if [ $# -ne 2 ]; then
    echo "Usage: $0 NUM1 NUM2"
    exit 1
fi

say "Beginning work"

# Call the sum() function with the provided arguments
result="$(sum "$1" "$2")"

# Print the result in a nice format
say "$1 + $2 = $result"
```

3.11.8 8) A practical example: a tiny backup script

This script creates a timestamped archive of a directory into ~/backups. It uses only base utilities available on FreeBSD Base System.

```
#!/bin/sh
# backup.sh  create a timestamped tar archive of a directory
# Usage: ./backup.sh /path/to/source
# Notes:
# - Uses /bin/sh so it runs on a clean FreeBSD 14.x install.
# - Creates ~/backups if it does not exist.
# - Names the archive sourcebasename-YYYYMMDD-HHMMSS.tar.gz

set -eu
# set -e: exit immediately if any command fails
# set -u: treat use of unset variables as an error

# Validate input
if [ $# -ne 1 ]; then
    echo "Usage: $0 /path/to/source" >&2
    exit 2
fi

src="$1"

# Verify that source is a directory
if [ ! -d "$src" ]; then
    echo "Error: $src is not a directory" >&2
    exit 3
fi
```

```
# Prepare destination directory
dest="${HOME}/backups"
mkdir -p "$dest"

# Build a safe archive name using only the last path component
base="$(basename "$src")"
stamp="$(date +%Y%m%d-%H%M%S)"
archive="${dest}/${base}-${stamp}.tar.gz"

# Create the archive
# tar(1) is in the base system. The flags mean:
# - c: create - z: gzip - f: file name - C: change to directory
tar -czf "$archive" -C "$(dirname "$src")" "$base"

echo "Backup created: $archive"
```

Run it:

```
% chmod +x backup.sh
% ./backup.sh ~/directory_you_want_to_backup
```

You will find the archive under ~/backups.

3.11.9 9) Working with temporary files safely

Never hardcode names like /tmp/tmpfile. Use `mktemp(1)` from the base system.

```
#!/bin/sh
# tmp_demo.sh  create and clean a temporary file safely

set -eu

tmpfile="$(mktemp -t myscript)"
# Arrange cleanup on exit for success or error
cleanup() {
    [ -f "$tmpfile" ] && rm -f "$tmpfile"
}
trap cleanup EXIT

echo "Temporary file is $tmpfile"
echo "Hello temp" > "$tmpfile"
echo "Contents: $(cat "$tmpfile")"
```

trap schedules a function to run when the script exits, which prevents stale files.

3.11.10 10) Debugging your scripts

- `set -x` prints each command before executing it. Add it near the top and remove once fixed.
- `echo` progress messages so the user knows what is happening.
- Check return codes and handle failures explicitly.
- Log to a file by redirecting output: `mycmd >> ~/my.log 2>&1`.

Example:

```
#!/bin/sh
# debug_demo.sh    show simple tracing

# set -x comment to disable verbose trace:
set -x

echo "Step 1"
ls /etc >/dev/null

echo "Step 2"
date
```

3.11.11 11) Putting it together: organize downloads by type

This small utility sorts files in `~/Downloads` into subfolders by extension. It demonstrates loops, case, tests and safety checks.

```
#!/bin/sh
# organize_downloads.sh - Tidy ~/Downloads by file extension
#
# Usage:
#   ./organize_downloads.sh
#
# Creates subdirectories like Documents, Images, Audio, Video, Archives, Other
# and moves matched files into them safely.

set -eu

downloads="${HOME}/Downloads"

# Create a temporary file to store the list of files
tmpfile=$(mktemp)
```

```
# Remove temporary file when script exits (normal or error)
trap 'rm -f "$tmpfile"' EXIT

# Ensure the Downloads directory exists
if [ ! -d "$downloads" ]; then
    echo "Downloads directory not found at $downloads" >&2
    exit 1
fi

cd "$downloads"

# Create target folders if missing
mkdir -p Documents Images Audio Video Archives Other

# Find all regular files in current directory (non-recursive, excluding hidden files)
# -maxdepth 1: don't search in subdirectories
# -type f: only regular files (not directories or symlinks)
# ! -name ".*": exclude hidden files (those starting with a dot)
count=0
find . -maxdepth 1 -type f ! -name "." > "$tmpfile"
while IFS= read -r f; do
    # Strip leading "./" from path
    fname=${f#./}

    # Skip if filename is empty (shouldn't happen, but safety check)
    [ -z "$fname" ] && continue

    # Convert filename extension to lowercase for matching
    lower=$(printf '%s' "$fname" | tr '[:upper:]' '[:lower:]')

    case "$lower" in
        *.pdf|*.txt|*.md|*.doc|*.docx) dest="Documents" ;;
        *.png|*.jpg|*.jpeg|*.gif|*.bmp) dest="Images" ;;
        *.mp3|*.wav|*.flac)             dest="Audio" ;;
        *.mp4|*.mkv|*.mov|*.avi)        dest="Video" ;;
        *.zip|*.tar|*.gz|*.tgz|*.bz2)    dest="Archives" ;;
        *)                               dest="Other" ;;
    esac

    echo "Moving '$fname' -> $dest/"
    mv -n -- "$fname" "$dest/" # -n prevents overwriting existing files
    count=$((count + 1))      # Increment the counter
done < "$tmpfile"            # Feed the temporary file into the while loop
```

```
if [ $count -eq 0 ]; then
    echo "No files to organize."
else
    echo "Done. Organized $count file(s)."
fi
```

3.11.12 Hands-on Lab: three mini tasks

1. **Write a logger** Create `logger.sh` that appends a timestamped line to `~/activity.log` with the current directory and user. Run it, then view the log with `tail`.
2. **Check disk space** Create `check_disk.sh` that warns when root filesystem usage exceeds 80 percent. Use `df -h /` and parse the percentage with `${var%%}` style trimming or a simple `awk`. Exit with status 1 if above the threshold so cron can alert you.
3. **Wrap your backup** Create `backup_cron.sh` that calls `backup.sh` from earlier and logs output to `~/backup.log`. Add a crontab entry to run it daily at 3 AM. Remember to use full paths inside the script.

All scripts should start with `#!/bin/sh`, contain comments explaining each step, use quotes around variable expansions, and handle errors where sensible.

3.11.13 Common beginner pitfalls and how to avoid them

- **Using bash features in `#!/bin/sh` scripts.** Stick to POSIX constructs. If you require bash, say so in the shebang and remember it is under `/usr/local/bin/bash` on FreeBSD.
- **Forgetting to quote variables.** Use `"${var}"` to prevent word splitting and globbing surprises.
- **Assuming the same environment under cron.** Always use full paths and redirect output to a log file.
- **Hardcoding temporary file names.** Use `mktemp` and `trap` to clean up.
- **Spaces around = in assignments.** `name=value` is correct. `name = value` is not.

3.11.14 Wrapping up

In this section, you learned the **native FreeBSD way** to automate work with portable scripts that run on any clean FreeBSD install. You can now write small programs with `/bin/sh`, handle arguments, test conditions, loop through files, define functions, use temporary files safely, and debug issues with simple tools. While writing drivers, scripts will help you repeat tests, gather logs, and package builds reliably.

Before we move on, a reminder: you do not need to memorise every construct or command option. Part of being productive in UNIX is knowing where to **find the right information at the right time**.

In the next section, we will tighten the screws on **portability** itself, looking at subtle differences between shells, the habits that keep scripts robust across systems, and how to choose features that will not surprise you later.

3.12 Shell Portability: Handling Edge Cases and `bash` vs `sh`

So far, we've written scripts using FreeBSD's native `/bin/sh` shell, which follows the POSIX standard. This makes our scripts portable across different UNIX systems. But as you explore shell scripting examples online

or receive contributions from other developers, you'll encounter scripts written for **bash** that use features not available in POSIX **sh**.

Understanding the differences between **bash** and **sh**, and knowing how to handle edge cases like unusual filenames, will help you write robust scripts and decide when portability matters more than convenience.

3.12.1 The Problem: Filenames with Special Characters

UNIX allows filenames to contain almost any character except forward slash / (which separates directories) and the null character `\0`. This means a filename can legally contain spaces, newlines, tabs, or other surprising characters.

Let's create a file with a newline in its name to see how this affects our scripts:

```
% cd ~
% touch '$file_with\nnewline.txt'
% ls
file_with?newline.txt
```

The `?` appears because `ls` substitutes unprintable characters when displaying filenames. The actual filename contains:

```
file_with
newline.txt
```

Now let's see what happens when a script tries to process this file.

3.12.2 A Naive Approach That Breaks

Here's a simple script that lists files in your home directory:

```
#!/bin/sh
# list_files.sh - count files in home directory

set -eu
cd "${HOME}"

count=0
while IFS= read -r f; do
    fname=${f#./}
    echo "File found: '$fname'"
    count=$((count + 1))
done <& EOF
$(find . -maxdepth 1 -type f ! -name ".*" -print)
```


EOF

```
echo "Total files found: $count"
```

Running this script with our unusual filename produces incorrect results:

```
% ./list_files.sh
File found: 'file_with'
File found: 'newline.txt'
Total files found: 2
```

The script thinks one file is actually two files because `find -print` outputs one path per line, and our filename contains a newline character. The script breaks on a perfectly valid UNIX filename.

3.12.3 The bash Solution: Using Null Delimiters

One way to fix this is to use null characters (`\0`) as delimiters instead of newlines. Bash supports this with the `-d` option to the `read` command:

```
#!/usr/local/bin/bash
# list_files_bash.sh - correctly handle unusual filenames with bash

set -eu
cd "${HOME}"

count=0
while IFS= read -r -d '' f; do
    fname=${f#./}
    echo "File found: '$fname'"
    count=$((count + 1))
done << (find . -maxdepth 1 -type f ! -name ".*" -print0)

echo "Total files found: $count"
```

Notice two changes:

1. **Shebang:** Changed to `#!/usr/local/bin/bash` (bash's location on FreeBSD after `pkg install bash`)
2. **find flag:** Changed from `-print` to `-print0` (outputs null-delimited paths)
3. **read option:** Added `-d ''` to tell `read` to use null as the delimiter

This version works correctly:

```
% ./list_files_bash.sh
File found: 'file_with
newline.txt'
Total files found: 1
```

The downside? **This script now requires bash**, which is not part of the FreeBSD base system. It creates a dependency.

3.12.4 The POSIX-Compliant Alternative

If portability matters more than handling every possible edge case, we can write a POSIX-compliant version that avoids bash-specific features:

```
#!/bin/sh
# list_files_posix.sh - POSIX-compliant file listing

set -eu
cd "${HOME}"

# Use a temporary file instead of a pipe
tmpfile=$(mktemp)
trap 'rm -f "$tmpfile"' EXIT

# Store find results in temporary file
find . -maxdepth 1 -type f ! -name "." > "$tmpfile"

count=0
while IFS= read -r f; do
    fname=${f#./}
    [ -z "$fname" ] && continue

    echo "File found: '$fname'"
    count=$((count + 1))
done < "$tmpfile"

echo "Total files found: $count"
```

This version:

- Works on any POSIX-compliant shell (no bash required)
- Uses a temporary file instead of a pipe to avoid subshell variable issues
- Cleans up automatically with `trap`
- Handles filenames with spaces and most special characters

The limitation? It still can't handle filenames with newlines correctly, because POSIX sh's read command has no way to use a different delimiter. For this version:

```
% ./list_files_posix.sh
File found: 'file_with'
File found: 'newline.txt'
Total files found: 2
```

3.12.5 Understanding the Trade-Off

This reveals an important decision point in shell scripting:

Portability vs Edge Case Coverage

Approach	Pros	Cons
POSIX sh	Runs everywhere, no dependencies	Cannot handle filenames with newlines
bash with -d	Handles all valid filenames	Requires bash installation
find -exec	POSIX-compliant, handles everything	More complex syntax

For most real-world scripting, the POSIX approach is sufficient. Filenames with newlines are extremely rare outside of contrived examples or security exploits. Files with spaces, unicode characters, and other printable characters work fine with the POSIX version.

3.12.6 When to Choose bash

Use bash when:

- You're writing personal tools where bash is guaranteed to be available
- You genuinely need to handle filenames with newlines (very rare)
- You need bash-specific features like arrays, extended regex, or advanced string manipulation
- The script is part of a project that already depends on bash

Use POSIX sh when:

- Writing system administration scripts that need to run on any FreeBSD system
- Contributing to FreeBSD base system scripts
- Maximum portability is required
- The script might run in rescue mode or minimal environments

3.12.7 A Third Option: find -exec

For completeness, here's a POSIX-compliant approach that handles all filenames correctly without requiring bash:

```
#!/bin/sh
# list_files_exec.sh - handle all filenames using find -exec

set -eu
cd "${HOME}"

find . -maxdepth 1 -type f ! -name ".*" -exec sh -c '
    for f; do
        fname=${f#./}
        printf "File found: \'\'%s\'\'\'\'n" "$fname"
    done
' sh {} +
```

This works because `find -exec` passes filenames as arguments, not through pipes or line-based reading. It's POSIX-compliant and handles every edge case, but the syntax is less intuitive for beginners.

3.12.8 Practical Advice

When writing shell scripts:

1. **Start with `/bin/sh`** - Begin with POSIX-compliant scripts
2. **Quote your variables** - Always use `"$var"` to handle spaces
3. **Test with unusual filenames** - Create test files with spaces in their names
4. **Document dependencies** - If you use `bash`, state it clearly in comments
5. **Accept reasonable limitations** - Don't sacrifice portability for edge cases you'll never encounter

The `organize_downloads.sh` script we wrote earlier uses the POSIX-compliant temporary file approach. It handles the vast majority of real-world filenames correctly while remaining portable across any FreeBSD system.

Remember: **the best script is one that works reliably in your target environment**. Don't add `bash` as a dependency for edge cases you'll never hit, but don't torture yourself with POSIX constraints if you're writing personal tools on a system where `bash` is already installed.

3.12.9 Wrapping Up

You have now seen why portability is a design choice, not an afterthought. You learned to favour POSIX `/bin/sh` for scripts that must run everywhere on FreeBSD, to avoid `bash`-only features, to use `printf` over a loose `echo`, to quote variables by default, to check exit codes, and to pick a clear shebang so the right interpreter runs your code. Along the way, we revisited the building blocks you already know: arguments, conditionals, loops, functions, and safe temporary files and tuned them for predictable behaviour across systems.

No one keeps all the details in their head, and you do not need to. The next section shows you where to **look things up the FreeBSD way**: manual pages, apropos, built-in help, the FreeBSD Handbook, and community resources. These will become your day-to-day companions as you go deeper into device driver development.

3.13 Seeking Help and Documentation in FreeBSD

No one, not even the most experienced developer, remembers every command, option, or system call. The real strength of a UNIX system like FreeBSD is that it ships with **excellent documentation** and has a supportive community that can help when you get stuck.

In this section, we'll explore the main ways to get information: **man pages, the FreeBSD Handbook, online resources, and the community**. By the end, you'll know exactly where to look when you have a question, whether it's about using `ls` or writing a device driver.

3.13.1 The Power of man Pages

The **manual pages**, or **man pages**, are the built-in reference system for UNIX. Every command, system call, library function, configuration file, and kernel interface has a man page.

You read them with the `man` command, for example:

```
% man ls
```

This opens the documentation for `ls`, the command to list directory contents. Use the spacebar to scroll, `q` to quit.

3.13.1.1 Man Page Sections

FreeBSD organizes man pages into numbered sections. The same name may exist in multiple sections, so you specify which one you want.

- **1** - User commands (e.g., `ls`, `cp`, `ps`)
- **2** - System calls (e.g., `open(2)`, `write(2)`)
- **3** - Library functions (C standard library, math functions)
- **4** - Device drivers and special files (e.g., `null(4)`, `random(4)`)
- **5** - File formats and conventions (`passwd(5)`, `rc.conf(5)`)
- **7** - Miscellaneous (protocols, conventions)
- **8** - System administration commands (e.g., `ifconfig(8)`, `shutdown(8)`)
- **9** - Kernel developer interfaces (critical for driver writers!)

Example:

```
% man 2 open      # system call open()
% man 9 bus_space # kernel function for accessing device registers
```

3.13.1.2 Man Section 9: The Kernel Developer's Manual

Most FreeBSD users live in section **1** (user commands) and administrators spend a lot of time in section **8** (system management). But as a driver developer, you'll spend much of your time in **section 9**.

Section 9 contains the **kernel developer interfaces** documentation for functions, macros, and subsystems that are only available inside the kernel.

Some examples:

- `man 9 device` - Overview of device driver interfaces.
- `man 9 bus_space` - Accessing hardware registers.
- `man 9 mutex` - Synchronization primitives for the kernel.
- `man 9 taskqueue` - Scheduling deferred work in the kernel.
- `man 9 malloc` - Memory allocation inside the kernel.

Unlike section 2 (system calls) or section 3 (libraries), these are **not available in user space**. They are part of the kernel itself, and you'll use them when writing drivers and kernel modules.

Think of section 9 as the **developer's API manual for the FreeBSD kernel**.

3.13.1.3 Hands-On Preview

You don't need to understand all the details yet, but you can take a peek:

```
% man 9 device
% man 9 bus_dma
% man 9 sysctl
```

You'll see that the style is different from user command man pages: these are focused on **kernel functions, structures, and usage examples**.

Later in this book, we'll constantly refer to section 9 as we introduce new kernel features. Consider it your most important companion for the road ahead.

3.13.1.4 Searching the man Pages

If you don't know the exact command name, use the `-k` flag (equivalent to `apropos`):

```
man -k network
```

This shows every man page related to networking.

Another example:

```
man -k disk | less
```

This will show you tools, drivers, and system calls related to disks.

3.13.2 The FreeBSD Handbook

The **FreeBSD Handbook** is the official, comprehensive guide to the operating system.

You can read it online:

<https://docs.freebsd.org/en/books/handbook/>

The Handbook covers:

- Installing FreeBSD
- System administration
- Networking
- Storage and filesystems
- Security and jails
- Advanced topics

The Handbook is an **excellent complement to this book**. While we focus on device driver development, the Handbook gives you broad system knowledge that you can always come back to.

3.13.2.1 Other Documentation

- **Online man pages:** <https://man.freebsd.org>
- **FreeBSD Wiki:** <https://wiki.freebsd.org> (community-maintained notes, HOWTOs, and work-in-progress documentation).
- **Developer's Handbook:** <https://docs.freebsd.org/en/books/developers-handbook> is aimed at programmers.
- **Porter's Handbook:** <https://docs.freebsd.org/en/books/porters-handbook> if you package software for FreeBSD.

3.13.3 Community and Support

Documentation will get you far, but sometimes you need to talk to real people. FreeBSD has an active and welcoming community.

- **Mailing lists:** <https://lists.freebsd.org>
 - `freebsd-questions@` is for general user help.
 - `freebsd-hackers@` is for development discussions.
 - `freebsd-drivers@` is specific to device driver development.
- **FreeBSD Forums:** <https://forums.freebsd.org> a friendly and beginner-friendly place to ask questions.
- **User Groups:**
 - Around the world, there are **FreeBSD and BSD user groups** that organize meetups, talks, and workshops.
 - Examples include *NYCBUG (New York City BSD User Group)*, *BAFUG (Bay Area FreeBSD User Group)*, and many university-based clubs.
 - You can usually find them via the FreeBSD Wiki, local tech mailing lists, or [meetup.com](https://www.meetup.com).

- If you don't find one nearby, consider starting a small group, even a handful of enthusiasts meeting online or in person can become a valuable support network.
- **Chat:**
 - **IRC** on Libera.Chat (#freebsd).
 - **Discord** communities exist and are pretty active, use this link to join: <https://discord.com/invite/freebsd>
- **Reddit:** <https://reddit.com/r/freebsd>

User groups and forums are especially valuable because you can often ask questions in your native language, or even meet people who are contributing to FreeBSD in your area.

3.13.3.1 How to Ask for Help

At some point, everyone gets stuck. One of FreeBSD's strengths is its active and supportive community, but to get useful answers, you need to ask clear, complete, and respectful questions.

When you post to a mailing list, forum, IRC, or Discord channel, include:

- **Your FreeBSD version** Run:

```
% uname -a
```

This tells helpers exactly which release, patch level, and architecture you are using.

- **What you were trying to do** Describe your goal, not just the command that failed. Helpers can sometimes suggest a better approach than the one you attempted.
- **The exact error messages** Copy and paste the error text instead of paraphrasing it. Even small differences matter.
- **Steps to reproduce the problem** If someone else can repeat your issue, they can often solve it much faster.
- **What you already tried** Mention commands, configuration changes, or documentation you consulted. This shows you've made an effort and prevents people from suggesting things you already did.

3.13.3.2 Example of a Poor Help Request

```
"Ports aren't working, how do I fix it?"
```

This leaves out version, commands, errors, and context. Nobody can answer without guessing.

3.13.3.3 Example of a Good Help Request

```
"I'm running FreeBSD 14.3-RELEASE on amd64. I tried building htop from ports with cd /usr/ports/sysutils/htop && make install clean. The build failed with the error:
```

```
error: ncurses.h: No such file or directory
```


I already tried `pkg install ncurses`, but the error remains. What should I check next?”

This is short but complete; version, command, error, and troubleshooting steps are all there.

Tip: Always stay polite and patient. Remember, most FreeBSD contributors are **volunteers**. A clear, respectful question not only increases your chances of a helpful reply, but also builds goodwill in the community.

3.13.4 Hands-On Lab: Exploring Documentation

1. Open the man page for `ls`. Find and try at least two options you didn’t know.

```
% man ls
```

2. Use `man -k` to search for commands related to disks.

```
% man -k disk | less
```

3. Open the man page for `open(2)` and compare it to `open(3)`. What’s the difference?
4. Peek into the kernel developer documentation:

```
% man 9 device
```

5. Visit <https://docs.freebsd.org/> and find the page on system startup (`rc.d`). Compare it to `man rc.conf`.

3.13.5 Wrapping Up

FreeBSD gives you solid tools to teach yourself. The **man pages** are your first stop; they’re always on your system, always up to date, and cover everything from basic commands to kernel APIs. The **Handbook** is your big-picture guide, and the **community** mailing lists, forums, user groups, and online chat are there to help when you need human answers.

Later, as you write drivers, you’ll rely heavily on man pages (especially section 9) and on discussions in FreeBSD mailing lists and forums. Knowing how to find information is just as important as memorizing commands.

Next, we will look inside the system to **peek at kernel messages and tunables**. Tools like `dmesg` and `sysctl` let you see what the kernel is doing and will become essential when you start loading and testing your own device drivers.

3.14 Peeking into the Kernel and System State

At this point, you know how to move around in FreeBSD, manage files, control processes, and even write scripts. That makes you a capable user. But writing drivers means stepping into the **mind of the kernel**. You’ll need to see what FreeBSD itself sees:

- What hardware was detected?
- Which drivers were loaded?
- What tunable knobs exist inside the kernel?

- How do devices appear to the operating system?

FreeBSD gives you **three magical windows into the kernel's state**:

1. **dmesg** - the kernel's diary.
2. **sysctl** - the control panel full of switches and meters.
3. **/dev** - the doorway where devices show up as files.

These three tools will become your **companions**. Every time you add or debug a driver, you'll use them. Let's look at them now, step by step.

3.14.1 dmesg: Reading the Kernel's Diary

Imagine FreeBSD as a pilot starting an airplane. As the system boots, the kernel checks its hardware: CPUs, memory, disks, USB devices and each driver reports back. Those messages are not lost; they're stored in a buffer you can read at any time with:

```
% dmesg | less
```

You'll see lines like:

```
Copyright (c) 1992-2023 The FreeBSD Project.
Copyright (c) 1979, 1980, 1983, 1986, 1988, 1989, 1991, 1992, 1993, 1994
    The Regents of the University of California. All rights reserved.
FreeBSD is a registered trademark of The FreeBSD Foundation.
FreeBSD 14.3-RELEASE releng/14.3-n271432-8c9ce319fef7 GENERIC amd64
FreeBSD clang version 19.1.7 (https://github.com/llvm/llvm-project.git
↳  llvmorg-19.1.7-0-gcd708029e0b2)
VT(vga): text 80x25
CPU: AMD Ryzen 7 5800U with Radeon Graphics          (1896.45-MHz K8-class CPU)
    Origin="AuthenticAMD" Id=0xa50f00 Family=0x19 Model=0x50 Stepping=0
    Fea-
↳   tures=0x1783fbff<FPU,VME,DE,PSE,TSC,MSR,PAE,MCE,CX8,APIC,SEP,MTRR,PGE,MCA,CMOV,PAT,PSE36,MMX,
    Fea-
↳   tures2=0xffff83203<SSE3,PCLMULQDQ,SSSE3,FMA,CX16,SSE4.1,SSE4.2,x2APIC,MOVBE,POPCNT,TSCDLT,AESNI
    AMD Features=0x2e500800<SYSCALL,NX,MMX+,FFXSR,Page1GB,RDTSCP,LM>
    AMD Features2=0x8003f7<LAHF,CMP,SVM,CR8,ABM,SSE4A,MAS,Prefetch,OSVW,PCXC>
    Structured Extended Fea-
↳   tures=0x219c07ab<FSGSBASE,TSCADJ,BMI1,AVX2,SMEP,BMI2,ERMS,INVPCID,RDSEED,ADX,SMAP,CLFLUSHOPT,
    Structured Extended Features2=0x40061c<UMIP,PKU,OSPKE,VAES,VPCLMULQDQ,RDPID>
    Structured Extended Features3=0xac000010<FSRM,IBPB,STIBP,ARCH_CAP,SSBD>
    XSAVE Features=0xf<XSAVEOPT,XSAVEC,XINUSE,XSAVES>
    IA32_ARCH_CAPS=0xc000069<RDCL_NO,SKIP_L1DFL_VME,MDS_NO>
    AMD Extended Feature Extensions ID
↳   EBX=0x1302d205<CLZERO,XSaveErPtr,WBNOINVD,IBPB,IBRS,STIBP,STIBP_ALWAYS_ON,SSBD,VIRT_SSBD,PSFD>
SVM: NP,NRIP,VClean,AFlush,NASids=16
```

```
...  
...
```

This is the kernel telling you:

- **what hardware it found,**
- **which driver claimed it,**
- and sometimes, **what went wrong.**

Later in this book, when you load your own driver, `dmesg` is where you'll look for your first "Hello, kernel!" message.

The output of `dmesg` can be very long, you can use `grep` to filter and see only what you need, for example:

```
% dmesg | grep ada
```

This will show only messages about disk devices (`ada0`, `ada1`).

3.14.2 sysctl: The Kernel's Control Panel

If `dmesg` is the diary, `sysctl` is the **dashboard full of knobs and meters**. It exposes thousands of kernel variables at runtime: some read-only (system info), others tunable (system behavior).

Try this commands:

```
% sysctl kern.ostype  
% sysctl kern.osrelease  
% sysctl hw.model  
% sysctl hw.ncpu
```

Output might look like:

```
kern.ostype: FreeBSD  
kern.osrelease: 14.3-RELEASE  
hw.model: AMD Ryzen 7 5800U with Radeon Graphics  
hw.ncpu: 8
```

Here you just asked the kernel:

- What OS am I running?
- What version?
- What CPU?
- How many cores?

3.14.2.1 Exploring Everything

To see all parameters that you can finetune with `sysctl`, you can run the command below:

```
% sysctl -a | less
```

This prints the **entire control panel** thousands of values. They are organized by categories:

- `kern.*` - kernel properties and settings.
- `hw.*` - hardware information.
- `net.*` - network stack details.
- `vfs.*` - filesystem settings.
- `debug.*` - debugging variables (often useful for developers).

It's overwhelming at first, but don't worry, you'll learn to fish out what matters.

3.14.2.2 Changing Values

Some `sysctls` are writable. For example:

```
% sudo sysctl kern.hostname=myfreebsd
% hostname
```

You've just changed your hostname at runtime.

Important: Changes made this way disappear after reboot unless saved in `/etc/sysctl.conf`.

3.14.3 /dev: Where Devices Come to Life

Now for the most exciting part.

FreeBSD represents devices as **special files** inside `/dev`. This is one of UNIX's most elegant ideas:

If everything is a file, then everything can be accessed in a consistent way.

Run:

```
% ls -ld /dev/* | less
```

You'll see a sea of names:

- `/dev/null` - the "black hole" where data goes to disappear.
- `/dev/zero` - an infinite stream of zeros.
- `/dev/random` - cryptographically secure random numbers.
- `/dev/tty` - your terminal.
- `/dev/ada0` - your SATA disk.

- /dev/da0 - a USB disk.

Try interacting:

```
echo "Testing" > /dev/null          # silently discards output
head -c 16 /dev/zero | hexdump      # shows zeros in hex
head -c 16 /dev/random | hexdump    # random bytes from the kernel
```

Later, when you create your first driver, it will show up here as a file named /dev/hello. Reading or writing to that file will trigger **your kernel code**. This is the moment when you'll feel the bridge between userland and the kernel.

3.14.4 Hands-On Lab: Your First Peek Inside

1. View all kernel messages:

```
% dmesg | less
```

2. Find your storage devices:

```
% dmesg | grep ada
```

3. Ask the kernel about your CPU:

```
% sysctl hw.model
% sysctl hw.ncpu
```

4. Change your hostname temporarily:

```
% sudo sysctl kern.hostname=mytesthost
% hostname
```

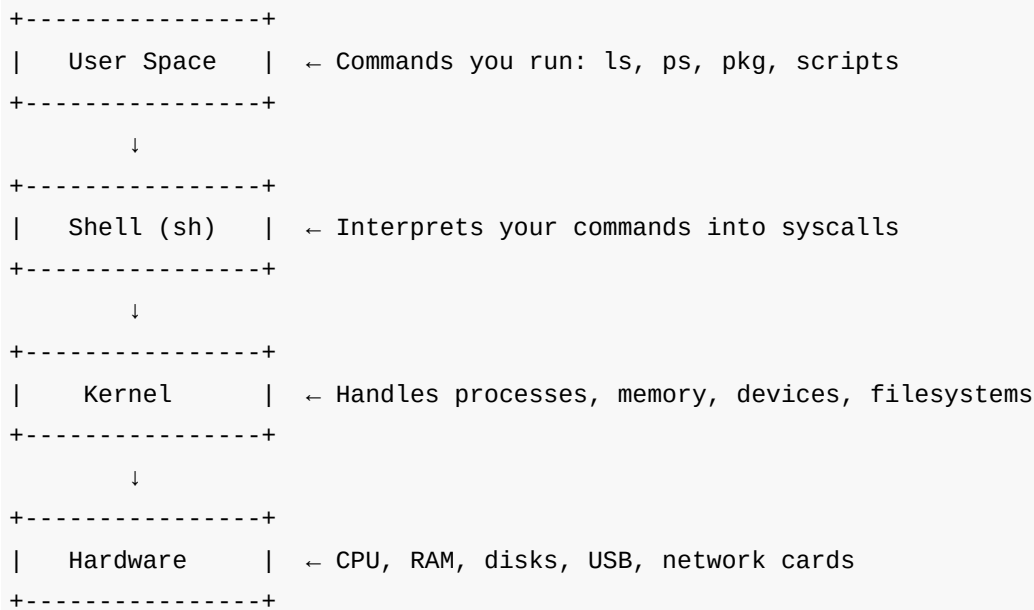
5. Interact with special device files:

```
% echo "Hello FreeBSD" > /dev/null
% head -c 8 /dev/zero | hexdump
% head -c 8 /dev/random | hexdump
```

With this short lab, you've already read kernel messages, queried kernel variables, and touched device nodes, exactly what professional developers do daily.

3.14.5 From Shell to Hardware: The Big Picture

To understand why tools like `dmesg`, `sysctl`, and `/dev` are so useful, it helps to picture how FreeBSD is layered:



Whenever you type a command in the shell, it travels down this stack:

- The **shell** interprets it.
- The **kernel** executes it by managing processes, memory, and devices.
- The **hardware** responds.

Then the results bubble back up for you to see.

Understanding this flow is essential for driver developers: when you interact with `/dev`, you're connecting directly to the kernel, which in turn talks to the hardware.

3.14.6 Common Beginner Pitfalls

Exploring the kernel can be exciting, but here are some common mistakes to watch out for:

1. Confusing `dmesg` with system logs

- `dmesg` only shows the kernel's ring buffer, not all logs.
- Old messages may disappear after new ones push them out.
- For complete logs, check `/var/log/messages`.

2. Forgetting that `sysctl` changes don't persist

- If you change a setting with `sysctl`, it resets at reboot.
- To make it permanent, add it to `/etc/sysctl.conf`.
- Example:

```
% echo 'kern.hostname="myhost"' | sudo tee -a /etc/sysctl.conf
```

3. Overwriting files in `/dev`

- `/dev` entries aren't normal files; they're live connections to the kernel.
- Redirecting output to them can have real effects.
- Writing to `/dev/null` is safe, but writing random data to `/dev/ada0` (your disk) could destroy it.
- Rule of thumb: explore `/dev/null`, `/dev/zero`, `/dev/random`, and `/dev/tty`, but leave storage devices (`ada0`, `da0`) alone unless you know exactly what you're doing.

4. Expecting `/dev` entries to stay the same

- Devices appear and disappear as hardware is added or removed.
- For example, plugging in a USB stick may create `/dev/da0`.
- Don't hardcode device names into scripts without verifying.

5. Not using full paths in automation

- Cron and other automated tools may not have the same `PATH` as your shell.
- Always use full paths (`/sbin/sysctl`, `/bin/echo`) when scripting kernel interactions.

3.14.7 Wrapping Up

In this section, you opened three magical windows into FreeBSD's kernel:

- `dmesg`- the diary of the system, recording hardware detection and driver messages.
- `sysctl` - the control panel that reveals (and sometimes adjusts) kernel settings.
- `/dev`- the place where devices come to life as files.

The **big picture** to remember is this: whenever you type a command, it travels through the shell, down into the kernel, and finally to the hardware. The results then bubble back up for you to see. Tools like `dmesg`, `sysctl`, and `/dev` let you peek into that flow and see what the kernel is doing behind the scenes.

These aren't just abstract tools; they're exactly how you'll see your **own driver** appear in the system. When you load your module, you'll watch `dmesg` light up, you may expose a knob with `sysctl`, and you'll interact with your device node under `/dev`.

It is worth pausing for a moment on what this tells you about the road ahead. Every line of `dmesg` that describes hardware attaching, every `sysctl` name that starts with `kern.` or `vm.`, and every file under `/dev` is the visible face of kernel code written in C. When you ran `dmesg`, you were reading strings that a driver passed to `device_printf` or `printf` during attach. When you walked `sysctl -a`, you were traversing a tree that drivers and subsystems populate with `SYSCTL_INT`, `SYSCTL_ULONG`, and related macros. When you opened `/dev/null`, the kernel dispatched your `read` or `write` into a driver whose structure you will meet in Chapter 6. You have been looking at the outputs of driver code all along; the next two chapters teach you to write the inputs.

Chapter 5 takes you into the **C language as the kernel actually uses it**: fixed-width integer types, explicit memory management with `malloc(9)` and `free(9)`, pointer discipline under interrupt context, and the subset of C that FreeBSD's Kernel Normal Form (KNF) considers idiomatic. It is not "just C again." The kernel cannot call `printf` from `libc`, cannot allocate with plain `malloc`, and cannot assume the usual userland safety nets. Chapter 5 shows you what changes and why, so that the code you write later compiles, loads, and behaves predictably.

Chapter 6 then takes the C you have just relearned and assembles it into a first driver, stepping through the **anatomy of a FreeBSD driver** piece by piece: the `softc` structure, the Newbus method table, the

`DRIVER_MODULE` macro, and the path a `read` or `write` takes from your `/dev/foo` device node to the driver routine that services it. By the end of Chapter 6 you will have loaded your own module, seen it announce itself in `dmesg`, and used the same commands you practised in this chapter to confirm that it works.

Before we move on and start learning about C programming, let's pause to consolidate everything you've learned in this chapter. In the next section, we'll review the key ideas and give you a set of challenges to practice, exercises that will help lock in these new skills and prepare you for the work ahead.

3.15 Wrapping Up

Congratulations! You've just taken your **first guided tour through UNIX and FreeBSD**. What started as abstract ideas is now becoming practical skills. You can move around the system, manage files, edit and install software, control processes, automate tasks, and even peek into the kernel's inner workings.

Let's take a moment to review what you've accomplished in this chapter:

- **What UNIX is and why it matters** - A philosophy of simplicity, modularity, and "everything is a file," inherited by FreeBSD.
- **The Shell** - Your window into the system, where commands follow the consistent structure of `command [options] [arguments]`.
- **Filesystem Layout** - A single hierarchy starting at `/`, with special roles for directories like `/etc`, `/usr/local`, `/var`, and `/dev`.
- **Users, Groups, and Permissions** - The foundation of FreeBSD's security model, controlling who can read, write, or execute.
- **Processes** - Programs in motion, with tools like `ps`, `top`, and `kill` to manage them.
- **Installing Software** - Using `pkg` for quick binary installs, and the **Ports Collection** for source-based flexibility.
- **Automation** - Scheduling tasks with `cron`, one-time jobs with `at`, and maintenance with `periodic`.
- **Shell Scripting** - Turning repetitive commands into reusable programs using FreeBSD's native `/bin/sh`.
- **Peeking into the Kernel** - Using `dmesg`, `sysctl`, and `/dev` to observe the system at a deeper level.

That's a lot, but don't worry if you don't feel like an expert yet. The goal of this chapter was not perfection, but **comfort**: comfort in the shell, comfort in exploring FreeBSD, and comfort in seeing how UNIX works under the hood. That comfort will carry forward as we begin writing real code for the system.

3.15.1 Practice Ground

If you would like a hands-on way to reinforce what you just read, the next pages collect **46 optional exercises**. None of them are required to continue with the book, so treat them as extras: pick the ones that cover ground where you still feel unsure, skip the ones that feel redundant, and come back to the rest later if they become useful.

They are grouped by topic, so you can practice section by section or mix them up as you like.

3.15.2 Filesystem and Navigation (8 exercises)

1. Use `pwd` to confirm your current directory, then move into `/etc` and back into your home directory using `cd`.
2. Create a directory `unix_playground` in your home. Inside it, create three subdirectories: `docs`, `code`, and `tmp`.
3. Inside `unix_playground/docs`, create a file called `readme.txt` with the text “Welcome to FreeBSD”. Use `echo` and output redirection.
4. Copy `readme.txt` into the `tmp` directory. Verify both files exist with `ls -l`.
5. Rename the file in `tmp` to `copy.txt`. Then delete it with `rm`.
6. Use `find` to locate every `.conf` file inside `/etc`.
7. Use absolute paths to copy `/etc/hosts` into your `docs` directory. Then use relative paths to move it into `tmp`.
8. Use `ls -lh` to display file sizes in human-readable format. Which file in `/etc` is the largest?

3.15.3 Users, Groups, and Permissions (6 exercises)

1. Create a file called `secret.txt` in your home directory. Make it readable only by you.
2. Create a directory `shared` and give read/write access to everyone (mode `777`). Test it by writing a file into it.
3. Use `id` to list your user’s UID, GID, and groups.
4. Use `ls -l` on `/etc/passwd` and `/etc/master.passwd`. Compare their permissions and explain why they differ.
5. Create a file and change its owner to `root` using `sudo chown`. Try editing it as a normal user. What happens?
6. Add a new user with `sudo adduser`. Set a password, log in as that user, and check their default home directory.

3.15.4 Processes and System Monitoring (7 exercises)

1. Start a process in the foreground with `sleep 60`. While it runs, open another terminal and use `ps` to find it.
2. Start the same process in the background with `sleep 60 &`. Use `jobs` and `fg` to bring it back to the foreground.
3. Use `top` to find which process is consuming the most CPU at the moment.
4. Start a `yes` process (`yes > /dev/null &`) to flood the CPU. Watch it in `top`, then stop it with `kill`.
5. Check how long your system has been running with `uptime`.
6. Use `df -h` to see how much disk space is available on your system. Which filesystem is mounted on `/`?
7. Run `sysctl vm.stats.vm.v_page_count` to see the number of memory pages on your system.

3.15.5 Installing and Managing Software (pkg and Ports) (6 exercises)

1. Use `pkg search` to look for a text editor other than `nano`. Install it, run it, then remove it.
2. Install the `htop` package with `pkg`. Compare its output to the built-in `top`.

3. Explore the Ports Collection by navigating to `/usr/ports/editors/nano`. Look at the Makefile.
4. Build nano from ports with `sudo make install clean`. Did it ask you about options?
5. Update your ports tree using git. Which categories were updated?
6. Use `which` to locate where the binary of nano or htop was installed. Check whether it's under `/usr/bin` or `/usr/local/bin`.

3.15.6 Automation and Scheduling (cron, at, periodic) (6 exercises)

1. Write a cron job that logs the current date and time every 2 minutes into `~/time.log`. Wait and check it with `tail`.
2. Write a cron job that cleans up all `.tmp` files in your home directory every night at midnight.
3. Use the `at` command to schedule a message to yourself 5 minutes from now.
4. Run `sudo periodic daily` and read its output. What kinds of tasks does it perform?
5. Add a cron job that runs `df -h` every day at 8 AM and logs the result to `~/disk.log`.
6. Redirect cron job output into a custom log file (`~/cron_output.log`). Confirm that both normal output and errors are captured.

3.15.7 Shell Scripting (/bin/sh) (7 exercises)

1. Write a script `hello_user.sh` that prints your username, current date, and number of processes running. Make it executable and run it.
2. Write a script `organize.sh` that moves all `.txt` files from your home directory into a folder called `texts`. Add comments to explain each step.
3. Modify `organize.sh` to also create subdirectories by file type (images, docs, archives).
4. Write a script `disk_alert.sh` that warns you if root filesystem usage exceeds 80%.
5. Write a script `logger.sh` that appends a timestamped entry to `~/activity.log` with the current directory and user.
6. Write a script `backup.sh` that creates a `.tar.gz` archive of `~/unix_playground` into `~/backups/`.
7. Extend `backup.sh` so that it keeps only the last 5 backups and deletes older ones automatically.

3.15.8 Peeking into the Kernel (dmesg, sysctl, /dev) (6 exercises)

1. Use `dmesg` to find the model of your primary disk.
2. Use `sysctl hw.model` to display your CPU model and `sysctl hw.ncpu` to display how many cores you have.
3. Change your hostname temporarily using `sysctl kern.hostname=mytesthost`. Check it with `hostname`.
4. Use `ls /dev` to list device nodes. Identify which ones represent disks, terminals, and virtual devices.
5. Use `head -c 16 /dev/random | hexdump` to read 16 random bytes from the kernel.
6. Plug in a USB stick (if available) and run `dmesg | tail`. Can you see which new `/dev/` entry appeared?

3.15.9 Wrapping Up

With these **46 exercises**, you've covered every major topic in this chapter:

- Filesystem navigation and layout
- Users, groups, and permissions
- Processes and monitoring
- Software installation with pkg and ports
- Automation with cron, at, and periodic
- Shell scripting with FreeBSD's native `/bin/sh`
- Kernel introspection with `dmesg`, `sysctl`, and `/dev`

By completing them, you'll move from being a *passive reader* to an **active UNIX practitioner**. You'll not only know how FreeBSD works, you'll have *lived inside it*.

These exercises are the **muscle memory** you'll need when we start programming. When we reach C and later kernel development, you'll already be fluent in the daily tools of a UNIX developer.

3.15.10 Looking Ahead

The next chapter will introduce the **C programming language**, the language of the FreeBSD kernel. This is the tool you'll use to create device drivers. Don't worry if you've never programmed before, we'll build your understanding step by step, just as we did with UNIX in this chapter.

By combining your new UNIX literacy with C programming skills, you'll be ready to begin shaping the FreeBSD kernel itself.

4 A First Look at the C Programming Language

Before we can start on writing FreeBSD device drivers, we need to learn the language they're written in. That language is C, short and, admittedly, a little quirky. But don't worry, you don't need to be a programming expert to get started.

In this chapter, I'll walk you through the basics of the C programming language, assuming absolutely no prior experience. If you've never written a line of code in your life, you're in the right place. If you've done some programming in other languages like Python or JavaScript, that's fine too; C might feel a little more manual, but we'll tackle it together.

Our goal here isn't to become master C programmers in one chapter. Instead, I want to introduce you to the language gently, showing you its syntax, its building blocks, and how it works in the context of UNIX systems like FreeBSD. Along the way, I'll point out real-world examples taken directly from the FreeBSD source code to help ground the theory in actual practice.

By the time we're done, you'll be able to read and write basic C programs, understand the core syntax, and feel confident enough to take the next steps toward kernel development. But that part will come later, for now, let's focus on learning the essentials.

4.1 Reader Guidance: How to Use This Chapter

This chapter is not just a quick read; it is both a **reference** and a **hands-on bootcamp** in C programming with a FreeBSD flavour. The depth of material is significant, covering everything from the very first "Hello, World!" to pointers, memory safety, modular code, debugging, good practices, labs, and challenge exercises. How much time you'll spend here depends on how deep you go:

- **Reading only:** Around **12 hours** to read all explanations and FreeBSD kernel examples at a beginner's pace. This assumes careful reading but not stopping for practice.
- **Reading + labs:** Around **18 hours** if you pause to type, compile, and run each of the practical labs on your FreeBSD system, making sure you understand the results.
- **Reading + labs + challenges:** Around **22 hours or more**, since the challenge exercises are designed to make you stop, think, debug, and sometimes revisit earlier material before moving forward.

4.1.1 How to Get the Most Out of This Chapter

- **Take it in sections.** Don't attempt the whole chapter in one sitting. Each section (variables, operators, control flow, pointers, arrays, structs, etc.) is self-contained and can be studied, practiced, and digested before moving on.
- **Type the code yourself.** Copy-pasting may feel faster, but it skips the muscle memory that makes coding natural. Typing every example builds fluency with both C and FreeBSD's environment.

- **Explore the FreeBSD source tree.** Many examples come directly from kernel code. Open the referenced files, read surrounding context, and see how the pieces fit in real-world systems.
- **Treat labs as checkpoints.** Each lab is a moment to pause, apply what you've learned, and verify that the concepts are solid.
- **Leave the challenges for last.** They are meant to consolidate all material. Attempt them only once you feel comfortable with the main text and labs.
- **Set a realistic pace.** If you dedicate 1-2 hours per day, expect this chapter to take a week or more to complete with labs, and longer if you also tackle all challenges. Think of it as a **training programme** rather than a single reading task.

This chapter is deliberately long because **C is the foundation** for everything else you'll do in FreeBSD device driver programming. Treat it as your **toolbox**. Once you master it, the material in later chapters will click into place much more naturally.

4.2 Introduction

Let's start at the beginning: what is C, and why is it important to us?

4.2.1 What is C?

C is a programming language created in the early 1970s by Dennis Ritchie at Bell Labs. It was designed to write operating systems, and that's still one of its biggest strengths today. In fact, most modern operating systems, including FreeBSD, Linux, and even parts of Windows and macOS, are written mainly in C.

C is fast, compact, and close to the hardware, but unlike assembly language, it's still readable and expressive. You can write efficient code with C, but it also expects you to be careful. There's no safety net: no automatic memory management, no runtime error messages, and not even built-in strings like in Python or JavaScript.

This might sound scary, but it's actually a feature. When writing drivers or working inside the kernel, you want control, and C gives you that control.

4.2.2 Why Should I Learn C for FreeBSD?

FreeBSD is written almost entirely in C, and that includes the kernel, device drivers, userland tools, and system libraries. If you want to write code that interacts with the operating system, whether it's a new device driver or a custom kernel module, C is your entry point. More specifically:

- The FreeBSD kernel APIs are written in C.
- All device drivers are implemented in C.
- Even debugging tools like dtrace and kgdb understand and expose C-level information.

So, to work with FreeBSD's internals, you'll need to understand how C code is written, structured, compiled, and used in the system.

4.2.3 What If I've Never Programmed Before?

No problem! I'm writing this chapter with you in mind. We'll take it one step at a time, starting with the simplest possible program and slowly building our way up. You'll learn about:

- Variables and data types
- Functions and flow control
- Pointers, arrays, and structures
- How to read real code from the FreeBSD kernel

And don't worry if any of those terms are unfamiliar right now, they'll all make sense soon. I'll provide plenty of examples, explain every step in plain language, and help you build your confidence as we go.

4.2.4 How Is This Chapter Organized?

Here's a quick preview of what's coming up:

- We'll begin by setting up your development environment on FreeBSD.
- Then, we'll walk through your first C program, the classic "Hello, World!".
- From there, we'll cover the syntax and semantics of C: variables, loops, functions, and more.
- We'll show you real examples from the FreeBSD source tree, so you can start learning how the system works under the hood.
- Finally, we'll wrap up with some good practices and a look at what's coming in the next chapter, where we begin applying C to the kernel world.

Are you ready? Let's jump into the next section and get your environment set up so you can run your first C program on FreeBSD.

If You Already Know C

Not every section in this chapter will be new territory for you. If you are comfortable writing C that compiles, links, and runs on a UNIX system, and if you already read pointers, structs, and function pointers without effort in unfamiliar code, a fast path through the chapter is the better use of your time.

Read these sections carefully, because they cover the places where kernel C differs meaningfully from the C you already know:

- **Pointers and Memory**, in particular the subsections *Pointers and Functions* and *Pointers to Structs*. The first introduces function pointers the way the FreeBSD kernel actually uses them; the second shows the softc-and-handle idioms that recur in every driver.
- **Dynamic Memory Allocation** introduces `malloc(9)`, `M_*` memory types, and the rules around sleeping versus non-sleeping allocations. None of this is standard C, and skimming it is a false economy.
- **Memory Safety in Kernel Code** covers the kernel-specific failure modes (double free, use-after-free, unchecked copyin/copyout, sleeping while holding a spin lock) that standard C handbooks do not teach.
- **Structures and typedef in C** reviews the idioms FreeBSD uses for softc layouts, kobj method tables, and opaque type handles. Read it even if you already know what a `struct` is.
- **Good Practices for C Programming** closes the chapter with FreeBSD's Kernel Normal Form (KNF)

and the conventions described in `style(9)`. Every patch submitted upstream is judged against these.

Skim the rest of the labs. Most of them drill standard-C material you already know. A handful are kernel-flavoured and worth doing even if your C is strong: **Hands-On Lab 1: Crashing with an Uninitialized Pointer** and **Mini-Lab 3: Memory Allocation in a Kernel Module** in the memory sections, and **Lab 4: Function Pointer Dispatch (“Mini devsw”)** in the Final Practice Labs. These three give you kernel-specific muscle memory that the prose alone cannot.

Once the sections above are behind you, **jump ahead to Chapter 5** and engage with the kernel directly. If Chapter 5 flags a concept that feels unfamiliar, come back to the matching section here and read it properly. A reader who already knows C can comfortably compress Chapter 4 into a single focused evening while still leaving with everything Chapter 5 needs.

4.3 Setting Up Your Environment

Before we can start writing C code, we need to set up a working development environment. The good news? If you’re running FreeBSD, **you already have most of what you need**.

In this section, we’ll:

- Verify your C compiler is installed
- Compile your first program manually
- Learn how to use Makefiles for convenience

Let’s go step by step.

4.3.1 Installing a C Compiler on FreeBSD

FreeBSD includes the Clang compiler as part of the base system, so you typically don’t need to install anything extra to start writing C code.

To confirm that Clang is installed and working, open a terminal and run:

```
% cc --version
```

You should see output like this:

```
FreeBSD clang version 19.1.7 (https://github.com/llvm/llvm-project.git
  llvmorg-19.1.7-0-gcd708029e0b2)
Target: aarch64-unknown-freebsd14.3
Thread model: posix
InstalledDir: /usr/bin
```

If `cc` is not found, you can install the base development utilities running the following command as root:

```
# pkg install llvm
```

But for almost all standard FreeBSD setups, Clang should already be ready to use.

Let's write the classic "Hello, World!" program in C. This will verify that your compiler and terminal are working correctly.

Open a text editor like `ee`, `vi`, or `nano`, and create a file called `hello.c`:

```
#include <stdio.h>

int main(void) {
    printf("Hello, World!\n");
    return 0;
}
```

Let's break this down:

- `#include <stdio.h>` tells the compiler to include the Standard I/O header file, which provides `printf`.
- `int main(void)` defines the main entry point of the program.
- `printf(...)` writes a message to the terminal.
- `return 0;` indicates successful execution.

Now save the file and compile it:

```
% cc -o hello hello.c
```

This tells Clang to:

- Compile `hello.c`
- Output the result to a file called `hello`

Run it:

```
% ./hello
Hello, World!
```

Congratulations! You just compiled and ran your first C program on FreeBSD.

4.3.2 Behind the Scenes: The Compilation Pipeline

When you run:

```
% cc -o hello hello.c
```

a lot happens under the hood. The process goes through several stages:

1. **Preprocessing**

- Handles `#include` and `#define`.
- Expands macros, includes headers, and produces a pure C source file.

2. Compilation

- Translates the preprocessed C code into **assembly language** for your CPU architecture.

3. Assembly

- Converts the assembly into **machine code instructions**, producing an **object file** (`hello.o`).

4. Linking

- Combines object files with the standard library (e.g., `printf()` from `libc`).
- Produces the final executable (`hello`).

Finally, when you run:

```
% ./hello
```

the operating system loads the program into memory and starts executing it at the `main()` function.

Quick Mental Model: Think of it like **building a house**:

- Preprocessing = gathering the blueprints
- Compilation = turning blueprints into instructions
- Assembly = workers cutting raw materials
- Linking = putting all parts together into the finished house

Later in Section 4.15, we'll go further into compiling, linking, and debugging, but for now, this mental picture will help you understand what's really happening when you build your first programs.

4.3.3 Using Makefiles

Typing long compile commands can get annoying as your programs grow. That's where **Makefiles** come in handy.

A Makefile is a plain text file named `Makefile` that defines how to build your program. Here's a very simple one for our Hello World example:

```
# Makefile for hello.c

hello: hello.c
    cc -o hello hello.c
```

Attention: Every command line that will be executed by the shell within a Makefile rule must begin with a tab character, not spaces. If you use spaces, the make execution will fail.”

To use it:

Save this in a file called `Makefile` (note the capital “M”) Run `make` in the same directory:

```
% make
cc -o hello hello.c
```

This is especially helpful when your project grows to include multiple files.

Important Note: One of the most common mistakes when writing your first Makefile is forgetting to use a TAB character at the beginning of each command line in a rule. In Makefiles, every line that should be executed by the shell must start with a TAB, not spaces. If you accidentally use spaces instead, make will produce an error and fail to run. This detail often trips up beginners, so be sure to check your indentation carefully!

This error will appear as shown below:

```
% make
make: "/home/ebrandi/hello/Makefile" line 4: Invalid line type
make: Fatal errors encountered -- cannot continue
make: stopped in /home/ebrandi/hello
```

For now, we're only compiling one source file at a time. In Section 4.14, you'll see how larger programs are organized into multiple .c files with shared headers, just like in the FreeBSD kernel.

4.3.4 Installing the FreeBSD Source Code

As we move forward, we'll look at examples from the actual FreeBSD kernel source. To follow along, it's useful to have the FreeBSD source tree installed locally.

To store a complete local copy of the FreeBSD source code, you will need approximately 3.6 GB of free disk space. You can install it using Git by running the following command:

```
# git clone --branch releng/14.3 --depth 1 https://git.FreeBSD.org/src.git src /usr/src
```

This will give you access to all source code, which we'll reference frequently throughout this book.

4.3.5 Summary

You now have a working development setup on FreeBSD! It was simple, wasn't it?

Here's what you've accomplished:

- Verified the C compiler is installed
- Wrote and compiled your first C program
- Learned how to use Makefiles
- Cloned the FreeBSD source tree for future reference

These tools are all you need to start learning C, and later, to build your own kernel modules and drivers. In the next section, we'll look at what makes up a typical C program and how it's structured.

4.4 Anatomy of a C Program

Now that you've compiled your first "Hello, World!" program, let's take a closer look at what's actually going on inside that code. In this section, we'll break down the basic structure of a C program and explain what each part does, step by step.

We'll also introduce how this structure appears in the FreeBSD kernel code, so you can begin recognising familiar patterns in real-world systems programming.

4.4.1 The Basic Structure

Every C program follows a similar structure:

```
#include <stdio.h>

int main(void) {
    printf("Hello, World!\n");
    return 0;
}
```

Let's dissect this line by line.

4.4.2 #include Directives: Adding Libraries

```
#include <stdio.h>
```

This line is handled by the **C preprocessor** before the program is compiled. It tells the compiler to include the contents of a system header file.

- `<stdio.h>` is a standard header file that provides I/O functions like `printf`.
- Anything you include this way is pulled into your program at compile time.

In FreeBSD source code, you'll often see many `#include` directives at the top of a file. Here's an example from the FreeBSD kernel file `sys/kern/kern_shutdown.c`:

```
#include <sys/cdefs.h>
#include "opt_ddb.h"
#include "opt_ekcd.h"
#include "opt_kdb.h"
#include "opt_panic.h"
#include "opt_printf.h"
#include "opt_sched.h"
#include "opt_watchdog.h"
```

```
#include <sys/param.h>
#include <sys/system.h>
#include <sys/bio.h>
#include <sys/boottrace.h>
#include <sys/buf.h>
#include <sys/conf.h>
#include <sys/compressor.h>
#include <sys/cons.h>
#include <sys/disk.h>
#include <sys/eventhandler.h>
#include <sys/filedesc.h>
#include <sys/jail.h>
#include <sys/kdb.h>
#include <sys/kernel.h>
#include <sys/kerneldump.h>
#include <sys/kthread.h>
#include <sys/ktr.h>
#include <sys/malloc.h>
#include <sys/mbuf.h>
#include <sys/mount.h>
#include <sys/priv.h>
#include <sys/proc.h>
#include <sys/reboot.h>
#include <sys/resourcevar.h>
#include <sys/rwlock.h>
#include <sys/sbuf.h>
#include <sys/sched.h>
#include <sys/smp.h>
#include <sys/stdarg.h>
#include <sys/sysctl.h>
#include <sys/sysproto.h>
#include <sys/taskqueue.h>
#include <sys/vnode.h>
#include <sys/watchdog.h>

#include <crypto/chacha20/chacha.h>
#include <crypto/rijndael/rijndael-api-fst.h>
#include <crypto/sha2/sha256.h>

#include <ddb/ddb.h>

#include <machine/cpu.h>
#include <machine/dump.h>
#include <machine/pcb.h>
```

```
#include <machine/smp.h>

#include <security/mac/mac_framework.h>

#include <vm/vm.h>
#include <vm/vm_object.h>
#include <vm/vm_page.h>
#include <vm/vm_pager.h>
#include <vm/swap_pager.h>

#include <sys/signalvar.h>
```

These headers define macros, constants, and function prototypes used in the kernel. For now, just remember: `#include` brings in definitions you want to use.

4.4.3 The `main()` Function: Where Execution Begins

```
int main(void) {
```

- This is the **entry point of your program**. When your program runs, it starts here.
- The `int` means the function returns an integer to the operating system.
- `void` means it takes no arguments.

In user programs, `main()` is where you write your logic. In the kernel, however, there's **no** `main()` function like this; the kernel has its own bootstrapping process. But FreeBSD kernel modules and subsystems still define **entry points** that act in similar ways.

For example, device drivers use functions like:

```
static int
mydriver_probe(device_t dev)
```

And they are registered with the kernel during initialisation; these behave like a `main()` for specific subsystems.

4.4.4 Statements and Function Calls

```
printf("Hello, World!\n");
```

This is a **statement**, a single instruction that performs some action.

- `printf()` is a function provided by `<stdio.h>` that prints formatted output.

- "Hello, World!\n" is a string literal, with \n meaning “new line”.

Important Note: In kernel code, you don’t use the `printf()` function from the Standard C Library (`libc`). Instead, the FreeBSD kernel provides its own internal version of `printf()` tailored for kernel-space output, a distinction we’ll explore in more detail later in the book.

4.4.5 Return Values

```
    return 0;
}
```

This tells the operating system that the program completed successfully. Returning 0 usually means “no error”.

You’ll see a similar pattern in kernel code where functions return 0 for success and a non-zero value for failure.

4.4.6 Bonus Learning Point About Return Values

Let’s see a practical example from `sys/kern/kern_exec.c`:

```
exec_map_first_page(struct image_params *imgp)
{
    vm_object_t object;
    vm_page_t m;
    int error;

    if (imgp->firstpage != NULL)
        exec_unmap_first_page(imgp);

    object = imgp->vp->v_object;
    if (object == NULL)
        return (EACCES);
    #if VM_NRESERVLEVEL > 0
    if ((object->flags & OBJ_COLORED) == 0) {
        VM_OBJECT_WLOCK(object);
        vm_object_color(object, 0);
        VM_OBJECT_WUNLOCK(object);
    }
    #endif
    error = vm_page_grab_valid_unlocked(&m, object, 0,
        VM_ALLOC_COUNT(VM_INITIAL_PAGEIN) |
        VM_ALLOC_NORMAL | VM_ALLOC_NOBUSY | VM_ALLOC_WIRED);

    if (error != VM_PAGER_OK)
```

```

        return (EIO);
    imgp->firstpage = sf_buf_alloc(m, 0);
    imgp->image_header = (char *)sf_buf_kva(imgp->firstpage);

    return (0);
}

```

Return Values in `exec_map_first_page()`:

- `return (EACCES);` Returned when the executable file's vnode (`imgp->vp`) has no associated virtual memory object (`v_object`). Without this object, the kernel cannot map the file into memory. This is treated as a **permission/access error**, using the standard `EACCES` error code ("Permission Denied").
- `return (EIO);` Returned when the kernel fails to retrieve a valid memory page from the file via `vm_page_grab_valid_unlocked()`. This may happen due to an I/O failure, memory issue, or file corruption. The `EIO` code ("Input/Output Error") signals a **low-level failure** in reading or allocating memory for the file.
- `return (0);` Returned on successful completion of the function. This indicates that the kernel has successfully grabbed the first page of the executable file, mapped it into memory, and stored the address of the header in `imgp->image_header`. A return value of `0` is the standard kernel convention for indicating success.

The use of `errno`-style error codes like `EIO` and `EACCES` ensures consistent error handling throughout the kernel, making it easier for driver developers and kernel programmers to propagate errors reliably and interpret failure conditions in a familiar, standardised way.

The FreeBSD kernel makes extensive use of `errno`-style error codes to represent different failure conditions consistently. Don't worry if they seem unfamiliar at first, as we move forward, you'll naturally encounter many of them, and I'll help you understand how they work and when to use them.

For a complete list of standard error codes and their meanings, you can refer to the FreeBSD manual page:

```
% man 2 intro
```

4.4.7 Putting It All Together

Let's revisit our Hello World program, now with full comments:

```

#include <stdio.h>           // Include standard I/O library

int main(void) {            // Entry point of the program
    printf("Hello, World!\n"); // Print a message to the terminal
    return 0;                // Exit with success
}

```

In this short example, you've already seen:

- A preprocessor directive
- A function definition
- A standard library call
- A return statement

These are the **building blocks of C** and you'll see them repeated everywhere, including deep inside FreeBSD's kernel source code.

4.4.8 A First Glimpse at Good Practices in C

Before we move on to variables, types, and control flow, it is worth pausing for a short note on **style and discipline**. You are just starting out, but if you learn a few habits now, you will save yourself a lot of trouble later. FreeBSD, like every mature project, follows its own coding conventions called **KNF (Kernel Normal Form)**. We will study these in depth near the end of this chapter, but here are four essentials you should keep in mind right from the start:

4.4.8.1 1. Always Use Braces

Even if an `if` or `for` only controls a single statement, always write it with braces:

```
if (x > 0) {  
    printf(Positive\n);  
}
```

This avoids a whole class of bugs and keeps your code safe when you later add more statements.

4.4.8.2 2. Indentation Matters

The FreeBSD kernel style guide requires **tabs, not spaces**, for indentation. Your editor should insert a tab character at each indent level. This is not about aesthetics: consistent indentation makes kernel code readable and reviewable.

4.4.8.3 3. Prefer Meaningful Names

Avoid calling your variables `a`, `b`, or `tmp1`. A name like `counter`, `buffer_size`, or `error_code` makes the code immediately self-explanatory. Remember: in FreeBSD, your code will one day be read by someone else, and often years after you wrote it.

4.4.8.4 4. No “Magic Numbers”

If you find yourself writing something like:


```
if (users > 64) { ... }
```

Replace 64 with a named constant:

```
#define MAX_USERS 64  
if (users > MAX_USERS) { ... }
```

This makes your code easier to maintain and avoids hidden assumptions.

4.4.8.5 Why This Matters for You

Right now, these may feel like small details. But by building these habits early, you will avoid learning “sloppy” C that has to be unlearned when you reach kernel development. Think of this as a **survival kit**: a few essential rules that will keep your code clear, safe, and closer to what you will see in the FreeBSD source tree.

Later in this chapter, you will revisit these practices in depth, along with many more advanced conventions used by FreeBSD developers. For now, just keep these four in mind as you start coding.

4.4.9 Summary

In this section, you’ve learned:

- The structure of a C program
- How `#include` and `main()` work
- What `printf()` and `return` do
- How similar structures appear in FreeBSD’s kernel code
- Early good practices to keep your code clear and safe

The more C code you read, both your own and from FreeBSD, the more these patterns will become second nature.

4.5 Variables and Data Types

In any programming language, variables are how you store and manipulate data. In C, variables are a little more “manual” than in higher-level languages, but they give you the control you need to write fast, efficient programs, and that’s precisely what operating systems like FreeBSD require.

In this section, we’ll explore:

- How to declare and initialise variables
- The most common data types in C
- How FreeBSD uses them in kernel code
- Some tips to avoid common beginner mistakes

Let’s start with the basics.

4.5.1 What Is a Variable?

A variable is like a labeled box in memory where you can store a value, such as a number, a character, or even a block of text.

Here's a simple example:

```
int counter = 0;
```

This tells the compiler:

- Allocate enough memory to store an integer
- Call that memory location counter
- Put the number 0 in it to start

4.5.2 Declaring Variables

In C, you must declare the type of every variable before using it. This is different from languages like Python, where the type is determined automatically.

Here's how to declare different types of variables:

```
int age = 30;           // Integer (whole number)
float temperature = 98.6; // Floating-point number
char grade = 'A';       // Single character
```

You can also declare multiple variables at once:

```
int x = 10, y = 20, z = 30;
```

Or leave them uninitialized (but be careful, as uninitialized variables contain garbage values!):

```
int count; // May contain anything!
```

Always initialise your variables, not just because it's good C practice, but because in kernel development, uninitialized values can lead to subtle and dangerous bugs, including kernel panics, unpredictable behaviour, and security vulnerabilities. In userland, mistakes might crash your program; in the kernel, they can compromise the stability of the entire system.

Unless you have a very specific and justified reason not to (such as performance-critical code paths where the value is immediately overwritten), make initialisation the rule, not the exception.

4.5.3 Common C Data Types

Here are the core types you'll use most often:

Type	Description	Example
int	Integer (typically 32-bit)	int count = 1;
unsigned	Non-negative integer	unsigned size = 10;
char	A single 8-bit character	char c = 'A';
float	Floating-point number (~6 decimal digits)	float pi = 3.14;
double	Double-precision float (~15 digits)	double g = 9.81;
void	Represents “no value” (used for functions)	void print()

4.5.4 Type Qualifiers

C provides **type qualifiers** to give more information about how a variable should behave:

- **const**: This variable can't be changed.
- **volatile**: The value can change unexpectedly (used with hardware!).
- **unsigned**: The variable cannot hold negative numbers.

Example:

```
const int max_users = 100;
volatile int status_flag;
```

The `volatile` qualifier can be important in FreeBSD kernel development, but only in very specific contexts, such as accessing hardware registers or dealing with interrupt-driven updates. It tells the compiler not to optimise accesses to a variable, which is critical when values can change outside of normal program flow.

However, `volatile` is not a substitute for proper synchronisation and should not be used for coordinating access between threads or CPUs. For that, the FreeBSD kernel provides dedicated primitives like mutexes and atomic operations, which offer both compiler and CPU-level guarantees.

4.5.5 Constant Values and #define

In C programming and especially in kernel development, it's very common to define constant values using the `#define` directive:

```
#define MAX_DEVICES 64
```

This line doesn't declare a variable. Instead, it's a **preprocessor macro**, which means the C preprocessor will **replace every occurrence of** `MAX_DEVICES` **with** `64` before the actual compilation begins. This replacement happens **textually**, and the compiler never even sees the name `MAX_DEVICES`.

4.5.6 Why Use #define for Constants?

Using #define for constant values has several advantages in kernel code:

- **Improves readability:** Instead of seeing magic numbers (like 64) scattered throughout the code, you see meaningful names like MAX_DEVICES.
- **Makes code easier to maintain:** If the maximum number of devices ever needs to change, you update it in one place, and the change is reflected wherever it's used.
- **Keeps kernel code lightweight:** Kernel code often avoids runtime overhead, and #define constants don't allocate memory or exist in the symbol table; they simply get replaced during preprocessing.

4.5.7 Real Example From FreeBSD

You will find many #define lines in sys/sys/param.h, for example:

```
#define MAXHOSTNAMELEN 256 /* max hostname size */
```

This defines the maximum number of characters allowed in a system hostname, and it's used throughout the kernel and system utilities to enforce a consistent limit. The value 256 is now standardised and can be reused wherever the hostname length is relevant.

4.5.8 Watch Out: There Is No Type Checking

Because #define simply performs textual substitution, it does not respect types or scoping.

For example:

```
#define PI 3.14
```

This works, but it can lead to problems in certain contexts (e.g., integer promotion, unintended precision loss). For more complex or type-sensitive constants, you may prefer using const variables or enums in userland, but in the kernel, especially in headers, #define is often chosen for efficiency and compatibility.

4.5.9 Best Practices for #define Constants in Kernel Development

- Use **ALL CAPS** for macro names to distinguish them from variables.
- Add comments to explain what the constant represents.
- Avoid defining constants that depend on runtime values.
- Prefer #define over const in header files or when targeting C89 compatibility (which is still common in kernel code).

4.5.10 Best Practices for Variables

Writing correct and robust kernel code starts with disciplined variable usage. The tips below will help you avoid subtle bugs, improve code readability, and align with FreeBSD kernel development conventions.

Always initialise your variables: Never assume a variable starts at zero or any default value, especially in kernel code, where behaviour must be deterministic. An uninitialized variable could hold random garbage from the stack, leading to unpredictable behaviour, memory corruption, or kernel panics. Even when the variable will be overwritten soon, it's often safer and more transparent to initialise it explicitly unless performance measurements prove otherwise.

Don't use variables before assigning a value: This is one of the most common bugs in C, and compilers won't always catch it. In the kernel, using an uninitialized variable can result in silent failures or catastrophic system crashes. Always trace your logic to ensure every variable is assigned a valid value before use, especially if it influences memory access or hardware operations.

Use const whenever the value shouldn't change: Using `const` is more than good style; it helps the compiler enforce read-only constraints and catch unintended modifications. This is particularly important when:

- Passing read-only pointers into functions
- Protecting configuration structures or table entries
- Marking driver data that must not change after initialisation

In kernel code, this can even lead to compiler optimisations and make the code easier to reason about for reviewers and maintainers.

Use unsigned for values that can't be negative (like sizes or counters): Variables that represent quantities like buffer sizes, loop counters, or device counts should be declared as unsigned types (`unsigned int`, `size_t`, or `uint32_t`, etc.). This improves clarity and prevents logic bugs, especially when comparing with other unsigned types, which can cause unexpected behaviour if signed values are mixed in.

Prefer fixed-width types in kernel code (`uint32_t`, `int64_t`, etc.): Kernel code must behave predictably across architectures (e.g., 32-bit vs 64-bit systems). Types like `int`, `long`, or `short` can vary in size depending on the platform, which can lead to portability issues and alignment bugs. Instead, FreeBSD uses standard types from `<sys/types.h>` such as:

- `uint8_t`, `uint16_t`, `uint32_t`, `uint64_t`
- `int32_t`, `int64_t`, etc.

These types ensure your code has a known, fixed layout and avoids surprises when compiling or running on different hardware.

Pro Tip: When in doubt, look at existing FreeBSD kernel code, especially drivers and subsystems close to what you're working on. The variable types and initialisation patterns used there are often based on years of hard-earned lessons from real-world systems.

4.5.11 Summary

In this section, you've learned:

- How to declare and initialise variables

- The most important data types in C
- What type qualifiers like `const` and `volatile` do
- How to spot and understand variable declarations in FreeBSD's kernel code

You now have the tools to store and work with data in C, and you've already seen how FreeBSD uses the same concepts in production-quality kernel code.

4.6 Operators and Expressions

So far, we've learned how to declare and initialise variables. Now it's time to make them do something! In this section, we'll look at operators and expressions, the mechanisms in C that allow you to compute values, compare them, and control program logic.

We'll cover:

- Arithmetic operators
- Comparison operators
- Logical operators
- Bitwise operators (lightly)
- Assignment operators
- Real examples from FreeBSD kernel code

4.6.1 What Is an Expression?

In C, an expression is anything that produces a value. For example:

```
int a = 3 + 4;
```

Here, `3 + 4` is an expression that evaluates to 7. The result is then assigned to `a`.

Operators are what you use to **build expressions**.

4.6.2 Arithmetic Operators

These are used for basic math:

Operator	Meaning	Example	Result
+	Addition	5 + 2	7
-	Subtraction	5 - 2	3
*	Multiplication	5 * 2	10
/	Division	5 / 2	2 (integer division!)
%	Modulus	5 % 2	1 (remainder)

Note: In C, division of two integers **discards the decimal part**. To get floating-point results, at least one operand must be a float or double.

4.6.3 Comparison Operators

These are used to compare two values and return either true (1) or false (0):

Operator	Meaning	Example	Result
==	Equal to	a == b	1 if equal
!=	Not equal to	a != b	1 if not equal
<	Less than	a < b	1 if true
>	Greater than	a > b	1 if true
<=	Less than or equal	a <= b	1 if true
>=	Greater than or equal	a >= b	1 if true

These are heavily used in `if`, `while`, and `for` statements to control program flow.

4.6.4 Logical Operators

Used to combine or invert conditions:

Operator	Name	Description	Example	Result
&&	Logical AND	True if both conditions are true	(a > 0) && (b < 5)	1 if both are true
	Logical OR	True if either condition is true	(a == 0) (b > 10)	1 if at least one is true
!	Logical NOT	Reverses the truth value of the condition	!done	1 if done is false

These are especially useful in complex conditionals, like:

```
if ((a > 0) && (b < 100)) {
    // both conditions must be true
}
```

Tip: In C, any non-zero value is considered “true,” and zero is considered “false”.

4.6.5 Assignment and Compound Assignment

The = operator assigns a value:

```
x = 5; // assign 5 to x
```

Compound assignment combines operation and assignment:

Operator	Meaning	Example	Equivalent to
+=	Add and assign	x += 3;	x = x + 3;
-=	Subtract and assign	x -= 2;	x = x - 2;
*=	Multiply and assign	x *= 4;	x = x * 4;
/=	Divide and assign	x /= 2;	x = x / 2;
%=	Modulus and assign	x %= 3;	x = x % 3;

4.6.6 Bitwise Operators

In kernel development, bitwise operators are standard. Here's a light preview:

Operator	Meaning	Example
&	Bitwise AND	a & b
	Bitwise OR	a b
^	Bitwise XOR	a ^ b
~	Bitwise NOT	~a
«	Left shift	a « 2
»	Right shift	a » 1

We'll cover these in detail later when we work with flags, registers, and hardware I/O.

4.6.7 Real Example from FreeBSD: sys/kern/tty_info.c

Let's look at a real example from the FreeBSD source code.

Open the file sys/kern/tty_info.c and look for the function thread_compare(), you will see the code below:

```
static int
thread_compare(struct thread *td, struct thread *td2)
{
```



```
int runa, runb;
int slpa, slpb;
fixpt_t esta, estb;

if (td == NULL)
    return (1);

/*
 * Fetch running stats, pctcpu usage, and interruptable flag.
 */
thread_lock(td);
runa = TD_IS_RUNNING(td) || TD_ON_RUNQ(td);
slpa = td->td_flags & TDF_SINTR;
esta = sched_pctcpu(td);
thread_unlock(td);
thread_lock(td2);
runb = TD_IS_RUNNING(td2) || TD_ON_RUNQ(td2);
estb = sched_pctcpu(td2);
slpb = td2->td_flags & TDF_SINTR;
thread_unlock(td2);

/*
 * see if at least one of them is runnable
 */
switch (TESTAB(runb, runa)) {
case ONLYA:
    return (0);
case ONLYB:
    return (1);
case BOTH:
    break;
}

/*
 * favor one with highest recent cpu utilization
 */
if (estb > esta)
    return (1);
if (esta > estb)
    return (0);

/*
 * favor one sleeping in a non-interruptible sleep
 */
switch (TESTAB(slpb, slpa)) {
case ONLYA:
    return (0);
```

```
    case ONLYB:
        return (1);
    case BOTH:
        break;
}

return (td < td2);
}
```

We are interested in this fragment of code:

```
...
runa = TD_IS_RUNNING(td) || TD_ON_RUNQ(td);
...
return (td < td2);
```

Explanation:

- `TD_IS_RUNNING(td)` and `TD_ON_RUNQ(td)` are macros that return boolean values.
- The logical OR `||` checks if either condition is true.
- The result is assigned to 'runa'.

Later, this line:

```
return (td < td2);
```

Uses the less-than operator to compare two pointers (`td` and `td2`). This is valid in C; pointer comparisons are common when choosing between resources.

Another real expression in that same file, inside `tty_info()`, is:

```
pctcpu = (sched_pctcpu(td) * 10000 + FSCALE / 2) >> FSHIFT;
```

This expression:

- Multiplies the CPU usage estimate by 10,000
- Adds half the scale factor for rounding
- Then performs a **bitwise right shift** to scale it down
- It's an optimised way to compute $(\text{value} * \text{scale}) / \text{divisor}$ using bit shifts instead of division

4.6.8 Summary

In this section, you've learned:

- What expressions are in C

- How to use arithmetic, comparison, and logical operators
- How to assign values and use compound assignments
- How bitwise operations show up in kernel code
- How FreeBSD uses these expressions to control logic and calculations

This section builds the foundation for conditional execution and looping, which we'll explore next.

4.7 Control Flow

So far, we've learned how to declare variables and write expressions. But programs need to do more than compute values; they need to **make decisions** and **repeat actions**. This is where **control flow comes in**.

Control flow statements allow you to:

- Choose between different paths (`if`, `else`, `switch`)
- Repeat operations using loops (`for`, `while`, `do...while`)
- Exit loops early (`break`, `continue`)

These are the **decision-making tools of C**, and they're essential for writing meaningful programs, from small utilities to operating system kernels.

4.7.1 Understanding the `if`, `else`, and `else if`

One of the most basic ways to control the flow of a C program is with the `if` statement. It lets your code make decisions based on whether a condition is true or false.

```
if (x > 0) {
    printf("x is positive\n");
} else if (x < 0) {
    printf("x is negative\n");
} else {
    printf("x is zero\n");
}
```

Here's how it works step by step:

1. `if (x > 0)`: The program checks the first condition. If it's true, the block inside runs and the rest of the chain is skipped.
2. `else if (x < 0)`: If the first condition was false, this second one is checked. If it's true, its block runs and the chain ends.
3. `else`: If none of the previous conditions are true, the code inside `else` runs.

Important syntax rules:

- Each condition must be inside **parentheses** ().

- Each block of code is surrounded by **curly braces { }**, even if it's only one line (this prevents common mistakes).

You can see a real example of `if`, `if else` and `else` usage flow control in the function `ifhwioctl()` in `sys/net/if.c`. The fragment that we are interested in is:

```
/* Copy only (length-1) bytes so if_description is always NUL-terminated. */
/* The length parameter counts the terminating NUL. */
if (ifr_buffer_get_length(ifr) > ifdescr_maxlen)
    return (ENAMETOOLONG);
else if (ifr_buffer_get_length(ifr) == 0)
    descrbuf = NULL;
else {
    descrbuf = if_allocdescr(ifr_buffer_get_length(ifr), M_WAITOK);
    error = copyin(ifr_buffer_get_buffer(ifr), descrbuf,
        ifr_buffer_get_length(ifr) - 1);
    if (error) {
        if_freedescrip(descrbuf);
        break;
    }
}
```

This fragment handles a request from user space to set a description for a network interface, for example, giving `em0` a human-readable label like “Main uplink port”. The code checks the length of the description provided and decides what to do next.

Let's walk through the flow control step by step:

1. First `if`: Checks whether the description is too long to fit.
 - If **true**, the function immediately stops and returns an error code (`ENAMETOOLONG`).
 - If **false**, execution moves on to the next condition.
2. `else if`: Runs only if the first condition was **false**.
 - If the length is exactly zero, it means the user didn't provide a description, so the code sets `descrbuf` to `NULL`.
 - If **false**, the program moves on to the final `else`.
3. Final `else`: Executes when neither of the previous conditions are true.
 - Allocates memory for the description and copies the provided text into it.
 - If copying fails, it frees the memory and exits the loop or function.

How the flow works:

- Only one of these three paths runs each time.
- The first matching condition “wins”, and the rest are skipped.
- This is a classic example of using `if / else if / else` to handle mutually exclusive conditions, reject invalid input, handle the empty case, or process a valid value.

In C, `if / else if / else` chains provide a straightforward way to handle several possible outcomes in a single structure. The program checks each condition in order, and as soon as one is true, that block runs and the rest are skipped. This simple rule keeps your logic predictable and easy to follow. In the FreeBSD kernel, you'll see this pattern everywhere, from network stack functions to device drivers, because it ensures that only the correct code path runs for each situation, making the system's decision-making both efficient and reliable.

4.7.2 Understanding the `switch` and `case`

A `switch` statement is a decision-making structure that's useful when you need to compare one variable against multiple possible values. Instead of writing a long chain of `if` and `else if` statements, you can list each possible value as a case.

Here's a simple example:

```
switch (cmd) {
    case 0:
        printf("Zero\n");
        break;
    case 1:
        printf("One\n");
        break;
    default:
        printf("Unknown\n");
        break;
}
```

- The `switch` checks the value of `cmd`.
- Each case is a possible value that `cmd` might have.
- The `break` statement tells the program to stop checking further cases once a match is found. Without `break`, execution will continue into the next case, a behaviour called **fall-through**.
- The `default` case runs if none of the listed cases match.

You can see a real use of `switch` in the FreeBSD kernel inside the function `thread_compare()` in `sys/kern/tty_info.c`. The fragment we're interested in is:

```
switch (TESTAB(runb, runa)) {
    case ONLYA:
        return (0);
    case ONLYB:
        return (1);
    case BOTH:
        break;
}
```

What This Code Does

This code decides which of two threads is “more interesting” for the scheduler based on whether each thread is runnable.

- `runa` and `runb` are flags that indicate if the first thread (a) and the second thread (b) are runnable.
- The macro `TESTAB(a, b)` combines those flags into a single value. This result can be one of three predefined constants:
 - `ONLYA` - Only thread A is runnable.
 - `ONLYB` - Only thread B is runnable.
 - `BOTH` - Both threads are runnable.

The switch works like this:

1. Case `ONLYA`: If only thread A is runnable, return 0.
2. Case `ONLYB`: If only thread B is runnable, return 1.
3. Case `BOTH`: If both threads are runnable, don’t return immediately; instead, break so the rest of the function can handle this situation.

In short, `switch` statements provide a clean and efficient way to handle multiple possible outcomes from a single expression, avoiding the clutter of long `if / else if` chains. In the FreeBSD kernel, they are often used to react to different commands, flags, or states, as in our example, which decides between thread A, thread B, or both. Once you become comfortable reading switch structures, you’ll start to recognise them throughout kernel code as a go-to pattern for organising decision-making logic in a clear, maintainable way.

4.7.3 Understanding the for Loops

A `for` loop in C is perfect when you know **how many times** you want to repeat something. It sets things up in a compact, easy-to-read style:

```
for (int i = 0; i < 10; i++) {  
    printf("%d\n", i);  
}
```

- Start at `i = 0`
- Repeat while `i < 10`
- Increment `i` each time by 1 (`i++`)

A widespread beginner error is related to off-by-one errors (`<=` vs `<`), and forgetting the increment (which can cause an infinite loop).

You can see a real `for` loop inside `/usr/src/sys/net/iflib.c`, in the function `netmap_fl_refill()`. The fragment we care about is the inner batching loop nested inside the function’s outer `while (n > 0)` body.

A note on line numbers. Line numbers accurate for the tree at time of writing; function names are the durable reference. For the curious reader who wants to open the file and look around, `netmap_fl_refill()` begins near line 859 of `/usr/src/sys/net/iflib.c`, the outer `while (n > 0)` body starts near line 915, and the inner batching `for` loop we dissect below runs from roughly line 922 to line 949. These numbers will drift as the file is revised; the function name will not.

```
for (i = 0; n > 0 && i < IFLIB_MAX_RX_REFRESH; n--, i++) {
    struct netmap_slot *slot = &ring->slot[nm_i];
    uint64_t paddr;
    void *addr = PNMB(na, slot, &paddr);
    /* ... work per buffer ... */
    nm_i = nm_next(nm_i, lim);
    nic_i = nm_next(nic_i, lim);
}
```

What this loop does

- The driver is refilling receive buffers so the NIC can keep receiving packets.
- It processes buffers in batches: up to IFLIB_MAX_RX_REFRESH each time.
- *i* counts how many buffers we've handled in this batch.
- *n* is the total remaining buffers to refill; it decrements every iteration.
- For each buffer, the code grabs its slot, figures out the physical address, readies it for DMA, then advances the ring indices (*nm_i*, *nic_i*).
- The loop stops when either the batch is full (*i* hits the max) or there's nothing left to do (*n* == 0). The batch is then “published” to the NIC by the code right after the loop.

In essence, a `for` loop is the go-to choice when you have a clear limit on how many times something should run. It packages initialisation, condition checking, and iteration updates into a single, compact header, making the flow easy to follow.

In FreeBSD's kernel code, this structure is everywhere from scanning arrays to walking network ring buffers, because it keeps repetitive work both predictable and efficient. Our example from `netmap_fl_refill()` shows precisely how this works in practice:

the loop counts through a fixed-size batch of buffers, stopping either when the batch is full or when there's no more work left, then hands that batch off to the NIC. Once you get comfortable reading for loops like this, you'll spot them throughout the kernel and understand how they keep complex systems running smoothly.

4.7.4 Understanding the `while` Loop

In C, a `while` loop is a control structure that allows your program to repeat a block of code as long as a certain condition remains true.

Think of it like telling your program, “Keep doing this task while this rule is true. Stop as soon as the rule becomes false.”

Lets see a example:

```
int i = 0;

while (i < 10) {
    printf("%d\n", i);
    i++;
}
```

Variable Initialization

```
int i = 0;
```

- We create a variable `i` and set its value to 0.
- This will be our counter, keeping track of how many times the loop has run.

The while Condition

```
while (i < 10)
```

- Before each repetition, C checks the condition `i < 10`.
- If the condition is **true**, the block inside the loop is executed.
- If the condition is **false**, the loop stops, and the program continues after the loop.

The Loop Body

```
{  
    printf("%d\n", i);  
    i++;  
}
```

`printf("%d\n", i);` - Prints the value of `i` followed by a newline (`\n`). `i++;` - Increases `i` by 1 after each iteration. Without this step, `i` would stay 0 forever, and the loop would never end, creating an infinite loop.

Key Points to Remember

- A while loop **may not run at all** if the condition is false from the start.
- Always ensure something inside the loop **changes the condition** over time, or you risk an infinite loop.
- In FreeBSD kernel code, `while` loops are common for:
 - Polling hardware status registers until a device is ready.
 - Waiting for a buffer to be filled.
 - Implementing retry mechanisms.

You can see a real example of `while` loop usage in the function `netmap_fl_refill()`, defined in `/usr/src/sys/net/iflib.c`.

This time, I've decided to show you the complete source code for this FreeBSD kernel function because it offers an excellent opportunity to see several concepts from this chapter working together in a real-world context.

To make it easier to follow, I've added explanatory comments at key points so you can connect the theory to the actual implementation. Don't worry if you don't fully understand every detail right now; this is normal when first looking at kernel code.

For our discussion, pay special attention to the `while` loop inside `netmap_fl_refill()`, as it's the part we will study in depth. Look for `while (n > 0) {` in the code below:


```

/*
 * netmap_fl_refill
 * -----
 * This function refills receive (RX) buffers in a netmap-enabled
 * FreeBSD network driver so the NIC can continue receiving packets.
 *
 * It is called in two main situations:
 * 1. Initialization (driver start/reset)
 * 2. RX synchronization (during packet reception)
 *
 * The core idea: figure out how many RX slots need refilling,
 * then load and map each buffer so the NIC can use it.
 */
static int
netmap_fl_refill(iflib_rxq_t rxq, struct netmap_kring *kring, bool init)
{
    struct netmap_adapter *na = kring->na;
    u_int const lim = kring->nkr_num_slots - 1;
    struct netmap_ring *ring = kring->ring;
    bus_dmamap_t *map;
    struct if_rxd_update iru;
    if_ctx_t ctx = rxq->ifr_ctx;
    iflib_fl_t fl = &rxq->ifr_fl[0];
    u_int nic_i_first, nic_i; // NIC descriptor indices
    u_int nm_i; // Netmap ring index
    int i, n; // i = batch counter, n = buffers to process
#ifdef IFLIB_DEBUG_COUNTERS
    int rf_count = 0;
#endif

    /*
     * Figure out how many buffers (n) we need to refill.
     * - In init mode: refill almost the whole ring (minus those in use)
     * - In normal mode: refill from hardware current to ring head
     */
    if (__predict_false(init)) {
        n = kring->nkr_num_slots - nm_kr_rxspace(kring);
    } else {
        n = kring->rhead - kring->nr_hwcur;
        if (n == 0)
            return (0); /* Nothing to do, ring already full. */
        if (n < 0)
            n += kring->nkr_num_slots; /* wrap-around adjustment */
    }
}

```

```

    // Prepare refill update structure
    iru_init(&iru, rxq, 0 /* flid */);
    map = fl->ifl_sds.ifsd_map;

    // Starting positions
    nic_i = fl->ifl_pidx;           // NIC producer index
    nm_i = netmap_idx_n2k(kring, nic_i); // Convert NIC index to netmap index

    // Sanity checks for init mode
    if (__predict_false(init)) {
        MPASS(nic_i == 0);
        MPASS(nm_i == kring->nr_hwtail);
    } else
        MPASS(nm_i == kring->nr_hwcur);

    DBG_COUNTER_INC(fl_refills);

    /*
     * OUTER LOOP:
     * Keep processing until we have refilled all 'n' needed buffers.
     */
    while (n > 0) {
#ifdef IFLIB_DEBUG_COUNTERS
        if (++rf_count == 9)
            DBG_COUNTER_INC(fl_refills_large);
#endif
        nic_i_first = nic_i; // Save where this batch starts

        /*
         * INNER LOOP:
         * Process up to IFLIB_MAX_RX_REFRESH buffers in one batch.
         * This avoids calling hardware refill for every single buffer.
         */
        for (i = 0; n > 0 && i < IFLIB_MAX_RX_REFRESH; n--, i++) {
            struct netmap_slot *slot = &ring->slot[nm_i];
            uint64_t paddr;
            void *addr = PNMB(na, slot, &paddr); // Get buffer address and phys addr

            MPASS(i < IFLIB_MAX_RX_REFRESH);

            // If the buffer address is invalid, reinitialize the ring
            if (addr == NETMAP_BUF_BASE(na))
                return (netmap_ring_reinit(kring));
        }
    }

```

```
// Save the physical address and NIC index for this buffer
fl->ifl_bus_addrs[i] = paddr + nm_get_offset(kring, slot);
fl->ifl_rxd_idxes[i] = nic_i;

// Load or reload DMA mapping if necessary
if (__predict_false(init)) {
    netmap_load_map(na, fl->ifl_buf_tag,
        map[nic_i], addr);
} else if (slot->flags & NS_BUF_CHANGED) {
    netmap_reload_map(na, fl->ifl_buf_tag,
        map[nic_i], addr);
}

// Synchronize DMA so the NIC can safely read the buffer
bus_dmamap_sync(fl->ifl_buf_tag, map[nic_i],
    BUS_DMASYNC_PREREAD);

// Clear "buffer changed" flag
slot->flags &= ~NS_BUF_CHANGED;

// Move to next position in both netmap and NIC rings (circular
    ↪ increment)
nm_i = nm_next(nm_i, lim);
nic_i = nm_next(nic_i, lim);
}

/*
 * Tell the hardware to make these new buffers available.
 * This happens once per batch for efficiency.
 */
iru.iru_pidx = nic_i_first;
iru.iru_count = i;
ctx->isc_rxd_refill(ctx->ifc_softc, &iru);
}

// Update software producer index
fl->ifl_pidx = nic_i;

// Ensure we refilled exactly up to the intended position
MPASS(nm_i == kring->rhead);
kring->nr_hwcur = nm_i;

// Final DMA sync for descriptors
```

```
bus_dmamap_sync(fl->ifl_ifdi->idi_tag, fl->ifl_ifdi->idi_map,
    BUS_DMASYNC_PREREAD | BUS_DMASYNC_PREWRITE);

// Flush the buffers to the NIC
ctx->isc_rxd_flush(ctx->ifc_softc, rxq->ifr_id, fl->ifl_id,
    nm_prev(nic_i, lim));
DBG_COUNTER_INC(rxd_flush);

return (0);
}
```

Understanding the while (n > 0) Loop in netmap_fl_refill()

The loop we're about to study looks like this:

```
while (n > 0) {
    ...
}
```

It comes from **iflib** (the Interface Library) in FreeBSD's network stack, in a section of code that connects **netmap** with network drivers.

Netmap is a high-performance packet I/O framework designed for very fast packet processing. In this context, the kernel uses the loop to **refill receive buffers**, ensuring the network interface card (NIC) always has space ready to store incoming packets, keeping data flowing smoothly at high speed.

Here, *n* is simply the number of buffers that still need to be prepared. The loop works through them in **efficient batches**, processing a few at a time until all are ready. This batching approach reduces overhead and is a common technique in high-performance network drivers.

What the while (n > 0) Really Does

As we've just seen, *n* is the count of receive buffers still waiting to be prepared. This loop's job is simple in concept:

“Work through those buffers in batches until there are none left.”

Each pass of the loop prepares a group of buffers and hands them off to the NIC. If there's still work to do, the loop runs again, ensuring that by the end, all required buffers are ready for incoming packets.

What Happens Inside the while (n > 0) Loop

Each time the loop runs, it processes one batch of buffers. Here's the breakdown:

1. **Debug Tracking:** If the driver is compiled with debugging enabled, it may update counters that track how often large batches of buffers are refilled. This is just for performance monitoring.
2. **Batch Setup:** The driver remembers where this batch starts (*nic_i_first*) so it can later tell the NIC exactly which slots were updated.
3. **Inner Batch Processing:** Inside the loop, there's another for loop that refills up to a maximum number of buffers at a time (*IFLIB_MAX_RX_REFRESH*). For each buffer in this batch:

- Look up the buffer's address and physical location in memory.
- Check if the buffer is valid. If not, reinitialise the receive ring.
- Store the physical address and slot index so the NIC knows where to place incoming data.
- If the buffer has changed or this is the first initialisation, update its DMA (Direct Memory Access) mapping.
- Synchronise the buffer for reading so the NIC can safely use it.
- Clear any "buffer changed" flags.
- Move to the next buffer position in the ring.

4. **Publishing the Batch to the NIC:** Once the batch is ready, the driver calls a function to tell the NIC:

"These new buffers are ready for use."

By breaking the work into manageable batches and looping until every buffer is ready, this while loop ensures the NIC is always prepared to receive incoming data without interruption. It's a small but necessary part of keeping packet flow continuous in a high-performance networking environment.

Even if some of the lower-level details (like DMA mapping or ring indices) aren't fully clear yet, the key takeaway is this:

Loops like this are the engine that quietly keeps the system running at full speed. As you progress through the book, these concepts will become second nature, and you'll start to recognise similar patterns across many parts of the FreeBSD kernel.

4.7.5 Understanding do...while Loops

A do...while loop is a variation of the while loop where the **loop body runs at least once**, and then repeats only **if the condition remains true**:

```
int i = 0;
do {
    printf("%d\n", i);
    i++;
} while (i < 10);
```

- The loop always executes the code inside at least once, even if the condition is false to begin with.
- Afterwards, it checks the condition (`i < 10`) to decide whether to repeat.

In the FreeBSD kernel, you'll often see this pattern inside macros designed to behave like single statements. For example, `sys/sys/timespec.h` defines the `TIMESPEC_TO_TIMEVAL` macro using exactly this idiom:

```
#define TIMESPEC_TO_TIMEVAL(tv, ts) \
do { \
    (tv)->tv_sec = (ts)->tv_sec; \
    (tv)->tv_usec = (ts)->tv_nsec / 1000; \
} while (0)
```

What This Macro Does

1. **Assign Seconds:** Copies `tv_sec` from the source (`ts`) to the target (`tv`).
2. **Convert and Assign Microseconds:** Divides `ts->tv_nsec` by 1000 to convert nanoseconds to microseconds and stores that in `tv_usec`.
3. `do...while (0)`: Wraps the two statements so that when this macro is used, it behaves syntactically like a single statement, even if followed by a semicolon, preventing issues in constructs like:

```
if (x) TIMESPEC_TO_TIMEVAL(tv, ts);
else ...
```

While `do...while (0)` may look odd, it's a solid C idiom used to make macro expansions safe and predictable in all contexts (like inside conditional statements). It ensures that the entire macro behaves like one statement and avoids accidentally creating half-executed code. Understanding this helps you read and avoid subtle bugs in kernel code that rely heavily on macros for clarity and safety.

4.7.6 Understanding `break` and `continue`

When working with loops in C, sometimes you need to change the normal flow:

1. `break`: Immediately exits the loop, even if the loop condition could still be true.
2. `continue`: Skips the rest of the current iteration and jumps directly to the loop's next iteration.

Here's a simple example:

```
for (int i = 0; i < 10; i++) {
    if (i == 5)
        continue; // Skip the number 5, move to the next i
    if (i == 8)
        break;    // Stop the loop entirely when i reaches 8
    printf("%d\n", i);
}
```

How This Works Step-by-Step

1. The loop starts with `i = 0` and runs while `i < 10`.
2. When `i == 5`, the `continue` statement runs:
 - The rest of the loop body is skipped.
 - The loop moves directly to `i++` and checks the condition again.
3. When `i == 8`, the `break` statement runs:
 - The loop stops immediately.
 - Control jumps to the first line of code after the loop.

Output of the Code

```
0
1
2
3
4
6
7
```

5 is skipped because of `continue`.

The loop ends at 8 because of `break`.

You can see a real example of `break` and `continue` usage in the function `if_purgeaddrs(ifp)` of `sys/net/if.c`.

```
/*
 * Remove any unicast or broadcast network addresses from an interface.
 */
void
if_purgeaddrs(struct ifnet *ifp)
{
    struct ifaddr *ifa;

#ifdef INET6
    /*
     * Need to leave multicast addresses of proxy NDP llentries
     * before in6_purgeifaddr() because the llentries are keys
     * for in6_multi objects of proxy NDP entries.
     * in6_purgeifaddr()s clean up llentries including proxy NDPs
     * then we would lose the keys if they are called earlier.
     */
    in6_purge_proxy_ndp(ifp);
#endif
    while (1) {
        struct epoch_tracker et;

        NET_EPOCH_ENTER(et);
        CK_STAILQ_FOREACH(ifa, &ifp->if_addrhead, ifa_link) {
            if (ifa->ifa_addr->sa_family != AF_LINK)
                break;
        }
        NET_EPOCH_EXIT(et);

        if (ifa == NULL)
            break;
    }
}
```

```

#ifdef INET
    /* XXX: Ugly!! ad hoc just for INET */
    if (ifa->ifa_addr->sa_family == AF_INET) {
        struct ifreq ifr;

        bzero(&ifr, sizeof(ifr));
        ifr.ifr_addr = *ifa->ifa_addr;
        if (in_control(NULL, SIOCDIFADDR, (caddr_t)&ifr, ifp,
            NULL) == 0)
            continue;
    }
#endif /* INET */
#ifdef INET6
    if (ifa->ifa_addr->sa_family == AF_INET6) {
        in6_purgeifaddr((struct in6_ifaddr *)ifa);
        /* ifp_addrhead is already updated */
        continue;
    }
#endif /* INET6 */
    IF_ADDR_WLOCK(ifp);
    CK_STAILQ_REMOVE(&ifp->if_addrhead, ifa, ifaddr, ifa_link);
    IF_ADDR_WUNLOCK(ifp);
    ifa_free(ifa);
}
}

```

What this function does

`if_purgeaddrs(ifp)` removes all non-link-layer addresses from a network interface. In plain words, it walks the list of addresses attached to the interface and deletes unicast or broadcast addresses that belong to IPv4 or IPv6. Some families are handled by calling helpers who update the lists for us. Anything not handled by a helper is explicitly removed and freed.

How the loop is organised

The outer while (1) repeats until there are no more removable addresses. Each pass:

1. Enters the network epoch (`NET_EPOCH_ENTER`) to safely walk the interface address list.
2. Scans the list with `CK_STAILQ_FOREACH` to find the **first address after the ‘AF_LINK entries**. Link-layer entries come first and are not purged here.
3. Leaves the epoch and then decides what to do with the address it found.

Where the break statements act

Break inside the list scan:


```
if (ifa->ifa_addr->sa_family != AF_LINK)
    break;
```

The scan stops as soon as it reaches the first non-AF_LINK address. We only need one target per pass.

Break after the scan:

```
if (ifa == NULL)
    break;
```

If the scan did not find any non-AF_LINK address, there is nothing left to purge. The outer while ends.

Where the continue statements act

IPv4 address handled by ioctl:

```
if (in_control(...) == 0)
    continue;
```

For IPv4, `in_control(SIOCDIFADDR)` removes the address and updates the list. Since that work is done, we skip the manual removal below and continue to the next outer-loop pass to look for the next address.

IPv6 address removed by helper:

```
in6_purgeifaddr((struct in6_ifaddr *)ifa);
/* list already updated */
continue;
```

For IPv6, `in6_purgeifaddr()` also updates the list. There is nothing more to do in this pass, so we continue to the next one.

The fallback removal path

If the address was neither handled by the IPv4 nor the IPv6 helpers, the code takes the generic path:

```
IF_ADDR_WLOCK(ifp);
CK_STAILQ_REMOVE(&ifp->if_addrhead, ifa, ifaddr, ifa_link);
IF_ADDR_WUNLOCK(ifp);
ifa_free(ifa);
```

This explicitly removes the address from the list and frees it.

In loops, `break` and `continue` are precision tools for controlling execution flow. In the `if_purgeaddrs()` function from FreeBSD's `sys/net/if.c`, `break` stops the search when there are no more addresses to remove, or halts the inner scan as soon as a target address is found. `continue` skips the generic removal step when a specialised IPv4 or IPv6 routine has already handled the work, jumping straight to the next pass through the

outer loop. This design lets the function repeatedly find one removable address at a time, remove it using the most appropriate method, and keep going until no non-link-layer addresses remain.

The key takeaway is that well-placed break and continue statements keep loops efficient and focused, avoiding wasted work and making the code's intent clear, a pattern you'll encounter often in FreeBSD's kernel for both clarity and performance.

4.7.7 Pro Tip: Always Use Braces {}

In C, if you omit braces after an if, only one statement is actually controlled by the if. This can easily lead to mistakes:

```
if (x > 0)
    printf("Positive\n");    // Runs only if x > 0
    printf("Always runs\n"); // Always runs! Not part of the if
```

This is a common source of bugs because the second printf appears to be inside the if, but it isn't.

To avoid confusion and accidental logic errors, always use braces, even for a single statement:

```
if (x > 0) {
    printf("Positive\n");
}
```

This makes your intent explicit, keeps your code safe from subtle changes, and follows the style used in the FreeBSD source tree.

Also Safer for Future Changes

When you always use braces, it's much safer to modify the code later:

```
if (x > 0) {
    printf("x is positive\n");
    log_positive(x);    // Adding this won't break logic!
}
```

4.7.8 Summary

In this section, you've learned:

- How to make decisions using if, else, and switch
- How to write loops using for, while, and do...while
- How to exit or skip iterations with break and continue
- How FreeBSD uses control flow to walk lists and make kernel decisions

You now have the tools to control the logic and flow of your programs, which is the core of programming itself.

4.8 Functions

In C, a **function** is like a dedicated workshop in a large factory; it is a self-contained area where a specific task is carried out, start to finish, without disturbing the rest of the production line. When you need that task done, you simply send the work there, and the function delivers the result.

Functions are one of the most important tools you have as a programmer because they let you:

- **Break down complexity:** Large programs become easier to understand when split into smaller, focused operations.
- **Reuse logic:** Once written, a function can be called anywhere, saving you from typing (and debugging) the same code repeatedly.
- **Improve clarity:** A descriptive function name turns a block of cryptic code into a clear statement of intent.

You've already seen functions at work:

- `main()`: the starting point of every C program.
- `printf()`: a library function that handles formatted output for you.

In the FreeBSD kernel, you'll find functions everywhere, from low-level routines that copy data between memory regions to specialised ones that communicate with hardware. For example, when a network packet arrives, the kernel doesn't put all the processing logic in one giant block of code. Instead, it calls a series of functions, each responsible for a clear, isolated step in the process.

In this section, you'll learn how to create your own functions, giving you the power to write clean, modular code. This isn't just good style in FreeBSD device driver development; it's the foundation for stability, reusability, and long-term maintainability.

How a Function Call Works in Memory

When your program calls a function, something important happens behind the scenes:

```
+-----+
| Return Address      | <- Where to resume execution after the function ends
+-----+
| Function Arguments  | <- Copies of the values you pass in
+-----+
| Local Variables     | <- Created fresh for this call
+-----+
| Temporary Data      | <- Space the compiler needs for calculations
+-----+
... Stack ...
```

Step-by-step:

1. **The caller pauses** - your program stops at the function call and saves the **return address** on the stack so it knows where to continue afterwards.
2. **Arguments are placed** - the values you pass to the function (parameters) are stored, either in registers or on the stack, depending on the platform.

3. **Local variables are created:** the function gets its own workspace in memory, separate from the caller's variables.
4. **The function runs:** it executes its statements in order, possibly calling other functions along the way.
5. **A return value is sent back:** if the function produces a result, it is placed in a register (commonly `eax` on x86) for the caller to pick up.
6. **Cleanup and resume:** the function's workspace is removed from the stack, and the program continues where it left off.

Why do you need to understand that?

In kernel programming, every function call has a cost in both time and memory. Understanding this process will help you write efficient driver code and avoid subtle bugs, especially when working with low-level routines where stack space is limited.

4.8.1 Defining and Declaring Functions

Every function in C follows a simple recipe. To create one, you need to specify four things:

1. **Return type:** what kind of value the function gives back to the caller.
 - Example: `int` means the function will return an integer.
 - If it doesn't return anything, we use the keyword `void`.
2. **Name:** a unique, descriptive label for your function, so you can call it later.
 - Example: `read_temperature()` is much clearer than `rt()`.
3. **Parameters:** zero or more values the function needs to do its job.
 - Each parameter has its own type and name.
 - If there are no parameters, use `void` in the list to make it explicit.
4. **Body:** the block of code, enclosed in `{}` braces, that performs the task.
 - This is where you write the actual instructions.

General form:

```
return_type function_name(parameter_list)
{
    // statements
    return value; // if return_type is not void
}
```

Example: A function to add two numbers and return the result

```
int add(int a, int b)
{
    int sum = a + b;
    return sum;
}
```

Declaration vs. Definition

A lot of beginners get tripped up here, so let's make it crystal clear:

- **Declaration** tells the compiler that a function exists, what it's called, what parameters it takes, and what it returns, but it does not provide the code for it.
- **Definition** is where you actually write the body of the function, the full implementation that does the work.

Think of it like planning and building a workshop:

- **Declaration:** putting up a sign saying *"This workshop exists, here's what it's called, and here's the kind of work it does."*
- **Definition:** actually building the workshop, stocking it with tools, and hiring workers to do the job.

Example:

```
// Function declaration (prototype)
int add(int a, int b);

// Function definition
int add(int a, int b)
{
    int sum = a + b;
    return sum;
}
```

Why declarations are useful

In small single-file programs, you can just put the definition before you call the function and be done. But in larger programs, especially in FreeBSD drivers, code is often split across many files.

For example:

- The function `mydevice_probe()` might be **defined** in `mydevice.c`.
- Its **declaration** will go into a header file `mydevice.h` so that other parts of the driver, or even the kernel, can call it without knowing the details of how it works.

When the compiler sees the declaration, it knows how to check that calls to `mydevice_probe()` use the right number and types of parameters, even before it sees the definition.

FreeBSD driver perspective

When writing a driver:

- Declarations often live in `.h` header files.
- Definitions live in `.c` source files.
- The kernel will call your driver's functions (like `probe()`, `attach()`, `detach()`) based on the declarations it sees in your driver's headers, without caring exactly how you implement them as long as the signatures match.

Understanding this difference will save you a lot of compiler errors, especially “implicit declaration of function” or “undefined reference” errors, which are among the most common mistakes beginners hit when starting with C.

How Declarations and Definitions Work Together

In small programs, you might write the function’s definition before `main()` and be done. But in real projects, like a FreeBSD device driver, code is split into header files (`.h`) for declarations and source files (`.c`) for definitions.

```
+-----+
| mydevice.h | <-- Header file
|-----|
| int my_probe(void); | // Declaration (prototype)
| int my_attach(void); | // Declaration
+-----+
      |
      v
+-----+
| mydevice.c | <-- Source file
|-----|
| #include "mydevice.h" | // Include declarations
|
| int my_probe(void) | // Definition
| {
|     /* detect hw */
|     return 0;
| }
|
| int my_attach(void) | // Definition
| {
|     /* init hw */
|     return 0;
| }
+-----+
```

How it works:

1. Declaration in the header file tells the compiler: *“These functions exist somewhere, here’s what they look like.”*
2. Definition in the source file provides the actual code.
3. Any other `.c` file that includes `mydevice.h` can now call these functions, and the compiler will check the parameters and return types.
4. At link time, the function calls are connected to their definitions.

In the context of FreeBSD drivers:

- You might have `mydevice.c` containing the driver logic, and `mydevice.h` holding the function declarations shared across the driver.
- The kernel build system will compile your `.c` files and link them into a kernel module.
- If the declarations don't match the definitions exactly, you'll get compiler errors, which is why keeping them in sync is critical.

Common mistakes with functions and how to fix them

- 1) Calling a function before the compiler knows it exists Symptom: "implicit declaration of function" warning or error. Fix: Add a declaration in a header file and include it, or place the definition above its first use.
- 2) Declaration and definition do not match Symptom: "conflicting types" or odd runtime bugs. Fix: Make the signature identical in both places. Same return type, parameter types, and qualifiers in the same order.
- 3) Forgetting `void` for a function with no parameters Symptom: The compiler may think the function takes unknown arguments. Fix: Use `int my_fn(void)` instead of `int my_fn()`.
- 4) Returning a value from a `void` function or forgetting to return a value Symptom: "void function cannot return a value" or "control reaches end of non-void function." Fix: For non-void functions, always return the right type. For `void`, do not return a value.
- 5) Returning pointers to local variables Symptom: Random crashes or garbage data. Fix: Do not return the address of a stack variable. Use dynamically allocated memory or pass a buffer in as a parameter.
- 6) Mismatched `const` or pointer levels between declaration and definition Symptom: Type mismatch errors or subtle bugs. Fix: Keep qualifiers consistent. If the declaration has `const char *`, the definition must match exactly.
- 7) Multiple definitions across files Symptom: Linker error "multiple definition of ...". Fix: Only one definition per function. If a helper should be private to a file, mark it `static` in that `.c` file.
- 8) Putting function definitions in headers by accident Symptom: Multiple definition linker errors when the header is included by several `.c` files. Fix: Headers should usually have declarations only. If you really need code in a header, make it `static inline` and keep it small.
- 9) Missing includes for functions you call Symptom: Implicit declarations or wrong default types. Fix: Include the correct system or project header that declares the function you are calling, for example `#include <stdio.h>` for `printf`.
- 10) Kernel specific: undefined symbols when building a module Symptom: Linker error "undefined reference" while building your KMOD. Fix: Ensure the function is actually defined in your module or exported by the kernel, that the declaration matches the definition, and that the right source files are part of the module build.
- 11) Kernel specific: using a helper that is meant to be file local Symptom: "undefined reference" from other files or unexpected symbol visibility. Fix: Mark internal helpers as `static` to restrict visibility. Expose only what other files must call through your header.
- 12) Choosing poor names Symptom: Hard to read code and name collisions. Fix: Use descriptive, project prefixed names, for example `mydev_read_reg`, not `readreg`.

Hands-on exercise: Splitting Declarations and Definitions

For this exercise, we will create 3 files.

`mydevice.h` - This header file declares the functions and makes them available to any `.c` file that includes it.

```
#ifndef MYDEVICE_H
#define MYDEVICE_H

// Function declarations (prototypes)
void mydevice_probe(void);
void mydevice_attach(void);
void mydevice_detach(void);

#endif // MYDEVICE_H
```

`mydevice.c` - This source file contains the actual definitions (the working code).

```
#include <stdio.h>
#include "mydevice.h"

// Function definitions
void mydevice_probe(void)
{
    printf("[mydevice] Probing hardware... done.\n");
}

void mydevice_attach(void)
{
    printf("[mydevice] Attaching device and initializing resources...\n");
}

void mydevice_detach(void)
{
    printf("[mydevice] Detaching device and cleaning up.\n");
}
```

`main.c` - This is the “user” of the functions. It just includes the header and calls them.

```
#include "mydevice.h"

int main(void)
{
    mydevice_probe();
    mydevice_attach();
    mydevice_detach();
}
```



```
    return 0;  
}
```

How to Compile and Run on FreeBSD

Open a terminal in the folder with the three files and run:

```
cc -Wall -o myprogram main.c mydevice.c  
./myprogram
```

Expected output:

```
[mydevice] Probing hardware... done.  
[mydevice] Attaching device and initialising resources...  
[mydevice] Detaching device and cleaning up.
```

Why this matters for FreeBSD driver development

In a real FreeBSD kernel module,

- `mydevice.h` would hold your driver's public API (function declarations).
- `mydevice.c` would have the full implementations of those functions.
- The kernel (or other parts of the driver) would include the header to know how to call into your code, without needing to see the actual implementation details.

This exact pattern is how `probe()`, `attach()`, and `detach()` routines are structured in actual device drivers. Learning it now will make those later chapters feel familiar.

Understanding the relationship between declarations and definitions is a cornerstone of C programming, and it becomes even more important when you step into the world of FreeBSD device drivers. In kernel development, functions are rarely defined and used in the same file; they are spread across multiple source and header files, compiled separately, and linked together into a single module. A clear separation between **what a function does** (its declaration) and **how it does it** (its definition) keeps code organized, reusable, and easier to maintain. Master this concept now, and you'll be well-prepared for the more complex modular structures you'll encounter when we begin building real kernel drivers.

4.8.2 Calling Functions

Once you've defined a function, the next step is to call it, that is, to tell the program, *"Hey, go run this block of code now and give me the result"*.

Calling a function is as simple as writing its name followed by parentheses containing any required arguments.

If the function returns a value, you can store that value in a variable, pass it to another function, or use it directly in an expression.

Example:

```
int result = add(3, 4);
printf("Result is %d\n", result);
```

Here's what happens step-by-step when this code runs:

1. The program encounters `add(3, 4)` and pauses its current work.
2. It jumps to the `add()` function's definition, giving it two arguments: 3 and 4.
3. Inside `add()`, the parameters `a` and `b` receive the values 3 and 4.
4. The function calculates `sum = a + b` and then executes `return sum;`.
5. The returned value 7 travels back to the calling point and gets stored in the variable `result`.
6. The `printf()` function then displays:

```
Result is 7
```

FreeBSD Driver Connection

When you call a function in a FreeBSD driver, you're often asking the kernel or your own driver logic to perform a very specific task, for example:

- Calling `bus_space_read_4()` to read a 32-bit hardware register.
- Calling your own `mydevice_init()` to prepare a device for use.

The principle is exactly the same as the `add()` example:

The function takes parameters, does its job, and returns control to where it was called. The difference in kernel space is that the "job" might involve talking directly to hardware or managing system resources, but the calling process is identical.

Tip for Beginners Even if a function doesn't return a value (its return type is `void`), calling it still triggers its entire body to run. In drivers, many important functions don't return anything but perform critical work like initializing hardware or setting up interrupts.

Function Call Flow When your program calls a function, control jumps from the current point in your code to the function's definition, runs its statements, and then comes back. Example flow for `add(3, 4)` inside `main()`:

```
main() starts
|
v
Calls add(3, 4) -----+
|                         |
v                         |
Inside add():             |
a = 3                     |
b = 4                     |
sum = a + b // sum=7      |
return sum;               |
|                         |
```

```

+----- Back to main()
|
v

result = 7
printf("Result is 7")
main() ends

```

What to notice:

- The program's "path" temporarily leaves `main()` when the function is called.
- The parameters in the function get copies of the values passed in.
- The return statement sends a value back to where the function was called.
- After the call, execution continues right where it left off.

FreeBSD driver analogy:

When the kernel calls your driver's `attach()` function, the exact same process happens. The kernel jumps into your code, you run your initialization logic, and then control returns to the kernel so it can continue loading devices. Whether in user space or kernel space, function calls follow the same flow.

Try It Yourself: Simulating a Driver Function Call

In this exercise, you'll write a small program that mimics calling a driver function to read a "hardware register" value.

We'll simulate it in user space so you can compile and run it easily on your FreeBSD system.

Step 1: Define the function

Create a file called `driver_sim.c` and start with this function:

```

#include <stdio.h>

// Function definition
unsigned int read_register(unsigned int address)
{
    // Simulated register value based on address
    unsigned int value = address * 2;
    printf("[driver] Reading register at address 0x%X...\n", address);
    return value;
}

```

Step 2: Call the function from `main()`

In the same file, add `main()` below your function:

```

int main(void)
{
    unsigned int reg_addr = 0x10; // pretend this is a hardware register address
}

```

```
    unsigned int data;

    data = read_register(reg_addr); // Call the function
    printf("[driver] Value read: 0x%X\n", data);

    return 0;
}
```

Step 3: Compile and run

```
% cc -Wall -o driver_sim driver_sim.c
./driver_sim
```

Expected output:

```
[driver] Reading register at address 0x10...
[driver] Value read: 0x20
```

What You Learned * You called a function by name, passing it a parameter. * The parameter got a copy of your value (0x10) inside the function. * The function calculated a result and sent it back with return. * Execution continued exactly where it left off.

In a real driver, `read_register()` might use the `bus_space_read_4()` kernel API to access a physical hardware register instead of multiplying a number. The function call flow, however, is exactly the same.

4.8.3 Functions with No Return Value: void

Not every function needs to send data back to the caller.

Sometimes, you just want the function to do something, print a message, initialise hardware, log a status and then finish.

In C, when a function **does not return anything**, you declare its return type as void.

Example:

```
void say_hello(void)
{
    printf("Hello, World!\n");
}
```

Here's what's happening:

- void before the name means: *"This function will not return a value"*.
- The (void) in the parameter list means: *"This function takes no arguments"*.

- Inside the braces {}, we place the statements we want to execute when the function is called.

Calling it:

```
say_hello();
```

This will print:

```
Hello, World!
```

Common beginner mistakes with void functions

1. Forgetting void in the parameter list

```
void say_hello()      // Works, but less explicit - avoid in new code
void say_hello(void)  // Best practice
```

In old C code, () without void means “this function takes an unspecified number of arguments”, which can cause confusion.

2. Trying to return a value from a void function

```
void test(void)
{
    return 42; // Compiler error
}
```

3. Assigning the result of a void function

```
int x = say_hello(); // Compiler error
```

Now that you’ve seen the most common pitfalls, let’s take a step back and understand why the void keyword is important in the first place.

Why void matters

Marking a function with void clearly tells both the compiler and human readers that this function’s purpose is to perform an action, not to produce a result.

If you try to use the “return value” from a void function, the compiler will stop you, which helps catch mistakes early.

FreeBSD driver perspective

In FreeBSD drivers, many important functions are void because they are all about doing work, not returning data.

For example:

- `mydevice_reset(void)`: might reset the hardware to a known state.
- `mydevice_led_on(void)`: might turn on a status LED.

- `mydevice_log_status(void)`: might print debugging information to the kernel log.

The kernel doesn't care about a return value in these cases, it just expects your function to perform its action.

While `void` functions in drivers don't return values, that doesn't mean they can't communicate important information. There are still several ways to signal events or issues back to the rest of the system.

Tip for Beginners

In driver code, even though `void` functions don't return data, they can still report errors or events by:

- Writing to a global or shared variable.
- Logging messages with `device_printf()` or `printf()`.
- Triggering other functions that handle error states.

Understanding `void` functions is important because in real-world FreeBSD driver development, not every task produces data to return; many simply perform an action that prepares the system or the hardware for something else. Whether it's initializing a device, cleaning up resources, or logging a status message, these functions still play a critical role in the overall behavior of your driver. By recognizing when a function should return a value and when it should simply do its job and return nothing, you'll write cleaner, more purposeful code that matches the way the FreeBSD kernel itself is structured.

4.8.4 Function Declarations (Prototypes)

In C, it's a good habit and often essential to **declare** a function before you use it.

A function declaration, also called a **prototype**, tells the compiler:

- The function's name.
- The type of value it returns (if any).
- The number, types, and order of its parameters.

This way, the compiler can check that your function calls are correct, even if the actual definition (the body of the function) appears later in the file or in a different file entirely.

Let's see a basic example

```
#include <stdio.h>

// Function declaration (prototype)
int add(int a, int b);

int main(void)
{
    int result = add(3, 4);
    printf("Result: %d\n", result);
    return 0;
}

// Function definition
```

```
int add(int a, int b)
{
    return a + b;
}
```

When the compiler reads the prototype for `add()` before `main()`, it immediately knows:

- the function's name is `add`,
- it takes two `int` parameters, and
- it will return an `int`.

Later, when the compiler finds the definition, it checks that the name, parameters, and return type match the prototype exactly. If they don't, it raises an error.

4.8.5 Why prototypes matter

Placing the prototype before a function is called provides several benefits:

1. **Prevents unnecessary warnings and errors:** If you call a function before the compiler knows it exists, you'll often get an *"implicit declaration of function"* warning or even a compilation error.
2. **Catches mistakes early:** If your call passes the wrong number or types of arguments, the compiler will flag the problem immediately instead of letting it cause unpredictable behaviour at runtime.
3. **Enables modular programming:** Prototypes allow you to split your program into multiple source files. You can keep the function definitions in one file and the calls to them in another, with the prototypes stored in a shared header file.

By declaring your functions before you use them, either at the top of your `.c` file or in a `.h` header, you're not just keeping the compiler happy; you're building code that's easier to organise, maintain, and scale.

Now that you understand why prototypes are important, let's look at the two most common places to put them: directly in your `.c` file or in a shared header file.

4.8.6 Prototypes in header files

Although you can write prototypes directly at the top of your `.c` file, the more common and scalable approach is to place them in **header files** (`.h`).

This allows multiple `.c` files to share the same declarations without repeating them.

Example:

`mathutils.h`

```
#ifndef MATHUTILS_H
#define MATHUTILS_H

int add(int a, int b);
```

```
int subtract(int a, int b);

#endif // MATHUTILS_H
```

main.c

```
#include <stdio.h>
#include "mathutils.h"

int main(void)
{
    printf("Sum: %d\n", add(3, 4));
    printf("Difference: %d\n", subtract(10, 4));
    return 0;
}
```

mathutils.c

```
#include "mathutils.h"

int add(int a, int b)
{
    return a + b;
}

int subtract(int a, int b)
{
    return a - b;
}
```

This pattern keeps your code organized and avoids having to manually keep multiple prototypes in sync across files.

4.8.7 FreeBSD driver perspective

In FreeBSD driver development, prototypes are essential because the kernel often needs to call into your driver without knowing how your functions are implemented.

For example, in your driver's header file you might declare:

```
int mydevice_init(void);
void mydevice_start_transmission(void);
```


These tell the kernel or bus subsystem that your driver has these functions available, even if the actual definitions live deep inside your .c files.

The build system compiles all the pieces together and links the calls to the correct implementations.

4.8.8 Try It Yourself: Moving a Function Below main()

One of the main reasons to use prototypes is so you can call a function that hasn't been defined yet in the file. Let's see this in action.

Step 1: Start without a prototype

```
#include <stdio.h>

int main(void)
{
    int result = add(3, 4); // This will cause a compiler warning or error
    printf("Result: %d\n", result);
    return 0;
}

// Function definition
int add(int a, int b)
{
    return a + b;
}
```

Compile it:

```
cc -Wall -o testprog testprog.c
```

You'll likely get a warning such as:

```
testprog.c:5:18:
warning: call to undeclared function 'add'; ISO C99 and later do not support implicit
↪ function declarations [-Wimplicit-function-declaration]
    5 |     int result = add(3, 4); // This will cause a compiler warning or error
      |                   ^
1 warning generated.
```

Step 2: Fix it with a prototype

Add the function prototype before main() like this:

```
#include <stdio.h>

// Function declaration (prototype)
int add(int a, int b);

int main(void)
{
    int result = add(3, 4); // No warnings now
    printf("Result: %d\n", result);
    return 0;
}

// Function definition
int add(int a, int b)
{
    return a + b;
}
```

Recompile, the warning is gone, and the program runs:

```
Result: 7
```

Note: Depending on the compiler you use, the warning message might look a little different from the example shown above, but the meaning will be the same.

By adding a prototype, you’ve just seen how the compiler can recognize a function and validate its use even before it sees the actual code. This same principle is what allows the FreeBSD kernel to call into your driver; it doesn’t need the whole function body up front, only the declaration. In the next section, we’ll look at how this works in a real driver, where prototypes in header files act as the kernel’s “map” to your driver’s capabilities.

4.8.9 FreeBSD Driver Connection

In the FreeBSD kernel, function prototypes are the way the system “introduces” your driver’s functions to the rest of the codebase.

When the kernel wants to interact with your driver, it doesn’t search for the function’s code directly; it relies on the function’s declaration to know the name, parameters, and return type.

For example, during device detection, the kernel might call your `probe()` function to check whether a specific piece of hardware is present. The actual definition of `probe()` could be deep inside your `mydriver.c` file, but the **prototype** lives in your driver’s header file (`mydriver.h`). That header is included by the kernel or bus subsystem so it can compile code that calls `probe()` without needing to see its full implementation.

This arrangement ensures two critical things:

1. **Compiler validation:** The compiler can confirm that any calls to your functions use the correct parameters and return type.

2. **Linker resolution:** When building the kernel or your driver module, the linker knows exactly which compiled function body to connect to the calls.

Without correct prototypes, the kernel build could fail or, worse, compile but behave unpredictably at runtime. In kernel programming, that's not just a bug, it could mean a crash.

Example: Prototypes in a FreeBSD Driver

`mydriver.h`, the driver header file with prototypes:

```
#ifndef _MYDRIVER_H_
#define _MYDRIVER_H_

#include <sys/types.h>
#include <sys/bus.h>

// Public entry points: declared here so bus/framework code can call them
// These are the entry points the kernel will call during the device's lifecycle
// Function prototypes (declarations)
int mydriver_probe(device_t dev);
int mydriver_attach(device_t dev);
int mydriver_detach(device_t dev);

#endif /* _MYDRIVER_H_ */
```

Here, we declare three key entry points `probe()`, `attach()`, and `detach()`, but don't include their bodies.

The kernel or bus subsystem will include this header so it knows how to call these functions during device lifecycle events.

`mydriver.c`, the driver source file with definitions:

```
/*
 * FreeBSD device driver lifecycle (quick map)
 *
 * 1) Kernel enumerates devices on a bus
 *    The bus framework walks hardware and creates device_t objects.
 *
 * 2) probe()
 *    The kernel asks your driver if it supports a given device.
 *    You inspect IDs or capabilities and return a score.
 *    - Return ENXIO if this driver does not match.
 *    - Return BUS_PROBE_DEFAULT or a better score if it matches.
 *
 * 3) attach()
 *    Called after a successful probe to bring the device online.
 *    Typical work:
```

```

*      - Allocate resources (memory, IRQ) with bus_alloc_resource_any()
*      - Map registers and set up bus_space
*      - Initialize hardware to a known state
*      - Set up interrupts and handlers
*      Return 0 on success, or an errno if something fails.
*
* 4) Runtime
*      Your driver services requests. This may include:
*      - Interrupt handlers
*      - I/O paths invoked by upper layers or devfs interfaces
*      - Periodic tasks, callouts, or taskqueues
*
* 5) detach()
*      Called when the device is being removed or the module unloads.
*      Cleanup tasks:
*      - Quiesce hardware, stop DMA, disable interrupts
*      - Tear down handlers and timers
*      - Unmap registers and release resources with bus_release_resource()
*      Return 0 on success, or an errno if detach must be denied.
*
* 6) Optional lifecycle events
*      - suspend() and resume() during power management
*      - shutdown() during system shutdown
*
* Files to remember
*      - mydriver.h declares the entry points that the kernel and bus code will call
*      - mydriver.c defines those functions and contains the implementation details
*/
#include <sys/param.h>
#include <sys/kernel.h>
#include <sys/module.h>
#include <sys/bus.h>
#include <sys/systm.h>    // device_printf
#include "mydriver.h"

/*
* mydriver_probe()
* Called early during device enumeration.
* Purpose: decide if this driver matches the hardware represented by dev.
* Return: BUS_PROBE_DEFAULT for a normal match, a better score for a strong match,
*         or ENXIO if the device is not supported.
*/
int
mydriver_probe(device_t dev)

```



```
{
    device_printf(dev, "Probing device...\n");

    /*
     * Here you would usually check vendor and device IDs or use bus-specific
     * helper routines. If the device is not supported, return (ENXIO).
     */

    return (BUS_PROBE_DEFAULT);
}

/*
 * mydriver_attach()
 * Called after a successful probe when the kernel is ready to attach the device.
 * Purpose: allocate resources, map registers, initialise hardware, register
 * ↪ interrupts,
 *         and make the device ready for use.
 * Return: 0 on success, or an errno value (like ENOMEM or EIO) on failure.
 */
int
mydriver_attach(device_t dev)
{
    device_printf(dev, "Attaching device and initializing resources...\n");

    /*
     * Typical steps you will add here:
     * 1) Allocate device resources (I/O memory, IRQs) with bus_alloc_resource_any().
     * 2) Map register space and set up bus_space tags and handles.
     * 3) Initialise hardware registers to a known state.
     * 4) Set up interrupt handlers if needed.
     * 5) Create device nodes or child devices if this driver exposes them.
     * On any failure, release what you allocated and return an errno.
     */

    return (0);
}

/*
 * mydriver_detach()
 * Called when the device is being detached or the module is unloading.
 * Purpose: stop the hardware, free resources, and leave the system clean.
 * Return: 0 on success, or an errno value if detach must be refused.
 */
int
```

```
mydriver_detach(device_t dev)
{
    device_printf(dev, "Detaching device and cleaning up...\n");

    /*
     * Typical steps you will add here:
     * 1) Disable interrupts and stop DMA or timers.
     * 2) Tear down interrupt handlers.
     * 3) Unmap register space and free bus resources with bus_release_resource().
     * 4) Destroy any device nodes or sysctl entries created at attach time.
     */

    return (0);
}
```

Why this works:

- The `.h` file exposes only the **function interfaces** to the rest of the kernel.
- The `.c` file contains the **full implementations of the functions declared in the header**.
- The build system compiles all the source files, and the linker connects calls to the correct function bodies.
- The kernel can call these functions without knowing how they work internally; it only needs the prototypes.

Understanding how the kernel uses your driver's function prototypes is more than just a formality; it's a safeguard for correctness and stability. In kernel programming, even a slight mismatch between a declaration and a definition can lead to build failures or unpredictable runtime behaviour. That's why experienced FreeBSD developers follow a few best practices to keep their prototypes clean, consistent, and easy to maintain. Let's go over some of those tips next.

4.8.10 Tip for Kernel Code

When you start writing FreeBSD drivers, function prototypes aren't just a formality; they're a key part of keeping your code organised and error-free in a large, multi-file project. In the kernel, where functions are often called from deep within the system, a mismatch between a declaration and its definition can cause build failures or subtle bugs that are hard to track down.

To avoid problems and keep your headers clean:

- **Always match parameter types exactly** between the declaration and the definition; the return type, parameter list, and order must be identical.
- **Include qualifiers like `const` and `*` consistently** so you don't accidentally change how parameters are treated between the declaration and the definition.
- **Group related prototypes together** in header files so they're easy to find. For example, put all initialisation functions in one section, and hardware access functions in another.

Function prototypes may seem like a small detail in C, but they are the glue that holds multi-file projects and especially kernel code together. By declaring your functions before they are used, you give the compiler the

information it needs to catch mistakes early, keep your code organised, and allow different parts of a program to communicate cleanly.

In FreeBSD driver development, well-structured prototypes in header files enable the kernel to interact with your driver reliably, without knowing its internal details. Mastering this habit now is non-negotiable if you want to write stable, maintainable drivers.

In the next section, we'll explore real examples from the FreeBSD source tree to see exactly how prototypes are used throughout the kernel, from core subsystems to actual device drivers. This will not only reinforce what you've learned here, but also help you recognise the patterns and conventions that experienced FreeBSD developers follow every day.

4.8.11 Real Example from the FreeBSD 14.3 Source Tree: `device_printf()`

Now that you understand how function declarations and definitions work, let's walk through a concrete example from the FreeBSD kernel. We will follow `device_printf()` from its prototype in a header, to its definition in the kernel source, and finally to a real driver that calls it during initialisation. This shows the full path a function takes in real code and why prototypes are critical in driver development.

1) Prototype: where it is declared

The `device_printf()` function is declared in the FreeBSD kernel's bus interface header `sys/sys/bus.h`. Any driver source that includes this header can call it safely because the compiler knows its signature in advance.

```
int device_printf(device_t dev, const char *, ...) __printflike(2, 3);
```

What each part means:

- `int` is the return type. The function returns the number of characters printed, similar to `printf(9)`.
- `device_t dev` is a handle to the device that owns the message, which allows the kernel to prefix the output with the device name and unit, for example `vtnet0:`.
- `const char *` is the format string, the same idea used by `printf`.
- `...` indicates a variable argument list. You can pass values that match the format string.
- `__printflike(2, 3)` is a compiler hint used in FreeBSD. It tells the compiler that parameter 2 is the format string and that type checking for additional arguments starts at parameter 3. This enables compile time checks for format specifiers and argument types.

Because this declaration lives in a shared header, any driver that includes `<sys/sys/bus.h>` can call `device_printf()` without needing to know how it is implemented.

2) Definition: where it is implemented

Here is the actual implementation of `device_printf()` in `sys/kern/subr_bus.c` from **FreeBSD 14.3**. The function builds a prefix with the device name and unit, appends your formatted message, and counts how many characters are produced. I have added extra comments to help you understand how this function works.

```
/**
 * @brief Print the name of the device followed by a colon, a space
 * and the result of calling vprintf() with the value of @p fmt and
```

```

* the following arguments.
*
* @returns the number of characters printed
*/
int
device_printf(device_t dev, const char * fmt, ...)
{
    char buf[128];                // Fixed buffer for sbuf to use
    struct sbuf sb;                // sbuf structure that manages safe
    ↪ string building
    const char *name;             // Will hold the device's base name
    ↪ (e.g., "igc")
    va_list ap;                   // Handle for variable argument list
    size_t retval;                // Count of characters produced by
    ↪ the drain

    retval = 0;                   // Initialise the output counter

    sbuf_new(&sb, buf, sizeof(buf), SBUF_FIXEDLEN);
    ↪ Initialise sbuf 'sb' over 'buf'
    ↪ with fixed length

    sbuf_set_drain(&sb, sbuf_printf_drain, &retval);
    ↪ Set a "drain" callback that counts
    ↪ characters
    ↪ Every time sbuf emits bytes,
    ↪ sbuf_printf_drain
    ↪ updates 'retval' through this pointer

    name = device_get_name(dev);  // Query the device base name (may
    ↪ be NULL)

    if (name == NULL)             // If we do not know the name
        sbuf_cat(&sb, "unknown: "); // Prefix becomes "unknown: "
    else
        sbuf_printf(&sb, "%s%d: ", name, device_get_unit(dev));
    ↪ Otherwise prefix "name" + unit
    ↪ number, e.g., "igc0: "

    va_start(ap, fmt);            // Start reading the variable
    ↪ arguments after 'fmt'
    sbuf_vprintf(&sb, fmt, ap);    // Append the formatted message into
    ↪ the sbuf
    va_end(ap);                   // Clean up the variable argument list

```

```

        sbuf_finish(&sb);                                // Finalise the sbuf so its contents
↪ are complete
        sbuf_delete(&sb);                                // Release sbuf resources associated
↪ with 'sb'

        return (retval);                                // Return the number of characters
        ↪ printed
    }

```

What to notice

- The code uses sbuf to assemble the message safely. The drain callback updates retval so the function can return the number of characters produced.
- The device prefix comes from device_get_name() and device_get_unit(). If the name is not available, it falls back to unknown:.
- It accepts a format string and variable arguments, handled by va_list, va_start, and va_end, then forwards them to sbuf_vprintf().

3) Real driver use: where it is called in practice

Here is a clear example from sys/dev/virtio/virtqueue.c that calls device_printf() while initialising a virtqueue to use indirect descriptors. And like I did for step 2 above, I have added extra comments to help you understand how this function works.

```

static int
virtqueue_init_indirect(struct virtqueue *vq, int indirect_size)
{
    device_t dev;
    struct vq_desc_extra *dvp;
    int i, size;

    dev = vq->vq_dev;                                // Cache the device handle for
↪ logging and feature checks

    if (VIRTIO_BUS_WITH_FEATURE(dev, VIRTIO_RING_F_INDIRECT_DESC) == 0) {
        /*
         * Driver asked to use indirect descriptors, but the device did
         * not negotiate this feature. We do not fail the init here.
         * Return 0 so the queue can still be used without this feature.
         */
        if (bootverbose)
            device_printf(dev, "virtqueue %d (%s) requested "
                           "indirect descriptors but not negotiated\n",
                           vq->vq_queue_index, vq->vq_name);
    }
}

```

```

        return (0);                                // Continue without indirect
        ↪ descriptors
    }

    size = indirect_size * sizeof(struct vring_desc); // Total bytes for one
    ↪ indirect table
    vq->vq_max_indirect_size = indirect_size;        // Remember maximum entries
    ↪ per indirect table
    vq->vq_indirect_mem_size = size;                 // Remember bytes per indirect
    ↪ table
    vq->vq_flags |= VIRTQUEUE_FLAG_INDIRECT;         // Mark the queue as using
    ↪ indirect descriptors

    for (i = 0; i < vq->vq_nentries; i++) {          // For each descriptor in the
        ↪ main queue
        dxp = &vq->vq_descx[i];                     // Access per-descriptor extra
        ↪ bookkeeping

        dxp->indirect = malloc(size, M_DEVBUF, M_NOWAIT);
                                                    // Allocate an indirect descriptor
                                                    ↪ table for this entry

        if (dxp->indirect == NULL) {
            device_printf(dev, "cannot allocate indirect list\n");
                                                    // Tag the error with the device
                                                    ↪ name and unit
            return (ENOMEM);                        // Tell the caller that
            ↪ allocation failed
        }

        dxp->indirect_paddr = vtophys(dxp->indirect);
                                                    // Record the physical address
                                                    ↪ for DMA use
        virtqueue_init_indirect_list(vq, dxp->indirect);
                                                    // Initialise the table contents
                                                    ↪ to a known state
    }

    return (0);                                // Success. The queue now
        ↪ supports indirect descriptors
}

```

What this driver code is doing

This helper prepares a virtqueue to use indirect descriptors, a VirtIO feature that allows each top level descriptor to reference a separate table of descriptors. That makes it possible to describe larger I/O requests efficiently.

The function first checks whether the device actually negotiated the `VIRTIO_RING_F_INDIRECT_DESC` feature. If not, and if `bootverbose` is enabled, it uses `device_printf()` to log an informative message that includes the device prefix, then carries on without the feature. If the feature is present, it computes the size of the indirect descriptor table, marks the queue as indirect capable, and iterates over every descriptor in the ring. For each one it allocates an indirect table, logs an error with `device_printf()` if allocation fails, records the physical address for DMA, and initialises the table. This is a typical pattern in real drivers: check a feature, allocate resources, log meaningful messages tagged with the device, and handle errors cleanly.

Why this example matters

You have now seen the full sequence:

- **Prototype** in a shared header tells the compiler how to call the function and enables compile time checks.
- **Definition** in the kernel source implements the behaviour, using helpers like `sbuf` to assemble messages safely.
- **Real usage** in a driver shows how the function is called during initialisation and error paths, producing logs that are easy to trace back to a specific device.

This is the same pattern you will follow when writing your own driver helpers. Declare them in your header so the rest of the driver, and sometimes the kernel, can call them. Implement them in your `.c` files with small, focused logic. Call them from `probe()`, `attach()`, interrupt handlers, and teardown. Prototypes are the bridge that lets these pieces work together cleanly.

By now, you've seen how a function prototype, its implementation, and its real-world usage come together inside the FreeBSD kernel. From the declaration in a shared header, through the implementation in kernel code, to the call site inside a real driver, each step shows why prototypes are the “glue” that lets different parts of the system communicate cleanly. In driver development, they ensure the kernel can call into your code with complete confidence about the parameters and return type no guesswork, no surprises. Getting this right is a matter of both correctness and maintainability, and it's a habit you'll use in every driver you write.

Before we go further into writing complex driver logic, we need to understand one of the most fundamental concepts in C programming: variable scope. Scope determines where a variable can be accessed in your code, how long it stays alive in memory, and what parts of the program can modify it. In FreeBSD driver development, misunderstanding scope can lead to elusive bugs from uninitialised values corrupting hardware state to variables mysteriously changing between function calls. By mastering scope rules, you'll gain fine-grained control over your driver's data, ensuring that values are only visible where they should be, and that critical state is preserved or isolated as needed. In the next section, we'll break down scope into clear, practical categories and show you how to apply them effectively in kernel code.

4.8.12 Variable Scope in Functions

In programming, **scope** defines the boundaries within which a variable can be seen and used. In other words, it tells us where in the code a variable is visible and who is allowed to read or change its value.

When a variable is declared inside a function, we say it has **local scope**. Such a variable comes into existence when the function starts running and disappears as soon as the function finishes. No other function can see it, and even within the same function, it may be invisible if declared inside a more restricted block, such as inside a loop or an `if` statement.

This form of isolation is an important safeguard. It prevents accidental interference from other parts of the program, ensures that one function cannot inadvertently change the internal workings of another, and makes the program's behaviour more predictable. By keeping variables confined to the places they are needed, you make your code easier to reason about, maintain, and debug.

To make this idea more concrete, let's look at a short example in C. We'll create a function with a variable that lives entirely inside it. You'll see how the variable works perfectly within its own function, but becomes completely invisible the moment we step outside that function's boundaries.

```
#include <stdio.h>

void print_number(void) {
    int x = 42;          // x has local scope: only visible inside print_number()
    printf("%d\n", x);
}

int main(void) {
    print_number();
    // printf("%d\n", x); // ERROR: 'x' is not in scope here
    return 0;
}
```

Here, the variable `x` is declared inside `print_number()`, which means it is created when the function starts and destroyed when the function ends. If we try to use `x` in `main()`, the compiler complains because `main()` has no knowledge of `x`; it lives in a separate, private workspace. This “one workspace per function” rule is one of the foundations of reliable programming: it keeps code modular, avoids accidental changes from unrelated parts of the program, and helps you reason about the behaviour of each function independently.

Why Local Scope Is Good

Local scope brings three key benefits to your code:

- Prevents bugs: a variable inside one function cannot accidentally overwrite or be overwritten by another function's variable, even if they share the same name.
- Keeps code predictable: you always know exactly where a variable can be read or modified, making it easier to follow and reason about the program's flow.
- Improves efficiency: the compiler can often keep local variables in CPU registers, and any stack space they use is automatically freed when the function returns.

By keeping variables confined to the smallest area where they're needed, you reduce the chances of interference, make debugging easier, and help the compiler optimise performance.

Why scope matters in driver development

In FreeBSD device drivers, you'll often manipulate temporary values, such as buffer sizes, indices, error codes, and flags that are relevant only within a specific operation (e.g., probing a device, initialising a queue, handling an interrupt). Keeping these values local prevents cross-talk between concurrent paths and avoids subtle race conditions. In kernel space, small mistakes propagate fast; tight, local scope is your first line of defence.

From Simple Scope to Real Kernel Code

You’ve just seen how a local variable inside a small C program lives and dies within its function. Now, let’s step into a real FreeBSD driver and see exactly the same principle at work, but this time in code that interacts with actual hardware.

We’ll look at part of the VirtIO subsystem, which is used for virtual devices in environments like QEMU or bhyve. This example comes from the function `virtqueue_init_indirect()` in the file `sys/dev/virtio/virtqueue.c` (FreeBSD 14.3 source code), which sets up “indirect descriptors” for a virtual queue. Watch how variables are declared, used, and limited to the function’s own scope, just like in our earlier `print_number()` example.

Note: I’ve added some extra comments to highlight what’s happening at each step.

```
static int
virtqueue_init_indirect(struct virtqueue *vq, int indirect_size)
{
    // Local variable: holds the device reference for easy access
    // Only exists inside this function
    device_t dev;

    // Local variable: temporary pointer to a descriptor structure
    // Used during loop iterations to point to the right element
    struct vq_desc_extra *dvp;

    // Local variables: integer values used for temporary calculations
    // 'i' will be our loop counter, 'size' will hold the calculated memory size
    int i, size;

    // Initialise 'dev' with the device associated with this virtqueue
    // 'dev' is local, so it's only valid here - other functions cannot touch it
    dev = vq->vq_dev;

    // Check if the device supports the INDIRECT_DESC feature
    // This is done through a bus-level feature negotiation
    if (VIRTIO_BUS_WITH_FEATURE(dev, VIRTIO_RING_F_INDIRECT_DESC) == 0) {
        /*
         * If the driver requested indirect descriptors, but they were not
         * negotiated, we print a message (only if bootverbose is on).
         * Then we return 0 to indicate initialisation continues without them.
         */
        if (bootverbose)
            device_printf(dev, "virtqueue %d (%s) requested "
                "indirect descriptors but not negotiated\n",
                vq->vq_queue_index, vq->vq_name);
        return (0); // At this point, all locals are destroyed
    }
}
```

```

// Calculate the memory size needed for the indirect descriptors
size = indirect_size * sizeof(struct vring_desc);

// Store these values in the virtqueue structure for later use
vq->vq_max_indirect_size = indirect_size;
vq->vq_indirect_mem_size = size;

// Mark this virtqueue as using indirect descriptors
vq->vq_flags |= VIRTQUEUE_FLAG_INDIRECT;

// Loop through all entries in the virtqueue
for (i = 0; i < vq->vq_nentries; i++) {
    // Point 'dvp' to the i-th descriptor entry in the queue
    dvp = &vq->vq_descx[i];

    // Allocate memory for the indirect descriptor list
    dvp->indirect = malloc(size, M_DEVBUF, M_NOWAIT);

    // If allocation fails, log an error and stop initialisation
    if (dvp->indirect == NULL) {
        device_printf(dev, "cannot allocate indirect list\n");
        return (ENOMEM); // Locals are still destroyed upon return
    }

    // Get the physical address of the allocated memory
    dvp->indirect_paddr = vtophys(dvp->indirect);

    // Initialise the allocated descriptor list
    virtqueue_init_indirect_list(vq, dvp->indirect);
}

// Successfully initialised indirect descriptors - locals end their life here
return (0);
}

```

Understanding the Scope in This Code

Even though this is production-level kernel code, the principle is the same as in the tiny example we just saw. The variables `dev`, `dvp`, `i`, and `size` are all declared inside `virtqueue_init_indirect()` and exist only while this function is running. Once the function returns, whether it's at the end or early via a return statement, those variables vanish, freeing their stack space for other uses.

Notice that this keeps things safe: the loop counter `i` can't be accidentally reused in another part of the driver, and the `dvp` pointer is re-initialised for each call to the function. In driver development, this is a critical local

scope that ensures that temporary work variables won't collide with names or data in other parts of the kernel. The isolation you learned about in the simple `print_number()` example applies here in exactly the same way, just at a higher level of complexity and with real hardware resources involved.

Common Beginner Mistakes (and How to Avoid Them)

One of the quickest ways to get into trouble is to store the address of a local variable in a structure that outlives the function. Once the function returns, that memory is reclaimed and can be overwritten at any time, leading to mysterious crashes. Another issue is “over-sharing”, using too many global variables for convenience, which can cause unpredictable results if multiple execution paths modify them at the same time. And finally, be careful not to shadow variables (reusing a name inside an inner block), which can lead to confusion and hard-to-spot bugs.

Wrapping Up and Moving Forward

The lesson here is simple: local scope makes your code safer, easier to test, and more maintainable. In FreeBSD device drivers, it is the right tool for per-call, temporary data. Long-lived information should be stored in properly designed per-device structures, keeping your driver organised and avoiding accidental data sharing.

Now that you understand **where** a variable can be used, it is time to look at **how long** it exists. This is called **variable storage duration**, and it affects whether your data lives on the stack, in static storage, or on the heap. Knowing the difference is key to writing robust, efficient drivers, and that's precisely where we are headed next.

4.8.13 Variable Storage Duration

So far, you've learned where a variable can be used in your program, as well as its scope. But there's another equally important property: how long the variable actually exists in memory. This is called its storage duration.

While scope is about visibility in the code, storage duration is about lifetime in memory. A variable's storage duration determines:

- **When** the variable is created.
- **When** it is destroyed.
- **Where** it lives (stack, static storage, heap).

Understanding storage duration is critical in FreeBSD driver development because we often handle resources that must persist across function calls (like device state) alongside temporary values that must vanish quickly (like loop counters or temporary buffers).

4.8.14 The Three Main Storage Durations in C

When you create a variable in C, you're not just giving it a name and a value, you're also deciding **how long that value will live in memory**. This “lifetime” is what we call the **storage** duration. Even two variables that look similar in the code can behave very differently depending on how long they stick around.

Let's break down the three main types you'll encounter, starting with the most common in day-to-day programming.

Automatic Storage Duration (stack variables)

Think of these as short-term helpers. They are born the moment a function starts running and disappear the instant the function finishes. You don't have to create or destroy them manually; C takes care of that for you.

Automatic variables:

- Are declared inside functions without the `static` keyword.
- Are created when the function is called and destroyed when it returns.
- Live on the **stack**, a section of memory that's automatically managed by the program.
- Are perfect for quick, temporary jobs like loop counters, temporary pointers, or small scratch buffers.

Because they vanish when the function ends, you can't keep their address for later use; doing so leads to one of the most common beginner mistakes in C.

Small Example:

```
#include <stdio.h>

void greet_user(void) {
    char name[] = "FreeBSD"; // automatic storage, stack memory
    printf("Hello, %s!\n", name);
} // 'name' is destroyed here

int main(void) {
    greet_user();
    return 0;
}
```

Here, `name` lives only while `greet_user()` runs. When the function exits, the stack space is freed automatically.

Static Storage Duration (globals and static variables)

Now imagine a variable that doesn't come and go with a function call, instead, it's **always there** from the moment your program (or in kernel space, your driver module) loads until it ends. This is **static storage**.

Static variables:

- Are declared outside functions or inside functions with the `static` keyword.
- Are created **once** when the program/module starts.
- Remain in memory until the program/module ends.
- Live in a dedicated **static memory** area.
- Are great for things like per-device state structures or lookup tables that are needed throughout the program's lifetime.

However, since they stick around, you must be careful in driver code shared, long-lived data can be accessed by multiple execution paths, so you may need locks or other synchronization to avoid conflicts.

Small Example:

```
#include <stdio.h>

static int counter = 0; // static storage, exists for the entire program

void increment(void) {
    counter++;
    printf("Counter = %d\n", counter);
}

int main(void) {
    increment();
    increment();
    return 0;
}
```

counter keeps its value between calls to `increment()` because it never leaves memory until the program ends.

Dynamic Storage Duration (heap allocation)

Sometimes you don't know in advance how much memory you'll need, or you need to keep something around even after the function that created it has finished. That's where dynamic storage comes in: you request memory at runtime, and you decide when it goes away.

Dynamic variables:

- Are created explicitly at runtime with `malloc()/free()` in user space, or `malloc(9)/free(9)` in the FreeBSD kernel.
- Exist until you explicitly free them.
- Live in the **heap**, a pool of memory managed by the operating system or kernel.
- Are perfect for things like buffers whose size depends on hardware parameters or user input.

The flexibility comes with responsibility: forget to free them, and you'll have a memory leak. Free them too soon, and you might crash the system by accessing invalid memory.

Small Example:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void) {
    char *msg = malloc(32); // dynamic storage
    if (!msg) return 1;
    strcpy(msg, "Hello from dynamic memory!");
    printf("%s\n", msg);
    free(msg); // must free to avoid leaks
}
```

```
    return 0;
}
```

Here, the program decides at runtime to allocate 32 bytes. The memory is under your control, so you must free it when done.

4.8.15 Bridging Theory and Practice

So far, we’ve looked at these storage durations in an abstract way. But concepts really sink in when you see them in the wild, inside a real FreeBSD driver or subsystem function. Kernel code often mixes these durations: a few automatic locals for temporary values, some static structures for persistent state, and carefully managed dynamic memory for resources that come and go during runtime.

To make this clearer, let’s walk through a real function from the FreeBSD 14.3 source tree. By following each variable and seeing how it’s declared, used, and eventually discarded or freed, you’ll gain an intuitive feel for how lifetime and scope interact in real-world kernel work.

Duration	Created	Destroyed	Memory area	Typical declarations	Good driver use cases	Common pitfalls	FreeBSD APIs to know
Automatic	On function entry	On function return	Stack	Local variables without static	Scratch values in fast paths and interrupt handlers	Returning addresses of locals. Oversized locals	N/A
Static	When module loads	When module unloads	Static storage	File scope variables or static inside funcs	Persistent device state. Constant tables. Tunables	Hidden shared state. Missing locks on SMP	sysctl(9) patterns for tunables
Dynamic	When you call allocator	When you free it	Heap	Pointers returned by allocators	Buffers sized at probe time. Lifetime spans calls	Leaks. Use after free. Double free	malloc(9), free(9), M_* types

4.8.16 Real Example from FreeBSD 14.3

Before we move on, let’s look at how these storage duration concepts appear in production-quality FreeBSD code. Our example comes from the network interface subsystem, specifically from the `_if_delgroup_locked()` function in `sys/net/if.c` (FreeBSD 14.3). This function removes an interface from a named interface group, updates reference counts, and frees memory when the group becomes empty.

As in our earlier, simpler examples, you’ll see **automatic** variables created and destroyed entirely within the function, **dynamic** memory being released explicitly with `free(9)`, and, elsewhere in the same file, **static**

variables that persist for the module's entire lifetime. By walking through this function, you'll see lifetime and scope management in action not just in an isolated snippet, but in the complex, interconnected world of the FreeBSD kernel.

Note: I've added some extra comments to highlight what's happening at each step.

```

/*
 * Helper function to remove a group out of an interface. Expects the global
 * ifnet lock to be write-locked, and drops it before returning.
 */
static void
_if_delgroup_locked(struct ifnet *ifp, struct ifg_list *ifgl,
    const char *groupname)
{
    struct ifg_member *ifgm;    // [Automatic] (stack) pointer: used only in this call
    bool freeifgl;              // [Automatic] (stack) flag: should we free the group?

    IFNET_WLOCK_ASSERT();       // sanity: we entered with the write lock held

    /* Remove the (interface,group) link from the interface's local list. */
    IF_ADDR_WLOCK(ifp);
    CK_STAILQ_REMOVE(&ifp->if_groups, ifgl, ifg_list, ifgl_next);
    IF_ADDR_WUNLOCK(ifp);

    /*
     * Find and remove this interface from the group's member list.
     * 'ifgm' is a LOCAL cursor; it does not escape this function
     * (classic automatic storage).
     */
    CK_STAILQ_FOREACH(ifgm, &ifgl->ifgl_group->ifg_members, ifgm_next) {
        /* [Automatic] 'ifgm' is a local iterator only */
        if (ifgm->ifgm_ifp == ifp) {
            CK_STAILQ_REMOVE(&ifgl->ifgl_group->ifg_members, ifgm,
                ifg_member, ifgm_next);
            break;
        }
    }

    /*
     * Decrement the group's reference count. If we just removed the
     * last member, mark the group for freeing after we drop locks.
     */
    if (--ifgl->ifgl_group->ifg_refcnt == 0) {
        CK_STAILQ_REMOVE(&V_ifg_head, ifgl->ifgl_group, ifg_group,
            ifg_next);
    }
}

```

```

        freeifgl = true;
    } else {
        freeifgl = false;
    }
    IFNET_WUNLOCK();           // we promised to drop the global lock before return

    /*
     * Wait for readers in the current epoch to finish before freeing memory
     * (RCU-style safety in the networking stack).
     */
    NET_EPOCH_WAIT();

    /* Notify listeners that the group membership changed. */
    EVENTHANDLER_INVOKE(group_change_event, groupname);

    if (freeifgl) {
        /* Group became empty: fire detach event and free the group object. */
        EVENTHANDLER_INVOKE(group_detach_event, ifgl->ifgl_group);
        free(ifgl->ifgl_group, M_TEMP); // [Dynamic] (heap) storage being returned
    }

    /* Free the (interface,group) membership nodes allocated earlier. */
    free(ifgm, M_TEMP); // [Dynamic] the 'member' record
    free(ifgl, M_TEMP); // [Dynamic] the (ifnet, group) link record
}

```

What to notice

- [Automatic] ifgm and freeifgl live only for this call. They cannot outlive the function.
- [Dynamic] frees return heap objects that were allocated earlier in the driver life cycle. The lifetime crosses function boundaries and must be released on the exact success path shown here.
- [Static] is not used in this function. In the same file you will find persistent configuration and counters that exist from load to unload. Those are [Static].

Understanding the Storage Durations in This Function

If you follow `_if_delgroup_locked()` from start to finish, you can watch all three storage durations in C play their part. The variables `ifgm` and `freeifgl` are automatic, which means they are born when the function is called, live entirely on the stack, and disappear the moment the function returns. They are private to this call, so nothing outside can accidentally change them, and they cannot change anything outside either.

A little further down, the calls to `free(...)` deal with dynamic storage. The pointers passed to `free()` were created earlier in the driver's life, often with `malloc()` during initialisation routines like `if_addgroup()`. Unlike stack variables, this memory stays around until the driver deliberately lets it go. Freeing it here tells the kernel, "I'm done with this; you can reuse it for something else."

This function doesn't use static variables directly, but in the same file (`if.c`), you will find examples like

debugging flags declared with `YSCTL_INT` that live for as long as the kernel module is loaded. These variables keep their values across function calls and are a reliable place to store configuration or diagnostics that need to persist.

Each choice here is intentional.

- Automatic variables keep temporary state safe inside the function.
- Dynamic memory gives flexibility at runtime, allowing the driver to adjust and then clean up when done.
- Static storage, found elsewhere in the same codebase, supports persistent, shared information.

Put together, this is a clear, real-world example of how lifetime and visibility work hand in hand in FreeBSD driver code. It is not just theory from a C textbook, it is the day-to-day reality of writing drivers that are reliable, efficient, and safe to run in the kernel.

4.8.17 Why Storage Duration Matters in FreeBSD Drivers

In kernel development, storage duration is not just an academic detail; it's directly tied to system stability, performance, and even security. A wrong choice here can take down the entire operating system.

In FreeBSD drivers, the right storage duration ensures that data lives exactly as long as needed, no more and no less:

- **Automatic variables** are ideal for short-lived, private state, such as temporary values in an interrupt handler. They vanish automatically when the function ends, avoiding long-term clutter in memory.
- **Static variables** can safely store hardware state or configuration that must persist across calls, but they introduce shared state that may require locking in SMP systems to avoid race conditions.
- **Dynamic allocations** give you flexibility when buffer sizes depend on runtime conditions like device probing results, but they must be explicitly freed to avoid leaks and freeing too soon risks accessing invalid memory.

Mistakes with storage duration can be catastrophic in the kernel. Keeping a pointer to a stack variable beyond the function's life is almost guaranteed to cause corruption. Forgetting to free dynamic memory ties up resources until a reboot. Overusing static variables can turn shared state into a performance bottleneck.

Understanding these trade-offs is not optional. In driver code, often triggered by hardware events in unpredictable contexts, correct lifetime management is a foundation for writing code that is safe, efficient, and maintainable.

4.8.18 Common Beginner Mistakes

When you are new to C and especially to kernel programming, it is surprisingly easy to misuse storage duration without even realising it. One classic trap with automatic variables is returning the address of a local variable from a function. At first, it might seem harmless after all, the variable was right there a moment ago, but the moment the function returns, that memory is reclaimed for other uses. Accessing it later is like reading a letter you already burned; the result is undefined behaviour, and in the kernel, that can mean an instant crash.

Static variables can cause trouble differently. Because they persist across function calls, a value left over from a previous run of the function might influence the next run in unexpected ways. This is particularly dangerous

if you assume that every call starts with a “clean slate.” In reality, static variables remember everything, even when you wish they wouldn’t.

Dynamic memory has its own set of hazards. Forgetting to `free()` something you allocated means the memory will be tied up until the system is restarted, a problem known as a memory leak. In kernel space, where resources are precious, a leak can slowly degrade the system. Freeing the same pointer twice is even worse, it can corrupt kernel memory structures and bring down the whole machine.

Being aware of these patterns early on helps you avoid them when working on real driver code, where the cost of a mistake is often far greater than in user-space programming.

4.8.19 Wrapping Up

We have explored the three main storage durations in C: automatic, static, and dynamic. Each one has its place, and the right choice depends on how long you need the data to live and who should be able to see it. The safest general rule is to choose the smallest necessary lifetime for your variables. This limits their exposure, reduces the risk of unintended interactions, and often makes the compiler’s job easier.

In FreeBSD driver development, careful management of variable lifetimes is not optional; it is a fundamental skill. Done right, it helps you write code that is predictable, efficient, and resilient under load. With these principles in mind, you are ready to explore the next piece of the puzzle: understanding how variable linkage affects visibility across files and modules.

4.8.20 Variable Linkage (Visibility Across Files)

So far, we’ve explored **scope** (where a name is visible inside your code) and **storage duration** (how long an object exists in memory). The third and final piece in this visibility puzzle is **linkage**, the rule that decides whether code in other source files can refer to a given name.

In C (and in FreeBSD kernel code), programs are often split into multiple `.c` files plus the header files they include. Each `.c` file and its headers form a translation unit. By default, most names you define are visible only inside the translation unit where they’re declared. If you want other files to see them or, **often more importantly**, to hide them, linkage is the mechanism that controls that access.

4.8.21 The three kinds of linkage in C

Think of linkage as “*who outside this file can see this name?*”:

- **External linkage:** A name is visible across translation units. Global variables and functions defined at file scope without `static` have external linkage. Other files can refer to them by declaring `extern` (for variables) or including a prototype (for functions).
- **Internal linkage:** A name is visible only within the current file. You get internal linkage by writing `static` at file scope (for variables or functions). This is how you keep helpers and the private state hidden from the rest of the kernel/program.
- **No linkage:** A name is visible only within its own block (e.g., variables inside a function). These are locals; they can’t be named from outside their scope at all.

4.8.22 A tiny two-file illustration

To really see linkage in action, let's build the smallest possible program that spans two .c files. This will let us test all three cases, external, internal, and no linkage, side by side. We'll create one file (foo.c) that defines a few variables and a helper function, and another file (main.c) that tries to use them.

Below, shared_counter has **external linkage** (visible in both files), internal_flag has **internal linkage** (visible only inside foo.c), and the locals inside increment() have **no linkage** (visible only in that function).

foo.c

```
#include <stdio.h>

/*
 * Global variable with external linkage:
 * - Visible to other files (translation units) in the program.
 * - No 'static' keyword means it has external linkage by default.
 */
int shared_counter = 0;

/*
 * File-private variable with internal linkage:
 * - The 'static' keyword at file scope means this name
 *   is only visible inside foo.c.
 */
static int internal_flag = 1;

/*
 * Function with external linkage by default:
 * - Can be called from other files if they declare its prototype.
 */
void increment(void) {
    /*
     * Local variable with no linkage:
     * - Exists only during this function call.
     * - Cannot be accessed from anywhere else.
     */
    int step = 1;

    if (internal_flag)          // Only code in foo.c can see internal_flag
        shared_counter += step; // Modifies the global shared_counter

    printf("Counter: %d\n", shared_counter);
}
```

main.c

```

#include <stdio.h>

/*
 * 'extern' tells the compiler:
 * - This variable exists in another file (foo.c).
 * - Do not allocate storage for it here.
 */
extern int shared_counter;

/*
 * Forward declaration for the function defined in foo.c:
 * - Lets us call increment() from this file.
 */
void increment(void);

int main(void) {
    increment();           // Calls increment() from foo.c
    shared_counter += 10;  // Legal: shared_counter has external linkage
    increment();

    // internal_flag = 0;  // ERROR: not visible here (internal linkage in foo.c)
    return 0;
}

```

The pattern generalizes directly to kernel code: keep helpers and private state static in one .c file, expose only the minimal surface via headers (prototypes) or intentionally exported globals.

4.8.23 Real FreeBSD 14.3 Example: External vs. Internal vs. No Linkage

Let's ground this in the FreeBSD network stack (sys/net/if.c). We'll look at:

1. a **global** variable with **external** linkage (ifqmaxlen),
2. **file-private** toggles with **internal linkage** (log_link_state_change, log_promisc_mode_change), and
3. a **function** with a **local variable** (no linkage) (sysctl_ifcount()), plus how it's exposed via SYSCTL_PROC.

1) External linkage: a tunable global

In sys/net/if.c, ifqmaxlen is a global integer that other parts of the kernel can reference. That's **external linkage**.

```
int ifqmaxlen = IFQ_MAXLEN; // external linkage: visible to other files
```

You'll also see it referenced from the SYSCTL tree setup:

```

SYSCTL_INT(_net_link, OID_AUTO, ifqmaxlen, CTLFLAG_RDTUN,
    &ifqmaxlen, 0, "max send queue size");

```

This exposes the global through `sysctl`, so administrators can read/tune it at boot (depending on flags).

2) Internal linkage: file-private toggles

Right above, the file defines two static integers. Because they're **static** at file scope, they have **internal** linkage only `if.c` can name them:

```

/* Log link state change events */
static int log_link_state_change = 1;

SYSCTL_INT(_net_link, OID_AUTO, log_link_state_change, CTLFLAG_RW,
    &log_link_state_change, 0,
    "log interface link state change events");

/* Log promiscuous mode change events */
static int log_promisc_mode_change = 1;

SYSCTL_INT(_net_link, OID_AUTO, log_promisc_mode_change, CTLFLAG_RDTUN,
    &log_promisc_mode_change, 1,
    "log promiscuous mode change events");

```

Later in the same file, `log_link_state_change` is used to decide whether to print a message, but only code inside `if.c` can refer to that symbol by name:

```

if (log_link_state_change)
    if_printf(ifp, "link state changed to %s\n",
        (link_state == LINK_STATE_UP) ? "UP" : "DOWN");

```

See `sys/net/if.c` for the static definitions and the reference in `do_link_state_change()`.

3) No linkage (locals) + how a private function is exported via SYSCTL

Here's the full `sysctl_ifcount()` function (as in FreeBSD 14.3), with line-by-line commentary. Notice how `rv` is a local; it has no linkage and exists only for the duration of this call.

Note: I've added some extra comments to highlight what's happening at each step.

```

/* sys/net/if.c */

/*
 * 'static' at file scope:
 * - Gives the function internal linkage (only visible in if.c).
 * - Other files cannot call sysctl_ifcount() directly.

```

```

*/
static int
sysctl_ifcount(SYSCTL_HANDLER_ARGS) // SYSCTL handler signature used in the kernel
{
    /*
     * Local variable with no linkage:
     * - Exists only during this function call.
     * - Tracks the highest interface index in the current vnet.
     */
    int rv = 0;

    /*
     * IFNET_RLOCK():
     * - Acquires a read lock on the ifnet index table.
     * - Ensures safe concurrent access in an SMP kernel.
     */
    IFNET_RLOCK();

    /*
     * Loop through interface indices from 1 up to the current max (if_index).
     * If an entry is in use and belongs to the current vnet,
     * update rv with the highest index seen.
     */
    for (int i = 1; i <= if_index; i++)
        if (ifindex_table[i].ife_ifnet != NULL &&
            ifindex_table[i].ife_ifnet->if_vnet == curvnet)
            rv = i;

    /*
     * Release the read lock on the ifnet index table.
     */
    IFNET_RUNLOCK();

    /*
     * Return rv to user space via the sysctl framework.
     * - sysctl_handle_int() handles copying the value to the request buffer.
     */
    return (sysctl_handle_int(oidp, &rv, 0, req));
}

```

The function is then **registered** with the sysctl tree so other kernel parts (and user space via sysctl) can invoke it without needing external linkage to the function name:

```

/*
 * SYSCTL_PROC:
 * - Creates a sysctl entry named 'ifcount' under:
 *   net.link.generic.system
 * - Flags: integer type, vnet-aware, read-only.
 * - Calls sysctl_ifcount() when queried.
 * - Even though sysctl_ifcount() is static, the sysctl framework
 *   acts as the public interface to its result.
 */
SYSCTL_PROC(_net_link_generic_system, IFMIB_IFCOUNT, ifcount,
    CTLTYPE_INT | CTLFLAG_VNET | CTLFLAG_RD, NULL, 0,
    sysctl_ifcount, "I", "Maximum known interface index");

```

This pattern is common in the kernel: the function itself has **internal linkage** (`static`), but it's exposed through a registration mechanism (`sysctl`, `eventhandler`, `devfs` methods, etc.).

4.8.24 Why this matters for drivers

- **Encapsulation with internal linkage:** Use `static` at file scope to keep helpers and private state inside a single `.c` file. This reduces accidental coupling and eliminates a whole class of “who changed this?” bugs under SMP.
- **Safe temporaries with no linkage:** Prefer locals for per-call data so nothing outside the function can modify it. This helps ensure correctness and makes concurrency easier to reason about.
- **Intentional exposure through interfaces:** When you need to share information, expose it through a registration mechanism such as `SYSCTL_PROC`, an `eventhandler`, or `devfs` methods, rather than exporting function names directly.

In `sys/net/if.c`, you can see all three visibility levels in action:

- **External linkage:** `ifqmaxlen` is a global variable accessible to other files.
- **Internal linkage:** `log_link_state_change` and `log_promisc_mode_change` are file-private toggles.
- **No linkage:** Local variable `rv` inside `sysctl_ifcount()`, exposed intentionally via `SYSCTL_PROC`.

4.8.25 Common beginner pitfalls (and how to sidestep them)

A few patterns trip people up when they first juggle scope, storage duration, and linkage:

- **Using file-private helpers from another file.** If you see “undefined reference” at link time for a helper you thought was global, check for a `static` on its definition. If it's truly meant to be shared, move the prototype to a header and remove `static` from the definition. If not, keep it private and call it indirectly via a registered interface (like `sysctl` or an ops table).
- **Accidentally exporting private state.** A bare `int myflag;` at file scope has external linkage. If you intended it to be file-local, write `static int myflag;`. This one keyword prevents cross-file name collisions and unintended writes.

- **Leaning on globals instead of passing arguments.** If two unrelated call paths tweak the same global, you’ve invited heisenbugs. Prefer locals and function parameters, or encapsulate shared state in a per-device struct referenced through `softc`.
- **Beginners often confuse `static` in file scope (linkage control) with `static` inside a function (storage duration control).** In file scope, `static` hides a symbol from other files (linkage control). Inside a function, `static` makes a variable keep its value between calls (storage duration control).

4.8.26 Wrapping up

You now understand **scope**, **storage duration**, and **linkage**, the three pillars that define where a variable can be used, how long it exists, and who can access it. These concepts form the foundation for managing state in any C program, and they are especially critical in FreeBSD drivers, where per-call locals, file-private helpers, and global kernel state must coexist without interfering with one another.

Next, we’ll see what happens when you pass those variables into a function. In C, function parameters are copies of the original values, so changes inside the function won’t affect the originals unless you pass their addresses. Understanding this behaviour is key to writing driver code that updates state intentionally, avoids subtle bugs, and communicates data effectively between functions.

4.9 Parameters Are Copies

When you call a function in C, the values you pass to it are **copied** into the function’s parameters. The function then works with those copies, not the originals. This is known as **call by value**, and it means that any changes made to the parameter inside the function are lost when the function returns; the caller’s variables remain untouched.

This is different from some other programming languages that use “pass by reference” by default, where a function can directly modify a caller’s variable without special syntax. In C, if you want a function to modify something outside its own scope, you must give it the **address** of that thing. That’s done using **pointers**, which we’ll explore in depth in the next section.

Understanding this behaviour is critical in FreeBSD driver development. Many driver functions perform setup work, check for conditions, or calculate values without touching the caller’s variables unless they are explicitly passed a pointer. This design helps maintain isolation between different parts of the kernel, reducing the risk of unintended side effects.

4.9.1 A Simple Example: Modifying a Copy

To see this in action, we’ll write a short program that passes an integer to a function. Inside the function, we’ll try to change it. If C worked the way many beginners expect, this would update the original value. But because parameters in C are **copies**, the change will only affect the function’s local version, leaving the original untouched.

```
#include <stdio.h>
```



```

void modify(int x) {
    x = 42; // Only updates the function's own copy
}

int main(void) {
    int original = 5;
    modify(original);
    printf("%d\n", original); // Still prints 5, not 42!
    return 0;
}

```

Here, `modify()` changes its local version of `x`, but the original variable in `main()` stays at 5. The copy disappears as soon as `modify()` returns, leaving `main()`'s data untouched.

If you do want to change the original variable inside a function, you must pass a reference to it rather than a copy. In C, that reference takes the form of a pointer, which lets the function work directly with the original data in memory. Don't worry if pointers sound mysterious, we'll cover them thoroughly in the next section.

4.9.2 A Real Example from FreeBSD 14.3

This concept shows up in production kernel code all the time. Let's see a real function from 'sys/net/if.c' in FreeBSD 14.3 that removes an interface from a group: `_if_delgroup_locked()`. Pay special attention to the **parameters** at the top: `ifp`, `ifgl`, and `groupname`. Each is a **copy** of the value that the caller passed in. They're local to this function call, even though they **refer to** shared kernel objects.

In the listing below, I've added extra comments so you can see exactly what's happening at each step.

Notice how these parameters are local copies, even though they hold pointers to shared kernel data.

```

/*
 * Helper function to remove a group out of an interface. Expects the global
 * ifnet lock to be write-locked, and drops it before returning.
 */
static void
_if_delgroup_locked(struct ifnet *ifp, struct ifg_list *ifgl,
    const char *groupname)
{
    struct ifg_member *ifgm; // local (automatic) variable: lives only during
    ↪ this call
    bool freeifgl;           // local flag on the stack: also per-call

    /*
     * PARAMETERS ARE COPIES:
     * - 'ifp' is a copy of a pointer to the interface object.
     * - 'ifgl' is a copy of a pointer to a (interface,group) link record.

```

```

    * - 'groupname' is a copy of a pointer to constant text.
    * The pointer VALUES are copied, but they still refer to the same kernel data
    * as the caller's originals. Reassigning 'ifp' or 'ifgl' here wouldn't affect
    * the caller; modifying the *pointed-to* structures does persist.
    */

    IFNET_WLOCK_ASSERT(); // sanity: we entered with the global ifnet write lock
↪ held

    // Remove the (ifnet,group) link from the interface's list.
    IF_ADDR_WLOCK(ifp);
    CK_STAILQ_REMOVE(&ifp->if_groups, ifgl, ifg_list, ifgl_next);
    IF_ADDR_WUNLOCK(ifp);

    // Walk the group's member list and remove this interface from it.
    CK_STAILQ_FOREACH(ifgm, &ifgl->ifgl_group->ifg_members, ifgm_next) {
        if (ifgm->ifgm_ifp == ifp) {
            CK_STAILQ_REMOVE(&ifgl->ifgl_group->ifg_members, ifgm,
                            ifg_member, ifgm_next);
            break;
        }
    }

    // Decrement the group's reference count; if it hits zero, mark for free.
    if (--ifgl->ifgl_group->ifg_refcnt == 0) {
        CK_STAILQ_REMOVE(&V_ifg_head, ifgl->ifgl_group, ifg_group,
                        ifg_next);
        freeifgl = true;
    } else {
        freeifgl = false;
    }

    IFNET_WUNLOCK(); // drop the global ifnet lock before potentially freeing
↪ memory

    // Wait for current readers to exit the epoch section before freeing (RCU-style
    ↪ safety).
    NET_EPOCH_WAIT();

    // Notify listeners that a group membership changed (uses the 'groupname'
    ↪ pointer).
    EVENTHANDLER_INVOKE(group_change_event, groupname);

    if (freeifgl) {
        // If the group is now empty: announce detach and free the group object.

```

```
        EVENTHANDLER_INVOKE(group_detach_event, ifgl->ifgl_group);
        free(ifgl->ifgl_group, M_TEMP);
    }

    // Free the membership record and the (ifnet,group) link record.
    free(ifgm, M_TEMP);
    free(ifgl, M_TEMP);
}
```

In this kernel example, the parameters behave like they’re passed “by reference” because they hold addresses to kernel objects. However, the pointer values themselves are still copies.

What This Shows

Here, `ifp`, `ifgl`, and `groupname` are copies of what the caller passed. If we reassigned `ifp = NULL`; inside this function, the caller’s `ifp` would be unaffected. But because the pointer values still point to real kernel structures, changes to those structures, like removing from lists or freeing memory, are seen system-wide.

Meanwhile, `ifgm` and `freeifgl` are purely local automatic variables. They live only while this function runs and vanish immediately after it returns.

This mirrors our tiny user-space example exactly; the only difference is that here, the parameters are pointers into complex, shared kernel data.

4.9.3 Why This Matters in FreeBSD Driver Development

In driver code, understanding that parameters are copies helps you avoid dangerous assumptions:

- If you change the parameter variable itself (like reassigning a pointer), the caller won’t see that change.
- If you change the object the pointer refers to, the caller and possibly the rest of the kernel will see the change, so you must be sure it’s safe.
- Passing large structures by value creates full copies on the stack; passing pointers shares the same data.

This distinction is essential for writing predictable, race-free kernel code.

4.9.4 Common Beginner Mistakes

When working with parameters in C, especially in FreeBSD kernel code, beginners often get caught in subtle traps that stem from not fully grasping the “copy” rule.

Let’s look at some of the most common:

1. **Passing a structure by value instead of a pointer:** You expect changes to update the original, but they only update your local copy. Example: passing a struct `ifreq` by value and wondering why the interface isn’t reconfigured.
2. **Forgetting that a pointer grants write access:** Passing `struct mydev *` gives the callee full ability to change the device state. Without proper locking, this can corrupt kernel data.

3. **Confusing a pointer copy with copying data:** Reassigning the pointer parameter (`ptr = NULL;`) doesn't affect the caller's pointer. Modifying the pointed-to object (`ptr->field = 42;`) does affect the caller.
4. **Copying large structures by value in kernel space** This wastes CPU time and risks overflowing the limited kernel stack.
5. **Failing to document modification intent:** If your function will modify its input, make it evident in the function name, comments, and parameter type.

Rule of Thumb: Pass by value to keep data safe. Pass a pointer only when you intend to modify the data and make that intent explicit.

4.9.5 Wrapping Up

You've now seen that parameters in C work by **value**: every function receives its own private copy of what you pass, even if that value is an address pointing to shared data. This model gives you both safety and responsibility: safety, because variables themselves are isolated between caller and callee; responsibility, because the data being pointed to may still be shared and mutable.

Next, we'll shift focus from individual variables to collections of data that C programmers (and FreeBSD drivers) use constantly: **arrays and strings**.

4.10 Arrays and Strings in C

In the previous section, you learned that function parameters are passed by value. That lesson sets the stage for working with **arrays and strings**, two of the most common structures in C. Arrays give you a way to handle collections of elements in contiguous memory. In contrast, strings are simply arrays of characters with a special terminator.

Both are central to FreeBSD driver development: arrays become buffers that move data in and out of hardware, and strings carry device names, configuration options, and environment variables.

We build from the basics, highlight common pitfalls, and then connect the concepts to real FreeBSD kernel code, concluding with hands-on labs.

4.10.1 Declaring and Using Arrays

An array in C is a fixed-size collection of elements, all of the same type, stored in contiguous memory. Once defined, its size cannot change.

```
int numbers[5];           // Declares an array of 5 integers
```

You can initialize an array at the time of declaration:

```
int primes[3] = {2, 3, 5}; // Initialize with values
```

Each element is accessed by its index, starting at zero:

```
primes[0] = 7;           // Change the first element
int second = primes[1]; // Read the second element (3)
```

In memory, arrays are laid out sequentially. If numbers starts at address 1000 and each integer takes 4 bytes, then numbers[0] is at 1000, numbers[1] at 1004, numbers[2] at 1008, and so on. This detail becomes very important when we study pointers.

4.10.2 Strings in C

Unlike some languages where strings are a distinct type, in C, a string is simply an array of characters terminated with a special `'\0'` character known as the **null terminator**.

```
char name[6] = {'E', 'd', 's', 'o', 'n', '\0'};
```

A more convenient form lets the compiler insert the null terminator for you:

```
char name[] = "Edson"; // Stored as E d s o n \0
```

Strings can be accessed and modified character by character:

```
name[0] = 'A'; // Now the string reads "Adson"
```

If the terminating `'\0'` is missing, functions that expect a string will continue reading memory until they hit a zero byte somewhere else. This often results in garbage output, memory corruption, or kernel crashes.

4.10.3 Common String Functions (<string.h>)

The C standard library provides helper functions for strings. Although you cannot use the full standard library within the FreeBSD kernel, many equivalents are available. It is important to know the standard ones first:

```
#include <string.h>

char src[] = "FreeBSD";
char dest[20];

strcpy(dest, src);           // Copy src into dest
int len = strlen(dest);      // Get string length
int cmp = strcmp(src, dest); // Compare two strings
```

Frequently used functions include:

- `strcpy()` - copy one string into another (unsafe, no bounds checking).

- `strncpy()` - safer variant, lets you specify maximum characters.
- `strlen()` - count characters before the null terminator.
- `strcmp()` - compare two strings lexicographically.

Warning: many standard functions like `strcpy()` are unsafe because they do not check buffer sizes. In kernel development, this can corrupt memory and cause the system to crash. Safer variants such as `strncpy()` or kernel-provided helpers should always be preferred.

4.10.4 Why This Matters in FreeBSD Drivers

Arrays and strings are not just a basic C feature; they're at the heart of how FreeBSD drivers manage data. Nearly every driver you write or study relies on them in one form or another:

- **Buffers** that temporarily hold data moving between hardware and the kernel, such as keystrokes, network packets, or bytes written to disk.
- **Device names** like `/dev/ttyu0` or `/dev/random` are presented to user space by the kernel.
- **Configuration tunables (sysctl)** that depend on arrays and strings to store parameter names and values.
- **Lookup tables** are fixed-size arrays that hold supported hardware IDs, feature flags, or hardware-to-human-readable name mappings.

Because arrays and strings interact closely with hardware interfaces, mistakes here have consequences far beyond a user-space crash. While a runaway write in user-space might only crash that process, the same bug in kernel-space **can overwrite critical memory**, cause a kernel panic, corrupt data, or even open a security hole.

A real-world example makes this point clear. In **CVE-2024-45288**, the FreeBSD `libnv` library (used in both the kernel and userland) mishandled arrays of strings: it assumed strings were null-terminated without verifying their termination. A maliciously crafted `nvlist` could cause memory beyond the allocated buffer to be read or written, leading to kernel panic or even privilege escalation. The fix required explicit checks, safer memory allocation, and overflow protection.

Here's a simplified before/after look at the bug and its correction:

```
/*
 * CVE-2024-45288 Analysis: Missing Null-Termination in libnv String Arrays
 *
 * VULNERABILITY: A missing null-termination character in the last element
 * of an nvlist array string can lead to writing outside the allocated buffer.
 */

// BEFORE (Vulnerable Code):
static char **
nvpair_unpack_string_array(bool isbe __unused, nvpair_t *nvp,
    const char *data, size_t *leftp)
{
    char **value, *tmp, **valuep;
    size_t ii, size, len;
```

```

tmp = (char *) (uintptr_t) data;
size = nvp->nvp_datasize;

for (ii = 0; ii < nvp->nvp_nitems; ii++) {
    len = strlen(tmp, size - 1) + 1;
    size -= len;
    // BUG: No check if tmp[len-1] is actually '\0'!
    tmp += len;
}

// BUG: nv_malloc does not zero-initialize
value = nv_malloc(sizeof(*value) * nvp->nvp_nitems);
if (value == NULL)
    return (NULL);
// ...
}

// AFTER (Fixed Code):
static char **
nvpair_unpack_string_array(bool isbe __unused, nvpair_t *nvp,
    const char *data, size_t *leftp)
{
    char **value, *tmp, **valuep;
    size_t ii, size, len;

    tmp = (char *) (uintptr_t) data;
    size = nvp->nvp_datasize;

    for (ii = 0; ii < nvp->nvp_nitems; ii++) {
        len = strlen(tmp, size - 1) + 1;
        size -= len;

        // FIX: Explicitly check null-termination
        if (tmp[len - 1] != '\0') {
            errno_set(EINVAL);
            return (NULL);
        }
        tmp += len;
    }

    // FIX: Use nv_calloc to zero-initialize
    value = nv_calloc(nvp->nvp_nitems, sizeof(*value));
    if (value == NULL)
        return (NULL);

```

```
// ...
}
```

4.10.4.1 Visualising the Missing ‘\0’ Problem

CVE-2024-45288

Legend:

[..] = allocated bytes for one string element in the nvlist array

\0 = null terminator

XX = unrelated memory beyond the allocated buffer (must not be touched)

BEFORE (vulnerable): last string not null-terminated

nvlist data region (simplified):

```
+---- element[0] ----+ +---- element[1] ----+ +---- element[2] ----+
| 'F' 'r' 'e' 'e' \0 | | 'B' 'S' 'D'  \0  | | 'b' 'u' 'g'  '!'  |XX|XX|XX|...
+-----+ +-----+ +-----+ +-----+
                                     ^
                                     |
                                strlen(tmp, size-1) walks here,
                                never sees '\0', keeps going...
                                size -= len is computed as if '\0'
                                existed, later code writes past end
```

Effect:

- Readers assume a proper C-string. They continue until a random zero byte in XX.
- Writers may copy len bytes including overflow into XX.
- Result can be buffer overflow, memory corruption, or kernel panic.

AFTER (fixed): explicit check for null-termination + safer allocation

```
+---- element[0] ----+ +---- element[1] ----+ +---- element[2] ----+
| 'F' 'r' 'e' 'e' \0 | | 'B' 'S' 'D'  \0  | | 'b' 'u' 'g'  '!'  \0|XX|XX|XX|...
+-----+ +-----+ +-----+ +-----+
                                     ^
                                     |
                                check: tmp[len-1] == '\0' ? OK : EINVAL
```


Changes:

- The loop validates the final byte of each element is `'\0'`.
- If not, it fails early with `EINVAL`. No overflow occurs.
- Allocation uses `nv_calloc(nitems, sizeof(*value))`, memory is zeroed.

Tip for kernel developers:

Always check termination when parsing external or untrusted data.
Do not rely on `strlen` alone. Validate `tmp[len-1] == '\0'` before use.

4.10.4.2 Root Cause Analysis:**1. Missing null-termination check**

- `strlen()` was used to find string length.
- The code assumed strings ended with `'\0'`.
- No verification that `tmp[len-1] == '\0'`.

2. Uninitialized memory

- `nv_malloc()` does not clear memory.
- Changed to `nv_calloc()` to avoid leaking old memory contents.

3. Integer overflow

- In related header checks, `nvlh_size` could overflow when added to `sizeof(nvlhdrp)`
- Added explicit overflow checks.

4.10.4.3 Impact:

- Buffer overflow in kernel or userland.
- Privilege escalation is possible.
- System panic and memory corruption.

4.10.5 Mini Lab: The Danger of a Missing `'\0'`

To illustrate the subtlety of this class of bug, try the following small program in user space.

```
#include <stdio.h>
#include <string.h>

int main() {
    // Fill stack with non-zero garbage
    char garbage[100];
    memset(garbage, 'Z', sizeof(garbage));
```

```
// Deliberately forget the null terminator
char broken[5] = {'B', 'S', 'D', '!', 'X'};

// Print as if it were a string
printf("Broken string: %s\n", broken);

// Now with proper termination
char fixed[6] = {'B', 'S', 'D', '!', 'X', '\0'};
printf("Fixed string: %s\n", fixed);

return 0;
}
```

What to do:

- Compile and run.
- The first print may show random garbage after "BSD!X", because `printf("%s")` keeps reading memory until it stumbles on a zero byte.
- The second print works as expected.

Lesson: This is the same mistake that caused CVE-2024-45288 in FreeBSD. In user space, you get garbage or a crash. In kernel space, you risk a panic or privilege escalation. Always remember: **no '\0', no string.**

Note: This example shows how a **tiny omission, forgetting to check for a '\0', can become a serious vulnerability**. That's why professional FreeBSD driver developers are disciplined when handling arrays and strings: they always track buffer sizes, always validate string termination, and always use safe allocation and copy functions. The security and stability of the system depend on it.

4.10.6 Real Example from FreeBSD 14.3 Source Code

FreeBSD stores its kernel environment as an **array of C strings**, each of the form "name=value". This is a perfect real-world example of arrays and strings in action.

The array itself is declared in `sys/kern/kern_environment.c`:

```
// sys/kern/kern_environment.c - file-scope kenvp array
char **kenvp;    // Array of pointers to strings like "name=value"
```

Each `kenvp[i]` points to a null-terminated string. For example:

```
kenvp[0] → "kern.ostype=FreeBSD"
kenvp[1] → "hw.model=Intel(R) Core(TM) i7"
...
```

To look up a variable by name, FreeBSD uses the helper `_getenv_dynamic_locked()`:

```
// sys/kern/kern_environment.c - _getenv_dynamic_locked()
static char *
_getenv_dynamic_locked(const char *name, int *idx)
{
    char *cp;    // Pointer to the current "name=value" string
    int len, i;

    len = strlen(name); // Get the length of the variable name

    // Walk through each string in kenvp[]
    for (cp = kenvp[0], i = 0; cp != NULL; cp = kenvp[++i]) {
        // Compare prefix: does "cp" start with "name"?
        if ((strncmp(cp, name, len) == 0) &&
            (cp[len] == '=')) { // Ensure it's exactly "name="

            if (idx != NULL)
                *idx = i; // Optionally return the index

            // Return pointer to the value part (after '=')
            return (cp + len + 1);
        }
    }

    // Not found
    return (NULL);
}
```

Step by step explanation:

1. The function receives a variable name, such as "kern.ostype".
2. It measures its length.
3. It loops through the array kenvp[]. Each entry is a string like "name=value".
4. It compares the prefix of each entry with the requested name.
5. If it matches and is followed by '=', it returns a pointer **just past the '='**, so the caller gets only the value.
 - For "kern.ostype=FreeBSD", the return value points to "FreeBSD".
6. If no entry matches, it returns NULL.

The public interface kern_getenv() wraps this logic with safe copying and locking:

```
// sys/kern/kern_environment.c - kern_getenv()
char *
kern_getenv(const char *name)
{
    char *cp, *ret;
```

```

int len;

if (dynamic_kenv) {
    // Compute maximum safe size for a "name=value" string
    len = KENV_MNAMELEN + 1 + kenv_mvallen + 1;

    // Allocate a buffer (zeroed) for the result
    ret = uma_zalloc(kenv_zone, M_WAITOK | M_ZERO);

    mtx_lock(&kenv_lock);
    cp = _getenv_dynamic(name, NULL);    // Look up variable
    if (cp != NULL)
        strlcpy(ret, cp, len);          // Safe copy into buffer
    mtx_unlock(&kenv_lock);

    // If not found, free the buffer and return NULL
    if (cp == NULL) {
        uma_zfree(kenv_zone, ret);
        ret = NULL;
    }
} else {
    // Early boot path: static environment
    ret = _getenv_static(name);
}

return (ret);
}

```

What to notice:

- `kenvp` is an **array of strings** used as a lookup table.
- `_getenv_dynamic_locked()` walks the array, uses `strncmp()` and pointer arithmetic to isolate the value.
- `kern_getenv()` wraps this in a safe API: it locks access, copies the value with `strlcpy()`, and ensures memory ownership is clear (the caller must later `freeenv()` the result).

This real kernel code ties together almost everything we have discussed so far: **arrays of strings, null-terminated strings, standard string functions, and pointer arithmetic.**

4.10.7 Common Beginner Pitfalls

Arrays and strings in C look simple, but they hide many traps for beginners. Small mistakes that in user space would only crash your program can, in kernel space, bring down the entire operating system. Here are the most common issues:

- **Off-by-one errors** The most classic mistake is writing outside the valid range of an array. If you declare `int items[5];`, the valid indices are 0 through 4. Writing to `items[5]` is already one past the end, and you are corrupting memory. *Avoid it:* always think in terms of “zero to size minus one,” and double-check loop bounds carefully.
- **Forgetting the null terminator** A string in C must end with `'\0'`. If you forget it, functions like `printf("%s", ...)` will keep reading memory until they randomly find a zero byte, often printing garbage or causing a crash. *Avoid it:* let the compiler add the terminator by writing `char name[] = "FreeBSD";` instead of manually filling character arrays.
- **Using unsafe functions** Functions like `strcpy()` and `strcat()` perform no bounds checking. If the destination buffer is too small, they will happily overwrite memory past its end. In kernel code, this can cause panics or even security vulnerabilities. *Avoid it:* use safer alternatives such as `strncpy()` or `strlcat()`, which require you to pass the size of the buffer.
- **Assuming arrays know their own length** In higher-level languages, arrays often “know” how big they are. In C, an array is just a pointer to a block of memory; its size is not stored anywhere. *Avoid it:* keep track of the size explicitly, usually in a separate variable, and pass it along with the array whenever you share it between functions.
- **Mixing up arrays and pointers** Arrays and pointers are closely related in C, but not identical. For example, you cannot reassign an array the way you reassign a pointer, and `sizeof(array)` is not the same as `sizeof(pointer)`. Confusing the two leads to subtle bugs. *Avoid it:* remember: arrays “decay” into pointers when passed to functions, but at the declaration level they are distinct.

In user programs, these mistakes usually stop at a segmentation fault. In kernel drivers, they can overwrite scheduler data, corrupt I/O buffers, or break synchronization structures, leading to crashes or exploitable vulnerabilities. This is why FreeBSD developers are disciplined when working with arrays and strings: every buffer has a known size, every string has a checked terminator, and safe functions are preferred by default.

4.10.8 Hands-On Lab 1: Arrays in Practice

In this first lab you will practice the mechanics of arrays: declaring, initializing, looping over them, and modifying elements.

```
#include <stdio.h>

int main() {
    // Declare and initialize an array of 5 integers
    int values[5] = {10, 20, 30, 40, 50};

    printf("Initial array contents:\n");
    for (int i = 0; i < 5; i++) {
        printf("values[%d] = %d\n", i, values[i]);
    }

    // Modify one element
    values[2] = 99;
```

```
printf("\nAfter modification:\n");
for (int i = 0; i < 5; i++) {
    printf("values[%d] = %d\n", i, values[i]);
}

return 0;
}
```

What to Try Next

1. Change the array size to 10 but only initialize the first 3 elements. Print all 10 and notice that uninitialized ones default to zero (in this case, because the array was initialized with braces).
2. Move the `values[2] = 99;` line into the loop and try modifying every element. This is the same pattern drivers use when filling buffers with new data from hardware.
3. (Optional curiosity) Try printing `values[5]`. This is one step past the last valid element. On your system you might see garbage or nothing unusual, but in the kernel it could overwrite sensitive memory and crash the OS. Treat it as forbidden.

4.10.9 Hands-On Lab 2: Strings and the Null Terminator

This lab focuses on strings. You will see what happens when you forget the terminating `'\0'`, and then you'll practice comparing strings in a way that mirrors how FreeBSD drivers search configuration options.

Incorrect version (missing `'\0'`):

```
#include <stdio.h>

int main() {
    char word[5] = {'H', 'e', 'l', 'l', 'o'};
    printf("Broken string: %s\n", word);
    return 0;
}
```

Correct version:

```
#include <stdio.h>

int main() {
    char word[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
    printf("Fixed string: %s\n", word);
    return 0;
}
```

What to Try Next

1. Replace "Hello" with a longer word but keep the array size the same. See what happens when the word does not fit.
2. Declare `char msg[] = "FreeBSD";` without specifying a size and print it. Notice how the compiler automatically adds the null terminator for you.

Kernel-Flavoured Bonus Challenge

In the kernel, environment variables are stored as strings of the form "name=value". Drivers often need to compare names to find the right variable. Let's simulate that:

```
#include <stdio.h>
#include <string.h>

int main() {
    // Simulated environment variables (like entries in kenvp[])
    char *env[] = {
        "kern.ostype=FreeBSD",
        "hw.model=Intel(R) Core(TM)",
        "kern.version=14.3-RELEASE",
        NULL
    };

    // Target to search
    const char *name = "kern.ostype";
    int len = strlen(name);

    for (int i = 0; env[i] != NULL; i++) {
        // Compare prefix
        if (strncmp(env[i], name, len) == 0 && env[i][len] == '=') {
            printf("Found %s, value = %s\n", name, env[i] + len + 1);
            break;
        }
    }

    return 0;
}
```

Run it, and you will see:

```
Found kern.ostype, value = FreeBSD
```

This is almost exactly what `_getenv_dynamic_locked()` does inside the FreeBSD kernel: it compares names and, if they match, returns a pointer to the value after the '='.

4.10.10 Wrapping Up

In this section, you explored arrays and strings from both the C language perspective and the FreeBSD kernel perspective. You saw how arrays give you fixed-size storage, how strings depend on the null terminator, and how these simple constructs underpin core driver mechanisms such as device names, sysctl parameters, and kernel environment variables.

You also discovered how subtle mistakes, like writing past an array boundary or forgetting a terminator, can escalate into severe bugs or vulnerabilities, as illustrated by real FreeBSD CVEs.

4.10.11 Recap Quiz - Arrays and Strings

Instructions: answer without running code first, then verify on your system. Keep answers short and specific.

1. In C, what makes a character array a “string”? Explain what happens if that element is missing.
2. Given `int a[5];`, list the valid indices and say what is undefined behavior for indexing.
3. Why is `strcpy(dest, src)` risky in kernel code, and what should you prefer instead? Briefly explain why.
4. Look at this snippet and say exactly what the return value points to if it matches:

```
int len = strlen(name);  
if (strncmp(cp, name, len) == 0 && cp[len] == '=')  
    return (cp + len + 1);
```

1. In `sys/kern/kern_environment.c`, what is the type and role of `kenvp`, and how does `_getenv_dynamic_locked()` use it at a high level?

4.10.12 Challenge Exercises

If you feel confident, try these challenges. They are designed to push your skills a bit further and prepare you for real driver work.

1. **Array Rotation:** Write a program that rotates the contents of an integer array by one position. For example, `{1, 2, 3, 4}` becomes `{2, 3, 4, 1}`.
2. **String Trimmer:** Write a function that removes the newline character (`'\n'`) from the end of a string if present. Test it with input from `fgets()`.
3. **Environment Lookup Simulation:** Extend the kernel-flavoured lab from this section. Add a function `char *lookup(char *env[], const char *name)` that takes an array of `"name=value"` strings and returns the value part. Handle the case where the name is not found by returning `NULL`.
4. **Buffer Size Check:** Write a function that safely copies one string into another buffer and explicitly reports an error if the destination is too small. Compare your implementation with `strncpy()`.

4.10.13 Looking Ahead

In the next section, we will connect arrays and strings to the deeper concept of **pointers and memory**. You will learn how arrays decay into pointers, how memory addresses are manipulated, and how the FreeBSD kernel

allocates and frees memory safely. This is where you begin to see how data structures and memory management form the backbone of every device driver.

4.11 Pointers and Memory

Welcome to one of the most mysterious and magical topics in your C learning: **pointers**.

By now, you've probably heard things like:

- “Pointers are hard.”
- “Pointers are what give C its power.”
- “You can shoot yourself in the foot with pointers.”

Those statements aren't wrong, but don't worry. I'm going to walk you through it carefully, step by step. Our goal is to **demystify pointers**, not memorize obscure syntax. And because we're learning with FreeBSD in mind, I'll also point out where and how pointers are used in the real kernel source (without overwhelming you).

When you understand pointers, you'll unlock the true potential of C, especially when it comes to writing system-level code and interacting with the operating system at a low level.

4.11.1 What Is a Pointer?

So far, we've worked with variables like `int`, `char`, and `float`. These are familiar and friendly; you declare them, assign them values, and print them. Easy, right?

Now we're going to talk about something that doesn't store a value directly, but rather **stores the location of a value**.

This magical concept is called a **pointer**, and it's one of the most important tools in C, especially when you're writing low-level code like device drivers in FreeBSD.

4.11.1.1 Analogy: Memory as a Row of Lockers

Imagine computer memory as a long row of lockers, each with its own number:

```
[1000] = 42
[1004] = 99
[1008] = ???
```

Each **locker** is a memory address, and the **value** inside is your data.

When you create a variable in C:

```
int score = 42;
```

You're saying:

“Please give me a locker big enough to store an `int`, and put 42 in it.”

But what if you want to *know where* that variable is stored?

That’s where **pointers** come in.

4.11.1.2 A First Pointer Program

Here’s a gentle example to show what a pointer is, and what it can do:

```
#include <stdio.h>

int main(void) {
    int score = 42;           // A normal integer variable
    int *ptr;                 // Declare a pointer to an integer

    ptr = &score;             // Set ptr to the address of score using the & operator

    printf("score = %d\n", score);           // Prints: 42
    printf("ptr points to address %p\n", (void *)ptr); // Prints the memory address
    ↪ of score
    printf("value at that address = %d\n", *ptr); // Dereference the pointer to get
    ↪ the value: 42

    return 0;
}
```

Line-by-Line Breakdown

Line	Explanation
<code>int score = 42;</code>	Declares a regular <code>int</code> and sets it to 42.
<code>int *ptr;</code>	Declares a pointer named <code>ptr</code> that can store the address of an <code>int</code> .
<code>ptr = &score;</code>	The <code>&</code> operator gets the memory address of <code>score</code> . That address is now stored in <code>ptr</code> .
<code>*ptr</code>	The <code>*</code> operator (called dereference) means: “go to the address stored in <code>ptr</code> and get the value there.”

4.11.1.3 Pointers in the Kernel

Let’s look at a real pointer declaration in FreeBSD.

Remember our good friend, the `tty_info()` function from `sys/kern/tty_info.c`?

Inside it, you’ll find this declaration:

```
struct proc *p, *ppick;
```

Here, `p` and `ppick` are **pointers** to a `struct proc`, which represents a process.

What this line means:

- `p` and `ppick` don't *store* processes; they **point to** process structures in memory.
- In FreeBSD, almost all kernel structures are accessed via pointers because data is passed around and shared across kernel subsystems.

Later in the same function, we see:

```
/*  
 * Pick the most interesting process and copy some of its  
 * state for printing later. This operation could rely on stale  
 * data as we can't hold the proc slock or thread locks over the  
 * whole list. However, we're guaranteed not to reference an exited  
 * thread or proc since we hold the tty locked.  
 */  
p = NULL;  
LIST_FOREACH(ppick, &tp->t_pgrp->pg_members, p_pglist)  
    if (proc_compare(p, ppick))  
        p = ppick;
```

Here:

- `LIST_FOREACH()` is walking over a linked list of processes.
- `ppick` is pointing to each process in the group.
- The function `proc_compare()` helps pick the “*most interesting*” process.
- And `p` gets assigned to point to that process.

Don't worry if the kernel example feels a little dense right now. The key takeaway is simple:

in FreeBSD, pointers are everywhere because kernel structures are almost always shared and referenced instead of copied.

4.11.1.4 A Simple Analogy

Think of pointers as **labels with GPS coordinates**. Instead of carrying the treasure, they tell you where to dig.

- A **regular variable** holds the value.
- A **pointer** holds the address of the value.

This is extremely useful in systems programming, where we often pass **references to data** rather than the data itself.

4.11.1.5 Quick Check: Test Your Understanding

Can you tell what this code will print?

```
int num = 25;
int *p = &num;

printf("num = %d\n", num);
printf("*p = %d\n", *p);
```

Answer:

```
num = 25
*p = 25
```

Because both `num` and `*p` refer to the **same location** in memory.

4.11.1.6 Summary

- A pointer is a variable that **stores a memory address**.
- Use `&` to get the address of a variable.
- Use `*` to access the value at a pointer's address.
- Pointers are heavily used in FreeBSD (and all OS kernels) because they allow efficient access to shared and dynamic data.

4.11.1.7 Mini Hands-On Lab: Your First Pointers

Goal Build confidence with the three core pointer moves: taking an address with `&`, storing it in a pointer, and reading or writing through that pointer with `*`.

Starter code

```
#include <stdio.h>

int main(void) {
    int value = 10;
    int *p = NULL;           // Good habit: initialize pointers

    /* 1. Point p to value using the address of operator */

    /* 2. Print value, the address stored in p, and the value via *p */

    /* 3. Change value through the pointer, then print value again */
```

```
/* 4. Declare another int named other = 99 and re point p to it.
   Print *p and other to confirm they match. */

return 0;
}
```

Tasks

1. Set `p` to the address of `value`.
2. Print:
 - `value = ...`
 - `p` points to address ... (use `printf("p points to address %p\n", (void*)p);`)
 - `*p = ...`
3. Write through the pointer with `*p = 20`; and print `value` again.
4. Create `int other = 99`; then set `p = &other`; and print `*p` and `other`.

Expected output example (address will differ):

```
value = 10
p points to address 0x...
*p = 10
value after write through p = 20
other = 99
*p after re pointing = 99
```

Stretch exercise

- Add `int *q = p`; and then set `*q = 123`; . Print both `*p` and `other`. What happened?
- Write a helper function:

```
void set_twice(int *x) { *x = *x * 2; }
```

Call it with `set_twice(&value)`; and observe the result.

4.11.1.8 Common Beginner Pitfalls with Pointers

If pointers feel slippery, you're not alone. Most C beginners run into the same traps over and over, and even experienced developers occasionally fall into them.

- **Using an uninitialized pointer** Declaring `int *p`; without setting it to something valid leaves `p` pointing to "somewhere" in memory. → Always initialize pointers (to `NULL` or a valid address).
- **Confusing the pointer with the data** Beginners often mix up `p` (the address) with `*p` (the value at that address). Writing to the wrong one can silently corrupt memory. → Ask yourself: am I working with the pointer or the pointee?

- **Losing track of ownership** If a pointer refers to memory that was freed or that belongs to a different part of the program, using it again is a serious bug (a “dangling pointer”). → We’ll learn strategies to manage memory safely later.
- **Forgetting about types** A pointer to `int` is not the same as a pointer to `char`. Mixing types can cause subtle errors because the compiler uses the type to decide how many bytes to step through in memory. → Always match pointer types carefully.
- **Assuming all addresses are valid** Just because a pointer contains a number doesn’t mean that address is safe to use. The kernel is full of memory that your code must not touch without permission. → Never invent or guess addresses; only use valid ones from the kernel or OS.

These mistakes are not small annoyances; in kernel development, they can bring down the entire system. The good news is that by understanding how pointers work and building safe habits, you’ll learn to avoid them.

4.11.1.9 Why Pointers Matter in Driver Development

So why spend so much time learning pointers? Because pointers are the language of the kernel!

- In user programs, you often work with copies of data. In the kernel, **copying is too expensive**, so we pass around pointers instead.
- Device drivers constantly need to share state between different parts of the system (processes, threads, hardware buffers). Pointers are how we make that possible.
- Pointers let us build flexible structures like **linked lists, queues, and tables**, which are everywhere in FreeBSD’s source code.
- Most importantly, **hardware itself is accessed through memory addresses**. If you want to talk to a device, you’ll often be handed a pointer to its registers or buffers.

Understanding pointers is not just about writing clever C code. It’s about speaking the kernel’s native tongue. Without them, you cannot build safe, efficient, or even functional device drivers.

4.11.1.10 Wrapping Up

We’ve just taken our first careful step into the world of pointers: variables that don’t hold data directly but instead hold the location of data. This shift in perspective is what makes C so flexible and, at the same time, so dangerous if misunderstood.

Pointers let us share information between parts of a program without copying it around, which is essential in an operating system kernel where efficiency and precision matter. But this power also comes with responsibility: mixing up addresses, dereferencing invalid pointers, or forgetting what memory a pointer refers to can easily crash your program or, in the case of a driver, the entire operating system.

That’s why understanding pointers is not just an academic exercise, but a survival skill for FreeBSD driver development.

In the next section, we’ll move from the big picture into the nuts and bolts: how to correctly **declare pointers**, how to **use them in practice**, and how to start building **safe habits from the very beginning**.

4.11.2 Declaring and Using Pointers

Now that you know what a pointer is, a variable that stores a memory address instead of a direct value, it is time to learn how to declare and use them in your programs. This is where the idea of a pointer stops being abstract and becomes something you can experiment with in code.

We will move carefully, step by step, with small and fully commented examples. You will see how to declare a pointer, how to assign it the address of another variable, how to dereference it to reach the stored value, and how to modify data indirectly. Along the way, we will look at real FreeBSD kernel code that relies on pointers every day.

4.11.2.1 Declaring a Pointer

In C, you declare a pointer using the star symbol *. The general pattern looks like this:

```
int *ptr;
```

This line means:

"I am declaring a variable called ptr, and it will hold the address of an integer."

The * here does not mean that the name of the variable is *ptr. It is part of the type declaration, telling the compiler that ptr is not a plain integer but a pointer to an integer.

Let's see a complete program that you can type and run:

```
#include <stdio.h>

int main(void) {
    int value = 10;          // Declare a regular integer
    int *ptr;                // Declare a pointer to an integer

    ptr = &value;            // Store the address of 'value' in 'ptr'

    printf("The value is: %d\n", value);
    printf("Address of value is: %p\n", (void *)&value);
    printf("Pointer ptr holds: %p\n", (void *)ptr);
    printf("Value through ptr: %d\n", *ptr);

    return 0;
}
```

Run this program and compare the output. You will see that ptr contains the same address as &value, and when you use *ptr, you get back the integer stored there.

Think of it like this: writing down a friend's street address is &value. Saving that address in your contacts list is ptr. Actually going to that house to say hello (or grab a snack) is *ptr.

4.11.2.2 The Importance of Initialisation

One of the most important rules about pointers is: never use them before you assign them a valid address. An uninitialised pointer holds garbage data, which means it may point to a random memory location. If you try to dereference it, the program will almost certainly crash.

Here is an unsafe example:

```
int *dangerous_ptr;  
printf("%d\n", *dangerous_ptr); // Undefined behaviour!
```

Since `dangerous_ptr` was never assigned a valid address, the program will attempt to read some unpredictable area of memory. In user programs this usually causes a crash. In kernel code it can be far worse, leading to corruption of critical data structures and even security vulnerabilities. This is why being disciplined about initialisation is so important when programming for FreeBSD.

4.11.2.3 A Real Example from FreeBSD

If you open the file `sys/kern/tty_info.c`, you will find the following declaration inside the `tty_info()` function:

```
struct thread *td, *tdpick;
```

Both `td` and `tdpick` are pointers to a structure called `thread`. FreeBSD uses these pointers to walk through all threads that belong to a process. Later in the same function, you will see these pointers being utilised:

```
FOREACH_THREAD_IN_PROC(p, tdpick)  
    if (thread_compare(td, tdpick))  
        td = tdpick;
```

The kernel is looping through each thread in process `p`. It compares them using the helper function `thread_compare()`, and if one is a better match, it updates the pointer `td` to refer to that thread.

Notice that `td` itself is just a label. What changes is the address it holds, which in turn tells the kernel which thread to focus on. This pattern is extremely common in the kernel: pointers are declared at the top of a function, then updated step by step as the function works its way through structures.

4.11.2.4 Modifying a Value Through a Pointer

Another classic use of pointers is indirect modification. Let's walk through a simple program:

```
#include <stdio.h>  
  
int main(void) {
```



```
int age = 30;           // A regular variable
int *p = &age;          // Pointer to that variable

printf("Before: age = %d\n", age);

*p = 35;                // Change the value through the pointer

printf("After: age = %d\n", age);

return 0;
}
```

When we run this code, it prints 30 before and 35 after. We never assigned directly to `age`, but by dereferencing `p`, we reached into its memory location and changed the value stored there.

This technique is used everywhere in system programming. Functions that need to return more than one value, or that must directly alter data structures inside the kernel, rely on pointers. Without them, it would be impossible to write efficient drivers or manage complex objects like processes, devices, and memory buffers.

4.11.2.5 Mini Hands-On Lab: Pointer Chains

Goal Learn how multiple pointers can point to the same variable, and how a pointer-to-pointer works in practice. This prepares you for real kernel patterns where functions receive not just data but pointers to pointers that need updating.

Starter Code

```
#include <stdio.h>

int main(void) {
    int value = 42;
    int *p = &value;      // p points to value
    int **pp = &p;        // pp points to p (pointer to pointer)

    /* 1. Print value directly, through p, and through pp */

    /* 2. Change value to 100 using *p */

    /* 3. Change value to 200 using **pp */

    /* 4. Make p point to a new variable other = 77, via pp */

    /* 5. Print value, other, *p, and **pp to observe the changes */
}
```

```
    return 0;
}
```

Tasks

1. Print the same integer in three different ways:
 - Directly (value)
 - Indirectly with *p
 - Double indirection with **pp
2. Assign 100 to value using *p, then print value.
3. Assign 200 to value using **pp, then print value.
4. Declare a second variable `int other = 77;` Use the double pointer (`*pp = &other;`) to make p point to other instead.
5. Print other, *p, and **pp. Confirm that all three match.

Expected Output (addresses will differ)

```
value = 42
*p = 42
**pp = 42
value after *p write = 100
value after **pp write = 200
other = 77
*p now points to other = 77
**pp also sees 77
```

Stretch Exercise

Write a function that takes a pointer to a pointer:

```
void redirect(int **pp, int *new_target) {
    *pp = new_target;
}
```

- Call it to redirect p from value to other.
- Print *p afterwards to see the result.

This is a common idiom in kernel code, where functions receive a pointer to a pointer so they can safely update what the caller's pointer points to.

4.11.2.6 Common Beginner Pitfalls at Declaration and Use

When you start declaring and using pointers, a new set of mistakes can creep in. They are different from the conceptual traps we already covered, and each has a simple way to avoid it.

One mistake is misplacing the star `*` in a declaration. Writing `int* a, b;` looks like you declared two pointers, but in reality only `a` is a pointer, and `b` is a plain integer. To avoid this confusion, always write the star next to each variable name: `int *a, *b;`. This makes it explicit that both are pointers.

Another trap is assigning a pointer without matching the type. For instance, storing the address of a `char` in an `int *` may compile with warnings but is unsafe. Always ensure the pointer type matches the type of the variable you are pointing to. If you need to work with different types, use casts carefully and deliberately, not by accident.

A common error is dereferencing too early. Beginners sometimes write `*p` before assigning `p` a valid address, which leads to undefined behaviour. Get into the habit of initialising pointers at declaration. Use `NULL` if you do not yet have a valid address, and only dereference after you have assigned a real target.

Another pitfall is overthinking addresses. Printing or comparing the raw numeric value of pointers is rarely meaningful. What matters is the relationship between a pointer and its pointee. Focus on using `%p` in `printf` to display addresses for debugging, but remember that addresses themselves are not portable values you can calculate casually.

Finally, declaring multiple pointers in one line without care often causes subtle errors. A line like `int *a, b, c;` gives you one pointer and two integers, not three pointers. To avoid mistakes, keep pointer declarations simple and clear, and never assume all variables in a list share the same pointer type.

By adopting these habits early, clear declarations, matching types, safe initialisation, and careful dereferencing, you will build a strong foundation for working with pointers in larger programs and in FreeBSD kernel code.

4.11.2.7 Why This Matters in Driver Development

Declaring and using pointers correctly is more than a matter of style. In FreeBSD drivers, you will often see entire groups of pointers declared together, each one tied to a subsystem of the kernel or a hardware resource. If you misdeclare one of these pointers, you might end up mixing integers and addresses, leading to very subtle bugs.

Consider linked lists in the kernel. A device driver might declare several pointers to structures like `struct mbuf` (network buffers) or `struct cdev` (device entries). These pointers are chained together to form queues, and every declaration must be precise. One missing star or one mismatched type can mean that a list traversal ends in a kernel panic.

Another reason declaration matters is efficiency. Kernel code often creates pointers that refer to existing objects instead of making copies. Declaring pointers correctly means you can traverse large structures, like the list of threads in a process, without duplicating data or wasting memory.

The lesson is clear: understanding how to declare and use pointers properly gives you the vocabulary to describe kernel objects, navigate them, and connect them together safely.

4.11.2.8 Wrapping Up

At this point, you have moved beyond simply knowing what a pointer is. You can now declare a pointer, initialise it, print its address, dereference it, and even use it to modify a variable indirectly. You have seen how FreeBSD uses these techniques in real code, such as iterating through process threads or updating kernel structures. You

also practised with pointer chains and saw how a pointer-to-pointer lets you redirect another pointer, a pattern that will appear often in kernel APIs.

What makes this knowledge valuable is that it transforms your ability to work with functions. Passing pointers into functions allows those functions to update the caller's data, to redirect a pointer to a new target, or to return multiple results at once. In kernel development, this is not a rare pattern but the norm. Drivers almost always interact with the kernel and hardware by passing pointers into functions and receiving updated pointers back.

The next section covers **Pointers and Functions**, where you will see how this combination becomes the standard way to write flexible, efficient, and safe code inside FreeBSD.

4.11.3 Pointers and Functions

Back in Section 4.7, we learned that function parameters are always passed by value. This means that when you call a function, it usually receives only a copy of the variable you provide. As a result, the function can't change the original value that lives in the caller.

Now that we've introduced pointers, we have a new possibility. By passing a pointer into a function, you give it direct access to the caller's memory. This is the standard way in C and especially in FreeBSD kernel code to let functions modify data outside their own scope or return multiple results.

Let's walk through the difference step by step.

4.11.3.1 First Try: Passing by Value (Doesn't Work)

```
#include <stdio.h>

void set_to_zero(int n) {
    n = 0; // Only changes the copy
}

int main(void) {
    int x = 10;

    set_to_zero(x);
    printf("x is now: %d\n", x); // Still prints 10!

    return 0;
}
```

Here, the function `set_to_zero()` receives a copy of `x`. That copy is modified, but the real `x` in `main()` never changes.

4.11.3.2 Second Try: Passing by Pointer (Works)

```
#include <stdio.h>

void set_to_zero(int *n) {
    *n = 0; // Follow the pointer and change the real variable
}

int main(void) {
    int x = 10;

    set_to_zero(&x); // Give the function the address of x
    printf("x is now: %d\n", x); // Prints 0!

    return 0;
}
```

This time the caller sends the address of `x` using `&x`. Inside the function, `*n` lets us reach into that memory location and actually change the variable in `main()`.

This pattern is simple but essential. It transforms functions from isolated workshops into tools that can work directly on the caller's data.

4.11.3.3 Why This Matters in the Kernel

In kernel code, this technique is not just helpful, it's essential. Kernel functions often need to report back multiple pieces of information. Since C does not allow multiple return values, the usual approach is to pass pointers to variables or structures that the function can fill in.

Here's an example you can find inside `tty_info()` in the FreeBSD source file `sys/kern/tty_info.c`:

```
rufetchcalc(p, &ru, &utime, &stime);
```

The function `rufetchcalc()` fills in statistics about CPU usage for a process. It can't simply "return" all three results, so it accepts pointers to variables where it will write the data.

Let's simplify this with a small simulation:

```
#include <stdio.h>

// A simplified kernel-style function
void get_times(int *user, int *system) {
    *user = 12; // Fake "user time"
    *system = 8; // Fake "system time"
}
```

```
int main(void) {
    int utime, stime;

    get_times(&utime, &stime);

    printf("User time: %d\n", utime);
    printf("System time: %d\n", stime);

    return 0;
}
```

Here, `get_times()` updates both `utime` and `stime` in one call. That's precisely how kernel code returns complex results without extra overhead.

4.11.3.4 Common Beginner Pitfalls

Pointers with functions are a frequent stumbling block for beginners. Watch out for these mistakes:

- **Forgetting the & in the call:** If you write `set_to_zero(x)` instead of `set_to_zero(&x)`, you'll pass the value instead of the address, and nothing will change.
- **Assigning the pointer, not the value:** Inside the function, writing `n = 0;` only overwrites the pointer itself. You must use `*n = 0;` to change the caller's variable.
- **Overstepping responsibilities:** A function should not free or reallocate memory that belongs to the caller unless it is explicitly designed to do so. Otherwise, you risk creating dangling pointers.

The safest habit is always to be clear about what the pointer represents, and to think carefully before modifying anything that belongs to the caller.

4.11.3.5 Hands-On Lab: Write Your Own Setter

Here's a small challenge to test your understanding. Complete the function so that it doubles the value of the variable given:

```
#include <stdio.h>

void double_value(int *n) {
    // TODO: Write code that makes *n twice as large
}

int main(void) {
    int x = 5;

    double_value(&x);
}
```

```
    printf("x is now: %d\n", x); // Should print 10

    return 0;
}
```

If your function works, you’ve just written your first function that modifies a caller’s variable using a pointer, precisely the kind of operation you’ll use constantly in device drivers.

4.11.3.6 Wrapping Up: Pointers Unlock New Possibilities

By themselves, functions only work with copies. With pointers, functions gain the ability to modify the caller’s variables and to pass back multiple results efficiently. This pattern shows up everywhere in FreeBSD, from memory management to process scheduling, and it is one of the most essential techniques you can master as a future driver developer.

Now that we’ve seen how pointers connect with functions, let’s take the next step. Pointers also form a natural partnership with another key feature of C: arrays. In the next section, we’ll explore **Pointers and Arrays: A Natural Pair**, and you’ll discover how these two concepts work together to make memory access both flexible and efficient.

4.11.4 Pointers and Arrays: A Natural Pair

In C, arrays and pointers are like close friends. They are not the same thing, but they are deeply connected, and in practice, they often work together. This connection shows up constantly in kernel code, where performance and direct memory access matter. If you can understand how arrays and pointers interact, you will unlock a flexible toolset for navigating buffers, strings, and hardware data.

4.11.4.1 What’s the Connection?

There is a straightforward rule that explains most of the relationship:

In most expressions, the name of an array acts like a pointer to its first element.

That means if you declare:

```
int numbers[3] = {10, 20, 30};
```

Then the name `numbers` can be treated as if it were the same as `&numbers[0]`. So the line:

```
int *ptr = numbers;
```

Is equivalent to:

```
int *ptr = &numbers[0];
```

The pointer `ptr` now points directly to the first element of the array.

4.11.4.2 A Simple Example

```
#include <stdio.h>

int main(void) {
    int values[3] = {100, 200, 300};

    int *p = values; // 'values' behaves like &values[0]

    printf("First value: %d\n", *p);          // prints 100
    printf("Second value: %d\n", *(p + 1));  // prints 200
    printf("Third value: %d\n", *(p + 2));   // prints 300

    return 0;
}
```

Here, the pointer `p` starts at the first element. Adding 1 to `p` moves it forward by one integer, and so on. This is called **pointer arithmetic**, and we will study its rules more carefully in the next section. For now, the key idea is that arrays and pointers share the same memory layout, which makes moving through an array with a pointer both natural and efficient.

4.11.4.3 Using Arrays and Pointers in FreeBSD

The FreeBSD kernel makes heavy use of this connection. A good example is found inside `sys/kern/tty_info.c` in the `tty_info()` function:

```
strncpy(comm, p->p_comm, sizeof comm);
```

Here, `p->p_comm` is a character array that belongs to a process structure. The variable `comm` is another array declared locally at the top of `tty_info()`:

```
char comm[MAXCOMLEN + 1];
```

The function `strncpy()` copies the string from one array into another. Under the hood, it uses pointer arithmetic to walk through each character until the copy is done. You do not need to see those details to use it, but it is important to know that arrays and pointers make this possible. This is why so many kernel functions operate on “char *” even though you often start with a character array.

4.11.4.4 Arrays and Pointers: The Differences That Matter

Since arrays and pointers behave in similar ways, it is tempting to think they are the same thing. But they are not, and understanding the differences will help you avoid many subtle bugs.

When you declare an array, the compiler reserves a fixed block of memory large enough to hold all its elements. The array's name represents this memory location, and that association cannot be changed. For example, if you declare `int a[5];`, the compiler allocates space for five integers, and `a` will always refer to that same block of memory. You cannot later reassign `a` to point somewhere else.

A pointer, by contrast, is a variable that stores an address. It does not allocate storage for multiple items by itself. Instead, it can point to any valid memory location you choose. For instance, `int *p;` creates a pointer that may later hold the address of the first element of an array, the address of a single variable, or memory that has been allocated dynamically. You can also reassign the pointer freely, making it a much more flexible tool.

Another key distinction is that the compiler knows the size of an array, but it does not track the size of the memory a pointer refers to. This means the array's boundaries are known at compile time, while a pointer only knows where it starts, not how far it extends. That responsibility falls on you as the programmer.

These rules can be summarised in plain language. An array is a fixed block of storage, like a house built on a specific lot of land. A pointer is like a set of keys: you can use them to access that house, but tomorrow you might use the same keys to open another house entirely. Both are useful, but they serve different purposes, and the distinction matters in kernel code where memory management and safety cannot be left to chance.

4.11.4.5 Common Beginner Pitfall: Off-by-One Errors

Because arrays and pointers are so closely related, the same mistake can happen in two different guises: stepping one element too far.

With arrays, the error looks like this:

```
int items[3] = {1, 2, 3};
printf("%d\n", items[3]); // Invalid! Out of bounds
```

Here, the valid indices are 0, 1, and 2. Using 3 goes past the end.

With pointers, the same error can happen more subtly:

```
int items[3] = {1, 2, 3};
int *p = items;

printf("%d\n", *(p + 3)); // Also invalid, same as items[3]
```

In both cases, you are asking for memory beyond the array's boundary. The compiler will not stop you, and the program may even seem to run correctly sometimes, which makes the bug even more dangerous.

In user programs, this usually means corrupted data or a crash. In kernel code, it can mean memory corruption, a kernel panic, or even a security hole. That is why experienced FreeBSD developers are extremely careful when writing loops that walk through arrays or buffers with pointers. The loop condition is just as important as the loop body.

4.11.4.5.1 Safety Habit The best way to avoid off-by-one errors is to make your loop boundaries explicit and double-check them. If an array has n elements, valid indices always run from 0 to $n - 1$. When using a pointer, think in terms of “how many elements have I advanced?” rather than “how many bytes.”

For example:

```
for (int i = 0; i < 3; i++) {
    printf("%d\n", items[i]); // Safe, i goes 0..2
}

for (int i = 0; i < 3; i++) {
    printf("%d\n", *(p + i)); // Safe, same rule
}
```

By making the upper limit part of your loop condition, you ensure you never walk past the end of the array. This habit will save you from many subtle bugs, especially when you move from small exercises into real kernel code.

4.11.4.6 Why FreeBSD Style Embraces This Duo

Many FreeBSD subsystems manage buffers as arrays while navigating them with pointers. This combination lets the kernel avoid unnecessary copying, keep operations efficient, and interact directly with hardware. Whether you are looking at character device buffers, network packet rings, or process command names, you will see this pattern again and again.

By mastering the array-pointer relationship, you will be able to read and write kernel code more confidently, recognizing when the code is simply walking through memory one element at a time.

4.11.4.7 Hands-On Lab: Walking an Array with a Pointer

Try writing a small program that prints all the elements of an array using a pointer instead of array indexing.

```
#include <stdio.h>

int main(void) {
    int numbers[5] = {10, 20, 30, 40, 50};
    int *p = numbers;

    for (int i = 0; i < 5; i++) {
        printf("numbers[%d] = %d\n", i, *(p + i));
    }

    return 0;
}
```

Now, modify the loop so that instead of writing `*(p + i)`, you increment the pointer directly:

```
for (int i = 0; i < 5; i++) {  
    printf("numbers[%d] = %d\n", i, *p);  
    p++;  
}
```

Notice that the result is the same. This is the power of combining pointers and arrays. Try experimenting by starting the pointer at `&numbers[2]` and see what gets printed.

4.11.4.8 Wrapping Up: Arrays and Pointers Work Best Together

You have now seen how arrays and pointers fit together. Arrays provide the structure, while pointers provide the flexibility to navigate memory efficiently. In the FreeBSD kernel, this combination is everywhere, from device buffers to string manipulation. Always remember the two golden rules: `array[i]` is equivalent to `*(array + i)`, and you must never step outside the bounds of the array.

The next section covers Pointer Arithmetic more deeply. You will learn how incrementing a pointer works under the hood, why it follows the size of the type, and what boundaries you must respect to avoid stepping into dangerous memory.

4.11.5 Pointer Arithmetic and Boundaries

Now that you know how pointers can point to individual variables and even to arrays, we are ready to take the next step: learning how to move those pointers around. This ability is called **pointer arithmetic**.

The name might sound intimidating, but the idea is simple. Imagine a row of boxes placed neatly side by side. A pointer is like your finger pointing to one of those boxes. Pointer arithmetic is nothing more than moving your finger forward or backward to reach another box.

4.11.5.1 What Is Pointer Arithmetic?

When you add or subtract an integer from a pointer, C advances the pointer by **elements**, not by raw bytes. The step size depends on the type the pointer refers to:

- If `p` is an `int *`, then `p + 1` moves by `sizeof(int)` bytes.
- If `q` is a `char *`, then `q + 1` moves by `sizeof(char)` bytes (which is always 1).
- If `r` is a `double *`, then `r + 1` moves by `sizeof(double)` bytes.

This behaviour is what makes pointer arithmetic natural for walking arrays, because arrays live in contiguous memory. Each “+1” lands you precisely on the next element, not in the middle of it.

Let’s see a complete program that demonstrates this. I have added comments to allow you to understand what’s happening at each step.

Save it as `pointer_arithmetic_demo.c`:

```
/*
 * Pointer Arithmetic Demo (commented)
 *
 * This program shows that when you add 1 to a pointer, it moves by
 * the size of the element type (in elements, not raw bytes).
 * It prints addresses and values for int, char, and double arrays,
 * and then shows how to compute the distance between two pointers
 * using ptrdiff_t. All accesses stay within bounds.
 */

#include <stdio.h>    // printf
#include <stddef.h>    // ptrdiff_t, size_t

int main(void) {
    /*
     * Three arrays of different element types.
     * Arrays live in contiguous memory, which is why pointer arithmetic
     * can step through them safely when we stay within bounds.
     */
    int    arr[] = {10, 20, 30, 40};
    char    text[] = {'A', 'B', 'C', 'D'};
    double  nums[] = {1.5, 2.5, 3.5};

    /*
     * Pointers to the first element of each array.
     * In most expressions, an array name "decays" to a pointer to its first element.
     * So arr has type "array of int", but here it becomes "int *" automatically.
     */
    int    *p = arr;
    char    *q = text;
    double  *r = nums;

    /*
     * Compute the number of elements in each array.
     * sizeof(array) gives the total bytes in the array object.
     * sizeof(array[0]) gives the bytes in one element.
     * Dividing the two gives the element count.
     */
    size_t n_ints    = sizeof(arr) / sizeof(arr[0]);
    size_t n_chars    = sizeof(text) / sizeof(text[0]);
    size_t n_doubles  = sizeof(nums) / sizeof(nums[0]);

    printf("Pointer Arithmetic Demo\n\n");
}
```

```
/*
 * INT DEMO
 * p + i moves i elements forward, which is i * sizeof(int) bytes.
 * We print both the address and the value to see how it steps.
 * %zu is the correct format specifier for size_t.
 * %p prints a pointer address; cast to (void *) for correct printf typing.
 */
printf("== int demo ==\n");
printf("sizeof(int) = %zu bytes\n", sizeof(int));
for (size_t i = 0; i < n_ints; i++) {
    printf("p + %zu -> address %p, value %d\n",
        i, (void*)(p + i), *(p + i));    // *(p + i) reads the i-th element
}
printf("\n");

/*
 * CHAR DEMO
 * For char, sizeof(char) is 1 by definition.
 * q + i advances one byte per step.
 */
printf("== char demo ==\n");
printf("sizeof(char) = %zu byte\n", sizeof(char));
for (size_t i = 0; i < n_chars; i++) {
    printf("q + %zu -> address %p, value '%c'\n",
        i, (void*)(q + i), *(q + i));
}
printf("\n");

/*
 * DOUBLE DEMO
 * For double, sizeof(double) is typically 8 on modern systems.
 * r + i advances by 8 bytes per step on those systems.
 */
printf("== double demo ==\n");
printf("sizeof(double) = %zu bytes\n", sizeof(double));
for (size_t i = 0; i < n_doubles; i++) {
    printf("r + %zu -> address %p, value %.1f\n",
        i, (void*)(r + i), *(r + i));
}
printf("\n");

/*
 * Pointer difference
 * Subtracting two pointers that point into the same array
```

```
* yields a value of type ptrdiff_t that represents the distance
* in elements, not bytes.
*
* Here, a points to arr[0] and b points to arr[3].
* The difference b - a is 3 elements.
*/
int *a = &arr[0];
int *b = &arr[3];
ptrdiff_t diff = b - a; // distance in ELEMENTS

/*
* %td is the correct format specifier for ptrdiff_t.
* We also verify that advancing a by diff elements lands exactly on b.
*/
printf("Pointer difference (b - a) = %td elements\n", diff);
printf("Check: a + diff == b ? %s\n", (a + diff == b) ? "yes" : "no");

/*
* Program ends successfully.
*/
return 0;
}
```

Compile and run it on FreeBSD:

```
% cc -Wall -Wextra -o pointer_arithmetic_demo pointer_arithmetic_demo.c
% ./pointer_arithmetic_demo
```

You will see how each type moves by its element size. The `int *` steps by 4 bytes (on most modern FreeBSD systems), the `char *` steps by 1, and the `double *` steps by 8. Notice how the addresses jump accordingly, while the values are fetched correctly with `*(pointer + i)`.

The final check with `b - a` shows that pointer subtraction is also measured in **elements, not bytes**. If `a` points to the start of the array and `b` points three elements ahead, then `b - a` gives 3.

This program demonstrates the essential rule of pointer arithmetic: **C moves pointers in units of the pointed-to type**. That is why it works so well with arrays, but also why you must be careful, a wrong step can quickly take you beyond the valid memory region.

4.11.5.2 Walking Through Arrays with Pointers

This property makes pointers especially useful when working with arrays. Since arrays are laid out in contiguous blocks of memory, a pointer can naturally step through them one element at a time.

```
#include <stdio.h>

int main() {
    int numbers[] = {1, 2, 3, 4, 5};
    int *ptr = numbers; // Start at the first element

    for (int i = 0; i < 5; i++) {
        printf("Element %d: %d\n", i, *(ptr + i));
        // *(ptr + i) is equivalent to numbers[i]
    }

    return 0;
}
```

Here, the expression `*(ptr + i)` asks C to move forward `i` positions from the starting pointer, then fetch the value at that location. The result is identical to `numbers[i]`. In fact, C allows both notations interchangeably. Whether you write `numbers[i]` or `*(numbers + i)`, you are doing the same thing.

4.11.5.3 Staying Within Boundaries

Pointer arithmetic is flexible, but it comes with a serious responsibility: you must never move beyond the memory that belongs to your array. If you do, you step into undefined behaviour. That can mean a crash, corrupted memory, or silent errors that appear much later.

```
#include <stdio.h>

int main() {
    int data[] = {42, 77, 99};
    int *ptr = data;

    // Wrong! This goes past the last element.
    printf("Invalid access: %d\n", *(ptr + 3));

    return 0;
}
```

The array `data` has three elements, valid at indices 0, 1, and 2. But `ptr + 3` points to a place immediately after the last element. C does not stop you, but the result is unpredictable.

The safe way is to always respect the number of elements in your array. Instead of hardcoding the size, you can calculate it:

```
for (int i = 0; i < sizeof(data) / sizeof(data[0]); i++) {  
    printf("%d\n", *(data + i));  
}
```

This expression divides the total size of the array by the size of a single element, giving the correct element count regardless of the array's length.

4.11.5.4 A Glimpse into the FreeBSD Kernel

Pointer arithmetic shows up frequently in the FreeBSD kernel. Sometimes it is used directly with arrays, but more often it appears while navigating through structures linked together in memory. Let us look at a small excerpt adapted from `tty_info()` in `sys/kern/tty_info.c`:

```
struct proc *p, *ppick;  
  
p = NULL;  
LIST_FOREACH(ppick, &tp->t_pgrp->pg_members, p_pglist) {  
    if (proc_compare(p, ppick))  
        p = ppick;  
}
```

This loop is not using `ptr + 1` as in our array examples, but it is doing the same conceptual job: moving through memory by following pointer links. Instead of stepping through integers in a row, it walks through a chain of process structures connected in a list. The lesson is the same: pointers let you move from one element to the next, but you must always be careful to stay within the intended bounds of the structure.

4.11.5.5 Common Beginner Pitfalls

1. **Forgetting that pointers move by type size:** If you assume `p + 1` moves one byte, you will misunderstand the result. It always moves by the size of the type the pointer refers to.
2. **Going past array limits:** Accessing `arr[5]` when the array has only 5 elements (valid indices 0-4) is a classic off-by-one mistake.
3. **Mixing arrays and pointers too casually:** Although `arr[i]` and `*(arr + i)` are equivalent, an array name itself is not a modifiable pointer. Beginners sometimes try to reassign an array name as if it were a variable, which is not allowed.

To avoid these traps, always calculate sizes carefully, use `sizeof` when possible, and keep in mind that arrays and pointers are close relatives but not identical twins.

4.11.5.6 Tip: Arrays Decay into Pointers

When you pass an array to a function, C actually gives the function only a pointer to the first element. The function has no way to know how many elements exist. For safety, always pass the array size along with the pointer. This is why so many C library and FreeBSD kernel functions include both a buffer pointer and a length parameter.

4.11.5.7 Mini Hands-On Lab: Walking Safely with Pointer Arithmetic

In this lab, you will experiment with pointer arithmetic on arrays, see how to walk through memory step by step, and learn how to detect and prevent stepping outside array boundaries.

Create a file called `lab_pointer_bounds.c` with the following code:

```
#include <stdio.h>

int main(void) {
    int values[] = {5, 10, 15, 20};
    int *ptr = values; // Start at the first element
    int length = sizeof(values) / sizeof(values[0]);

    printf("Array has %d elements\n", length);

    // Walk forward through the array
    for (int i = 0; i < length; i++) {
        printf("Forward step %d: %d\n", i, *(ptr + i));
    }

    // Walk backwards using pointer arithmetic
    for (int i = length - 1; i >= 0; i--) {
        printf("Backward step %d: %d\n", i, *(ptr + i));
    }

    // Demonstrate boundary checking
    int index = 4; // Out-of-bounds index
    if (index >= 0 && index < length) {
        printf("Safe access: %d\n", *(ptr + index));
    } else {
        printf("Index %d is out of bounds, refusing to access\n", index);
    }

    return 0;
}
```

```
% cc -o lab_pointer_bounds lab_pointer_bounds.c
% ./lab_pointer_bounds
```

4.11.5.7.1 Step 1: Compile and run You should see output that walks forward and backward through the array. Notice how both directions are handled using the same pointer, just with different arithmetic.

4.11.5.7.2 Step 2: Try breaking the rule

Now, change the boundary check:

```
printf("Unsafe access: %d\n", *(ptr + 4));
```

Compile and run again. On your system, it might print a random number, or it might crash. This is **undefined behaviour** in action. The program stepped outside the array's safe memory and read garbage. On FreeBSD in user space this might only crash your program, but in kernel space the same mistake could crash the whole system.

4.11.5.7.3 Step 3: Think like a kernel developer

Kernel code often passes around buffers and pointers, but the kernel does not automatically check array limits for you. Safe coding habits like the check we used earlier, are critical:

```
if (index >= 0 && index < length) { ... }
```

Always validate that you are within valid bounds before dereferencing a pointer.

Key Takeaways from This Lab

- Pointer arithmetic lets you move forward and backwards through arrays.
- Arrays do not carry their length with them; you must track it yourself.
- Accessing memory beyond array boundaries is undefined behaviour.
- In FreeBSD kernel code, these mistakes can lead to panics or vulnerabilities, so always include boundary checks.

4.11.5.8 Challenge Questions

1. **Forward and backwards in one pass:** Write a function `void walk_both(const int *base, size_t n)` that prints pairs `(base[i], base[n-1-i])` using only pointer arithmetic, no array indexing. Stop when the pointers meet or cross.
2. **Bounds checked accessor:** Implement `int get_at(const int *base, size_t n, size_t i, int *out)` that returns 0 on success and a non-zero error code if `i` is out of range. Use only pointer arithmetic to read the value.
3. **Find first match:** Write `int *find_first(int *base, size_t n, int target)` that returns a pointer to the first occurrence or NULL if not found. Walk using a moving pointer from `base` to `base + n`.
4. **Reverse in place:** Create `void reverse_in_place(int *base, size_t n)` that swaps elements from the ends toward the middle using two pointers. Do not use indexing.
5. **Safe slice print:** Write `void print_slice(const int *base, size_t n, size_t start, size_t count)` that prints at most `count` elements beginning at `start`, but never steps beyond `n`.
6. **Off by one detector:** Introduce an off-by-one bug in a loop, then add a runtime check that detects it. Fix the loop and confirm the check remains silent.
7. **Stride walk:** Treat the array as records of `stride` ints. Write `void walk_stride(const int *base, size_t n, size_t stride)` that visits only the first element of each record.

8. **Pointer difference:** Given two pointers `a` and `b` into the same array, compute the element distance using `ptrdiff_t`. Verify that `a + distance == b`.
9. **Guarded dereference:** Implement `int try_deref(const int *p, const int *begin, const int *end, int *out)` that only dereferences if `p` lies within `[begin, end)`.
10. **Refactor into functions:** Rewrite the lab so that walking, printing, and boundary checking are separate functions that all use pointer arithmetic.

4.11.5.9 Wrapping Up

Pointer arithmetic gives you a new way to move through memory. It allows you to scan arrays efficiently, traverse structures, and interact with hardware buffers. But with this power comes the danger of stepping outside the safe zone. In user programs, that might crash only your program. In kernel code, a mistake could crash the entire system or open a security hole.

As you continue your work, keep the mental image of walking along a row of boxes. Step carefully, never wander off the edge, and always count how many boxes you really have. With this habit, you will be ready for the next topic: using pointers to access not just plain values, but entire **structures**, the building blocks of more complex data in FreeBSD.

4.11.6 Pointers to Structs

In C programming, especially when writing device drivers or working inside the FreeBSD kernel, you will constantly encounter **structs**. A struct is a way of grouping several related variables together under one name. These variables, called *fields*, can be of different types, and together they represent a more complex entity.

Because structs are often large and frequently shared between parts of the kernel, we usually interact with them through **pointers**. Understanding how to work with pointers to structs is, therefore, a fundamental skill for reading kernel code and writing drivers of your own.

Let's build this step by step.

4.11.6.1 What Is a Struct?

A struct groups multiple variables into a single unit. For example:

```
struct Point {  
    int x;  
    int y;  
};
```

This defines a new type called `struct Point`, which has two integer fields: `x` and `y`.

We can create a variable of this type and assign values to its fields:

```
struct Point p1;  
p1.x = 10;  
p1.y = 20;
```

At this point, `p1` is a concrete object in memory, holding two integers side by side.

4.11.6.2 Introducing Pointers to Structs

Just like we can have a pointer to an integer or a pointer to a character, we can also have a pointer to a struct:

```
struct Point *ptr;
```

If we already have a variable of type `struct Point`, we can store its address in the pointer:

```
ptr = &p1;
```

Now `ptr` holds the address of `p1`. To reach the fields of the struct through this pointer, C provides two notations.

4.11.6.3 Accessing Struct Fields Through Pointers

Suppose we have:

```
struct Point p1 = {10, 20};  
struct Point *ptr = &p1;
```

There are two ways to access the fields via the pointer:

```
// Method 1: Explicit dereference  
printf("x = %d\n", (*ptr).x);  
  
// Method 2: Arrow operator  
printf("y = %d\n", ptr->y);
```

Both are correct, but the arrow operator (`->`) is much cleaner and is the style you will see everywhere in FreeBSD kernel code.

So:

```
ptr->x
```

Means the same as:

```
(*ptr).x
```

But it is easier to read and write.

4.11.6.4 Why Pointers Are Preferred

Passing an entire struct around by value can be expensive, since C would need to copy every field each time. Instead, the kernel almost always passes around *pointers to structs*. This way, only an address (a small integer) is passed, and different parts of the system can all work with the same underlying object.

This is particularly important in device drivers, where structs often represent significant and complex entities such as devices, processes, or threads.

4.11.6.5 A Real FreeBSD Example

Let's look at a real snippet from the FreeBSD source tree.

In `sys/kern/tty_info.c`, the `tty_info()` function works with a struct `proc`, which represents a process. The relevant fragment (inside `tty_info()` in the FreeBSD 14.3 source) is:

```
struct proc *p, *ppick;

p = NULL;

// Walk through all members of the foreground process group
LIST_FOREACH(ppick, &tp->t_pgrp->pg_members, p_pglst)
    // Use proc_compare() to decide if this candidate is "better"
    if (proc_compare(p, ppick))
        p = ppick;

// Later in the function, access the chosen process
pid = p->p_pid; // get the process ID
strncpy(comm, p->p_comm, sizeof comm); // copy the process name into a local buffer
```

Here's what happens, step by step:

- `p` and `ppick` are pointers to struct `proc`. The code initialises `p` to `NULL`, then iterates over the foreground process group with `LIST_FOREACH`.
- On each step, it calls `proc_compare()` to decide whether the current candidate `ppick` is “better” than the one already chosen; if so, it updates `p`. Later, it reads fields from the selected process via `p->p_pid` and `p->p_comm`.

This is a typical kernel pattern: **select a struct instance via pointer traversal, then access its fields through `->`**.

Note: `proc_compare()` encapsulates the selection logic: it prefers runnable processes, then the one with higher recent CPU usage, de-prioritises zombies, and breaks ties by choosing the higher PID.

4.11.6.6 A Quick Bridge: what `LIST_FOREACH` does

Beginners often see `LIST_FOREACH(...)` and wonder what magic is happening. There's no magic, it's a macro that walks a **singly-linked list**. Its common BSD style is:

```
LIST_FOREACH(item, &head->list_field, link_field) {  
    /* use item->field ... */  
}
```

- `item` is the loop variable that points to each element.
- `&head->list_field` is the list head you're iterating.
- `link_field` is the name of the link that chains elements together (the "next" pointer field in each node).

In our snippet, `ppick` is the loop variable, `&tp->t_pgrp->pg_members` is the list of processes in the foreground process group, and `p_pglis` is the link field inside each `struct proc`. Each iteration points `ppick` at the next process, allowing the code to compare and ultimately select the one stored in `p`.

This small macro hides the pointer-chasing details so your code reads like: "for each process in this process group, consider it as a candidate."

4.11.6.7 A Minimal User-Space Example

Here's a simple program you can compile and run yourself to get hands-on practice:

```
#include <stdio.h>  
#include <string.h>  
  
// Define a simple struct  
struct Device {  
    int id;  
    char name[20];  
};  
  
int main() {  
    struct Device dev1 = {42, "tty0"};  
    struct Device *dev_ptr = &dev1;  
  
    // Access fields through the pointer  
    printf("Device ID: %d\n", dev_ptr->id);  
    printf("Device Name: %s\n", dev_ptr->name);  
  
    // Change values through the pointer  
    dev_ptr->id = 43;  
    strcpy(dev_ptr->name, "ttyS1");  
  
    // Show updated values  
    printf("Updated Device ID: %d\n", dev_ptr->id);  
    printf("Updated Device Name: %s\n", dev_ptr->name);  
}
```

```
    return 0;
}
```

This mirrors what happens in the kernel. We define a struct, create an instance of it, then use a pointer to access and modify its fields.

4.11.6.8 Common Beginner Pitfalls with Struct Pointers

Working with pointers to structs is straightforward once you get used to it, but beginners often fall into the same traps. Let's look at the most frequent mistakes and how to avoid them.

1. Forgetting to Initialise the Pointer A pointer that hasn't been given a value points to "somewhere" in memory, which usually means garbage. Accessing it causes undefined behaviour, often leading to crashes.

```
struct Device *d; // Uninitialised, points to who-knows-where
d->id = 42;       // Undefined behaviour
```

How to avoid it: Always initialise your pointer, either to NULL or to the address of a real struct.

```
struct Device dev1;
struct Device *d = &dev1; // Safe: d points to dev1
```

2. Confusing . and -> Remember: use the dot (.) to access a field when you have a real struct variable, and use the arrow (->) when you have a pointer to a struct. Mixing them up is a common beginner error.

```
struct Point p1 = {1, 2};
struct Point *ptr = &p1;

printf("%d\n", p1.x); // Dot for variables
printf("%d\n", ptr->x); // Arrow for pointers
printf("%d\n", ptr.x); // Error: ptr is not a struct
```

How to avoid it: Ask yourself: "Am I working with a pointer or the struct itself?" That tells you which operator to use.

3. Dereferencing NULL Pointers If a pointer is set to NULL (which is common as an initial or error state), dereferencing it will immediately crash your program.

```
struct Device *d = NULL;
printf("%d\n", d->id); // Crash: d is NULL
```

How to avoid it: Always check pointers before dereferencing:

```
if (d != NULL) {  
    printf("%d\n", d->id);  
}
```

In kernel code, this check is especially important. Dereferencing a NULL pointer inside the kernel can bring the whole system down.

4. Using a Pointer After the Struct Has Gone Out of Scope Pointers don't "own" the struct they point to. If the struct disappears (for example, because it was a local variable in a function that has returned), the pointer becomes invalid, a *dangling pointer*.

```
struct Device *make_device(void) {  
    struct Device d = {99, "tty0"};  
    return &d; // Dangerous: d disappears after function returns  
}
```

How to avoid it: Never return the address of a local variable. Allocate dynamically (with `malloc` in user programs or `malloc(9)` in the kernel) if you need a struct to outlive the function that creates it.

5. Assuming a Struct Is Small Enough to Copy Around In user programs, you can sometimes get away with passing structs by value. But in kernel code, structs often represent large, complex objects, sometimes with embedded lists or pointers of their own. Copying them accidentally can cause subtle and serious bugs.

```
struct Device dev1 = {1, "tty0"};  
struct Device dev2 = dev1; // Copies all fields, not a shared reference
```

How to avoid it: Pass pointers to structs instead of copying them, unless you are certain that a shallow copy is intended.

Main Takeaway: The most common mistakes with struct pointers come from forgetting what a pointer really is: just an address. Always initialise pointers, distinguish carefully between `.` and `->`, check for NULL, and be mindful of scope and lifetime. In kernel development, a single misstep with a struct pointer can destabilise the entire system, so adopting safe habits early will serve you well.

4.11.6.9 Mini Hands-On Lab: Struct Pointer Pitfalls (No `malloc` yet)

This lab reproduces common mistakes with pointers to structs and then shows safe, beginner-friendly alternatives using only stack variables and function "out-parameters".

Create a file `lab_struct_pointer_pitfalls_nomalloc.c`:

```
/*  
 * lab_struct_pointer_pitfalls_nomalloc.c  
 *  
 * Classic pitfalls with pointers to structs, rewritten to avoid malloc.
```



```

* Build and run each case and read the console output as you go.
*
* NOTE: This file intentionally contains ONE warning for make_device_bad()
*       to demonstrate what the compiler catches. This is expected!
*/

#include <stdio.h>
#include <string.h>

struct Device {
    int id;
    char name[16];
};

/* -----
 * Case 1: Uninitialised pointer (UB) vs. correctly initialised
 * ----- */
static void case_uninitialised_pointer(void) {
    printf("\n[Case 1] Uninitialised pointer vs initialised\n");

    /* Wrong: d has no valid target. Dereferencing is undefined. */
    /* struct Device *d; */
    /* d->id = 42; // Do NOT do this */

    /* Right: point to a real object or keep NULL until assigned. */
    struct Device dev = { 1, "tty0" };
    struct Device *ok = &dev;
    ok->id = 2;
    strcpy(ok->name, "ttyS0");
    printf(" ok->id=%d ok->name=%s\n", ok->id, ok->name);

    struct Device *maybe = NULL;
    if (maybe == NULL) {
        printf(" maybe is NULL, avoiding dereference\n");
    }
}

/* -----
 * Case 2: Dot vs arrow confusion
 * ----- */
static void case_dot_vs_arrow(void) {
    printf("\n[Case 2] Dot vs arrow\n");

    struct Device dev = { 7, "console0" };

```

```

    struct Device *p = &dev;

    /* Correct usage */
    printf(" dev.id=%d dev.name=%s\n", dev.id, dev.name);    /* variable: use . */
    printf(" p->id=%d p->name=%s\n", p->id, p->name);        /* pointer: use -> */

    /* Uncomment to observe the compiler error for teaching purposes */
    /* printf("%d\n", p.id); */ /* p is a pointer, not a struct */
}

/* -----
 * Case 3: NULL pointer dereference vs guarded access
 * ----- */
static void case_null_deref(void) {
    printf("\n[Case 3] NULL dereference guard\n");

    struct Device *p = NULL;

    /* Wrong: would crash */
    /* printf("%d\n", p->id); */

    /* Right: guard before dereferencing if pointer may be NULL */
    if (p != NULL) {
        printf(" id=%d\n", p->id);
    } else {
        printf(" p is NULL, skipping access\n");
    }
}

/* -----
 * Case 4: Dangling pointer (returning address of a local)
 * and two safe alternatives WITHOUT malloc
 * ----- */

/* Dangerous factory: returns address of a local variable (dangling) */
static struct Device *make_device_bad(void) {
    struct Device d = { 99, "bad-local" };
    return &d; /* The address becomes invalid when the function returns */
}

/* Safe alternative A: initialiser that writes into caller-provided struct */
static void init_device(struct Device *out, int id, const char *name) {
    if (out == NULL) return;
    out->id = id;

```

```

    strncpy(out->name, name, sizeof(out->name) - 1);
    out->name[sizeof(out->name) - 1] = '\0';
}

/* Safe alternative B: return-by-value (fine for small, plain structs) */
static struct Device make_device_value(int id, const char *name) {
    struct Device d;
    d.id = id;
    strncpy(d.name, name, sizeof(d.name) - 1);
    d.name[sizeof(d.name) - 1] = '\0';
    return d; /* Returned by value: the caller receives its own copy */
}

static void case_dangling_and_safe_alternatives(void) {
    printf("\n[Case 4] Dangling vs safe init (no malloc)\n");

    /* Wrong: pointer becomes dangling immediately */
    struct Device *bad = make_device_bad();
    (void)bad; /* Do not dereference; it is invalid. */
    printf(" bad points to invalid memory; we will not dereference it\n");

    /* Safe A: caller owns storage; callee fills it via pointer */
    struct Device owned_a;
    init_device(&owned_a, 123, "owned-A");
    printf(" owned_a.id=%d owned_a.name=%s\n", owned_a.id, owned_a.name);

    /* Safe B: small plain struct returned by value */
    struct Device owned_b = make_device_value(124, "owned-B");
    printf(" owned_b.id=%d owned_b.name=%s\n", owned_b.id, owned_b.name);
}

/* -----
 * Case 5: Accidental struct copy vs pointer sharing
 * ----- */
static void case_copy_vs_share(void) {
    printf("\n[Case 5] Copy vs share\n");

    struct Device a = { 1, "tty0" };

    /* Accidental copy: b is a separate struct */
    struct Device b = a;
    strcpy(b.name, "tty1");
    printf(" after copy+edit: a.name=%s, b.name=%s\n", a.name, b.name);
}

```

```

    /* Intentional sharing via pointer */
    struct Device *pa = &a;
    strcpy(pa->name, "tty2");
    printf(" after shared edit: a.name=%s (via pa)\n", a.name);
}

/* Bonus: safe initialisation pattern via out-parameter */
static void case_safe_init_pattern(void) {
    printf("\n[Bonus] Safe initialisation via pointer\n");
    struct Device dev;
    init_device(&dev, 55, "control0");
    printf(" dev.id=%d dev.name=%s\n", dev.id, dev.name);
}

int main(void) {
    case_uninitialised_pointer();
    case_dot_vs_arrow();
    case_null_deref();
    case_dangling_and_safe_alternatives();
    case_copy_vs_share();
    case_safe_init_pattern();
    return 0;
}

```

Build and run:

```

% cc -Wall -Wextra -o lab_struct_pointer_pitfalls_nomalloc
  ↪ lab_struct_pointer_pitfalls_nomalloc.c
% ./lab_struct_pointer_pitfalls_nomalloc

```

What to notice:

1. **Uninitialised versus initialised** You never dereference an uninitialised pointer. Point it to a real object or keep it as NULL until you have one.
2. **Dot versus arrow** Use `.` with a struct variable, `->` with a pointer. If you uncomment the `p.id` line, the compiler will flag it.
3. **NULL dereference** Always guard against NULL when there is any chance the pointer might be unset.
4. **Dangling pointer without malloc** Returning the address of a local variable is unsafe because the local goes out of scope. Two safe options that require no heap:
 - Let the caller **provide the storage** and pass a pointer to be filled in.
 - **Return a small, plain struct by value** when a copy is intended.
5. **Copy vs share** Copying a struct makes a separate object; editing one does not alter the other. Using a pointer means both names refer to the same object.

4.11.6.10 Why this matters for driver code

Kernel code passes pointers to structs everywhere. The habits you just practiced are foundational: initialise pointers, choose the correct operator, guard against NULL, avoid dangling pointers by respecting scope, and be deliberate about copying versus sharing. These patterns keep kernel code safe and predictable, long before you ever need dynamic allocation.

4.11.6.11 Challenge Questions: Pointers to Structs

Try these exercises to make sure you really understand how struct pointers work. Write small C programs for each one and experiment with the output.

1. **Dot vs Arrow** Write a program that creates a struct `Point` and a pointer to it. Print the fields twice: once using the dot operator (`.`) and once using the arrow operator (`->`). Explain why one works with the variable and the other with the pointer.
2. **Struct Initialiser Function** Write a function `void init_point(struct Point *p, int x, int y)` that fills in a struct given a pointer. Call it from main with a local variable and print the result.
3. **Returning by Value vs Returning a Pointer** Write two functions:
 - `struct Point make_point_value(int x, int y)` that returns a struct by value.
 - `struct Point *make_point_pointer(int x, int y)` that (wrongly) returns a pointer to a local struct. What happens if you use the second function? Why is it dangerous?
4. **Safe NULL Handling** Modify the program so that a pointer may be set to NULL. Write a function `print_point(const struct Point *p)` that safely prints "(null)" if `p` is NULL instead of crashing.
5. **Copy vs Share** Create two structs: one by copying another (`struct Point b = a;`) and one by sharing via pointer (`struct Point *pb = &a;`). Change the values in each and print both. What differences do you observe?
6. **Mini Linked List** Define a simple struct:

```
struct Node {  
    int value;  
    struct Node *next;  
};
```

Manually create three nodes and chain them together (`n1 -> n2 -> n3`). Use a pointer to walk through the list and print the values. This mimics what `LIST_FOREACH` does in the kernel.

4.11.6.12 Wrapping Up

Pointers to structs are one of the most important idioms in kernel programming. They allow you to work with complex objects efficiently, without copying large blocks of memory, and they provide the foundation for navigating linked lists and device tables.

You've now seen how:

- Structs group related fields into a single object.
- Pointers to structs let you access and modify those fields efficiently.
- The arrow operator (->) is the preferred way to reach struct fields through a pointer.
- Real kernel code relies heavily on struct pointers to represent processes, threads, and devices.

With the challenge questions, you can now test yourself and confirm you really understand how struct pointers behave in C.

The natural next step is to see what happens when we combine these ideas with **arrays**. Arrays of pointers and pointer arrays appear everywhere in kernel code, from device tables to argument lists.

Let's continue and learn about the next topic: **Arrays of Pointers and Pointer Arrays**.

4.11.7 Arrays of Pointers and Pointer Arrays

This is one of those topics that many C beginners find tricky: the difference between an **array of pointers** and a **pointer to an array**. The declarations look confusingly similar, but they describe very different things. The difference is not just academic. Arrays of pointers are everywhere in FreeBSD code, while pointers to arrays are less common but still important to understand because they appear in contexts where hardware requires contiguous memory blocks.

We will first look at arrays of pointers, then at pointers to arrays, and then connect them to real FreeBSD examples and driver development.

4.11.7.1 Array of Pointers

An array of pointers is simply an array where each element is itself a pointer. Instead of holding values directly, the array holds addresses pointing to values stored elsewhere.

4.11.7.1.1 Example: Array of Strings

```
#include <stdio.h>

int main() {
    // Array of 3 pointers to const char (i.e., strings)
    const char *messages[3] = {
        "Welcome",
        "to",
        "FreeBSD"
    };

    for (int i = 0; i < 3; i++) {
        printf("Message %d: %s\n", i, messages[i]);
    }

    return 0;
}
```

Here, `messages` is an array of three pointers. Each element, such as `messages[0]`, holds the address of a string literal. When passed to `printf`, it prints the string.

This is exactly the same structure as the `argv` parameter to `main()`: it is just an array of pointers to characters.

4.11.7.1.2 Real FreeBSD Example: Locale Names Looking into FreeBSD 14.3 source code, more specifically in `bin/sh/var.c`, the `locale_names` array:

```
static const char *const locale_names[7] = {
    "LANG", "LC_ALL", "LC_COLLATE", "LC_CTYPE",
    "LC_MESSAGES", "LC_MONETARY", "LC_NUMERIC",
};
```

This is an **array of constant character pointers**. Each element points to a string literal with the name of a locale category. The shell uses this array to check or set environment variables consistently. This is a compact, idiomatic way to store a table of names.

4.11.7.1.3 Real FreeBSD Example: SNMP Transport Names Looking into FreeBSD 14.3 source code, more specifically, in `contrib/bsnmp/lib/snmpclient.c`, the `trans_list` array:

```
static const char *const trans_list[] = {
    "udp", "tcp", "local", NULL
};
```

This is another array of string pointers, terminated by `NULL`. The SNMP client library uses this list to recognise valid transport names. The use of `NULL` as a terminator is a very common C idiom.

Note: Arrays of pointers are common in FreeBSD because they allow flexible, dynamic lookup tables without copying large amounts of data. Instead of storing the strings inline, the array stores pointers to them.

4.11.7.2 Pointer to an Array

A pointer to an array is very different from an array of pointers. Instead of pointing to scattered objects, it points to one single, contiguous array block. The syntax can look intimidating, but the underlying idea is simple: the pointer represents the entire array as a unit.

4.11.7.2.1 Example: Pointer to an Array of Integers

```
#include <stdio.h>

int main() {
    int numbers[5] = {1, 2, 3, 4, 5};
```

```
// Pointer to an array of 5 integers
int (*p)[5] = &numbers;

printf("Third number: %d\n", (*p)[2]);

return 0;
}
```

Breaking it down:

- `int (*p)[5];` declares `p` as a pointer to an array of 5 integers.
- `p = &numbers;` makes `p` point to the whole `numbers` array.
- `(*p)[2]` first dereferences the pointer (giving us the array) and then indexes into it.

4.11.7.2.2 Another Example with Structures

```
#include <stdio.h>

#define SIZE 4

struct Point {
    int x, y;
};

int main(void) {
    struct Point pts[SIZE] = {
        {0, 0}, {1, 2}, {2, 4}, {3, 6}
    };

    struct Point (*parray)[SIZE] = &pts;

    printf("Third element: x=%d, y=%d\n",
        (*parray)[2].x, (*parray)[2].y);

    // Modify via the pointer
    (*parray)[2].x = 42;
    (*parray)[2].y = 84;

    printf("After modification: x=%d, y=%d\n",
        pts[2].x, pts[2].y);

    return 0;
}
```


Here, `parray` points to the entire array of four `struct Point`. Accessing through it is equivalent to accessing the original array directly, but it emphasises that the pointer represents the array as a single unit.

You have now seen two different examples of pointers to arrays. Both show how a single pointer can name a whole, contiguous region of memory. The natural question is how often this appears in real FreeBSD code.

4.11.7.2.3 Why You Rarely See This in FreeBSD Literal `T (*p)[N]` declarations are uncommon in the kernel source. FreeBSD developers usually represent fixed-size blocks in one of two ways:

- Wrap the array inside a `struct`, which keeps size and type information together and leaves room for metadata.
- Pass a base pointer along with an explicit length, especially for buffers and I/O regions.

This style makes the code more transparent, easier to maintain, and integrates better with kernel subsystems. The random subsystem is a good example, where structures carry fixed arrays that are treated as single units by the code paths that process them. For background on how drivers and subsystems feed entropy into the kernel, see the `random_harvest(9)` manual page.

4.11.7.2.4 Real FreeBSD Example: Struct with a Fixed-Size Array FreeBSD 14.3 source, `sys/dev/random/random_harvestq.h`, the `HARVESTSIZE` macro and `struct harvest_event` definition:

```
#define HARVESTSIZE    2        /* Max length in words of each harvested entropy unit */

/* These are used to queue harvested packets of entropy. The entropy
 * buffer size is pretty arbitrary.
 */
struct harvest_event {
    uint32_t    he_somcounter;    /* fast counter for clock jitter */
    uint32_t    he_entropy[HARVESTSIZE]; /* some harvested entropy */
    uint8_t     he_size;          /* harvested entropy byte count */
    uint8_t     he_destination;   /* destination pool of this entropy */
    uint8_t     he_source;        /* origin of the entropy */
};
```

This is not a raw `T (*p)[N]` declaration, yet it captures the same idea in a form that is clearer and more practical for the kernel. The `struct` groups a fixed array `he_entropy[HARVESTSIZE]` with related fields. Code then passes a pointer to `struct harvest_event`, treating the entire block as one object. In `random_harvestq.c` you can see how an instance is filled and processed, including copying into `he_entropy` and setting the size and metadata fields, which reinforces that the array is handled as part of a single unit.

Even though raw pointers to arrays are rare in the tree, understanding them helps you recognise why kernel code tends to wrap arrays in structures or pair a base pointer with an explicit length. Conceptually, it is the same pattern of referring to a contiguous block as a whole.

4.11.7.3 Mini Hands-On Lab

Let's test your understanding. For each declaration, decide whether it is an **array of pointers** or a **pointer to an array**. Then explain how you would access the third element.

1. `const char *names[] = { "a", "b", "c", NULL };`
2. `int (*ring)[64];`
3. `struct foo *ops[8];`
4. `char (*line)[80];`

Check yourself:

1. Array of pointers. Third element with `names[2]`.
2. Pointer to an array of 64 ints. Use `(*ring)[2]`.
3. Array of pointers to `struct foo`. Use `ops[2]`.
4. Pointer to an array of 80 chars. Use `(*line)[2]`.

4.11.7.4 Challenge Questions

1. Why is `argv` in `main(int argc, char *argv[])` considered an array of pointers rather than a pointer to an array?
2. In kernel code, why do developers prefer to use a `struct` wrapping a fixed-size array instead of a raw pointer-to-array declaration?
3. How does using `NULL` as a terminator in arrays of pointers simplify iteration?
4. Imagine a driver that manages a ring of DMA descriptors. Would you expect this to be represented as an array of pointers or a pointer to an array? Why?
5. What could go wrong if you mistakenly treated a pointer to an array as if it were an array of pointers?

4.11.7.5 Why This Matters for Kernel and Driver Development

In FreeBSD device drivers, **arrays of pointers** appear constantly. They are used for option lists, function pointer tables, protocol name arrays, and `sysctl` handlers. This idiom saves space and allows code to iterate flexibly through lists without knowing their exact size in advance.

Pointers to arrays, while rarer, are conceptually important because they match the way hardware often works. A NIC, for example, may expect a contiguous ring buffer of descriptors. In practice FreeBSD developers usually hide the raw pointer-to-array inside a `struct` that describes the ring, but the underlying idea is identical: the driver is passing around “a single block of fixed-size elements.”

Understanding both patterns is part of thinking like a systems programmer. It ensures you will not confuse two declarations that look similar but behave differently, which prevents subtle and painful bugs.

4.11.7.6 Wrapping Up

By now you can clearly see the difference between an array of pointers and a pointer to an array. You have also seen why this distinction matters when reading or writing real code. Arrays of pointers give flexibility by

letting each element point to different objects, while a pointer to an array treats a whole block of memory as a single unit.

With this foundation in place, we are ready to take the next step: moving from fixed arrays to dynamically allocated memory. In the following section on **Dynamic Memory Allocation**, you will learn how to use functions such as `malloc`, `calloc`, `realloc`, and `free` to create arrays at runtime. We will connect this to pointers by showing how to allocate each element separately, how to request one contiguous block when you need a pointer to an array, and how to clean up properly if something goes wrong. This transition from static to dynamic memory is essential for real systems programming and will prepare you for the way memory is managed inside the FreeBSD kernel.

4.12 Dynamic Memory Allocation

So far, most of the memory we used in examples was **fixed-size**: arrays with a known length or structures allocated on the stack. But when writing system-level code like FreeBSD device drivers, you often don't know in advance how much memory you'll need. Maybe a device reports the number of buffers only after probing, or the amount of data depends on user input. That's when **dynamic memory allocation** comes into play.

Dynamic allocation allows your code to **ask the system for memory while it's running**, and to give it back when it's no longer needed. This flexibility is essential for drivers, where hardware and workload conditions can change at runtime.

4.12.1 User Space vs Kernel Space

In user space, you've probably seen functions like:

- `malloc(size)` - allocates a block of memory.
- `calloc(count, size)` - allocates and zeroes a block.
- `free(ptr)` - releases previously allocated memory.

Example:

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int *nums;
    int count = 5;

    nums = malloc(count * sizeof(int));
    if (nums == NULL) {
        printf("Memory allocation failed!\n");
        return 1;
    }

    for (int i = 0; i < count; i++) {
```

```

    nums[i] = i * 10;
    printf("nums[%d] = %d\n", i, nums[i]);
}

free(nums); // Always release what you allocated
return 0;
}

```

In user space, memory comes from the **heap**, managed by the C runtime and the operating system.

But inside the **FreeBSD kernel**, we cannot use `<stdlib.h>`'s `malloc()` or `free()`. The kernel has its own allocator, designed with stricter rules and better tracking. The kernel API is documented in `malloc(9)`.

4.12.2 Visualising Memory in User Space

```

+-----+
|          STACK (grows down)          |
|  - Local variables                   |
|  - Function call frames              |
|  - Fixed-size arrays                 |
+-----+
|          ^                           |
|          |                           |
|          HEAP (grows up)             |
|  - malloc()/calloc()/free()          |
|  - Dynamic structures and arrays     |
+-----+
|          DATA SEGMENT               |
|  - Globals                          |
|  - static variables                  |
|  - .data (initialized) / .bss (zero-initialized) |
+-----+
|          CODE / TEXT SEGMENT         |
|  - Program instructions              |
+-----+

```

Note: Stack and heap grow toward each other at runtime. Fixed-size data lives on the stack or in the data segment, while dynamic allocations come from the heap.

4.12.3 Kernel-Space Allocation with `malloc(9)`

To allocate memory in the FreeBSD kernel you use:

```
#include <sys/malloc.h>
```

```
void *malloc(size_t size, struct malloc_type *type, int flags);
void free(void *addr, struct malloc_type *type);
```

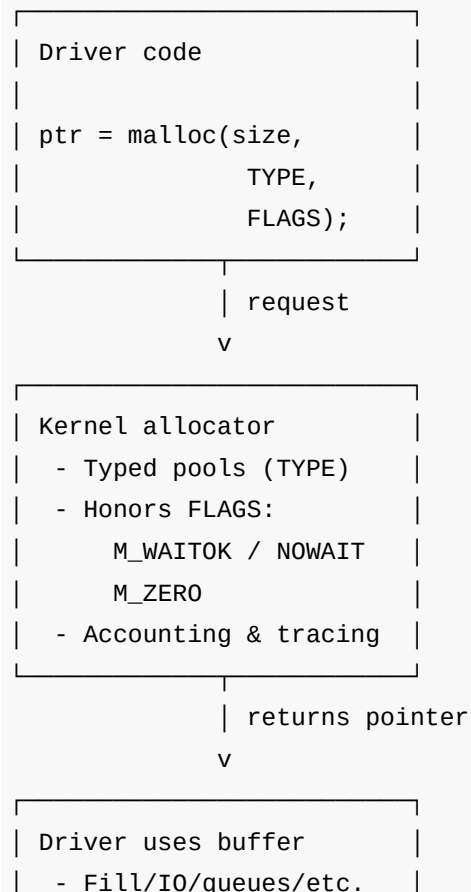
Example from the kernel:

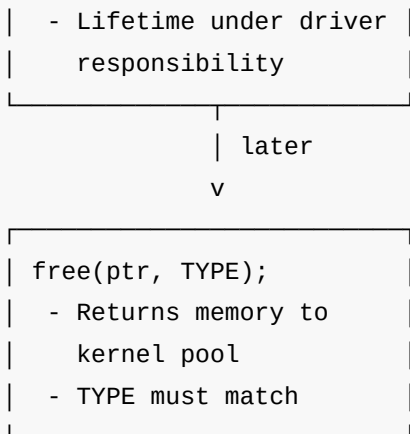
```
char *buf = malloc(1024, M_TEMP, M_WAITOK | M_ZERO);
/* ... use buf ... */
free(buf, M_TEMP);
```

Breaking it down:

- 1024 → the number of bytes.
- M_TEMP → memory type tag (explained below).
- M_WAITOK → wait if memory is temporarily unavailable.
- M_ZERO → ensure the block is zeroed.
- free(buf, M_TEMP) → release the memory.

4.12.4 Kernel malloc(9) Workflow





Note: In the kernel, every allocation is **typed** and controlled by flags. Pair every `malloc(9)` with a `free(9)` on all code paths, including errors.

4.12.5 Memory Types and Flags

One unique aspect of FreeBSD's kernel allocator is the **type system**: every allocation must be tagged. This makes debugging and leak tracking easier.

Some common types:

- `M_TEMP` - temporary allocations.
- `M_DEVBUF` - buffers for device drivers.
- `M_TTY` - terminal subsystem memory.

Common flags:

- `M_WAITOK` - sleep until memory is available.
- `M_NOWAIT` - return immediately if memory cannot be allocated.
- `M_ZERO` - zero out memory before returning.

This explicit style encourages safe, predictable memory usage in critical kernel code.

4.12.6 Hands-On Lab 1: Allocating and Freeing a Buffer

In this exercise, we'll create a simple kernel module that allocates memory when loaded and frees it when unloaded.

my_malloc_module.c

```

#include <sys/param.h>
#include <sys/module.h>
#include <sys/kernel.h>
#include <sys/malloc.h>
#include <sys/systm.h>
  
```

```
static char *buffer;
#define BUFFER_SIZE 128

MALLOC_DEFINE(M_MYBUF, "my_malloc_buffer", "Buffer for malloc module");

static int
load_handler(module_t mod, int event, void *arg)
{
    switch (event) {
        case MOD_LOAD:
            buffer = malloc(BUFFER_SIZE, M_MYBUF, M_WAITOK | M_ZERO);
            if (buffer == NULL)
                return (ENOMEM);

            snprintf(buffer, BUFFER_SIZE, "Hello from kernel space!\n");
            printf("my_malloc_module: %s", buffer);
            return (0);

        case MOD_UNLOAD:
            if (buffer != NULL) {
                free(buffer, M_MYBUF);
                printf("my_malloc_module: Memory freed\n");
            }
            return (0);
        default:
            return (EOPNOTSUPP);
    }
}

static moduledata_t my_malloc_mod = {
    "my_malloc_module", load_handler, NULL
};

DECLARE_MODULE(my_malloc_module, my_malloc_mod, SI_SUB_DRIVERS, SI_ORDER_MIDDLE);
```

What to do:

1. Build with make.
2. Load with `kldload ./my_malloc_module.ko`.
3. Check `dmesg` for the message.
4. Unload with `kldunload my_malloc_module`.

You'll see how memory is reserved and later freed.

When you load and unload the module, you will see the allocation and freeing in action.

4.12.7 Hands-On Lab 2: Allocating an Array of Structures

Now let's extend the idea by creating an array of structs dynamically.

```
struct my_entry {
    int id;
    char name[32];
};

MALLOC_DEFINE(M_MYSTRUCT, "my_struct_array", "Array of my_entry");

static struct my_entry *entries;
#define ENTRY_COUNT 5
```

On load:

- Allocate memory for five entries.
- Initialize each one with an ID and name.
- Print them out.

On unload:

- Free the memory.

This exercise mirrors what real drivers do when they keep track of device states, DMA buffers, or I/O queues.

4.12.8 Real FreeBSD Example: Building the corefile path in kern_sig.c

Let's look at a real example from FreeBSD's source code: `/usr/src/sys/kern/kern_sig.c`. This file is the heart of the kernel's signal-handling machinery, and tucked inside it are the two routines that together build and write a process core dump when a program crashes. The helper `corefile_open()` constructs the core-file path in a temporary kernel buffer, and `coredump()` drives the whole operation, from policy checks to the final cleanup. We will walk through both, because together they illustrate almost every `malloc(9)` habit you will need when writing a driver. I've added additional comments to the example code below to make it easier for you to understand what happens at each step.

Reading this example. The two listings below are an abbreviated view of the real `corefile_open()` and `coredump()` functions in `/usr/src/sys/kern/kern_sig.c`. We have kept the signatures, the control-flow spine, and the allocation and free sites intact, but we have replaced some blocks of defensive work with `/* ... omitted for brevity ... */` or `/* ... policy checks ... */` comments so the `malloc(9)` and `free(9)` discipline stays in the foreground. Every symbol the listing names is real and can be found with a symbol search; the real file is larger. We use this convention again in Chapters 13, 21, and 22 whenever a listing abbreviates a real FreeBSD function for teaching purposes.

```
/*
 * corefile_open(...) builds the final core file path into a temporary
```



```

* kernel buffer named `name`. The buffer must be large enough to hold
* a path, so MAXPATHLEN is used. The buffer is returned to the caller
* through *namep; the caller then becomes responsible for freeing it.
*/
static int
corefile_open(const char *comm, uid_t uid, pid_t pid, struct thread *td,
              int compress, int signum, struct vnode **vpp, char **namep)
{
    struct sbuf sb;
    struct nameidata nd;
    const char *format;
    char *hostname, *name;
    int cmode, error, flags, i, indexpos, indexlen, oflags, ncores;

    hostname = NULL;
    format = corefilename;

    /*
    * Allocate a zeroed path buffer from the kernel allocator.
    * - size: MAXPATHLEN bytes (fits a full path)
    * - M_TEMP: a temporary allocation type tag (helps tracking/debug)
    * - M_WAITOK: ok to sleep if memory is briefly unavailable
    * - M_ZERO: return the buffer zeroed (no stale data)
    */
    name = malloc(MAXPATHLEN, M_TEMP, M_WAITOK | M_ZERO); /* <-- allocate */
    indexlen = 0;
    indexpos = -1;
    ncores = num_cores;

    /*
    * Initialize an sbuf that writes directly into `name`.
    * SBUF_FIXEDLEN means: do not auto-grow, error if too long.
    */
    (void)sbuf_new(&sb, name, MAXPATHLEN, SBUF_FIXEDLEN);

    /*
    * The format string (kern.corefile) may include tokens like %N, %P, %U.
    * Iterate, expand tokens, and append to `sb`. If %H (hostname) appears,
    * allocate a second small buffer for it, then free it immediately after.
    */
    /* ... formatting loop omitted for brevity ... */

    /* hostname was conditionally allocated above; free it now if used */
    free(hostname, M_TEMP); /* <-- free small temp */

```

```

/*
 * If compression is requested, append a suffix like ".gz" or ".zst".
 * If the sbuf overflowed, clean up and return ENOMEM.
 */
if (sbuf_error(&sb) != 0) {
    sbuf_delete(&sb);          /* dispose sbuf wrapper (no malloc here) */
    free(name, M_TEMP);        /* <-- free on error path */
    return (ENOMEM);
}
sbuf_finish(&sb);
sbuf_delete(&sb);

/* ... open or create the vnode for the corefile into *vpp ... */

/*
 * On success, return `name` to the caller via namep.
 * Ownership of `name` transfers to the caller, who must free it.
 */
*namep = name;
return (0);
}

```

What to notice here:

- The buffer is **typed** with `M_TEMP` to help kernel memory accounting and leak detection. This is a FreeBSD kernel convention that you'll reuse for your own drivers by defining your own `MALLOC_DEFINE` tag.
- `M_WAITOK` is chosen because this path can safely sleep; the kernel allocator will wait rather than fail spuriously. If you are in a context where sleeping is unsafe, you must use `M_NOWAIT` and handle allocation failure immediately.
- Error paths **free what they allocate** before returning. This is the habit to ingrain early: every `malloc(9)` must have a clear and reliable `free(9)` in all paths.

Now let's see where the caller cleans up:

```

/*
 * coredump(td) is the main entry point for writing a process core
 * dump. It calls corefile_open() to obtain both the vnode (vp) and
 * the dynamically built `name` path, writes the core through the
 * process ABI's coredump routine, optionally emits a devctl
 * notification, and finally frees `name`. The snippet below shows
 * another temporary allocation (for the working directory, used
 * while building the notification) and the final free(name).
 *
 * Notice the signature: coredump() takes a single struct thread *

```

```

* and reaches the process and its credentials through it. The
* corefile size limit is read from the process's resource limits
* inside the function, not passed in as a separate argument.
*/
static int
coredump(struct thread *td)
{
    struct proc *p = td->td_proc;
    struct ucred *cred = td->td_ucred;
    struct vnode *vp;
    char *name; /* corefile path returned by corefile_open() */
    char *fullpath, *freepath = NULL;
    size_t fullpathsize;
    off_t limit;
    struct sbuf *sb;
    int error, error1;

    /* ... policy checks on do_coredump, P_SUGID, P2_NOTRACE ... */
    /* ... then read RLIMIT_CORE into `limit`; bail out if it is 0 ... */

    /* Build name + open/create target vnode */
    error = corefile_open(p->p_comm, cred->cr_uid, p->p_pid, td,
        compress_user_cores, p->p_sig, &vp, &name);
    if (error != 0)
        return (error);

    /* ... write the core through p->p_sysent->sv_coredump, manage the
    * range lock on the vnode, set file attributes, etc ...
    */

    /*
    * Build a devctl notification describing the core. Start an
    * auto-growing sbuf (sbuf_new_auto() allocates its own backing
    * storage, which we release later with sbuf_delete()).
    */
    sb = sbuf_new_auto(); /* <-- sbuf allocates */
    /* ... append comm="..." core="..." to sb ... */

    /*
    * If the core path is relative, allocate a small temporary
    * buffer to fetch the current working dir, then free it once
    * we've appended it to the sbuf.
    */
    if (name[0] != '/') {

```

```

    fullpathsize = MAXPATHLEN;
    freepath = malloc(fullpathsize, M_TEMP, M_WAITOK);  /* <-- allocate temp */
    if (vn_getcwd(freepath, &fullpath, &fullpathsize) != 0) {
        free(freepath, M_TEMP);                          /* <-- free on error */
        goto out2;
    }
    /* append fullpath to sb ... */
    free(freepath, M_TEMP);                              /* <-- free on success */
    /* ... continue building notification ... */
}

out2:
    sbuf_delete(sb);                                     /* <-- release sbuf */
out:
    /*
     * Close the vnode we opened and then free the dynamically built
     * `name`. This free pairs with the malloc(MAXPATHLEN, ...) in
     * corefile_open().
     */
    error1 = vn_close(vp, FWRITE, cred, td);
    if (error == 0)
        error = error1;
    free(name, M_TEMP);                                  /* <-- final free */
    return (error);
}

```

This example highlights several important practices that apply directly to device driver development. Temporary kernel strings and small work buffers are often created with `malloc(9)` and must always be released, whether the code succeeds or fails, as shown in the careful cleanup logic of `coredump()`. The same pattern applies to other allocating helpers used nearby: `sbuf_new_auto()` quietly allocates its own backing storage, so the paired `sbuf_delete()` is just as important as any manual `free(9)`. Whenever the kernel hands you something, ask at once where you will give it back.

The choice between `M_WAITOK` and `M_NOWAIT` also depends on context: in code paths where the kernel can safely sleep, `M_WAITOK` ensures the allocation will eventually succeed, while in contexts such as interrupt handlers, where sleeping is forbidden, `M_NOWAIT` must be used and a `NULL` pointer handled immediately. Finally, keeping allocations local and freeing them as soon as their last use is complete reduces the risk of memory leaks and use-after-free errors.

The handling of the short-lived `freepath` buffer is a clear demonstration of this principle in practice.

4.12.9 Why This Matters in FreeBSD Device Drivers

In real-world drivers, memory needs are rarely predictable. A network card might advertise the number of receive descriptors only after you probe it. A storage controller could require buffers sized according to device-

specific registers. Some devices maintain tables that grow or shrink depending on the workload, such as pending I/O requests or active sessions. All of these cases require **dynamic memory allocation**.

Static arrays cannot cover such situations because they are fixed at compile time, wasting memory if oversized or failing outright if undersized. With `malloc(9)` and `free(9)`, a driver can adapt to the actual hardware and workload, allocating exactly what is needed and returning memory once it is no longer in use.

However, this flexibility comes with responsibility. Unlike in user space, memory management errors in the kernel can destabilise the entire system. A missed `free()` becomes a memory leak that weakens long-term stability. An invalid pointer access after freeing can crash the kernel instantly. Overruns and underruns can silently corrupt memory structures used by other subsystems, sometimes turning into security vulnerabilities.

This is why learning to allocate, use, and release memory correctly is one of the foundational skills for FreeBSD driver developers. Getting this right ensures that your driver not only works under normal conditions but also behaves safely under stress, making the system reliable as a whole.

4.12.9.1 Real Driver Scenarios

Here are some practical cases where dynamic memory allocation is essential in FreeBSD device drivers:

- **Network drivers:** Allocate rings of packet descriptors whose size depends on the NIC's capabilities.
- **USB drivers:** Create transfer buffers sized to the maximum packet length reported by the device.
- **PCI storage controllers:** Build command tables that expand with the number of active requests.
- **Character devices:** Manage per-open data structures that exist only while a user process holds the device open.

These examples show that dynamic allocation is not just an academic exercise: it is a daily requirement for making real drivers interact safely and efficiently with hardware.

4.12.10 Common Beginner Pitfalls

Dynamic allocation in kernel code introduces some traps that are easy to overlook:

- 1. Leaking memory on error paths** It's not enough to free memory in the "happy path." If an error occurs after you've allocated but before the function exits, forgetting to free will leak memory inside the kernel. *Tip:* Always trace every exit path and make sure each allocated block is either used or freed. Using a single cleanup label at the end of your function is a common pattern in FreeBSD.
- 2. Freeing with the wrong type** Every `malloc(9)` call is tagged with a type. Freeing with a mismatched type may confuse the kernel's memory accounting and debugging tools. *Tip:* Define a custom tag for your driver with `MALLOC_DEFINE()` and always free with that same tag.
- 3. Assuming allocation always succeeds** In user space, `malloc()` often succeeds unless the system is badly constrained. In the kernel, especially with `M_NOWAIT`, allocation can legitimately fail. *Tip:* Always check for `NULL` and handle the failure gracefully.
- 4. Choosing the wrong allocation flag** Using `M_WAITOK` in contexts that cannot sleep (like interrupt handlers) can deadlock the kernel. Using `M_NOWAIT` when sleeping is safe may force needless failure handling. *Tip:* Understand the context of your allocation and pick the correct flag.

4.12.11 Challenge Questions

1. In a driver's `detach()` routine, what can happen if you forget to free dynamically allocated buffers?
2. Why is it important that the type passed to `free(9)` matches the one used in `malloc(9)`?
3. Imagine you allocate memory with `M_NOWAIT` during an interrupt. The call returns `NULL`. What should your driver do next?
4. Why is checking every error path after a successful allocation just as important as freeing on the success path?
5. If you use `M_WAITOK` inside an interrupt filter, what dangerous condition might arise?

4.12.12 Wrapping Up

You have now seen how C's dynamic memory allocation works in user space and how FreeBSD extends this idea with its own `malloc(9)` and `free(9)` for the kernel. You learned why allocations must always be paired with cleanups, how memory types and flags guide safe allocation, and how real FreeBSD code uses these patterns every day.

Dynamic allocation gives your driver the flexibility to adapt to hardware and workload demands, but it also introduces new responsibilities. Handling every error path, choosing the right flags, and keeping allocations short-lived are the habits that separate safe kernel code from fragile code.

In the next section, we will build directly on this foundation by looking at **memory safety in kernel code**. There, you will learn techniques to protect against leaks, overflows, and use-after-free errors, making your driver not only functional but also reliable and secure.

4.13 Memory Safety in Kernel Code

When writing kernel code, especially device drivers, we are working in a privileged and unforgiving environment. There is no safety net. In user-space programming, a crash usually terminates only your process. In kernel-space, a single bug can panic or reboot the entire operating system. That is why memory safety is not optional. It is the foundation of stable and secure FreeBSD driver development.

You must constantly remember that the kernel is persistent and long-running. A memory leak will accumulate for as long as the system remains up. A buffer overflow can silently overwrite unrelated data structures and later trigger a mysterious crash. Using an uninitialized pointer can panic the system instantly.

This section introduces the most common mistakes, shows you how to avoid them, and gives you practice through real experiments, both in user space and inside a small kernel module.

4.13.1 What Can Go Wrong?

Most kernel bugs caused by beginners can be traced to unsafe memory handling. Let's list the most frequent and dangerous ones:

- **Using uninitialized pointers:** a pointer that is not set to a valid address contains garbage. Dereferencing it usually causes a panic.

- **Accessing freed memory (use-after-free):** once memory is released, it must never be touched again. Doing so corrupts memory and destabilises the kernel.
- **Memory leaks:** failing to call `free()` after `malloc()` means the memory remains reserved forever, slowly consuming kernel resources.
- **Buffer overflows:** writing beyond the end of a buffer overwrites unrelated memory. This can corrupt kernel state or introduce security vulnerabilities.
- **Off-by-one array errors:** accessing one index past the end of an array is enough to destroy adjacent kernel data.

Unlike user space, where tools like `valgrind` can sometimes save you, in kernel programming these errors can lead to instant crashes or subtle corruption that is very difficult to debug.

4.13.2 Best Practices for Safer Kernel Code

FreeBSD provides mechanisms and conventions to help developers write robust code. Follow these guidelines:

1. **Always initialise pointers.** If you do not yet have a valid memory address, set the pointer to `NULL`. This makes accidental dereferences easier to detect.

```
struct my_entry *ptr = NULL;
```

2. **Check the result of `malloc()`.** Memory allocation may fail. Never assume success.

```
ptr = malloc(sizeof(*ptr), M_MYTAG, M_NOWAIT);
if (ptr == NULL) {
    // Handle gracefully, avoid panic
}
```

3. **Free what you allocate.** Every `malloc()` must have a matching `free()`. In kernel space, leaks accumulate until reboot.

```
free(ptr, M_MYTAG);
```

4. **Avoid buffer overflows.** Use safer functions such as `strncpy()` or `snprintf()`, which take the buffer size as an argument.

```
strncpy(buffer, "FreeBSD", sizeof(buffer));
```

5. **Use `M_ZERO` to avoid garbage values.** This flag ensures allocated memory starts clean.

```
ptr = malloc(sizeof(*ptr), M_MYTAG, M_WAITOK | M_ZERO);
```

6. **Use proper allocation flags.**

- `M_WAITOK` is used when allocation can safely sleep until memory becomes available.
- `M_NOWAIT` must be used in interrupt handlers or any context where sleeping is forbidden.

4.13.3 A Real Example from FreeBSD 14.3

In the FreeBSD source tree, memory is often managed through pre-allocated buffers rather than frequent dynamic allocations. Here is a snippet from the `tty_info()` function in `sys/kern/tty_info.c`:

```
(void)sbuf_new(&sb, tp->t_prbuf, tp->t_prbufsz, SBUF_FIXEDLEN);
sbuf_set_drain(&sb, sbuf_tty_drain, tp);
```

What happens here?

- `sbuf_new()` creates a string buffer (`sb`) using an already allocated memory region (`tp->t_prbuf`).
- The size is fixed (`tp->t_prbufsz`) and protected by the `SBUF_FIXEDLEN` flag, ensuring no writes beyond the limit.
- `sbuf_set_drain()` then specifies a controlled function (`sbuf_tty_drain`) to handle buffer output.

This pattern demonstrates a safe kernel strategy: memory is allocated once during subsystem initialisation and carefully reused, rather than repeatedly allocated and freed. It reduces fragmentation, avoids allocation failures at runtime, and keeps memory usage predictable.

4.13.4 Dangerous Code to Avoid

The following snippet is **wrong** because it uses a pointer that was never given a valid address:

```
struct my_entry *ptr;    // Declared, but not initialised. 'ptr' contains garbage.

ptr->id = 5;              // Crash risk: dereferencing an uninitialised pointer
```

`ptr` does not point anywhere valid. When you try to access `ptr->id`, the kernel will likely panic because you are touching memory you do not own. In user space, this would usually be a segmentation fault. In kernel space, it can crash the whole system.

4.13.5 The Correct Pattern

Below is a safe version that allocates memory, checks that the allocation worked, uses the memory, and then releases it. The comments explain each step and why it matters in kernel code.

```
#include <sys/param.h>
#include <sys/malloc.h>

/*
 * Give your allocations a custom tag. This helps the kernel track who owns
 * the memory and makes debugging leaks much easier.
 */
MALLOC_DECLARE(M_MYTAG);           // Declare the tag (usually in a header)
```



```
MALLOC_DEFINE(M_MYTAG, "mydriver", "My driver allocations"); // Define the tag

struct my_entry {
    int id;
};

void example(void)
{
    struct my_entry *ptr = NULL; // Start with NULL to avoid using garbage

    /*
     * Allocate enough space for ONE struct my_entry.
     * We use sizeof(*ptr) so if the type of 'ptr' changes,
     * the size stays correct automatically.
     *
     * Flags:
     * - M_WAITOK: allocation is allowed to sleep until memory is available.
     *             Use this only in contexts where sleeping is safe
     *             (for example, during driver attach or module load).
     *
     * - M_ZERO:   zero-fill the memory so all fields start in a known state.
     *             This prevents accidental use of uninitialised data.
     */
    ptr = malloc(sizeof(*ptr), M_MYTAG, M_WAITOK | M_ZERO);

    if (ptr == NULL) {
        /*
         * Always check for failure, even with M_WAITOK.
         * If allocation fails, handle it gracefully: log, unwind, or return.
         */
        printf("mydriver: allocation failed\n");
        return;
    }

    /*
     * At this point 'ptr' is valid and zero-initialised.
     * It is now safe to access its fields.
     */
    ptr->id = 5;

    /*
     * ... use 'ptr' for whatever work is needed ...
     */
}
```

```
/*
 * When you are done, free the memory with the SAME tag you used to allocate.
 * Pairing malloc/free is essential in kernel code to avoid leaks
 * that accumulate for the entire uptime of the machine.
 */
free(ptr, M_MYTAG);
ptr = NULL; // Optional but helpful to prevent accidental reuse
}
```

4.13.6 Why this pattern matters

1. **Initialise pointers:** starting with `NULL` makes accidental use obvious during reviews and easier to catch in tests.
2. **Size safely:** `sizeof(*ptr)` follows the pointer's type automatically, reducing the chance of wrong sizes when refactoring.
3. **Pick the right flags:**
 - Use `M_WAITOK` when the code can sleep, such as during attach, open, or module load paths.
 - Use `M_NOWAIT` in interrupt handlers or other non-sleepable contexts, and handle `NULL` immediately.
4. **Zero on allocate:** `M_ZERO` prevents hidden state from previous allocations, which avoids surprising behaviour.
5. **Always free:** every `malloc()` must be paired with `free()` using the same tag. This is non-negotiable in kernel code.
6. **Set to `NULL` after free:** it reduces the risk of use-after-free bugs if the pointer is referenced later by mistake.

4.13.7 If your context do not allow sleep

Sometimes you are in a context where sleeping is forbidden, such as an interrupt handler. In that case use `M_NOWAIT`, check immediately for failure, and defer work if needed:

```
ptr = malloc(sizeof(*ptr), M_MYTAG, M_NOWAIT | M_ZERO);
if (ptr == NULL) {
    /* Defer the work or drop it safely; do NOT block here. */
    return;
}
```

Keeping these habits from the beginning will save you from many of the most painful kernel crashes and mid-night debugging sessions.

4.13.8 Hands-On Lab 1: Crashing with an Uninitialized Pointer

This exercise demonstrates why you must never use a pointer before giving it a valid address. We will first write a broken program that uses an uninitialised pointer, then fix it with `malloc()`.

4.13.8.1 Broken Version: lab1_crash.c

```
#include <stdio.h>

struct data {
    int value;
};

int main(void) {
    struct data *ptr; // Declared but not initialised

    // At this point, 'ptr' points to some random location in memory.
    // Trying to use it will cause undefined behaviour.
    ptr->value = 42; // Segmentation fault very likely here

    printf("Value: %d\n", ptr->value);

    return 0;
}
```

4.13.8.2 What to Expect

- This program compiles without warnings.
- When you run it, it will almost certainly crash with a segmentation fault.
- The crash happens because `ptr` does not point to valid memory, yet we try to write to `ptr->value`.
- In the kernel, this same mistake would likely panic the entire system.

4.13.8.3 What is wrong here?

```
STACK (main)
+-----+
| ptr : ??? (uninitialised) | -> not a real address you own
+-----+

HEAP
+-----+
|   no allocation yet   |
+-----+

Action performed:
  write 42 into ptr->value
```

Result:

You are dereferencing a garbage address. Crash.

4.13.8.4 Fixed Version: lab1_fixed.c

```
#include <stdio.h>
#include <stdlib.h> // For malloc() and free()

struct data {
    int value;
};

int main(void) {
    struct data *ptr;

    // Allocate memory for ONE struct data on the heap
    ptr = malloc(sizeof(struct data));
    if (ptr == NULL) {
        // Always check in case malloc fails
        printf("Allocation failed!\n");
        return 1;
    }

    // Now 'ptr' points to valid memory, safe to use
    ptr->value = 42;
    printf("Value: %d\n", ptr->value);

    // Always free what you allocate
    free(ptr);

    return 0;
}
```

4.13.8.5 What Changed

- We used `malloc()` to allocate enough space for one `struct data`.
- We checked that the result was not `NULL`.
- We safely wrote into the struct's field.
- We freed the memory before exiting, preventing a leak.

4.13.8.6 Why this works

```

STACK (main)
+-----+
| ptr : 0xHHEE...      | -> valid heap address returned by malloc
+-----+

HEAP
+-----+
| struct data block    |
|   value = 42         |
+-----+

Actions:
  1) ptr = malloc(sizeof(struct data)) -> ptr now points to a valid block
  2) ptr->value = 42                    -> write inside your own block
  3) free(ptr)                         -> return memory to the system

Result:
  No crash. No leak.

```

4.13.9 Hands-On Lab 2: Memory Leak and the Forgotten Free

This exercise shows what happens when you forget to release allocated memory. In user space, leaks disappear when your program exits. In the kernel, leaks accumulate for the system's entire uptime, which is why this habit must be fixed early.

4.13.9.1 Leaky Version: lab2_leak.c

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void) {
    // Allocate 128 bytes of memory
    char *buffer = malloc(128);
    if (buffer == NULL) return 1;

    // Copy a string into the allocated buffer
    strcpy(buffer, "FreeBSD device drivers are awesome!");
    printf("%s\n", buffer);
}

```

```

    // Memory was allocated but never freed
    // This is a memory leak
    return 0;
}

```

4.13.9.2 What to Expect

- The program prints the string normally.
- You may not notice the problem right away because the OS reclaims process memory when the program exits.
- In the kernel this would be serious. The memory would remain allocated across operations and only a reboot clears it.

4.13.9.3 Leak vs program exit

Before exit:

STACK (main)

```

+-----+
| buffer : 0xABCD...      | -> heap address
+-----+

```

HEAP

```

+-----+
| 128-byte block          |
| "FreeBSD device drivers are awesome!\0 ..." |
+-----+

```

Action:

Program returns without `free(buffer)`

Consequence in user space:

OS reclaims process memory at exit, so you do not notice.

Consequence in kernel space:

The block remains allocated across operations and accumulates.

4.13.9.4 Fixed Version: `lab2_fixed.c`

```

#include <stdio.h>
#include <stdlib.h>

```

```
#include <string.h>

int main(void) {
    char *buffer = malloc(128);
    if (buffer == NULL) return 1;

    strcpy(buffer, "FreeBSD device drivers are awesome!");
    printf("%s\n", buffer);

    // Free the memory once we are done
    free(buffer);

    return 0;
}
```

4.13.9.5 What Changed

- We added `free(buffer);`.
- This single line ensures that all memory is returned to the system. Make this a habit.

4.13.9.6 Proper lifecycle

```
1) Allocation
   HEAP: [ 128-byte block ]  <- buffer points here

2) Use
   Write string into the block, then print it

3) Free
   free(buffer)
   HEAP: [ block returned to allocator ]
   buffer (optional) -> set to NULL to avoid accidental reuse
```

4.13.10 Detecting Memory Leaks with AddressSanitizer

On FreeBSD, when compiling user-space programs with Clang, you can detect leaks automatically using AddressSanitizer:

```
% cc -fsanitize=address -g -o lab2_leak lab2_leak.c
% ./lab2_leak
```

You will see a report indicating memory was allocated and never freed. Although AddressSanitizer does not apply to kernel code, the lesson is identical. Always release what you allocate.

4.13.11 Mini-Lab 3: Memory Allocation in a Kernel Module

Now let's try a FreeBSD kernel experiment. Create `memlab.c`:

```
#include <sys/param.h>
#include <sys/module.h>
#include <sys/kernel.h>
#include <sys/malloc.h>

MALLOC_DEFINE(M_MEMLAB, "memlab", "Memory Lab Example");

static void *buffer = NULL;

static int
memlab_load(struct module *m, int event, void *arg)
{
    int error = 0;

    switch (event) {
    case MOD_LOAD:
        printf("memlab: Loading module\n");

        buffer = malloc(128, M_MEMLAB, M_WAITOK | M_ZERO);
        if (buffer == NULL) {
            printf("memlab: malloc failed!\n");
            error = ENOMEM;
        } else {
            printf("memlab: allocated 128 bytes\n");
        }
        break;

    case MOD_UNLOAD:
        printf("memlab: Unloading module\n");

        if (buffer != NULL) {
            free(buffer, M_MEMLAB);
            printf("memlab: memory freed\n");
        }
        break;

    default:
        error = EOPNOTSUPP;
    }
}
```



```

        break;
    }

    return error;
}

static moduledata_t memlab_mod = {
    "memlab", memlab_load, NULL
};

DECLARE_MODULE(memlab, memlab_mod, SI_SUB_DRIVERS, SI_ORDER_MIDDLE);

```

Compile and load:

```

% cc -O2 -pipe -nostdinc -I/usr/src/sys -D_KERNEL -DKLD_MODULE \
    -fno-common -o memlab.o -c memlab.c
% cc -shared -nostdlib -o memlab.ko memlab.o
% sudo kldload ./memlab.ko

```

Unload with:

```
% sudo kldunload memlab
```

4.13.11.1 Detecting Leaks

Comment out the `free()` line, recompile, and load/unload several times. Now inspect memory:

```
% vmstat -m | grep memlab
```

You will see lines like:

```
memlab      128    4    4    0    0    1
```

indicating four allocations of 128 bytes are still “in use” because they were never freed. With the fix in place, the line disappears after unload.

4.13.11.2 Optional: DTrace View

To see allocations live:

```
% sudo dtrace -n 'fbt::malloc:entry { trace(arg1); }'
```

When you load the module, you will see the 128 bytes being allocated.

4.13.11.3 Challenge: Prove the Leak

It is one thing to read that kernel leaks accumulate, but another to **see it with your own eyes**. This short experiment will let you prove it on your own system.

1. Open the `memlab.c` module you created earlier and **comment out the `free(buffer, M_MEMLAB);` line** in the unload function. This means the module will allocate memory on load, but never release it on unload.
2. Rebuild the module and then **load and unload it four times in a row**:

```
% sudo kldload ./memlab.ko
% sudo kldunload memlab
% sudo kldload ./memlab.ko
% sudo kldunload memlab
% sudo kldload ./memlab.ko
% sudo kldunload memlab
% sudo kldload ./memlab.ko
% sudo kldunload memlab
```

3. Now inspect the kernel's memory allocation table with:

```
% vmstat -m | grep memlab
```

You should see output similar to:

```
memlab      128    4    4    0    0    1
```

This means four allocations of 128 bytes were made, and none were freed. Each time you loaded the module, the kernel allocated more memory that was never released.

4. Finally, **restore the `free()` line**, recompile, and repeat the load/unload cycle. This time, when you run `vmstat -m | grep memlab`, the line should disappear after unload, confirming that memory is released properly.

This simple test demonstrates a critical fact: in user space, leaks usually vanish when your process exits. In kernel space, leaks **survive across module reloads** and continue to accumulate. In production systems, such mistakes are not just messy; they are fatal. Over time, leaks can exhaust all available kernel memory and cause the system to crash.

4.13.12 Common Beginner Pitfalls

Memory safety is one of the hardest lessons for new C programmers, and in kernel space the consequences are much harsher. Let's highlight a few traps that beginners often fall into, and how you can avoid them:

- **Forgetting to free memory.** Every `malloc()` must have a matching `free()` in the proper cleanup path. If you allocate during module load, remember to free during module unload. This habit prevents leaks that otherwise accumulate for the entire uptime of the system.
- **Using freed memory.** Accessing a pointer after `free()` is called is a classic bug known as *use-after-free*. The pointer may still contain the old address, tricking you into thinking it is valid. A safe habit is to set the pointer to `NULL` immediately after freeing it. That way, any accidental use will be obvious.
- **Choosing the wrong allocation flag.** FreeBSD provides different allocation behaviours for different contexts. If you call `malloc()` with `M_WAITOK`, the kernel may put the thread to sleep until memory becomes available, which is fine during module load or attach, but catastrophic inside an interrupt handler. Conversely, `M_NOWAIT` never sleeps and fails immediately if memory is not available. Learning to pick the correct flag is an essential skill.
- **Skipping malloc tags.** Always use `MALLOC_DEFINE()` to give your driver a custom memory tag. These tags appear in `vmstat -m` and make debugging leaks much easier. Without them, your allocations may be lumped into generic categories, making it difficult to trace where memory is coming from.

By keeping these pitfalls in mind and practising the good habits shown earlier, you will dramatically reduce the risk of introducing memory bugs into your drivers. These lessons might feel repetitive now, but in real kernel development they are the difference between a stable driver and one that crashes production systems.

4.13.13 Golden Rules for Kernel Memory

1. Every `malloc()` must have a matching `free()`.
2. Never use a pointer before initialising it (or after freeing it).
3. Use the correct allocation flag (`M_WAITOK` or `M_NOWAIT`) for the context.

Keep these three rules in mind whenever you write kernel code. They may look simple, but following them consistently is what separates a stable FreeBSD driver from a crash-prone one.

4.13.14 Pointers Recap: The Lifecycle of Memory in C

Step 1: Declare a pointer

```
struct data *ptr;
```

STACK

+-----+

| ptr : ??? | -> uninitialised (dangerous!)

+-----+

Step 2: Allocate memory

```
ptr = malloc(sizeof(struct data));
```

STACK		HEAP
+-----+		+-----+
ptr : 0xABCD...	----->	struct data block
+-----+		value = ???
		+-----+

Step 3: Use the memory

```
-----
```

```
ptr->value = 42;
```

HEAP
+-----+
struct data block
value = 42
+-----+

Step 4: Free the memory

```
-----
```

```
free(ptr);
```

```
ptr = NULL;
```

STACK
+-----+
ptr : NULL -> safe, prevents reuse
+-----+

HEAP
+-----+
(block released)
+-----+

Golden Reminder:

- Never use an uninitialised pointer.
- Always check your allocations.
- Free what you allocate, and set pointers to NULL after freeing.

4.13.15 Pointers Recap Quiz

Test yourself with these quick questions before moving on. Answers are at the end of this chapter.

1. What happens if you declare a pointer but never initialise it and then dereference it?
2. Why should you always check the return value of `malloc()`?
3. What is the purpose of the `M_ZERO` flag when allocating memory in the FreeBSD kernel?
4. After calling `free(ptr, M_TAG);`, why is it a good habit to set `ptr = NULL;`?
5. In which contexts must you use `M_NOWAIT` instead of `M_WAITOK` when allocating memory in kernel code?

4.13.16 Wrapping Up

With this section, we have reached the end of our tour through **pointers in C**. Along the way you learned what pointers are, how they relate to arrays and structures, and why they are so flexible but also so dangerous. We concluded with one of the most important lessons: **memory safety**.

Every pointer must be treated with care. Every allocation must be checked. Every buffer must have a known size. In user space, mistakes usually crash only your program. In kernel space, the very same mistakes can corrupt memory or bring down the entire operating system.

By following FreeBSD's allocation patterns, checking results, freeing memory diligently, and using debugging tools like `vmstat -m` and `DTrace`, you will be on the path to writing drivers that are both stable and reliable.

In the next section, we'll cover **structures** and **typedefs** in C. Structures allow you to group related data, making your code more organized and expressive. Typedefs will enable you to assign meaningful names to complex types, improving readability. Together, they form the basis of almost every real kernel subsystem and are a natural step after mastering pointers.

A good moment to pause. You have now worked through the C foundations: variables, operators, control flow, functions, arrays, pointers, and the rules that keep `malloc(9)` and `free(9)` safe in kernel code. The rest of the chapter turns to the tools that shape larger programs: structures and `typedef`, header files and modular code, the compile-and-link workflow, the C preprocessor, and the practices that keep kernel-style C maintainable. If you want to close the book and come back, this is a natural place to stop.

4.14 Structures and typedef in C

So far, you have worked with single variables, arrays, and pointers. These are helpful tools, but real programs and especially operating system kernels need a way to organize related pieces of information together. Imagine trying to keep track of a device's name, its state, and its configuration using separate variables scattered everywhere. The result would be messy and hard to maintain.

C solves this problem with **structures**. A structure allows you to group related data under one roof, giving it a clear and logical shape. And once you have that structure, the `typedef` keyword can make your code shorter and easier to read by giving that structure a simpler name.

You will quickly discover that FreeBSD code is built on this foundation. Almost every kernel subsystem is defined as a set of structures. Once you understand them, the FreeBSD source tree starts looking far less intimidating.

4.14.1 What is a struct?

Up until now, you have stored data in single variables (`int`, `char`, etc.) or in arrays of the same type. But in real programming, we often need to keep *different* pieces of information together. For example, if we want to represent a **point in two-dimensional space**, we need both an x coordinate and a y coordinate. Keeping them in separate variables quickly becomes messy.

This is where a **structure** (or **struct**) comes in. A struct is a *user-defined type* that allows you to group several variables under a single name. Each variable inside the struct is referred to as a **member** or a **field**.

A useful way to imagine a struct is to think of it as a **folder**. A folder can hold different types of documents, a text file, an image, and a spreadsheet, but they are all kept together because they belong to the same project. A struct works the same way in C: it keeps related data together so you can handle it as one logical unit.

Here's a diagram of how a struct `Point` looks conceptually:

```
struct Point (like a folder)
├─ x  (an integer)
└─ y  (an integer)
```

Now let's see how this looks in C code:

```
#include <stdio.h>

// Define a structure to represent a point in 2D space
struct Point {
    int x;    // The X coordinate
    int y;    // The Y coordinate
};

int main() {
    // Declare a variable of type struct Point
    struct Point p1;

    // Assign values to its fields
    p1.x = 10;    // Set the X coordinate
    p1.y = 20;    // Set the Y coordinate

    // Print the values
    printf("Point p1 is at (%d, %d)\n", p1.x, p1.y);

    return 0;
}
```

Let's break down what happens here:

1. We define a new **blueprint** called `struct Point` that describes what every “Point” should contain: two integers, one called `x` and one called `y`.
2. In `main()`, we create a real variable `p1` that follows this blueprint.
3. We fill in its fields by assigning values to `p1.x` and `p1.y`.
4. We print the values, treating `p1` as one object with two related pieces of data.

This way, instead of juggling separate `int x` and `int y` variables, we now have a single object `p1` that neatly contains both.

In FreeBSD drivers, you will often find the same pattern, just with more complex fields: a device struct might keep its ID, its name, a pointer to its memory buffer, and its current state all in one place.

4.14.2 Accessing Structure Members

Once you define a structure, you need a way to reach its individual fields. C gives you two operators for this:

- The **dot operator** (`.`) is used when you have a **structure variable**.
- The **arrow operator** (`->`) is used when you have a **pointer to a structure**.

A good way to remember this is: **dot for direct, arrow for indirect**.

Let’s see both in action:

```
#include <stdio.h>

// Define a structure to represent a device
struct Device {
    char name[20];
    int id;
};

int main() {
    // Create a structure variable
    struct Device dev = {"uart0", 1};

    // Create a pointer to that structure
    struct Device *ptr = &dev;

    // Access using the dot operator (direct access)
    printf("Device name: %s\n", dev.name);

    // Access using the arrow operator (indirect access via pointer)
    printf("Device ID: %d\n", ptr->id);

    return 0;
}
```

Here's what is happening step by step:

1. We create a structure variable `dev` that represents a device with a name and an ID.
2. We then create a pointer `ptr` that points to `dev`.
3. When we write `dev.name`, we are directly accessing the name field of the struct variable.
4. When we write `ptr->id`, we are saying *"follow the pointer `ptr` to the struct it points to, then access the `id` field."*

The arrow operator is essentially shorthand for writing:

```
(*ptr).id
```

But because this looks clumsy, C provides `ptr->id` instead.

This distinction is fundamental in kernel programming. In FreeBSD, most kernel APIs will not give you a structure directly; they give you a **pointer to a structure**. For example, when working with processes, you will often get a struct `proc *p`, not a plain struct `proc`. That means you will spend most of your time using the arrow operator when writing drivers.

4.14.3 Making Code Cleaner with typedef

When we define a struct, we usually have to write the word `struct` every time we declare a variable. For small programs that is fine, but in large projects like FreeBSD this quickly becomes repetitive and makes the code harder to read.

The **typedef keyword** allows us to create a **shorter alias** for a type. It does not invent a new type; it simply gives an existing type a new name. This alias typically makes the code easier to read and clarifies the programmer's intent.

Here's an example that makes our struct `Point` easier to work with:

```
#include <stdio.h>

// Define a structure and immediately create a typedef for it
typedef struct {
    int x;    // X coordinate
    int y;    // Y coordinate
} Point;     // "Point" is now an alias for this struct

int main() {
    // We can now declare a Point without writing "struct Point"
    Point p = {1, 2};

    printf("Point is at (%d, %d)\n", p.x, p.y);

    return 0;
}
```


Without typedef, we would have to write:

```
struct Point p;
```

With typedef, we can simply write:

```
Point p;
```

This may look like a slight difference, but in real FreeBSD code, where structs can be very large and complex, using typedef keeps the source much cleaner.

4.14.4 Typedefs in FreeBSD

FreeBSD uses typedef extensively to:

1. **Give clearer names to primitive types** For example, in `/usr/include/sys/types` you will find lines like the lines below. Be aware that these lines presented below are spread across the file:

```
typedef __pid_t    pid_t;      /* process id */
typedef __uid_t    uid_t;      /* user id */
typedef __gid_t    gid_t;      /* group id */
typedef __dev_t    dev_t;      /* device number or struct cdev */
typedef __off_t    off_t;      /* file offset */
typedef __size_t    size_t;    /* object size */
```

Instead of scattering “int” or “long” everywhere in kernel code, FreeBSD developers use these typedefs. The code then speaks the language of the operating system: `pid_t` for process IDs, `uid_t` for user IDs, `size_t` for sizes, and so on.

2. **Hide system-dependent details** On some architectures, `pid_t` might be a 32-bit integer, while on others, it might be a 64-bit integer. The code that uses `pid_t` doesn’t care; the typedef handles that detail.
3. **Simplify complex declarations** FreeBSD also uses typedef to simplify long or pointer-heavy declarations. For example:

```
typedef struct _device *device_t;
typedef struct vm_page *vm_page_t;
```

Instead of always writing `struct _device *` throughout the kernel, developers can simply write `device_t`. This makes the code shorter and easier to read.

4.14.5 Why this matters in driver development

When you start writing drivers, you’ll constantly encounter types like `device_t`, `vm_page_t`, or `bus_space_tag_t`. They are all typedefs hiding more complex structures or pointers. Understanding that typedef is just an **alias** helps you avoid confusion and allows you to read FreeBSD code more fluently.

4.14.6 Real Example from FreeBSD 14.3: Creating a USB Character Device

Now that you've seen how structures and typedef work in isolation, let's step into real FreeBSD kernel code. This example comes from the USB subsystem, specifically the function `usb_make_dev` in `sys/dev/usb/usb_device.c`. This function is responsible for creating a character device node under `/dev` for a USB endpoint, allowing user programs to interact with it.

As you read the code, pay attention to two things:

1. How FreeBSD uses **structures** (`struct usb_fs_privdata` and `struct make_dev_args`) to bundle together all the information needed.
2. How it relies on **typedefs** like `uid_t` and `gid_t` to give clearer meaning to simple integers.

Here's the relevant code, with additional comments to help you connect it to the concepts we just studied:

File: `sys/dev/usb/usb_device.c`

```
[...]
struct usb_fs_privdata *
usb_make_dev(struct usb_device *udev, const char *devname, int ep,
             int fi, int rwmode, uid_t uid, gid_t gid, int mode)
{
    struct usb_fs_privdata* pd;
    struct make_dev_args args;
    char buffer[32];

    /* Allocate and initialise a private data structure for this device */
    pd = malloc(sizeof(struct usb_fs_privdata), M_USBDEV,
               M_WAITOK | M_ZERO);

    /* Fill in identifying fields */
    pd->bus_index = device_get_unit(udev->bus->bdev);
    pd->dev_index = udev->device_index;
    pd->ep_addr   = ep;
    pd->fifo_index = fi;
    pd->mode      = rwmode;

    /* Build the device name string if none was provided */
    if (devname == NULL) {
        devname = buffer;
        snprintf(buffer, sizeof(buffer), USB_DEVICE_DIR "%u.%u.%u",
                 pd->bus_index, pd->dev_index, pd->ep_addr);
    }

    /* Initialise and populate the make_dev_args structure */
    make_dev_args_init(&args);
    args.mda_devsw    = &usb_devsw; // Which device switch to use
```

```
args.mda_uid      = uid;          // Owner user ID (typedef uid_t)
args.mda_gid      = gid;          // Owner group ID (typedef gid_t)
args.mda_mode     = mode;         // Permission bits
args.mda_si_drv1  = pd;           // Attach our private data

/* Create the device node */
if (make_dev_s(&args, &pd->cdev, "%s", devname) != 0) {
    DPRINTF(0, "Failed to create device %s\n", devname);
    free(pd, M_USBDEV);
    return (NULL);
}
return (pd);
}
[...]
```

4.14.7 Why this example matters

This short function highlights several key lessons about structs and typedefs in real kernel work:

- **Grouping related data:** Instead of passing a dozen parameters around, FreeBSD collects them into struct `make_dev_args`. This makes it easier to extend the API later and keeps function calls readable.
- **Dot vs arrow:** Notice `args.mda_uid = uid;` uses the **dot** operator, because `args` is a direct struct variable. But `pd->dev_index = udev->device_index;` uses the **arrow**, because `pd` and `udev` are pointers. This is exactly the pattern you'll use constantly in your own drivers.
- **Initialisation:** The call to `make_dev_args_init(&args)` ensures `args` starts from a clean state, avoiding the uninitialised-field bug we discussed earlier.
- **Typedefs for clarity:** Instead of raw integers, the function signature uses `uid_t`, `gid_t`, and `mode`. These typedefs tell you exactly what kind of value is expected: a user ID, a group ID, and a permission mode. This is both safer and more self-documenting.

4.14.8 Takeaways

This example demonstrates how the building blocks you're learning (structures, pointers, and typedefs) are not academic curiosities, but the everyday language of FreeBSD drivers. Whenever a driver needs to configure a device, pass state between layers, or expose a new node in `/dev`, you'll find structs collecting the data and typedefs making it readable.

As you continue, remember this pattern:

1. Bundle related fields into a struct.
2. Use pointers with `->` when passing around instances.
3. Use typedefs where they make the code more expressive.

4.14.9 Hands-On Lab 1: Building a TTY Device Struct

In this exercise, you will define a **structure** to represent a simple TTY (teletypewriter) device. FreeBSD uses similar structures internally to keep track of terminals like `ttyv0` (the first virtual console).

```
#include <stdio.h>
#include <string.h>

// Step 1: Define the structure
struct tty_device {
    char name[16];           // Device name, e.g., "ttyv0"
    int minor_number;        // Device minor number
    int enabled;             // 1 = enabled, 0 = disabled
};

int main() {
    // Step 2: Declare a variable of the struct
    struct tty_device dev;

    // Step 3: Assign values to the fields
    strcpy(dev.name, "ttyv0"); // Copy string into 'name'
    dev.minor_number = 0;      // Minor number = 0 for ttyv0
    dev.enabled = 1;           // Enabled (1 = yes, 0 = no)

    // Step 4: Print the values
    printf("Device: %s\n", dev.name);
    printf("Minor number: %d\n", dev.minor_number);
    printf("Enabled: %s\n", dev.enabled ? "Yes" : "No");

    return 0;
}
```

Compile and run it:

```
% cc -o tty_device tty_device.c
% ./tty_device
```

Expected output:

```
Device: ttyv0
Minor number: 0
Enabled: Yes
```

What you learned: A struct groups related data into one logical unit. This mirrors how FreeBSD kernel code groups device metadata in struct `tty`.

4.14.10 Hands-On Lab 2: Using typedef for Cleaner Code

Now let's simplify the same program using **typedef**, which allows us to create a shorter alias for the **struct**. This makes code easier to read and avoids writing struct every time.

```
#include <stdio.h>
#include <string.h>

// Step 1: Define the struct and typedef it to TTYDevice
typedef struct {
    char name[16];           // Device name
    int minor_number;        // Minor number
    int enabled;             // Enabled flag (1 = enabled, 0 = disabled)
} TTYDevice;

int main() {
    // Step 2: Declare a variable of the new type
    TTYDevice dev;

    // Step 3: Fill in its fields
    strcpy(dev.name, "ttyu0"); // Another device example
    dev.minor_number = 1;      // Minor = 1 for ttyu0
    dev.enabled = 0;           // Disabled

    // Step 4: Print the values
    printf("Device: %s\n", dev.name);
    printf("Minor number: %d\n", dev.minor_number);
    printf("Enabled: %s\n", dev.enabled ? "Yes" : "No");

    return 0;
}
```

Compile and run:

```
% cc -o tty_typedef tty_typedef.c
% ./tty_typedef
```

Expected output:

```
Device: ttyu0
Minor number: 1
Enabled: No
```

Challenge Yourself

To make the struct more realistic (closer to FreeBSD's real struct `tty`), try these variations:

1. Add a baud rate:

```
int baud_rate;
```

Set it to 9600 or 115200 and print it.

2. Add a driver name:

```
char driver[32];
```

Assign "console" or "uart" to it.

3. Simulate multiple devices: Create an **array of structs** and initialise two or three devices, then print them all in a loop.

4.14.11 Extra Hands-On Lab: Managing Multiple TTY Devices

So far, you have created a single TTY device at a time. But in real drivers, you often manage **many devices at once**. FreeBSD keeps tables of devices, updating them as hardware is discovered, configured, or removed.

In this lab, you will build a small program that manages an **array of TTY devices**, searches for them by name, toggles their state, and even sorts them. This mirrors what real drivers do when working with multiple terminals, USB endpoints, or network interfaces.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>    // for qsort

// Define a TTY device structure with typedef
typedef struct {
    char name[16];        // e.g., "ttyv0", "ttyu0"
    int  minor_number;    // minor device number
    int  enabled;         // 1 = enabled, 0 = disabled
    int  baud_rate;       // communication speed, e.g., 9600 or 115200
    char driver[32];      // driver name, e.g., "console", "uart"
} TTYDevice;

// Print a single device (note: const pointer = read-only access)
void print_device(const TTYDevice *d) {
    printf("Device %-6s  minor=%d  enabled=%s  baud=%d  driver=%s\n",
        d->name, d->minor_number, d->enabled ? "Yes" : "No",
        d->baud_rate, d->driver);
}

// Print all devices in the array
void print_all(const TTYDevice *arr, size_t n) {
    for (size_t i = 0; i < n; i++)
        print_device(&arr[i]);
}
```

```
}

// Find a device by name. Returns a pointer or NULL if not found.
TTYDevice *find_by_name(TTYDevice *arr, size_t n, const char *name) {
    for (size_t i = 0; i < n; i++) {
        if (strcmp(arr[i].name, name) == 0)
            return &arr[i];
    }
    return NULL;
}

// Comparator for qsort: sort by minor number ascending
int cmp_by_minor(const void *a, const void *b) {
    const TTYDevice *A = (const TTYDevice *)a;
    const TTYDevice *B = (const TTYDevice *)b;
    return (A->minor_number - B->minor_number);
}

int main(void) {
    // 1) Create an array of devices using initialisers
    TTYDevice table[] = {
        { "ttyv0", 0, 1, 115200, "console" },
        { "ttyu0", 1, 0, 9600, "uart" },
        { "ttyu1", 2, 1, 9600, "uart" },
    };
    size_t N = sizeof(table) / sizeof(table[0]);

    printf("Initial device table:\n");
    print_all(table, N);
    printf("\n");

    // 2) Disable ttyv0 and enable ttyu0 using find_by_name()
    TTYDevice *p = find_by_name(table, N, "ttyv0");
    if (p != NULL) p->enabled = 0;

    p = find_by_name(table, N, "ttyu0");
    if (p != NULL) p->enabled = 1;

    printf("After toggling enabled flags:\n");
    print_all(table, N);
    printf("\n");

    // 3) Sort the devices by minor number (already sorted, but shows the pattern)
    qsort(table, N, sizeof(table[0]), cmp_by_minor);
}
```

```
printf("After sorting by minor number:\n");
print_all(table, N);
printf("\n");

// 4) Update all devices: set baud_rate to 115200
for (size_t i = 0; i < N; i++)
    table[i].baud_rate = 115200;

printf("After setting all baud rates to 115200:\n");
print_all(table, N);

return 0;
}
```

Compile and run:

```
% cc -o tty_table tty_table.c
% ./tty_table
```

Expected output:

Initial device table:

Device ttyv0	minor=0	enabled=Yes	baud=115200	driver=console
Device ttyu0	minor=1	enabled=No	baud=9600	driver=uart
Device ttyu1	minor=2	enabled=Yes	baud=9600	driver=uart

After toggling enabled flags:

Device ttyv0	minor=0	enabled=No	baud=115200	driver=console
Device ttyu0	minor=1	enabled=Yes	baud=9600	driver=uart
Device ttyu1	minor=2	enabled=Yes	baud=9600	driver=uart

After sorting by minor number:

Device ttyv0	minor=0	enabled=No	baud=115200	driver=console
Device ttyu0	minor=1	enabled=Yes	baud=9600	driver=uart
Device ttyu1	minor=2	enabled=Yes	baud=9600	driver=uart

After setting all baud rates to 115200:

Device ttyv0	minor=0	enabled=No	baud=115200	driver=console
Device ttyu0	minor=1	enabled=Yes	baud=115200	driver=uart
Device ttyu1	minor=2	enabled=Yes	baud=115200	driver=uart

Challenge Yourself!

Try extending this program:

1. **Filter enabled devices only** Write a function that prints only devices where `enabled == 1`.
2. **Find by minor number** Add a helper `find_by_minor(TTYDevice *arr, size_t n, int minor)`.
3. **Sort by name** Replace the comparator to sort alphabetically by name.
4. **Safer strings** Replace `strcpy` with `snprintf` to avoid buffer overflows.
5. **New field** Add a boolean field `is_console` and print `[CONSOLE]` next to those devices.

4.14.12 What did you learn in these labs?

The three labs you just completed began with the basics of defining a single struct, progressed to simplifying code with `typedef`, and concluded with managing an array of devices that could easily be part of a real driver. Along the way, you learned how C structures can group related fields into one logical unit, how `typedef` can make code more readable, and how arrays and pointers extend these ideas to represent and manage multiple devices at once.

You also practiced searching for a device by name and updating its fields, which reinforced the difference between dot and arrow access. You saw how pointers to structs (`p->enabled`) are the natural way to work with objects that are passed around in kernel code. You explored how to sort devices using `qsort()` and a comparator function, a pattern you will find in kernel subsystems that need to keep lists ordered. Finally, you learned how to apply changes to all devices in one sweep with a simple loop, a technique that reflects how drivers often update state across a whole set of devices when an event occurs.

By completing these labs, you have moved closer to the way FreeBSD drivers are really written: maintaining tables of devices, searching them efficiently, updating their state as hardware events happen, and relying on well-defined structs and typedefs to keep the code both safe and understandable.

4.14.13 Common Beginner Pitfalls with Structs and typedefs

Working with structs in C is straightforward once you get the hang of it, but beginners often trip over the same mistakes. In kernel programming, these mistakes are more than academic; they can lead to corrupted state, unexpected panics, or unreadable code.

1. Forgetting to initialise fields Declaring a struct does not automatically clear its memory. The fields contain whatever random values were left behind.

```
struct tty_device dev;           // uninitialised fields contain garbage
printf("%d\n", dev.minor_number); // undefined value
```

Safe practice: Always initialise structs. You can:

- Use designated initialisers:

```
struct tty_device dev = {.minor_number = 0, .enabled = 1};
```

- Or clear everything:

```
struct tty_device dev = {0}; // all fields zeroed
```

- Or, in kernel code, call an init helper (like `make_dev_args_init(&args)` in FreeBSD).

2. Mixing up dot and arrow The **dot** (.) is used with struct variables. The **arrow** (->) is used with pointers to structs. Beginners often use the wrong one.

```
struct tty_device dev;
struct tty_device *p = &dev;

dev.minor_number = 1;    // correct (variable)
p->minor_number = 2;     // correct (pointer)

p.minor_number = 3;      // wrong since p is a pointer
```

Safe practice: remember **dot** (.) for direct, **arrow** (->) for indirect.

3. Assuming copying structs = copying everything safely In C, assigning one struct to another copies all its fields **by value**. This can be dangerous if the struct contains pointers, because both structs will then point to the same memory.

```
struct buffer {
    char *data;
    int len;
};

struct buffer a, b;
a.data = malloc(100);
a.len = 100;

b = a;           // shallow copy: b.data points to the same memory as a.data
free(a.data);    // now b.data is a dangling pointer
```

Safe practice: If your struct holds pointers, be explicit about ownership. In kernel code, be clear about who allocates and frees memory. Sometimes you need to write a custom “copy” function.

4. Misunderstanding padding and alignment The compiler may insert padding between struct fields to meet alignment requirements. This can surprise beginners who assume the struct’s size is just the sum of its fields.

```
struct example {
    char a;
    int b;
};
printf("%zu\n", sizeof(struct example)); // Often 8, not 5
```

Safe practice: Don’t assume layout. If you need tight packing for hardware structures, use fixed-size types (uint32_t) and consult sys/param.h. In FreeBSD, bus structures often rely on precise layouts.

5. Overusing typedef typedef makes code shorter, but using it everywhere can hide meaning.

```
typedef struct {  
    int x, y, z;  
} Foo;    // "Foo" tells us nothing
```

Safe practice: Reserve typedef for:

- Very common or standardised types (pid_t, uid_t, device_t).
- Hiding architecture details (uintptr_t, vm_offset_t).
- Complex or pointer-heavy types (typedef struct _device *device_t;).

Avoid typedefs for one-off structs where struct makes intent clearer.

4.14.14 Recap Quiz: Structs and typedef Pitfalls

Before moving on, take a moment to test yourself. The following quick quiz revisits the most common pitfalls we discussed with structs and typedefs.

These questions are not about memorizing syntax, but about checking whether you understand the reasoning behind safe practices. If you can answer them confidently, you are well prepared to recognise and avoid these mistakes when you start writing real FreeBSD driver code.

Let's try?

1. What happens if you declare a struct without initialising it?

- a) It is filled with zeroes.
- b) It contains leftover garbage values.
- c) The compiler raises an error.

2. You have this code:

```
struct device dev;  
struct device *p = &dev;
```

Which are correct ways to set a field?

- a) dev.id = 1;
- b) p->id = 1;
- c) p.id = 1;

3. Why is assigning one struct to another (b = a;) dangerous if the struct contains pointers?

4. Why might sizeof(struct example) be larger than the sum of its fields?

5. When is it appropriate to use typedef with a struct, and when should you avoid it?

4.14.15 Wrapping Up

You have now learned how to define and use structures, how to make code cleaner with typedef, and how to recognize real patterns inside the FreeBSD kernel. Structures are the backbone of kernel programming. Almost

every subsystem is represented as a struct, and device drivers rely on them to maintain state, configuration, and runtime data.

In the next section we will take this one step further by looking at **header files and modular code in C**. This will show you how structs and typedefs are shared across multiple source files, which is precisely how large projects like FreeBSD stay organized and maintainable.

4.15 Header Files and Modular Code

So far, all of our programs have lived in a **single .c file**. That's fine for small examples, but real software quickly outgrows this model. The FreeBSD kernel itself is made up of thousands of small files, each with a clear responsibility. To manage this complexity, C provides a way to build **modular code**: splitting definitions into .c files and declarations into .h files.

We already saw in Section 4.3 that `#include` pulls in header files, and in Section 4.7 that declarations tell the compiler what a function looks like. Now we'll go one step further and combine these ideas into a method that scales: **header files for modular programs**.

4.15.1 What Is a Header File?

When programs grow beyond a single file, you need a way for different .c files to “agree” on what functions, structures, or constants exist. That's the job of a **header file**, which normally has the extension .h.

Think of a header as a **contract**: it doesn't do the work itself, but it describes what is available for other files to use. A .c file can then include this contract with `#include` so that the compiler knows what to expect.

Typical things you will find in a header are:

- **Function prototypes** (so other files know how to call them)
- **Struct and enum definitions** (so data types are shared consistently)
- **Macros and constants** defined with `#define`
- **extern variable declarations** (so a variable can be shared without creating multiple copies)
- Occasionally, **small inline helper functions**

A header file is **never compiled independently**. Instead, it is included by one or more .c files, which then provide the actual implementations.

4.15.2 Header Guards: Why They Exist and How They Work

One of the first problems you encounter in modular programs is **accidental duplication**.

Imagine this scenario:

- `main.c` includes `mathutils.h`.
- `mathutils.c` also includes `mathutils.h`.
- When the compiler combines everything, it may attempt to include the same header file **twice**.

If the header defines the same functions or structures more than once, the compiler will throw errors like “*re-definition of struct ...*”.

To prevent this, C programmers wrap every header in a **guard**, which is a set of preprocessor commands that tell the compiler:

1. “*If this header hasn’t been included yet, include it now.*”
2. “*If it has already been included, skip it.*”

Here’s what it looks like in practice:

```
#ifndef MATHUTILS_H
#define MATHUTILS_H

int add(int a, int b);
int subtract(int a, int b);

#endif
```

Step by step:

1. **#ifndef MATHUTILS_H** Reads as “*if not defined.*” The preprocessor checks whether MATHUTILS_H is already defined.
2. **#define MATHUTILS_H** If it wasn’t defined, we define it now. This marks the header as “processed.”
3. **Contents of the header** Prototypes, constants, or struct definitions should be placed here.
4. **#endif** Ends the guard.

If another file includes `mathutils.h` again later:

- The preprocessor checks `#ifndef MATHUTILS_H`.
- But now MATHUTILS_H is already defined.
- Result: the compiler **skips the entire file**.

This ensures the header’s contents are only included once, even if you `#include` it from multiple places.

4.15.3 Why the Name Matters

The symbol used in the guard (MATHUTILS_H in our example) is arbitrary, but there are conventions:

- Written in **all caps**
- Usually based on the file name
- Sometimes includes path information for uniqueness (e.g., `_SYS_PROC_H` for `/usr/src/sys/sys/proc.h`)

4.15.4 A Real FreeBSD Example

If you open the file `/usr/src/sys/sys/proc.h`, you’ll see a very similar guard:

```
#ifndef _SYS_PROC_H_
#define _SYS_PROC_H_

/* contents of the header ... */

#endif /* !_SYS_PROC_H_ */
```

This is the same pattern, just with a different naming style. It ensures that the definition of `struct proc` (and everything else in that header) is only read once, no matter how many other files include it.

4.15.5 Hands-On Lab 1: Seeing Header Guards in Action

Let's prove why header guards matter by creating two small programs. The first will fail without guards, and the second will succeed once we add them.

4.15.5.1 Step 1: Create a Header Without a Guard

File: `badheader.h`

```
// badheader.h
int add(int a, int b);
```

File: `badmain.c`

```
#include <stdio.h>
#include "badheader.h"
#include "badheader.h" // included twice on purpose!

int add(int a, int b) {
    return a + b;
}

int main(void) {
    printf("Result: %d\n", add(2, 3));
    return 0;
}
```

Now compile it:

```
% cc badmain.c -o badprog
```

You should see an error similar to:

```
badmain.c:3:5: error: redefinition of 'add'
```

This occurred because the header was included twice, causing the compiler to see the same function declaration twice.

4.15.5.2 Step 2: Fix It with a Header Guard

Now edit `badheader.h` to add a guard:

```
#ifndef BADHEADER_H
#define BADHEADER_H

int add(int a, int b);

#endif
```

Recompile:

```
% cc badmain.c -o goodprog
% ./goodprog
Result: 5
```

With the guard in place, the compiler ignored the duplicate include, and the program compiled successfully.

4.15.5.3 What You Learned

- Including the same header twice **without guards** causes errors.
- A header guard prevents those errors by ensuring the file's contents are only processed once.
- This is why every professional C project, including the FreeBSD kernel, consistently uses header guards.

4.15.6 Why Use Modular Code?

As programs grow, keeping everything in one `.c` file quickly becomes messy. A single file full of unrelated functions is hard to read, hard to maintain, and almost impossible to scale. Modular code solves this problem by **splitting the program into logical parts**, with each file handling a clear responsibility.

Think of it like building with LEGO: each block is small and specialised, but they connect cleanly to form something larger. In C, the “connecting pieces” are header files. They allow one part of the program to know about the functions and data structures defined in another part without copying the same declarations everywhere.

This separation has important advantages:

- **Organisation** - Each `.c` file focuses on a single task, making the code easier to navigate.

- **Reusability** - Headers allow you to use the same function declarations in many different files without rewriting them.
- **Maintainability** - If a function's signature changes, you update its header once and every file that includes it stays consistent.
- **Scalability** - Adding new functionality is easier when each piece of code has its own place.
- **Kernel compatibility** - This is exactly how FreeBSD is built: the kernel, drivers, and subsystems all rely on thousands of small .c and .h files that fit together. Without modularity, a project the size of FreeBSD would be unmanageable.

4.15.7 A Simple Multi-File Example

Let's write a small program split into three files.

4.15.7.1 mathutils.h

```
#ifndef MATHUTILS_H
#define MATHUTILS_H

int add(int a, int b);
int subtract(int a, int b);

#endif
```

4.15.7.2 mathutils.c

```
#include "mathutils.h"

int add(int a, int b) {
    return a + b;
}

int subtract(int a, int b) {
    return a - b;
}
```

4.15.7.3 main.c

```
#include <stdio.h>
#include "mathutils.h"
```



```
int main(void) {
    int x = 10, y = 4;

    printf("Add: %d\n", add(x, y));
    printf("Subtract: %d\n", subtract(x, y));

    return 0;
}
```

Compile them together:

```
% cc main.c mathutils.c -o program
% ./program
Add: 14
Subtract: 6
```

Notice how the header allowed `main.c` to call `add()` and `subtract()` without knowing their implementation details.

4.15.8 How FreeBSD Uses Headers (Example from FreeBSD 14.3)

Kernel headers in FreeBSD do more than just hold declarations. They are carefully structured: first they protect against double inclusion, then they pull in only the dependencies they need, then they give you guidance on how to read the data types they define, and finally they publish those shared types. A good example of this is the header `sys/sys/proc.h`, which defines the `struct proc` used to represent processes.

Let's walk through three important parts of this header.

4.15.8.1 1) Header guard and focused includes

File: `/usr/src/sys/sys/proc.h` - **Lines 37-49**

```
#ifndef _SYS_PROC_H_                // Guard: process this header only once
#define _SYS_PROC_H_

#include <sys/callout.h>             /* For struct callout. */
#include <sys/event.h>               /* For struct klist. */
#ifdef _KERNEL
#include <sys/_eventhandler.h>
#endif
#include <sys/_exterr.h>
#include <sys/condvar.h>
#ifdef _KERNEL
```

```
#include <sys/filedesc.h>
#endif
```

What to notice

- The **guard** `_SYS_PROC_H_` ensures the file is processed only once, no matter how many times it is included.
- The includes are **minimal and purposeful**. For example, `<sys/filedesc.h>` is pulled in only when compiling for userland (`#ifndef _KERNEL`). This keeps the kernel build lean and avoids unnecessary dependencies.

This shows the discipline you should adopt in your own drivers: guard every header, and only include what you truly need.

4.15.8.2 2) Preparing the reader: documentation + forward declarations

Lines 129-206 (excerpt)

```
/*-
 * Description of a process.
 *
 * This structure contains the information needed to manage a thread of
 * control, known in UN*X as a process; it has references to substructures
 * containing descriptions of things that the process uses, but may share
 * with related processes. ...
 *
 * Below is a key of locks used to protect each member of struct proc. ...
 *   a - only touched by curproc or parent during fork/wait
 *   b - created at fork, never changes
 *   c - locked by proc mtx
 *   d - locked by allproc_lock lock
 *   e - locked by proctree_lock lock
 *   ...
 */

struct cpuset;
struct filecaps;
struct filemon;
/* many more forward declarations ... */

struct syscall_args {
    u_int code;
    u_int original_code;
    struct sysent *callp;
```

```
    register_t args[8];
};
```

What to notice

- Before introducing `struct proc`, the header provides a **long explanatory comment**. This is not just filler; it tells you what a process is and introduces the **locking key** (letters like a, b, c, d, e). These letters are used as shorthand later, next to each field of the process structure, to show which lock protects it.
- Then the file defines a number of **forward declarations** (e.g., `struct cpuset`;). These act as placeholders, saying “this type exists, but you don’t need its full definition here.” This keeps the header lightweight and avoids pulling in lots of other headers unnecessarily.

Think of this section as a **map legend**: it gives you the symbols (locking keys) and placeholders you’ll need before you reach the main data structure. And in the next item, you’ll see exactly how those locking keys are applied to real fields inside `struct proc`.

4.15.8.3 3) The shared data structure definition (`struct proc`)

Lines 652-779 (start and a few fields shown)

```
/*
 * Process structure.
 */
struct proc {
    LIST_ENTRY(proc) p_list;          /* (d) List of all processes. */
    TAILQ_HEAD(, thread) p_threads; /* (c) all threads. */
    struct mtx      p_slock;          /* process spin lock */
    struct ucred *p_ucred;            /* (c) Process owner's identity. */
    struct filedesc *p_fd;            /* (b) Open files. */
    ...
    pid_t          p_pid;             /* (b) Process identifier. */
    ...
    struct mtx      p_mtx;            /* (n) Lock for this struct. */
    ...
    struct vmSPACE *p_vmSPACE;        /* (b) Address space. */
    ...
    char            p_comm[MAXCOMLEN + 1]; /* (x) Process name. */
    ...
    struct pgrp *p_pgrp;              /* (c + e) Process group linkage. */
    ...
};
```

What to notice

- This is the central definition: the `struct proc`.

- Every field is annotated with one or more **locking keys** from the earlier comment. For example, `p_list` is marked (d), meaning it is protected by the `allproc_lock`, while `p_threads` is marked (c), meaning it requires the `proc_mtx`.
- The structure connects to other subsystems through pointers (`ucred`, `vm_space`, `pgrp`). Because the header only needs to declare the pointer types, the earlier forward declarations were enough; no need to include all of those subsystem headers here.

This shows the modularity pattern in action: a single header publishes a large, shared structure in a way that is safe, documented, and efficient to include across the kernel.

4.15.8.4 How to verify on your machine

- Guard and includes: **lines 37-49**
- Comment with locking key and forward declarations: **lines 129-206**
- Start of `struct proc`: **lines 652-779**

You can confirm these with:

```
% nl -ba /usr/src/sys/sys/proc.h | sed -n '37,49p'
% nl -ba /usr/src/sys/sys/proc.h | sed -n '129,206p'
% nl -ba /usr/src/sys/sys/proc.h | sed -n '652,679p'
```

Note: `nl -ba` prints line numbers for every line (even blanks). The ranges above are for the FreeBSD 14.3 tree at the time of writing and may drift a few lines in later releases; if the output does not start at the section shown, scroll up or down to the nearest `#ifndef _SYS_PROC_H_`, locking-key comment, or `struct proc {` landmark.

4.15.8.5 Why this matters

When you write FreeBSD drivers, you'll follow the same structure:

- Put **shared types and prototypes** in headers.
- Protect every header with a **guard**.
- Use **forward declarations** to avoid dragging in unnecessary dependencies.
- Document your headers so future developers know how to use them.

The modular, disciplined structure you see here is exactly what makes a large system like FreeBSD maintainable.

4.15.9 Common Beginner Pitfalls with Headers

Headers make modular code possible, but they also introduce a few traps that beginners often fall into. Here are the main ones to watch for:

1. Defining variables in headers

```
int counter = 0;    // Wrong inside a header
```

Every .c file that includes this header will try to create its own counter, leading to *multiple definition* errors at link time. **Fix:** In the header, declare it with `extern int counter;`. Then, in exactly one .c file, write `int counter = 0;`.

2. Forgetting header guards Without guards, including the same header twice will cause “redefinition” errors. This is especially easy to do in large projects where one header includes another. **Fix:** Always wrap the file in a guard or use `#pragma once` if your compiler supports it.

3. Declaring but never defining A prototype in a header tells the compiler that a function exists, but if you never write it in a .c file, the **linker** will fail with an “undefined reference” error. **Fix:** For every declaration you publish in a header, make sure there’s a matching definition somewhere in your code.

4. Circular includes If a.h includes b.h and b.h includes a.h, the compiler will chase its tail until it errors out. This can happen by accident in large modular projects. **Fix:** Break the cycle. Usually you can move one of the includes into the corresponding .c file, or replace it with a **forward declaration** if you only need a pointer type.

4.15.10 Hands-On Lab 2: Triggering Pitfalls on Purpose

Let’s reproduce two of these mistakes so you can see what they look like.

4.15.10.1 Part A: Defining a variable in a header

Create `badvar.h`:

```
// badvar.h
int counter = 0;    // Wrong: definition in header
```

Create `file1.c`:

```
#include "badvar.h"
int inc1(void) { return ++counter; }
```

Create `file2.c`:

```
#include "badvar.h"
int inc2(void) { return ++counter; }
```

And finally `main.c`:

```
#include <stdio.h>

int inc1(void);
int inc2(void);

int main(void) {
    printf("%d\n", inc1());
    printf("%d\n", inc2());
    return 0;
}
```

Now compile:

```
% cc main.c file1.c file2.c -o badprog
```

You'll see an error like:

```
multiple definition of `counter'
```

Both `file1.c` and `file2.c` created their own counter from the header, and the linker refused to merge them.

Fix: Change `badvar.h` to:

```
extern int counter;    // Only a declaration
```

And in *one* .c file (say, `file1.c`):

```
int counter = 0;       // The single definition
```

Now the program compiles and runs correctly.

4.15.10.2 Part B: Declaring but not defining a function

Create `mylib.h`:

```
int greet(void);    // Declaration only
```

Create `main.c`:

```
#include <stdio.h>
#include "mylib.h"

int main(void) {
    printf("Message: %d\n", greet());
    return 0;
}
```

Compile:

```
% cc main.c -o badprog
```

The compiler accepts it, but the linker will fail:

```
undefined reference to `greet'
```

The declaration promised that `greet()` exists, but no `.c` file ever defined it.

Fix: Add a `mylib.c` with the real function:

```
#include "mylib.h"

int greet(void) {
    return 42;
}
```

Recompile with:

```
% cc main.c mylib.c -o goodprog
% ./goodprog
Message: 42
```

4.15.10.3 What You Learned

- Defining variables directly in headers creates *multiple copies* across `.c` files; use `extern` instead.
- A declaration without a definition will compile but fail later at the linking stage.
- Header guards and forward declarations prevent common inclusion issues.
- These small mistakes are easy to make, but once you've seen the error messages, you'll recognize them instantly in your own projects.

This is the same discipline followed in FreeBSD: headers only **declare** things, while the actual work lives in `.c` files.

4.15.11 Recap Quiz: Header Pitfalls

1. Why does putting `int counter = 0;` inside a header cause multiple definition errors?
 - a) Because each `.c` file gets its own copy
 - b) Because the compiler does not allow global variables
 - c) Because headers cannot contain integers
2. If a function is declared in a header but never defined in a `.c` file, which stage of the build process will report the error?
 - a) Preprocessor
 - b) Compiler
 - c) Linker
3. What is the purpose of a forward declaration (e.g., `struct device;`) in a header?
 - a) To save memory at runtime
 - b) To tell the compiler that the type exists without pulling in its full definition
 - c) To define the entire structure immediately
4. What problem do header guards prevent?
 - a) Declaring but not defining a function
 - b) Multiple inclusion of the same header
 - c) Circular dependencies between headers

4.15.12 Hands-On Lab 3: Your First Modular Program

Up to now, you've written all your programs in a single `.c` file. That works for small exercises, but real-world projects, especially the FreeBSD kernel, rely on many files working together. Let's build a tiny modular program that mimics this structure.

You'll create three files:

Step 1: The Header (`greetings.h`)

This file declares what the other `.c` files need to know.

```
#ifndef GREETINGS_H
#define GREETINGS_H

void say_hello(void);
void say_goodbye(void);

#endif
```

Notice the **header guard**. Without it, including the same header more than once would cause errors.

Step 2: The Implementation (`greetings.c`)

This file provides the actual definitions.


```
#include <stdio.h>
#include "greetings.h"

void say_hello(void) {
    printf("Hello from greetings.c!\n");
}

void say_goodbye(void) {
    printf("Goodbye from greetings.c!\n");
}
```

Step 3: The Main Program (main.c)

This file uses the functions.

```
#include "greetings.h"

int main(void) {
    say_hello();
    say_goodbye();
    return 0;
}
```

Step 4: Compile and Run

Compile all three files together:

```
% cc main.c greetings.c -o greetings
% ./greetings
```

Expected output:

```
Hello from greetings.c!
Goodbye from greetings.c!
```

Understanding the Output

- main.c doesn't know how the functions are implemented; it only knows their prototypes from greetings.h.
- greetings.c contains the real code, which the linker combines with main.c.
- This is exactly how modular programming works: **headers declare, .c files define**.

This lab is different from the pitfalls lab because here you're building a **working modular program** from scratch, not deliberately breaking things.

Hands-On Lab 4: Exploring FreeBSD Headers

Now that you've built your own modular program, let's look at how FreeBSD does the same thing at scale.

Step 1: Locate the Process Header

Go to the system headers:

```
% cd /usr/src/sys/sys
% grep -n "struct proc" proc.h
```

This shows you where the definition of `struct proc` begins.

Step 2: Examine Members of `struct proc`

Scroll through the file and write down at least **five fields** you find, for example:

- `p_pid` - the process identifier
- `p_comm` - the process name
- `p_ucred` - the user credentials
- `p_vmspace` - the address space
- `p_threads` - the list of threads

Each of these fields connects the process to another subsystem in the kernel.

Step 3: Find Where They Are Used

Search for one field (for example, `p_pid`) across the source tree:

```
% cd /usr/src/sys
% grep -r p_pid .
```

You'll see dozens of files that use this field. That's the power of modularity: a single definition in `proc.h` is shared by the entire kernel.

What You Learned

- The definition of `struct proc` lives in one header file.
- Any `.c` file that needs process information just includes `<sys/proc.h>`.
- This avoids duplication and guarantees consistency: every `.c` file sees the same structure.

This exercise helps you start reading kernel headers the way kernel developers do by following types from their declaration to their usage across the system.

4.15.13 Why This Matters for Driver Development

At first glance, writing headers might seem like bookkeeping, but for FreeBSD driver developers, headers are the glue that holds everything together.

When you write a driver, you will:

- **Declare your driver's interface** in a header, so other parts of the kernel know how to call your functions (for example, `probe`, `attach`, and `detach` routines).

- **Include kernel subsystem headers** such as `<sys/bus.h>`, `<sys/conf.h>`, or `<sys/proc.h>`, so you can use core types like `device_t`, `struct cdev`, and `bus_space_tag_t`.
- **Reuse subsystem APIs** without rewriting them; you include the right header and suddenly your driver has access to the correct functions and structures.
- **Keep your driver maintainable.** Clean headers make it easy for other developers to understand what your driver exposes.

Headers are also one of the first things reviewers check when you submit code to FreeBSD. If your driver puts definitions in the wrong place, forgets guards, or pulls in unnecessary dependencies, it will be flagged immediately. Clean, minimal headers show that you understand how to integrate properly into the kernel.

In short, **headers are not optional details; they are the foundation of collaboration inside the FreeBSD kernel.**

4.15.14 Wrapping Up

Header files are the backbone of modular C programming. They allow multiple files to work together safely, prevent duplication, and give structure to large systems like FreeBSD.

By now, you've:

- Written your own modular program with headers
- Explored how FreeBSD uses headers to define `struct proc`
- Learned common pitfalls to avoid
- Understood why headers are critical for writing drivers

In the next section, we'll see how the compiler and linker combine all these separate `.c` and `.h` files into a single program and how debugging tools help when things go wrong. This is where your modular code truly comes to life.

4.16 Compiling, Linking, and Debugging C Programs

Up to this point you have written short C programs and even split them into more than one file. Now it is time to understand what really happens between writing your code and running it on FreeBSD. This is the step where your text file becomes a living program. It is also the step where you first learn to read the compiler's messages and start using a debugger when things go wrong. These skills are the difference between "just typing code" and truly programming with confidence.

4.16.1 The compilation pipeline, in plain words

Think of the compiler as a factory line. You give it raw material (your `.c` source file) and it sends the material through several machines. By the end, you have a finished product: an executable program you can run.

Here is the sequence in simple terms:

1. **Preprocessing** - This is like preparing the ingredients before cooking. The compiler looks at `#include` and `#define` lines and replaces them with the actual content or values. At this stage, your program is expanded into the final "recipe" that will be cooked.

2. **Compilation** - Now the recipe is turned into instructions your CPU can understand at a more abstract level. Your C code is translated into **assembly language**, which is close to the language of the hardware but still readable to humans.
3. **Assembling** - This step turns the assembly instructions into pure machine code, stored in an **object file** with the extension `.o`. These files are not yet a full program; they are like separate puzzle pieces.
4. **Linking** - Finally, the puzzle pieces are put together. The linker combines all your object files, and also pulls in any libraries you need (such as the standard C library for `printf`). The result is a single executable file you can run.

On FreeBSD the command `cc` (which uses Clang by default) takes care of this whole process for you. You usually do not see each step, but knowing that they exist makes it much easier to understand error messages. For example, a syntax error happens during **compilation**, while a message like “undefined reference” comes from the **linking** stage.

4.16.2 Reading compiler messages with care

By now, you have compiled many small programs. Instead of repeating those steps, let us pause to look more closely at **what the compiler tells you during compilation**. The messages from `cc` are not just obstacles; they are hints, sometimes even teaching moments.

Take this small example:

```
#include <stdio.h>

int main(void) {
    prinft("Oops!\n"); // Typo on purpose
    return 0;
}
```

If you compile with:

```
% cc -Wall -o hello hello.c
```

you will see a message similar to:

```
hello.c: In function 'main':
hello.c:4:5: error: implicit declaration of function 'prinft'
```

The compiler is saying: *“I do not know what `prinft` is, maybe you meant `printf`?”*

The `-Wall` flag is important because it enables the standard set of warnings. Even when your program does compile, warnings can alert you to suspicious code that may later cause a bug.

From this point forward, make it a habit:

- Always compile with warnings enabled.

- Always read the **first** error or warning carefully. Often fixing that one will also fix the rest.

This practice may look simple, but it is the same discipline you will need when building large drivers in the FreeBSD kernel, where build logs can be long and intimidating.

4.16.3 Multi-file programs and linker errors

You already know how to split code into multiple `.c` and `.h` files. What matters now is understanding **why we do this** and what kinds of errors appear when something goes wrong.

When you compile each file separately, the compiler produces **object files** (`.o`). These are like puzzle pieces: each piece has some functions and data, but cannot run alone. The linker is the one that joins all the pieces into a complete picture.

For example, suppose you have this setup:

```
main.c
#include <stdio.h>
#include "utils.h"

int main(void) {
    greet();
    return 0;
}
utils.c
#include <stdio.h>
#include "utils.h"

void greet(void) {
    printf("Hello from utils!\n");
}
utils.h
#ifndef UTILS_H
#define UTILS_H
void greet(void);
#endif
```

Build step by step:

```
% cc -Wall -c main.c    # produces main.o
% cc -Wall -c utils.c   # produces utils.o
% cc -o app main.o utils.o
```

Now imagine you accidentally change the function name in `utils.c`:

```
void greet_user(void) {    // oops: name does not match
    printf("Hello!\n");
}
```

Recompile and link, and you will see something like:

```
undefined reference to `greet'
```

This is a **linker error**, not a compiler error. The compiler was satisfied that each file made sense on its own. The linker, however, could not find the missing piece: the real machine code for `greet`.

By noticing whether an error comes from the compiler or the linker, you can immediately narrow your search:

- Compiler errors = the syntax or types inside one file are wrong.
- Linker errors = the files do not agree with each other, or a function is missing.

This difference is small but important. FreeBSD drivers are often spread across multiple files, so being able to distinguish between compiler and linker messages will save you hours of frustration.

4.16.4 Why make matters in real projects

Compiling a single file by hand with `cc` is manageable. Even two or three files are fine; you can still type out the commands without losing track. But as soon as your program grows beyond that, the manual approach starts to collapse under its own weight.

Imagine a project with ten `.c` files, each including two or three different headers. You fix one small typo in a header and suddenly *every file that includes that header needs to be recompiled*. If you forget even one, the linker may quietly stitch together a program that half-knows about your change and half-doesn't. These out-of-sync builds are some of the most confusing bugs for beginners, because the program “compiles” yet produces unpredictable results.

This is exactly the problem that `make` was created to solve. Think of `make` as the guardian of consistency.

A **Makefile** describes:

- what files your program consists of,
- how each one is compiled,
- and how they fit together into the final executable.

Once that description exists, `make` takes care of the tedious part. If you change a single source file, `make` recompiles only that file and then relinks the program. If you only edit comments, nothing gets rebuilt at all. This saves you both time and mistakes.

In FreeBSD, `make` is not an optional tool, it is the backbone of the build system. When you run `make build-world`, you are rebuilding the entire operating system with the same rules you are learning here. When you compile a kernel module, the `bsd.kmod.mk` file you include in your `Makefile` is just a highly polished version of the simple `Makefiles` you are writing now.

The real lesson is this: by learning `make` now, you are not just avoiding repetitive typing. You are practicing the exact workflow you will need when you reach device drivers. Those drivers will almost always be split across multiple files, include kernel headers, and need to be rebuilt every time you adjust an interface. Without `make`, you would spend more time typing commands than actually writing code.

Common beginner pitfalls with `make`:

- Forgetting to list a header file as a dependency, which means changes in the header do not trigger a rebuild.
- Mixing spaces and tabs in a Makefile (a classic frustration). Only tabs are allowed at the start of a command line.
- Hardcoding flags instead of using variables like `CFLAGS`, which makes your Makefile less flexible.

By practicing with small Makefiles now, you will be prepared for the larger, system-wide ones that power FreeBSD itself.

4.16.5 Hands-On Lab 1: When `make` misses a change

Objective: See what happens when a Makefile is missing a dependency, and why accurate dependency tracking is essential in real projects.

Step 1 - Set up a tiny project.

File: `main.c`

```
#include <stdio.h>
#include "greet.h"

int main(void) {
    greet();
    return 0;
}

greet.c
#include <stdio.h>
#include "greet.h"

void greet(void) {
    printf("Hello!\n");
}
```

File: `greet.h`

```
#ifndef GREET_H
#define GREET_H

void greet(void);

#endif
```

File: Makefile

```
CC=cc
CFLAGS=-Wall -g

app: main.o greet.o
    $(CC) $(CFLAGS) -o app main.o greet.o

main.o: main.c
    $(CC) $(CFLAGS) -c main.c

greet.o: greet.c
    $(CC) $(CFLAGS) -c greet.c

clean:
    rm -f *.o app
```

Notice that the dependencies for `main.o` and `greet.o` **do not list the header file** `greet.h`.

Step 2 - Build and run.

```
% make
% ./app
```

Output:

```
Hello!
```

Everything works.

Step 3 - Change the header.

Edit `greet.h`:

```
#ifndef GREET_H
#define GREET_H

void greet(void);
void greet_twice(void); // New function added
#endif
```

Do **not** change `main.c` or `greet.c`. Now run:


```
% make
```

make responds with:

```
make: 'app' is up to date.
```

But this is wrong! The header changed, so `main.o` and `greet.o` should have been rebuilt.

Step 4 - Fix the Makefile.

Update the dependencies:

```
main.o: main.c greet.h
    $(CC) $(CFLAGS) -c main.c

greet.o: greet.c greet.h
    $(CC) $(CFLAGS) -c greet.c
```

Now run:

```
% make
```

Both object files are rebuilt as they should be.

What You Learned This is a small example of a very real problem. Without correct dependencies, `make` may skip rebuilding files, leaving you with inconsistent results that are extremely hard to debug. In driver development this becomes critical: kernel headers change frequently, and if your Makefile doesn't track them, your module may compile but crash the system.

This lab teaches the **real “aha!” moment**: `make` is not just convenience, it is your safeguard against subtle, inconsistent builds.

4.16.6 Recap Quiz: Why make Matters

1. If you change a header file but your Makefile does not list it as a dependency, what happens when you run `make`?
 - a) All source files are automatically rebuilt.
 - b) Only the files you changed are rebuilt.
 - c) `make` may skip rebuilding, leaving your program out of sync.
2. Why is it risky to have an out-of-sync build in FreeBSD driver development?
 - a) The program may still run but silently ignore your new code.
 - b) The kernel module may load but behave unpredictably, potentially crashing the system.
 - c) Both of the above.

3. In a Makefile, why do we use variables like `CFLAGS` instead of hardcoding flags in every rule?
 - a) It keeps the file shorter but less flexible.
 - b) It makes it easier to change compiler options in one place.
 - c) It forces you to type more at the command line.
4. What is the main advantage of `make` compared to running `cc` by hand?
 - a) `make` compiles faster.
 - b) `make` ensures only the necessary files are rebuilt, keeping everything consistent.
 - c) `make` automatically fixes logic errors in your code.

4.16.7 Debugging with GDB: more than just “fixing bugs”

When your program crashes or behaves strangely, it is tempting to sprinkle `printf` statements everywhere and hope the output tells you the story. That can work for very small programs, but it quickly becomes messy and unreliable. What if the bug happens only once in fifty runs? What if printing the variable actually changes the timing and hides the problem?

This is where **GDB**, the GNU Debugger, becomes your best ally. A debugger does more than help you “fix bugs.” It teaches you how your program *really* runs, step by step, and helps you build an accurate mental model of execution. With GDB, you are no longer guessing from the outside, you are looking inside the program while it runs.

Here are some of the things you can do with GDB:

- **Set breakpoints** at functions or even at specific lines, so the program pauses exactly where you want to observe it.
- **Inspect variables and memory** at that instant, seeing both values and addresses.
- **Step through code line by line**, watching how control flow really moves.
- **Check the call stack**, which tells you not only where you are, but *how you got there*.
- **Watch variables change over time**, a direct way to confirm whether your logic matches your expectations.

For example, imagine your calculator program suddenly prints the wrong result for multiplication. With GDB, you can stop inside the `multiply` function, look at the input values, then step through the lines. You may discover that the function is accidentally adding instead of multiplying. The compiler could not warn you; syntactically, the code was fine. But GDB shows you the truth.

In user space, this is already helpful. In kernel space, it becomes essential. Drivers are often asynchronous, event-driven, and far harder to debug with `printf`. Later in this book you will learn to use **kgdb**, the kernel version of GDB, to step through drivers, inspect kernel memory, and analyse crashes. By learning the GDB workflow now, you are building reflexes that will carry directly into your FreeBSD driver work.

Common beginner pitfalls with GDB:

- Forgetting to compile with `-g`, which leaves the debugger with no source-level information.
- Using optimisations (`-O2`, `-O3`) while debugging. The compiler may rearrange or inline code, making the debugger’s view confusing. Use `-O0` for clarity.

- Expecting GDB to fix logic errors automatically. A debugger does not correct code; it helps *you* see what the code is really doing.
- Quitting too early. Often, the first bug you notice is only a symptom. Use the stack trace to follow the chain of calls back to the real source.

Why this matters for driver development: Kernel code has little tolerance for trial-and-error debugging. A careless mistake can crash the entire system. GDB trains you to be methodical: set breakpoints, inspect state, confirm assumptions, and step carefully. By the time you reach kernel debugging with kgdb, this disciplined approach will feel natural, and it will be the difference between hours of frustration and a clear path to the fix.

4.16.8 Hands-On Lab 2: Fixing a Logic Bug with GDB

Objective: Learn how to use GDB to *step through* a program and find a logic error that compiles fine but produces the wrong result.

Step 1 - Write a buggy program.

Create a file called `math.c`:

```
#include <stdio.h>

int multiply(int a, int b) {
    return a + b;    // BUG: this adds instead of multiplies
}

int main(void) {
    int result = multiply(3, 4);
    printf("Result = %d\n", result);
    return 0;
}
```

At a glance, this looks fine; it compiles without warnings. But the logic is wrong.

Step 2 - Compile with debugging support.

Run:

```
% cc -Wall -g -O0 -o math math.c
```

- `-Wall` turns on useful warnings.
- `-g` includes extra information so GDB can show you the original source code.
- `-O0` disables optimisations, which keeps the code structure simple for debugging.

Step 3 - Run the program normally.

```
% ./math
```

Output:

```
Result = 7
```

Clearly, $3 * 4$ is not 7, we have a hidden logic bug.

Step 4 - Start GDB.

```
% gdb ./math
```

This opens your program inside the GNU Debugger. You'll see a (gdb) prompt.

Step 5 - Set a breakpoint.

Tell GDB to pause when it enters the multiply function:

```
(gdb) break multiply
```

Step 6 - Run the program under GDB.

```
(gdb) run
```

The program starts, but pauses as soon as multiply is called.

Step 7 - Step through the function.

Type:

```
(gdb) step
```

This moves into the first line of the function.

Step 8 - Inspect the variables.

```
(gdb) print a  
(gdb) print b
```

GDB shows you the values of a and b: 3 and 4.

Now type:

```
(gdb) next
```

This executes the buggy line `return a + b;`.

Step 9 - Spot the problem.

Look back at the code: multiply is returning `a + b` instead of `a * b`. The compiler could not warn you, because both are valid C. But GDB let you *see the bug in action*.

Step 10 - Fix and recompile.

Correct the function:

```
int multiply(int a, int b) {  
    return a * b;    // fixed  
}
```

Recompile:

```
% cc -Wall -g -O0 -o math math.c  
% ./math
```

Output:

```
Result = 12
```

Bug fixed!

Why this matters:

This exercise shows you that not all bugs cause crashes. Some are *logic errors* that only a debugger (and your careful attention) can reveal. In kernel programming, this mindset matters: you cannot rely on warnings or `printf` alone.

4.16.9 Hands-On Lab 3: Chasing a Segmentation Fault with GDB

Objective: Experience how GDB helps you investigate a crash by stopping at the exact point where the program failed.

Step 1 - Write a buggy program.

```
crash.c  
#include <stdio.h>  
  
int main(void) {  
    int *p = NULL;           // a pointer with no valid target  
    printf("About to crash...\n");  
    *p = 42;                 // attempt to write to address 0  
    printf("This line will never be printed.\n");  
    return 0;  
}
```

Compile it with debug info:

```
% cc -Wall -g -O0 -o crash crash.c
```

Step 2 - Run without GDB.

```
% ./crash
```

Output:

```
About to crash...
Segmentation fault (core dumped)
```

You know the program crashed, but not *where* or *why*. This is the frustration most beginners face.

Step 3 - Investigate with GDB.

```
% gdb ./crash
```

Inside GDB:

```
(gdb) run
```

The program will crash again, but this time GDB will stop at the exact line that caused the fault. You should see something like:

```
Program received signal SIGSEGV, Segmentation fault.
0x0000000000401136 in main () at crash.c:7
7          *p = 42;    // attempt to write to address 0
```

Now ask GDB to show the backtrace:

```
(gdb) backtrace
```

This confirms that the crash happened in `main` at line 7, with no deeper calls.

Step 4 - Reflect and fix.

The debugger shows you the *real cause*: you dereferenced a NULL pointer. This is one of the most common mistakes in C and one of the most dangerous in kernel code. To fix it, you would need to make sure `p` points to valid memory before using it.

Note: Without GDB you would only see “Segmentation fault,” leaving you to guess. With GDB, you immediately see the exact line and cause. In kernel programming, where a single NULL pointer can crash the whole operating system, this skill is essential.

4.16.10 Recap Quiz: Debugging Segmentation Faults with GDB

1. When you run a buggy program outside of GDB and it crashes with a **segmentation fault**, what information do you usually get?

- a) The exact line of code that failed.
 - b) Only that a segmentation fault occurred, without details.
 - c) A list of variables that caused the crash.
2. In the lab, why did the crash occur?
- a) The pointer `p` was set to `NULL` and then dereferenced.
 - b) The compiler miscompiled the program.
 - c) The operating system refused to allow printing to the screen.
3. How does running the program inside GDB improve your ability to diagnose the crash?
- a) GDB automatically fixes the bug.
 - b) GDB pauses execution at the exact line of failure and shows the call stack.
 - c) GDB re-runs the program until it succeeds.
4. Why is mastering this debugging habit especially important for FreeBSD driver development?
- a) Because kernel bugs are harmless.
 - b) Because a single invalid pointer can crash the entire operating system.
 - c) Because drivers never use pointers.

4.16.11 A peek into real FreeBSD build rules

FreeBSD keeps the build system simple at the surface and flexible underneath. Your module's `Makefile` can be only a few lines because the heavy lifting is centralised in shared rules.

4.16.11.1 The include chain you are stepping into

When you write:

```
KMOD= hello
SRCS= hello.c

.include <bsd.kmod.mk>
```

`bsd.kmod.mk` looks very small, but it is the gateway into the kernel's build system. On a FreeBSD 14.3 system it does three important things:

1. Optionally includes `local.kmod.mk` so you can override defaults without editing system files.
2. Includes `bsd.sysdir.mk`, which resolves the kernel source directory in the variable `SYSDIR`.
3. Includes `${SYSDIR}/conf/kmod.mk`, where the real kernel module rules live.

This indirection is deliberate. Your tiny `Makefile` defines *what* you want to build. The shared `mk` files decide *how* to build it, using the same flags, headers, and link commands that the rest of the kernel uses. This ensures your module is always treated like a first-class citizen of the system.

4.16.11.2 Setting up your own module directory

To follow along, create a small working directory in your home folder, for example `~/hello_kmod`, and place inside it the following two files:

File: `hello.c`

```
#include <sys/param.h>
#include <sys/module.h>
#include <sys/kernel.h>
#include <sys/system.h>

static int
hello_modevent(module_t mod __unused, int event, void *arg __unused)
{
    switch (event) {
        case MOD_LOAD:
            printf("Hello, kernel world!\n");
            break;
        case MOD_UNLOAD:
            printf("Goodbye, kernel world!\n");
            break;
        default:
            return (EOPNOTSUPP);
    }
    return (0);
}

static moduledata_t hello_mod = {
    "hello",
    hello_modevent,
    NULL
};

DECLARE_MODULE(hello, hello_mod, SI_SUB_DRIVERS, SI_ORDER_MIDDLE);
```

File Makefile:

```
KMOD= hello
SRCS= hello.c

.include <bsd.kmod.mk>
```

Whenever we refer to “*your module directory*”, this is the folder we mean: the one containing `hello.c` and `Makefile`.

4.16.11.3 A guided tour of the build system

Now run the following commands from inside that directory:

- **See where the kernel sources are:**

```
% make -V SYSDIR
```

This shows the kernel source directory being used. If it does not match your running kernel, fix this before continuing.

- **Reveal the actual build commands:**

```
% make clean
% make -n
```

The `-n` option prints the compile and link commands without running them. You will see the full `cc` invocations with all the flags and include paths that `kmod.mk` added for you.

- **Inspect important variables:**

```
% make -V CFLAGS
% make -V KMODDIR
% make -V SRCS
```

These tell you which flags are being passed, where your `.ko` will be installed, and which source files are included.

- **Try a local override:** Create a file called `local.kmod.mk` in the same directory:

```
DEBUG_FLAGS+= -g
```

Rebuild and notice that `-g` now appears in the compile lines. This demonstrates how you can customise the build without touching system files.

- **Test dependency tracking:** Add a second source file `helper.c`, include it in `SRCS`, and build. Then modify only `helper.c` and run `make` again. Notice that only that file recompiles before the linker runs. The same mechanism scales to the hundreds of files in the FreeBSD kernel.

4.16.11.4 What you will find in `${SYSDIR}/conf/kmod.mk`

When you include `<bsd.kmod.mk>` in your tiny `Makefile`, you are not telling the compiler how to build anything yourself. Instead, you are delegating the “how” to a much larger set of rules that live in the FreeBSD source tree under:

```
${SYSDIR}/conf/kmod.mk
```

This file is the **blueprint** for building kernel modules. It encodes decades of careful adjustments, ensuring that every module, whether written by you or a long-time committer, is built in exactly the same way.

If you open it, do not try to read every line. Instead, look for patterns that match the concepts you have just learned:

- **Compilation rules:** you will see how each `*.c` in `SRCS` is transformed into an object file, and how `CFLAGS` are extended with kernel-specific defines and include paths. This is why you did not have to manually add `-I/usr/src/sys` or worry about the proper warning levels, the system did it for you.
- **Linking rules:** deeper down, you will find the recipe that takes all those object files and links them into a single `.ko` kernel object. This is the equivalent of the final `cc -o app ...` you ran for user programs, but tuned for the kernel environment.
- **Dependency tracking:** the file also generates `.depend` information so that `make` knows exactly which files to rebuild when headers or sources change. This is the mechanism that protects you from “phantom bugs” caused by stale object files.

You do not need to memorise these rules, and you certainly do not need to re-implement them. But skimming through them once is valuable: it shows you that your three-line `Makefile` really expands into dozens of careful steps handled on your behalf.

Tip: A fun experiment is to run `make -n` in your module directory. This prints all the real commands that `kmod.mk` generates. Compare those with what you see in the file; it will help you connect theory (the rules) with practice (the commands).

By doing this, you begin to see the FreeBSD philosophy in action: developers declare *what* they want (`KMOD=hello`, `SRCS=hello.c`), and the build system decides *how* it is done. This ensures consistency across the whole kernel, whether the module is written by a beginner following this book or by a maintainer working on a complex driver.

4.16.11.5 Philosophy in practice

The FreeBSD build system is not just a technical detail; it reflects a philosophy that runs through the whole project.

- **Small, declarative Makefiles.** Your module’s `Makefile` only says *what* you want to build: its name and its source files. You do not need to describe every compiler flag or linker step. Beginners can focus on the logic of the driver itself, not the plumbing.
- **Centralised rules.** The real “how” is handled once, in the shared makefiles under `/usr/share/mk` and `${SYSDIR}/conf`. By putting the rules in one place, FreeBSD ensures that every driver, yours included, is built with the same standards. If the toolchain changes, or a flag needs to be added, it is fixed centrally and you benefit automatically.
- **Consistency above all.** The same approach builds the entire system: the kernel, userland programs, libraries, and your small module. This means there is no “special case” for beginners. The tiny “hello” module you wrote in this chapter is built in the same way as complex network drivers in the tree. That consistency gives confidence: if your module builds cleanly, you know it has passed through the same process as the rest of the system.

Imagine the alternative: every developer inventing their own build rules, each with slightly different flags or paths. Some modules would compile one way, others another. Debugging would be harder, maintenance nearly impossible. FreeBSD avoids this chaos by giving everyone a shared foundation.

By leaning on this infrastructure, you are not cutting corners; you are standing on decades of accumulated best practices. This is why FreeBSD code often looks “cleaner” than its equivalents elsewhere: developers do not waste time reinventing build logic. They spend their energy on correctness, clarity, and performance.

4.16.11.6 Common Beginner Pitfalls with FreeBSD Build Rules

Even with FreeBSD’s clean build system, beginners often trip over the same issues. Let’s look at them more closely.

- 1. Forgetting to install the kernel sources.** FreeBSD does not ship the full source tree by default. If you try to build a module without `/usr/src` installed, you will get confusing errors like “*sys/module.h: No such file or directory.*” Always make sure you have the matching source tree for your kernel (for example, `releng/14.3`).
- 2. Misusing relative paths in Makefiles.** Beginners sometimes write `SRCS= ../otherdir/helper.c` or try to pull files from odd locations. While it may compile, it breaks consistency and can confuse `make clean` or dependency tracking. Keep your module self-contained in its own directory, or use proper include paths via `CFLAGS+= -I...` if you must share headers.
- 3. Hardcoding output locations.** Some newcomers try to force the `.ko` file into `/boot/modules` by hand, or set absolute paths in the Makefile. This breaks the standard `make install` workflow and can overwrite system files. Always let the build system decide the correct `KMODDIR`. Use `make install` to put your module in the right place.
- 4. Mixing userland headers with kernel headers.** It’s tempting to `#include <stdio.h>` or other `libc` headers in your kernel code, but kernel modules cannot use the C library. If you need printing, use `printf()` from `<sys/system.h>`, not `<stdio.h>`. Mixing headers compiles sometimes, but fails at link or load time.
- 5. Not cleaning before switching branches or kernel versions.** If you update your kernel or switch source trees but reuse the same object files, you may get baffling build errors. Always run `make clean` when switching environments, to avoid stale objects from polluting your build.

With this knowledge, you can now see why FreeBSD’s module Makefiles are so short yet so effective. The system does the heavy lifting, and your job is simply to avoid working *against* it. In the next chapters, when you start writing real drivers, this consistency will save you time and frustration, letting you focus on device logic rather than boilerplate.

4.16.11.7 Reflection Quiz: FreeBSD Build Rules

1. Why can a FreeBSD kernel module Makefile be as short as three lines?
 - a) Because kernel modules don’t need compiler flags.
 - b) Because `bsd.kmod.mk` pulls in all the rules and flags from the kernel build system.
 - c) Because the kernel automatically guesses how to compile your files.
2. What does the variable `SYSDIR` represent when you run `make -V SYSDIR` inside your module directory?
 - a) The system log directory.
 - b) The path to the kernel source tree used for building modules.
 - c) The folder where `.ko` files are installed.

3. Why is it dangerous to edit files under `/usr/share/mk` or `${SYSDIR}` directly?
 - a) Because they are read-only on all FreeBSD systems.
 - b) Because your changes will be lost on the next update and may break consistency.
 - c) Because the compiler will refuse to use modified rules.
4. If you want to add debugging flags (like `-g`) without touching system files, what is the recommended way?
 - a) Add them directly into `/usr/share/mk/bsd.kmod.mk`.
 - b) Set them in a local `local.kmod.mk` file.
 - c) Recompile the kernel with the `-g` flag built in.
5. What is the core philosophy behind FreeBSD's build system?
 - a) Every developer writes their own Makefiles in full detail.
 - b) Boilerplate is avoided by centralising rules, so all components are built consistently.
 - c) Drivers are compiled manually with `cc` and then hand-linked into the kernel.

4.16.12 Common beginner pitfalls

Working with FreeBSD's build system is straightforward once you understand the rules. But newcomers often fall into a few traps that can waste hours of debugging time. Let's look at the most common ones, with an eye on how they show up in real FreeBSD development.

Ignoring compiler warnings. It is easy to shrug off warnings when the program still compiles. In user space, this is already risky; in kernel space, it can be catastrophic. A missing prototype, an implicit type conversion, or a discarded return value can all compile but still lead to unpredictable behaviour at runtime. FreeBSD's own build system is configured to treat many warnings as fatal, precisely because the project culture assumes "a warning today is a crash tomorrow." During your learning, adopt the same discipline: use `-Wall -Werror` so that warnings stop you early.

Mixing compiler and linker errors. Many beginners blur the line between these two. Compiler errors mean the C code is not valid in isolation; linker errors mean the different parts of the program do not agree. In kernel module development this distinction matters. For example, you might declare a driver method in a header but forget to provide its implementation. The compiler will happily generate an object file; the linker will later complain with "undefined reference." Understanding that difference quickly points you to whether the problem is *in one file* or *across files*.

Incomplete rebuilds leading to "phantom" bugs. When your program is more than one file, it is tempting to recompile just the file you edited. But headers ripple changes into many files, and if you do not rebuild all of them, you can end up with an inconsistent module. The bug might vanish when you do a clean build, which makes it especially frustrating. The FreeBSD kernel build system avoids this with precise dependency tracking, but if you are experimenting in your own directory, remember to use `make clean`; `make` when in doubt. Phantom bugs are almost always build artefacts.

Mismatched kernel and headers. This is the most FreeBSD-specific trap. Kernel modules must be built against the exact headers of the kernel they are loaded into. If you compile a `.ko` against FreeBSD 14.2 headers and try to load it into a 14.3 kernel, it may load with missing symbols, fail with cryptic errors, or even cause a crash. This mismatch is subtle because the compiler does not know your running kernel version, it will happily

compile. Only at `kldload` time will the system refuse or, worse, misbehave. Always ensure make `-V SYSDIR` points to the same source tree as your running kernel. This is why, earlier, we insisted you check out the `releng/14.3` sources when working through this book.

An example of mismatched headers in action

Suppose you have a FreeBSD 14.3 system running a kernel built from the **14.3 release branch**. You then check out the main branch of the source tree (which may already contain 15.0 development changes) and try to build your hello kernel module against it. The code may compile without errors:

```
% cd ~/hello_kmod
% make clean
% make
```

This produces `hello.ko` as usual. But when you try to load it:

```
% sudo kldload ./hello.ko
```

you may see an error like:

```
linker_load_file: /boot/modules/hello.ko - unsupported file layout
kldload: can't load ./hello.ko: Exec format error
```

or, in more subtle cases:

```
linker_load_file: symbol xyz not found
```

This happens because the module was compiled against headers that declare kernel structures or functions differently from the kernel you are actually running. The compiler cannot detect this mismatch, because both sets of headers are “valid” C, but they no longer agree with your kernel’s ABI.

The fix is simple but essential: always build modules against the matching source tree. For FreeBSD 14.3, clone the release branch:

```
% sudo git clone --branch releng/14.3 --depth 1 https://git.FreeBSD.org/src.git
↪ /usr/src
```

Then confirm that your module build points to it:

```
% make -V SYSDIR
```

This should print `/usr/src/sys`. If it does not, adjust your environment or symlink so that the build system uses the correct headers.

4.16.12.1 Hands-On Lab: Experiencing Common Pitfalls

Objective: See what happens when you fall into some of the classic traps we just discussed and learn how to avoid them.

Step 1 - Ignoring a warning.

Create `warn.c`:

```
#include <stdio.h>

int main(void) {
    int x;
    printf("Value: %d\n", x); // using uninitialized variable
    return 0;
}
```

Compile with warnings enabled:

```
% cc -Wall -o warn warn.c
```

You'll see:

```
warning: variable 'x' is uninitialized when used here
```

Run it anyway:

```
% ./warn
```

Output might be `Value: 32767` or some random number, because `x` contains garbage.

Lesson: Warnings often point to *real bugs*. Treat them seriously.

Step 2 - A linker error from missing function.

Create `main.c`:

```
#include <stdio.h>

void greet(void); // declared but not defined

int main(void) {
    greet();
    return 0;
}
```

Compile:

```
% cc -o test main.c
```

The compiler is happy, but the linker fails:

```
undefined reference to 'greet'
```

Lesson: The compiler only checks syntax. The linker enforces that declared symbols must exist somewhere.

Step 3 - Header mismatch (phantom bug demo).

1. Create util.h:

```
int add(int a, int b);
```

2. Create util.c:

```
#include "util.h"
int add(int a, int b) { return a + b; }
```

3. Create main.c:

```
#include <stdio.h>
#include "util.h"

int main(void) {
    printf("%d\n", add(2, 3));
    return 0;
}
```

Compile:

```
% cc -Wall -c util.c
% cc -Wall -c main.c
% cc -o demo main.o util.o
% ./demo
```

Output: 5

Now change util.h to:

```
int add(int a, int b, int c); // prototype changed
```

And also update main.c to match:

```
#include <stdio.h>
#include "util.h"

int main(void) {
    printf("%d\n", add(2, 3, 4)); // now passing 3 arguments
    return 0;
}
```

But don't touch `util.c` (it still only accepts two parameters). Now recompile only the changed files:

```
% cc -Wall -c main.c          # recompiles fine - header matches the call
% cc -o demo main.o util.o    # links without error!
% ./demo
```

Result: Now `./demo` may run incorrectly, print garbage, crash, or appear to work by accident, because `main.c` thinks `add()` takes three parameters, but `util.o` was compiled with a function that only takes two.

Lesson: Partial rebuilds can leave object files out of sync with headers, creating phantom bugs. The linker doesn't verify that function signatures match between compilation units, only that the symbol exists. This is why we use make with proper dependency tracking.

4.16.12.2 Recap Quiz: Common Pitfalls

1. Why should you never ignore compiler warnings in kernel development?
 - a) They are harmless.
 - b) They often signal real bugs that can crash the system.
 - c) They only matter in user space.
2. If you declare a function in a header but forget to provide its definition, which stage will fail?
 - a) Preprocessor.
 - b) Compiler.
 - c) Linker.
3. What is a "phantom bug" and how does it usually appear?
 - a) A bug caused by cosmic rays.
 - b) A bug that disappears after a full rebuild because old object files were out of sync with changed headers.
 - c) A bug in the debugger itself.
4. Why is it important to build kernel modules with the correct version of the FreeBSD source tree?
 - a) To get the newest features.
 - b) Because mismatched headers can make modules fail to load or behave unpredictably.
 - c) It does not matter; headers are always backward compatible.

4.16.13 Why this matters for FreeBSD driver development

Writing drivers is not like writing toy programs from a tutorial. Real drivers are made up of many source files, depend on a maze of kernel headers, and must be rebuilt every time the kernel changes. If you tried to manage this manually with raw `cc` commands, you would drown in complexity. That is why FreeBSD's build system, and your understanding of it, is so important.

When you learn to read compiler and linker messages carefully, you are not just fixing mistakes, you are building the habit of interpreting what the system is telling you. This habit will pay off when your driver spans multiple files, each depending on kernel interfaces that may change from one release to the next.

Debugging is another area where beginners often carry the wrong reflex. Sprinkling `printf` everywhere works in user space for very small programs, but in kernel space it can distort timing, slow down interrupts, or even mask the bug you are trying to chase. FreeBSD gives you better tools: symbolic debuggers, kernel core dumps, and `kgdb`. The way you practiced with `gdb` in this chapter is training for the same workflow you will later use on real drivers controlled, step-by-step, with the ability to look inside the system rather than guess from the outside.

Takeaway: Every habit you built here, compiling with warnings, using `make` to manage dependencies, reading errors with care, and debugging with intent, is not just academic. These are the working muscles of a FreeBSD driver developer.

4.16.14 Wrapping up

In this section, you moved beyond the one-line `cc hello.c` and began thinking like a system builder. You saw that a pipeline of stages produces programs, that compiler and linker errors tell different stories, that `make` keeps complex projects consistent, and that debuggers let you see code *as it really runs*.

With these foundations in place, you are ready to explore the **C Preprocessor**, the very first stage of the pipeline. This is where `#include`, `#define`, and conditional compilation reshape your source before the compiler ever sees it. Understanding this stage will explain why almost every FreeBSD kernel header begins with a dense block of preprocessor directives, and it will prepare you to read and write them with confidence.

4.17 The C Preprocessor: Directives Before Compilation

Before your C code is compiled, it makes a stop at an earlier stage called **preprocessing**. You can think of this as the **pre-flight checklist for your program**: it prepares the source, pulls in external declarations, expands macros, removes comments, and decides which parts of the code will even reach the compiler. In practice, this means the compiler never works directly on the file you wrote. Instead, it sees a transformed version that has already been “cleaned up” and adapted by the preprocessor.

This may sound like a small detail, but it has a huge impact. The preprocessor explains why a simple driver file in FreeBSD often starts with a long series of `#include` and `#define` lines before any actual function appears. It is also the reason the **same kernel source code can build successfully on different CPU architectures, with or without debugging support, and with optional subsystems like networking or USB enabled**.

For a beginner, the preprocessor is your first taste of how C code adapts itself to different situations without needing to maintain dozens of separate files. For a kernel developer, it is a daily tool that makes large projects like FreeBSD manageable, modular, and portable.

4.17.1 What the Preprocessor Does

The C preprocessor is best thought of as a **text substitution engine**. It does not understand C syntax, data types, or even whether what it produces makes logical sense. Its only job is to transform the source text according to the directives it finds, all before the compiler ever sees the file.

Here are its main responsibilities:

- **Including headers** so you can reuse declarations, macros, and constants from other files.
- **Defining constants and macros** that the compiler will see as if they were literally typed into the source code.
- **Conditional compilation** that allows sections of code to be included or excluded depending on configuration.
- **Preventing duplicate inclusion** of the same header, which avoids errors and keeps compilation efficient.

Because the preprocessor works purely at the text level, it is both **flexible and risky**. Used wisely, it makes your code portable, configurable, and easier to maintain. Used carelessly, it can lead to cryptic error messages and behaviour that is difficult to debug.

4.17.2 A Quick Example

Here is a tiny program that shows how the preprocessor can change behaviour without touching the logic of `main()`:

```
#include <stdio.h>

#define DEBUG    /* Try commenting this line out */

int main(void) {
    printf("Program started.\n");

#ifdef DEBUG
    printf("[DEBUG] Extra information here.\n");
#endif

    printf("Program finished.\n");
    return 0;
}
```

Compile and run it normally, then recompile after commenting out the `#define DEBUG`.

What to Observe:

- The `main()` function is identical in both cases.
- The only difference is whether the preprocessor includes or skips the `printf("[DEBUG] ...");` line.
- This demonstrates the central idea: **the preprocessor decides what code the compiler even gets to see.**

In FreeBSD drivers, this exact pattern is everywhere. Developers often wrap diagnostic output in preprocessor checks. For example, macros like `DPRINTF()` or `device_printf()` are enabled only when a debug flag is set. This means the same driver source code can produce a “quiet” production build or a “verbose” debugging build without changing the actual driver logic.

4.17.3 Real Example from FreeBSD 14.3 Source

In `sys/dev/usb/usb_debug.h` you will find the canonical USB debug pattern used across USB drivers:

```
/* sys/dev/usb/usb_debug.h */

#ifndef _USB_DEBUG_H_
#define _USB_DEBUG_H_

/* Declare global USB debug variable. */
extern int usb_debug;

/* Check if USB debugging is enabled. */
#ifdef USB_DEBUG_VAR
#ifdef USB_DEBUG
#define DPRINTFN(n, fmt, ...) do { \
    if ((USB_DEBUG_VAR) >= (n)) { \
        printf("%s: " fmt, __FUNCTION__ , ##__VA_ARGS__); \
    } \
} while (0)
#define DPRINTF(...) DPRINTFN(1, __VA_ARGS__)
#define __usbdebug_used
#else
#define DPRINTF(...) do { } while (0)
#define DPRINTFN(...) do { } while (0)
#define __usbdebug_used __unused
#endif
#endif

/* ... later in the same header ... */

#ifdef USB_DEBUG
extern unsigned usb_port_reset_delay;
/* more externs... */
#else
#define usb_port_reset_delay USB_PORT_RESET_DELAY
```

```
/* more compile-time constants... */  
#endif  
  
#endif /* _USB_DEBUG_H_ */
```

How this works, in plain language:

- Drivers that want USB debug output define a macro `USB_DEBUG_VAR` to name their debug-level variable, most often `usb_debug`.
- If you also compile with `USB_DEBUG` defined, `DPRINTF(...)` and `DPRINTFN(level, ...)` expand to `printf` calls that prefix messages with the current function name. If `USB_DEBUG` is not defined, both macros expand to do-nothing statements and vanish from the final binary.
- The second block shows another common pattern: when `USB_DEBUG` is defined you get tunable variables exported as `extern unsigned` so you can tweak timings while testing. When it is not defined, the same names become compile-time constants like `USB_PORT_RESET_DELAY`.

What to Observe:

- Everything is controlled by **preprocessor switches**, not by runtime `if` statements. When `USB_DEBUG` is off, the debug code is not even compiled.
- `USB_DEBUG_VAR` gives each driver a simple knob: raise it to see more messages, lower it to stay quiet.
- This mirrors the “Quick Example” you compiled earlier, but in a real subsystem that many drivers rely on.

Try It Yourself: A driver source file that wants debug logging would add:

```
#define USB_DEBUG_VAR usb_debug  
#include <dev/usb/usb_debug.h>
```

If you build that driver with `-DUSB_DEBUG`, the `DPRINTF()` calls inside it will print messages. If you build without `-DUSB_DEBUG`, the same calls vanish from the compiled binary. This is the preprocessor in action, shaping how the same code behaves depending on build options.

4.17.4 Common Preprocessor Directives

Now that you know what the preprocessor does, let us look at the most common directives you will encounter. Each one begins with the `#` symbol and is processed before the compiler sees your code.

4.17.4.1 1. `#include` - Bringing in Header Files

The most visible directive is `#include`. It literally copies the contents of another file into your source before compilation.

```
#include <stdio.h>      /* System header */
#include "myheader.h"    /* Local header */
```

Angle brackets (< >) tell the preprocessor to look in the system's include directories, while quotes (" ") tell it first to search the current directory.

In FreeBSD, every kernel source file begins with `#include` lines. For example, most device drivers start with:

```
#include <sys/param.h>
#include <sys/bus.h>
#include <sys/kernel.h>
```

- `<sys/param.h>` brings in constants such as system version and limits.
- `<sys/bus.h>` defines the bus infrastructure that almost all drivers depend on.
- `<sys/kernel.h>` provides kernel-wide symbols and macros.

Without `#include`, you would need to manually copy declarations into every driver file, which would quickly become unmanageable.

4.17.4.2 2. `#define` - Creating Macros and Constants

`#define` is used to create symbolic names or small code fragments that are replaced before compilation.

```
#define BUFFER_SIZE 1024
#define MAX(a, b) ((a) > (b) ? (a) : (b))
```

Macros are not variables. They are direct text replacements. This makes them fast, but also prone to subtle bugs if parentheses are forgotten.

In FreeBSD, if you look at `sys/sys/ttydefaults.h`, you will find:

```
#define TTYDEF_IFLAG (BRKINT | ICRNL | IXON | IMAXBEL | ISTRIP)
#define CTRL(x)      ((x) & 0x1f)
```

Here `CTRL(x)` maps a character into its control-key equivalent. It is used throughout the terminal subsystem.

In device drivers, macros are often used for register offsets or bit masks:

```
#define REG_STATUS 0x04
#define STATUS_READY 0x01
```

This makes driver code much more readable than writing raw numbers everywhere.

4.17.4.3 3. `#undef` - Removing a Definition

Sometimes a macro is defined differently depending on build conditions. `#undef` removes a previous definition so it can be replaced.

```
#undef BUFFER_SIZE
#define BUFFER_SIZE 2048
```

This is less common in FreeBSD drivers, but it appears in portability code that must adapt to different compilers or architectures.

4.17.4.4 4. `#ifdef`, `#ifndef`, `#else`, `#endif` - Conditional Compilation

These directives decide whether a block of code should be compiled depending on whether a macro is defined.

```
#ifdef DEBUG
printf("Debug mode is on\n");
#endif
```

- `#ifdef` checks if a macro exists.
- `#ifndef` checks if it does *not* exist.

In FreeBSD, every header file in the kernel tree uses an *include guard* to prevent multiple inclusion. For example, at the top of `sys/sys/param.h`:

```
#ifndef _SYS_PARAM_H_
#define _SYS_PARAM_H_
/* declarations here */
#endif /* _SYS_PARAM_H_ */
```

Without this guard, including the same header twice would lead to duplicate definitions.

4.17.4.5 5. `#if`, `#elif`, `#else`, `#endif` - Numerical Conditions

These allow conditional compilation based on constant expressions.

```
#define VERSION 14

#if VERSION >= 14
printf("This is FreeBSD 14 or later\n");
#else
printf("Older version\n");
#endif
```

In FreeBSD kernel builds, numerical conditions are widely used to check for optional features. For instance, many files contain:

```
#if defined(INVARIANTS)
/* Add extra runtime checks */
#endif
```

The INVARIANTS flag enables additional sanity checks that help developers catch bugs during testing but are removed in production builds.

4.17.4.6 6. #error and #warning - Forcing Messages at Build Time

Sometimes you want to stop the build if a condition is not met.

```
#ifndef DRIVER_SUPPORTED
#error "This driver is not supported on your system."
#endif
```

This ensures that the driver will not even compile in an unsupported configuration.

In FreeBSD, build-time errors like this are common in sys/conf and device headers to prevent mismatched configurations. They provide an early warning before you even attempt to load a broken module.

4.17.4.7 7. Include Guards vs. #pragma once

Most FreeBSD headers use the #ifndef / #define / #endif pattern. Some modern compilers support #pragma once as a simpler alternative; however, FreeBSD maintains the portable style to ensure headers work on every compiler supported by the project.

Think of them as the switches and dials that configure how the compiler views your program. Without them, FreeBSD's single source tree could never build across dozens of architectures and feature sets.

4.17.5 Hands-On Lab 1: Protecting Headers with Include Guards

In large projects like FreeBSD, header files are included by many different source files. Without protection, including the same header twice can cause compilation errors. Including guards solves this problem.

Step 1. Create a header file called myheader.h:

```
/* myheader.h */
#ifndef MYHEADER_H
#define MYHEADER_H

#define GREETING "Hello from the header file!\n"

#endif /* MYHEADER_H */
```

Step 2. Create a source file called `main.c`:

```
#include <stdio.h>
#include "myheader.h"
#include "myheader.h"    /* included twice on purpose */

int main(void) {
    printf(GREETING);
    return 0;
}
```

Step 3. Compile and run:

```
% cc -o main main.c
% ./main
```

What to Observe:

- The program compiles and runs successfully even though the header was included twice.
- Without the `#ifndef` / `#define` / `#endif` pattern in `myheader.h`, you would get duplicate definition errors.
- This is exactly how FreeBSD headers like `<sys/param.h>` protect themselves.

4.17.6 Hands-On Lab 2: Enforcing Build Requirements with `#error`

In kernel development, it is often better to **fail early** than to build a driver that will not work. The preprocessor allows you to enforce conditions at compile time.

Step 1. Create a file named `require_version.c`:

```
#include <stdio.h>

#define KERNEL_VERSION 14

int main(void) {
    #if KERNEL_VERSION < 14
        #error "This code requires FreeBSD 14 or later."
    #endif

    printf("Building against FreeBSD version %d\n", KERNEL_VERSION);
    return 0;
}
```

Step 2. Compile it normally:


```
% cc -o require_version require_version.c
```

You should see a successful build.

Step 3. Change the line `#define KERNEL_VERSION 14` to 13 and try compiling again.

What to Observe:

- With `KERNEL_VERSION 14`, the build succeeds.
- With `KERNEL_VERSION 13`, the build fails immediately and prints the error message.
- This is the same pattern used in FreeBSD's build system to stop unsupported configurations before they reach runtime.

4.17.7 Common Beginner Pitfalls with the Preprocessor

The preprocessor is simple and flexible, but it can also create subtle bugs if used carelessly. These are the issues beginners most often face when writing C and especially when moving into kernel code.

Forgetting that macros are plain text. A macro does not behave like a function or a variable. It is copied into your code as text. This easily goes wrong if you skip parentheses:

```
#define SQUARE(x) x*x      /* buggy */
int a = SQUARE(1+2);      /* expands to 1+2*1+2 -> 5, not 9 */

#define SQUARE_OK(x) ((x)*(x)) /* safe */
```

Multi-statement macros that misbehave. A macro with more than one statement can break `if/else` logic unless wrapped properly:

```
#define LOG_BAD(msg) printf("%s\n", msg); counter++;

#define LOG_OK(msg) do { \
    printf("%s\n", msg); \
    counter++;           \
} while (0)
```

Always prefer the `do { ... } while (0)` pattern so the macro behaves like a single statement.

Side effects in macro arguments. Arguments may be evaluated more than once, which can cause surprising results:

```
#define MAX(a,b) ((a) > (b) ? (a) : (b))
int i = 0;
int m = MAX(i++, 10); /* i++ may execute even when not needed */
```

When side effects matter, a real function or an inline function is safer.

Overusing macros where `const`, `enum`, or `inline` would be clearer. Prefer `const int size = 4096;`, `enum { BUFSZ = 4096 };`, or `static inline` functions when type safety or single evaluation is important. Use macros for bit masks, compile-time switches, or code that truly must be erased at build time.

Weak or missing include guards. Use a unique, file-based guard for every header:

```
#ifndef _DEV_FOO_BAR_H_
#define _DEV_FOO_BAR_H_
/* ... */
#endif
```

Avoid short or generic guard names that could clash across the tree.

Mixing userland and kernel headers. In kernel modules, do not include userland headers like `<stdio.h>`. Use kernel equivalents and follow kernel include order, for example:

```
#include <sys/param.h>
#include <sys/systm.h>
#include <sys/bus.h>
/* ... */
```

Using quotes vs angle brackets incorrectly. Use `#include "local.h"` for headers in your own directory and `#include <sys/param.h>` for system headers. In the kernel, most includes are system headers.

Sprinkling `#ifdef` everywhere. Scattered conditionals make code hard to read. Prefer centralising options in a small header (`opt_*.h` or a driver-local `config.h`). Keep long conditional blocks to a minimum and document what each flag controls.

Redefining build options manually. Kernel options are often generated into `opt_*.h` headers during the build. Do not hand-define them in source files. Let the build system provide those macros, and include the correct option header when needed.

Undocumented `-D` flags. If a module depends on a `-DDEBUG` or `-DUSB_DEBUG` flag, document it in the driver's README or the top of the source file. Future you will thank present you.

4.17.8 Why This Matters for FreeBSD Driver Development

The preprocessor is one of the main reasons a single FreeBSD source tree can target many CPUs, chipsets, and build profiles.

Portability across architectures. Conditionals select the right code paths for amd64, arm64, riscv, and others, without maintaining separate files.

Feature toggles without runtime cost. Flags like `INVARIANTS`, `WITNESS`, or subsystem-specific `*_DEBUG` macros enable deep checks and logs during development. In release builds, those blocks disappear so there is no performance penalty.

Readable hardware access. `#define` constants give meaningful names to register offsets and bit masks. This is essential for driver clarity and review.

Configuration headers as single sources of truth. A small header can control compile-time options for a driver. This keeps `#ifdefs` out of the main logic and makes the build behaviour explicit.

Fail early, fail clearly. `#error` helps stop unsupported builds before you reach a kernel panic. A precise build-time message is better than a mysterious runtime failure.

4.17.9 Recap Quiz: The C Preprocessor

1. What does the C preprocessor do with your source file before the compiler sees it?
 - a) Optimises the code for performance
 - b) Transforms the text according to directives like `#include` and `#define`
 - c) Converts C into assembly
2. Why do almost all FreeBSD header files begin with `#ifndef ... #define ... #endif`?
 - a) To make the file easier to read
 - b) To ensure the file is only included once, avoiding duplicate definitions
 - c) To reserve space in memory for the header
3. Which of these definitions of a macro is safer?
 - a) `#define SQUARE(x) x*x`
 - b) `#define SQUARE(x) ((x)*(x))`
4. If you see a driver that uses `#ifdef USB_DEBUG`, what does that mean?
 - a) USB debugging code will only be compiled if the macro is defined
 - b) The driver requires USB hardware to be attached
 - c) The driver is always compiled in debug mode
5. What happens if you compile this program?

```
#include <stdio.h>
#define VERSION 13

int main(void) {
    #if VERSION < 14
        #error "This driver requires FreeBSD 14 or later."
    #endif
    printf("Driver loaded.\n");
    return 0;
}
```

- a) It prints “Driver loaded.”
- b) It fails to compile and shows the error message
- c) It compiles but prints nothing

1. In FreeBSD drivers, why is it better to use macros like `REG_STATUS` or `STATUS_READY` instead of writing the raw numbers directly?
 - a) It makes the compiler run faster
 - b) It improves readability and maintainability of the code
 - c) It reduces memory usage

4.17.10 Wrapping Up

You have seen how the preprocessor sets the stage before the compiler arrives. It pulls in declarations, expands macros, and decides which code exists in a given build. Used with care, it makes your drivers portable, testable, and tidy. Used without discipline, it can hide bugs and make code difficult to reason about.

In the next section, **Good Practices for C Programming**, we will shift from mechanisms to habits. You will learn how to choose between macros and `inline`, how to name and document flags, how to structure includes in kernel code, and how to keep your driver readable for future contributors. We will also connect these practices with FreeBSD's coding style so your code feels at home in the tree.

4.18 Good Practices for C Programming

By now you know the basic building blocks of the C language and how they fit together. But writing code that simply compiles and runs is not the same as writing code that others can read, maintain, and trust inside an operating system kernel. In FreeBSD, style and clarity are not cosmetic details; they are part of what keeps the system stable and maintainable over decades.

Consistent code makes bugs easier to spot, reviews faster to complete, and long-term maintenance less painful. It also helps your own future self when you return to a file months later and need to understand what you were thinking.

Here we go beyond syntax and into **habits**. These habits may look small in isolation, choosing clear names, checking return values, keeping functions short, but together they shape code that fits naturally into the FreeBSD source tree. If you adopt them early, every driver you write will not only work, but also feel like part of FreeBSD itself.

Let us now explore these practices step by step, beginning with how to make your code readable through indentation, naming, and comments.

4.18.1 Readability First: Indentation, Names, and Comments

Programming is not just about instructing the computer; it is also about communicating with people. The next person who reads your code might be a FreeBSD maintainer reviewing your driver, or it might be you six months later trying to fix a bug you barely remember. Clear, consistent formatting and naming make that task easier and prevent misunderstandings that lead to mistakes.

4.18.1.1 Indentation

Indentation is like punctuation in writing: it does not change the meaning, but it makes the text easier to read and understand. In the FreeBSD kernel, code is formatted according to **KNF (Kernel Normal Form)**. KNF specifies details such as where braces go, how many spaces follow a keyword, and how to line up code blocks.

FreeBSD uses **tabs for indentation** and spaces only for alignment. This keeps code consistent across different editors and makes diffs smaller when changes are committed.

Good indentation example (KNF style):

```
static int
my_device_probe(device_t dev)
{
    struct mydev_softc *sc;

    sc = device_get_softc(dev);
    if (sc == NULL)
        return (ENXIO);

    return (0);
}
```

Notice how:

- The function name is on its own line, aligned under the return type.
- The opening brace { is placed on a new line.
- Each nested block is indented with a tab.

The result is a structure you can “see” at a glance.

4.18.1.2 Naming

Names should describe purpose. Computers do not care what you call things, but humans do. In kernel code, **self-explanatory names** make reviews smoother and reduce errors when different developers touch the same driver years apart.

- **Functions:** Use verbs and, in drivers, often a prefix for the driver or subsystem. Example: `uart_attach()`, `mydev_init()`.
- **Variables:** Use descriptive names, except for very short-lived counters in loops (`i`, `j` are fine there). Avoid meaningless letters like `f` or `x` for important values.

Unclear:

```
int f(int x) { return x * 2; }
```

Clearer:

```
int
double_value(int input)
{
    return (input * 2);
}
```

The second example requires no guesswork. The reader instantly knows what the function does.

4.18.1.3 Comments

A well-placed comment explains **why** code exists, not just what it does. If the code is already obvious, repeating it in a comment wastes space. Use comments to describe assumptions, tricky conditions, or design choices.

Example:

```
/*
 * The softc (software context) is allocated and zeroed by the bus.
 * If it is not available here, something went wrong in probe.
 */
sc = device_get_softc(dev);
if (sc == NULL)
    return (ENXIO);
```

The comment explains the reasoning, not the mechanics. Without it, a new reader might not know *why* failing here is the correct thing to do.

4.18.1.4 Why This Matters for Driver Development

Drivers live in the kernel tree for years. Many people will read and edit them. Reviewers have limited time, and confusing names, inconsistent indentation, or misleading comments slow them down and increase the risk of errors slipping through.

Readable code:

- Makes bugs easier to spot during review.
- Reduces merge conflicts by following the same indentation and naming conventions as everyone else.
- Helps future developers (including you) understand intent when debugging a crash at 3 a.m.

In FreeBSD, clarity and consistency are part of *correctness*.

4.18.1.5 Mini Lab: Make It Readable

Goal Take a small, messy function and turn it into code that a FreeBSD maintainer would enjoy reading. You will practice KNF indentation with tabs, clear naming, and comments that explain intent.

Starting file: `readable_lab.c`

```
#include <stdio.h>
```

```
int f(int x,int y){int r=0; for(int i=0;i<=y;i++){r=r+x;} if(r>100)printf(big\n);else  
↪ printf(%d\n,r);return r;}
```

What to do

1. Reformat the function to KNF style. Use tabs for indentation and keep braces and line breaks consistent with the example patterns you have seen.
2. Rename the function and variables so a new reader understands the purpose without reading comments first.
3. Add a short comment that explains why the loop starts where it starts and what the threshold check represents.
4. Keep the logic identical. This lab is about readability, not algorithm changes.

Compile and run

```
% cc -Wall -Wextra -o readable_lab readable_lab.c  
% ./readable_lab
```

You should see output when you later add a `main()` to test it. For now, focus on the function shape.

A clean result might look like this

```
#include <stdio.h>
```

```
/*  
 * Compute a repeated addition of value 'addend' exactly 'count + 1' times.  
 * We use <= in the loop to match the original behaviour.  
 * If the result crosses a simple threshold, print a note; otherwise print the value.  
 */  
int  
accumulate_and_report(int addend, int count)  
{  
    int total;  
    int i;  
  
    total = 0;  
  
    for (i = 0; i <= count; i++)  
        total = total + addend;  
  
    if (total > 100)  
        printf(big\n);  
    else
```

```
    printf("%d\n", total);

    return (total);
}
```

Checklist for yourself

- Function name states intent, not a single letter.
- Parameters and locals have descriptive names.
- Tabs used for indentation, spaces only for alignment.
- Braces and line breaks follow the KNF pattern shown earlier.
- Comment explains why, not what each line does.

Stretch test (optional, 3 minutes) Add a tiny `main()` that calls your function twice with different inputs. Confirm the output looks sensible.

```
int
main(void)
{
    accumulate_and_report(7, 5);
    accumulate_and_report(20, 3);
    return (0);
}
```

Compile and run:

```
% cc -Wall -Wextra -o readable_lab readable_lab.c
% ./readable_lab
```

What you learned Readable code is a sequence of small, consistent choices: names that tell a story, indentation that shows structure, and comments that capture intent. These habits make reviews faster and bugs easier to spot.

With readability in place, you are ready to apply the same discipline to error handling and small helper functions in the following subsection.

4.18.2 Small, Focused Functions and Early Returns

Short functions are easier to read, reason about, and test. In kernel code they also reduce the chance of subtle errors hiding inside long control paths.

Keep a single responsibility per function. If a function starts to branch into several tasks, split it into helpers. Use early returns for error cases so the happy path stays visible.

Before:


```

static int
mydev_configure(device_t dev, int flags)
{
    struct mydev_softc *sc;
    int err;

    sc = device_get_softc(dev);
    if (sc == NULL) {
        device_printf(dev, no softc\n);
        return (ENXIO);
    }

    /* allocate resources, set up interrupts, register sysctl, etc */
    err = mydev_alloc_resources(dev);
    if (err != 0) {
        device_printf(dev, alloc failed: %d\n, err);
        return (err);
    }
    if ((flags & 0x1) != 0) {
        err = mydev_optional_setup(dev);
        if (err != 0) {
            mydev_release_resources(dev);
            return (err);
        }
    }
    err = mydev_register_sysctl(dev);
    if (err != 0) {
        mydev_teardown_optional(dev);
        mydev_release_resources(dev);
        return (err);
    }
    /* lots more steps here... */
    return (0);
}

```

After: the same logic becomes a sequence of small, named steps.

```

static int mydev_setup_core(device_t dev);
static int mydev_setup_optional(device_t dev, int flags);
static int mydev_setup_sysctl(device_t dev);

static int
mydev_configure(device_t dev, int flags)
{

```

```
int err;

err = mydev_setup_core(dev);
if (err != 0)
    return (err);

err = mydev_setup_optional(dev, flags);
if (err != 0)
    goto fail_core;

err = mydev_setup_sysctl(dev);
if (err != 0)
    goto fail_optional;

return (0);

fail_optional:
    mydev_teardown_optional(dev);
fail_core:
    mydev_release_resources(dev);
    return (err);
}
```

You can now read the top of the function like a story. The cleanup is centralised and easy to audit.

Why this matters for drivers

Attach and detach paths can grow over time. Splitting them into helpers keeps diffs small and reduces the likelihood of merge conflicts. Centralized cleanup reduces leaks and makes failure paths predictable.

4.18.3 Check Return Values and Propagate Errors

In user programs, you can sometimes ignore a failed call, print a warning, and continue. At worst, your program crashes or produces wrong output. In the kernel, ignoring an error almost always causes **bigger problems later**. An unchecked failure might look harmless in one place but can turn into a panic or data corruption in a completely different subsystem hours later.

The rule in FreeBSD driver code is simple: **check every return value, handle the error, and propagate it up the call chain.**

4.18.3.1 The pattern

1. **Call the routine.**
2. **If it fails, log briefly** with the device as context. Do not spam the logs; one concise line is enough.
3. **Clean up** any resources you acquired.
4. **Return the error code** to the caller.

4.18.3.2 Example: Interrupt setup

```
int err;

err = bus_setup_intr(dev, sc->irq_res, INTR_TYPE_TTY,
    NULL, mydev_intr, sc, &sc->irq_cookie);
if (err != 0) {
    device_printf(dev, interrupt setup failed: %d\n, err);
    goto fail_irq;
}
```

This makes the failure explicit, prints context (dev), and leaves cleanup to the fail_irq: label.

4.18.3.3 What *not* to do

- **Ignore the result entirely:**

```
bus_setup_intr(dev, sc->irq_res, INTR_TYPE_TTY,
    NULL, mydev_intr, sc, &sc->irq_cookie); /* return value dropped */
```

This may work during testing, but will cause hard-to-trace bugs in production.

- **Clever chaining:**

```
if ((err = bus_setup_intr(dev, sc->irq_res, INTR_TYPE_TTY,
    NULL, mydev_intr, sc, &sc->irq_cookie)) &&
    (err = some_other_setup(dev)) &&
    (err = yet_another(dev))) {
    /* ... */
}
```

This hides which step failed and makes the logs confusing.

4.18.3.4 Practical tips for drivers

- **Keep error checks local.** Use the result right after the call, rather than collecting them and checking later.
- **Use the smallest scope.** Declare variables near their first use. Don't reuse one err variable for unrelated operations in long functions.
- **Be consistent.** Always return an error code (errno value) that matches FreeBSD conventions (for example, ENXIO when no hardware is found).
- **Log once.** Do not log repeatedly at every level of the call stack; let the highest-level function print the context.

4.18.3.5 Why this matters for FreeBSD drivers

Kernel drivers interact with hardware and subsystems that the rest of the operating system depends on. A single unchecked error can:

- Leave resources half-initialised.
- Cause crashes in unrelated code later.
- Waste hours in debugging, because the *first cause* of the failure was silently ignored.

By always checking and propagating errors, you make your driver predictable, debuggable, and trusted by the rest of the system.

4.18.4 Mini Lab: Check, Log, Clean Up, Propagate

Goal Take a “works on my machine” setup path that ignores return codes and turn it into FreeBSD-quality code that:

- checks every call,
- logs once with device context,
- unwinds resources in reverse order,
- returns an appropriate error code.

Starting file: `error_handling_lab.c`

```
#include <stdio.h>
#include <stdlib.h>

/* Simulated errno-style values (keep small and distinct) */
#define E_OK      0
#define E_RES    -1
#define E_IRQ    -2
#define E_SYS    -3

/* Pretend "device context" */
struct softc {
    int res_ready;
    int irq_ready;
    int sysctl_ready;
    const char *name;
};

/* Simulated setup steps; each may succeed (E_OK) or fail (E_*) */
static int
alloc_resources(struct softc *sc, int should_fail)
{
    (void)sc;
```

```
    return should_fail ? E_RES : E_OK;
}

static int
setup_irq(struct softc *sc, int should_fail)
{
    (void)sc;
    return should_fail ? E_IRQ : E_OK;
}

static int
register_sysctl(struct softc *sc, int should_fail)
{
    (void)sc;
    return should_fail ? E_SYS : E_OK;
}

/* Teardown in reverse order */
static void
teardown_sysctl(struct softc *sc)
{
    (void)sc;
}

static void
teardown_irq(struct softc *sc)
{
    (void)sc;
}

static void
release_resources(struct softc *sc)
{
    (void)sc;
}

/* TODO: Make this robust:
 * - Check return values
 * - Log briefly with device context (name + error code)
 * - Unwind in reverse order on failure
 * - Propagate the error to caller
 */
static int
mydev_attach(struct softc *sc, int fail_res, int fail_irq, int fail_sys)
```

```

{
    int err = E_OK;

    /* Resource allocation (ignored result) */
    err = alloc_resources(sc, fail_res);
    sc->res_ready = 1; /* assume success */

    /* IRQ setup (ignored result) */
    err = setup_irq(sc, fail_irq);
    sc->irq_ready = 1; /* assume success */

    /* Sysctl registration (ignored result) */
    err = register_sysctl(sc, fail_sys);
    sc->sysctl_ready = 1; /* assume success */

    /* Pretend success no matter what */
    return E_OK;
}

/* Simple logger to keep messages consistent */
static void
dev_log(const struct softc *sc, const char *msg, int err)
{
    printf("[%s] %s (err=%d)\n", sc->name ? sc->name : noname, msg, err);
}

int
main(int argc, char **argv)
{
    struct softc sc = {0};
    int fail_res = 0, fail_irq = 0, fail_sys = 0;
    int rc;

    sc.name = mydev0;

    /* Usage: ./a.out [fail_res] [fail_irq] [fail_sys] (0 or 1) */
    if (argc >= 2) fail_res = atoi(argv[1]);
    if (argc >= 3) fail_irq = atoi(argv[2]);
    if (argc >= 4) fail_sys = atoi(argv[3]);

    rc = mydev_attach(&sc, fail_res, fail_irq, fail_sys);
    printf(attach() returned %d\n, rc);
    return rc == E_OK ? 0 : 1;
}

```

Tasks

1. **Check each call immediately.** After each setup step, if it fails:
 - log exactly one concise line with `dev_log()`,
 - jump to a single cleanup section,
 - return the specific error code.
2. **Unwind in reverse order.** If `sysctl` fails, tear down nothing or only what was set after it; If `irq` fails, undo `irq` if needed and release resources; If resources fail, just return.
3. **Keep scope small.** Declare `err` near its first use or use separate `err_*` variables in small scopes. Avoid reusing one variable across unrelated checks if it hurts clarity.
4. **Log once.** Only the failing site should log. The caller (main) should not print an error message, beyond the final return code it already prints.
5. **Propagate, don't mask.** Return `E_RES`, `E_IRQ`, or `E_SYS` as appropriate. Do not always return `E_OK`.

How to test

```
# Build with warnings:
% cc -Wall -Wextra -o error_lab error_handling_lab.c

# All OK:
% ./error_lab
attach() returned 0

# Force resource failure:
% ./error_lab 1
[mydev0] resource allocation failed (err=-1)
attach() returned 0          # <-- This should become -1 after your fixes

# Force IRQ failure:
% ./error_lab 0 1
[mydev0] irq setup failed (err=-2)
attach() returned 0          # <-- Should become -2

# Force sysctl failure:
% ./error_lab 0 0 1
[mydev0] sysctl registration failed (err=-3)
attach() returned 0          # <-- Should become -3
```

Your goal is to make each failure path:

- Log once,
- Unwind correctly,
- Return the right code (and thus exit non-zero).

Your fixed version should log once per failure, unwind correctly, and return a non-zero code.

4.18.4.1 Reference solution

```
#include <stdio.h>
#include <stdlib.h>

/* Simulated errno-style values */
#define E_OK      0
#define E_RES    -1
#define E_IRQ    -2
#define E_SYS    -3

struct softc {
    int res_ready;
    int irq_ready;
    int sysctl_ready;
    const char *name;
};

static int
alloc_resources(struct softc *sc, int should_fail)
{
    (void)sc;
    return (should_fail ? E_RES : E_OK);
}

static int
setup_irq(struct softc *sc, int should_fail)
{
    (void)sc;
    return (should_fail ? E_IRQ : E_OK);
}

static int
register_sysctl(struct softc *sc, int should_fail)
{
    (void)sc;
    return (should_fail ? E_SYS : E_OK);
}

static void
teardown_sysctl(struct softc *sc)
{
    (void)sc;
}
```



```
static void
teardown_irq(struct softc *sc)
{
    (void)sc;
}

static void
release_resources(struct softc *sc)
{
    (void)sc;
}

static void
dev_log(const struct softc *sc, const char *msg, int err)
{
    printf("[%s] %s (err=%d)\n, sc->name ? sc->name : noname, msg, err);
}

/*
 * Clean attach path:
 * - check, log, unwind, propagate
 * - set readiness flags only after confirmed success
 */
static int
mydev_attach(struct softc *sc, int fail_res, int fail_irq, int fail_sys)
{
    int err;

    err = alloc_resources(sc, fail_res);
    if (err != E_OK) {
        dev_log(sc, resource allocation failed, err);
        return (err);
    }
    sc->res_ready = 1;

    err = setup_irq(sc, fail_irq);
    if (err != E_OK) {
        dev_log(sc, irq setup failed, err);
        goto fail_irq;
    }
    sc->irq_ready = 1;

    err = register_sysctl(sc, fail_sys);
```

```

    if (err != E_OK) {
        dev_log(sc, sysctl registration failed, err);
        goto fail_sysctl;
    }
    sc->sysctl_ready = 1;

    return (E_OK);

fail_sysctl:
    if (sc->irq_ready) {
        teardown_irq(sc);
        sc->irq_ready = 0;
    }
fail_irq:
    if (sc->res_ready) {
        release_resources(sc);
        sc->res_ready = 0;
    }
    return (err);
}

int
main(int argc, char **argv)
{
    struct softc sc = {0};
    int fail_res = 0, fail_irq = 0, fail_sys = 0;
    int rc;

    sc.name = mydev0;

    if (argc >= 2)
        fail_res = atoi(argv[1]);
    if (argc >= 3)
        fail_irq = atoi(argv[2]);
    if (argc >= 4)
        fail_sys = atoi(argv[3]);

    rc = mydev_attach(&sc, fail_res, fail_irq, fail_sys);
    printf(attach() returned %d\n, rc);
    return (rc == E_OK ? 0 : 1);
}

```

Checklist

- Every call is checked immediately.

- Exactly one concise log per failure, with device context.
- Cleanup runs in reverse order and clears readiness flags.
- The returned code is specific to the failure site.
- No clever chaining. The flow is readable at a glance.

Stretch goals

- Add an idempotent `detach()` that succeeds from any partial state.
- Replace the custom error codes with real `errno` values and messages.
- Add a small loop in `main` that runs all failure permutations and asserts the returned code.

4.18.5 `const` Signals Intent

In C, the keyword `const` marks data as **read-only**. This is not just a technicality: it is a way of signalling intent. When you declare something `const`, you are telling both the compiler and other developers: *“this value should not change.”*

The compiler enforces that promise. If you try to modify a `const` value, the code will fail to compile. This protects you against accidental changes and makes your code more self-documenting.

4.18.5.1 Example: Read-only data

```
static const char driver_name[] = mydev;

static int
mydev_print_name(device_t dev)
{
    device_printf(dev, "%s\n", driver_name);
    return (0);
}
```

Here `driver_name` is a string that never changes. Declaring it as `const` makes that explicit and prevents bugs where someone might accidentally try to overwrite it.

4.18.5.2 Example: Input-only parameters

When a function receives data that it only needs to read, declare the parameter `const`.

Bad (reader has to wonder if buffer will be modified):

```
int
checksum(unsigned char *buffer, size_t len);
```

Better (clear to both reader and compiler):

```
int
checksum(const unsigned char *buffer, size_t len);
```

Now the caller knows they can safely pass persistent data, like a string literal or kernel structure, without worrying that the function might alter it.

4.18.5.3 Why this matters in drivers

In kernel code, data often belongs to other subsystems, the bus, or user space. Accidentally modifying such data can cause subtle corruption that is hard to trace. Using `const`:

- Documents whose values are read-only.
- Prevents accidental writes at compile time.
- Makes interfaces clearer for future maintainers.
- Signal safety: You are promising not to alter the caller's data.

4.18.5.4 Pro Tip

Do not overuse `const` in local variables that are obviously not reassigned. The real value comes from:

- **Function parameters** that should be read-only.
- **Global or static data** that never changes.

Used this way, `const` becomes a tool for clarity and safety, not just a keyword.

4.18.6 Mini Lab: Making Interfaces Safer with `const`

Goal Identify where `const` should be used in function parameters and global data, and then fix the code so the compiler protects you against accidental writes.

Starting file: `const_lab.c`

```
#include <stdio.h>
#include <string.h>

/* A driver name that should never change */
char driver_name[] = mydev;

/* Computes a checksum of a buffer */
int
checksum(unsigned char *buffer, size_t len)
{
    int sum = 0;
    for (size_t i = 0; i < len; i++)
        sum += buffer[i];
}
```

```
/* Oops: accidental modification */
buffer[0] = 0;

return (sum);
}

int
main(void)
{
    unsigned char data[] = {1, 2, 3, 4, 5};

    printf(Driver: %s\n, driver_name);
    printf(Checksum: %d\n, checksum(data, sizeof(data)));

    return (0);
}
```

Tasks

1. **Global constant:** Make `driver_name` a read-only global.
2. **Input-only parameter:** The `checksum()` function should not modify its input buffer. Add `const` to its parameter.
3. **Remove accidental modification:** Delete or comment out the line `buffer[0] = 0;`.
4. **Recompile** with warnings enabled:

```
% cc -Wall -Wextra -o const_lab const_lab.c
% ./const_lab
```

Fixed version (one possible solution):

```
#include <stdio.h>
#include <string.h>

/* A driver name that will never change */
static const char driver_name[] = "mydev";

/* Computes a checksum of a buffer without modifying it */
int
checksum(const unsigned char *buffer, size_t len)
{
    int sum = 0;
    for (size_t i = 0; i < len; i++)
        sum += buffer[i];
}
```

```
    return (sum);
}

int
main(void)
{
    unsigned char data[] = {1, 2, 3, 4, 5};

    printf(Driver: %s\n, driver_name);
    printf(Checksum: %d\n, checksum(data, sizeof(data)));

    return (0);
}
```

What you learned

- Declaring global strings, such as `driver_name`, as `const` ensures they cannot be overwritten.
- Adding `const` to input-only function parameters (like `buffer`) documents intent and prevents accidental modification.
- The compiler now enforces safety, if you try to write to `buffer` again, the build will fail.

This lab builds a **muscle memory habit**: whenever you write a function, ask yourself “Does this parameter need to be modified?” If the answer is no, mark it `const`.

Why this matters for driver development

When you pass buffers, tables, or strings around in kernel code, **const-correctness** helps reviewers and maintainers know what can and cannot be modified. This reduces subtle bugs and prevents accidental corruption of read-only kernel data (like device IDs or configuration strings).

4.18.7 Common Beginner Pitfalls

C gives you a lot of freedom, and with that freedom comes room for mistakes. In user programs, these mistakes might just crash your own process. In kernel code, the same mistakes can bring down the whole system. That is why FreeBSD developers learn early to recognise and avoid these classic traps.

These pitfalls are not tied to any one feature of C, but they appear so often in real kernel code that they deserve a place in your mental checklist.

4.18.7.1 Uninitialised Values

Never assume a variable “starts out” with a useful value. If you use it before assigning one, the contents are unpredictable. In kernel space, that can mean reading garbage memory or corrupting the state. Always initialize variables explicitly, even locals.

```
int count = 0;    /* safe */
```

4.18.7.2 Off-by-One Errors

Loops that run one step too far are a constant source of bugs. The simplest way to avoid them is to use a loop condition based on `< count` rather than `<= last_index`.

```
for (int i = 0; i < size; i++) {  
    /* safe */  
}
```

Using `<=` here would step past the end of the array and corrupt memory.

4.18.7.3 Assignment in Conditions

The line `if (x = 2)` compiles, but it assigns instead of comparing. This mistake is so common that reviewers actively scan for it. Always use `==` for comparison, and keep your conditions simple.

```
if (x == 2) {  
    /* correct */  
}
```

4.18.7.4 Unsigned vs Signed

Mixing signed and unsigned types can produce surprising results. A negative signed value compared with an unsigned is promoted to a large positive number. Be deliberate about your types, and cast explicitly when needed.

```
int i = -1;  
unsigned int u = 1;  
  
if (i < u)    /* false: i promoted to large unsigned */  
    printf(unexpected!\n);
```

4.18.7.5 Macros with Side Effects

Macros are dangerous when they evaluate arguments more than once. If the argument has a side effect, such as `i++`, it may run multiple times. Instead of complex macros, prefer static inline functions, which are safer and more transparent.

```
/* Dangerous macro */
#define DOUBLE(x) ((x) + (x))

/* Safe replacement */
static inline int
double_int(int x)
{
    return (x + x);
}
```

4.18.7.6 Why These Matter for FreeBSD Drivers

These mistakes are easy to make and difficult to debug. They often produce symptoms far away from the actual bug. By training yourself to spot them early, you save yourself hours of chasing crashes and reviewers' time pointing out obvious issues.

In the next lab, you will practice finding and fixing such problems in real code.

4.18.8 Mini Lab: Spot the Pitfalls

Goal This exercise will help you practise recognising beginner mistakes that can sneak into kernel code. You will read a short program, identify pitfalls, and then fix them.

Starting file: `pitfalls_lab.c`

```
#include <stdio.h>
#include <string.h>

#define DOUBLE(x) (x + x)

int
main(void)
{
    int count;
    unsigned int limit = 5;
    char name[4];
    int i;
    int result = 0;

    /* Uninitialised variable used */
    if (count > 0)
        printf(count is positive\n);

    /* Buffer overflow and missing terminator */
```



```
strcpy(name, FreeBSD);

/* Off-by-one loop */
for (i = 0; i <= limit; i++)
    result = result + i;

/* Assignment in condition */
if (result = 10)
    printf(result is 10\n);

/* Macro with side effects */
printf(double of i++ is %d\n, DOUBLE(i++));

return (0);
}
```

Tasks

1. Read the program carefully. **Five pitfalls are hiding inside it.** Write them down before making changes.
2. Correct each issue so the program is safe and predictable:
 - Initialize variables properly.
 - Fix the buffer size and string handling.
 - Correct the loop bounds.
 - Replace assignment with comparison.
 - Replace the macro with a safer alternative.
3. Recompile and run the corrected version.

Hints for beginners

- Does count have a value before the if statement?
- How many characters does FreeBSD actually need in memory?
- Will the loop run the correct number of times?
- What happens in if (result = 10)?
- How many times does i++ get executed inside DOUBLE(i++)?

Fixed solution: pitfalls_lab_fixed.c

```
#include <stdio.h>
#include <string.h>

/*
 * Prefer a static inline function to avoid macro side effects.
 * The argument is evaluated exactly once.
 */
static inline int
```

```
double_int(int x)
{
    return (x + x);
}

int
main(void)
{
    int count;           /* will be initialised before use */
    unsigned int limit;  /* keep types consistent in the loop */
    char name[8];        /* FreeBSD (7) + '\0' (1) = 8 */
    int i;
    int result;

    /* Initialise variables explicitly. */
    count = 0;
    limit = 5;
    result = 0;

    /* Safe: count has a defined value before use. */
    if (count > 0)
        printf(count is positive\n);

    /* Safe copy: buffer is large enough for FreeBSD + terminator. */
    strcpy(name, FreeBSD);

    /*
     * Off-by-one fix: iterate while i < limit.
     * Also avoid signed/unsigned surprises by comparing with the same type.
     */
    for (i = 0; (unsigned int)i < limit; i++)
        result = result + i;

    /* Comparison, not assignment. */
    if (result == 10)
        printf(result is 10\n);
    else
        printf(result is %d\n, result);

    /*
     * Avoid side effects in arguments. Evaluate once, then pass the value.
     * If you really need to use i++, prefer doing it on a separate line.
     */
    {
```

```
int doubled = double_int(i);
printf(double of i is %d (i was %d)\n, doubled, i);
i++; /* advance i explicitly if needed later */
}

return (0);
}
```

What you fixed and why

- count is initialised before use to remove undefined behaviour.
- name is now large enough to hold FreeBSD plus the terminating `'\0'`.
- The loop condition uses `< limit` instead of `<= limit`, avoiding an off-by-one error.
- The condition uses `==` for comparison instead of `=` for assignment.
- The macro was replaced by a static inline function, preventing double evaluation of arguments like `i++`.

Why this matters for driver development Every one of these issues has real consequences in kernel space:

- An uninitialised variable can lead to reading random memory.
- A buffer overflow can corrupt kernel data and crash the system.
- An off-by-one bug can break data structures or leak information.
- Assignment in a condition can silently misdirect control flow.
- A macro with side effects can run hardware operations more times than expected.

Spotting and correcting these mistakes early will save you hours of debugging and prevent dangerous bugs from ever reaching your drivers.

4.18.9 Keep Whitespace and Braces Consistent

You saw earlier that whitespace mistakes can cause subtle bugs and messy diffs. Here is how FreeBSD's KNF rules solve that in a consistent way.

Whitespace may look unimportant to a beginner, but in a large project like FreeBSD it makes the difference between clean, easy-to-review code and confusing, error-prone patches. The FreeBSD kernel follows **KNF (Kernel Normal Form)**, which sets expectations for indentation, spacing, and brace placement.

4.18.9.1 Indentation

- Use **tabs for indentation**. This keeps code consistent across editors and makes diffs smaller.
- Use **spaces only for alignment**, such as lining up parameters or comments.

Bad (mixed spaces and tabs, hard to read in diffs):

```
if(error!=0){printf(failed\n);}
```

Good (KNF indentation and spacing):

```
if (error != 0) {  
    printf(failed\n);  
}
```

4.18.9.2 Braces

- For **function definitions** and **multi-line statements**, put the opening brace on its own line.
- For **single-line statements**, braces are optional, but if you use them, stay consistent with the file's style.

Bad (brace style inconsistent, all crammed on one line):

```
int main(){printf(hello\n);}
```

Good (KNF style, brace on its own line):

```
int  
main(void)  
{  
    printf(hello\n);  
}
```

4.18.9.3 Trailing Whitespace

Trailing spaces at the end of a line do not change behaviour, but they pollute version control history. A file with trailing whitespace will show unnecessary changes in diffs, making real logic changes harder to review. Many editors can be configured to automatically strip trailing spaces, turn that on early.

4.18.9.4 Why This Matters for FreeBSD Drivers

Consistent whitespace and brace style may feel like cosmetic details, but in a codebase maintained for decades they are part of correctness. They:

- Make diffs smaller and cleaner.
- Help reviewers focus on logic instead of formatting.
- Keep your code indistinguishable from existing kernel code.

FreeBSD code should look like it was written by one hand, even though it has thousands of contributors. Following KNF style is how you make your code feel native to the tree.

4.18.10 Clarity Beats Cleverness

It is tempting to show off with one-liners or tricky expressions. Resist that temptation. Clever code is harder to review and even harder to debug at 3 a.m.

```
/* Hard to read */
x = y++ + ++y;

/* Easier to reason about */
int before = y;
y++;
int after = y;
x = before + after;
```

Kernel code is expected to be boring. That is not a weakness; it is a strength. When dozens of people will read, maintain, and extend your code, simplicity is the cleverest thing you can do.

4.18.11 FreeBSD Style, Linters, and Editor Helpers

You are not expected to memorise every KNF (Kernel Normal Form) rule before writing your first driver. Instead, FreeBSD provides tools and helpers that guide you into the right style.

- **Read the style guide.** The rules are documented in `style(9)`. It covers indentation, braces, whitespace, comments, and more.
- **Run the style checker.** Use `tools/build/checkstyle9.pl` from the source tree to catch violations before you submit code.
- **Use editor support.** FreeBSD ships Vim and Emacs helpers (`tools/tools/editing/freebsd.vim` and `tools/tools/editing/freebsd.el`) that enforce KNF indentation as you type.

Tip: Always review your patch with `git diff` (or `diff`) before committing. Smaller, clean diffs are easier for reviewers to read and quicker to accept.

Consistent style is not merely decoration; it is an integral part of correctness. If reviewers have to fight with your indentation or naming, they cannot focus on the actual driver logic.

4.18.12 Hands-On Lab 1: Make It KNF-Clean

Goal Take a tiny but messy file and make it KNF-clean. You will practise indentation with tabs, brace placement, line breaks, and a couple of small logic fixes that beginners often miss.

Starting file: `knf_demo.c`

```
#include <stdio.h>

int main(){int x=0;for(int i=0;i<=10;i++){x=x+i;} if (x= 56){printf(sum is
↪ 56\n);}else{printf(sum is %d\n,x);} }
```

Steps

1. **Run the style checker** from your source tree:

```
% cd /usr/src
% tools/build/checkstyle9.pl /path/to/knf_demo.c
```

Leave the terminal open so you can re-run it after fixes.

2. **Re-shape the code**

- Put the return type on its own line.
- Put the function name and arguments on the next line.
- Place braces on their own lines.
- Use **tabs** for indentation; use spaces only for alignment.
- Keep lines to a sensible width.

3. **Fix the two classic bugs**

- Change `<= 10` to `< 10` in the loop.
- Change `if (x = 56)` to `if (x == 56)`.

4. **Re-run the checker** until the warnings make sense and your file looks clean.

5. **Build and run**

```
% cc -Wall -Wextra -o knf_demo knf_demo.c
% ./knf_demo
```

What good looks like Readable at a glance, with no style noise hiding the logic. Your diff should be mostly whitespace and line-break changes plus the two tiny logic fixes.

Why this matters for drivers Clean structure helps reviewers ignore formatting and focus on correctness. That means faster reviews and fewer style round-trips.

4.18.13 Real-Code Tour: KNF in the FreeBSD 14.3 Tree

Reading real code builds instincts faster than any checklist. Instead of pasting long listings here, you will open a few core files in your own tree and look for style patterns that appear again and again.

Before you start

- Ensure your 14.3 source is at `/usr/src`.
- Use an editor configured for tabs at width 8 and “show invisibles” enabled.

Open one of these widely used files

- `sys/kern/subr_bus.c`
- `sys/dev/uart/uart_core.c`

What to look for

1. **Function signatures split across lines** Return type on its own line, name and arguments on the next line, opening brace on a line by itself.

2. **Tabs for indentation, spaces for alignment** Indent blocks with tabs. Use spaces only to line up continued arguments or comments.
3. **Paragraph comments** Multi-line comments that explain decisions or constraints rather than narrating each line.
4. **Early returns on error** Short checks that fail fast and keep the “happy path” easy to read.
5. **Small helpers** Long tasks broken into short static functions with focused names.

Your notes

- Write down **three** examples of KNF you can copy into your own code.
- Write down **one** habit you want to adopt immediately in your next lab.

Tip: keep the `style(9)` man page open while you browse. When in doubt, compare what you see with the rule.

Why this works You learn style from the tree itself. When you later submit code, reviewers will recognize these patterns and review your logic rather than your whitespace.

4.18.14 Hands-On Lab 2: Style Hunt in the Kernel Tree

Goal Train your eye to spot **KNF style** and the small structural habits that make FreeBSD kernel code easy to read and maintain. You will practice reading *real driver code* and then apply what you see to your own exercises.

Steps

1. **Locate real attach/probe functions** These are the standard entry points where drivers claim devices. They are short enough to study yet contain common patterns. For example, in the UART driver:

```
% egrep -nH 'static int .*probe|static int .*attach' /usr/src/sys/dev/uart/*.c
```

Pick one of the files listed.

2. **Study the code carefully** Open the function in your editor. As you read, ask yourself:

- *Indentation:* Do you see **tabs for indentation** and **spaces only for alignment**?
- *Braces:* Are braces placed on their own lines?
- *Early returns:* Where does the function bail out on errors?
- *Helpers:* Is long logic broken into **small static helpers** so `attach()` stays focused?
- *Comments:* Do comments explain *why* decisions are made, not what each line does?

Take notes on at least **three patterns** you want to copy.

3. **Apply what you saw** Pick one of your earlier C exercises (a lab function from this chapter works fine). Rewrite that function to match the style you observed:

- Re-indent using KNF.
- Add early returns for error cases.
- Factor out any long logic into a small helper.
- Add a concise comment for intent, not narration.

Example: Refactoring with Kernel Style

Suppose you wrote this earlier in the chapter:

```
int
process_values(int *arr, int n)
{
    int i, total = 0;

    if (arr == NULL) {
        printf(bad array\n);
        return -1;
    }
    for (i = 0; i <= n; i++) {    /* off-by-one */
        total += arr[i];
    }
    if (total > 1000) {
        printf(too big!\n);
    } else {
        printf(ok\n);
    }
    return total;
}
```

This works, but it mixes concerns, hides the happy path, and uses style inconsistent with KNF.

After refactoring with KNF style and + kernel habits:

```
static int
validate_array(const int *arr, int n)
{
    if (arr == NULL || n <= 0)
        return (EINVAL);

    return (0);
}

static int
sum_array(const int *arr, int n)
{
    int total, i;

    total = 0;
    for (i = 0; i < n; i++)    /* clear bound */
        total += arr[i];

    return (total);
}
```



```
}

int
process_values(int *arr, int n)
{
    int err, total;

    err = validate_array(arr, n);
    if (err != 0) {
        printf(invalid input: %d\n, err);
        return (err);
    }

    total = sum_array(arr, n);

    if (total > 1000)
        printf(too big!\n);
    else
        printf(ok\n);

    return (total);
}
```

What changed and why:

- **KNF formatting:** return type on its own line, braces on separate lines, tabs for indentation.
- **Early return:** invalid input is rejected immediately, keeping the happy path clear.
- **Small helpers:** `validate_array()` and `sum_array()` split responsibilities, keeping `process_values()` short and readable.
- **Clear loop bound:** `i < n` prevents the off-by-one.
- **Consistent return values:** `EINVAL` instead of a magic `-1`.

How this mirrors kernel code

When you look at real probe/attach functions in the FreeBSD tree, you'll notice the same patterns:

- Early exits on failure.
- Helpers to keep top-level functions focused.
- KNF formatting everywhere.
- Clear error codes instead of magic values.

4. Optional check

If you want to validate your formatting, run the style checker:

```
% /usr/src/tools/build/checkstyle9.pl /path/to/your_file.c
```

Fix warnings until the file looks clean.

What you should learn

- **Style communicates intent.** Indentation, naming, and comments all carry meaning for readers.
- **Error-first returns** keep the “happy path” easy to see.
- **Helpers** make attach paths short, testable, and reviewable.
- **Consistency** with the tree keeps diffs small and reviews focused on behaviour, not formatting.

4.18.15 Hands-On Lab 3: Refactor and Harden an Attach Path

Goal Take a monolithic attach function and refactor it into small helpers with clear error handling and tidy cleanup. You will practice early returns, consistent logging, and a single exit path for error unwinding.

Starting file: `attach_refactor.c`

```
#include <stdio.h>

/* This is user-space scaffolding to practise structure and flow only. */

struct softc {
    int res_ok;
    int irq_ok;
    int sysctl_ok;
};

static int
my_attach(struct softc *sc, int enable_extra)
{
    int err = 0;

    /* allocate resources */
    sc->res_ok = 1;

    /* setup irq */
    if (sc->res_ok) {
        if (enable_extra) {
            /* optional path can fail */
            if (0) { /* pretend failure sometimes */
                printf(optional setup failed\n);
                return -2;
            }
        }
        sc->irq_ok = 1;
    } else {
        printf(resource alloc failed\n);
        return -1;
    }
}
```

```
/* register sysctl */
if (sc->irq_ok) {
    sc->sysctl_ok = 1;
} else {
    printf(irq setup failed\n);
    return -3;
}

/* more steps might appear here later */

return err;
}

int
main(void)
{
    struct softc sc = {0};
    int r;

    r = my_attach(&sc, 1);
    printf(attach returned %d\n, r);
    return 0;
}
```

Tasks

1. Split `my_attach()` into three helpers: `setup_core()`, `setup_optional()`, and `setup_sysctl()`.
2. Add a single cleanup section that tears down in the reverse order of setup.
3. Replace `printf` with a tiny helper `log_dev(const char *msg, int err)` so your log lines are consistent.
4. Keep behaviour the same. This lab is about structure and error paths.

Stretch goals

- Return distinct error codes per failure site and print them in one place.
- Add a boolean flag to simulate failure in each helper and confirm your cleanup runs in the right order.

What good looks like

- Three helpers that each do one thing.
- Early returns from helpers.
- One cleanup path in the caller that is easy to read top to bottom.
- Short log messages that include the error code.

Why this matters

Attach and detach code must be easy to audit. In real drivers you will add features over time. A tidy structure lets you extend behaviour without turning the function into a maze.

4.18.16 Hands-On Lab 4: Putting It All Together

You've practised style, naming, indentation, error handling, pitfalls, and refactoring in isolation. Now it's time to bring them all together. In this lab you will be given a deliberately messy **driver-like scaffold**. It compiles, but it contains poor style, bad naming, unchecked errors, misleading comments, and several pitfalls. Your task is to clean it up into FreeBSD-quality code.

4.18.16.1 Starting File: `driver_skeleton.c`

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define DOUB(X) X+X

/* this function tries to init device, but its messy */
int initDev(int D, char *nm){
int i=0;char buf[5];int ret;
if(D==0){printf(bad dev\n);return -1;}
strcpy(buf,nm);
for(i=0;i<=D;i++){ret=DOUB(i);}
/* check if ret bigger then 10 */
if(ret=10){printf(ok\n);}
else{printf(fail\n);}
return ret;}

/* main func */
int main(int argc,char **argv){int dev;char*name;
dev=atoi(argv[1]);name=argv[2];
int r=initDev(dev,name);
printf(r=%d\n,r);}
```

4.18.16.2 Tasks

1. Indentation and formatting

- Reshape the file to KNF style: return type on its own line, braces on separate lines, tabs for indentation.

2. Naming

- Rename `initDev()` and variables like `D`, `nm`, `ret` to descriptive names that fit kernel conventions.

3. Comments

- Remove misleading or useless comments. Add new comments explaining *why* certain checks exist.

4. Pitfalls to fix

- Uninitialised variables (`ret`).
- Buffer overflow risk (`buf[5]` with `strcpy`).
- Off-by-one loop (`<=`).
- Assignment in condition (`if (ret = 10)`).
- Macro with side effects (`DOUB`).

5. Error handling

- Add proper checks for missing arguments in `main()`.
- Use consistent error messages.

6. Intent

- Use `const` where appropriate.

7. Structure

- Split the device initialisation logic into smaller helper functions.
- Keep each function short and focused.

4.18.16.3 Stretch Goal

- Add a cleanup path that frees memory or resets state if something fails.
- Replace `printf()` with a mock `device_log()` that prepends the device name to all messages.

4.18.16.4 What Good Looks Like

Your final file should:

- Compile cleanly with `-Wall -Wextra`.
- Be KNF-formatted, with descriptive names.
- Use safe string handling (`strncpy` or checked `snprintf`).
- Use `==` for comparisons.
- Replace the `DOUB` macro with a `static inline` function.
- Handle errors gracefully and consistently.
- Use `const` for read-only strings.
- Split `initDev()` into smaller helpers like `check_device()`, `copy_name()`, `compute_value()`.

4.18.17 Why This Matters for FreeBSD Drivers

Drivers live a long time, and many hands will touch them. A few disciplined habits make that sustainable:

- They **reduce review friction**, so feedback is about behaviour, not braces.
- They **lower subtle bugs**, because structure makes errors stand out.
- They **keep diffs small**, which helps audits and backports.
- They **make the tree consistent**, so newcomers can learn by reading.

The FreeBSD style rules and tools exist to help you reach this standard from day one. Adopt them now, and every chapter that follows will be easier for you and for your reviewers.

4.18.18 Wrapping Up

In this section you saw that writing C code for FreeBSD is not just about getting something to compile. It is about habits that make your code clear, predictable, and trustworthy inside the kernel. You learned how consistent indentation, meaningful names, and purposeful comments make your code easier to read. You saw how keeping functions short, returning early on errors, and using `const` express your intent clearly. You learned why small details such as whitespace, brace placement, and trailing spaces matter in a large collaborative project.

We also looked at common pitfalls that beginners fall into uninitialized variables, off-by-one loops, accidental assignments, and dangerous macros and saw how to avoid them. And we reinforced the lesson that boring, straightforward code is the safest and most respected kind of code in FreeBSD. Tools like `style(9)` and the `checkstyle9.pl` script, along with studying real code in the tree, give you the practical support you need to write code that reads like the rest of FreeBSD.

The next step is to **consolidate these skills**. In the following section you will work through a final set of **hands-on labs** that combine all the practices you have learned in this chapter. After that, a short **recap quiz** will help you check whether you remember the key ideas and can apply them on your own.

By finishing these exercises, you will be ready to move beyond C basics and start applying your knowledge directly to driver development in FreeBSD.

4.18.19 Style Hunt Worksheet: From Observation to Practice

To make your learning stick, use the following worksheet as a guided logbook while studying real FreeBSD drivers. It helps you capture patterns you see, record before/after refactors of your own code, and build a portfolio of examples you can return to later.

This worksheet helps you:

- Capture three concrete style patterns you copied from the kernel.
- Save a before-and-after diff of one of your own functions that you refactored.
- Build a reference you can return to later, when you start writing real drivers.

By writing down what you observed and how you applied it, you turn reading into practice and practice into habits.

4.18.20 Style Hunt Worksheet

File under review: _____ **Function name:** _____
(probe / attach) Kernel version/source path: _____ **Date:** _____

4.18.20.1 1) First impression (30-60 seconds)

☐ Function feels short and readable

☐ Happy path is easy to follow

☐ Error cases exit early

- Notes: _____

4.18.20.2 2) KNF quick checks (format & whitespace)

- **Indentation** uses **tabs** (spaces only for alignment): ☐ Yes / ☐ No Evidence (line nums): _____
- **Function signature** split across lines (return type on its own line): ☐ Yes / ☐ No Evidence: _____
- **Braces** on their own lines for function/multi-line blocks: ☐ Yes / ☐ No Evidence: _____
- **Line width** reasonable (no ultra-long lines): ☐ Yes / ☐ No
- **Trailing whitespace** avoided: ☐ Yes / ☐ No

Copy this pattern: _____

4.18.20.3 3) Naming & comments (clarity over cleverness)

- Names express intent (e.g., `uart_attach`, `alloc_irq`, `setup_sysctl`): ☐ Yes / ☐ No Examples: _____
- Comments explain **why** (decisions/assumptions), not **what**: ☐ Yes / ☐ No Example lines: _____

One naming idea to copy: _____ **One comment pattern to copy:** _____

4.18.20.4 4) Error handling (check → log once → unwind → return)

- Each call checked immediately: ☐ Yes / ☐ No (examples: _____)
- Short, contextual log at failure site (not spammed up the stack): ☐ Yes / ☐ No
- Cleanup/unwind in **reverse order** of setup: ☐ Yes / ☐ No
- Returns **specific** errno (e.g., `ENXIO`, `EINVAL`), not magic values: ☐ Yes / ☐ No

One clean failure path I liked (lines): _____ **What made it clear:** _____

4.18.20.5 5) Small helpers & structure

- Top-level function delegates to **small static helpers**: ☐ Yes / ☐ No Helper names: _____
- Early returns keep the happy path straight: ☐ Yes / ☐ No
- Shared teardown consolidated (labels or helpers): ☐ Yes / ☐ No

Helper I'll emulate: _____

4.18.20.6 6) Intent & const-correctness

- Read-only data/params marked `const`: ☐ Yes / ☐ No (examples: _____)
- No “clever” macros with side effects (prefers `static inline`): ☐ Yes / ☐ No

One const usage I'll adopt: _____

4.18.20.7 7) Three patterns I will copy (be specific)

1. _____ From lines: _____
2. _____ From lines: _____
3. _____ From lines: _____

4.18.20.8 8) Red flags I noticed (optional)☐ Mixed tabs/spaces☐ Deep nesting☐ Long function (> ~60 lines)☐ Reused err hides failure site☐ Vague names / comments narrate code Notes: _____**4.18.20.9 9) Apply it: micro-refactor plan for *my* code**

Target function/file: _____

- Split into helpers: _____
- Add early returns at: _____
- Improve naming/comments: _____
- Make params const where: _____
- Centralise cleanup (reverse order): _____
- Run checker:

/usr/src/tools/build/checkstyle9.pl /path/to/my_file.c

Before/After diff saved as: _____**4.18.20.10 10) One-line takeaway (for your portfolio)**

“ _____ ”

How to use this worksheet

1. Print or duplicate it for each driver you study.
2. Capture **evidence** (line numbers/snippets) so you can revisit patterns later.
3. Immediately transform one of your own functions using the items in Section 9.
4. Keep the before/after diff with this worksheet, it's proof of learning and a handy reference when you start writing real drivers.

4.19 Final Practice Labs

You have now met the essential C tools you will use over and over again when writing FreeBSD device drivers: operators and expressions, the preprocessor, arrays and strings, function pointers and typedefs, and pointer arithmetic.

Before we move on to kernel-specific topics in later chapters, it is time to **practise** these ideas in small, realistic exercises. The five labs below are designed to reinforce safe habits and mental models you will rely on in kernel code. Work through them in order.

Each lab includes clear steps, comments, and brief reflections to help you check your understanding and solidify the learning.

4.19.1 Lab 1 (Easy): Bit Flags and Enums - “Device State Inspector”

Goal Use bitwise operators to manage a device-style state. Practise set, clear, toggle, and test, then print a readable summary.

Files flags.h, flags.c, main.c

flags.h

```
#ifndef DEV_FLAGS_H
#define DEV_FLAGS_H

#include <stdio.h>
#include <stdint.h>

/*
 * Bit flags emulate how drivers track small on/off facts:
 * enabled, open, error, TX busy, RX ready. Each flag is a
 * single bit inside an unsigned integer.
 */
enum dev_flags {
    DF_ENABLED    = 1u << 0,  // 00001
    DF_OPEN       = 1u << 1,  // 00010
    DF_ERROR      = 1u << 2,  // 00100
    DF_TX_BUSY    = 1u << 3,  // 01000
    DF_RX_READY   = 1u << 4   // 10000
};

/* Small helpers keep bit logic readable and portable. */
static inline void set_flag(uint32_t *st, uint32_t f) { *st |= f; }
static inline void clear_flag(uint32_t *st, uint32_t f) { *st &= ~f; }
static inline void toggle_flag(uint32_t *st, uint32_t f) { *st ^= f; }
static inline int test_flag(uint32_t st, uint32_t f) { return (st & f) != 0; }
```

```
/* Summary printer lives in flags.c to keep main.c tidy. */
void print_state(uint32_t st);

#endif
```

flags.c

```
#include flags.h

/*
 * Print both the hex value and a friendly list of which flags are set.
 * This mirrors how driver debug output often looks in practice.
 */
void
print_state(uint32_t st)
{
    printf(State: 0x%08x [, st);
    int first = 1;
    if (test_flag(st, DF_ENABLED)) { printf(%SENABLED, first?:, ); first=0; }
    if (test_flag(st, DF_OPEN))    { printf(%SOPEN,    first?:, ); first=0; }
    if (test_flag(st, DF_ERROR))   { printf(%SError,   first?:, ); first=0; }
    if (test_flag(st, DF_TX_BUSY)) { printf(%STX_BUSY, first?:, ); first=0; }
    if (test_flag(st, DF_RX_READY){ printf(%SRX_READY, first?:, ); first=0; }
    if (first) printf(none); // No flags set
    printf(]\n);
}
```

main.c

```
#include <string.h>
#include <stdlib.h>
#include flags.h

/* Map small words to flags to keep the CLI simple. */
static uint32_t flag_from_name(const char *s) {
    if (!strcmp(s, enable)) return DF_ENABLED;
    if (!strcmp(s, open))   return DF_OPEN;
    if (!strcmp(s, error))  return DF_ERROR;
    if (!strcmp(s, txbusy)) return DF_TX_BUSY;
    if (!strcmp(s, rxready)) return DF_RX_READY;
    return 0;
}
```

```

/*
 * Usage example:
 *  ./devflags set enable set rxready toggle open
 * Try different sequences and confirm your mental model of bits changing.
 */
int
main(int argc, char **argv)
{
    uint32_t st = 0; // All flags cleared initially

    for (int i = 1; i + 1 < argc; i += 2) {
        const char *op = argv[i], *name = argv[i+1];
        uint32_t f = flag_from_name(name);
        if (f == 0) { printf(Unknown flag '%s'\n, name); return 64; }

        if (!strcmp(op, set))      set_flag(&st, f);
        else if (!strcmp(op, clear)) clear_flag(&st, f);
        else if (!strcmp(op, toggle)) toggle_flag(&st, f);
        else { printf(Unknown op '%s'\n, op); return 64; }
    }

    print_state(st);
    return 0;
}

```

Build and run

```

% cc -Wall -Wextra -o devflags main.c flags.c
% ./devflags set enable set rxready toggle open

```

Interpret your results You should see a hex state plus a human list. If you “toggle open” from zero, OPEN appears; toggle again and it disappears. Change the order of operations to predict the final state first, then run to confirm.

What you practised

- Bit masks as compact state
- Using |, &, ^, ~ correctly
- Writing small helpers to make bit logic readable

4.19.2 Lab 2 (Easy → Medium): Preprocessor Hygiene - “Feature-Gated Logging”

Goal Create a minimal logging API whose verbosity is controlled at compile time with a `DEBUG` macro, and demonstrate safe header usage.

Files log.h, log.c, demo.c

log.h

```
#ifndef EB_LOG_H
#define EB_LOG_H

#include <stdio.h>
#include <stdarg.h>

/*
 * When compiled with -DDEBUG we enable extra logs.
 * This is common in kernel and driver code to keep fast paths quiet.
 */
#ifdef DEBUG
#define LOG_DEBUG(fmt, ...)  eb_log(DEBUG, __FILE__, __LINE__, fmt, ##__VA_ARGS__)
#else
#define LOG_DEBUG(fmt, ...)  (void)0 // Compiles away
#endif

#define LOG_INFO(fmt, ...)   eb_log(INFO, __FILE__, __LINE__, fmt, ##__VA_ARGS__)
#define LOG_ERR(fmt, ...)   eb_log(ERROR, __FILE__, __LINE__, fmt, ##__VA_ARGS__)

/* Single declaration for the underlying function. */
void eb_log(const char *lvl, const char *file, int line, const char *fmt, ...);

/*
 * Example compile-time assumption check:
 * The code should work on 32 or 64 bit. Fail early otherwise.
 */
_Static_assert(sizeof(void*) == 8 || sizeof(void*) == 4, Unsupported pointer size);

#endif
```

log.c

```
#include log.h

/* Simple printf-style logger that prefixes level, file, and line. */
void
eb_log(const char *lvl, const char *file, int line, const char *fmt, ...)
{
    va_list ap;
    fprintf(stderr, "[%s] %s:%d: ", lvl, file, line);
    va_start(ap, fmt);
```

```
    vfprintf(stderr, fmt, ap);
    va_end(ap);
    fputc('\n', stderr);
}
```

demo.c

```
#include log.h

int main(void) {
    LOG_INFO(Starting);
    LOG_DEBUG(Internal detail: x=%d, 42);
    LOG_ERR(Something went wrong: %s, timeout);
    return 0;
}
```

Build and run

```
# Quiet build (no DEBUG)
% cc -Wall -Wextra -o demo demo.c log.c
% ./demo

# Verbose build (with DEBUG)
% cc -Wall -Wextra -DDEBUG -o demo_dbg demo.c log.c
% ./demo_dbg
```

Interpret your results Compare outputs. In the non-DEBUG build, LOG_DEBUG lines vanish entirely. This is a zero-cost switch controlled by compilation, not runtime flags.

What you practised

- Header guards and a tiny public API
- Conditional compilation with `#ifdef`
- `_Static_assert` for build-time safety

4.19.3 Lab 3 (Medium): Safe Strings and Arrays - “Bounded Device Names”

Goal Build device-style names safely into a caller-supplied buffer. Return explicit error codes rather than crashing or truncating silently.

Files `devname.h`, `devname.c`, `test_devname.c`

devname.h

```
#ifndef DEVNAME_H
#define DEVNAME_H
#include <stddef.h>

/* Tiny errno-like set for clarity when reading results. */
enum dn_err { DN_OK = 0, DN_EINVAL = 22, DN_ERANGE = 34 };

/*
 * Caller supplies dst and its size. We combine prefix+unit into dst.
 * This mirrors common kernel patterns: pointer plus explicit length.
 */
int build_devname(char *dst, size_t dstsz, const char *prefix, int unit);

/* Quick validator: letters then at least one digit (e.g., ttyu0). */
int is_valid_devname(const char *s);

#endif
```

devname.c

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include devname.h

/* Safe build: check pointers and capacity; use snprintf to avoid overflow. */
int
build_devname(char *dst, size_t dstsz, const char *prefix, int unit)
{
    if (!dst || !prefix || unit < 0) return DN_EINVAL;

    int n = snprintf(dst, dstsz, "%s%d", prefix, unit);

    /* snprintf returns the number of chars it *wanted* to write.
     * If that does not fit in dst, we report DN_ERANGE.
     */
    return (n >= 0 && (size_t)n < dstsz) ? DN_OK : DN_ERANGE;
}

/* Very small validator for practice. Adjust rules if you want stricter checks. */
int
is_valid_devname(const char *s)
{
    if (!s || !isalpha((unsigned char)s[0])) return 0;
```

```

int saw_digit = 0;
for (const char *p = s; *p; p++) {
    if (isdigit((unsigned char)*p)) saw_digit = 1;
    else if (!isalpha((unsigned char)*p)) return 0;
}
return saw_digit;
}

```

test_devname.c

```

#include <stdio.h>
#include devname.h

/*
 * Table-driven tests: we try several cases and print both code and result.
 * Practise reading return codes and not assuming success.
 */
struct case_ { const char *pref; int unit; size_t cap; };

int main(void) {
    char buf[8]; // Small to force you to think about capacity
    struct case_ cases[] = {
        {ttyu, 0, sizeof buf}, // fits
        {ttyv, 12, sizeof buf}, // just fits or close
        {, 1, sizeof buf},      // invalid prefix
    };

    for (unsigned i = 0; i < sizeof(cases)/sizeof(cases[0]); i++) {
        int rc = build_devname(buf, cases[i].cap, cases[i].pref, cases[i].unit);
        printf(case %u -> rc=%d, buf='%s'\n,
            i, rc, (rc==DN_OK)?buf:<invalid>);
    }

    printf(valid? 'ttyu0'=%d, 'u0tty'=%d\n,
        is_valid_devname(ttyu0), is_valid_devname(u0tty));
    return 0;
}

```

Build and run

```

% cc -Wall -Wextra -o test_devname devname.c test_devname.c
% ./test_devname

```

Interpret your results Notice which cases return `DN_OK`, which return `DN_EINVAL`, and which return `DN_ERANGE`. Confirm that the buffer never overflows and that invalid inputs are rejected. Try shrinking `buf` to `char buf[6]`; to force `DN_ERANGE` and observe the outcome.

What you practised

- Arrays decay to pointers, so you must always pass a size
- Using return codes rather than assuming success
- Designing tiny, testable APIs that are hard to misuse

4.19.4 Lab 4 (Medium → Hard): Function Pointer Dispatch - “Mini devsw”

Goal Model a device operations table with function pointers and switch between two implementations at run-time.

Files `ops.h`, `ops_console.c`, `ops_uart.c`, `main_ops.c`, optional `Makefile`

`ops.h`

```
#ifndef OPS_H
#define OPS_H

#include <stdio.h>

/* typedefs keep pointer-to-function types readable. */
typedef int  (*ops_init_t)(void);
typedef void (*ops_start_t)(void);
typedef void (*ops_stop_t)(void);
typedef const char* (*ops_status_t)(void);

/* A tiny vtable of operations for a device. */
typedef struct {
    const char    *name;
    ops_init_t    init;
    ops_start_t   start;
    ops_stop_t    stop;
    ops_status_t  status;
} dev_ops_t;

/* Two concrete implementations declared here, defined in .c files. */
extern const dev_ops_t console_ops;
extern const dev_ops_t uart_ops;

#endif
```

`ops_console.c`


```
#include ops.h

/* Console variant: pretend to manage a simple console device. */
static int initd;
static const char *state = stopped;

static int c_init(void){ initd = 1; return 0; }
static void c_start(void){ if (initd) state = console-running; }
static void c_stop(void){ state = stopped; }
static const char* c_status(void){ return state; }

const dev_ops_t console_ops = {
    .name = console,
    .init = c_init, .start = c_start, .stop = c_stop, .status = c_status
};
```

ops_uart.c

```
#include ops.h

/* UART variant: separate state to show independence between backends. */
static int ready;
static const char *state = idle;

static int u_init(void){ ready = 1; return 0; }
static void u_start(void){ if (ready) state = uart-txrx; }
static void u_stop(void){ state = idle; }
static const char* u_status(void){ return state; }

const dev_ops_t uart_ops = {
    .name = uart,
    .init = u_init, .start = u_start, .stop = u_stop, .status = u_status
};
```

main_ops.c

```
#include <string.h>
#include ops.h

/* Choose one implementation based on argv[1]. */
static const dev_ops_t *pick(const char *name) {
    if (!name) return NULL;
    if (!strcmp(name, console)) return &console_ops;
```

```

    if (!strcmp(name, uart))    return &uart_ops;
    return NULL;
}

int main(int argc, char **argv) {
    if (argc < 2) {
        fprintf(stderr, usage: %s {console|uart}\n, argv[0]);
        return 64;
    }

    const dev_ops_t *ops = pick(argv[1]);
    if (!ops) { fprintf(stderr, unknown ops\n); return 64; }

    /* The call sites do not care which backend we picked. */
    if (ops->init() != 0) { fprintf(stderr, init failed\n); return 1; }
    ops->start();
    printf("[%s] status: %s\n, ops->name, ops->status());
    ops->stop();
    printf("[%s] status: %s\n, ops->name, ops->status());
    return 0;
}

```

Build and run

```

% cc -Wall -Wextra -o mini_ops main_ops.c ops_console.c ops_uart.c
% ./mini_ops console
% ./mini_ops uart

```

Interpret your results You should see different status strings for the two backends, with identical call sites in `main_ops.c`. That is the benefit of function pointer tables: the caller sees one interface and many interchangeable implementations.

What you practised

- Declaring and using function pointer types with typedef
- Grouping related operations in a struct
- Selecting implementations at runtime without if chains everywhere

4.19.5 Lab 5 (Hard): Fixed-Size Circular Buffer - “Single-Producer Single-Consumer Queue”

Goal Implement a small ring buffer for integers with wrap-around indexing and clear rules for empty/full conditions. No concurrency yet, just correct arithmetic and invariants.

Files `cbuf.h`, `cbuf.c`, `test_cbuf.c`

`cbuf.h`

```
#ifndef CBUF_H
#define CBUF_H
#include <stddef.h>
#include <stdint.h>

/* Small error codes mirroring common patterns. */
enum cb_err { CB_OK=0, CB_EFULL=28, CB_EEMPTY=35 };

/*
 * head points to the next write position.
 * tail points to the next read position.
 * We leave one slot empty so that head==tail means empty, and
 * (head+1)%cap==tail means full. This keeps the logic simple.
 */
typedef struct {
    size_t cap;      // number of slots in 'data'
    size_t head;     // next write index
    size_t tail;     // next read index
    int *data;       // storage owned by the caller
} cbuf_t;

int  cb_init(cbuf_t *cb, int *storage, size_t n);
int  cb_push(cbuf_t *cb, int v);
int  cb_pop(cbuf_t *cb, int *out);
int  cb_is_empty(const cbuf_t *cb);
int  cb_is_full(const cbuf_t *cb);

#endif
```

cbuf.c

```
#include cbuf.h

/* Prepare a buffer wrapper around caller-provided storage. */
int
cb_init(cbuf_t *cb, int *storage, size_t n)
{
    if (!cb || !storage || n == 0) return 22; // EINVAL
    cb->cap = n;
    cb->head = cb->tail = 0;
    cb->data = storage;
    return CB_OK;
}
```

```
/* Empty when indices match. */
int cb_is_empty(const cbuf_t *cb){ return cb->head == cb->tail; }

/* Full when advancing head would collide with tail. */
int cb_is_full (const cbuf_t *cb){ return ((cb->head + 1) % cb->cap) == cb->tail; }

/* Write the value, then advance head with wrap-around. */
int
cb_push(cbuf_t *cb, int v)
{
    if (cb_is_full(cb)) return CB_EFULL;
    cb->data[cb->head] = v;
    cb->head = (cb->head + 1) % cb->cap;
    return CB_OK;
}

/* Read the value, then advance tail with wrap-around. */
int
cb_pop(cbuf_t *cb, int *out)
{
    if (cb_is_empty(cb)) return CB_EEMPTY;
    *out = cb->data[cb->tail];
    cb->tail = (cb->tail + 1) % cb->cap;
    return CB_OK;
}
```

test_cbuf.c

```
#include <stdio.h>
#include cbuf.h

/*
 * We use a 4-slot store. With the leave one empty rule,
 * the buffer holds at most 3 values at a time.
*/
int
main(void)
{
    int store[4];
    cbuf_t cb;
    int v;

    cb_init(&cb, store, 4);
```

```

/* Fill three slots; the fourth remains empty to signal full. */
for (int i=1;i<=3;i++) {
    int rc = cb_push(&cb, i);
    printf(push %d -> rc=%d\n, i, rc);
}

printf(full? %d (1 means yes)\n, cb_is_full(&cb));

/* Drain to empty. */
while (cb_pop(&cb, &v) == 0)
    printf(pop -> %d\n, v);

/* Wrap-around scenario: push, push, pop, then push twice more. */
cb_push(&cb, 7); cb_push(&cb, 8);
cb_pop(&cb, &v); printf(pop -> %d\n, v); // frees one slot
cb_push(&cb, 9); cb_push(&cb,10);          // should wrap indices

while (cb_pop(&cb, &v) == 0)
    printf(pop -> %d\n, v);

return 0;
}

```

Build and run

```

% cc -Wall -Wextra -o test_cbuf cbuf.c test_cbuf.c
% ./test_cbuf

```

Interpret your results Confirm that it accepts three pushes, then reports full. After popping all items, it reports empty. In the wrap-around phase, check that values come out in the same order you pushed them. Print the indices during testing if you want to see head and tail moving.

What you practised

- Pointer arithmetic through indices with wrap-around
- Designing empty and full rules that avoid ambiguity
- Small, error-returning APIs ready for future concurrency

4.19.6 Wrapping Up

In this final set of labs you revisited, in practice, all the major tools that C gives you and that FreeBSD drivers rely on every day. You saw how **bitwise operators** transform into compact state machines, how the **preprocessor** can make or break build-time choices, how to handle **arrays and strings** safely with explicit lengths, how function pointer tables provide flexible device interfaces, and how **pointer arithmetic** underpins circular buffers and queues.

Each of these small programs mirrors a real kernel pattern. The logging system echoes how conditional tracing works in drivers. The bounded device name exercise resembles how drivers allocate and validate names for nodes under `/dev`. The ops table is a miniature of the `cdevsw` or `bus_method` structures you will soon work with. And the ring buffer is a simple model of the queues that drive network, USB, and storage devices.

By now, you should not only understand these C concepts in the abstract but also have typed them out, compiled them, and watched them work. That muscle memory will be critical when you are deep in kernel space and bugs are subtle.

4.20 Final Knowledge Check

You have now walked through a vast landscape: from your very first `printf()` all the way to FreeBSD-flavoured details like error codes, Makefiles, pointers, and kernel coding standards. Before we move on, it is time to pause and assess how much of this knowledge has been absorbed.

The following **60 questions** are designed to make you reflect on what you learned. They are not here to intimidate you, quite the opposite: they are a mirror for you to see how much ground you have already covered. If you can answer most of them, even approximately, it means you have built a strong foundation for device driver programming.

Treat this as a self-assessment: keep a notebook open, write down your answers, and don't be afraid to go back to the text or labs if you feel uncertain. Remember: mastery is built by revisiting, practising, and questioning.

4.20.1 Questions

1. What makes C particularly suitable for operating system development compared to high-level languages?
2. Why is it helpful that C compiles down to predictable machine instructions on FreeBSD?
3. What role does the `main()` function play in a C program, and how is this different inside the kernel?
4. Why does every C source file in FreeBSD begin with `#include` lines?
5. What is the difference between declaring a variable and initialising it?
6. Why might leaving a variable uninitialised be more dangerous in kernel code than in user programs?
7. What happens if you assign a `char` into an `int` variable?
8. Give one example of a kernel structure where you expect to see a `char []` rather than a `char *`.
9. What does the `%` operator do, and why is it common in buffer management?
10. Why should kernel code avoid “clever” chained assignments such as `a = b = c = 0;`?
11. How does operator precedence affect the expression `x + y << 2`?
12. Why is `==` not the same as `=`, and what bug appears if you confuse them?
13. What danger exists when omitting braces `{}` in `if` statements?
14. Why are `switch` statements often preferred over many `if/else if` chains in FreeBSD?
15. In loops, what is the functional difference between `break` and `continue`?
16. How does a `for` loop make buffer iteration safer than `while(1)` with manual updates?
17. Why should you mark parameters as `const` where possible in driver helper functions?
18. What is an inline function, and why might it be better than a macro?
19. How does returning error codes like `ENXIO` help standardise driver behaviour?
20. What does it mean that function parameters are passed “by value” in C?
21. Why does passing a pointer to a struct give the illusion of “by reference”?

22. How does this behaviour influence the design of `probe()` or `attach()` functions?
23. What could happen if you modify a pointer argument inside a function without informing the caller?
24. Why must C strings be null-terminated, and what happens if they are not?
25. In the kernel, why are fixed-length arrays sometimes safer than dynamically sized ones?
26. How is buffer overflow in a kernel array different from buffer overflow in a user program?
27. What mistake do beginners make when copying strings into fixed buffers?
28. What does the `*` operator do in the expression `*p = 10;`?
29. How does pointer arithmetic take into account the type of the pointer?
30. Why should you be careful when comparing two unrelated pointers?
31. What is the difference between a dangling pointer and a NULL pointer?
32. Why does FreeBSD require allocation flags like `M_WAITOK` or `M_NOWAIT`?
33. Why are structs heavily used in kernel drivers instead of sets of separate variables?
34. Why can copying a struct containing pointers be dangerous?
35. Why might you want to use `typedef` with a struct, and when should you avoid it?
36. How do structs in C mirror the design of kernel subsystems like `proc` or `ifnet`?
37. What problem do header guards solve in multi-file projects?
38. Why do FreeBSD headers often contain forward declarations (`struct foo;`)?
39. Why is it bad practice to put function definitions directly into header files?
40. How do modular `.c` and `.h` files make teamwork easier in kernel development?
41. At which compilation stage would a typo in a function name be detected?
42. What does the linker do with object files?
43. Why is compiling with `-Wall` recommended for beginners?
44. How does using `kgdb` or `lldb` differ from debugging in userland?
45. How can `#define` be used to enable or disable debugging messages?
46. What is the risk of using macros with side effects like `#define DOUB(X) X+X`?
47. Why do kernel headers rely on `#ifdef _KERNEL` sections?
48. What problem do `#pragma once` or header guards prevent?
49. How does conditional compilation let FreeBSD support many CPU architectures?
50. Why is `volatile` sometimes necessary when dealing with hardware registers?
51. Why is it important to always initialise local variables, even if you plan to overwrite them soon?
52. How does using fixed-width integer types (`uint32_t`) improve portability across architectures?
53. Why should you prefer descriptive variable names over single letters in kernel code?
54. What coding pattern in loops helps prevent “off-by-one” errors?
55. Why should every `malloc()` in kernel space have a corresponding `free()` in the unload path?
56. How can consistent indentation and braces improve long-term maintainability of drivers?
57. Why is it recommended to avoid “magic numbers” and instead use named constants or macros?
58. How can reading similar drivers in FreeBSD source guide you toward better style?
59. Why is it important to check the return value of every system or library call in kernel code?
60. How does following KNF (Kernel Normal Form) style make your contributions easier to review and accept?

4.21 Wrapping Up

You've reached the end of Chapter 4, and that is no small achievement. In this single chapter, you have gone from zero C knowledge to handling the same building blocks used daily in the FreeBSD kernel. Along the way, you wrote real programs, explored actual kernel code, and faced questions that sharpened both your memory and your reasoning.

The key takeaway is this: **C is not just a language; it is the medium through which the kernel communicates.** Every pointer you follow, every struct you design, and every header you include brings you closer to understanding how FreeBSD itself works inside.

As we close, remember that learning C is not about memorizing syntax. It is about cultivating precision, discipline, and a sense of curiosity. Those same qualities are what make great driver developers.

In the following chapters, you will apply these foundations in practice, structuring a FreeBSD driver, opening device files, and gradually working toward hardware interaction.

Keep these lessons close, revisit them often, and, above all, stay curious.

5 Understanding C for FreeBSD Kernel Programming

In the last chapter, you learned the **language of C**, including its vocabulary of variables and operators, its grammar of control flow and functions, and its tools such as arrays, pointers, and structures. With practice, you can now write and understand complete C programs. That was a huge milestone; you can *speak C*.

The FreeBSD kernel, however, speaks C with its own **dialect**: the same words, but with special rules, idioms, and constraints. A user-space program may call `malloc()`, `printf()`, or use floating-point numbers without a second thought. In kernel space, those choices are either unavailable or dangerous. Instead, you'll see `malloc(9)` with flags like `M_WAITOK`, kernel-specific string functions like `strncpy()`, and strict rules against recursion or floating point. Chapter 4 taught you the language; this chapter teaches you the dialect, so that your code will be understood, and accepted, inside the kernel.

This chapter is about making that shift. You'll see how kernel code adapts C to work under different conditions: no runtime library, limited stack space, and absolute demands for performance and safety. You'll discover the types, functions, and coding practices that every FreeBSD driver relies on, and you'll learn how to avoid the mistakes that even experienced C programmers make when they first step into kernel space. For a compact reference to the kernel C idioms and macros you will meet along the way, you can also consult **Appendix A**, which collects them in one place for quick lookup while you read.

By the end of this chapter, you won't just know C, you'll know how to **think in C the way the FreeBSD kernel thinks in C**, a mindset that will carry you through the rest of this book and into your own driver projects.

5.1 Reader Guidance: How to Use This Chapter

This chapter is both a **reference** and a **practical bootcamp** in kernel C programming.

Unlike the previous chapter, which introduced C from scratch, this one assumes you're already comfortable with the language and now focuses on the kernel-specific mindset and adaptations you must master.

The time you'll spend here depends on how deeply you engage:

- **Reading only:** Around **10-11 hours** to read all explanations and FreeBSD kernel examples at a comfortable pace.
- **Reading + labs:** Around **15-17 hours** if you compile and test each of the practical kernel modules as you go.
- **Reading + labs + challenges:** Around **18-22 hours or more** if you also complete the Challenge Exercises and explore the corresponding kernel sources in `/usr/src`.

5.1.1 How to Get the Most Out of This Chapter

- **Have your FreeBSD source tree ready.** Many examples reference real kernel files.
- **Practise in your lab environment.** The kernel modules you build are safe only inside the sandbox prepared earlier.
- **Take breaks and review.** Each section builds on the last. Pace yourself as you internalise the kernel's logic.
- **Treat defensive programming as a habit, not an option.** In kernel space, correctness is survival.

This chapter is your **field guide to kernel C**, a dense, hands-on, and essential preparation for the structural work that begins in Chapter 6.

5.2 Introduction

When I first stepped into kernel programming after years of user-space C development, I thought the transition would be straightforward. After all, C is C, right? I quickly discovered that kernel programming felt like visiting a foreign country where everyone speaks your language but with completely different customs, etiquette, and unspoken rules.

In user space, you have luxuries you might not even realize: a vast standard library, garbage collection (in some languages), virtual memory protection that forgives many mistakes, and debugging tools that can inspect your program's every move. The kernel strips all of this away. You're working directly with hardware, managing physical memory, and operating under constraints that would make a user-space program impossible.

5.2.1 Why Kernel C Is Different

The kernel lives in a fundamentally different world:

- **No standard library:** Functions like `printf()`, `malloc()`, and `strcpy()` either don't exist or work completely differently.
- **Limited stack space:** Where user programs might have megabytes of stack, kernel stacks are typically only 16 KB per thread on FreeBSD 14.3 for amd64 and arm64. That is four 4 KB pages, corresponding to the default `KSTACK_PAGES=4` set in `/usr/src/sys/amd64/include/param.h` and `/usr/src/sys/arm64/include/param.h`; kernels built with KASAN or KMSAN raise `KSTACK_PAGES` to six, or roughly 24 KB.
- **No floating point:** The kernel can't use floating-point operations without special handling, because it would interfere with user processes.
- **Atomic context:** Much of your code runs in contexts where it cannot sleep or be interrupted.
- **Shared state:** Everything you do affects the entire system, not just your program.

These aren't limitations; they're the constraints that make the kernel fast, reliable, and capable of running the entire system.

5.2.2 The Mental Shift

Learning kernel C isn't just about memorizing different function names. It's about developing a new mindset:

- **Paranoid programming:** Always assume the worst. Check every pointer, validate every parameter, handle every error.
- **Resource consciousness:** Memory is precious, stack space is limited, and CPU cycles matter.
- **System thinking:** Your code doesn't run in isolation; it's part of a complex system where one bug can bring everything down.

This might sound intimidating, but it's also empowering. Kernel programming gives you control over the machine at a level that few programmers ever experience.

5.2.3 What You'll Learn

This chapter will teach you:

- The data types and memory models used in the FreeBSD kernel
- How to handle strings and buffers safely in kernel space
- Function calling conventions and return patterns
- The restrictions that keep kernel code safe and fast
- Coding idioms that make your drivers robust and maintainable
- Defensive programming techniques that prevent subtle bugs
- How to read and understand real FreeBSD kernel code

By the end, you'll be able to look at a kernel function and immediately understand not just what it does, but why it's written that way.

Let's begin with the foundation: understanding how the kernel organizes data.

5.3 Kernel-Specific Data Types

When you write user-space C programs, you might casually use types like `int`, `long`, or `char *` without much thought about their precise size or behavior. In the kernel, this casual approach can lead to bugs that are subtle, dangerous, and often system-dependent. FreeBSD provides a rich set of **kernel-specific data types** designed to make code portable, safe, and clear about its intentions.

5.3.1 Why Standard C Types Aren't Enough

Consider this seemingly innocent user-space code:

```
int file_size = get_file_size(filename);
if (file_size > 1000000) {
    // Handle large file
}
```

This works fine until you encounter a file larger than 2GB on a 32-bit system, where `int` is typically 32 bits and can only hold values up to about 2.1 billion. Suddenly, a 3GB file appears to have a negative size due to integer overflow.

In the kernel, this kind of problem is amplified because:

- Your code must work across different architectures (32-bit, 64-bit)
- Data corruption can affect the entire system
- Performance-critical code paths can't afford to check for overflow at runtime

FreeBSD solves this with explicit, fixed-size types that make your intentions clear.

5.3.2 Fixed-Size Integer Types

FreeBSD provides types that are guaranteed to be the same size regardless of architecture:

```
#include <sys/types.h>

uint8_t  flags;           // Always 8 bits (0-255)
uint16_t port_number;     // Always 16 bits (0-65535)
uint32_t ip_address;      // Always 32 bits
uint64_t file_offset;     // Always 64 bits
```

Here is an illustrative layout that uses explicit-width types, the shape you'll see in many kernel headers for protocol structures:

```
struct my_packet_header {
    uint8_t  version;      /* protocol version, always 1 byte */
    uint8_t  flags;        /* feature flags, always 1 byte */
    uint16_t length;       /* total length, always 2 bytes */
    uint32_t sequence;     /* sequence number, always 4 bytes */
    uint64_t timestamp;    /* timestamp, always 8 bytes */
};
```

Notice how every field uses an explicit-width type. This ensures the structure is exactly the same size whether you compile on a 32-bit or 64-bit system, or on little-endian and big-endian machines.

The real `struct ip` in `/usr/src/sys/netinet/ip.h` is shaped a little differently for historical reasons: it uses `u_char`, `u_short`, and bitfields (because IP predates `<stdint.h>`), but the goal is the same, every field has a fixed, portable width. Open that file and take a look when you're curious.

5.3.3 System-Specific Size Types

For sizes, lengths, and memory-related values, FreeBSD provides types that adapt to the system's capabilities:

```
size_t    buffer_size;    // Size of objects in bytes
ssize_t   bytes_read;     // Signed size, can indicate errors
off_t     file_position;  // File offsets, can be very large
```

Consider this illustrative loop:

```
static int
flush_until(struct my_queue *q, int target)
{
    int flushed = 0;

    while (flushed < target && !my_queue_empty(q)) {
        my_queue_flush_one(q);
        flushed++;
    }
    return (flushed);
}
```

The function returns `int` for the count of flushed items. If it dealt with memory sizes directly, it would use `size_t` instead so the value couldn't silently truncate on systems with very large buffers. You can see the same `int` convention in functions like `flushbufqueues()` in `/usr/src/sys/kern/vfs_bio.c`, which returns the number of buffers it actually flushed.

5.3.4 Pointer and Address Types

The kernel frequently needs to work with memory addresses and pointers in ways that user-space programs rarely encounter:

```
vm_offset_t    virtual_addr;    // Virtual memory address
vm_paddr_t     physical_addr;   // Physical memory address
uintptr_t      addr_as_int;     // Address stored as integer
```

From `/usr/src/sys/vm/vm_page.c`, here's how FreeBSD looks up a page within a VM object:

```
vm_page_t
vm_page_lookup(vm_object_t object, vm_pindex_t pindex)
{
    VM_OBJECT_ASSERT_LOCKED(object);
    return (vm_radix_lookup(&object->rtree, pindex));
}
```

The `vm_pindex_t` type represents a page index in a virtual memory object, and `vm_page_t` is a pointer to a page structure. These typedefs make the code's intent clear and ensure portability across different memory architectures.

5.3.5 Time and Timing Types

The kernel has sophisticated requirements for time measurement:

```
sbintime_t    precise_time;    // High-precision system time
time_t        unix_time;       // Standard Unix timestamp
int           ticks;           // System timer ticks since boot
```

From `/usr/src/sys/kern/kern_tc.c`, the kernel exposes several helpers that return the current time in different precisions:

```
void
getnanotime(struct timespec *tsp)
{
    GETTHMEMBER(tsp, th_nanotime);
}
```

The `GETTHMEMBER` macro expands to a small loop that reads the current “timehands” structure with the proper atomic and memory-barrier discipline, so that `getnanotime()` returns a consistent snapshot of the system clock even while another CPU is updating it. We will look at atomics and memory barriers later in this chapter.

5.3.6 Device and Resource Types

When writing drivers, you’ll encounter types specific to hardware interaction:

```
device_t      dev;              // Device handle
bus_addr_t    hw_address;       // Hardware bus address
bus_size_t    reg_size;         // Size of hardware register region
```

5.3.7 Boolean and Status Types

The kernel provides clear types for boolean values and operation results:

```
bool          success;          /* C99 boolean (true/false) */
int           error_code;       /* errno-style codes; 0 means success */
```

Throughout the kernel you will see `int` used as the universal “success or errno” return type, and `bool` reserved for truly two-valued conditions. The convention is: a function that can fail returns an `int` error code, and a function that simply answers “yes or no” returns `bool`.

5.3.8 Hands-On Lab: Exploring Kernel Types

Let's create a simple kernel module that demonstrates these types:

1. Create a file called `types_demo.c`:

```
/*
 * types_demo.c - Demonstrate FreeBSD kernel data types
 */
#include <sys/param.h>
#include <sys/kernel.h>
#include <sys/module.h>
#include <sys/systm.h>
#include <sys/types.h>

static int
types_demo_load(module_t mod, int cmd, void *arg)
{
    switch (cmd) {
    case MOD_LOAD:
        printf("=== FreeBSD Kernel Data Types Demo ===\n");

        /* Fixed-size types */
        printf("uint8_t size: %zu bytes\n", sizeof(uint8_t));
        printf("uint16_t size: %zu bytes\n", sizeof(uint16_t));
        printf("uint32_t size: %zu bytes\n", sizeof(uint32_t));
        printf("uint64_t size: %zu bytes\n", sizeof(uint64_t));

        /* System types */
        printf("size_t size: %zu bytes\n", sizeof(size_t));
        printf("off_t size: %zu bytes\n", sizeof(off_t));
        printf("time_t size: %zu bytes\n", sizeof(time_t));

        /* Pointer types */
        printf("uintptr_t size: %zu bytes\n", sizeof(uintptr_t));
        printf("void* size: %zu bytes\n", sizeof(void *));

        printf("Types demo module loaded successfully.\n");
        break;

    case MOD_UNLOAD:
        printf("Types demo module unloaded.\n");
        break;

    default:
```

```
        return (EOPNOTSUPP);
    }

    return (0);
}

static moduledata_t types_demo_mod = {
    "types_demo",
    types_demo_load,
    NULL
};

DECLARE_MODULE(types_demo, types_demo_mod, SI_SUB_DRIVERS, SI_ORDER_MIDDLE);
MODULE_VERSION(types_demo, 1);
```

2. Create a Makefile:

```
# Makefile for types_demo kernel module
KMOD=    types_demo
SRCS=    types_demo.c

.include <bsd.kmod.mk>
```

3. Build and load the module:

```
% make clean && make
% sudo kldload ./types_demo.ko
% dmesg | tail -10
% sudo kldunload types_demo
```

You should see output showing the sizes of different kernel types on your system.

5.3.9 Common Type Mistakes to Avoid

Using int for sizes: Don't use `int` for memory sizes or array indices. Use `size_t` instead.

```
/* Wrong */
int buffer_size = malloc_size;

/* Right */
size_t buffer_size = malloc_size;
```

Mixing signed and unsigned: Be careful when comparing signed and unsigned values.


```
/* Dangerous - can cause infinite loops */
int i;
size_t count = get_count();
for (i = count - 1; i >= 0; i--) {
    /* If count is 0, i becomes SIZE_MAX */
}

/* Better */
size_t i;
size_t count = get_count();
for (i = count; i > 0; i--) {
    /* Process element i-1 */
}
```

Assuming pointer sizes: Never assume a pointer fits in an `int` or `long`.

```
/* Wrong on 64-bit systems where int is 32-bit */
int addr = (int)pointer;

/* Right */
uintptr_t addr = (uintptr_t)pointer;
```

5.3.10 Summary

Kernel-specific data types aren't just about being precise, they're about writing code that:

- Works correctly across different architectures
- Clearly expresses its intentions
- Avoids subtle bugs that can crash the system
- Uses the same interfaces as the rest of the kernel

In the next section, we'll explore how the kernel manages the memory these types live in a world where `malloc()` has flags and every allocation must be carefully planned.

5.4 Memory in Kernel Space

If kernel data types are the vocabulary of kernel C, then memory management is its grammar, the rules that determine how everything fits together. In user space, memory management often feels automatic: you call `malloc()`, use the memory, call `free()`, and trust the system to handle the details. In the kernel, memory is a precious, carefully managed resource where every allocation decision affects the entire system's performance and stability.

5.4.1 The Kernel Memory Landscape

The FreeBSD kernel divides memory into distinct regions, each with its own purpose and constraints:

Kernel text: The kernel's executable code, typically read-only and shared. **Kernel data:** Global variables and static data structures. **Kernel stack:** Limited space for function calls and local variables (typically 16KB per thread on FreeBSD 14.3 for amd64 and arm64; see `KSTACK_PAGES` in `/usr/src/sys/<arch>/include/param.h`). **Kernel heap:** Dynamically allocated memory for buffers, data structures, and temporary storage.

Unlike user processes, the kernel can't simply request more memory from the operating system; it *is* the operating system. Every byte must be accounted for, and running out of kernel memory can bring the entire system to its knees.

5.4.2 `malloc(9)`: The Kernel's Memory Allocator

The kernel provides its own `malloc()` function, but it's quite different from the user-space version. Here's the signature from `sys/sys/malloc.h`:

```
void *malloc(size_t size, struct malloc_type *type, int flags);
void free(void *addr, struct malloc_type *type);
```

A simple illustrative pattern you will recognise all over the kernel looks like this:

```
struct my_object *
my_object_alloc(int id)
{
    struct my_object *obj;

    /* Allocate and zero the structure in one step. */
    obj = malloc(sizeof(*obj), M_DEVBUF, M_WAITOK | M_ZERO);

    /* Initialise non-zero fields. */
    obj->id = id;
    TAILQ_INIT(&obj->children);

    return (obj);
}
```

The real `vfs_mount_alloc()` in `/usr/src/sys/kern/vfs_mount.c` uses a UMA zone (`uma_zalloc(mount_zone, M_WAITOK)`) instead of `malloc()` because `struct mount` is allocated often enough to justify a dedicated object cache. We will look at UMA zones in a few pages; for now, notice the overall rhythm: allocate, zero, initialise lists and non-zero fields, return.

5.4.3 Memory Types: Organizing Allocations

The `M_MOUNT` parameter is a **memory type**; a way to categorize allocations for debugging and resource tracking. FreeBSD defines dozens of these types in `sys/sys/malloc.h`:

```
MALLOC_DECLARE(M_DEVBUF);    /* Device driver buffers */
MALLOC_DECLARE(M_TEMP);      /* Temporary allocations */
MALLOC_DECLARE(M_MOUNT);     /* Filesystem mount structures */
MALLOC_DECLARE(M_VNODE);     /* Vnode structures */
MALLOC_DECLARE(M_CACHE);     /* Dynamically allocated cache */
```

You can see the system's current memory usage by type:

```
% vmstat -m
```

This shows you exactly how much memory each subsystem is using, invaluable for debugging memory leaks or understanding system behaviour.

5.4.4 Allocation Flags: Controlling Behavior

The `flags` parameter controls how the allocation behaves. The most important flags are:

M_WAITOK: The allocation can sleep waiting for memory. This is the default for most kernel code.

M_NOWAIT: The allocation must not sleep. Returns `NULL` if memory isn't immediately available. Used in interrupt context or when holding certain locks.

M_ZERO: Clear the allocated memory to zero. Similar to `calloc()` in user space.

M_USE_RESERVE: Use emergency memory reserves. Only for critical system operations.

Here is the shape of a typical allocation path using `M_WAITOK` and `M_ZERO`:

```
static struct my_softc *
my_softc_alloc(u_char type)
{
    struct my_softc *sc;

    sc = malloc(sizeof(*sc), M_DEVBUF, M_WAITOK | M_ZERO);
    /*
     * With M_WAITOK the kernel will sleep until memory is available,
     * so a NULL return is not expected. The defensive NULL check is
     * still a good habit, and is mandatory with M_NOWAIT.
     */
    sc->type = type;
    return (sc);
}
```

In `/usr/src/sys/net/if.c`, the kernel's `if_alloc(u_char type)` is a thin wrapper over `if_alloc_domain()`, which is where the real allocation happens. The internal helper uses the same `M_WAITOK | M_ZERO` pattern shown above.

5.4.5 The Critical Difference: Sleep vs. No-Sleep Contexts

One of the most important concepts in kernel programming is understanding when your code can and cannot sleep. **Sleeping** means voluntarily giving up the CPU to wait for something more memory to become available, I/O to complete, or a lock to be released.

Sleep-safe contexts: Regular kernel threads, system call handlers, and most driver entry points can sleep.

Atomic contexts: Interrupt handlers, spin lock holders, and some callback functions cannot sleep.

Using the wrong allocation flag can cause deadlocks or kernel panics:

```
/* In an interrupt handler - WRONG! */
void
my_interrupt_handler(void *arg)
{
    char *buffer;

    /* This can panic the system! */
    buffer = malloc(1024, M_DEVBUF, M_WAITOK);
    /* ... */
}

/* In an interrupt handler - RIGHT */
void
my_interrupt_handler(void *arg)
{
    char *buffer;

    buffer = malloc(1024, M_DEVBUF, M_NOWAIT);
    if (buffer == NULL) {
        /* Handle allocation failure gracefully */
        return;
    }
    /* ... */
}
```

5.4.6 Memory Zones: High-Performance Allocation

For frequently allocated objects of the same size, FreeBSD provides **UMA (Universal Memory Allocator)** zones. These are more efficient than general-purpose `malloc()` for repeated allocations:

```
#include <vm/uma.h>

uma_zone_t my_zone;

/* Initialize zone during module load */
my_zone = uma_zcreate("myobjs", sizeof(struct my_object),
    NULL, NULL, NULL, NULL, UMA_ALIGN_PTR, 0);

/* Allocate from zone */
struct my_object *obj = uma_zalloc(my_zone, M_WAITOK);

/* Free to zone */
uma_zfree(my_zone, obj);

/* Destroy zone during module unload */
uma_zdestroy(my_zone);
```

A condensed pattern showing how a subsystem creates a UMA zone at boot:

```
static uma_zone_t my_zone;

static void
my_subsystem_init(void)
{
    my_zone = uma_zcreate("MYZONE", sizeof(struct my_object),
        NULL, NULL, NULL, NULL, UMA_ALIGN_PTR, 0);
}
```

The real `procinit()` in `/usr/src/sys/kern/kern_proc.c` does exactly this for `struct proc`, passing real constructor/destructor/init/finish callbacks (`proc_ctor`, `proc_dtor`, `proc_init`, `proc_fini`) so the kernel can keep preallocated `proc` structures warm in the zone. The argument order for `uma_zcreate()` is name, size, ctor, dtor, init, fini, alignment, flags.

5.4.7 Stack Considerations: The Kernel's Precious Resource

User-space programs typically have stack sizes measured in megabytes. Kernel stacks are much smaller, typically 16KB per thread on FreeBSD 14.3 (four pages on amd64 and arm64; see `KSTACK_PAGES` in `/usr/src/sys/amd64/include/param.h`). This includes space for interrupt handling. The practical consequences:

Avoid large local arrays:

```
/* BAD - can overflow kernel stack */
void
bad_function(void)
```

```

{
    char huge_buffer[8192]; /* Dangerous! */
    /* ... */
}

/* GOOD - allocate on heap */
void
good_function(void)
{
    char *buffer;

    buffer = malloc(8192, M_TEMP, M_WAITOK);
    if (buffer == NULL) {
        return (ENOMEM);
    }

    /* Use buffer... */

    free(buffer, M_TEMP);
}

```

Limit recursion depth: Deep recursion can quickly exhaust the stack.

Be conscious of struct sizes: Large structures should be allocated dynamically, not as local variables.

5.4.8 Memory Barriers and Cache Coherency

In multiprocessor systems, the kernel must sometimes ensure that memory operations happen in a specific order. This is accomplished with **memory barriers**:

```

#include <machine/atomic.h>

/* Ensure all previous writes complete before this write */
atomic_store_rel_int(&status_flag, READY);

/* Ensure this read happens before subsequent operations */
int value = atomic_load_acq_int(&shared_counter);

```

From `/usr/src/sys/kern/kern_synch.c`, the real `wakeup_one()` is pleasantly short:

```

void
wakeup_one(const void *ident)
{
    int wakeup_swapper;

```

```
sleepq_lock(ident);
wakeup_swapper = sleepq_signal(ident, SLEEPQ_SLEEP | SLEEPQ_DROP, 0, 0);
if (wakeup_swapper)
    kick_proc0();
}
```

All the detail (finding the sleep queue, picking a thread to wake, dropping the lock) is hidden inside `sleepq_signal()`. This is a recurring pattern: the public function reads like a short declarative sentence, and the interesting work lives in a handful of well-tested helpers.

5.4.9 Hands-On Lab: Kernel Memory Management

Let's create a kernel module that demonstrates memory allocation patterns:

1. Create `memory_demo.c`:

```
/*
 * memory_demo.c - Demonstrate kernel memory management
 */
#include <sys/param.h>
#include <sys/kernel.h>
#include <sys/module.h>
#include <sys/system.h>
#include <sys/malloc.h>
#include <vm/uma.h>

MALLOC_DEFINE(M_DEMO, "demo", "Memory demo allocations");

static uma_zone_t demo_zone;

struct demo_object {
    int id;
    char name[32];
};

static int
memory_demo_load(module_t mod, int cmd, void *arg)
{
    void *ptr1, *ptr2, *ptr3;
    struct demo_object *obj;

    switch (cmd) {
    case MOD_LOAD:
```

```
printf("=== Kernel Memory Management Demo ===\n");

/* Basic allocation */
ptr1 = malloc(1024, M_DEMO, M_WAITOK);
printf("Allocated 1024 bytes at %p\n", ptr1);

/* Zero-initialized allocation */
ptr2 = malloc(512, M_DEMO, M_WAITOK | M_ZERO);
printf("Allocated 512 zero bytes at %p\n", ptr2);

/* No-wait allocation (might fail) */
ptr3 = malloc(2048, M_DEMO, M_NOWAIT);
if (ptr3) {
    printf("No-wait allocation succeeded at %p\n", ptr3);
} else {
    printf("No-wait allocation failed (memory pressure)\n");
}

/* Create a UMA zone */
demo_zone = uma_zcreate("demo_objects", sizeof(struct demo_object),
    NULL, NULL, NULL, NULL, UMA_ALIGN_PTR, 0);

if (demo_zone) {
    obj = uma_zalloc(demo_zone, M_WAITOK);
    obj->id = 42;
    strncpy(obj->name, "demo_object", sizeof(obj->name));
    printf("Zone allocation: object %d named '%s' at %p\n",
        obj->id, obj->name, obj);
    uma_zfree(demo_zone, obj);
}

/* Clean up basic allocations */
free(ptr1, M_DEMO);
free(ptr2, M_DEMO);
if (ptr3) {
    free(ptr3, M_DEMO);
}

printf("Memory demo loaded successfully.\n");
break;

case MOD_UNLOAD:
    if (demo_zone) {
        uma_zdestroy(demo_zone);
    }
}
```



```
    }
    printf("Memory demo module unloaded.\n");
    break;

default:
    return (EOPNOTSUPP);
}

return (0);
}

static moduledata_t memory_demo_mod = {
    "memory_demo",
    memory_demo_load,
    NULL
};

DECLARE_MODULE(memory_demo, memory_demo_mod, SI_SUB_DRIVERS, SI_ORDER_MIDDLE);
MODULE_VERSION(memory_demo, 1);
```

2. Build and test:

```
% make clean && make
% sudo kldload ./memory_demo.ko
% dmesg | tail -10
% sudo kldunload memory_demo
```

5.4.10 Memory Debugging and Leak Detection

FreeBSD provides excellent tools for debugging memory issues:

INVARIANTS kernel: Enable debugging checks in kernel data structures.

vmstat -m: Show memory usage by type.

vmstat -z: Show UMA zone statistics.

```
% vmstat -m | grep M_DEMO
% vmstat -z | head -20
```

5.4.11 Safe String and Memory Operations in Kernel Space

In user programs you might freely use `strcpy()`, `memcpy()`, or `sprintf()`. In the kernel, these are potential sources of crashes and buffer overflows. The kernel substitutes them with safer, bounded functions designed for predictable behaviour.

5.4.11.1 Why Safety Functions Are Needed

- The kernel cannot rely on virtual memory protection to catch overruns.
- Most buffers are fixed-size and often map directly to hardware or shared memory.
- Crashing or corrupting memory in kernel space compromises the whole system.

5.4.11.2 Common Safe Alternatives

Category	Unsafe Function	Kernel-safe Equivalent	Notes
String copy	<code>strcpy()</code>	<code>strncpy(dest, src, size)</code>	Guarantees NUL-termination
String concat	<code>strcat()</code>	<code>strlcat(dest, src, size)</code>	Prevents overflow
Memory copy	<code>memcpy()</code>	<code>bcopy(src, dest, len)</code>	Used widely; same semantics
Memory clear	<code>memset()</code>	<code>bzero(dest, len)</code>	Zeroes buffers explicitly
Formatted print	<code>sprintf()</code>	<code>snprintf(dest, size, fmt, ...)</code>	Bounds checking
User <-> Kernel copy	N/A	<code>copyin()</code> , <code>copyout()</code>	Transfer data across address spaces

A typical “clear and copy” pattern you’ll see throughout the kernel:

```
struct my_record mr;

bzero(&mr, sizeof(mr));
strncpy(mr.name, src, sizeof(mr.name));
```

And a driver handling user requests through an ioctl-style path:

```
error = copyin(uap->data, &local, sizeof(local));
if (error != 0)
    return (error);
```

`copyin()` safely copies data from user memory to kernel memory, returning an `errno` on failure (typically `EFAULT` if the user pointer is bad). Its sibling `copyout()` performs the reverse. These functions validate access rights and handle page faults safely, so they are the only correct way to cross the user-kernel boundary.

5.4.11.3 Best Practices

1. Always pass the **destination buffer size** to string functions.
2. Prefer `strncpy()` and `snprintf()`; they are consistent across the kernel.
3. Never assume user memory is valid; always use `copyin()/copyout()`.
4. Use `bzero()` or `explicit_bzero()` to clear sensitive data like keys.
5. Treat any pointer from user space as **untrusted input**.

5.4.11.4 Hands-On Mini-Lab

Modify your previous `memory_demo.c` module to test safe string handling:

```
char buf[16];
bzero(buf, sizeof(buf));
strncpy(buf, "FreeBSD-Kernel", sizeof(buf));
printf("String safely copied: %s\n", buf);
```

Compiling and loading the kernel will print your message, proving a safe bounded copy.

5.4.12 Summary

Kernel memory management requires discipline and understanding:

- Use appropriate allocation flags (`M_WAITOK` vs `M_NOWAIT`)
- Always specify a memory type for tracking
- Check return values, even with `M_WAITOK`
- Prefer UMA zones for frequent, same-size allocations
- Keep stack usage minimal
- Understand when your code can and cannot sleep

Memory bugs in the kernel are system-wide disasters. The defensive programming techniques we'll cover later in this chapter will help you avoid them.

In the next section, we'll explore how the kernel handles text and binary data, another area where user-space assumptions don't apply.

5.5 Error Handling Patterns in Kernel C

In user-space programming, you might throw exceptions or print messages when something goes wrong. In kernel programming, there are no exceptions and no runtime safety nets. A single unchecked error can lead to undefined behaviour or a full system panic. Therefore, error handling in kernel C is not an afterthought; it's a discipline.

5.5.1 Return Values: Zero Means Success

By long-standing UNIX and FreeBSD convention:

- 0 -> Success
- Non-zero -> Failure (often an `errno`-style code such as `EIO`, `EINVAL`, `ENOMEM`)

Consider this illustrative function that follows the same convention you'll see throughout `/usr/src/sys/kern/`:

```
int
my_operation(struct my_object *obj)
{
    int error;

    if (obj == NULL)
        return (EINVAL);    /* invalid argument */

    error = do_dependent_step(obj);
    if (error != 0)
        return (error);    /* propagate cause */

    do_final_step(obj);
    return (0);            /* success */
}
```

The function clearly signals failure conditions using standard `errno` codes (`EINVAL`, `ENOMEM`, and so on), and forwards unexpected errors from helper functions without reinterpreting them.

Tip: Always propagate upstream errors instead of silently ignoring them. It allows higher-level subsystems to decide what to do next.

5.5.2 Using `goto` for Cleanup Paths

Beginners sometimes fear the `goto` keyword, but in kernel code it is the standard idiom for structured cleanup. It avoids deep nesting and guarantees that every resource is freed exactly once.

A pedagogical sketch of the same pattern, inspired by the open path in `/usr/src/sys/kern/vfs_syscalls.c`:

```
int
my_setup(struct thread *td, struct my_args *uap)
{
    struct file *fp = NULL;
    struct resource *res = NULL;
    int error;

    error = falloc(td, &fp, NULL, 0);
    if (error != 0)
        goto fail;

    res = acquire_resource(uap->id);
    if (res == NULL) {
        error = ENXIO;
        goto fail;
    }
}
```

```
/* Success path: hand ownership over to the caller. */
return (0);

fail:
    if (res != NULL)
        release_resource(res);
    if (fp != NULL)
        fdrop(fp, td);
    return (error);
}
```

Each allocation step is followed by an immediate check. If something fails, execution jumps to a single cleanup label. This pattern keeps kernel functions readable and leak-free.

5.5.3 Defensive Strategy

1. **Check every pointer** before dereferencing.
2. **Validate user input** received from `ioctl()`, `read()`, `write()`.
3. **Propagate error codes**, don't reinterpret them unless necessary.
4. **Free in reverse order of allocation**.
5. **Avoid partial initialisation** - always initialise before use.

5.5.4 Summary

- `return (0);` -> success
- Return `errno` codes for specific failures
- Use `goto fail:` to simplify cleanup
- Never ignore an error path

These conventions make FreeBSD's kernel code easy to audit and prevent subtle memory or resource leaks.

5.6 Assertions and Diagnostics in the Kernel

Kernel developers rely on lightweight diagnostic tools built directly into C macros. These don't replace debuggers; they complement them.

5.6.1 `KASSERT()` - Enforcing Invariants

`KASSERT(expr, message)` halts the kernel (in debug builds) if the condition is false.

```
KASSERT(m != NULL, ("vm_page_lookup: NULL page pointer"));
```

If this assertion fails, the kernel prints the message and triggers a panic, revealing the file and line number. Assertions are invaluable for detecting logic errors early.

Use assertions to verify **things that should never happen** under correct logic, not for routine error checking.

5.6.2 panic() - The Last Resort

`panic(const char *fmt, ...)` stops the system and dumps state for post-mortem analysis. A typical use looks like this:

```
if (mp->ks_magic != M_MAGIC)
    panic("my_subsystem: bad magic 0x%x on %p", mp->ks_magic, mp);
```

A panic is catastrophic, but sometimes necessary to prevent data corruption. Use it for impossible states, corrupted invariants, or situations where letting the kernel continue would risk destroying user data.

5.6.3 printf() and Friends

In kernel space you still have `printf()`, but it writes to the console or system log:

```
printf("Driver initialised: %s\n", device_get_name(dev));
```

For user-facing messages, use:

- `uprntf()` prints to the terminal of the calling user.
- `device_printf(dev, ...)` prefixes messages with the device name (used in drivers).

An illustrative attach-time log from a driver:

```
device_printf(dev, "attached, speed: %d Mbps\n", speed);
```

The output appears in `dmesg` as something like `em0: attached, speed: 1000 Mbps`, which makes it easy to spot in a log full of messages from many different devices.

5.6.4 Tracing with CTRn() and SDT_PROBE()

Advanced diagnostics use macros like `CTR0`, `CTR1`, ... to emit trace points, or the **Statically Defined Tracing (SDT)** framework (DTrace):

```
SDT_PROBE1(proc, , , create, p);
```

These integrate with DTrace for live kernel instrumentation.

5.6.5 Summary

- Use `KASSERT()` for logic invariants.
- Use `panic()` only for unrecoverable conditions.
- Prefer `device_printf()` or `printf()` for diagnostics.
- Tracing macros help observe behaviour without stopping the kernel.

Proper diagnostics are part of writing reliable, maintainable drivers, and they make debugging far easier later.

5.7 Strings and Buffers in the Kernel

String handling in user-space C is fraught with pitfalls: buffer overflows, null terminator bugs, and encoding issues. In the kernel, these problems are magnified because a single mistake can compromise system security or crash the entire machine. FreeBSD provides a comprehensive set of string and buffer manipulation functions designed to make kernel code both safer and more efficient than user-space equivalents.

5.7.1 Why Standard String Functions Don't Work

In user space, you might write:

```
char buffer[256];
strcpy(buffer, user_input); /* Dangerous! */
```

This code is problematic because:

- `strcpy()` doesn't check buffer boundaries
- If `user_input` is longer than 255 characters, memory corruption occurs
- In the kernel, this could overwrite critical data structures

The kernel needs functions that:

- Always respect buffer boundaries
- Handle partially-filled buffers gracefully
- Work efficiently with both kernel and user data
- Provide clear error indication

5.7.2 Safe String Copying: `strncpy()` and `strlcat()`

FreeBSD uses `strncpy()` and `strlcat()` instead of the dangerous `strcpy()` and `strcat()`:

```
size_t strncpy(char *dst, const char *src, size_t size);
size_t strlcat(char *dst, const char *src, size_t size);
```

A concise pattern that uses `strncpy()` the way kernel code typically does:

```
struct my_label {
    char    name[MAXHOSTNAMELEN];
};

static int
my_label_set(struct my_label *lbl, const char *src, size_t srclen)
{
    if (srclen >= sizeof(lbl->name))
        return (ENAMETOOLONG);

    /*
     * strncpy always NUL-terminates the destination and never
     * writes past sizeof(lbl->name) bytes, regardless of how long
     * src actually is.
     */
    strncpy(lbl->name, src, sizeof(lbl->name));
    return (0);
}
```

Key advantages of `strncpy()`:

- **Always null-terminates** the destination buffer
- **Never overflows** the destination buffer
- **Returns the length** of the source string (useful for detecting truncation)
- **Works correctly** even if source and destination overlap

5.7.3 String Length and Validation: `strlen()` and `strnlen()`

The kernel provides both standard `strlen()` and the safer `strnlen()`:

```
size_t strlen(const char *str);
size_t strnlen(const char *str, size_t maxlen);
```

An illustrative sanity check on the length of a user-supplied path string:

```
static int
my_validate_path(const char *path)
{
    if (strnlen(path, PATH_MAX) >= PATH_MAX)
        return (ENAMETOOLONG);
    return (0);
}
```


The `strlen()` function prevents runaway length calculations on malformed strings that might not be null-terminated.

5.7.4 Memory Operations: `memcpy()`, `memset()`, and `memcmp()`

While string functions work with null-terminated text, memory functions work with binary data of explicit length:

```
void *memcpy(void *dst, const void *src, size_t len);
void *memset(void *ptr, int value, size_t len);
int memcmp(const void *ptr1, const void *ptr2, size_t len);
```

An illustrative sketch that uses the same binary-safe primitives you'll see across network code:

```
static void
my_forward(struct mbuf *m)
{
    struct ip *ip = mtod(m, struct ip *);
    struct in_addr dest;

    /* Copy the destination address into a local buffer. */
    memcpy(&dest, &ip->ip_dst, sizeof(dest));

    /* Zero a header annotation before we fill it in again. */
    memset(&m->m_pkthdr.PH_loc, 0, sizeof(m->m_pkthdr.PH_loc));

    /* ... forward the packet ... */
}
```

`memcpy()` and `memset()` take explicit lengths and work on arbitrary binary data, which is exactly what protocol code needs.

5.7.5 User Space Data Access: `copyin()` and `copyout()`

One of the kernel's most critical responsibilities is safely transferring data between kernel space and user space. You cannot simply dereference user pointers; they might be invalid, point to kernel memory, or cause page faults.

```
int copyin(const void *udaddr, void *kaddr, size_t len);
int copyout(const void *kaddr, void *udaddr, size_t len);
```

From `/usr/src/sys/kern/sys_generic.c`:

```
int
sys_read(struct thread *td, struct read_args *uap)
{
    struct uio auio;
    struct iovec aiov;
    int error;

    if (uap->nbyte > IOSIZE_MAX)
        return (EINVAL);
    aiov.iov_base = uap->buf;
    aiov.iov_len = uap->nbyte;
    auio.uio_iov = &aiov;
    auio.uio_iovcnt = 1;
    auio.uio_resid = uap->nbyte;
    auio.uio_segflg = UIO_USERSPACE;
    error = kern_readv(td, uap->fd, &auio);
    return (error);
}
```

The kernel uses `struct uio` (User I/O) to safely describe data transfers. The `uio_segflg` field tells the system whether the buffer addresses live in kernel space (`UIO_SYSSPACE`) or user space (`UIO_USERSPACE`), and the copyin/copyout machinery invoked deeper in the stack reads that flag to pick the safe copy primitive.

5.7.6 String Formatting: `sprintf()` vs `snprintf()`

The kernel provides both `sprintf()` and the safer `snprintf()`:

```
int sprintf(char *str, const char *format, ...);
int snprintf(char *str, size_t size, const char *format, ...);
```

Illustrative pattern for building a bounded string in a fixed-size buffer:

```
void
format_device_label(char *buf, size_t bufsz, const char *name, int unit)
{
    /* snprintf never writes past buf[bufsz - 1], and always NUL-terminates. */
    snprintf(buf, bufsz, "%s%d", name, unit);
}
```

Always prefer `snprintf()` over `sprintf()` to avoid buffer overflows, and pass `sizeof(buf)` (or an explicit size argument, as above) so that the function knows the real capacity of the destination.

5.7.7 Buffer Management: mbuf Chains

Network code and some I/O operations use **mbufs** (memory buffers) for efficient data handling. These are chainable buffers that can represent data scattered across multiple memory regions:

```
#include <sys/mbuf.h>

struct mbuf *m;
m = m_get(M_WAITOK, MT_DATA); /* Allocate an mbuf */

/* Add data to the mbuf */
m->m_len = snprintf(mtod(m, char *), MLEN, "Hello, network!");

/* Free the mbuf */
m_freem(m);
```

An illustrative mbuf lifecycle:

```
static int
my_build_packet(struct mbuf **mp, size_t optlen)
{
    struct mbuf *m;

    m = m_get(M_NOWAIT, MT_DATA);
    if (m == NULL)
        return (ENOMEM);

    if (optlen > MLEN) {
        /* Too big to fit in a single mbuf; caller should chain. */
        m_freem(m);
        return (EINVAL);
    }

    m->m_len = optlen;
    *mp = m;
    return (0);
}
```

Real network code such as `tcp_adoptions()` in `/usr/src/sys/netinet/tcp_output.c` builds TCP option strings into buffers that later end up in an mbuf chain. The detail worth internalising here is the pairing: `m_get()` on acquire, `m_freem()` on release.

5.7.8 Hands-On Lab: Safe String Handling

Let's create a kernel module that demonstrates safe string operations:

```
/*
 * strings_demo.c - Demonstrate kernel string handling
 */
#include <sys/param.h>
#include <sys/kernel.h>
#include <sys/module.h>
#include <sys/systm.h>
#include <sys/malloc.h>
#include <sys/libkern.h>

MALLOC_DEFINE(M_STRDEMO, "strdemo", "String demo buffers");

static int
strings_demo_load(module_t mod, int cmd, void *arg)
{
    char *buffer1, *buffer2;
    const char *test_string = "FreeBSD Kernel Programming";
    size_t len, copied;

    switch (cmd) {
    case MOD_LOAD:
        printf("=== Kernel String Handling Demo ===\n");

        buffer1 = malloc(64, M_STRDEMO, M_WAITOK | M_ZERO);
        buffer2 = malloc(32, M_STRDEMO, M_WAITOK | M_ZERO);

        /* Safe string copying */
        copied = strlcpy(buffer1, test_string, 64);
        printf("strlcpy: copied %zu chars: '%s'\n", copied, buffer1);

        /* Demonstrate truncation */
        copied = strlcpy(buffer2, test_string, 32);
        printf("strlcpy to small buffer: copied %zu chars: '%s'\n",
            copied, buffer2);
        if (copied >= 32) {
            printf("Warning: string was truncated!\n");
        }

        /* Safe string length */
        len = strnlen(buffer1, 64);
        printf("strnlen: length is %zu\n", len);

        /* Safe string concatenation */
        strlcat(buffer2, " rocks!", 32);
    }
}
```

```
    printf("strlcat result: '%s'\n", buffer2);

    /* Memory operations */
    memset(buffer1, 'X', 10);
    buffer1[10] = '\0';
    printf("memset result: '%s'\n", buffer1);

    /* Safe formatting */
    snprintf(buffer1, 64, "Module loaded at tick %d", ticks);
    printf("snprintf: '%s'\n", buffer1);

    free(buffer1, M_STRDEMO);
    free(buffer2, M_STRDEMO);

    printf("String demo completed successfully.\n");
    break;

case MOD_UNLOAD:
    printf("String demo module unloaded.\n");
    break;

default:
    return (EOPNOTSUPP);
}

return (0);
}

static moduledata_t strings_demo_mod = {
    "strings_demo",
    strings_demo_load,
    NULL
};

DECLARE_MODULE(strings_demo, strings_demo_mod, SI_SUB_DRIVERS, SI_ORDER_MIDDLE);
MODULE_VERSION(strings_demo, 1);
```

5.7.9 String Handling Best Practices

Always use safe functions: Prefer `strncpy()` over `strcpy()`, `snprintf()` over `sprintf()`.

Check buffer sizes: Use `strlen()` when you need to limit string length checks.

Validate user data: Never trust user-provided strings or lengths.

Handle truncation: Check return values from `strncpy()` and `snprintf()` to detect truncation.

Zero-initialize buffers: Use `M_ZERO` or `memset()` to ensure clean initial state.

5.7.10 Common String Pitfalls

Off-by-one errors: Remember that string buffers need space for the null terminator.

```
/* Wrong - no space for null terminator */
char name[8];
strcpy(name, "FreeBSD", 8); /* Only 7 chars fit + null */

/* Right */
char name[8];
strcpy(name, "FreeBSD", sizeof(name)); /* 7 chars + null = OK */
```

Integer overflow in length calculations:

```
/* Dangerous */
size_t total_len = len1 + len2; /* Could overflow */

/* Safer */
if (len1 > SIZE_MAX - len2) {
    return (EINVAL); /* Overflow would occur */
}
size_t total_len = len1 + len2;
```

5.7.11 Summary

Kernel string handling requires constant vigilance:

- Use safe functions that respect buffer boundaries
- Always validate lengths and check for truncation
- Handle user data with `copyin()/copyout()`
- Prefer explicit-length operations over null-terminated functions when working with binary data
- Initialize buffers and check for allocation failures

The defensive programming mindset extends to every string operation in the kernel. In the next section, we'll explore how this mindset applies to function design and error handling.

5.8 Functions and Return Conventions

Function design in the kernel follows patterns that might seem strange if you're coming from user-space programming. These patterns aren't arbitrary; they reflect decades of experience with the constraints and requirements of system-level code. Understanding these conventions will help you write functions that match the kernel's own patterns and meet the expectations of other kernel developers.

5.8.1 The Kernel's Function Signature Patterns

Here is a typical kernel function signature and body. The following pseudo-example shows the KNF layout you'll encounter everywhere in `/usr/src/sys/kern/`:

```
int
my_acquire(struct my_object *obj, int flags)
{
    int error;

    MPASS((flags & MY_FLAG_MASK) != 0);

    error = my_lock(obj, flags);
    if (error != 0)
        return (error);

    my_ref(obj);
    error = my_lock_upgrade(obj, flags | MY_FLAG_INTERLOCK);
    if (error != 0) {
        my_unref(obj);
        return (error);
    }

    return (0);
}
```

Real examples of this shape include `vget()` in `/usr/src/sys/kern/vfs_subr.c` and countless other subsystem functions. Notice several important patterns:

Return type comes first: The `int` return type is on its own line, making functions easy to scan.

Error codes are integers: Functions return `0` for success, positive integers for errors.

Multiple exit points are acceptable: Unlike some user-space style guides, kernel functions often have multiple return statements for early error exits.

Resource cleanup on failure: When the function fails, it cleans up any resources it allocated before returning the error code.

5.8.2 Error Return Conventions

FreeBSD kernel functions follow a strict convention for indicating success and failure:

- **Return 0 for success**
- **Return positive `errno` codes for failure** (like `ENOMEM`, `EINVAL`, `ENODEV`)
- **Never return negative values** (unlike Linux kernel)

An illustrative pattern for a function that validates its inputs, takes a lock, does work, and then releases the lock through a single exit label:

```
int
my_lookup(struct my_table *tbl, int key, struct my_entry **outp)
{
    struct my_entry *e;
    int error = 0;

    if (tbl == NULL || outp == NULL)
        return (EINVAL);
    if (key < 0)
        return (EINVAL);

    *outp = NULL;

    MY_TABLE_LOCK(tbl);
    e = my_table_find(tbl, key);
    if (e == NULL) {
        error = ENOENT;
        goto out;
    }
    my_entry_ref(e);
    *outp = e;

out:
    MY_TABLE_UNLOCK(tbl);
    return (error);
}
```

Real examples of this shape are easy to find in `/usr/src/sys/kern/kern_descrip.c` (for instance, `kern_dup()`, which duplicates file descriptors), `/usr/src/sys/kern/vfs_lookup.c`, and most other subsystems.

5.8.3 Parameter Patterns and Conventions

Kernel functions follow predictable patterns for parameter ordering and naming:

Context parameters first: Thread context (`struct thread *td`) or process context usually comes first.

Input parameters before output parameters: Read the parameters left-to-right like a sentence.

Flags and options last: Configuration parameters typically come at the end.

A simplified sketch of the decision `malloc(9)` makes internally:


```
void *
my_allocator(size_t size, struct malloc_type *mtp, int flags)
{
    void *va;

    if (size > ZONE_MAX_SIZE) {
        /* Large allocation: bypass the zone cache. */
        va = large_alloc(size, flags);
    } else {
        /* Small allocation: pick a size-bucketed zone. */
        va = zone_alloc(size_to_zone(size), mtp, flags);
    }
    return (va);
}
```

The real `malloc(9)` in `/usr/src/sys/kern/kern_malloc.c` is quite a bit more intricate than this, but the conceptual split, small allocations flow through a set of size-bucketed UMA zones, and large allocations bypass them entirely, is exactly what it does.

5.8.4 Output Parameters and Return Values

The kernel uses several patterns for returning data to the caller:

Simple success/failure: Return error code, no additional data.

Single output value: Use the function's return value directly.

Multiple outputs: Use pointer parameters to “return” additional values.

Complex outputs: Use a structure to package multiple return values.

A simplified version of how `/usr/src/sys/kern/kern_time.c` dispatches on a clock identifier, with the result delivered through an output parameter:

```
int
my_get_time(clockid_t clock_id, struct timespec *ats)
{
    int error = 0;

    switch (clock_id) {
    case CLOCK_REALTIME:
    case CLOCK_REALTIME_PRECISE:
        nanotime(ats);
        break;
    case CLOCK_REALTIME_FAST:
        getnanotime(ats);
        break;
    }
```

```
case CLOCK_MONOTONIC:
case CLOCK_MONOTONIC_PRECISE:
case CLOCK_UPTIME:
case CLOCK_UPTIME_PRECISE:
    nanouptime(ats);
    break;
default:
    error = EINVAL;
    break;
}
return (error);
}
```

The function returns an `int` error code and writes the actual value into the caller-supplied `ats` pointer. The real `kern_clock_gettime()` adds a few more clock IDs (for example `CLOCK_VIRTUAL` and `CLOCK_PROF`) and acquires process locks when needed, but the output-parameter pattern is the same.

5.8.5 Function Naming Conventions

FreeBSD follows consistent naming patterns that make code self-documenting:

Subsystem prefixes: Functions start with their subsystem name (`vn_` for vnode operations, `vm_` for virtual memory, etc.).

Action verbs: Function names clearly indicate what they do (`alloc`, `free`, `lock`, `unlock`, `create`, `destroy`).

Consistency within subsystems: Related functions follow parallel naming (`uma_zalloc` / `uma_zfree`).

From `sys/vm/vm_page.c`:

```
vm_page_t vm_page_alloc(vm_object_t object, vm_pindex_t pindex, int req);
void vm_page_free(vm_page_t m);
void vm_page_free_zero(vm_page_t m);
void vm_page_lock(vm_page_t m);
void vm_page_unlock(vm_page_t m);
```

5.8.6 Static Functions vs. External Functions

The kernel makes extensive use of static functions for internal implementation details:

```
/* Internal helper - not visible outside this file */
static int
validate_mount_options(struct mount *mp, const char *opts)
{
    /* Implementation details... */
```

```

    return (0);
}

/* External interface - visible to other kernel modules */
int
vfs_mount(struct thread *td, const char *fstype, char *fspath,
          int fsflags, void *data)
{
    int error;

    error = validate_mount_options(mp, fspath);
    if (error)
        return (error);

    /* Continue with mount... */
    return (0);
}

```

This separation keeps the external API clean while allowing complex internal implementation.

5.8.7 Inline Functions vs. Macros

For small, performance-critical operations, the kernel uses both inline functions and macros. Inline functions are generally preferred because they provide type checking:

From sys/sys/system.h:

```

/* Inline function - type safe */
static __inline int
imax(int a, int b)
{
    return (a > b ? a : b);
}

/* Macro - faster but less safe */
#define MAX(a, b) ((a) > (b) ? (a) : (b))

```

5.8.8 Function Documentation and Commenting

Well-written kernel functions include clear documentation:

```

/*
 * vnode_pager_alloc - allocate a vnode pager object

```

```

*
* This function creates a vnode-backed VM object for memory-mapped files.
* The object allows the VM system to page file contents in and out of
* physical memory on demand.
*
* Arguments:
*   vp      - vnode to create pager for
*   size    - size of the mapping in bytes
*   prot    - protection flags (read/write/execute)
*   offset  - offset within the file
*
* Returns:
*   Pointer to vm_object on success, NULL on failure
*
* Locking:
*   The vnode must be locked on entry and remains locked on exit.
*/
vm_object_t
vnode_pager_alloc(struct vnode *vp, vm_ooffset_t size, vm_prot_t prot,
    vm_ooffset_t offset)
{
    /* Implementation... */
}

```

5.8.9 Hands-On Lab: Function Design Patterns

Let's create a kernel module that demonstrates proper function design:

```

/*
 * function_demo.c - Demonstrate kernel function conventions
 */
#include <sys/param.h>
#include <sys/kernel.h>
#include <sys/module.h>
#include <sys/systm.h>
#include <sys/malloc.h>

MALLOC_DEFINE(M_FUNCDEMO, "funcdemo", "Function demo allocations");

/*
 * Internal helper function - validate buffer parameters
 * Returns 0 on success, errno on failure
 */
static int

```

```
validate_buffer_params(size_t size, int flags)
{
    if (size == 0) {
        return (EINVAL); /* Invalid size */
    }

    if (size > 1024 * 1024) {
        return (EFBIG); /* Buffer too large */
    }

    if ((flags & ~(M_WAITOK | M_NOWAIT | M_ZERO)) != 0) {
        return (EINVAL); /* Invalid flags */
    }

    return (0); /* Success */
}

/*
 * Allocate and initialize a demo buffer
 * Returns 0 on success with buffer pointer in *bufp
 */
static int
demo_buffer_alloc(char **bufp, size_t size, int flags)
{
    char *buffer;
    int error;

    /* Validate parameters */
    if (bufp == NULL) {
        return (EINVAL);
    }
    *bufp = NULL; /* Initialize output parameter */

    error = validate_buffer_params(size, flags);
    if (error != 0) {
        return (error);
    }

    /* Allocate the buffer */
    buffer = malloc(size, M_FUNCDEMO, flags);
    if (buffer == NULL) {
        return (ENOMEM);
    }
}
```

```
/* Initialize buffer contents */
snprintf(buffer, size, "Demo buffer of %zu bytes", size);

*bufp = buffer; /* Return buffer to caller */
return (0);      /* Success */
}

/*
 * Free a demo buffer allocated by demo_buffer_alloc
 */
static void
demo_buffer_free(char *buffer)
{
    if (buffer != NULL) {
        free(buffer, M_FUNCDEMO);
    }
}

/*
 * Process a demo buffer - returns number of bytes processed
 * Returns negative value on error
 */
static ssize_t
demo_buffer_process(const char *buffer, size_t size, bool verbose)
{
    size_t len;

    if (buffer == NULL || size == 0) {
        return (-EINVAL);
    }

    len = strlen(buffer, size);
    if (verbose) {
        printf("Processing buffer: '%.*s' (length %zu)\n",
            (int)len, buffer, len);
    }

    return ((ssize_t)len);
}

static int
function_demo_load(module_t mod, int cmd, void *arg)
{
    char *buffer;
```

```
ssize_t processed;
int error;

switch (cmd) {
case MOD_LOAD:
    printf("=== Function Design Demo ===\n");

    /* Demonstrate successful allocation */
    error = demo_buffer_alloc(&buffer, 256, M_WAITOK | M_ZERO);
    if (error != 0) {
        printf("Buffer allocation failed: %d\n", error);
        return (error);
    }

    printf("Allocated buffer: %p\n", buffer);

    /* Process the buffer */
    processed = demo_buffer_process(buffer, 256, true);
    if (processed < 0) {
        printf("Buffer processing failed: %zd\n", processed);
    } else {
        printf("Processed %zd bytes\n", processed);
    }

    /* Clean up */
    demo_buffer_free(buffer);

    /* Demonstrate parameter validation */
    error = demo_buffer_alloc(&buffer, 0, M_WAITOK);
    if (error != 0) {
        printf("Parameter validation works: error %d\n", error);
    }

    printf("Function demo completed successfully.\n");
    break;

case MOD_UNLOAD:
    printf("Function demo module unloaded.\n");
    break;

default:
    return (EOPNOTSUPP);
}
```

```
    return (0);
}

static moduledata_t function_demo_mod = {
    "function_demo",
    function_demo_load,
    NULL
};

DECLARE_MODULE(function_demo, function_demo_mod, SI_SUB_DRIVERS, SI_ORDER_MIDDLE);
MODULE_VERSION(function_demo, 1);
```

5.8.10 Function Design Best Practices

Validate all parameters: Check for NULL pointers, invalid sizes, and bad flags at the beginning of your function.

Use clear return conventions: Return 0 for success, errno codes for specific failures.

Initialize output parameters early: Set pointer outputs to NULL or structure outputs to zero before doing work.

Clean up on failure: If your function allocates resources and then fails, free those resources before returning.

Use static for internal functions: Keep implementation details hidden and the external API clean.

Document complex functions: Explain what the function does, what its parameters mean, what it returns, and any locking requirements.

5.8.11 Summary

Kernel function design is about predictability and safety:

- Follow consistent naming and parameter ordering conventions
- Use the standard error return pattern (0 for success)
- Validate parameters and handle all error conditions
- Clean up resources on failure paths
- Keep internal implementation details static
- Document the public interface clearly

These conventions make your code easier to understand, debug, and maintain. They also make it fit naturally into the larger FreeBSD codebase.

In the next section, we'll explore the restrictions that make kernel C different from user-space C constraints that shape how you write functions and structure your code.

5.9 Restrictions and Pitfalls of Kernel C

The kernel operates under constraints that simply don't exist in user-space programming. These aren't arbitrary limitations; they're the necessary boundaries that allow a kernel to manage system resources safely and efficiently while running the entire machine. Understanding these restrictions matters because violating them doesn't just cause your program to crash; it can bring down the entire system.

5.9.1 The Floating-Point Restriction

One of the most fundamental restrictions is that **kernel code cannot use floating-point operations** without special handling. This includes `float`, `double`, and any math library functions that use them.

Here's why this restriction exists:

FPU state belongs to user processes: The floating-point unit (FPU) maintains state (registers, flags) that belongs to whatever user process was last running. If kernel code modifies FPU state, it corrupts the user process's calculations.

Context switching overhead: To use floating-point safely, the kernel would need to save and restore FPU state on every kernel entry/exit, adding significant overhead to system calls and interrupts.

Interrupt handler complexity: Interrupt handlers can't predict when they'll run or what FPU state is currently loaded.

```
/* WRONG - will not compile or crash the system */
float
calculate_average(int *values, int count)
{
    float sum = 0.0; /* Error: floating-point in kernel */
    int i;

    for (i = 0; i < count; i++) {
        sum += values[i];
    }

    return sum / count; /* Error: floating-point division */
}

/* RIGHT - use integer arithmetic */
int
calculate_average_scaled(int *values, int count, int scale)
{
    long sum = 0;
    int i;

    if (count == 0)
        return (0);
```

```
for (i = 0; i < count; i++) {
    sum += values[i];
}

return ((int)((sum * scale) / count));
}
```

In practice, kernel algorithms use **fixed-point arithmetic** or **scaled integers** when they need fractional precision.

5.9.2 Stack Size Limitations

User-space programs typically have stack sizes measured in megabytes. Kernel stacks are much smaller: **16KB per thread** on FreeBSD 14.3 (four pages on amd64 and arm64), which includes space for interrupt handling.

```
/* DANGEROUS - can overflow kernel stack */
void
bad_recursive_function(int depth)
{
    char local_buffer[1024]; /* 1KB per recursion level */

    if (depth > 0) {
        /* This can quickly exhaust the kernel stack */
        bad_recursive_function(depth - 1);
    }
}

/* BETTER - limit stack usage and recursion */
int
good_iterative_function(int max_iterations)
{
    char *work_buffer;
    int i, error = 0;

    /* Allocate large buffers on the heap, not stack */
    work_buffer = malloc(1024, M_TEMP, M_WAITOK);
    if (work_buffer == NULL) {
        return (ENOMEM);
    }

    for (i = 0; i < max_iterations; i++) {
        /* Do work without deep recursion */
    }
}
```

```
    free(work_buffer, M_TEMP);  
    return (error);  
}
```

An illustrative pattern for careful stack management in a long-running path lookup:

```
int  
my_resolve(struct my_request *req)  
{  
    struct my_context *ctx;  
    char *work_buffer;          /* large buffer allocated dynamically */  
    int error;  
  
    if (req->path_len > MY_MAXPATHLEN)  
        return (ENAMETOOLONG);  
  
    work_buffer = malloc(MY_MAXPATHLEN, M_TEMP, M_WAITOK);  
  
    /* ... perform lookup using work_buffer ... */  
  
    free(work_buffer, M_TEMP);  
    return (error);  
}
```

You can read `namei()` in `/usr/src/sys/kern/vfs_lookup.c` for the real implementation that FreeBSD uses for every path resolution. The function itself is complex, but it keeps its stack footprint small by allocating the working buffer through `namei_zone` (a dedicated UMA zone) instead of declaring it on the stack.

5.9.3 Sleep Restrictions: Atomic vs. Preemptible Context

Understanding when your code can and cannot **sleep** (voluntarily give up the CPU) is critical for kernel programming.

Atomic context (cannot sleep):

- Interrupt handlers
- Code holding spinlocks
- Code in critical sections
- Some callback functions

Preemptible context (can sleep):

- System call handlers
- Kernel threads

- Most driver probe/attach functions

```
/* WRONG - sleeping in interrupt context */
void
my_interrupt_handler(void *arg)
{
    char *buffer;

    /* This will panic the system! */
    buffer = malloc(1024, M_DEVBUF, M_WAITOK);

    /* Process interrupt... */

    free(buffer, M_DEVBUF);
}

/* RIGHT - using non-sleeping allocation */
void
my_interrupt_handler(void *arg)
{
    char *buffer;

    buffer = malloc(1024, M_DEVBUF, M_NOWAIT);
    if (buffer == NULL) {
        /* Handle allocation failure gracefully */
        device_schedule_deferred_work(arg);
        return;
    }

    /* Process interrupt... */

    free(buffer, M_DEVBUF);
}
```

An illustrative sketch of the same pattern you'll see inside real drivers like `/usr/src/sys/dev/e1000/if_em.c`:

```
static void
my_intr(void *arg)
{
    struct my_softc *sc = arg;

    /* Interrupt context: fast, no sleeping allowed. */
    sc->intr_count++;
}
```

```
/*
 * Hand the heavy lifting over to a taskqueue so it runs in a
 * context where we're allowed to sleep (for instance, in case
 * it needs to allocate memory with M_WAITOK).
 */
taskqueue_enqueue(sc->tq, &sc->rx_task);
}
```

The interrupt handler does minimal work and schedules a task queue to handle the bulk of processing in a context where sleeping is allowed.

5.9.4 Recursion Limitations

Deep recursion is dangerous in the kernel due to limited stack space. Many kernel algorithms that might naturally use recursion in user space are rewritten iteratively:

```
/* Traditional recursive tree traversal - dangerous in kernel */
void
traverse_tree_recursive(struct tree_node *node, void (*func)(void *))
{
    if (node == NULL)
        return;

    func(node->data);
    traverse_tree_recursive(node->left, func); /* Stack grows */
    traverse_tree_recursive(node->right, func); /* Stack grows more */
}

/* Kernel-safe iterative version using explicit stack */
int
traverse_tree_iterative(struct tree_node *root, void (*func)(void *))
{
    struct tree_node **stack;
    struct tree_node *node;
    int stack_size = 100; /* Reasonable limit */
    int sp = 0; /* Stack pointer */
    int error = 0;

    if (root == NULL)
        return (0);

    stack = malloc(stack_size * sizeof(*stack), M_TEMP, M_WAITOK);
    if (stack == NULL)
        return (ENOMEM);
```

```

    stack[sp++] = root;

    while (sp > 0) {
        node = stack[--sp];
        func(node->data);

        /* Add children to stack (right first, then left) */
        if (node->right && sp < stack_size - 1)
            stack[sp++] = node->right;
        if (node->left && sp < stack_size - 1)
            stack[sp++] = node->left;

        if (sp >= stack_size - 1) {
            error = ENOMEM; /* Stack exhausted */
            break;
        }
    }

    free(stack, M_TEMP);
    return (error);
}

```

5.9.5 Global Variables and Thread Safety

Global variables in the kernel are shared among all threads and processes. Accessing them safely requires proper synchronization:

```

/* WRONG - race condition */
static int global_counter = 0;

void
increment_counter(void)
{
    global_counter++; /* Not atomic - can corrupt data */
}

/* RIGHT - using atomic operations */
static volatile u_int global_counter = 0;

void
increment_counter_safely(void)
{
    atomic_add_int(&global_counter, 1);
}

```

```
}

/* ALSO RIGHT - using locks for more complex operations */
static int global_counter = 0;
static struct mtx counter_lock;

void
increment_counter_with_lock(void)
{
    mtx_lock(&counter_lock);
    global_counter++;
    mtx_unlock(&counter_lock);
}
```

5.9.6 Memory Allocation Context Awareness

The flags you pass to `malloc()` must match your execution context:

```
/* Context-aware allocation wrapper */
void *
safe_malloc(size_t size, struct malloc_type *type)
{
    int flags;

    /* Choose flags based on current context */
    if (cold) {
        /* During early boot - very limited options */
        flags = M_NOWAIT;
    } else if (curthread->td_critnest != 0) {
        /* In critical section - cannot sleep */
        flags = M_NOWAIT;
    } else if (SCHEDULER_STOPPED()) {
        /* Scheduler is stopped (panic, debugger) */
        flags = M_NOWAIT;
    } else {
        /* Normal context - can sleep */
        flags = M_WAITOK;
    }

    return (malloc(size, type, flags));
}
```

5.9.7 Performance Considerations

Kernel code runs in a performance-critical environment where every CPU cycle matters:

Avoid expensive operations in hot paths:

```
/* SLOW - division is expensive */
int average = (total / count);

/* FASTER - bit shifting for powers of 2 */
int average = (total >> log2_count); /* If count is power of 2 */

/* COMPROMISE - cache the division result if used repeatedly */
static int cached_divisor = 0;
static int cached_result = 0;

if (divisor != cached_divisor) {
    cached_divisor = divisor;
    cached_result = SCALE_FACTOR / divisor;
}
int scaled_result = (total * cached_result) >> SCALE_SHIFT;
```

5.9.8 Hands-On Lab: Understanding Restrictions

Let's create a kernel module that demonstrates these restrictions safely:

```
/*
 * restrictions_demo.c - Demonstrate kernel programming restrictions
 */
#include <sys/param.h>
#include <sys/kernel.h>
#include <sys/module.h>
#include <sys/system.h>
#include <sys/malloc.h>
#include <machine/atomic.h>

MALLOC_DEFINE(M_RESTRICT, "restrict", "Restriction demo");

static volatile u_int atomic_counter = 0;
static struct mtx demo_lock;

/* Safe recursive function with depth limit */
static int
safe_recursive_demo(int depth, int max_depth)
{
```



```
int result = 0;

if (depth >= max_depth) {
    return (depth); /* Base case - avoid deep recursion */
}

/* Use minimal stack space */
result = safe_recursive_demo(depth + 1, max_depth);
return (result + 1);
}

/* Fixed-point arithmetic instead of floating-point */
static int
fixed_point_average(int *values, int count, int scale)
{
    long sum = 0;
    int i;

    if (count == 0)
        return (0);

    for (i = 0; i < count; i++) {
        sum += values[i];
    }

    /* Return average scaled by 'scale' factor */
    return ((int)((sum * scale) / count));
}

static int
restrictions_demo_load(module_t mod, int cmd, void *arg)
{
    int values[] = {10, 20, 30, 40, 50};
    int avg_scaled, recursive_result;
    u_int counter_val;

    switch (cmd) {
    case MOD_LOAD:
        printf("=== Kernel Restrictions Demo ===\n");

        mtx_init(&demo_lock, "demo_lock", NULL, MTX_DEF);

        /* Demonstrate fixed-point arithmetic */
        avg_scaled = fixed_point_average(values, 5, 100);
```

```

    printf("Average * 100 = %d (actual average would be %.%02d)\n",
           avg_scaled, avg_scaled / 100, avg_scaled % 100);

    /* Demonstrate safe recursion with limits */
    recursive_result = safe_recursive_demo(0, 10);
    printf("Safe recursive function result: %d\n", recursive_result);

    /* Demonstrate atomic operations */
    atomic_add_int(&atomic_counter, 42);
    counter_val = atomic_load_acq_int(&atomic_counter);
    printf("Atomic counter value: %u\n", counter_val);

    /* Demonstrate context-aware allocation */
    void *buffer = malloc(1024, M_RESTRICT, M_WAITOK);
    if (buffer) {
        printf("Successfully allocated buffer in safe context\n");
        free(buffer, M_RESTRICT);
    }

    printf("Restrictions demo completed successfully.\n");
    break;

case MOD_UNLOAD:
    mtx_destroy(&demo_lock);
    printf("Restrictions demo module unloaded.\n");
    break;

default:
    return (EOPNOTSUPP);
}

return (0);
}

static moduledata_t restrictions_demo_mod = {
    "restrictions_demo",
    restrictions_demo_load,
    NULL
};

DECLARE_MODULE(restrictions_demo, restrictions_demo_mod, SI_SUB_DRIVERS,
    ↵ SI_ORDER_MIDDLE);
MODULE_VERSION(restrictions_demo, 1);

```

5.9.9 Summary

Kernel programming restrictions exist for good reasons:

- No floating-point prevents corruption of user process state
- Limited stack size forces efficient algorithms and prevents overflow
- Sleep restrictions ensure system responsiveness and prevent deadlocks
- Recursion limits prevent stack exhaustion
- Atomic operations prevent race conditions in shared data

Understanding these constraints helps you write kernel code that is not just functional, but robust and performant. These restrictions shape the idioms and patterns we'll explore in the next section.

5.9.10 Atomic Operations and Inline Functions

Modern multiprocessor systems require special techniques to ensure that operations on shared data happen atomically, that is, completely and indivisibly from the perspective of other CPUs. FreeBSD provides a comprehensive set of atomic operations and makes extensive use of inline functions to ensure both correctness and performance in kernel code.

5.9.11 Why Atomic Operations Matter

Consider this seemingly simple operation:

```
static int global_counter = 0;

void increment_counter(void)
{
    global_counter++; /* Looks atomic, but isn't! */
}
```

On a multiprocessor system, `global_counter++` actually involves multiple steps:

1. Load the current value from memory
2. Increment the value in a register
3. Store the new value back to memory

If two CPUs execute this code simultaneously, you can get race conditions where both CPUs read the same initial value, increment it, and store the same result, effectively losing one of the increments.

You will see this pattern, “increment a shared counter atomically”, in many places across the kernel:

```
static volatile u_int active_consumers = 0;

static void
my_consumer_add(void)
```

```
{

    atomic_add_int(&active_consumers, 1);
}

static void
my_consumer_remove(void)
{

    atomic_subtract_int(&active_consumers, 1);
}
```

Using `atomic_add_int()` and `atomic_subtract_int()` instead of the C ++ and -- operators ensures that concurrent increments from different CPUs don't lose updates.

5.9.12 FreeBSD's Atomic Operations

FreeBSD provides atomic operations in `<machine/atomic.h>`. These operations are implemented using CPU-specific instructions that guarantee atomicity:

```
#include <machine/atomic.h>

/* Atomic arithmetic */
void atomic_add_int(volatile u_int *p, u_int val);
void atomic_subtract_int(volatile u_int *p, u_int val);

/* Atomic bit operations */
void atomic_set_int(volatile u_int *p, u_int mask);
void atomic_clear_int(volatile u_int *p, u_int mask);

/* Atomic compare and swap */
int atomic_cmpset_int(volatile u_int *dst, u_int expect, u_int src);

/* Atomic load and store with memory barriers */
u_int atomic_load_acq_int(volatile u_int *p);
void atomic_store_rel_int(volatile u_int *p, u_int val);
```

Here's how the counter example should be written:

```
static volatile u_int global_counter = 0;

void increment_counter_safely(void)
{
```

```
    atomic_add_int(&global_counter, 1);
}

u_int read_counter_safely(void)
{
    return (atomic_load_acq_int(&global_counter));
}
```

5.9.13 Memory Barriers and Ordering

Modern CPUs can reorder memory operations for performance. Sometimes you need to ensure that certain operations happen in a specific order. This is where **memory barriers** come in:

```
/* Write barrier - ensure all previous writes complete first */
atomic_store_rel_int(&status_flag, READY);

/* Read barrier - ensure this read happens before subsequent operations */
int status = atomic_load_acq_int(&status_flag);
```

The `_acq` (acquire) and `_rel` (release) suffixes indicate memory ordering: - **Acquire**: Operations after this one cannot be reordered before it - **Release**: Operations before this one cannot be reordered after it

An illustrative sketch of the acquire/release pattern at the heart of most lock primitives:

```
struct my_flag {
    volatile u_int value;
};

void
my_flag_set_ready(struct my_flag *f)
{
    /* "Release": all earlier writes are visible to any CPU that
    * later observes value == READY. */
    atomic_store_rel_int(&f->value, READY);
}

bool
my_flag_is_ready(struct my_flag *f)
{
    /* "Acquire": once we've read READY, subsequent reads see all
    * the writes that happened before the paired release. */
```

```

    return (atomic_load_acq_int(&f->value) == READY);
}

```

Real lock primitives in `/usr/src/sys/kern/kern_rwlock.c` and friends use exactly this `acq/rel` pairing around their internal state, which is how they guarantee that data protected by the lock becomes visible in the correct order.

5.9.14 Compare-and-Swap: The Building Block

Many lock-free algorithms are built on **compare-and-swap (CAS)** operations:

```

/*
 * Atomically compare the value at *dst with 'expect'.
 * If they match, store 'src' at *dst and return 1.
 * If they don't match, return 0.
 */
int result = atomic_cmpset_int(dst, expect, src);

```

Here's a lock-free stack implementation using CAS:

```

struct lock_free_stack {
    volatile struct stack_node *head;
};

struct stack_node {
    struct stack_node *next;
    void *data;
};

int
lockfree_push(struct lock_free_stack *stack, struct stack_node *node)
{
    struct stack_node *old_head;

    do {
        old_head = stack->head;
        node->next = old_head;

        /* Try to atomically update head pointer */
    } while (!atomic_cmpset_ptr((volatile uintptr_t *)&stack->head,
                               (uintptr_t)old_head, (uintptr_t)node));

    return (0);
}

```

5.9.15 Inline Functions for Performance

Inline functions are important in kernel programming because they provide the type safety of functions with the performance of macros. FreeBSD makes extensive use of static `__inline` functions:

```
/* From sys/sys/sysrm.h */
static __inline int
imax(int a, int b)
{
    return (a > b ? a : b);
}

static __inline int
imin(int a, int b)
{
    return (a < b ? a : b);
}

/* From sys/sys/libkern.h */
static __inline int
ffs(int mask)
{
    return (__builtin_ffs(mask));
}
```

Here's a more complex example from `sys/vm/vm_page.h`:

```
/*
 * Inline function to check if a VM page is wired
 * (pinned in physical memory)
 */
static __inline boolean_t
vm_page_wired(vm_page_t m)
{
    return ((m->wire_count != 0));
}

/*
 * Inline function to safely reference a VM page
 */
static __inline void
vm_page_wire(vm_page_t m)
{
    atomic_add_int(&m->wire_count, 1);
    if (m->wire_count == 1) {
```

```
    vm_cnt.v_wire_count++;
    if (m->object != NULL && (m->object->flags & OBJ_UNMANAGED) == 0)
        atomic_subtract_int(&vm_cnt.v_free_count, 1);
}
```

5.9.16 When to Use Inline Functions

Use inline for:

- Small, frequently called functions (< 10 lines typically)
- Functions in critical performance paths
- Simple accessor functions
- Functions that wrap complex macros to add type safety

Don't inline:

- Large functions (increases code size)
- Functions with complex control flow
- Functions that are rarely called
- Functions that take their address (can't be inlined)

5.9.17 Combining Atomics and Inlines

Many kernel subsystems combine atomic operations with inline functions for both performance and safety:

```
/* Reference counting with atomic operations */
static __inline void
obj_ref(struct my_object *obj)
{
    u_int old __diagused;

    old = atomic_fetchadd_int(&obj->refcount, 1);
    KASSERT(old > 0, ("obj_ref: object %p has zero refcount", obj));
}

static __inline int
obj_unref(struct my_object *obj)
{
    u_int old;

    old = atomic_fetchadd_int(&obj->refcount, -1);
    KASSERT(old > 0, ("obj_unref: object %p has zero refcount", obj));
}
```



```
    return (old == 1); /* Return true if this was the last reference */
}
```

5.9.18 Hands-On Lab: Atomic Operations and Performance

Let's create a kernel module that demonstrates atomic operations:

```
/*
 * atomic_demo.c - Demonstrate atomic operations and inline functions
 */
#include <sys/param.h>
#include <sys/kernel.h>
#include <sys/module.h>
#include <sys/systm.h>
#include <machine/atomic.h>

static volatile u_int shared_counter = 0;
static volatile u_int shared_flags = 0;

/* Inline function for safe counter increment */
static __inline void
safe_increment(volatile u_int *counter)
{
    atomic_add_int(counter, 1);
}

/* Inline function for safe flag manipulation */
static __inline void
set_flag_atomically(volatile u_int *flags, u_int flag)
{
    atomic_set_int(flags, flag);
}

static __inline void
clear_flag_atomically(volatile u_int *flags, u_int flag)
{
    atomic_clear_int(flags, flag);
}

static __inline boolean_t
test_flag_atomically(volatile u_int *flags, u_int flag)
{
    return ((atomic_load_acq_int(flags) & flag) != 0);
}
```

```
/* Compare-and-swap example */
static int
atomic_max_update(volatile u_int *current_max, u_int new_value)
{
    u_int old_value;

    do {
        old_value = *current_max;
        if (new_value <= old_value) {
            return (0); /* No update needed */
        }

        /* Try to atomically update if still the same value */
    } while (!atomic_cmpset_int(current_max, old_value, new_value));

    return (1); /* Successfully updated */
}

static int
atomic_demo_load(module_t mod, int cmd, void *arg)
{
    u_int counter_val, flags_val;
    int i, updated;

    switch (cmd) {
    case MOD_LOAD:
        printf("=== Atomic Operations Demo ===\n");

        /* Initialize shared state */
        atomic_store_rel_int(&shared_counter, 0);
        atomic_store_rel_int(&shared_flags, 0);

        /* Demonstrate atomic arithmetic */
        for (i = 0; i < 10; i++) {
            safe_increment(&shared_counter);
        }
        counter_val = atomic_load_acq_int(&shared_counter);
        printf("Counter after 10 increments: %u\n", counter_val);

        /* Demonstrate atomic bit operations */
        set_flag_atomically(&shared_flags, 0x01);
        set_flag_atomically(&shared_flags, 0x04);
        set_flag_atomically(&shared_flags, 0x10);
    }
```

```

    flags_val = atomic_load_acq_int(&shared_flags);
    printf("Flags after setting bits 0, 2, 4: 0x%02x\n", flags_val);

    printf("Flag 0x01 is %s\n",
           test_flag_atomically(&shared_flags, 0x01) ? "set" : "clear");
    printf("Flag 0x02 is %s\n",
           test_flag_atomically(&shared_flags, 0x02) ? "set" : "clear");

    clear_flag_atomically(&shared_flags, 0x01);
    printf("Flag 0x01 after clear is %s\n",
           test_flag_atomically(&shared_flags, 0x01) ? "set" : "clear");

    /* Demonstrate compare-and-swap */
    updated = atomic_max_update(&shared_counter, 5);
    printf("Attempt to update max to 5: %s\n", updated ? "success" : "failed");

    updated = atomic_max_update(&shared_counter, 15);
    printf("Attempt to update max to 15: %s\n", updated ? "success" : "failed");

    counter_val = atomic_load_acq_int(&shared_counter);
    printf("Final counter value: %u\n", counter_val);

    printf("Atomic operations demo completed successfully.\n");
    break;

case MOD_UNLOAD:
    printf("Atomic demo module unloaded.\n");
    break;

default:
    return (EOPNOTSUPP);
}

return (0);
}

static moduledata_t atomic_demo_mod = {
    "atomic_demo",
    atomic_demo_load,
    NULL
};

DECLARE_MODULE(atomic_demo, atomic_demo_mod, SI_SUB_DRIVERS, SI_ORDER_MIDDLE);

```

```
MODULE_VERSION(atomic_demo, 1);
```

5.9.19 Performance Considerations

Atomic operations have cost: While atomic operations ensure correctness, they're slower than regular memory operations. Use them only when necessary.

Memory barriers affect performance: Acquire/release semantics can prevent CPU optimizations. Use the weakest ordering that provides correctness.

Lock-free isn't always faster: For complex operations, traditional locking might be simpler and faster than lock-free algorithms.

5.9.20 Summary

Atomic operations and inline functions are essential tools for high-performance, correct kernel programming:

- Atomic operations ensure data consistency in multiprocessor systems
- Memory barriers control operation ordering when needed
- Compare-and-swap enables sophisticated lock-free algorithms
- Inline functions provide performance without sacrificing type safety
- Use these tools judiciously, correctness first, then optimize

These low-level primitives form the foundation for the higher-level synchronization and coding patterns we'll explore in the next section.

5.10 Coding Idioms and Style in Kernel Development

Every mature software project develops its own culture, including patterns of expression, conventions, and idioms, that make code readable and maintainable by the community. FreeBSD's kernel has evolved over decades, creating a rich set of coding idioms that reflect both practical experience and the system's architectural philosophy. Learning these patterns will help you write code that looks and feels like it belongs in the FreeBSD kernel.

5.10.1 FreeBSD Kernel Normal Form (KNF)

FreeBSD follows a coding style called **Kernel Normal Form (KNF)**, documented in `style(9)`. While this might seem like nitpicking, consistent style makes code reviews easier, reduces merge conflicts, and helps new developers understand existing code.

Key elements of KNF:

Indentation: Use tabs, not spaces. Each indentation level is one tab.

Braces: Opening brace goes on the same line for control structures, on a new line for functions.

```
/* Control structures - brace on same line */
if (condition) {
    statement;
} else {
    other_statement;
}

/* Function definitions - brace on new line */
int
my_function(int parameter)
{
    return (parameter + 1);
}
```

Line length: Keep lines under 80 characters when practical.

Variable declarations: Declare variables at the beginning of blocks, with a blank line separating declarations from code.

Here is an illustrative function in KNF:

```
static int
my_read_chunk(struct thread *td, struct my_source *src, off_t offset,
    void *buf, size_t len)
{
    struct iovec iov;
    struct uio uio;
    int error;

    if (len == 0)
        return (0);

    iov.iov_base = buf;
    iov.iov_len = len;
    uio.uio_iov = &iov;
    uio.uio_iovcnt = 1;
    uio.uio_offset = offset;
    uio.uio_resid = len;
    uio.uio_segflg = UIO_SYSSPACE;
    uio.uio_rw = UIO_READ;
    uio.uio_td = td;

    error = my_read_via_uio(src, &uio);
    return (error);
}
```

5.10.2 Error Handling Patterns

FreeBSD kernel code follows consistent patterns for error handling that make code predictable and reliable.

Early validation: Check parameters at the beginning of functions.

Single exit point pattern: Use goto for cleanup in complex functions.

```
int
complex_operation(struct device *dev, void *buffer, size_t size)
{
    void *temp_buffer = NULL;
    struct resource *res = NULL;
    int error = 0;

    /* Early validation */
    if (dev == NULL || buffer == NULL || size == 0)
        return (EINVAL);

    if (size > MAX_TRANSFER_SIZE)
        return (EFBIG);

    /* Allocate resources */
    temp_buffer = malloc(size, M_DEVBUF, M_WAITOK);
    if (temp_buffer == NULL) {
        error = ENOMEM;
        goto cleanup;
    }

    res = bus_alloc_resource_any(dev, SYS_RES_MEMORY, &rid, RF_ACTIVE);
    if (res == NULL) {
        error = ENXIO;
        goto cleanup;
    }

    /* Do the work */
    error = perform_transfer(res, temp_buffer, buffer, size);
    if (error != 0)
        goto cleanup;

cleanup:
    if (res != NULL)
        bus_release_resource(dev, SYS_RES_MEMORY, rid, res);
    if (temp_buffer != NULL)
        free(temp_buffer, M_DEVBUF);
}
```

```
    return (error);  
}
```

5.10.3 Resource Management Patterns

Kernel code must be extremely careful about resource management. FreeBSD uses several consistent patterns:

Acquire/Release Symmetry: Every resource acquisition has a corresponding release.

RAII-style Initialization: Initialize resources to NULL/invalid state, then check in cleanup code.

From `sys/dev/pci/pci.c`:

```
static int  
pci_attach(device_t dev)  
{  
    struct pci_softc *sc;  
    int busno, domain;  
    int error, rid;  
  
    sc = device_get_softc(dev);  
    domain = pcib_get_domain(dev);  
    busno = pcib_get_bus(dev);  
  
    if (bootverbose)  
        device_printf(dev, "domain=%d, physical bus=%d\n", domain, busno);  
  
    /* Initialize softc structure */  
    sc->sc_dev = dev;  
    sc->sc_domain = domain;  
    sc->sc_bus = busno;  
  
    /* Allocate bus resource */  
    rid = 0;  
    sc->sc_bus_res = bus_alloc_resource_any(dev, SYS_RES_MEMORY, &rid,  
                                           RF_ACTIVE);  
  
    if (sc->sc_bus_res == NULL) {  
        device_printf(dev, "Failed to allocate bus resource\n");  
        return (ENXIO);  
    }  
  
    /* Success - the detach method will handle cleanup */  
    return (0);  
}
```

```
static int
pci_detach(device_t dev)
{
    struct pci_softc *sc;

    sc = device_get_softc(dev);

    /* Release resources in reverse order of allocation */
    if (sc->sc_bus_res != NULL) {
        bus_release_resource(dev, SYS_RES_MEMORY, 0, sc->sc_bus_res);
        sc->sc_bus_res = NULL;
    }

    return (0);
}
```

5.10.4 Locking Patterns

FreeBSD provides several types of locks, each with specific usage patterns:

Mutexes: For protecting data structures and implementing critical sections.

```
static struct mtx global_lock;
static int protected_counter = 0;

/* Initialize during module load */
mtx_init(&global_lock, "global_lock", NULL, MTX_DEF);

void
increment_protected_counter(void)
{
    mtx_lock(&global_lock);
    protected_counter++;
    mtx_unlock(&global_lock);
}

/* Cleanup during module unload */
mtx_destroy(&global_lock);
```

Reader-Writer Locks: For data that's read frequently but written rarely.


```
static struct rwlock data_lock;
static struct data_structure shared_data;

int
read_shared_data(struct query *q, struct result *r)
{
    int error = 0;

    rw_rlock(&data_lock);
    error = search_data_structure(&shared_data, q, r);
    rw_runlock(&data_lock);

    return (error);
}

int
update_shared_data(struct update *u)
{
    int error = 0;

    rw_wlock(&data_lock);
    error = modify_data_structure(&shared_data, u);
    rw_wunlock(&data_lock);

    return (error);
}
```

5.10.5 Assertion and Debugging Patterns

FreeBSD makes extensive use of assertions to catch programming errors during development:

```
#include <sys/systm.h>

void
process_buffer(char *buffer, size_t size, int flags)
{
    /* Parameter assertions */
    KASSERT(buffer != NULL, ("process_buffer: null buffer"));
    KASSERT(size > 0, ("process_buffer: zero size"));
    KASSERT((flags & ~VALID_FLAGS) == 0,
        ("process_buffer: invalid flags 0x%x", flags));

    /* State assertions */
    KASSERT(device_is_attached(current_device),
```

```

        ("process_buffer: device not attached"));

    /* ... function implementation ... */
}

```

MPASS(): Similar to **KASSERT()** but always enabled, even in production kernels.

```

void
critical_function(void *ptr)
{
    MPASS(ptr != NULL); /* Always checked */
    /* ... */
}

```

5.10.6 Memory Allocation Patterns

Consistent patterns for memory management reduce bugs:

Initialization Pattern:

```

struct my_structure *
allocate_my_structure(int id)
{
    struct my_structure *ms;

    ms = malloc(sizeof(*ms), M_DEVBUF, M_WAITOK | M_ZERO);
    KASSERT(ms != NULL, ("malloc with M_WAITOK returned NULL"));

    /* Initialize non-zero fields */
    ms->id = id;
    ms->magic = MY_STRUCTURE_MAGIC;
    TAILQ_INIT(&ms->work_queue);
    mtx_init(&ms->lock, "my_struct", NULL, MTX_DEF);

    return (ms);
}

void
free_my_structure(struct my_structure *ms)
{
    if (ms == NULL)
        return;

    KASSERT(ms->magic == MY_STRUCTURE_MAGIC,

```

```

        ("free_my_structure: bad magic"));

    /* Cleanup in reverse order */
    mtx_destroy(&ms->lock);
    ms->magic = 0; /* Poison the structure */
    free(ms, M_DEVBUF);
}

```

5.10.7 Function Naming and Organization

FreeBSD follows consistent naming patterns that make code self-documenting:

Subsystem prefixes: `vm_` for virtual memory, `vfs_` for filesystem, `pci_` for PCI bus code.

Action suffixes: `_alloc/_free`, `_create/_destroy`, `_lock/_unlock`.

Static vs. external: Static functions often have shorter names since they're used only within the file.

```

/* External interface - full subsystem prefix */
int vfs_mount(struct mount *mp, struct thread *td);

/* Internal helper - shorter name */
static int validate_mount_args(struct mount *mp);

/* Paired operations */
struct vnode *vfs_cache_lookup(struct vnode *dvp, char *name);
void vfs_cache_enter(struct vnode *dvp, struct vnode *vp, char *name);

```

5.10.8 Hands-On Lab: Implementing Kernel Coding Patterns

Let's create a module that demonstrates proper kernel coding style:

```

/*
 * style_demo.c - Demonstrate FreeBSD kernel coding patterns
 *
 * This module shows proper KNF style, error handling, resource management,
 * and other kernel programming idioms.
 */

#include <sys/param.h>
#include <sys/kernel.h>
#include <sys/lock.h>
#include <sys/malloc.h>
#include <sys/module.h>

```

```
#include <sys/mutex.h>
#include <sys/queue.h>
#include <sys/system.h>

MALLOC_DEFINE(M_STYLEDemo, "styledemo", "Style demo structures");

/* Magic number for structure validation */
#define DEMO_ITEM_MAGIC    0xDEADBEEF

/*
 * Demo structure showing proper initialization and validation patterns
 */
struct demo_item {
    TAILQ_ENTRY(demo_item) di_link;    /* Queue linkage */
    uint32_t di_magic;                /* Structure validation */
    int di_id;                        /* Item identifier */
    char di_name[32];                 /* Item name */
    int di_refcount;                  /* Reference count */
};

TAILQ_HEAD(demo_item_list, demo_item);

/*
 * Module global state
 */
static struct demo_item_list item_list = TAILQ_HEAD_INITIALIZER(item_list);
static struct mtx item_list_lock;
static int next_item_id = 1;

/*
 * Forward declarations for static functions
 */
static struct demo_item *demo_item_alloc(const char *name);
static void demo_item_free(struct demo_item *item);
static struct demo_item *demo_item_find_locked(int id);
static void demo_item_ref(struct demo_item *item);
static void demo_item_unref(struct demo_item *item);

/*
 * demo_item_alloc - allocate and initialize a demo item
 *
 * Returns pointer to new item on success, NULL on failure.
 * The returned item has reference count 1.
 */
```

```

static struct demo_item *
demo_item_alloc(const char *name)
{
    struct demo_item *item;

    /* Parameter validation */
    if (name == NULL)
        return (NULL);

    if (strlen(name, sizeof(item->di_name)) >= sizeof(item->di_name))
        return (NULL);

    /* Allocate and initialize */
    item = malloc(sizeof(*item), M_STYLEDemo, M_WAITOK | M_ZERO);
    KASSERT(item != NULL, ("malloc with M_WAITOK returned NULL"));

    item->di_magic = DEMO_ITEM_MAGIC;
    item->di_refcount = 1;
    strncpy(item->di_name, name, sizeof(item->di_name));

    /* Assign ID while holding lock */
    mtx_lock(&item_list_lock);
    item->di_id = next_item_id++;
    TAILQ_INSERT_TAIL(&item_list, item, di_link);
    mtx_unlock(&item_list_lock);

    return (item);
}

/*
 * demo_item_free - free a demo item
 *
 * The item must have reference count 0 and must not be on any lists.
 */
static void
demo_item_free(struct demo_item *item)
{
    if (item == NULL)
        return;

    KASSERT(item->di_magic == DEMO_ITEM_MAGIC,
        ("demo_item_free: bad magic 0x%x", item->di_magic));
    KASSERT(item->di_refcount == 0,
        ("demo_item_free: refcount %d", item->di_refcount));
}

```

```
    /* Poison the structure */
    item->di_magic = 0;
    free(item, M_STYLEDEMO);
}

/*
 * demo_item_find_locked - find item by ID
 *
 * Must be called with item_list_lock held.
 * Returns item with incremented reference count, or NULL if not found.
 */
static struct demo_item *
demo_item_find_locked(int id)
{
    struct demo_item *item;

    mtx_assert(&item_list_lock, MA_OWNED);

    TAILQ_FOREACH(item, &item_list, di_link) {
        KASSERT(item->di_magic == DEMO_ITEM_MAGIC,
            ("demo_item_find_locked: bad magic"));

        if (item->di_id == id) {
            demo_item_ref(item);
            return (item);
        }
    }

    return (NULL);
}

/*
 * demo_item_ref - increment reference count
 */
static void
demo_item_ref(struct demo_item *item)
{
    KASSERT(item != NULL, ("demo_item_ref: null item"));
    KASSERT(item->di_magic == DEMO_ITEM_MAGIC,
        ("demo_item_ref: bad magic"));
    KASSERT(item->di_refcount > 0,
        ("demo_item_ref: zero refcount"));
}
```

```
    atomic_add_int(&item->di_refcount, 1);
}

/*
 * demo_item_unref - decrement reference count and free if zero
 */
static void
demo_item_unref(struct demo_item *item)
{
    int old_refs;

    if (item == NULL)
        return;

    KASSERT(item->di_magic == DEMO_ITEM_MAGIC,
            ("demo_item_unref: bad magic"));
    KASSERT(item->di_refcount > 0,
            ("demo_item_unref: zero refcount"));

    old_refs = atomic_fetchadd_int(&item->di_refcount, -1);
    if (old_refs == 1) {
        /* Last reference - remove from list and free */
        mtx_lock(&item_list_lock);
        TAILQ_REMOVE(&item_list, item, di_link);
        mtx_unlock(&item_list_lock);

        demo_item_free(item);
    }
}

/*
 * Module event handler
 */
static int
style_demo_load(module_t mod, int cmd, void *arg)
{
    struct demo_item *item1, *item2, *found_item;
    int error = 0;

    switch (cmd) {
    case MOD_LOAD:
        printf("=== Kernel Style Demo ===\n");

        /* Initialize module state */
    }
```

```
mtx_init(&item_list_lock, "item_list", NULL, MTX_DEF);

/* Demonstrate proper allocation and initialization */
item1 = demo_item_alloc("first_item");
if (item1 == NULL) {
    printf("Failed to allocate first item\n");
    error = ENOMEM;
    goto cleanup;
}
printf("Created item %d: '%s'\n", item1->di_id, item1->di_name);

item2 = demo_item_alloc("second_item");
if (item2 == NULL) {
    printf("Failed to allocate second item\n");
    error = ENOMEM;
    goto cleanup;
}
printf("Created item %d: '%s'\n", item2->di_id, item2->di_name);

/* Demonstrate lookup and reference counting */
mtx_lock(&item_list_lock);
found_item = demo_item_find_locked(item1->di_id);
mtx_unlock(&item_list_lock);

if (found_item != NULL) {
    printf("Found item %d (refcount was incremented)\n",
        found_item->di_id);
    demo_item_unref(found_item); /* Release lookup reference */
}

/* Clean up - items will be freed when refcount reaches 0 */
demo_item_unref(item1);
demo_item_unref(item2);

printf("Style demo completed successfully.\n");
break;

case MOD_UNLOAD:
    /* Verify all items were properly cleaned up */
    mtx_lock(&item_list_lock);
    if (!TAILQ_EMPTY(&item_list)) {
        printf("WARNING: item list not empty at module unload\n");
    }
    mtx_unlock(&item_list_lock);
```



```
        mtx_destroy(&item_list_lock);
        printf("Style demo module unloaded.\n");
        break;

default:
    error = EOPNOTSUPP;
    break;
}

cleanup:
    if (error != 0 && cmd == MOD_LOAD) {
        /* Cleanup on load failure */
        mtx_destroy(&item_list_lock);
    }

    return (error);
}

/*
 * Module declaration
 */
static moduledata_t style_demo_mod = {
    "style_demo",
    style_demo_load,
    NULL
};

DECLARE_MODULE(style_demo, style_demo_mod, SI_SUB_DRIVERS, SI_ORDER_MIDDLE);
MODULE_VERSION(style_demo, 1);
```

5.10.9 Key Takeaways from Kernel Coding Style

Consistency matters: Follow established patterns even if you prefer different approaches.

Defensive programming: Use assertions, validate parameters, and handle edge cases.

Resource discipline: Always pair allocation with deallocation, initialization with cleanup.

Clear naming: Use descriptive names that follow subsystem conventions.

Proper locking: Protect shared data and document locking requirements.

Error handling: Use consistent patterns for error detection, reporting, and recovery.

5.10.10 Summary

FreeBSD's coding idioms aren't arbitrary rules; they're distilled wisdom from decades of kernel development. Following these patterns makes your code:

- Easier for other developers to read and understand
- Less likely to contain subtle bugs
- More consistent with the existing kernel codebase
- Easier to maintain and debug

The patterns we've covered form the foundation for writing robust, maintainable kernel code. In the next section, we'll build on this foundation to explore defensive programming techniques that help prevent the subtle bugs that can bring down entire systems.

5.11 Defensive C in the Kernel

Writing defensive code means programming as if everything that can go wrong will go wrong. In user-space programming, this might seem paranoid; in kernel programming, it's essential for survival. A single null pointer dereference, buffer overflow, or race condition can crash the entire system, corrupt data, or create security vulnerabilities that affect every process on the machine.

Defensive kernel programming isn't just about avoiding bugs; it's about building robust systems that gracefully handle unexpected conditions, malicious input, and hardware failures. This section will teach you the mindset and techniques that separate reliable kernel code from code that works "most of the time."

5.11.1 The Paranoid Mindset

The first step in defensive programming is developing the right attitude: **assume the worst will happen**. This means:

- **Every pointer might be NULL**
- **Every buffer might be too small**
- **Every allocation might fail**
- **Every system call might be interrupted**
- **Every hardware operation might time out**
- **Every user input might be malicious**

Here's an example of non-defensive code that looks reasonable but has hidden dangers:

```
/* DANGEROUS - multiple assumptions that can be wrong */
void
process_user_data(struct user_request *req)
{
    char *buffer = malloc(req->data_size, M_TEMP, M_WAITOK);

    /* Assumption: req is not NULL */
```

```
/* Assumption: req->data_size is reasonable */
/* Assumption: malloc always succeeds with M_WAITOK */

copyin(req->user_buffer, buffer, req->data_size);
/* Assumption: user_buffer is valid */
/* Assumption: data_size matches actual user buffer size */

process_buffer(buffer, req->data_size);
free(buffer, M_TEMP);
}
```

Here's the defensive version:

```
/* DEFENSIVE - validate everything, handle all failures */
int
process_user_data(struct user_request *req)
{
    char *buffer = NULL;
    int error = 0;

    /* Validate parameters */
    if (req == NULL) {
        return (EINVAL);
    }

    if (req->data_size == 0 || req->data_size > MAX_USER_DATA_SIZE) {
        return (EINVAL);
    }

    if (req->user_buffer == NULL) {
        return (EFAULT);
    }

    /* Allocate buffer with error checking */
    buffer = malloc(req->data_size, M_TEMP, M_WAITOK);
    if (buffer == NULL) { /* Defensive: check even M_WAITOK */
        return (ENOMEM);
    }

    /* Safe copy from user space */
    error = copyin(req->user_buffer, buffer, req->data_size);
    if (error != 0) {
        goto cleanup;
    }
}
```

```
/* Process with error checking */
error = process_buffer(buffer, req->data_size);

cleanup:
    if (buffer != NULL) {
        free(buffer, M_TEMP);
    }

    return (error);
}
```

5.11.2 Input Validation: Trust No One

Never trust data that comes from outside your immediate control. This includes:

- User-space programs (via system calls)
- Hardware devices (via device registers)
- Network packets
- File system contents
- Even other kernel subsystems (they have bugs too)

Here is an illustrative system-call prologue in the same style as the real `sys_read()` in `/usr/src/sys/kern/sys_generic.c`:

```
int
my_syscall(struct thread *td, struct my_args *uap)
{
    struct file *fp;
    int error;

    if (uap->nbyte > IOSIZE_MAX)
        return (EINVAL);

    AUDIT_ARG_FD(uap->fd);
    error = fget_read(td, uap->fd, &cap_read_rights, &fp);
    if (error != 0)
        return (error);

    /* ... validated fd and fp are now safe to use ... */

    fdrop(fp, td);
    return (0);
}
```

Notice how the single size check is done first (cheap and uses no resources), the audit record is emitted next, and the file descriptor is resolved and reference-counted through `fget_read()` / `fdrop()`. The real `sys_read()` in FreeBSD 14.3 is even shorter: it validates the size and hands the rest off to `kern_readv()`, which does the work.

5.11.3 Integer Overflow Prevention

Integer overflow is a common source of security vulnerabilities in kernel code. Always check arithmetic operations that might overflow:

```
/* VULNERABLE - integer overflow can bypass size check */
int
allocate_user_buffer(size_t element_size, size_t element_count)
{
    size_t total_size = element_size * element_count; /* Can overflow! */

    if (total_size > MAX_BUFFER_SIZE) {
        return (EINVAL);
    }

    /* If overflow occurred, total_size might be small and pass the check */
    return (allocate_buffer(total_size));
}

/* SAFE - check for overflow before multiplication */
int
allocate_user_buffer_safe(size_t element_size, size_t element_count)
{
    size_t total_size;

    /* Check for multiplication overflow */
    if (element_count != 0 && element_size > SIZE_MAX / element_count) {
        return (EINVAL);
    }

    total_size = element_size * element_count;

    if (total_size > MAX_BUFFER_SIZE) {
        return (EINVAL);
    }

    return (allocate_buffer(total_size));
}
```

FreeBSD provides helper macros for safe arithmetic in `<sys/systm.h>`:

```
/* Safe arithmetic macros */
if (howmany(total_bytes, block_size) > max_blocks) {
    return (EFBIG);
}

/* Round up safely */
size_t rounded = roundup2(size, alignment);
if (rounded < size) { /* Check for overflow */
    return (EINVAL);
}
```

5.11.4 Buffer Management and Bounds Checking

Buffer overruns are among the most dangerous bugs in kernel code. Always use safe string and memory functions:

```
/* DANGEROUS - no bounds checking */
void
format_device_info(struct device *dev, char *buffer)
{
    sprintf(buffer, "Device: %s, ID: %d", dev->name, dev->id); /* Overflow! */
}

/* SAFE - explicit buffer size and bounds checking */
int
format_device_info_safe(struct device *dev, char *buffer, size_t bufsize)
{
    int len;

    if (dev == NULL || buffer == NULL || bufsize == 0) {
        return (EINVAL);
    }

    len = snprintf(buffer, bufsize, "Device: %s, ID: %d",
        dev->name ? dev->name : "unknown", dev->id);

    if (len >= bufsize) {
        return (ENAMETOOLONG); /* Indicate truncation */
    }

    return (0);
}
```

5.11.5 Error Propagation Patterns

In kernel code, errors must be handled promptly and correctly. Don't ignore return values or mask errors:

```
/* WRONG - ignoring errors */
void
bad_error_handling(void)
{
    struct resource *res;

    res = allocate_resource(); /* Might return NULL */
    use_resource(res);         /* Will crash if res is NULL */
    free_resource(res);
}

/* RIGHT - proper error handling and propagation */
int
good_error_handling(struct device *dev)
{
    struct resource *res = NULL;
    int error = 0;

    res = allocate_resource(dev);
    if (res == NULL) {
        error = ENOMEM;
        goto cleanup;
    }

    error = configure_resource(res);
    if (error != 0) {
        goto cleanup;
    }

    error = use_resource(res);
    /* Fall through to cleanup */

cleanup:
    if (res != NULL) {
        free_resource(res);
    }

    return (error);
}
```

5.11.6 Race Condition Prevention

In multiprocessor systems, race conditions can cause subtle corruption. Always protect shared data with appropriate synchronization:

```
/* DANGEROUS - race condition on shared counter */
static int request_counter = 0;

int
get_next_request_id(void)
{
    return (++request_counter); /* Not atomic! */
}

/* SAFE - using atomic operations */
static volatile u_int request_counter = 0;

u_int
get_next_request_id_safe(void)
{
    return (atomic_fetchadd_int(&request_counter, 1) + 1);
}

/* ALSO SAFE - using a mutex for more complex operations */
static int request_counter = 0;
static struct mtx counter_lock;

u_int
get_next_request_id_locked(void)
{
    u_int id;

    mtx_lock(&counter_lock);
    id = ++request_counter;
    mtx_unlock(&counter_lock);

    return (id);
}
```

5.11.7 Resource Leak Prevention

Kernel memory leaks and resource leaks can degrade system performance over time. Use consistent patterns to ensure cleanup:


```
/* Resource management with automatic cleanup */
struct operation_context {
    struct mtx *lock;
    void *buffer;
    struct resource *hw_resource;
    int flags;
};

static void
cleanup_context(struct operation_context *ctx)
{
    if (ctx == NULL)
        return;

    if (ctx->hw_resource != NULL) {
        release_hardware_resource(ctx->hw_resource);
        ctx->hw_resource = NULL;
    }

    if (ctx->buffer != NULL) {
        free(ctx->buffer, M_TEMP);
        ctx->buffer = NULL;
    }

    if (ctx->lock != NULL) {
        mtx_unlock(ctx->lock);
        ctx->lock = NULL;
    }
}

int
complex_operation(struct device *dev, void *user_data, size_t data_size)
{
    struct operation_context ctx = { 0 }; /* Zero-initialize */
    int error = 0;

    /* Acquire resources in order */
    ctx.lock = get_device_lock(dev);
    if (ctx.lock == NULL) {
        error = EBUSY;
        goto cleanup;
    }
    mtx_lock(ctx.lock);
```

```

    ctx.buffer = malloc(data_size, M_TEMP, M_WAITOK);
    if (ctx.buffer == NULL) {
        error = ENOMEM;
        goto cleanup;
    }

    ctx.hw_resource = acquire_hardware_resource(dev);
    if (ctx.hw_resource == NULL) {
        error = ENXIO;
        goto cleanup;
    }

    /* Perform operation */
    error = copyin(user_data, ctx.buffer, data_size);
    if (error != 0) {
        goto cleanup;
    }

    error = process_with_hardware(ctx.hw_resource, ctx.buffer, data_size);

cleanup:
    cleanup_context(&ctx); /* Always cleanup, regardless of errors */
    return (error);
}

```

5.11.8 Assertions for Development

Use assertions to catch programming errors during development. FreeBSD provides several assertion macros:

```

#include <sys/systm.h>

void
process_network_packet(struct mbuf *m, struct ifnet *ifp)
{
    struct ip *ip;
    int hlen;

    /* Parameter validation assertions */
    KASSERT(m != NULL, ("process_network_packet: null mbuf"));
    KASSERT(ifp != NULL, ("process_network_packet: null interface"));
    KASSERT(m->m_len >= sizeof(struct ip),
        ("process_network_packet: mbuf too small"));

    ip = mtod(m, struct ip *);
}

```

```

    /* Sanity check assertions */
    KASSERT(ip->ip_v == IPVERSION, ("invalid IP version %d", ip->ip_v));

    hlen = ip->ip_hl << 2;
    KASSERT(hlen >= sizeof(struct ip) && hlen <= m->m_len,
        ("invalid IP header length %d", hlen));

    /* State consistency assertions */
    KASSERT((ifp->if_flags & IFF_UP) != 0,
        ("processing packet on down interface"));

    /* Process the packet... */
}

```

5.11.9 Hands-On Lab: Building Defensive Kernel Code

Let's create a module that demonstrates defensive programming techniques:

```

/*
 * defensive_demo.c - Demonstrate defensive programming in kernel code
 */
#include <sys/param.h>
#include <sys/kernel.h>
#include <sys/lock.h>
#include <sys/malloc.h>
#include <sys/module.h>
#include <sys/mutex.h>
#include <sys/system.h>
#include <machine/atomic.h>

MALLOC_DEFINE(M_DEFTTEST, "defctest", "Defensive programming test");

#define MAX_BUFFER_SIZE    4096
#define MAX_NAME_LENGTH    64
#define DEMO_MAGIC         0x12345678

struct demo_buffer {
    uint32_t db_magic;        /* Structure validation */
    size_t db_size;          /* Allocated size */
    size_t db_used;          /* Used bytes */
    char db_name[MAX_NAME_LENGTH];
    void *db_data;           /* Buffer data */
    volatile u_int db_refcount;
}

```

```
};

/*
 * Safe buffer allocation with comprehensive validation
 */
static struct demo_buffer *
demo_buffer_alloc(const char *name, size_t size)
{
    struct demo_buffer *db;
    size_t name_len;

    /* Input validation */
    if (name == NULL) {
        printf("demo_buffer_alloc: NULL name\n");
        return (NULL);
    }

    name_len = strlen(name, MAX_NAME_LENGTH);
    if (name_len == 0 || name_len >= MAX_NAME_LENGTH) {
        printf("demo_buffer_alloc: invalid name length %zu\n", name_len);
        return (NULL);
    }

    if (size == 0 || size > MAX_BUFFER_SIZE) {
        printf("demo_buffer_alloc: invalid size %zu\n", size);
        return (NULL);
    }

    /* Check for potential overflow in total allocation size */
    if (SIZE_MAX - sizeof(*db) < size) {
        printf("demo_buffer_alloc: size overflow\n");
        return (NULL);
    }

    /* Allocate structure */
    db = malloc(sizeof(*db), M_DEFTEST, M_WAITOK | M_ZERO);
    if (db == NULL) { /* Defensive: check even with M_WAITOK */
        printf("demo_buffer_alloc: failed to allocate structure\n");
        return (NULL);
    }

    /* Allocate data buffer */
    db->db_data = malloc(size, M_DEFTEST, M_WAITOK);
    if (db->db_data == NULL) {
```

```
    printf("demo_buffer_alloc: failed to allocate data buffer\n");
    free(db, M_DEFTTEST);
    return (NULL);
}

/* Initialize structure */
db->db_magic = DEMO_MAGIC;
db->db_size = size;
db->db_used = 0;
db->db_refcount = 1;
strncpy(db->db_name, name, sizeof(db->db_name));

return (db);
}

/*
 * Safe buffer deallocation with validation
 */
static void
demo_buffer_free(struct demo_buffer *db)
{
    if (db == NULL)
        return;

    /* Validate structure */
    if (db->db_magic != DEMO_MAGIC) {
        printf("demo_buffer_free: bad magic 0x%x (expected 0x%x)\n",
            db->db_magic, DEMO_MAGIC);
        return;
    }

    /* Verify reference count */
    if (db->db_refcount != 0) {
        printf("demo_buffer_free: non-zero refcount %u\n", db->db_refcount);
        return;
    }

    /* Clear sensitive data and poison structure */
    if (db->db_data != NULL) {
        memset(db->db_data, 0, db->db_size); /* Clear data */
        free(db->db_data, M_DEFTTEST);
        db->db_data = NULL;
    }
}
```

```
    db->db_magic = 0xDEADBEEF;  /* Poison magic */
    free(db, M_DEFTTEST);
}

/*
 * Safe buffer reference counting
 */
static void
demo_buffer_ref(struct demo_buffer *db)
{
    u_int old_refs;

    if (db == NULL) {
        printf("demo_buffer_ref: NULL buffer\n");
        return;
    }

    if (db->db_magic != DEMO_MAGIC) {
        printf("demo_buffer_ref: bad magic\n");
        return;
    }

    old_refs = atomic_fetchadd_int(&db->db_refcount, 1);
    if (old_refs == 0) {
        printf("demo_buffer_ref: attempting to ref freed buffer\n");
        /* Try to undo the increment */
        atomic_subtract_int(&db->db_refcount, 1);
    }
}

static void
demo_buffer_unref(struct demo_buffer *db)
{
    u_int old_refs;

    if (db == NULL) {
        return;
    }

    if (db->db_magic != DEMO_MAGIC) {
        printf("demo_buffer_unref: bad magic\n");
        return;
    }
}
```

```
old_refs = atomic_fetchadd_int(&db->db_refcount, -1);
if (old_refs == 0) {
    printf("demo_buffer_unref: buffer already at zero refcount\n");
    atomic_add_int(&db->db_refcount, 1); /* Undo the decrement */
    return;
}

if (old_refs == 1) {
    /* Last reference - safe to free */
    demo_buffer_free(db);
}
}

/*
 * Safe data writing with bounds checking
 */
static int
demo_buffer_write(struct demo_buffer *db, const void *data, size_t len,
                  size_t offset)
{
    if (db == NULL || data == NULL) {
        return (EINVAL);
    }

    if (db->db_magic != DEMO_MAGIC) {
        printf("demo_buffer_write: bad magic\n");
        return (EINVAL);
    }

    if (len == 0) {
        return (0); /* Nothing to do */
    }

    /* Check for integer overflow in offset + len */
    if (offset > db->db_size || len > db->db_size - offset) {
        printf("demo_buffer_write: write would exceed buffer bounds\n");
        return (EOVERFLOW);
    }

    /* Perform the write */
    memcpy((char *)db->db_data + offset, data, len);

    /* Update used size */
    if (offset + len > db->db_used) {
```

```
        db->db_used = offset + len;
    }

    return (0);
}

static int
defensive_demo_load(module_t mod, int cmd, void *arg)
{
    struct demo_buffer *db1, *db2;
    const char *test_data = "Hello, defensive kernel world!";
    int error;

    switch (cmd) {
    case MOD_LOAD:
        printf("=== Defensive Programming Demo ===\n");

        /* Test normal allocation */
        db1 = demo_buffer_alloc("test_buffer", 256);
        if (db1 == NULL) {
            printf("Failed to allocate test buffer\n");
            return (ENOMEM);
        }
        printf("Allocated buffer '%s' with size %zu\n",
            db1->db_name, db1->db_size);

        /* Test safe writing */
        error = demo_buffer_write(db1, test_data, strlen(test_data), 0);
        if (error != 0) {
            printf("Write failed with error %d\n", error);
        } else {
            printf("Successfully wrote %zu bytes\n", strlen(test_data));
        }

        /* Test reference counting */
        demo_buffer_ref(db1);
        printf("Incremented reference count to %u\n", db1->db_refcount);

        demo_buffer_unref(db1);
        printf("Decrementd reference count to %u\n", db1->db_refcount);

        /* Test parameter validation (should fail gracefully) */
        db2 = demo_buffer_alloc(NULL, 100);          /* NULL name */
        if (db2 == NULL) {
```



```
        printf("Correctly rejected NULL name\n");
    }

    db2 = demo_buffer_alloc("test", 0);          /* Zero size */
    if (db2 == NULL) {
        printf("Correctly rejected zero size\n");
    }

    db2 = demo_buffer_alloc("test", MAX_BUFFER_SIZE + 1); /* Too large */
    if (db2 == NULL) {
        printf("Correctly rejected oversized buffer\n");
    }

    /* Test bounds checking */
    error = demo_buffer_write(db1, test_data, 1000, 0); /* Too much data */
    if (error != 0) {
        printf("Correctly rejected oversized write: %d\n", error);
    }

    /* Clean up */
    demo_buffer_unref(db1); /* Final reference */

    printf("Defensive programming demo completed successfully.\n");
    break;

case MOD_UNLOAD:
    printf("Defensive demo module unloaded.\n");
    break;

default:
    return (EOPNOTSUPP);
}

return (0);
}

static moduledata_t defensive_demo_mod = {
    "defensive_demo",
    defensive_demo_load,
    NULL
};

DECLARE_MODULE(defensive_demo, defensive_demo_mod, SI_SUB_DRIVERS, SI_ORDER_MIDDLE);
MODULE_VERSION(defensive_demo, 1);
```

5.11.10 Summary of Defensive Programming Principles

Validate everything: Check all parameters, return values, and assumptions.

Handle all errors: Don't ignore return codes or assume operations will succeed.

Use safe functions: Prefer bounds-checking versions of string and memory functions.

Prevent integer overflow: Check arithmetic operations that might wrap around.

Manage resources carefully: Use consistent allocation/deallocation patterns.

Protect against races: Use appropriate synchronization for shared data.

Assert invariants: Use KASSERT to catch programming errors during development.

Fail safely: When something goes wrong, fail in a way that doesn't compromise system security or stability.

Defensive programming isn't about being paranoid; it's about being realistic. In kernel space, the cost of failure is too high to take chances with assumptions or shortcuts.

5.11.11 Kernel Attributes and Error Handling Idioms

FreeBSD's kernel uses several compiler attributes and established error handling patterns to make code safer, more efficient, and easier to debug. Understanding these idioms will help you write kernel code that follows the patterns experienced FreeBSD developers expect.

5.11.12 Compiler Attributes for Kernel Safety

Modern C compilers provide attributes that help catch bugs at compile time and optimize code for specific usage patterns. FreeBSD makes extensive use of these in kernel code.

__unused: Suppress warnings about unused parameters or variables.

```
/* Callback function that doesn't use all parameters */
static int
my_callback(device_t dev __unused, void *arg, int flag __unused)
{
    struct my_context *ctx = arg;

    return (ctx->process());
}
```

__printflike: Enable format string checking for printf-style functions.

```
/* Custom logging function with printf format checking */
static void __printflike(2, 3)
device_log(struct device *dev, const char *fmt, ...)
{
```

```
va_list ap;
char buffer[256];

va_start(ap, fmt);
vsprintf(buffer, sizeof(buffer), fmt, ap);
va_end(ap);

printf("Device %s: %s\n", device_get_nameunit(dev), buffer);
}
```

__predict_true and **__predict_false**: Help the compiler optimize branch prediction.

```
int
allocate_with_fallback(size_t size, int flags)
{
    void *ptr;

    ptr = malloc(size, M_DEVBUF, flags | M_NOWAIT);
    if (__predict_true(ptr != NULL)) {
        return (0); /* Common case - success */
    }

    /* Rare case - try emergency allocation */
    if (__predict_false(flags & M_USE_RESERVE)) {
        ptr = malloc(size, M_DEVBUF, M_USE_RESERVE | M_NOWAIT);
        if (ptr != NULL) {
            return (0);
        }
    }

    return (ENOMEM);
}
```

Here's a real example from `sys/kern/kern_malloc.c`:

```
void *
malloc(size_t size, struct malloc_type *mtp, int flags)
{
    int indx;
    caddr_t va;
    uma_zone_t zone;

    if (__predict_false(size > kmem_zmax)) {
```

```

    /* Large allocation - uncommon case */
    va = uma_large_malloc(size, flags);
    if (va != NULL)
        malloc_type_allocated(mtp, va ? size : 0);
    return ((void *) va);
}

/* Small allocation - common case */
indx = zone_index_of(size);
zone = malloc_type_zone_idx_to_zone[indx];
va = uma_zalloc_arg(zone, mtp, flags);
if (__predict_true(va != NULL))
    size = zone_get_size(zone);
malloc_type_allocated(mtp, size);

return ((void *) va);
}

```

__diagnosed: Mark variables used only in diagnostic code (assertions, debugging).

```

static int
validate_buffer(struct buffer *buf)
{
    size_t expected_size __diagnosed;

    KASSERT(buf != NULL, ("validate_buffer: null buffer"));

    expected_size = calculate_expected_size(buf->type);
    KASSERT(buf->size == expected_size,
        ("buffer size %zu, expected %zu", buf->size, expected_size));

    return (buf->flags & BUFFER_VALID);
}

```

5.11.13 Error Code Conventions and Patterns

FreeBSD kernel functions follow consistent patterns for error handling that make code predictable and debuggable.

Standard Error Codes: Use `errno` values defined in `<sys/errno.h>`.

```

#include <sys/errno.h>

int

```

```
process_user_request(struct user_request *req)
{
    if (req == NULL) {
        return (EINVAL);    /* Invalid argument */
    }

    if (req->size > MAX_REQUEST_SIZE) {
        return (E2BIG);    /* Argument list too long */
    }

    if (!user_has_permission(req->uid)) {
        return (EPERM);    /* Operation not permitted */
    }

    if (system_resources_exhausted()) {
        return (EAGAIN);    /* Resource temporarily unavailable */
    }

    /* Success */
    return (0);
}
```

Error Aggregation Pattern: Collect multiple errors but return the most important one.

```
int
initialize_device_subsystems(struct device *dev)
{
    int error, final_error = 0;

    error = init_power_management(dev);
    if (error != 0) {
        device_printf(dev, "Power management init failed: %d\n", error);
        final_error = error;    /* Remember first serious error */
    }

    error = init_dma_engine(dev);
    if (error != 0) {
        device_printf(dev, "DMA engine init failed: %d\n", error);
        if (final_error == 0) { /* Only update if no previous error */
            final_error = error;
        }
    }

    error = init_interrupts(dev);
```

```
if (error != 0) {
    device_printf(dev, "Interrupt init failed: %d\n", error);
    if (final_error == 0) {
        final_error = error;
    }
}

return (final_error);
}
```

Error Context Pattern: Provide detailed error information for debugging.

```
struct error_context {
    int error_code;
    const char *operation;
    const char *file;
    int line;
    uintptr_t context_data;
};

#define SET_ERROR_CONTEXT(ctx, code, op, data) do { \
    (ctx)->error_code = (code); \
    (ctx)->operation = (op); \
    (ctx)->file = __FILE__; \
    (ctx)->line = __LINE__; \
    (ctx)->context_data = (uintptr_t)(data); \
} while (0)

static int
complex_device_operation(struct device *dev, struct error_context *err_ctx)
{
    int error;

    error = step_one(dev);
    if (error != 0) {
        SET_ERROR_CONTEXT(err_ctx, error, "device initialization", dev);
        return (error);
    }

    error = step_two(dev);
    if (error != 0) {
        SET_ERROR_CONTEXT(err_ctx, error, "hardware configuration", dev);
        return (error);
    }
}
```

```

    return (0);
}

```

5.11.14 Debugging and Diagnostic Idioms

FreeBSD provides several idioms for making code easier to debug and diagnose in production systems.

Debug Levels: Use different levels of diagnostic output.

```

#define DEBUG_LEVEL_NONE    0
#define DEBUG_LEVEL_ERROR   1
#define DEBUG_LEVEL_WARN    2
#define DEBUG_LEVEL_INFO    3
#define DEBUG_LEVEL_VERBOSE 4

static int debug_level = DEBUG_LEVEL_ERROR;

#define DPRINTF(level, fmt, ...) do {                                \
    if ((level) <= debug_level) {                                    \
        printf("%s: " fmt "\n", __func__, ##__VA_ARGS__); \
    }                                                                \
} while (0)

void
process_network_packet(struct mbuf *m)
{
    struct ip *ip = mtod(m, struct ip *);

    DPRINTF(DEBUG_LEVEL_VERBOSE, "processing packet of %d bytes", m->m_len);

    if (ip->ip_v != IPVERSION) {
        DPRINTF(DEBUG_LEVEL_ERROR, "invalid IP version %d", ip->ip_v);
        return;
    }

    DPRINTF(DEBUG_LEVEL_INFO, "packet from %s", inet_ntoa(ip->ip_src));
}

```

State Tracking: Maintain internal state for debugging and validation.

```

enum device_state {
    DEVICE_STATE_UNINITIALIZED = 0,
    DEVICE_STATE_INITIALIZING,

```

```
    DEVICE_STATE_READY,  
    DEVICE_STATE_ACTIVE,  
    DEVICE_STATE_SUSPENDED,  
    DEVICE_STATE_ERROR  
};  
  
struct device_context {  
    enum device_state state;  
    int error_count;  
    sbintime_t last_activity;  
    uint32_t debug_flags;  
};  
  
static const char *  
device_state_name(enum device_state state)  
{  
    static const char *names[] = {  
        [DEVICE_STATE_UNINITIALIZED] = "uninitialized",  
        [DEVICE_STATE_INITIALIZING] = "initializing",  
        [DEVICE_STATE_READY] = "ready",  
        [DEVICE_STATE_ACTIVE] = "active",  
        [DEVICE_STATE_SUSPENDED] = "suspended",  
        [DEVICE_STATE_ERROR] = "error"  
    };  
  
    if (state < nitems(names) && names[state] != NULL) {  
        return (names[state]);  
    }  
  
    return ("unknown");  
}  
  
static void  
set_device_state(struct device_context *ctx, enum device_state new_state)  
{  
    enum device_state old_state;  
  
    KASSERT(ctx != NULL, ("set_device_state: null context"));  
  
    old_state = ctx->state;  
    ctx->state = new_state;  
    ctx->last_activity = sbinuptime();  
  
    DPRINTF(DEBUG_LEVEL_INFO, "device state: %s -> %s",
```



```
        device_state_name(old_state), device_state_name(new_state));  
    }
```

5.11.15 Performance Monitoring Idioms

Kernel code often needs to track performance metrics and resource usage.

Counter Management: Use atomic counters for statistics.

```
struct device_stats {  
    volatile u_long packets_received;  
    volatile u_long packets_transmitted;  
    volatile u_long bytes_received;  
    volatile u_long bytes_transmitted;  
    volatile u_long errors;  
    volatile u_long drops;  
};  
  
static void  
update_rx_stats(struct device_stats *stats, size_t bytes)  
{  
    atomic_add_long(&stats->packets_received, 1);  
    atomic_add_long(&stats->bytes_received, bytes);  
}  
  
static void  
update_error_stats(struct device_stats *stats, int error_type)  
{  
    atomic_add_long(&stats->errors, 1);  
  
    if (error_type == ERROR_DROP) {  
        atomic_add_long(&stats->drops, 1);  
    }  
}
```

Timing Measurements: Track operation durations for performance analysis.

```
struct timing_context {  
    sbintime_t start_time;  
    sbintime_t end_time;  
    const char *operation;  
};  
  
static void
```

```

timing_start(struct timing_context *tc, const char *op)
{
    tc->operation = op;
    tc->start_time = sbinuptime();
    tc->end_time = 0;
}

static void
timing_end(struct timing_context *tc)
{
    sbintime_t duration;

    tc->end_time = sbinuptime();
    duration = tc->end_time - tc->start_time;

    /* Convert to microseconds for logging */
    DPRINTF(DEBUG_LEVEL_VERBOSE, "%s took %ld microseconds",
            tc->operation, sbintime_to_us(duration));
}

```

5.11.16 Hands-On Lab: Error Handling and Diagnostics

Let's create a comprehensive example that demonstrates these error handling and diagnostic idioms:

```

/*
 * error_demo.c - Demonstrate kernel error handling and diagnostic idioms
 */
#include <sys/param.h>
#include <sys/kernel.h>
#include <sys/malloc.h>
#include <sys/module.h>
#include <sys/system.h>
#include <sys/time.h>
#include <machine/atomic.h>

MALLOC_DEFINE(M_ERRTEST, "errtest", "Error handling test structures");

/* Debug levels */
#define DEBUG_ERROR    1
#define DEBUG_WARN     2
#define DEBUG_INFO     3
#define DEBUG_VERBOSE  4

static int debug_level = DEBUG_INFO;

```

```

#define DPRINTF(level, fmt, ...) do {                                \
    if ((level) <= debug_level) {                                    \
        printf("[%s:%d] " fmt "\n", __func__, __LINE__,           \
            ##__VA_ARGS__);                                         \
    }                                                                \
} while (0)

/* Error context for detailed error reporting */
struct error_context {
    int error_code;
    const char *operation;
    const char *file;
    int line;
    sbintime_t timestamp;
};

#define SET_ERROR(ctx, code, op) do {                                \
    if ((ctx) != NULL) {                                            \
        (ctx)->error_code = (code);                                \
        (ctx)->operation = (op);                                    \
        (ctx)->file = __FILE__;                                     \
        (ctx)->line = __LINE__;                                     \
        (ctx)->timestamp = sbinuptime();                           \
    }                                                                \
} while (0)

/* Statistics tracking */
struct operation_stats {
    volatile u_long total_attempts;
    volatile u_long successes;
    volatile u_long failures;
    volatile u_long invalid_params;
    volatile u_long resource_errors;
};

static struct operation_stats global_stats;

/* Test structure with validation */
#define TEST_MAGIC 0xABCDEF00
struct test_object {
    uint32_t magic;
    int id;
    size_t size;

```

```
void *data;
};

/*
 * Safe object allocation with comprehensive error handling
 */
static struct test_object *
test_object_alloc(int id, size_t size, struct error_context *err_ctx)
{
    struct test_object *obj = NULL;
    void *data = NULL;

    atomic_add_long(&global_stats.total_attempts, 1);

    /* Parameter validation */
    if (id < 0) {
        DPRINTF(DEBUG_ERROR, "Invalid ID %d", id);
        SET_ERROR(err_ctx, EINVAL, "parameter validation");
        atomic_add_long(&global_stats.invalid_params, 1);
        goto error;
    }

    if (size == 0 || size > 1024 * 1024) {
        DPRINTF(DEBUG_ERROR, "Invalid size %zu", size);
        SET_ERROR(err_ctx, EINVAL, "size validation");
        atomic_add_long(&global_stats.invalid_params, 1);
        goto error;
    }

    DPRINTF(DEBUG_VERBOSE, "Allocating object id=%d, size=%zu", id, size);

    /* Allocate structure */
    obj = malloc(sizeof(*obj), M_ERRTEST, M_NOWAIT | M_ZERO);
    if (obj == NULL) {
        DPRINTF(DEBUG_ERROR, "Failed to allocate object structure");
        SET_ERROR(err_ctx, ENOMEM, "structure allocation");
        atomic_add_long(&global_stats.resource_errors, 1);
        goto error;
    }

    /* Allocate data buffer */
    data = malloc(size, M_ERRTEST, M_NOWAIT);
    if (data == NULL) {
        DPRINTF(DEBUG_ERROR, "Failed to allocate data buffer");
    }
}
```

```
        SET_ERROR(err_ctx, ENOMEM, "data buffer allocation");
        atomic_add_long(&global_stats.resource_errors, 1);
        goto error;
    }

    /* Initialize object */
    obj->magic = TEST_MAGIC;
    obj->id = id;
    obj->size = size;
    obj->data = data;

    atomic_add_long(&global_stats.successes, 1);
    DPRINTF(DEBUG_INFO, "Successfully allocated object %d", id);

    return (obj);

error:
    if (data != NULL) {
        free(data, M_ERRTEST);
    }
    if (obj != NULL) {
        free(obj, M_ERRTEST);
    }

    atomic_add_long(&global_stats.failures, 1);
    return (NULL);
}

/*
 * Safe object deallocation with validation
 */
static void
test_object_free(struct test_object *obj, struct error_context *err_ctx)
{
    if (obj == NULL) {
        DPRINTF(DEBUG_WARN, "Attempt to free NULL object");
        return;
    }

    /* Validate object */
    if (obj->magic != TEST_MAGIC) {
        DPRINTF(DEBUG_ERROR, "Object has bad magic 0x%x", obj->magic);
        SET_ERROR(err_ctx, EINVAL, "object validation");
        return;
    }
}
```

```

    }

    DPRINTF(DEBUG_VERBOSE, "Freeing object %d", obj->id);

    /* Clear sensitive data */
    if (obj->data != NULL) {
        memset(obj->data, 0, obj->size);
        free(obj->data, M_ERRTEST);
        obj->data = NULL;
    }

    /* Poison object */
    obj->magic = 0xDEADBEEF;
    free(obj, M_ERRTEST);

    DPRINTF(DEBUG_INFO, "Object freed successfully");
}

/*
 * Print error context information
 */
static void
print_error_context(struct error_context *ctx)
{
    if (ctx == NULL || ctx->error_code == 0) {
        return;
    }

    printf("Error Context:\n");
    printf("  Code: %d (%s)\n", ctx->error_code, strerror(ctx->error_code));
    printf("  Operation: %s\n", ctx->operation);
    printf("  Location: %s:%d\n", ctx->file, ctx->line);
    printf("  Timestamp: %ld\n", (long)ctx->timestamp);
}

/*
 * Print operation statistics
 */
static void
print_statistics(void)
{
    u_long attempts, successes, failures, invalid, resource;

    /* Snapshot statistics atomically */

```

```
    attempts = atomic_load_acq_long(&global_stats.total_attempts);
    successes = atomic_load_acq_long(&global_stats.successes);
    failures = atomic_load_acq_long(&global_stats.failures);
    invalid = atomic_load_acq_long(&global_stats.invalid_params);
    resource = atomic_load_acq_long(&global_stats.resource_errors);

    printf("Operation Statistics:\n");
    printf("  Total attempts: %lu\n", attempts);
    printf("  Successes: %lu\n", successes);
    printf("  Failures: %lu\n", failures);
    printf("  Parameter errors: %lu\n", invalid);
    printf("  Resource errors: %lu\n", resource);

    if (attempts > 0) {
        printf("  Success rate: %lu%%\n", (successes * 100) / attempts);
    }
}

static int
error_demo_load(module_t mod, int cmd, void *arg)
{
    struct test_object *obj1, *obj2, *obj3;
    struct error_context err_ctx = { 0 };

    switch (cmd) {
    case MOD_LOAD:
        printf("=== Error Handling and Diagnostics Demo ===\n");

        /* Initialize statistics */
        memset(&global_stats, 0, sizeof(global_stats));

        /* Test successful allocation */
        obj1 = test_object_alloc(1, 1024, &err_ctx);
        if (obj1 != NULL) {
            printf("Successfully allocated object 1\n");
        } else {
            printf("Failed to allocate object 1\n");
            print_error_context(&err_ctx);
        }

        /* Test parameter validation errors */
        memset(&err_ctx, 0, sizeof(err_ctx));
        obj2 = test_object_alloc(-1, 1024, &err_ctx); /* Invalid ID */
        if (obj2 == NULL) {
```

```
        printf("Correctly rejected invalid ID\n");
        print_error_context(&err_ctx);
    }

    memset(&err_ctx, 0, sizeof(err_ctx));
    obj3 = test_object_alloc(3, 0, &err_ctx);    /* Invalid size */
    if (obj3 == NULL) {
        printf("Correctly rejected invalid size\n");
        print_error_context(&err_ctx);
    }

    /* Clean up successful allocation */
    if (obj1 != NULL) {
        test_object_free(obj1, &err_ctx);
    }

    /* Print final statistics */
    print_statistics();

    printf("Error handling demo completed successfully.\n");
    break;

case MOD_UNLOAD:
    printf("Error demo module unloaded.\n");
    break;

default:
    return (EOPNOTSUPP);
}

return (0);
}

static moduledata_t error_demo_mod = {
    "error_demo",
    error_demo_load,
    NULL
};

DECLARE_MODULE(error_demo, error_demo_mod, SI_SUB_DRIVERS, SI_ORDER_MIDDLE);
MODULE_VERSION(error_demo, 1);
```


5.11.17 Summary

Kernel error handling and diagnostic idioms provide structure and consistency to complex system code:

Compiler attributes help catch bugs early and optimize performance **Consistent error codes** make failures predictable and debuggable

Error contexts provide detailed information for problem diagnosis **Debug levels** allow tunable diagnostic output **Statistics tracking** enables performance monitoring and trend analysis **State validation** catches corruption and misuse early

These patterns aren't just good style; they're survival techniques for kernel programming. The combination of defensive coding, comprehensive error handling, and good diagnostics is what separates reliable system software from code that "usually works."

In the next section, we'll put all these concepts together by walking through real FreeBSD kernel code, showing how experienced developers apply these principles in production systems.

5.12 Real-World Kernel Code Walkthrough

Now that we've covered the principles, patterns, and idioms of FreeBSD kernel programming, it's time to see how they all come together in real production code. In this section, we'll walk through several examples from the FreeBSD 14.3 source tree, examining how experienced kernel developers apply the concepts we've learned.

We'll look at code from different subsystems, device drivers, memory management, and network stack, to see how the patterns you've learned are used in practice. This isn't just an academic exercise; understanding real kernel code is essential for becoming an effective FreeBSD developer.

5.12.1 A Simple Character Device Driver: /dev/null

Let's start with one of the simplest yet most essential device drivers in FreeBSD: the null device. It lives in `/usr/src/sys/dev/null/null.c` and provides three devices together: `/dev/null`, `/dev/zero`, and `/dev/full`.

Here's the `cdevsw` definition for `/dev/null`, excerpted from that file:

```
static struct cdevsw null_cdevsw = {
    .d_version =    D_VERSION,
    .d_read =      (d_read_t *)nullop,
    .d_write =     null_write,
    .d_ioctl =     null_ioctl,
    .d_name =      "null",
};
```

And the write handler itself:

```
static int
null_write(struct cdev *dev __unused, struct uio *uio, int flags __unused)
{
    uio->uio_resid = 0;

    return (0);
}
```

Key observations:

1. **Function attributes:** The `__unused` attribute prevents compiler warnings about parameters the function intentionally ignores. `/dev/null` does not look at the `cdev` or the `flags`; only `uio` matters.
2. **Consistent naming:** Functions follow the `subsystem_operation` pattern (`null_write`, `null_ioctl`).
3. **UIO abstraction:** Instead of working directly with user buffers, the driver uses the `uio` structure for safe data transfer. Setting `uio->uio_resid = 0` tells the caller “all bytes consumed,” which is how `/dev/null` pretends to have absorbed the whole write.
4. **Simple semantics:** Writing to `/dev/null` always succeeds (data is discarded); reads use the kernel-supplied `nullop` helper which returns end-of-file immediately.

The driver is registered with the kernel module system. We will study the full registration path in Chapter 6; for now, the important thing is how compact and focused the handler is.

5.12.2 Memory Allocation in Action: `malloc(9)` Implementation

The kernel memory allocator in `/usr/src/sys/kern/kern_malloc.c` is where `malloc(9)` and `free(9)` live. Reading the full implementation requires more vocabulary than we have introduced so far (memguard debugging, KASAN redzones, UMA slab mechanics, branch-prediction hints), but the top-level shape is easy to summarise:

```
/* Simplified sketch of malloc(9). */
void *
my_allocator(size_t size, struct malloc_type *mtp, int flags)
{
    void *va;

    if (size > ZONE_MAX_SIZE) {
        /* Large allocation: bypass the zone cache. */
        va = large_alloc(size, flags);
    } else {
        /* Small allocation: pick a size-bucketed zone. */
        va = zone_alloc(size_to_zone(size), mtp, flags);
    }
    return (va);
}
```

Patterns to recognise:

1. **Dual allocation strategy:** Large allocations bypass the fast per-size-bucket zones.
2. **Resource tracking:** Every successful allocation updates statistics tied to the `malloc_type`. That is what `vmstat -m` show per-subsystem memory use.
3. **Defensive programming:** The real `malloc()` uses `KASSERT()` to sanity-check the `malloc_type` it receives, and `__predict_false()` to tell the compiler which branch is the hot path.
4. **Graceful free(NULL):** The paired `free()` treats a `NULL` pointer as a no-op, so cleanup code can call `free(ptr, type)` unconditionally once `ptr` has been initialised to `NULL`.

Open `kern_malloc.c` and read it when you are comfortable; the patterns above will be easy to spot.

5.12.3 Network Packet Processing: IP Input

The IP input processing code in `/usr/src/sys/netinet/ip_input.c` is a concentrated example of the patterns we have just studied. The real function `ip_input()` is too long to reproduce here in full, but its shape is:

```
void
ip_input(struct mbuf *m)
{
    struct ip *ip;
    int hlen;

    M_ASSERTPKTHDR(m);           /* invariant check */
    IPSTAT_INC(ips_total);       /* stat counter */

    if (m->m_pkthdr.len < sizeof(struct ip))
        goto bad;               /* too short to be an IP header */

    if (m->m_len < sizeof(struct ip) &&
        (m = m_pullup(m, sizeof(struct ip))) == NULL) {
        IPSTAT_INC(ips_toosmall); /* pullup failed; freed the mbuf */
        return;
    }
    ip = mtod(m, struct ip *);

    if (ip->ip_v != IPVERSION) {
        IPSTAT_INC(ips_badvers);
        goto bad;
    }

    hlen = ip->ip_hl << 2;
    if (hlen < sizeof(struct ip)) {
```

```

        IPSTAT_INC(ips_badhlen);
        goto bad;
    }

    /* ... checksum, length checks, forwarding, delivery ... */
    return;

bad:
    m_freem(m);                /* drop and return */
}

```

Patterns to notice:

1. **Assertions:** `M_ASSERTPKTHDR(m)` validates the mbuf structure before anything else touches it.
2. **Statistics tracking:** `IPSTAT_INC()` updates counters so that tools like `netstat -s` can report per-protocol drop reasons.
3. **Early validation:** Every assumption (minimum length, version, header length) is checked before the code operates on it.
4. **Resource management:** `m_pullup()` ensures the IP header is contiguous in memory; if it fails, it has already freed the mbuf, so the driver must not touch it again.
5. **Single cleanup path:** The `bad:` label provides one central place to drop the packet. Every error path converges there.

This is network code in miniature: defend, measure, then do the work.

5.12.4 Device Driver Initialization: PCI Bus Driver

The PCI bus driver in `/usr/src/sys/dev/pci/pci.c` shows how complex hardware drivers handle initialization, resource management, and error recovery. The real `pci_attach()` is short and delegates most of the work to helpers:

```

int
pci_attach(device_t dev)
{
    int busno, domain, error;

    error = pci_attach_common(dev);
    if (error)
        return (error);

    domain = pcib_get_domain(dev);
    busno = pcib_get_bus(dev);
    pci_add_children(dev, domain, busno);
}

```

```
    return (bus_generic_attach(dev));
}
```

Patterns to notice:

1. **Delegation:** `pci_attach_common()` sets up per-instance state (softc, sysctl node, resources). When something new must happen on every PCI bus, it goes into that helper.
2. **Error propagation:** If `pci_attach_common()` returns non-zero, `pci_attach()` returns the same error immediately. Chapter 6 will show how Newbus treats a non-zero return as “this attach failed; roll back.”
3. **Subordinate enumeration:** `pci_add_children()` discovers the devices that sit on this PCI bus; `bus_generic_attach()` asks each one to attach.
4. **Symmetric detach:** The companion `pci_detach()` calls `bus_generic_detach()` first, and only then frees bus-level resources. That is the same reverse-order discipline you’ve been practising throughout this chapter.

We will follow this lifecycle, `probe -> attach -> operate -> detach`, in detail in Chapter 6.

5.12.5 Synchronization in Practice: Reference Counting

The vnode reference counting helpers in `/usr/src/sys/kern/vfs_subr.c` show how small the public API of a well-designed subsystem can be:

```
void
vref(struct vnode *vp)
{
    enum vgetstate vs;

    CTR2(KTR_VFS, "%s: vp %p", __func__, vp);
    vs = vget_prep(vp);
    vget_finish_ref(vp, vs);
}

void
vrele(struct vnode *vp)
{
    ASSERT_VI_UNLOCKED(vp, __func__);
    if (!refcount_release(&vp->v_usecount))
        return;
    vput_final(vp, VRELE);
}
```

Patterns to notice:

1. **Atomic reference counts:** `refcount_release()` decrements the count atomically and returns `true` only when the caller was the last holder. The two-step dance of “atomic decrement, then check for zero” is a standard FreeBSD idiom.
2. **Delegation:** All the interesting work, locking, last-reference teardown, lives in `vget_prep()`, `vget_finish_ref()`, and `vput_final()`. The public functions read like a clear sentence.
3. **Kernel tracing:** `CTR2(KTR_VFS, ...)` produces a low-overhead trace record that can be read back with `ktrdump` or `DTrace`. It is not a `printf()` and it does not appear in `dmesg`.
4. **Assertion strategy:** `ASSERT_VI_UNLOCKED(vp, ...)` documents a precondition: callers must not hold the vnode interlock when they call `vrele()`. If they do, the kernel will catch it immediately in a debug build.

We will come back to reference counting and the `refcount(9)` API when we look at driver lifetimes in later chapters.

5.12.6 What We’ve Learned from Real Code

Examining these real-world examples reveals several important patterns:

Defensive programming is everywhere: Every function validates its inputs and assumptions.

Error handling is systematic: Errors are caught early, propagated consistently, and resources are cleaned up properly.

Performance matters: Code uses branch prediction hints, atomic operations, and optimized data structures.

Debugging is built-in: Statistics, tracing, and assertions are integral parts of the code.

Patterns repeat: The same idioms appear across different subsystems, consistent error codes, resource management patterns, and synchronization techniques.

Simplicity wins: Even complex subsystems are built from simple, well-understood components.

These aren’t just academic examples; this is production code that handles millions of operations per second on systems around the world. The patterns we’ve studied aren’t theoretical; they’re battle-tested techniques that keep FreeBSD stable and performant.

5.12.7 Summary

Real FreeBSD kernel code demonstrates how all the concepts we’ve covered work together:

- Kernel-specific data types provide portability and clarity
- Defensive programming prevents subtle bugs
- Consistent error handling makes systems reliable
- Proper resource management prevents leaks and corruption
- Synchronization primitives enable safe multiprocessor operation
- Coding idioms make code readable and maintainable

The gap between learning these concepts and applying them in real code is smaller than you might think. FreeBSD’s consistent patterns and excellent documentation make it possible for new developers to contribute meaningfully to this mature, complex system.

In the next section, we’ll put your knowledge to the test with hands-on labs that let you write and experiment with your own kernel code.

5.13 Hands-On Labs (Beginner Kernel C)

It’s time to put everything you’ve learned into practice. These hands-on labs will guide you through writing, compiling, loading, and testing real FreeBSD kernel modules that demonstrate the key differences between user-space and kernel-space C programming.

Each lab focuses on a specific aspect of the kernel C “dialect” you’ve been learning. You’ll see firsthand how kernel code handles memory, communicates with user space, manages resources, and handles errors differently from ordinary C programs.

These aren’t academic exercises; you’ll be writing actual kernel code that runs in your FreeBSD system. By the end of this section, you’ll have concrete experience with the kernel programming patterns that every FreeBSD developer relies on.

5.13.1 Lab Prerequisites

Before starting the labs, ensure your FreeBSD system is properly set up:

- FreeBSD 14.3 with kernel source in `/usr/src`
- Development tools installed (base-devel package)
- A safe lab environment (virtual machine recommended)
- Basic familiarity with the FreeBSD command line

Safety reminder: These labs involve loading code into the kernel. While the exercises are designed to be safe, always work in a lab environment where kernel panics won’t affect important data.

5.13.2 Lab 1: Safe Memory Allocation and Cleanup

The first lab demonstrates one of the most critical differences between user-space and kernel-space programming: memory management. In user space, you might call `malloc()` and occasionally forget to call `free()`. In kernel space, every allocation must be perfectly balanced with deallocation, or you’ll create memory leaks that can crash the system.

Objective: Write a tiny kernel module that allocates and frees memory safely, demonstrating proper resource management patterns.

Create your lab directory:

```
% mkdir ~/kernel_labs
% cd ~/kernel_labs
% mkdir lab1 && cd lab1
```

Create `memory_safe.c`:

```

/*
 * memory_safe.c - Safe kernel memory management demonstration
 *
 * This module demonstrates the kernel C dialect of memory management:
 * - malloc(9) with proper type definitions
 * - M_WAITOK vs M_NOWAIT allocation strategies
 * - Mandatory cleanup on module unload
 * - Memory debugging and tracking
 */

#include <sys/param.h>
#include <sys/kernel.h>
#include <sys/module.h>
#include <sys/systm.h>
#include <sys/malloc.h>    /* Kernel memory allocation */

/*
 * Define a memory type for debugging and statistics.
 * This is how kernel C tracks different kinds of allocations.
 */
MALLOC_DEFINE(M_MEMLAB, "memory_lab", "Memory Lab Example Allocations");

/* Module state - global variables are acceptable in kernel modules */
static void *test_buffer = NULL;
static size_t buffer_size = 1024;

/*
 * safe_allocate - Demonstrate defensive memory allocation
 *
 * This shows the kernel C pattern for memory allocation:
 * 1. Validate parameters
 * 2. Use appropriate malloc flags
 * 3. Check for allocation failure
 * 4. Initialize allocated memory
 */
static int
safe_allocate(size_t size)
{
    /* Input validation - essential in kernel code */
    if (size == 0 || size > (1024 * 1024)) {
        printf("Memory Lab: Invalid size %zu (must be 1-%d bytes)\n",
            size, 1024 * 1024);
        return (EINVAL);
    }
}

```



```

    }

    if (test_buffer != NULL) {
        printf("Memory Lab: Memory already allocated\n");
        return (EBUSY);
    }

    /*
     * Kernel allocation with M_WAITOK - can sleep if needed
     * M_ZERO initializes the memory to zero (safer than malloc + memset)
     */
    test_buffer = malloc(size, M_MEMLAB, M_WAITOK | M_ZERO);
    if (test_buffer == NULL) {
        printf("Memory Lab: Allocation failed for %zu bytes\n", size);
        return (ENOMEM);
    }

    buffer_size = size;
    printf("Memory Lab: Successfully allocated %zu bytes at %p\n",
        size, test_buffer);

    /* Test the allocation by writing known data */
    snprintf((char *)test_buffer, size, "Allocated at ticks=%d", ticks);
    printf("Memory Lab: Test data: '%s'\n", (char *)test_buffer);

    return (0);
}

/*
 * safe_deallocate - Clean up allocated memory
 *
 * The kernel C rule: every malloc must have a matching free,
 * especially during module unload.
 */
static void
safe_deallocate(void)
{
    if (test_buffer != NULL) {
        printf("Memory Lab: Freeing %zu bytes at %p\n", buffer_size, test_buffer);

        /* Clear sensitive data before freeing (good practice) */
        explicit_bzero(test_buffer, buffer_size);

        /* Free using the same memory type used for allocation */

```

```
    free(test_buffer, M_MEMLAB);
    test_buffer = NULL;
    buffer_size = 0;

    printf("Memory Lab: Memory safely deallocated\n");
}
}

/*
 * Module event handler
 */
static int
memory_safe_handler(module_t mod, int what, void *arg)
{
    int error = 0;

    switch (what) {
    case MOD_LOAD:
        printf("Memory Lab: Module loading\n");

        /* Demonstrate safe allocation */
        error = safe_allocate(1024);
        if (error != 0) {
            printf("Memory Lab: Failed to allocate memory: %d\n", error);
            return (error);
        }

        printf("Memory Lab: Module loaded successfully\n");
        break;

    case MOD_UNLOAD:
        printf("Memory Lab: Module unloading\n");

        /* CRITICAL: Always clean up on unload */
        safe_deallocate();

        printf("Memory Lab: Module unloaded safely\n");
        break;

    default:
        error = EOPNOTSUPP;
        break;
    }
}
```

```
    return (error);
}

/* Module declaration */
static moduledata_t memory_safe_mod = {
    "memory_safe",
    memory_safe_handler,
    NULL
};

DECLARE_MODULE(memory_safe, memory_safe_mod, SI_SUB_DRIVERS, SI_ORDER_MIDDLE);
MODULE_VERSION(memory_safe, 1);
```

Create Makefile:

```
# Makefile for memory_safe module
KMOD=     memory_safe
SRCS=     memory_safe.c

.include <bsd.kmod.mk>
```

Build and test the module:

```
% make clean && make

# Load the module
% sudo kldload ./memory_safe.ko

# Check that it loaded and allocated memory
% dmesg | tail -5

# Check kernel memory statistics
% vmstat -m | grep memory_lab

# Unload the module
% sudo kldunload memory_safe

# Verify clean unload
% dmesg | tail -3
```

Expected output:

```
Memory Lab: Module loading
Memory Lab: Successfully allocated 1024 bytes at 0xfffff8000c123000
Memory Lab: Test data: 'Allocated at ticks=12345'
Memory Lab: Module loaded successfully
Memory Lab: Module unloading
Memory Lab: Freeing 1024 bytes at 0xfffff8000c123000
Memory Lab: Memory safely deallocated
Memory Lab: Module unloaded safely
```

Key Learning Points:

- Kernel C requires explicit memory type definitions (MALLOC_DEFINE)
- Every `malloc()` must be paired with exactly one `free()`
- Module unload handlers must clean up ALL allocated resources
- Input validation is critical in kernel code

5.13.3 Lab 2: User-Kernel Data Exchange

The second lab explores how kernel C handles data exchange with user space. Unlike user-space C where you can freely pass pointers between functions, kernel code must use special functions like `copyin()` and `copyout()` to safely transfer data across the user-kernel boundary.

Objective: Create a kernel module that echoes data between user and kernel space using proper boundary-crossing techniques.

Create your lab directory:

```
% cd ~/kernel_labs
% mkdir lab2 && cd lab2
```

Create `echo_safe.c`:

```
/*
 * echo_safe.c - Safe user-kernel data exchange demonstration
 *
 * This module demonstrates the kernel C dialect for crossing the
 * user-kernel boundary safely:
 * - copyin() for user-to-kernel data transfer
 * - copyout() for kernel-to-user data transfer
 * - Character device interface for testing
 * - Input validation and buffer management
 */

#include <sys/param.h>
#include <sys/kernel.h>
```

```
#include <sys/module.h>
#include <sys/system.h>
#include <sys/malloc.h>
#include <sys/conf.h>      /* Character device support */
#include <sys/uio.h>       /* User I/O operations */

#define BUFFER_SIZE 256

MALLOC_DEFINE(M_ECHOLAB, "echo_lab", "Echo Lab Allocations");

/* Module state */
static struct cdev *echo_device;
static char *kernel_buffer;

/*
 * Device write operation - demonstrates copyin() equivalent (uiomove)
 *
 * When user space writes to our device, this function receives the data
 * using the kernel-safe uiomove() function.
 */
static int
echo_write(struct cdev *dev, struct uio *uio, int flag)
{
    size_t bytes_to_copy;
    int error;

    printf("Echo Lab: Write request for %d bytes\n", (int)uio->uio_resid);

    if (kernel_buffer == NULL) {
        printf("Echo Lab: Kernel buffer not allocated\n");
        return (ENXIO);
    }

    /* Limit copy size to buffer capacity minus null terminator */
    bytes_to_copy = MIN(uio->uio_resid, BUFFER_SIZE - 1);

    /* Clear the buffer first */
    memset(kernel_buffer, 0, BUFFER_SIZE);

    /*
     * uiomove() is the kernel C way to safely copy data from user space.
     * It handles all the validation and protection boundary crossing.
     */
    error = uiomove(kernel_buffer, bytes_to_copy, uio);
}
```

```
    if (error != 0) {
        printf("Echo Lab: uiomove from user failed: %d\n", error);
        return (error);
    }

    /* Ensure null termination for safety */
    kernel_buffer[bytes_to_copy] = '\0';

    printf("Echo Lab: Received from user: '%s' (%zu bytes)\n",
        kernel_buffer, bytes_to_copy);

    return (0);
}

/*
 * Device read operation - demonstrates copyout() equivalent (uiomove)
 *
 * When user space reads from our device, this function sends the data
 * back using the kernel-safe uiomove() function.
 */
static int
echo_read(struct cdev *dev, struct uio *uio, int flag)
{
    char response[BUFFER_SIZE + 64]; /* Buffer for response with prefix */
    size_t response_len;
    int error;

    if (kernel_buffer == NULL) {
        return (ENXIO);
    }

    /* Create echo response with metadata */
    snprintf(response, sizeof(response),
        "Echo: '%s' (received %zu bytes at ticks %d)\n",
        kernel_buffer,
        strlen(kernel_buffer, BUFFER_SIZE),
        ticks);

    response_len = strlen(response);

    /* Handle file offset for proper read semantics */
    if (uio->uio_offset >= response_len) {
        return (0); /* EOF */
    }
}
```

```
/* Adjust read size based on offset and request */
if (uio->uio_offset + uio->uio_resid > response_len) {
    response_len -= uio->uio_offset;
} else {
    response_len = uio->uio_resid;
}

printf("Echo Lab: Read request, sending %zu bytes\n", response_len);

/*
 * uiomove() also handles kernel-to-user transfers safely.
 * This is the kernel C equivalent of copyout().
 */
error = uiomove(response + uio->uio_offset, response_len, uio);
if (error != 0) {
    printf("Echo Lab: uiomove to user failed: %d\n", error);
}

return (error);
}

/* Character device operations structure */
static struct cdevsw echo_cdevsw = {
    .d_version = D_VERSION,
    .d_read = echo_read,
    .d_write = echo_write,
    .d_name = "echolab"
};

/*
 * Module event handler
 */
static int
echo_safe_handler(module_t mod, int what, void *arg)
{
    int error = 0;

    switch (what) {
    case MOD_LOAD:
        printf("Echo Lab: Module loading\n");

        /* Allocate kernel buffer for storing echoed data */
        kernel_buffer = malloc(BUFFER_SIZE, M_ECHOLAB, M_WAITOK | M_ZERO);
    }
```

```
if (kernel_buffer == NULL) {
    printf("Echo Lab: Failed to allocate kernel buffer\n");
    return (ENOMEM);
}

/* Create character device for user interaction */
echo_device = make_dev(&echo_cdevsw, 0, UID_ROOT, GID_WHEEL,
                      0666, "echolab");
if (echo_device == NULL) {
    printf("Echo Lab: Failed to create character device\n");
    free(kernel_buffer, M_ECHOLAB);
    kernel_buffer = NULL;
    return (ENXIO);
}

printf("Echo Lab: Device /dev/echolab created\n");
printf("Echo Lab: Test with: echo 'Hello' > /dev/echolab\n");
printf("Echo Lab: Read with: cat /dev/echolab\n");
break;

case MOD_UNLOAD:
    printf("Echo Lab: Module unloading\n");

    /* Clean up device */
    if (echo_device != NULL) {
        destroy_dev(echo_device);
        echo_device = NULL;
        printf("Echo Lab: Character device destroyed\n");
    }

    /* Clean up buffer */
    if (kernel_buffer != NULL) {
        free(kernel_buffer, M_ECHOLAB);
        kernel_buffer = NULL;
        printf("Echo Lab: Kernel buffer freed\n");
    }

    printf("Echo Lab: Module unloaded successfully\n");
    break;

default:
    error = EOPNOTSUPP;
    break;
}
```



```
    return (error);
}

static moduledata_t echo_safe_mod = {
    "echo_safe",
    echo_safe_handler,
    NULL
};

DECLARE_MODULE(echo_safe, echo_safe_mod, SI_SUB_DRIVERS, SI_ORDER_MIDDLE);
MODULE_VERSION(echo_safe, 1);
```

Create Makefile:

```
KMOD=     echo_safe
SRCS=     echo_safe.c

.include <bsd.kmod.mk>
```

Build and test the module:

```
% make clean && make

# Load the module
% sudo kldload ./echo_safe.ko

# Test the echo functionality
% echo "Hello from user space!" | sudo tee /dev/echolab

# Read the echo response
% cat /dev/echolab

# Test with different data
% echo "Testing 123" | sudo tee /dev/echolab
% cat /dev/echolab

# Unload the module
% sudo kldunload echo_safe
```

Expected output:

```
Echo Lab: Module loading
Echo Lab: Device /dev/echolab created
Echo Lab: Write request for 24 bytes
Echo Lab: Received from user: 'Hello from user space!' (23 bytes)
Echo Lab: Read request, sending 56 bytes
Echo: 'Hello from user space!' (received 23 bytes at ticks 45678)
```

Key Learning Points:

- Kernel C cannot directly access user space pointers
- `uiomove()` safely transfers data across the user-kernel boundary
- Always validate buffer sizes and handle partial transfers
- Character devices provide a clean interface for user-kernel communication

5.13.4 Lab 3: Driver-Safe Logging and Device Context

The third lab demonstrates how kernel C handles logging and device context differently from user-space `printf()`. In kernel code, especially device drivers, you need to be careful about which `printf()` variant you use and when it's safe to call them.

Objective: Create a kernel module that demonstrates the difference between `printf()` and `device_printf()`, showing driver-safe logging practices.

Create your lab directory:

```
% cd ~/kernel_labs
% mkdir lab3 && cd lab3
```

Create `logging_safe.c`:

```
/*
 * logging_safe.c - Safe kernel logging demonstration
 *
 * This module demonstrates the kernel C dialect for logging:
 * - printf() for general kernel messages
 * - device_printf() for device-specific messages
 * - uprntf() for messages to specific users
 * - Log level awareness and timing considerations
 */

#include <sys/param.h>
#include <sys/kernel.h>
#include <sys/module.h>
#include <sys/systm.h>
#include <sys/malloc.h>
```

```
#include <sys/conf.h>
#include <sys/bus.h>          /* For device context */

MALLOC_DEFINE(M_LOGLAB, "log_lab", "Logging Lab Allocations");

/* Simulated device state */
struct log_lab_softc {
    device_t dev;              /* Device reference for device_printf */
    char device_name[32];
    int message_count;
    int error_count;
};

static struct log_lab_softc *lab_softc = NULL;

/*
 * demonstrate_printf_variants - Show different kernel logging functions
 *
 * This function demonstrates when to use each type of kernel logging
 * function and what information each provides.
 */
static void
demonstrate_printf_variants(struct log_lab_softc *sc)
{
    /*
     * printf() - General kernel logging
     * - Goes to kernel message buffer (dmesg)
     * - No specific device association
     * - Safe to call from most kernel contexts
     */
    printf("Log Lab: General kernel message (printf)\n");

    /*
     * In a real device driver with actual device_t, you would use:
     * device_printf(sc->dev, "Device-specific message\n");
     *
     * Since we're simulating, we'll show the pattern:
     */
    printf("Log Lab: [%s] Simulated device_printf message\n", sc->device_name);
    printf("Log Lab: [%s] Device message count: %d\n",
        sc->device_name, ++sc->message_count);

    /*
     * Log with different levels of information
     */
}
```

```

    */
    printf("Log Lab: INFO - Normal operation message\n");
    printf("Log Lab: WARNING - Something unusual happened\n");
    printf("Log Lab: ERROR - Operation failed, count=%d\n", ++sc->error_count);

    /*
     * Demonstrate structured logging with context
     */
    printf("Log Lab: [%s] status: messages=%d errors=%d ticks=%d\n",
           sc->device_name, sc->message_count, sc->error_count, ticks);
}

/*
 * demonstrate_logging_safety - Show safe logging practices
 *
 * This demonstrates important safety considerations for kernel logging:
 * - Avoid logging in interrupt context when possible
 * - Limit message frequency to avoid spam
 * - Include relevant context in messages
 */
static void
demonstrate_logging_safety(struct log_lab_softc *sc)
{
    static int call_count = 0;

    call_count++;

    /*
     * Rate limiting example - avoid spamming the log
     */
    if (call_count <= 5 || (call_count % 100) == 0) {
        printf("Log Lab: [%s] Safety demo call #%d\n",
               sc->device_name, call_count);
    }

    /*
     * Context-rich logging - include relevant state information
     */
    if (sc->error_count > 3) {
        printf("Log Lab: [%s] ERROR threshold exceeded: %d errors\n",
               sc->device_name, sc->error_count);
    }

    /*

```

```
    * Demonstrate debugging vs. operational messages
    */
#ifdef DEBUG
    printf("Log Lab: [%s] DEBUG - Internal state check passed\n",
        sc->device_name);
#endif

    /* Operational message that users care about */
    if ((call_count % 10) == 0) {
        printf("Log Lab: [%s] Operational status: %d operations completed\n",
            sc->device_name, call_count);
    }
}

/*
* lab_timer_callback - Demonstrate logging in timer context
*
* This shows how to log safely from timer callbacks and other
* asynchronous contexts.
*/
static void
lab_timer_callback(void *arg)
{
    struct log_lab_softc *sc = (struct log_lab_softc *)arg;

    if (sc != NULL) {
        /*
        * Timer context logging - keep it brief and informative
        */
        printf("Log Lab: [%s] Timer tick - uptime checks\n", sc->device_name);

        demonstrate_printf_variants(sc);
        demonstrate_logging_safety(sc);
    }
}

/* Timer handle for periodic logging demonstration */
static struct callout lab_timer;

/*
* Module event handler
*/
static int
logging_safe_handler(module_t mod, int what, void *arg)
```

```

{
    int error = 0;

    switch (what) {
    case MOD_LOAD:
        /*
         * Module loading - demonstrate initial logging
         */
        printf("Log Lab: =====\n");
        printf("Log Lab: Module loading - demonstrating kernel logging\n");
        printf("Log Lab: Build time: " __DATE__ " " __TIME__ "\n");

        /* Allocate softc structure */
        lab_softc = malloc(sizeof(struct log_lab_softc), M_LOGLAB,
                           M_WAITOK | M_ZERO);
        if (lab_softc == NULL) {
            printf("Log Lab: ERROR - Failed to allocate softc\n");
            return (ENOMEM);
        }

        /* Initialize softc */
        strncpy(lab_softc->device_name, "loglab0",
                sizeof(lab_softc->device_name));
        lab_softc->message_count = 0;
        lab_softc->error_count = 0;

        printf("Log Lab: [%s] Device context initialized\n",
               lab_softc->device_name);

        /* Demonstrate immediate logging */
        demonstrate_printf_variants(lab_softc);

        /* Set up periodic timer for ongoing demonstrations */
        callout_init(&lab_timer, 0);
        callout_reset(&lab_timer, hz * 5, /* 5 second intervals */
                      lab_timer_callback, lab_softc);

        printf("Log Lab: [%s] Module loaded, timer started\n",
               lab_softc->device_name);
        printf("Log Lab: Watch 'dmesg' for periodic log messages\n");
        printf("Log Lab: =====\n");
        break;

    case MOD_UNLOAD:

```

```

    printf("Log Lab: =====\n");
    printf("Log Lab: Module unloading\n");

    /* Stop timer first */
    if (callout_active(&lab_timer)) {
        callout_drain(&lab_timer);
        printf("Log Lab: Timer stopped and drained\n");
    }

    /* Clean up softc */
    if (lab_softc != NULL) {
        printf("Log Lab: [%s] Final stats: messages=%d errors=%d\n",
            lab_softc->device_name,
            lab_softc->message_count,
            lab_softc->error_count);

        free(lab_softc, M_LOGLAB);
        lab_softc = NULL;
        printf("Log Lab: Device context freed\n");
    }

    printf("Log Lab: Module unloaded successfully\n");
    printf("Log Lab: =====\n");
    break;

default:
    printf("Log Lab: Unsupported module operation: %d\n", what);
    error = EOPNOTSUPP;
    break;
}

return (error);
}

static moduledata_t logging_safe_mod = {
    "logging_safe",
    logging_safe_handler,
    NULL
};

DECLARE_MODULE(logging_safe, logging_safe_mod, SI_SUB_DRIVERS, SI_ORDER_MIDDLE);
MODULE_VERSION(logging_safe, 1);

```

Create Makefile:

```
KMOD=    logging_safe
SRCS=    logging_safe.c

.include <bsd.kmod.mk>
```

Build and test the module:

```
% make clean && make

# Load the module and observe initial messages
% sudo kldload ./logging_safe.ko
% dmesg | tail -10

# Wait a few seconds and check for timer messages
% sleep 10
% dmesg | tail -15

# Check ongoing activity
% dmesg | grep "Log Lab" | tail -5

# Unload and observe cleanup messages
% sudo kldunload logging_safe
% dmesg | tail -10
```

Expected output:

```
Log Lab: =====
Log Lab: Module loading - demonstrating kernel logging
Log Lab: Build time: Sep 30 2025 12:34:56
Log Lab: [loglab0] Device context initialized
Log Lab: General kernel message (printf)
Log Lab: [loglab0] Simulated device_printf message
Log Lab: [loglab0] Device message count: 1
Log Lab: [loglab0] Timer tick - uptime checks
Log Lab: [loglab0] Final stats: messages=5 errors=1
Log Lab: =====
```

Key Learning Points:

- Different printf() variants serve different purposes in kernel code
- Device context provides better diagnostics than generic messages
- Timer callbacks require careful consideration of logging frequency
- Structured logging with context makes debugging much easier

5.13.5 Lab 4: Error Handling and Graceful Failures

The fourth lab focuses on one of the most critical aspects of kernel C: proper error handling. Unlike user-space programs that can often crash gracefully, kernel code must handle every possible error condition without bringing down the entire system.

Objective: Create a kernel module that introduces controlled errors (like returning ENOMEM) to practice comprehensive error handling patterns.

Create your lab directory:

```
% cd ~/kernel_labs
% mkdir lab4 && cd lab4
```

Create `error_handling.c`:

```
/*
 * error_handling.c - Comprehensive error handling demonstration
 *
 * This module demonstrates the kernel C dialect for error handling:
 * - Proper error code usage (errno.h constants)
 * - Resource cleanup on error paths
 * - Graceful degradation strategies
 * - Error injection for testing robustness
 */

#include <sys/param.h>
#include <sys/kernel.h>
#include <sys/module.h>
#include <sys/system.h>
#include <sys/malloc.h>
#include <sys/conf.h>
#include <sys/uio.h>
#include <sys/errno.h>      /* Standard error codes */

#define MAX_BUFFERS 5
#define BUFFER_SIZE 1024

MALLOC_DEFINE(M_ERRORLAB, "error_lab", "Error Handling Lab");

/* Module state for tracking resources */
struct error_lab_state {
    void *buffers[MAX_BUFFERS];    /* Array of allocated buffers */
    int buffer_count;              /* Number of active buffers */
    int error_injection_enabled;    /* For testing error paths */
    int operation_count;           /* Total operations attempted */
}
```

```
int success_count;           /* Successful operations */
int error_count;             /* Failed operations */
};

static struct error_lab_state *lab_state = NULL;
static struct cdev *error_device = NULL;

/*
 * cleanup_all_resources - Complete resource cleanup
 *
 * This function demonstrates the kernel C pattern for complete
 * resource cleanup, especially important on error paths.
 */
static void
cleanup_all_resources(struct error_lab_state *state)
{
    int i;

    if (state == NULL) {
        return;
    }

    printf("Error Lab: Beginning resource cleanup\n");

    /* Free all allocated buffers */
    for (i = 0; i < MAX_BUFFERS; i++) {
        if (state->buffers[i] != NULL) {
            printf("Error Lab: Freeing buffer %d at %p\n",
                i, state->buffers[i]);
            free(state->buffers[i], M_ERRORLAB);
            state->buffers[i] = NULL;
        }
    }

    state->buffer_count = 0;
    printf("Error Lab: All %d buffers freed\n", MAX_BUFFERS);
}

/*
 * allocate_buffer_safe - Demonstrate defensive allocation
 *
 * This function shows how to handle allocation errors gracefully
 * and maintain consistent state even when operations fail.
 */
```

```
static int
allocate_buffer_safe(struct error_lab_state *state)
{
    void *new_buffer;
    int slot;

    /* Input validation */
    if (state == NULL) {
        printf("Error Lab: Invalid state pointer\n");
        return (EINVAL);
    }

    state->operation_count++;

    /* Check resource limits */
    if (state->buffer_count >= MAX_BUFFERS) {
        printf("Error Lab: Maximum buffers (%d) already allocated\n",
            MAX_BUFFERS);
        state->error_count++;
        return (ENOSPC);
    }

    /* Find empty slot */
    for (slot = 0; slot < MAX_BUFFERS; slot++) {
        if (state->buffers[slot] == NULL) {
            break;
        }
    }

    if (slot >= MAX_BUFFERS) {
        printf("Error Lab: No available buffer slots\n");
        state->error_count++;
        return (ENOSPC);
    }

    /* Simulate error injection for testing */
    if (state->error_injection_enabled) {
        printf("Error Lab: Simulating allocation failure (error injection)\n");
        state->error_count++;
        return (ENOMEM);
    }

    /*
     * Attempt allocation with M_NOWAIT to allow controlled failure
     */
}
```

```
    * In production code, choice of M_WAITOK vs M_NOWAIT depends on context
    */
new_buffer = malloc(BUFFER_SIZE, M_ERRORLAB, M_NOWAIT | M_ZERO);
if (new_buffer == NULL) {
    printf("Error Lab: Real allocation failure for %d bytes\n", BUFFER_SIZE);
    state->error_count++;
    return (ENOMEM);
}

/* Successfully allocated - update state */
state->buffers[slot] = new_buffer;
state->buffer_count++;
state->success_count++;

printf("Error Lab: Allocated buffer %d at %p (%d/%d total)\n",
       slot, new_buffer, state->buffer_count, MAX_BUFFERS);

return (0);
}

/*
* free_buffer_safe - Demonstrate safe deallocation
*/
static int
free_buffer_safe(struct error_lab_state *state, int slot)
{
    /* Input validation */
    if (state == NULL) {
        return (EINVAL);
    }

    if (slot < 0 || slot >= MAX_BUFFERS) {
        printf("Error Lab: Invalid buffer slot %d (must be 0-%d)\n",
               slot, MAX_BUFFERS - 1);
        return (EINVAL);
    }

    if (state->buffers[slot] == NULL) {
        printf("Error Lab: Buffer slot %d is already free\n", slot);
        return (ENOENT);
    }

    /* Free the buffer */
    printf("Error Lab: Freeing buffer %d at %p\n", slot, state->buffers[slot]);
}
```

```
    free(state->buffers[slot], M_ERRORLAB);
    state->buffers[slot] = NULL;
    state->buffer_count--;

    return (0);
}

/*
 * Device write handler - Command interface for testing error handling
 */
static int
error_write(struct cdev *dev, struct uio *uio, int flag)
{
    char command[64];
    size_t len;
    int error = 0;
    int slot;

    if (lab_state == NULL) {
        return (EIO);
    }

    /* Read command from user */
    len = MIN(uio->uio_resid, sizeof(command) - 1);
    error = uiomove(command, len, uio);
    if (error) {
        printf("Error Lab: Failed to read command: %d\n", error);
        return (error);
    }

    command[len] = '\0';

    /* Remove trailing newline */
    if (len > 0 && command[len - 1] == '\n') {
        command[len - 1] = '\0';
    }

    printf("Error Lab: Processing command: '%s'\n", command);

    /* Command processing with comprehensive error handling */
    if (strcmp(command, "alloc") == 0) {
        error = allocate_buffer_safe(lab_state);
        if (error) {
            printf("Error Lab: Allocation failed: %s (%d)\n",
```

```

        (error == ENOMEM) ? "Out of memory" :
        (error == ENOSPC) ? "No space available" : "Unknown error",
        error);
    }
} else if (strncmp(command, "free ", 5) == 0) {
    slot = strtol(command + 5, NULL, 10);
    error = free_buffer_safe(lab_state, slot);
    if (error) {
        printf("Error Lab: Free failed: %s (%d)\n",
            (error == EINVAL) ? "Invalid slot" :
            (error == ENOENT) ? "Slot already free" : "Unknown error",
            error);
    }
} else if (strcmp(command, "error_on") == 0) {
    lab_state->error_injection_enabled = 1;
    printf("Error Lab: Error injection ENABLED\n");
} else if (strcmp(command, "error_off") == 0) {
    lab_state->error_injection_enabled = 0;
    printf("Error Lab: Error injection DISABLED\n");
} else if (strcmp(command, "status") == 0) {
    printf("Error Lab: Status Report:\n");
    printf("  Buffers: %d/%d allocated\n",
        lab_state->buffer_count, MAX_BUFFERS);
    printf("  Operations: %d total, %d successful, %d failed\n",
        lab_state->operation_count, lab_state->success_count,
        lab_state->error_count);
    printf("  Error injection: %s\n",
        lab_state->error_injection_enabled ? "enabled" : "disabled");
} else if (strcmp(command, "cleanup") == 0) {
    cleanup_all_resources(lab_state);
    printf("Error Lab: Manual cleanup completed\n");
} else {
    printf("Error Lab: Unknown command '%s'\n", command);
    printf("Error Lab: Valid commands: alloc, free <n>, error_on, error_off,
↪ status, cleanup\n");
    error = EINVAL;
}

return (error);
}

/*
 * Device read handler - Status reporting
 */

```

```

static int
error_read(struct cdev *dev, struct uio *uio, int flag)
{
    char status[512];
    size_t len;
    int i;

    if (lab_state == NULL) {
        return (EIO);
    }

    /* Build comprehensive status report */
    len = snprintf(status, sizeof(status),
        "Error Handling Lab Status:\n"
        "=====\n"
        "Buffers: %d/%d allocated\n"
        "Operations: %d total (%d successful, %d failed)\n"
        "Error injection: %s\n"
        "Success rate: %d%%\n"
        "\nBuffer allocation map:\n",
        lab_state->buffer_count, MAX_BUFFERS,
        lab_state->operation_count, lab_state->success_count, lab_state->error_count,
        lab_state->error_injection_enabled ? "ENABLED" : "disabled",
        (lab_state->operation_count > 0) ?
            (lab_state->success_count * 100 / lab_state->operation_count) : 0);

    /* Add buffer map */
    for (i = 0; i < MAX_BUFFERS; i++) {
        len += snprintf(status + len, sizeof(status) - len,
            " Slot %d: %s\n", i,
            lab_state->buffers[i] ? "ALLOCATED" : "free");
    }

    len += snprintf(status + len, sizeof(status) - len,
        "\nCommands: alloc, free <n>, error_on, error_off, status,
        ↪ cleanup\n");

    /* Handle read with offset */
    if (uio->uio_offset >= len) {
        return (0);
    }

    return (uiomove(status + uio->uio_offset,
        MIN(len - uio->uio_offset, uio->uio_resid), uio));
}

```

```
}

/* Character device operations */
static struct cdevsw error_cdevsw = {
    .d_version = D_VERSION,
    .d_read = error_read,
    .d_write = error_write,
    .d_name = "errorlab"
};

/*
 * Module event handler with comprehensive error handling
 */
static int
error_handling_handler(module_t mod, int what, void *arg)
{
    int error = 0;

    switch (what) {
    case MOD_LOAD:
        printf("Error Lab: =====\n");
        printf("Error Lab: Module loading with error handling demo\n");

        /* Allocate main state structure */
        lab_state = malloc(sizeof(struct error_lab_state), M_ERRORLAB,
                           M_WAITOK | M_ZERO);
        if (lab_state == NULL) {
            printf("Error Lab: CRITICAL - Failed to allocate state structure\n");
            return (ENOMEM);
        }

        /* Initialize state */
        lab_state->buffer_count = 0;
        lab_state->error_injection_enabled = 0;
        lab_state->operation_count = 0;
        lab_state->success_count = 0;
        lab_state->error_count = 0;

        /* Create device with error handling */
        error_device = make_dev(&error_cdevsw, 0, UID_ROOT, GID_WHEEL,
                               0666, "errorlab");
        if (error_device == NULL) {
            printf("Error Lab: Failed to create device\n");
            free(lab_state, M_ERRORLAB);
        }
    }
}
```



```
        lab_state = NULL;
        return (ENXIO);
    }

    printf("Error Lab: Module loaded successfully\n");
    printf("Error Lab: Device /dev/errorlab created\n");
    printf("Error Lab: Try: echo 'alloc' > /dev/errorlab\n");
    printf("Error Lab: Status: cat /dev/errorlab\n");
    printf("Error Lab: =====\n");
    break;

case MOD_UNLOAD:
    printf("Error Lab: =====\n");
    printf("Error Lab: Module unloading\n");

    /* Clean up device */
    if (error_device != NULL) {
        destroy_dev(error_device);
        error_device = NULL;
        printf("Error Lab: Device destroyed\n");
    }

    /* Clean up all resources */
    if (lab_state != NULL) {
        printf("Error Lab: Final statistics:\n");
        printf("  Operations: %d total, %d successful, %d failed\n",
            lab_state->operation_count, lab_state->success_count,
            lab_state->error_count);

        cleanup_all_resources(lab_state);
        free(lab_state, M_ERRORLAB);
        lab_state = NULL;
        printf("Error Lab: State structure freed\n");
    }

    printf("Error Lab: Module unloaded successfully\n");
    printf("Error Lab: =====\n");
    break;

default:
    printf("Error Lab: Unsupported module operation: %d\n", what);
    error = EOPNOTSUPP;
    break;
}
```

```
    return (error);
}

static moduledata_t error_handling_mod = {
    "error_handling",
    error_handling_handler,
    NULL
};

DECLARE_MODULE(error_handling, error_handling_mod, SI_SUB_DRIVERS, SI_ORDER_MIDDLE);
MODULE_VERSION(error_handling, 1);
```

Create Makefile:

```
KMOD=    error_handling
SRCS=    error_handling.c

.include <bsd.kmod.mk>
```

Build and test the module:

```
% make clean && make

# Load the module
% sudo kldload ./error_handling.ko

# Check initial status
% cat /dev/errorlab

# Test normal allocation
% echo "alloc" | sudo tee /dev/errorlab
% echo "alloc" | sudo tee /dev/errorlab
% cat /dev/errorlab

# Test error injection
% echo "error_on" | sudo tee /dev/errorlab
% echo "alloc" | sudo tee /dev/errorlab # Should fail

# Turn off error injection and try again
% echo "error_off" | sudo tee /dev/errorlab
% echo "alloc" | sudo tee /dev/errorlab # Should succeed
```

```
# Test freeing buffers
% echo "free 0" | sudo tee /dev/errorlab
% echo "free 99" | sudo tee /dev/errorlab # Should fail

# Fill up all buffers to test resource exhaustion
% echo "alloc" | sudo tee /dev/errorlab
% echo "alloc" | sudo tee /dev/errorlab
% echo "alloc" | sudo tee /dev/errorlab
% echo "alloc" | sudo tee /dev/errorlab # Should hit limit

# Check final status
% cat /dev/errorlab

# Clean up and unload
% echo "cleanup" | sudo tee /dev/errorlab
% sudo kldunload error_handling
```

Expected output:

```
Error Lab: Module loading with error handling demo
Error Lab: Processing command: 'alloc'
Error Lab: Allocated buffer 0 at 0xfffff8000c456000 (1/5 total)
Error Lab: Processing command: 'error_on'
Error Lab: Error injection ENABLED
Error Lab: Processing command: 'alloc'
Error Lab: Simulating allocation failure (error injection)
Error Lab: Allocation failed: Out of memory (12)
Error Lab: Final statistics:
  Operations: 4 total, 2 successful, 2 failed
```

Key Learning Points:

- Always use standard `errno.h` error codes for consistent behavior
- Every resource allocation needs a corresponding cleanup path
- Error injection helps test failure paths that are hard to trigger naturally
- Comprehensive state tracking aids in debugging and maintenance
- Graceful degradation is often better than complete failure

5.13.6 Lab Summary: Mastering the Kernel C Dialect

Congratulations! You've completed four essential labs that demonstrate the core differences between user-space C and kernel-space C. These labs weren't just coding exercises; they were lessons in thinking like a kernel programmer.

What You've Accomplished:

1. **Safe Memory Management** - You learned that kernel C requires perfect resource accounting. Every `malloc()` must have exactly one `free()`, especially during module unload.
2. **User-Kernel Communication** - You discovered that kernel C cannot directly access user space memory. Instead, you must use functions like `uiomove()` to safely cross the protection boundary.
3. **Context-Aware Logging** - You explored how kernel C provides different logging functions for different contexts, and why `device_printf()` is often more useful than generic `printf()`.
4. **Defensive Error Handling** - You practiced the kernel C discipline of handling every possible error condition gracefully, using proper error codes, and maintaining system stability even when operations fail.

The Dialect Difference:

These labs have shown you concretely what we meant by “kernel C is a dialect of C.” The vocabulary is the same; `malloc`, `printf`, `if`, `for` but the grammar, idioms, and cultural expectations are different:

- **User-space C**: “Allocate memory, use it, and hopefully remember to free it”
- **Kernel C**: “Allocate memory with explicit type tracking, validate all inputs, handle allocation failure gracefully, and guarantee cleanup on every code path”
- **User-space C**: “Print error messages to `stderr`”
- **Kernel C**: “Log with appropriate context, consider interrupt safety, avoid spamming the kernel log, and include diagnostic information for system administrators”
- **User-space C**: “Pass pointers freely between functions”
- **Kernel C**: “Use `copyin/copyout` for user space, validate all pointers, and never trust data that crosses protection boundaries”

This is the **mindset shift** that makes someone a kernel programmer. You now think in terms of system-wide impact, resource consciousness, and defensive assumptions.

Next Steps:

The patterns you’ve learned in these labs appear everywhere in the FreeBSD kernel:

- Device drivers use these same memory management patterns
- Network protocols use these same error handling strategies
- File systems use these same user-kernel communication techniques
- System calls use these same defensive programming practices

You’re now ready to read and understand real FreeBSD kernel code. More importantly, you’re ready to write kernel code that follows the same professional patterns used throughout the system.

5.14 Wrapping Up

We began this chapter with a simple truth: learning kernel programming requires more than just knowing C; it requires learning the **dialect of C spoken inside the FreeBSD kernel**. Over the course of this chapter, you’ve mastered that dialect and much more.

5.14.1 What You've Accomplished

You started this chapter knowing basic C programming, and you're finishing it with a comprehensive understanding of:

Kernel-specific data types that ensure your code works across different architectures and use cases. You now know why `uint32_t` is better than `int` for hardware registers, and when to use `size_t` versus `ssize_t`.

Memory management in an environment where every byte matters and every allocation must be carefully planned. You understand the difference between `M_WAITOK` and `M_NOWAIT`, how to use memory types for tracking and debugging, and why UMA zones exist.

Safe string handling that prevents the buffer overflows and format string bugs that have plagued system software for decades. You know why `strncpy()` exists, how to validate string lengths, and how to handle user data safely.

Function design patterns that make code predictable, maintainable, and integrable with the rest of the FreeBSD kernel. Your functions now follow the same conventions used by thousands of other kernel functions.

Kernel restrictions that might seem limiting but actually enable FreeBSD to be fast, reliable, and secure. You understand why floating-point is forbidden, why stacks are small, and how these constraints shape good design.

Atomic operations and synchronization primitives that allow safe concurrent programming on multiprocessor systems. You know when to use atomic operations versus mutexes, and how memory barriers ensure correctness.

Coding idioms and style that make your code look and feel like it belongs in the FreeBSD kernel. You've learned not just the technical APIs, but the cultural expectations of the FreeBSD development community.

Defensive programming techniques that turn potentially catastrophic bugs into handled error conditions. Your code now validates inputs, handles edge cases, and fails safely when things go wrong.

Error handling patterns that make debugging and maintenance possible in a system as complex as an operating system kernel. You understand how to propagate errors, provide diagnostic information, and recover gracefully from failures.

5.14.2 The Dialect Mastery

But perhaps most importantly, you've developed **fluency in the kernel C dialect**. Just as learning a regional dialect requires understanding not just different words, but different cultural context and social expectations, you now understand the kernel's unique culture:

- **System-wide impact:** Every line of code you write can affect the entire machine; kernel C doesn't tolerate casual programming
- **Resource consciousness:** Memory, CPU cycles, and stack space are precious resources; kernel C demands accounting for every allocation
- **Defensive assumptions:** Always assume the worst-case scenario and plan for it; kernel C expects paranoid programming
- **Long-term maintainability:** Code must be readable and debuggable years after it's written; kernel C values clarity over cleverness

- **Community integration:** Your code must fit alongside decades of existing code; kernel C has established patterns and idioms

This isn't just a different way of using C; it's a different way of **thinking** about programming. You've learned to speak the language that the FreeBSD kernel understands.

5.14.3 From Dialect to Fluency

The hands-on labs you completed weren't just exercises; they were **immersion experiences** in the kernel C dialect. Like spending time in a foreign country, you've learned not just the vocabulary, but the cultural nuances:

- How kernel programmers think about memory (every allocation tracked, every free guaranteed)
- How kernel programmers communicate across boundaries (copyin/copyout, never trusting user data)
- How kernel programmers handle uncertainty (comprehensive error handling, graceful degradation)
- How kernel programmers document their intentions (structured logging, diagnostic information)

These patterns appear in every significant piece of kernel code. You're now ready to read FreeBSD source code and understand not just *what* it does, but *why* it's written that way.

5.14.4 A Personal Reflection

When I first began exploring kernel programming, I found it intimidating, the kind of programming where a simple mistake could bring down an entire system. But over time, I discovered something surprising: **kernel development rewards discipline far more than brilliance.**

Once you accept its constraints, everything starts to make sense. Defensive programming stops feeling paranoid and becomes instinctive. Manual memory management turns from a chore into a craft. Every line of code matters, and that precision is deeply satisfying.

FreeBSD's kernel is an exceptional learning environment because it values clarity, consistency, and collaboration. If you've taken the time to absorb the material in this chapter, you now understand how the kernel "thinks in C." That mindset will serve you for the rest of your systems programming work.

5.14.5 The Next Chapter: From Language to Structure

You now speak the kernel's dialect of C, but speaking a language and writing an entire book are two different things. **Chapter 6 will not yet have you writing a complete driver from scratch.** Instead, it will show you the *blueprint* that all FreeBSD drivers share: how they are structured, how they integrate into the kernel's device framework, and how the system recognises and manages hardware components.

Think of it as walking into the **architect's studio** before we start building. We'll study the floor plan: the data structures, callback conventions, and the registration process that every driver follows. Once you understand that architecture, the following chapters will add the real engineering details, interrupts, DMA, buses, and beyond.

5.14.6 The Foundation Is Complete

The kernel C concepts you’ve learned so far, from data types to memory handling, from safe programming patterns to error discipline, are the raw materials of your future drivers.

Chapter 6 will begin assembling those materials into a recognisable form. You’ll see where each concept fits within the structure of a FreeBSD driver, setting the stage for the deeper, hands-on chapters that follow.

You’re no longer just learning to *code* in C; you’re learning to *design* within the system. The rest of this book will build upon that mindset, step by step, until you can confidently write, understand, and contribute real FreeBSD drivers.

5.15 Challenge Exercises: Practising the Kernel C Mindset

These exercises are designed to consolidate everything you’ve learned in this chapter. They don’t require new kernel mechanisms, only the skills and discipline you’ve already developed: working with kernel data types, handling memory safely, writing defensive code, and understanding kernel-space limitations.

Take your time. Each challenge can be completed with the same lab environment you used in previous examples.

5.15.1 Challenge 1: Trace the Data Type Origins

Open `/usr/src/sys/types.h` and locate at least **five typedefs** that appear in this chapter (e.g., `vm_offset_t`, `bus_size_t`, `sbintime_t`). For each:

- Identify what underlying C type it maps to on your architecture.
- Explain in a comment *why* the kernel uses a typedef instead of a primitive type.

Goal: see how portability and readability are built into FreeBSD’s type system.

5.15.2 Challenge 2: Memory Allocation Scenarios

Create a short kernel module that allocates memory three different ways:

1. `malloc()` with `M_WAITOK`
2. `malloc()` with `M_NOWAIT`
3. A UMA zone allocation (`uma_zalloc()`)

Log the pointer addresses and note what happens if you try to load the module when memory pressure is high. Then answer in comments:

- Why is `M_WAITOK` unsafe in interrupt context?
- What would be the correct pattern for emergency allocations?

Goal: understand **sleep vs. no-sleep contexts** and safe allocation choices.

5.15.3 Challenge 3: Error-Handling Discipline

Write a dummy kernel function that performs three sequential actions (e.g., allocate -> initialise -> register). Simulate a failure in the second step and use the `goto fail:` pattern for cleanup.

After unloading the module, verify through `vmstat -m` that no memory from your custom type remains allocated.

Goal: practise the “**single exit / single cleanup**” idiom common in FreeBSD.

5.15.4 Challenge 4: Safe String Operations

Modify your previous `memory_demo.c` or create a new module that copies a user-supplied string into a kernel buffer using `copyin()` and `strncpy()`. Ensure the destination buffer is cleared with `bzero()` before copying. Log the result with `printf()` and verify that the kernel never over-reads the source string.

Goal: combine **user-kernel boundary safety** with safe string handling.

5.15.5 Challenge 5: Diagnostics and Assertions

Insert a deliberate logic check into any of your demo modules, such as verifying that a pointer or counter is valid. Guard it with `KASSERT()` and observe what happens when the condition fails (in a test VM only!).

Then replace the `KASSERT()` with graceful error handling and re-test.

Goal: learn when to use **assertions vs. recoverable errors**.

5.15.6 What You’ll Gain

By completing these challenges, you’ll reinforce:

- Precision with kernel data types
- Conscious memory-allocation decisions
- Structured error handling and cleanup
- Respect for stack limits and context safety
- The discipline that distinguishes **user-space coding** from **kernel engineering**

You’re now ready to approach Chapter 6, where we start assembling these pieces into the real structure of a FreeBSD driver.

5.16 Summary Reference: User Space vs. Kernel Space Equivalents

When you move from user space to kernel space, many familiar C library calls and idioms change meaning or become unsafe.

This table summarises the most common translations you’ll use while developing FreeBSD device drivers.

Purpose	User-Space Function or Concept	Kernel-Space Equivalent	Notes / Differences
Program entry point	<code>int main(void)</code>	Module/event handler (e.g. <code>module_t</code> , <code>MOD_LOAD</code> , <code>MOD_UNLOAD</code>)	Kernel modules don't have <code>main()</code> ; entry and exit are managed by the kernel.
Printing output	<code>printf()</code> / <code>fprintf()</code>	<code>printf()</code> / <code>uprntf()</code> / <code>device_printf()</code>	<code>printf()</code> logs to the kernel console; <code>uprntf()</code> prints to the user's terminal; <code>device_printf()</code> prefixes the driver name.
Memory allocation	<code>malloc()</code> , <code>calloc()</code> , <code>free()</code>	<code>malloc(9)</code> , <code>free(9)</code> , <code>uma_zalloc()</code> , <code>uma_zfree()</code>	Kernel allocators require type and flags (<code>M_WAITOK</code> , <code>M_NOWAIT</code> , etc.).
Error handling	<code>errno</code> , return codes	Same (<code>EIO</code> , <code>EINVAL</code> , etc.)	Returned directly as function result; no global <code>errno</code> .
File I/O	<code>read()</code> , <code>write()</code> , <code>fopen()</code>	<code>uiomove()</code> , <code>copyin()</code> , <code>copyout()</code>	Drivers handle user data manually through <code>uio</code> or <code>copy</code> functions.
Strings	<code>strcpy()</code> , <code>sprintf()</code>	<code>strncpy()</code> , <code>snprintf()</code> , <code>bcopy()</code> , <code>bzero()</code>	All kernel string ops are bounded for safety.
Dynamic arrays / structures	<code>realloc()</code>	Usually re-implemented manually via new allocation + <code>bcopy()</code>	No generic <code>realloc()</code> helper in kernel.
Threads / concurrency	<code>pthread_mutex_*</code> (), <code>pthread_*</code> ()	<code>mtx_*</code> (), <code>sx_*</code> (), <code>rw_*</code> ()	Kernel provides its own synchronisation primitives.
Timers	<code>sleep()</code> , <code>usleep()</code>	<code>pause()</code> , <code>tsleep()</code> , <code>callout_*</code> ()	Kernel timing functions are tick-based and non-blocking.
Debugging	<code>gdb</code> , <code>printf()</code>	<code>KASSERT()</code> , <code>panic()</code> , <code>dtrace</code> , <code>printf()</code>	Kernel debugging requires in-kernel tools or <code>kgdb</code> .
Exit / termination	<code>exit()</code> / <code>return</code>	<code>MOD_UNLOAD</code> / <code>module unload</code>	Modules unload through kernel events, not process termination.

Purpose	User-Space Function or Concept	Kernel-Space Equivalent	Notes / Differences
Standard library headers	<code><stdio.h></code> , <code><stdlib.h></code>	<code><sys/param.h></code> , <code><sys/systm.h></code> , <code><sys/malloc.h></code>	Kernel uses its own headers and API set.
User-memory access	Direct pointer access	<code>copyin()</code> , <code>copyout()</code>	Never dereference user pointers directly.
Assertions	<code>assert()</code>	<code>KASSERT()</code>	Compiled only in debug kernels; triggers panic on failure.

5.16.1 Key Takeaways

- Always check which API context you're in before calling familiar C functions.
- Kernel APIs are designed for safety under strict constraints: a limited stack, no user library, and no floating point.
- By internalising these equivalents, you'll write safer, more idiomatic FreeBSD kernel code.

Next stop: The Anatomy of a Driver, where the language you've mastered begins to take shape as the living structure of the FreeBSD kernel.

6 The Anatomy of a FreeBSD Driver

6.1 Introduction

Chapter 5 left you fluent in the kernel dialect of C: you know the safe way to allocate, lock, copy, and tear down inside the kernel, and you have seen how a one-line mistake can cost you a kernel panic. This chapter takes that fluency and points it at a concrete subject, the **shape of a FreeBSD driver**. Think of it as moving from learning carpentry techniques to understanding architectural blueprints: before you build a house, you need to know where the foundation goes, how the frame connects, where utilities run, and how all the pieces fit together.

Important: This chapter focuses on understanding driver structure and patterns. You will not yet write a complete, fully functional driver in this chapter; that begins in Chapter 7. Here, we're building your mental model and pattern recognition skills first.

Writing a device driver can feel mysterious at first. You know it talks to hardware, you know it lives in the kernel, but **how does it all work?** How does the kernel discover your driver? How does it decide when to call your code? What happens when a user program opens `/dev/yourdevice`? And most importantly, what does the **blueprint** of a real, working driver actually look like?

This chapter answers those questions by showing you the **anatomy of FreeBSD drivers**, the common structures, patterns, and lifecycles that all drivers share. You'll learn:

- How drivers **plug into** FreeBSD through newbus, devfs, and module packaging
- The common patterns that character, network, and storage drivers follow
- The lifecycle from discovery through probe, attach, operation, and detach
- How to recognize driver structure in real FreeBSD source code
- Where to find your way around when reading or writing drivers

By the end of this chapter, you won't just understand drivers conceptually, you'll be able to **read real FreeBSD driver code** and immediately recognize the patterns. You'll know where to look for device attachment, how initialization happens, and how cleanup works. This chapter is your **blueprint** for understanding any driver you encounter in the FreeBSD source tree.

6.1.1 What This Chapter *IS*

This chapter is your **architectural tour** of driver structure. It teaches you:

- **Pattern recognition:** The shapes and idioms all drivers follow
- **Navigation skills:** Where to find what in driver source code
- **Vocabulary:** The names and concepts (newbus, devfs, softc, cdevsw, ifnet)
- **Lifecycle understanding:** When and why each driver function is called
- **Structural overview:** How the pieces connect without deep implementation

Think of this as learning to read blueprints before you start building.

6.1.2 What This Chapter *Is Not*

This chapter deliberately **defers deep mechanics** so we can focus on structure without overwhelming beginners. We will **not** cover in detail:

- **Bus specifics (PCI/USB/ACPI/FDT):** We'll mention buses conceptually, but skip hardware- and bus-specific discovery/attachment details.
- **Interrupt handling:** You'll see where handlers fit in a driver's lifecycle, not how to program or tune them.
- **DMA programming:** We'll acknowledge DMA and why it exists, not how to set up maps, tags, or synchronization.
- **Hardware register I/O:** We'll preview `bus_space_*` at a high level, not full MMIO/PIO access patterns.
- **Network packet paths:** We'll point out how `ifnet` surfaces an interface, not implement packet TX/RX pipelines.
- **GEOM internals:** We'll introduce storage surfaces, not provider/consumer plumbing or graph transformations.

If you're curious about these topics while reading, **great**, bookmark the terms and keep going. This chapter gives you the **map**; the detailed territories come later in the book.

6.1.3 Where This Chapter Fits

You're entering the final chapter of **Part 1 - Foundations**. By the time you finish this chapter, you will have a clear **blueprint** of how a FreeBSD driver is shaped and how it plugs into the system, completing the foundation you've been building:

- **Chapters 1 to 5 (so far):** why drivers matter, a safe lab, UNIX/FreeBSD basics, C for userspace, and C in the kernel context.
- **Chapter 6 (this chapter):** the driver's anatomy; structure, lifecycle, and the user-visible surface, so you can recognise the pieces before you start coding.

With that foundation in place, **Part 2 - Building Your First Driver** will move from concepts to code, step by step:

- **Chapter 7: Writing Your First Driver** - scaffold and load a minimal driver.
- **Chapter 8: Working with Device Files** - create a `/dev` node and wire basic entry points.
- **Chapter 9: Reading and Writing to Devices** - implement simple data paths for `read(2)/write(2)`.
- **Chapter 10: Handling Input and Output Efficiently** - introduce tidy, responsive I/O patterns.

Think of Chapter 6 as the **bridge**: you now have the language (C) and the environment (FreeBSD), and with this anatomy in mind, you're ready to start **building** in Part 2.

If you're skimming: **Chapter 6 = the blueprint. Part 2 = the build.**

6.2 Reader Guidance: How to Use This Chapter

This chapter is designed as both a **structural reference** and a **guided reading experience**. Unlike the hands-on coding focus of Chapter 7, this chapter emphasizes **understanding, pattern recognition, and navigation**. You'll spend time examining real FreeBSD driver code, identifying structures, and building a mental model of how everything connects.

6.2.1 Estimated Time Investment

Your total time depends on how deeply you engage. Use the track that fits your pace.

Track A - Read-through only Plan **8-10 hours** to absorb the concepts, skim the diagrams, and read the code excerpts at a comfortable beginner pace. This gives you a solid mental model without hands-on steps.

Track B - Read + follow along in `/usr/src` Plan **12-14 hours** if you open the referenced files under `/usr/src/sys` as you read, scroll around the surrounding context, and type the micro-snippets into a scratch file. This reinforces pattern recognition and navigation skills.

Track C - Read + follow along + all four labs Add **2.5-3.5 hours** to complete **all four** beginner-safe labs for this chapter: Lab 1 (Scavenger Hunt), Lab 2 (Hello Module), Lab 3 (Device Node), Lab 4 (Error Handling). These are short, focused checkpoints that validate what you learned in the chapter's tours and explanations.

Optional - Challenge questions Add **2-4 hours** to tackle the end-of-chapter challenges. These deepen your understanding of entry points, error unwinding, dependencies, and classification by reading real drivers.

Suggested pacing Break this chapter into two or three sessions. A practical split is: Session 1: Read through the driver model, skeleton, and lifecycle while following along in `/usr/src`. Session 2: Complete Labs 1-2. Session 3: Complete Labs 3-4 and, if desired, the challenges.

Reminder Don't rush. The goal here is **driver literacy**: the ability to open any driver, locate its probe/attach/detach paths, recognise the `cdev/ifnet/GEOM` shapes, and understand how it plugs into `newbus` and `devfs`. Mastery here makes Chapter 7's build go much faster and with fewer surprises.

6.2.2 What to Have Ready

To get the most from this chapter, prepare your workspace:

1. **Your FreeBSD lab environment** from Chapter 2 (VM or physical machine)
2. **FreeBSD 14.3 with `/usr/src` installed** (we'll reference real files from the kernel source tree)
3. **A terminal** where you can run commands and examine files
4. **Your lab logbook** for notes and observations
5. **Access to manual pages**: You'll frequently consult `man 9 <function>` for reference

Note: All examples were tested on FreeBSD 14.3; adjust commands if you use a different release.

6.2.3 Pacing and Approach

This chapter works best when you:

- **Read sequentially:** Each section builds on the previous one. The order matters.
- **Keep `/usr/src` open:** When we reference a file like `/usr/src/sys/dev/null/null.c`, actually open it and look at the surrounding context
- **Use `man 9` as you go:** When you see a function like `device_get_softc()`, run `man 9 device_get_softc` to see the official documentation
- **Type micro-snippets yourself:** Even in this “read-only” chapter, typing key patterns (like a probe function or method table) cements the shapes in your memory
- **Don’t rush the labs:** They’re designed as checkpoints. Complete each one before moving to the next section

6.2.4 Managing Your Curiosity

As you read, you’ll encounter concepts that spark deeper questions:

- “How exactly do PCI interrupts work?”
- “What are all the flags in `bus_alloc_resource_any()`?”
- “How does the network stack call my transmit function?”

This is expected and healthy. But resist the urge to dive down every rabbit hole now. This chapter is about recognizing patterns and understanding structure. The deep mechanics have their own dedicated chapters.

Strategy: Keep a “*Curiosity List*” in your lab logbook. When something piques your interest, jot it down with a note about where in the book it will be covered. For example:

Curiosity List:

- Interrupt handler details -> Chapter 19: Handling Interrupts and
-> Chapter 20: Advanced Interrupt Handling
- DMA buffer setup -> Chapter 21: DMA and High-Speed Data Transfer
- Network packet queues -> Chapter 28: Writing a Network Driver
- PCI configuration space -> Chapter 18: Writing a PCI Driver

This lets you acknowledge your questions without derailing your current focus.

6.2.5 Success Criteria

When you close this chapter you should be able to:

- Open any FreeBSD driver and immediately locate its probe, attach, and detach functions.
- Identify whether a driver is character, network, storage, or bus-oriented.
- Recognize a device method table and understand what it maps.
- Find the `softc` structure and understand its role.
- Trace the basic lifecycle from module load through device operation.
- Read logs and match them to driver lifecycle events.
- Locate relevant manual pages for key functions.

If you can do these things, you’re ready for Chapter 7’s hands-on coding.

6.3 How to Get the Most Out of This Chapter

Now that you know what to expect and how to pace yourself, let's discuss specific **learning tactics** that will make driver structure click for you. These strategies have proven effective for beginners tackling FreeBSD's driver model.

6.3.1 Keep `/usr/src` Nearby

Every code example in this chapter comes from real FreeBSD 14.3 source files. **Don't just read the snippets in this book**, open the actual files and see them in context.

Why this matters:

Seeing the full file shows you:

- How includes are organized at the top
- How multiple functions relate to each other
- Comments and documentation the original developers left
- Real-world patterns and idioms

6.3.1.1 Quick Locator: Where in the Source Tree?

Shape you're studying	Typical place in <code>/usr/src/sys</code>	A concrete file to open first
Minimal character device (<code>cdevsw</code>)	<code>dev/null/</code>	<code>dev/null/null.c</code>
Simple infrastructure device (LED)	<code>dev/led/</code>	<code>dev/led/led.c</code>
Pseudo-network interface (tun/tap)	<code>net/</code>	<code>net/if_tuntap.c</code>
UART PCI "glue" example	<code>dev/uart/</code>	<code>dev/uart/uart_bus_pci.c</code>
Bus plumbing (for reference)	<code>dev/pci/</code> , <code>kern/</code> , <code>bus/</code>	skim <code>dev/pci/pcib*.c</code> and related

Tip: open one of these side-by-side with the explanations to strengthen pattern recognition.

Practical tip: Keep a second terminal or editor window open. When the text says:

```
"Here's an example from null_cdevsw in /usr/src/sys/dev/null/null.c:"
```

Actually navigate there:

```
% cd /usr/src/sys/dev/null
% less null.c
```

Use `/` in `less` to search for patterns like `probe` or `cdevsw`, and jump directly to relevant sections.

A note on line numbers. Wherever this chapter gives an occasional line number, treat it as accurate for the FreeBSD 14.3 tree at time of writing and nothing more. Function, structure, and table names are the durable reference. When a chapter exercise or hint would otherwise quote line numbers, we cite the enclosing function, cdevsw structure, or named array instead; open the file and jump to that symbol.

6.3.2 Type the Micro-Snippets Yourself

Even though Chapter 7 is where you'll write complete drivers, **typing short patterns now** builds fluency.

When you see a probe function example, don't just read it, **type it in a scratch file**:

```
static int
mydriver_probe(device_t dev)
{
    device_set_desc(dev, "My Example Driver");
    return (BUS_PROBE_DEFAULT);
}
```

Why this works: Typing engages muscle memory. Your fingers learn the shapes (`device_t`, `BUS_PROBE_DEFAULT`) faster than your eyes alone. By the time you reach Chapter 7, these patterns will feel natural.

Practical tip:

Create a scratch directory:

```
% mkdir -p ~/scratch/chapter06
% cd ~/scratch/chapter06
% vi patterns.c
```

Use this space to collect the patterns you're learning.

6.3.3 Treat Labs as Checkpoints

This chapter includes four hands-on labs (see “Hands-On Labs” section):

1. **Lab 1:** Read-only scavenger hunt through real drivers
2. **Lab 2:** Build and load a minimal module that just logs messages
3. **Lab 3:** Create and remove a device node in `/dev`
4. **Lab 4:** Error handling and defensive programming

Don't skip these. They're your validation that the concepts have moved from “I read about it” to “I can do it.”

Timing: Complete each lab when you reach the “Hands-On Labs” section, not before. The labs assume you've read the earlier sections covering driver structure and patterns. They're designed to synthesize everything into hands-on practice.

Success mindset: Labs are meant to be achievable. If you get stuck, revisit the relevant section, check the man 9 pages cited in the text, and use the **Summary Reference Table - Driver Building Blocks at a Glance** at the end of the chapter. Each lab should take 20-45 minutes.

6.3.4 Defer the Deep Mechanics

This chapter repeatedly says things like:

- “Interrupts are covered in Chapter 19 and 20”
- “DMA details in Chapter 21”
- “Network packet processing in Chapter 28”

Trust this structure. Trying to learn everything at once leads to confusion and burnout.

Analogy: When you learn to drive, you first understand the car’s controls (steering, pedals, gearshift) before studying engine mechanics. Similarly, learn driver *structure* now, and study *mechanisms* later when you have context.

Strategy: When you hit a “defer this” moment, acknowledge it and move on. The deep topics are coming, and they’ll make more sense once you’ve written a basic driver.

6.3.5 Use man 9 as Your Reference

FreeBSD’s section 9 manual pages document kernel interfaces. They’re invaluable but can be dense.

When to use them:

- You see a function name you don’t recognize
- You want to know all parameters and return values
- You need to confirm behavior

Example:

```
% man 9 device_get_softc
% man 9 bus_alloc_resource
% man 9 make_dev
```

Pro tip: Use apropos to search for related functions:

```
% apropos device | grep "^device"
```

This shows you all device-related functions at once.

Companion reference: For a curated, book-internal summary of the same APIs you will meet throughout this chapter (malloc(9), mtx(9), callout(9), bus_alloc_resource_*, bus_space(9), Newbus macros, and more), Appendix A groups them into themed cheat sheets. It is not a replacement for man 9; it is the short lookup you reach for while reading, so you can keep your place in the chapter.

6.3.6 Skim Code Before Reading Explanations

When a section references a source file, try this approach:

1. **Skim the file first** (30 seconds)
2. **Notice patterns** (where are probe/attach? what includes are there?)
3. **Then read the explanation** in this chapter
4. **Go back to the code** with newfound understanding

Why this works: Your brain creates a rough mental map first, then the explanation fills in details. This is more effective than reading explanation -> code, which treats code as an afterthought.

6.3.7 Visualize as You Read

Driver structure has a lot of moving parts: buses, devices, methods, lifecycles. **Draw diagrams** as you encounter new concepts.

Examples of useful diagrams:

- Device tree showing parent-child relationships
- Lifecycle flowchart (probe -> attach -> operate -> detach)
- Character device flow (open -> read/write -> close)
- Relationship between `device_t`, `softc`, and `cdev`

Tools: Paper and pencil work great. Or use simple text art:

```
root
|- nexus0
   |- acpi0
      |- pci0
         |- em0 (network)
         |- ahci0 (storage)
         |- ada0 (disk)
```

6.3.8 Study Patterns Across Multiple Drivers

The “Read-Only Tour of Tiny Real Drivers” section tours four real drivers (null, led, tun and minimal PCI). Don’t just read each in isolation, **compare them**:

- How does `null.c` structure its `cdevsw` vs. `led.c`?
- Where does each driver initialize its `softc`?
- What’s similar in their probe functions? What’s different?

Pattern recognition is the goal. Once you see the same shape repeated, you’ll recognize it everywhere.

6.3.9 Set Realistic Expectations

Plan for about 18-22 hours if you complete all activities in this chapter (reading, tours, labs, and reviews). If you also tackle the optional challenges, allow up to an additional 4 hours. At two hours per day, expect roughly a week or a little more, **that's normal and expected**.

This isn't a race. The goal is **mastery of structure**, which is foundational for every subsequent chapter.

Mindset: Think of this chapter as a **training program**, not a sprint. Athletes don't try to gain all their strength in one session. Similarly, you're building **driver literacy** gradually.

6.3.10 When to Take Breaks

You'll know you need a break when:

- You've read the same paragraph three times without absorbing it
- Function names start blurring together
- You feel overwhelmed by details

Solution: Step away. Go for a walk, do something else, then return refreshed. This material will still be here, and your brain processes complex information better with rest.

6.3.11 You're Building a Foundation

Remember: **This chapter is your blueprint**. Chapter 7 is where you'll build actual code. Investing time here pays enormous dividends later because you won't be guessing about the structure; you'll know it.

Let's begin with the big picture.

6.4 The Big Picture: How FreeBSD Sees Devices and Drivers

Before we examine any code, we need to establish a **mental model** of how FreeBSD conceptually organizes devices and drivers. Understanding this model is like learning how a building's plumbing works before you replace a pipe, you need to know where the water comes from and where it goes.

This section provides the **one-page overview** you'll carry forward through the rest of the chapter. We'll define key terms, show how pieces connect, and give you just enough vocabulary to navigate the rest of the material without drowning in details.

6.4.1 One-Screen Driver Lifecycle

```
Boot/Hot-plug
|
v
[ Device enumerated by bus ]
```

```
| (PCI/USB/ACPI/FDT discovers hardware and creates device_t)
v
[ probe(dev) ]
| Decide: "Am I the right driver?" (score and return priority)
| If not mine -> return ENXIO / lower score
v
[ attach(dev) ]
| Allocate softc/state
| Claim resources (memory BARs/IRQ/etc.)
| Create user surface (e.g., make_dev / ifnet)
| Register callbacks, start timers
v
[ operate ]
| Runtime: open/read/write/ioctl, TX/RX, interrupts, callouts
| Normal errors handled; resources reused
v
[ detach(dev) ]
| Quiesce I/O and timers
| Destroy user surface (destroy_dev / if_detach / etc.)
| Free resources and state
v
Goodbye
```

Keep this flow in mind while you read the tours, every driver you'll see fits this outline.

6.4.2 Devices, Drivers, and Devclasses

FreeBSD uses precise terminology for the components in its device model. Let's define them in plain language:

Device

A **device** is the kernel's representation of a hardware resource or logical entity. It's a `device_t` structure that the kernel creates and manages.

Think of it as a **name tag** for something the kernel needs to track: a network card, a disk controller, a USB keyboard, or even a pseudo-device like `/dev/null`.

Key insight: A device exists whether or not a driver is attached to it. During boot, buses enumerate hardware and create `device_t` structures for everything they find. These devices sit waiting for drivers to claim them.

Driver

A **driver** is **code** that knows how to control a specific type of device. It's the implementation, the probe, attach, and operational functions that make hardware useful.

A single driver can handle multiple device models. For example, the `em` driver handles dozens of different Intel Ethernet cards by checking device IDs and adapting behavior.

Devclass

A **devclass** (device class) is a **grouping** of related devices. It's how FreeBSD keeps track of, say, "all UART devices" or "all disk controllers."

When you run `sysctl dev.em`, you're querying the `em` devclass, which shows all instances (`em0`, `em1`, etc.) managed by that driver.

Example:

```
devclass: uart
devices in this class: uart0, uart1, uart2
each device has a driver attached (or not)
```

Relationship summary:

- **Devclass** = category (e.g., "network interfaces")
- **Device** = instance (e.g., "em0")
- **Driver** = code (e.g., the `em` driver's functions)

Why this matters: When you write a driver, you'll register it with a devclass, and each device your driver attaches to becomes part of that class.

6.4.3 The Bus Hierarchy and Newbus (One Page)

FreeBSD organizes devices in a **tree structure** called the **device tree**, with buses as internal nodes and devices as leaves. This is managed by a framework called **Newbus**.

What is a bus?

A **bus** is any device that can have children. Examples:

- **PCI bus:** Contains PCI cards (network, graphics, storage controllers)
- **USB hub:** Contains USB peripherals
- **ACPI bus:** Contains platform devices enumerated by ACPI tables

The device tree structure:

```
root
|- nexus0 (platform-specific root bus)
   |- acpi0 (ACPI bus)
      |- cpu0
      |- cpu1
      |- pci0 (PCI bus)
         |- em0 (network card)
         |- ahci0 (SATA controller)
            |   |- ada0 (disk)
            |   |- ada1 (disk)
            |- ehci0 (USB controller)
```

```
| - usb0 (USB bus)
    | - ukbd0 (USB keyboard)
```

What is Newbus?

Newbus is FreeBSD's object-oriented device framework. It provides:

- **Device discovery:** Buses enumerate their children
- **Driver matching:** Probe functions determine which driver fits each device
- **Resource management:** Buses allocate IRQs, memory ranges, and other resources to devices
- **Lifecycle management:** Coordinating probe, attach, detach

The probe-attach flow:

1. A bus (e.g., PCI) **enumerates** its devices by scanning hardware
2. For each device, the kernel creates a `device_t`
3. The kernel calls every compatible driver's **probe** function: "Can you handle this?"
4. The driver with the best match wins
5. The kernel calls that driver's **attach** function to initialize it

Why "Newbus"?

It replaced an older, less flexible device framework. The "new" is historical; it's been standard for decades now.

Your role as a driver author:

- You write probe, attach, and detach functions
- Newbus calls them at the right times
- You don't manually search for devices, Newbus brings them to you

See it in action:

```
% devinfo -rv
```

This shows the complete device tree with resource assignments.

6.4.4 From Kernel to /dev: What devfs Presents

Many devices (especially character devices) appear as **files in /dev**. How does that work?

devfs (device filesystem)

`devfs` is a special filesystem that dynamically presents device nodes as files. It's **kernel-managed**: when a driver creates a device node, it instantly appears in `/dev`. When the driver is unloaded, the node disappears.

Why files?

The UNIX philosophy: "everything is a file" means uniform access:

```
% ls -l /dev/null
crw-rw-rw-  1 root  wheel  0x14 Oct 14 12:34 /dev/null
```

That `c` means **character device**. The major number (part of `0x14`) identifies the driver; the minor number identifies which instance.

Note: Historically, the device number was split into ‘major’ (driver) and ‘minor’ (instance). With `devfs` and dynamic devices on modern FreeBSD, you don’t rely on fixed major/minor values; treat that number as an internal identifier, and use the `cdev` and `devfs` APIs instead.

User-space view:

When a program opens `/dev/null`, the kernel:

1. Looks up the device by major/minor number
2. Finds the associated `cdev` (character device structure)
3. Calls the driver’s **`d_open`** function
4. Returns a file descriptor to the program

For reads/writes:

- User program calls `read(fd, buf, len)`
- Kernel translates to driver’s **`d_read`** function
- Driver handles it, returns data or error
- Kernel passes result back to user program

Not all devices appear in `/dev`:

- **Network interfaces** (`em0`, `wlan0`) appear in `ifconfig`, not `/dev`
- **Storage layers** often use `/dev/ada0`, but `GEOM` adds complexity
- **Pseudo-devices** may or may not create nodes

Key takeaway: Character drivers typically create `/dev` entries using `make_dev()`, and `devfs` makes them visible. We’ll cover this in detail in the “Creating and Removing Device Nodes” section.

6.4.5 Your Manual-Page Map (Read, Don’t Memorize)

FreeBSD’s section 9 manual pages document kernel APIs. Here’s your **starter map** of the most important pages for driver development. You don’t need to memorize these, just know they exist so you can look them up later.

Core device and driver APIs:

- `device(9)` - Overview of the `device_t` abstraction
- `devclass(9)` - Device class management
- `DRIVER_MODULE(9)` - Registering your driver with the kernel
- `DEVICE_PROBE(9)` - How probe methods work
- `DEVICE_ATTACH(9)` - How attach methods work
- `DEVICE_DETACH(9)` - How detach methods work

Character devices:

- `make_dev(9)` - Creating device nodes in `/dev`
- `destroy_dev(9)` - Removing device nodes
- `cdev(9)` - Character device structure and operations

Network interfaces:

- `ifnet(9)` - Network interface structure and registration
- `if_attach(9)` - Attaching a network interface
- `mbuf(9)` - Network buffer management

Storage:

- `GEOM(4)` - Overview of FreeBSD's storage layer (note: section 4, not 9)
- `g_bio(9)` - Bio (block I/O) structure

Resources and hardware access:

- `bus_alloc_resource(9)` - Claiming IRQs, memory, etc.
- `bus_space(9)` - Portable MMIO and PIO access
- `bus_dma(9)` - DMA memory management

Module and lifecycle:

- `module(9)` - Kernel module infrastructure
- `MODULE_DEPEND(9)` - Declaring module dependencies
- `MODULE_VERSION(9)` - Versioning your module

Locking and synchronization:

- `mutex(9)` - Mutual exclusion locks
- `sx(9)` - Shared/exclusive locks
- `rmlock(9)` - Read-mostly locks

Utility functions:

- `printf(9)` - Kernel printf variants (including `device_printf`)
- `malloc(9)` - Kernel memory allocation
- `sysctl(9)` - Creating sysctl nodes for observability

How to use this map:

When you encounter an unfamiliar function or concept, check if it has a man page:

```
% man 9 <function_or_topic>
```

Examples:

```
% man 9 device_get_softc
% man 9 bus_alloc_resource
% man 9 make_dev
```


If you're not sure of the exact name, use apropos:

```
% apropos -s 9 device
```

Pro tip: Many man pages include **SEE ALSO** sections at the bottom, pointing to related topics. Follow those breadcrumbs when exploring.

This is your reference library. You don't read it cover-to-cover, you consult it when needed. As you work through this chapter and later chapters, you'll build familiarity with the most common pages naturally.

Summary

You now have the **big picture**:

- **Devices** are kernel objects, **drivers** are code, **devclasses** are groupings
- **Newbus** manages the device tree and driver lifecycle (probe/attach/detach)
- **devfs** presents devices as files in /dev (for character devices)
- **Manual pages** in section 9 are your reference library

This mental model is your foundation. In the next section, we'll explore the different **families of drivers** and how to choose the right shape for your hardware.

6.5 Driver Families: Choosing the Right Shape

Not all drivers are created equal. Depending on what your hardware does, you'll need to present the right "face" to the FreeBSD kernel. Think of driver families like professional specializations: a cardiologist and an orthopedic surgeon are both doctors, but they work very differently. Similarly, a character device driver and a network driver both interact with hardware, but they plug into different parts of the kernel.

This section helps you **identify which family your driver belongs to** and understand the structural differences between them. We'll keep this at a recognition level, later chapters will cover implementation.

6.5.1 Character Devices

Character devices are the simplest and most common driver family. They present a **stream-oriented interface** to user programs: open, close, read, write, and ioctl.

When to use:

- Hardware that sends or receives data byte-by-byte or in arbitrary chunks
- Control surfaces for configuration (LEDs, GPIO pins)
- Sensors, serial ports, sound cards, custom hardware
- Pseudo-devices that implement software functionality

User-space view:

```
% ls -l /dev/cuau0
crw-rw----  1 root  dialer  0x4d Oct 14 10:23 /dev/cuau0
```

Programs interact with character devices like files:

```
int fd = open("/dev/cuau0", O_RDWR);
write(fd, "Hello", 5);
read(fd, buffer, sizeof(buffer));
ioctl(fd, SOME_COMMAND, &arg);
close(fd);
```

Kernel view:

Your driver implements a struct `cdevsw` (character device switch) with function pointers:

```
static struct cdevsw mydev_cdevsw = {
    .d_version = D_VERSION,
    .d_open    = mydev_open,
    .d_close   = mydev_close,
    .d_read    = mydev_read,
    .d_write   = mydev_write,
    .d_ioctl   = mydev_ioctl,
    .d_name    = "mydev",
};
```

When a user program calls `read()`, the kernel routes it to your `mydev_read()` function.

Examples in FreeBSD:

- `/dev/null`, `/dev/zero`, `/dev/random` - Pseudo-devices
- `/dev/led/*` - LED control
- `/dev/cuau0` - Serial port
- `/dev/dsp` - Audio device

Why start here: Character devices are the **simplest family** to understand and implement. If you're learning driver development, you'll almost certainly start with a character device. Chapter 7's first driver is a character device.

6.5.2 Storage via GEOM (Why “Block Devices” Are Different Here)

FreeBSD's storage architecture centers on **GEOM** (Geometry Management), a modular framework for storage transformations and layering.

Historical note: Traditional UNIX had “block devices” and “character devices.” Modern FreeBSD **unified this**, all devices are character devices, and GEOM sits on top to provide block-level storage services.

GEOM conceptual model:

- **Providers:** Supply storage (e.g., a disk: `ada0`)
- **Consumers:** Use storage (e.g., a filesystem)
- **Geoms:** Transformations in between (partitioning, RAID, encryption)

Example stack:

```
Filesystem (UFS)
    -> consumes
GEOM LABEL (geom_label)
    -> consumes
GEOM PART (partition table)
    -> consumes
ada0 (disk driver via CAM)
    -> talks to
AHCI driver (hardware)
```

When to use:

- You're writing a disk controller driver (SATA, NVMe, SCSI)
- You're implementing a storage transformation (software RAID, encryption, compression)
- Your device presents block-oriented storage

User-space view:

```
% ls -l /dev/ada0
crw-r-----  1 root  operator   0xa9 Oct 14 10:23 /dev/ada0
```

Notice it's still a character device (c), but GEOM and the buffer cache provide block semantics.

Kernel view:

Storage drivers typically interact with **CAM (Common Access Method)**, FreeBSD's SCSI/ATA layer. You register a **SIM (SCSI Interface Module)** that handles I/O requests.

Alternatively, you can create a GEOM class that processes **bio (block I/O)** requests.

Examples:

- `ahci` - SATA controller driver
- `nvd` - NVMe disk driver
- `gmirror` - GEOM mirror (RAID 1)
- `geli` - GEOM encryption layer

Why this is advanced

Storage drivers involve understanding:

- DMA and scatter-gather lists
- Block I/O scheduling
- CAM or GEOM frameworks
- Data integrity and error handling

We won't cover this in depth until much later. For now, just recognize that storage drivers have a different shape than character devices.

6.5.3 Network via ifnet

Network drivers don't appear in `/dev`. Instead, they register as **network interfaces** that appear in `ifconfig` and integrate with the FreeBSD network stack.

When to use:

- Ethernet cards
- Wireless adapters
- Virtual network interfaces (tunnels, bridges, VPNs)
- Any device that sends/receives network packets

User-space view:

```
% ifconfig em0
em0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> metric 0 mtu 1500
    ether 00:0c:29:3a:4f:1e
    inet 192.168.1.100 netmask 0xffffffff broadcast 192.168.1.255
```

Programs don't open network interfaces directly. Instead, they create sockets and the kernel routes packets through the appropriate interface.

Kernel view:

Your driver allocates and registers an **if_t** (interface) structure:

```
if_t ifp;

ifp = if_alloc(IFT_ETHER);
if_setsoftc(ifp, sc);
if_initname(ifp, device_get_name(dev), device_get_unit(dev));
if_setflags(ifp, IFF_BROADCAST | IFF_SIMPLEX | IFF_MULTICAST);
if_setinitfn(ifp, mydriver_init);
if_setioctlfn(ifp, mydriver_ioctl);
if_settransmitfn(ifp, mydriver_transmit);
if_setqflushfn(ifp, mydriver_qflush);

ether_ifattach(ifp, sc->mac_addr);
```

Your driver must handle:

- **Transmit:** Kernel gives you packets (mbufs) to send
- **Receive:** You receive packets from hardware and pass them up the stack
- **Initialization:** Configure hardware when interface comes up
- **ioctl:** Handle configuration changes (address, MTU, etc.)

Examples:

- `em` - Intel Ethernet (e1000 family)

- **igb** - Intel Gigabit Ethernet
- **bge** - Broadcom Gigabit Ethernet
- **if_tun** - Tunnel device

Why this is different

Network drivers must:

- Manage packet queues and mbuf chains
- Handle link state changes
- Support multicast filtering
- Implement hardware offload features (checksums, TSO, etc.)

Chapter 28 covers network driver development in depth.

6.5.4 Pseudo and Clone Devices (Safe, Small, Instructive)

Pseudo-devices are software-only drivers with no backing hardware. They're **perfect for learning** because you can focus entirely on driver structure without worrying about hardware behavior.

Common pseudo-devices:

1. **null** (`/dev/null`) - Discards writes, returns EOF on reads
2. **zero** (`/dev/zero`) - Returns infinite zeros
3. **random** (`/dev/random`) - Random number generator
4. **md** - Memory disk (RAM disk)
5. **tun/tap** - Network tunnel devices

Why they're valuable for learning:

- No hardware complexity (no registers, no DMA, no interrupts)
- Focus purely on driver structure and lifecycle
- Easy to test (just read/write to `/dev`)
- Small, readable source code

Special case: Clone devices

Some pseudo-devices support **multiple simultaneous opens** by creating new device nodes on demand. Example: `/dev/bpf` (Berkeley Packet Filter).

When you open `/dev/bpf`, the driver allocates a new instance (`/dev/bpf0`, `/dev/bpf1`, etc.) for your session.

Example: tun device (hybrid)

The tun device is interesting because it's **both**:

- A **character device** (`/dev/tun0`) for control
- A **network interface** (`tun0` in `ifconfig`) for data

Programs open `/dev/tun0` to configure the tunnel, but packets flow through the network interface. This “mixed model” demonstrates how drivers can present multiple surfaces.

Where to find them in the source:

```
% ls /usr/src/sys/dev/null/  
% ls /usr/src/sys/dev/md/  
% ls /usr/src/sys/net/if_tuntap.c
```

The “Read-Only Tour of Tiny Real Drivers” section will tour these drivers in detail. For now, just recognize that pseudo-devices are your **training wheels**, simple enough to understand, real enough to be useful.

6.5.5 Decision Checklist: Which Shape Fits?

Use this checklist to identify the right driver family for your hardware:

Choose Character Device if:

- Hardware sends/receives arbitrary data streams (not packets, not blocks)
- User programs need direct file-like access (open/read/write)
- You’re implementing a control interface (GPIO, LED, sensor)
- It’s a pseudo-device providing software functionality
- It doesn’t fit the network or storage models

Choose Network Interface if:

- Hardware sends/receives network packets (Ethernet frames, etc.)
- Should integrate with the network stack (routing, firewalls, sockets)
- Appears in `ifconfig`, not `/dev`
- Needs to support protocols (TCP/IP, etc.)

Choose Storage/GEOM if:

- Hardware provides block-oriented storage
- Should appear as a disk in the system
- Needs to support filesystems
- Requires partitioning, or sits in a storage transformation stack

Mixed Models:

- Some devices (like `tun`) present both a control plane (character device) and a data plane (network interface or storage)
- This is less common but useful when needed

Still unsure?

- Look at similar existing drivers
- Check what user programs expect (do they open files, or use sockets?)
- Ask: “What subsystem does my hardware naturally integrate with?”

6.5.6 Mini Exercise: Classify Live Drivers

Let’s practice pattern recognition on your running FreeBSD system.

Instructions:

1. Identify one character device:

```
% ls -l /dev/null /dev/random /dev/cuau*
```

Pick one. What makes it a character device?

2. Identify one network interface:

```
% ifconfig -l
```

Pick one (e.g., em0, lo0). Look it up:

```
% man 4 em
```

What hardware does it drive?

3. Identify one storage participant:

```
% geom disk list
```

Pick a disk (e.g., ada0 or nvd0). What driver manages it?

4. Find the driver source:

For each, try to locate its source:

```
% find /usr/src/sys -name "null.c"
% find /usr/src/sys -name "if_em.c"
% find /usr/src/sys -name "ahci.c"
```

5. Record in your lab logbook:

```
Character: /dev/random -> sys/dev/random/randomdev.c
Network:   em0 -> sys/dev/e1000/if_em.c
Storage:   ada0 (via CAM) -> sys/dev/ahci/ahci.c
```

What you're learning: Recognition. By the time you've done this, you've connected abstract concepts (character, network, storage) to real, concrete examples on your system.

Summary

Drivers come in families with different shapes:

- **Character devices:** Stream I/O via /dev, simplest to learn
- **Storage devices:** Block I/O via GEOM/CAM, advanced
- **Network interfaces:** Packet I/O via ifnet, no /dev presence
- **Pseudo-devices:** Software-only, perfect for learning structure

Choosing the right shape: Match your hardware's purpose to the kernel subsystem it naturally integrates with.

In the next section, we'll examine the **minimal driver skeleton**, the universal scaffolding that all drivers share, regardless of family.

6.6 The Minimal Driver Skeleton

Every FreeBSD driver, from the simplest pseudo-device to the most complex PCI controller, shares a common **skeleton**, a scaffolding of required components that the kernel expects. Think of this skeleton as the chassis of a car: before you can add an engine, seats, or a stereo, you need the basic frame that everything else bolts onto.

This section introduces the universal pattern you'll see in every driver. We'll keep this **minimal**, just enough to load, attach, and unload cleanly. Later sections and chapters will add the muscles, organs, and features.

6.6.1 Core Types: `device_t` and the `softc`

Two fundamental types appear in every driver: `device_t` and your driver's **softc** (software context) structure.

6.6.1.1 `device_t` - the kernel's handle to *this* device

`device_t` is an **opaque handle** managed by the kernel. You never poke inside it; you ask the kernel for what you need via accessors.

```
#include <sys/bus.h>

const char *name    = device_get_name(dev);    // e.g., "mydriver"
int         unit     = device_get_unit(dev);    // 0, 1, 2, ...
device_t    parent   = device_get_parent(dev); // the parent bus (PCI, USB, etc.)
void        *cookie   = device_get_softc(dev);  // pointer to your softc (explained below)
```

Why opaque?

So the kernel can evolve its internal representation without breaking your code. You interact through a stable API instead of struct fields.

Where you see it

Every lifecycle callback (probe, attach, detach, ...) receives a `device_t dev`. That parameter is your “session” with the kernel for this particular device instance.

6.6.1.2 The `softc` - your driver's private state

Each device instance needs a place to keep state: resources, locks, stats, and any hardware-specific bits. That's the **softc** you define.

You define it

```
struct mydriver_softc {
    device_t    dev;           // back-pointer to device_t (handy for prints, etc.)
    struct resource *mem_res;   // MMIO resource
    int         mem_rid;       // resource ID (e.g., PCIR_BAR(0))
    struct mtx   mtx;          // driver lock
```



```
uint64_t      bytes_rx;    // example statistic
/* ... your driver-specific state ... */
};
```

The kernel allocates it for you

When you register the driver, you tell Newbus the size of your softc:

```
static driver_t mydriver_driver = {
    "mydriver",
    mydriver_methods,
    sizeof(struct mydriver_softc) // Newbus allocates and zeroes this per instance
};
```

Newbus creates (and zeroes) one softc **per device instance** during device creation. You don't malloc() it, and you don't free() it.

You retrieve it where you work

```
static int
mydriver_attach(device_t dev)
{
    struct mydriver_softc *sc;

    sc = device_get_softc(dev); // get your per-instance state
    sc->dev = dev;              // stash the handle for convenience

    /* initialize locks/resources, map registers, set up interrupts, etc. */
    return (0);
}
```

That one-liner

```
struct mydriver_softc *sc = device_get_softc(dev);
```

appears at the top of almost every driver method that needs state. It's the idiomatic way to enter your driver's world.

6.6.1.3 Mental model

- **device_t**: “ticket” the kernel hands you for *this* device.
- **softc**: your “backpack” of state tied to that ticket.
- **Access pattern**: kernel calls your method with dev -> you call device_get_softc(dev) -> operate via sc->...

6.6.1.4 Before we move on

- **Lifetime:** the softc exists once Newbus creates the device object and lasts until the device is deleted. You still must **destroy locks and release resources** in detach; Newbus only frees the softc memory.
- **Probe vs attach:** identify in probe; **don't** allocate resources there. Initialize hardware in attach.
- **Types:** `device_get_softc()` returns `void *`; assigning to `struct mydriver_softc *` is fine in C (no cast needed).

That's all you need for the skeleton. We'll layer in resources, interrupts, and power management in their dedicated sections, keeping this mental model as home base.

6.6.2 Method Tables and kobj - Why Callbacks Look “Magical”

FreeBSD drivers use **method tables** to connect your functions to Newbus. This might look a bit magical at first, but it's actually simple and elegant.

The method table:

```
static device_method_t mydriver_methods[] = {
    /* Device interface (device_if.m) */
    DEVMETHOD(device_probe,      mydriver_probe),
    DEVMETHOD(device_attach,     mydriver_attach),
    DEVMETHOD(device_detach,     mydriver_detach),

    DEVMETHOD_END
};
```

What this table means (practical view)

It's a routing table from Newbus “method names” to **your** functions:

- **device_probe -> mydriver_probe** Runs when the kernel is asking “does this driver match this device?” *Do:* check IDs/compat strings, set a description if you like, return a probe result. *Don't:* allocate resources or touch hardware yet.
- **device_attach -> mydriver_attach** Runs after your probe wins. *Do:* allocate resources (MMIO/IRQs), initialize hardware, set up interrupts, create your `/dev` node if applicable. Handle failures cleanly. *Don't:* leave partial state behind, either unwind or fail gracefully.
- **device_detach -> mydriver_detach** Runs when the device is being removed/unloaded. *Do:* stop hardware, tear down interrupts, destroy device nodes, free resources, destroy locks. *Don't:* return success if the device is still in use; return `EBUSY` when appropriate.

Why keep it this small?

This chapter focuses on the *driver skeleton*. We add power management and other hooks later, so you master the core lifecycle first.

The magic behind it: kobj

Under the hood, FreeBSD uses **kobj** (kernel objects) to implement method dispatch:

1. Interfaces (collections of methods) are defined in .m files (e.g., `device_if.m`, `bus_if.m`).
2. Build tools generate C glue from those .m files.
3. At runtime, `kobj` uses your method table to look up the right function to call.

Example

When the kernel wants to probe a device, it effectively does:

```
DEVICE_PROBE(dev); // The macro expands to a kobj lookup; kobj finds mydriver_probe
↪ here
```

Why this matters

- The kernel can call methods polymorphically (same call, different driver implementations).
- You override only what you need; unimplemented methods fall back to defaults where appropriate.
- Interfaces are composable: you'll add more (e.g., bus or power-management methods) as your driver grows.

What you'll add later (when ready)

- **device_shutdown** -> **mydriver_shutdown** Called during reboot/power-off to put hardware in a safe state. *(Add once your basic attach/detach path is solid.)*
- **device_suspend** / **device_resume** For sleep/hibernate support: quiesce and restore hardware. *(Covered when we tackle power management in Chapter 22.)*

Mental model

Think of the table as a dictionary: keys are method names like `device_attach`; values are your functions. The `DEVICE_*` macros ask `kobj` to “find the function for this method on this object,” and `kobj` consults your table to call it. No magic, just generated dispatch code.

6.6.3 Registration Macros You'll Always Meet

These macros are the driver's “business card.” They tell the kernel **what you are**, **where you attach**, and **what you depend on**.

6.6.3.1 1) DRIVER_MODULE - register your driver

```
/* Minimal pattern: pick the correct parent bus for your hardware */
DRIVER_MODULE(mydriver, pci, mydriver_driver, NULL, NULL);

/*
 * Use the parent bus your device lives on: 'pci', 'usb', 'acpi', 'simplebus', etc.
 * 'nexus' is the machine-specific root bus and is rarely what you want for ordinary
↪ drivers.
 */
```

Parameters (order matters):

- **mydriver** - the driver name (shows up in logs and as the base of the unit name, like mydriver0).
- **pci** - the **parent bus** where you attach (choose what matches your hardware: pci, usb, acpi, simplebus, ...).
- **mydriver_driver** - your driver_t (declares method table and softc size).
- **NULL** - optional **module event handler** (called on MOD_LOAD/MOD_UNLOAD; use NULL unless you need module-level init).
- **NULL** - optional **argument** passed to that event handler (use NULL when the handler is NULL).

When to keep it minimal

Early in this chapter we are focused on the skeleton. Passing NULL for both the event handler and its argument keeps things simple. **Note:** pick the real parent bus for your device; nexus is the root bus and almost never the right choice for ordinary drivers.

Historical note (pre-FreeBSD 13)

Older code you may find online sometimes shows a six-argument form such as DRIVER_MODULE(name, bus, driver, devclass, evh, arg) together with a separate devclass_t variable. Modern FreeBSD manages devclasses automatically, and the macro now takes exactly five arguments as shown above. If you copy a legacy example, drop the extra devclass argument before building.

What DRIVER_MODULE actually accomplishes

- Registers your driver with Newbus under a parent bus.
- Exposes your method table and softc size via driver_t.
- Ensures the loader knows how to **match** devices discovered on that bus with your driver.

6.6.3.2 2) MODULE_VERSION - tag your module with a version

```
MODULE_VERSION(mydriver, 1);
```

This stamps the module with a simple integer version.

Why it matters

- The kernel and other modules can check your version to **avoid mismatches**.
- If you make a breaking change to the module ABI or exported symbols, **bump** this number.

Convention: start at 1 and bump only when something external would break if an older version was loaded.

6.6.3.3 3) MODULE_DEPEND - declare dependencies (when you have them)

```
/* mydriver requires the USB stack to be present */
MODULE_DEPEND(mydriver, usb, 1, 1, 1);
```

Parameters:

- **mydriver** - your module.
- **usb** - the module you depend on.
- **1, 1, 1** - **min, preferred, max** versions of the dependency (all 1 is common when there's no nuanced versioning to enforce).

When to use

- Your driver needs another module to be loaded **first** (e.g., `usb`, `pci`, or a helper library module).
- You export or consume symbols that require consistent versions across modules.

6.6.3.4 Mental model

- `DRIVER_MODULE` tells Newbus **who you are** and **where you plug in**.
- `MODULE_VERSION` helps the loader keep **compatible** pieces together.
- `MODULE_DEPEND` ensures modules load in the **right order** so your symbols and subsystems are ready when your driver starts.

What you'll write now vs later

For the minimal driver skeleton in this chapter, you'll almost always include `DRIVER_MODULE` and `MODULE_VERSION`.

Add `MODULE_DEPEND` when you actually rely on another module; we'll introduce common dependencies (and when they're required) in later chapters for PCI/USB/ACPI/SoC buses.

6.6.4 Finding Your State and Speaking Clearly

Two patterns appear in nearly every driver function: retrieving your softc and logging messages.

Retrieving state: `device_get_softc()`

```
static int
mydriver_attach(device_t dev)
{
    struct mydriver_softc *sc;

    sc = device_get_softc(dev); // Get our private data

    // Now use sc-> for everything
    sc->dev = dev;
    sc->some_flag = 1;
}
```

This is your **first line** in almost every driver function. It connects the `device_t` the kernel gave you to your private state.

Logging: `device_printf()`

When your driver needs to log information, use `device_printf()`:

```
device_printf(dev, "Driver attached successfully\n");
device_printf(dev, "Hardware version: %d.%d\n", major, minor);
```

Why `device_printf` instead of regular `printf`?

- It **prefixes** output with your device name: `mydriver0: Driver attached successfully`
- Users immediately know **which device** is talking
- Essential when multiple instances exist (`mydriver0`, `mydriver1`, ...)

Example output:

```
em0: Intel PRO/1000 Network Connection 7.6.1-k
em0: Link is Up 1000 Mbps Full Duplex
```

Logging etiquette (we'll expand on this in the "Logging, Errors, and User-Facing Behaviour" section):

- **Attach**: Log one line on successful attach
- **Errors**: Always log why something failed
- **Verbose info**: Only during boot or when debugging
- **Avoid spam**: Don't log on every packet/interrupt (use counters instead)

Good example:

```
if (error != 0) {
    device_printf(dev, "Could not allocate memory resource\n");
    return (error);
}
device_printf(dev, "Attached successfully\n");
```

Bad example:

```
printf("Attaching...\n"); // No device name!
printf("Step 1\n");       // Too verbose
printf("Step 2\n");       // User doesn't care
```

6.6.5 Build & Load a Stub Safely (Preview Only)

We won't build a complete driver yet (that's Chapter 7 and Lab 2), but let's preview the **build and load cycle** so you know what's coming.

The minimal Makefile:

```
# Makefile
KMOD=    mydriver
SRCS=    mydriver.c

.include <bsd.kmod.mk>
```

That's it. FreeBSD's kernel module build system (`bsd.kmod.mk`) handles all the complexity.

Build:

```
% make clean
% make
```

This produces `mydriver.ko` (kernel object file).

Load:

```
% sudo kldload ./mydriver.ko
```

Verify:

```
% kldstat | grep mydriver
% dmesg | tail
```

Unload:

```
% sudo kldunload mydriver
```

What happens behind the scenes:

1. `kldload` reads your `.ko` file
2. Kernel resolves symbols and links it into the kernel
3. Kernel calls your module event handler with `MOD_LOAD`
4. If you registered devices/drivers, they're now available
5. Newbus may immediately probe/attach if devices are present

On unload:

1. Kernel checks if it's safe to unload (no devices attached, no active users)
2. Calls your module event handler with `MOD_UNLOAD`
3. Unlinks the code from the kernel
4. Frees the module

Safety note: In your lab VM, loading/unloading is safe. If your code crashes the kernel, the VM reboots, no harm done. **Never test new drivers on production systems.**

Lab preview: In the “Hands-On Labs” section, Lab 2 will walk you through building and loading a minimal module that just logs messages. For now, just know this is the cycle you’ll follow.

Summary

The minimal driver skeleton includes:

1. **device_t** - Opaque handle to your device
2. **softc structure** - Your per-device private data
3. **Method table** - Maps kernel method calls to your functions
4. **DRIVER_MODULE** - Registers your driver with the kernel
5. **MODULE_VERSION** - Declares your version
6. **device_get_softc()** - Retrieves your state in every function
7. **device_printf()** - Logs messages with device name prefix

This pattern appears in every FreeBSD driver. Master it, and you can read any driver code with confidence.

Next, we’ll explore the **Newbus lifecycle**, when and why each of these methods is called.

6.7 The Newbus Lifecycle: From Discovery to Goodbye

You’ve seen the skeleton (probe, attach, detach functions). Now let’s understand **when** and **why** the kernel calls them. The Newbus device lifecycle is a precisely orchestrated sequence, and knowing this flow is essential for writing correct initialization and cleanup code.

Think of it like the lifecycle of a restaurant: there’s a specific order to opening (inspect location, attach utilities, set up kitchen), operating (serve customers), and closing (clean up, turn off equipment, detach utilities). Drivers follow a similar lifecycle, and understanding the sequence helps you write robust code.

6.7.1 Where Enumeration Comes From

Before your driver ever runs, **hardware must be discovered**. This is called **enumeration**, and it’s the job of **bus drivers**.

How buses discover devices

PCI bus: Reads configuration space at every bus/device/function address. When it finds a responding device, it reads vendor ID, device ID, class code, and resource requirements (memory BARs, IRQ lines).

USB bus: When you plug in a device, the hub detects electrical changes, issues a USB reset, and queries the device descriptor to learn what it is.

ACPI bus: Parses tables provided by BIOS/UEFI that describe platform devices (UARTs, timers, embedded controllers, etc.).

Device tree (ARM/embedded): Reads a devicetree blob (DTB) that statically describes hardware layout.

Key insight: Your driver doesn’t search for devices. Devices are brought to you by bus drivers. You react to what the kernel presents.

The enumeration result

For each discovered device, the bus creates a `device_t` structure containing:

- Device name (e.g., pci0:0:2:0)
- Parent bus
- Vendor/device IDs or compatible strings
- Resource requirements

See it yourself:

```
% devinfo -v          # View device tree
% pciconf -lv         # PCI devices with vendor/device IDs
% sudo usbconfig dump_device_desc  # USB device descriptors
```

Timing: Enumeration happens during boot for built-in devices, or dynamically when you plug in hot-pluggable hardware (USB, Thunderbolt, PCIe hot-plug, etc).

6.7.2 probe: “Am I Your Driver?”

Once a device exists, the kernel needs to find the right driver for it. It does this by calling every compatible driver’s **probe** function.

The probe signature:

```
static int
mydriver_probe(device_t dev)
{
    /* Examine the device and decide if we can handle it */

    /* If yes: */
    device_set_desc(dev, "My Awesome Hardware");
    return (BUS_PROBE_DEFAULT);

    /* If no: */
    return (ENXIO);
}
```

Your job in probe:

1. **Examine device properties** (vendor/device ID, compatible string, etc.)
2. **Decide if you can handle it**
3. **Return a priority value** or error

Example: PCI driver probe

```
static int
mydriver_probe(device_t dev)
{
```

```

uint16_t vendor = pci_get_vendor(dev);
uint16_t device = pci_get_device(dev);

if (vendor == MY_VENDOR_ID && device == MY_DEVICE_ID) {
    device_set_desc(dev, "My PCI Device");
    return (BUS_PROBE_DEFAULT);
}

return (ENXIO); /* Not our device */
}

```

Probe return values and priority (from `/usr/src/sys/sys/bus.h`):

Return Value	Numeric Value	Meaning
BUS_PROBE_SPECIFIC	0	Exactly matches this device variant
BUS_PROBE_VENDOR	-10	Vendor-supplied driver
BUS_PROBE_DEFAULT	-20	Standard driver for this device class
BUS_PROBE_LOW_PRIORITY	-40	Works, but something else is probably better
BUS_PROBE_GENERIC	-100	Generic fallback (e.g., class-level match)
BUS_PROBE_HOOVER	-1000000	Catch-all for devices with no real driver (ugen)
BUS_PROBE_NOWILDCARD	-2000000000	Only attach when the parent asks for me by name
ENXIO	6 (positive)	Not our device

Closer to zero wins. All of these priorities (except ENXIO) are zero or negative, and Newbus picks the driver whose return value is the **greatest** one, which means the one least negative, that is, the most specific match. BUS_PROBE_SPECIFIC (0) beats everything; BUS_PROBE_DEFAULT (-20) beats BUS_PROBE_GENERIC (-100); and anything non-negative is treated as an error.

Why this matters: The priority scheme lets a specialised driver override a generic one without either of them knowing about the other. A vendor-optimised driver returning BUS_PROBE_VENDOR (-10) will beat a base-OS driver returning BUS_PROBE_DEFAULT (-20) for the same device.

Rules for probe:

- **Do:** Examine device properties
- **Do:** Set a descriptive device description with `device_set_desc()`
- **Do:** Return quickly (no long initialization)
- **Don't:** Modify hardware state
- **Don't:** Allocate resources (wait for attach)
- **Don't:** Assume you'll win (another driver might beat you)

Real example from `/usr/src/sys/dev/uart/uart_bus_pci.c`:

```

static int
uart_pci_probe(device_t dev)
{
    struct uart_softc *sc;
    const struct pci_id *id;
    struct pci_id cid = {
        .regshift = 0,
        .rclk = 0,
        .rid = 0x10 | PCI_NO_MSI,
        .desc = "Generic SimpleComm PCI device",
    };
    int result;

    sc = device_get_softc(dev);

    id = uart_pci_match(dev, pci_ns8250_ids);
    if (id != NULL) {
        sc->sc_class = &uart_ns8250_class;
        goto match;
    }
    if (pci_get_class(dev) == PCIC_SIMPLECOMM &&
        pci_get_subclass(dev) == PCIS_SIMPLECOMM_UART &&
        pci_get_progif(dev) < PCIP_SIMPLECOMM_UART_16550A) {
        /* XXX rclk what to do */
        id = &cid;
        sc->sc_class = &uart_ns8250_class;
        goto match;
    }
    /* Add checks for non-ns8250 IDs here. */
    return (ENXIO);

match:
    result = uart_bus_probe(dev, id->regshift, 0, id->rclk,
        id->rid & PCI_RID_MASK, 0, 0);
    /* Bail out on error. */
    if (result > 0)
        return (result);
    /*
     * If we haven't already matched this to a console, check if it's a
     * PCI device which is known to only exist once in any given system
     * and we can match it that way.
     */
    if (sc->sc_sysdev == NULL)
        uart_pci_unique_console_match(dev);
}

```

```
    /* Set/override the device description. */
    if (id->desc)
        device_set_desc(dev, id->desc);
    return (result);
}
```

What happens after probe: The kernel collects all successful probe results, sorts by priority, and selects the winner. That driver's attach function will be called next.

6.7.3 attach: "Prepare to Operate"

If your probe function won, the kernel calls your **attach** function. This is where **real initialization** happens.

The attach signature:

```
static int
mydriver_attach(device_t dev)
{
    struct mydriver_softc *sc;
    int error;

    sc = device_get_softc(dev);
    sc->dev = dev;

    /* Initialization steps go here */

    device_printf(dev, "Attached successfully\n");
    return (0); /* Success */
}
```

Typical attach flow:

Step 1: Get your_softc

```
struct mydriver_softc *sc = device_get_softc(dev);
sc->dev = dev; /* Store back-pointer */
```

Step 2: Allocate hardware resources

```
sc->mem_rid = PCIR_BAR(0);
sc->mem_res = bus_alloc_resource_any(dev, SYS_RES_MEMORY,
    &sc->mem_rid, RF_ACTIVE);
if (sc->mem_res == NULL) {
    device_printf(dev, "Could not allocate memory\n");
}
```

```
    return (ENXIO);  
}
```

Step 3: Initialize hardware

```
/* Reset hardware */  
/* Configure registers */  
/* Detect hardware capabilities */
```

Step 4: Set up interrupts (if needed)

```
sc->irq_rid = 0;  
sc->irq_res = bus_alloc_resource_any(dev, SYS_RES_IRQ,  
    &sc->irq_rid, RF_ACTIVE | RF_SHAREABLE);  
  
// Placeholder - interrupt handler implementation covered in Chapter 19  
error = bus_setup_intr(dev, sc->irq_res, INTR_TYPE_NET | INTR_MPSAFE,  
    NULL, mydriver_intr, sc, &sc->irq_hand);
```

Step 5: Create device nodes or register with subsystems

```
/* Character device: */  
sc->cdev = make_dev(&mydriver_cdevsw, unit,  
    UID_ROOT, GID_WHEEL, 0600, "mydriver%d", unit);  
  
/* Network interface: */  
ether_ifattach(ifp, sc->mac_addr);  
  
/* Storage: */  
/* Register with CAM or GEOM */
```

Step 6: Mark device ready

```
device_printf(dev, "Successfully attached\n");  
return (0);
```

Error handling is critical: If any step fails, you must clean up **everything** you've already done:

```
static int  
mydriver_attach(device_t dev)  
{  
    struct mydriver_softc *sc;
```

```
int error;

sc = device_get_softc(dev);

/* Step 1 */
sc->mem_res = bus_alloc_resource_any(...);
if (sc->mem_res == NULL) {
    error = ENXIO;
    goto fail;
}

/* Step 2 */
error = mydriver_hw_init(sc);
if (error != 0)
    goto fail;

/* Step 3 */
sc->irq_res = bus_alloc_resource_any(...);
if (sc->irq_res == NULL) {
    error = ENXIO;
    goto fail;
}

/* Success! */
return (0);

fail:
mydriver_detach(dev); /* Clean up partial state */
return (error);
}
```

Why jump to `fail` and call `detach`? Because `detach` is designed to clean up resources. By calling it on failure, you reuse cleanup logic instead of duplicating it.

6.7.4 `detach` and shutdown: “Leave No Footprints”

When your driver is unloaded or the device is removed, the kernel calls your **`detach`** function to cleanly shut down.

The `detach` signature:

```
static int
mydriver_detach(device_t dev)
{
```

```
struct mydriver_softc *sc;

sc = device_get_softc(dev);

/* Cleanup steps in reverse order of attach */

device_printf(dev, "Detached\n");
return (0);
}
```

Typical detach flow (reverse of attach):

Step 1: Check if safe to detach

```
if (sc->open_count > 0) {
    return (EBUSY); /* Device is in use, can't detach now */
}
```

Step 2: Stop hardware

```
mydriver_hw_stop(sc); /* Disable interrupts, stop DMA, reset chip */
```

Step 3: Teardown interrupts

```
if (sc->irq_hand != NULL) {
    bus_teardown_intr(dev, sc->irq_res, sc->irq_hand);
    sc->irq_hand = NULL;
}
if (sc->irq_res != NULL) {
    bus_release_resource(dev, SYS_RES_IRQ, sc->irq_rid, sc->irq_res);
    sc->irq_res = NULL;
}
```

Step 4: Destroy device nodes or unregister

```
if (sc->cdev != NULL) {
    destroy_dev(sc->cdev);
    sc->cdev = NULL;
}
/* or */
ether_ifdetach(ifp);
```

Step 5: Release hardware resources

```
if (sc->mem_res != NULL) {
    bus_release_resource(dev, SYS_RES_MEMORY, sc->mem_rid, sc->mem_res);
    sc->mem_res = NULL;
}
```

Step 6: Free other allocations

```
if (sc->buffer != NULL) {
    free(sc->buffer, M_DEVBUF);
    sc->buffer = NULL;
}
mtx_destroy(&sc->mtx);
```

Critical rules:

- **Do:** Release resources in reverse order of allocation
- **Do:** Always check pointers before freeing (detach might be called on partial attach)
- **Do:** Set pointers to NULL after freeing
- **Don't:** Access hardware after stopping it
- **Don't:** Free resources still in use

The shutdown method:

Some drivers also implement a shutdown method for graceful system shutdown:

```
static int
mydriver_shutdown(device_t dev)
{
    struct mydriver_softc *sc = device_get_softc(dev);

    /* Put hardware in a safe state for reboot */
    mydriver_hw_shutdown(sc);

    return (0);
}
```

Add to method table:

```
DEVMETHOD(device_shutdown, mydriver_shutdown),
```

This is called when the system reboots or powers down, allowing your driver to gracefully stop hardware.

6.7.5 The Failure-Unwinding Pattern

We've seen hints of this, but let's make it explicit. **Failure unwinding** is a reusable pattern for handling partial attach failures.

The pattern:

```
static int
mydriver_attach(device_t dev)
{
    struct mydriver_softc *sc;
    int error = 0;

    sc = device_get_softc(dev);
    sc->dev = dev;

    /* Initialize mutex */
    mtx_init(&sc->mtx, "mydriver", NULL, MTX_DEF);

    /* Allocate resource 1 */
    sc->mem_res = bus_alloc_resource_any(...);
    if (sc->mem_res == NULL) {
        error = ENXIO;
        goto fail_mtx;
    }

    /* Allocate resource 2 */
    sc->irq_res = bus_alloc_resource_any(...);
    if (sc->irq_res == NULL) {
        error = ENXIO;
        goto fail_mem;
    }

    /* Initialize hardware */
    error = mydriver_hw_init(sc);
    if (error != 0)
        goto fail_irq;

    /* Success! */
    device_printf(dev, "Attached\n");
    return (0);

    /* Cleanup labels in reverse order */
fail_irq:
    bus_release_resource(dev, SYS_RES_IRQ, sc->irq_rid, sc->irq_res);
fail_mem:
```

```
    bus_release_resource(dev, SYS_RES_MEMORY, sc->mem_rid, sc->mem_res);
fail_mtx:
    mtx_destroy(&sc->mtx);
    return (error);
}
```

Why this works:

- Each goto jumps to the right cleanup level
- Resources are released in reverse order
- No resource is left dangling
- Code is readable and maintainable

Alternative pattern

Call detach on failure:

```
fail:
    mydriver_detach(dev);
    return (error);
}
```

This works if your detach function checks pointers before freeing (it should!).

6.7.6 Watching the Lifecycle in Logs

The best way to understand the lifecycle is to **see it happen**. FreeBSD's logging makes this easy.

Watch in real-time:

Terminal 1:

```
% tail -f /var/log/messages
```

Terminal 2:

```
% sudo kldload if_em
% sudo kldunload if_em
```

What you'll see:

```
Oct 14 12:34:56 freebsd kernel: em0: <Intel(R) PRO/1000 Network Connection> port
↪ 0xc000-0xc01f mem 0xf0000000-0xf001ffff at device 2.0 on pci0
Oct 14 12:34:56 freebsd kernel: em0: Ethernet address: 00:0c:29:3a:4f:1e
Oct 14 12:34:56 freebsd kernel: em0: netmap queues/slots: TX 1/1024, RX 1/1024
```

The first line comes from the driver's attach function. You can see it detected the device, allocated resources, and initialized.

On unload:

```
Oct 14 12:35:10 freebsd kernel: em0: detached
```

Using dmesg:

```
% dmesg | grep em0
```

This shows all kernel messages related to em0 since boot.

Using devmatch:

FreeBSD's devmatch utility shows unattached devices and suggests drivers:

```
% devmatch
```

Example output:

```
pci0:0:2:0 needs if_em
```

Exercise: Load and unload a simple driver while watching logs. Try:

```
% sudo kldload null
% dmesg | tail
% kldstat | grep null
% sudo kldunload null
```

You won't see much from null (it's quiet), but the kernel confirms load/unload.

Summary

The Newbus lifecycle follows a strict sequence:

1. **Enumeration:** Bus drivers discover hardware and create device_t structures
2. **Probe:** Kernel asks drivers "Can you handle this?" via probe functions
3. **Driver selection:** Best match wins based on priority return values
4. **Attach:** Winner's attach function initializes hardware and resources
5. **Operation:** Device is ready for use (read/write, transmit/receive, etc.)
6. **Detach:** Driver cleanly shuts down and releases all resources
7. **Destruction:** Kernel frees device_t after successful detach

Key patterns:

- Probe: Examine only, don't modify

- **Attach:** Initialize everything, handle failures with cleanup jumps
- **Detach:** Reverse order of attach, check all pointers, set to NULL

Next, we'll explore character device entry points, including how your driver handles open, read, write, and ioctl operations.

6.8 Character Device Entry Points: Your I/O Surface

Now that you understand how drivers attach and detach, let's look at how they actually **do work**. For character devices, this means implementing the **cdevsw** (character device switch), a structure that routes user-space system calls to your driver functions.

Think of cdevsw as a **menu of services** your driver offers. When a program opens `/dev/yourdevice` and calls `read()`, the kernel looks up your driver's `d_read` function and calls it. This section shows you how that routing works.

6.8.1 cdev and cdevsw: The Routing Table

Two related structures power character device operations:

- **struct cdev** - Represents a character device instance
- **struct cdevsw** - Defines the operations your driver supports

The cdevsw structure (from `/usr/src/sys/sys/conf.h`):

```
struct cdevsw {
    int            d_version;    /* Always D_VERSION */
    u_int          d_flags;     /* Device flags */
    const char     *d_name;     /* Base device name */

    d_open_t       *d_open;     /* Open handler */
    d_close_t      *d_close;    /* Close handler */
    d_read_t       *d_read;     /* Read handler */
    d_write_t      *d_write;    /* Write handler */
    d_ioctl_t      *d_ioctl;    /* Ioctl handler */
    d_poll_t       *d_poll;     /* Poll/select handler */
    d_mmap_t       *d_mmap;     /* Mmap handler */
    d_strategy_t   *d_strategy; /* (Deprecated) */
    dumper_t       *d_dump;     /* Crash dump handler */
    d_kqfilter_t   *d_kqfilter; /* Kqueue filter */
    d_purge_t      *d_purge;    /* Purge handler */
    /* ... additional fields for advanced features ... */
};
```

Minimal example from `/usr/src/sys/dev/null/null.c`:

```
static struct cdevsw null_cdevsw = {
    .d_version =    D_VERSION,
    .d_read =      (d_read_t *)nullop,
    .d_write =      null_write,
    .d_ioctl =      null_ioctl,
    .d_name =       "null",
};
```

Notice what's missing: no `d_open`, no `d_close`, no `d_poll`, no `d_kqfilter`. If you don't implement a method, the kernel provides sensible defaults:

- Missing `d_open` -> Always succeeds
- Missing `d_close` -> Always succeeds
- Missing `d_read` -> Returns EOF (0 bytes)
- Missing `d_write` -> Returns ENODEV error

Why this works: Most simple devices don't need complex open/close logic. Implement only what you need.

6.8.2 open/close: Sessions and Per-open State

When a user program opens your device, the kernel calls your `d_open` function. This is your opportunity to initialize per-open state, check permissions, or reject the open if conditions aren't right.

The `d_open` signature:

```
typedef int d_open_t(struct cdev *dev, int oflags, int devtype, struct thread *td);
```

Parameters:

- `dev` - Your `cdev` structure
- `oflags` - Open flags (`O_RDONLY`, `O_RDWR`, `O_NONBLOCK`, etc.)
- `devtype` - Device type (usually ignored)
- `td` - Thread performing the open

Typical open function:

```
static int
mydriver_open(struct cdev *dev, int oflags, int devtype, struct thread *td)
{
    struct mydriver_softc *sc;

    sc = dev->si_drv1; /* Get_softc from cdev back-pointer */

    /* Check if already open (if exclusive access needed) */
    if (sc->flags & MYDRV_OPEN) {
```

```

        return (EBUSY);
    }

    /* Mark as open */
    sc->flags |= MYDRV_OPEN;
    sc->open_count++;

    device_printf(sc->dev, "Device opened\n");
    return (0);
}

```

The `d_close` signature:

```
typedef int d_close_t(struct cdev *dev, int fflag, int devtype, struct thread *td);
```

Typical close function:

```

static int
mydriver_close(struct cdev *dev, int fflag, int devtype, struct thread *td)
{
    struct mydriver_softc *sc;

    sc = dev->si_drv1;

    /* Clean up per-open state */
    sc->flags &= ~MYDRV_OPEN;
    sc->open_count--;

    device_printf(sc->dev, "Device closed\n");
    return (0);
}

```

When to use open/close:

- **Initialize per-session state** (buffers, cursors)
- **Enforce exclusive access** (only one opener at a time)
- **Reset hardware state** on open/close
- **Track usage** for debugging

When you can skip them:

- Device doesn't need setup on open
- Hardware is always ready (like `/dev/null`)

6.8.3 read/write: Moving Bytes Safely

Read and write are the heart of data transfer for character devices. The kernel provides a **uio (user I/O) structure** to abstract the buffer and handle copying safely between kernel and user space.

The **d_read** signature:

```
typedef int d_read_t(struct cdev *dev, struct uio *uio, int ioflag);
```

The **d_write** signature:

```
typedef int d_write_t(struct cdev *dev, struct uio *uio, int ioflag);
```

Parameters:

- dev - Your cdev
- uio - User I/O structure (describes buffer, offset, remaining bytes)
- ioflag - I/O flags (IO_NDELAY for non-blocking, etc.)

Simple read example:

```
static int
mydriver_read(struct cdev *dev, struct uio *uio, int ioflag)
{
    struct mydriver_softc *sc;
    char data[128];
    int error, len;

    sc = dev->si_drv1;

    /* How much does user want? */
    len = MIN(uio->uio_resid, sizeof(data));
    if (len == 0)
        return (0); /* EOF */

    /* Fill buffer with your data */
    snprintf(data, sizeof(data), "Hello from mydriver\\n");
    len = MIN(len, strlen(data));

    /* Copy to user space */
    error = uiomove(data, len, uio);

    return (error);
}
```

Simple write example:

```
static int
mydriver_write(struct cdev *dev, struct uio *uio, int ioflag)
{
    struct mydriver_softc *sc;
    char buffer[128];
    int error, len;

    sc = dev->si_drv1;

    /* Get write size (bounded by our buffer) */
    len = MIN(uio->uio_resid, sizeof(buffer) - 1);
    if (len == 0)
        return (0);

    /* Copy from user space */
    error = uiomove(buffer, len, uio);
    if (error != 0)
        return (error);

    buffer[len] = '\\0'; /* Null terminate if treating as string */

    /* Do something with the data */
    device_printf(sc->dev, "User wrote: %s\\n", buffer);

    return (0);
}
```

Key functions for I/O:

uiomove() - Copy between kernel buffer and user space

```
int uiomove(void *cp, int n, struct uio *uio);
```

uio_resid - Remaining bytes to transfer

```
if (uio->uio_resid == 0)
    return (0); /* Nothing to do */
```

Why uio exists

It handles:

- Multi-segment buffers (scatter-gather)
- Partial transfers
- Offset tracking
- Safe copying between kernel and user space

6.8.4 ioctl: Control Paths

Ioctl (I/O control) is the **Swiss Army knife** of device operations. It handles anything that doesn't fit read/write: configuration, querying status, triggering actions, etc.

The `d_ioctl` signature:

```
typedef int d_ioctl_t(struct cdev *dev, u_long cmd, caddr_t data,
                     int fflag, struct thread *td);
```

Parameters:

- dev - Your cdev
- cmd - Command code (user-defined constant)
- data - Pointer to data structure (already copied from user space by kernel)
- fflag - File flags
- td - Thread

Defining ioctl commands

Use the `_IO`, `_IOR`, `_IOW`, `_IOWR` macros:

```
#include <sys/ioccom.h>

/* Command with no data */
#define MYDRV_RESET      _IO('M', 0)

/* Command that reads data (kernel -> user) */
#define MYDRV_GETSTATUS  _IOR('M', 1, struct mydrv_status)

/* Command that writes data (user -> kernel) */
#define MYDRV_SETCONFIG  _IOW('M', 2, struct mydrv_config)

/* Command that does both */
#define MYDRV_EXCHANGE   _IOWR('M', 3, struct mydrv_data)
```

The `'M'` is your “magic number” (unique letter identifying your driver). Pick one not used by system ioctls.

Implementing ioctl:

```
static int
mydriver_ioctl(struct cdev *dev, u_long cmd, caddr_t data,
              int fflag, struct thread *td)
{
    struct mydriver_softc *sc;
    struct mydrv_status *status;
    struct mydrv_config *config;
```

```

sc = dev->si_drv1;

switch (cmd) {
case MYDRV_RESET:
    /* Reset hardware */
    mydriver_hw_reset(sc);
    return (0);

case MYDRV_GETSTATUS:
    /* Return status to user */
    status = (struct mydrv_status *)data;
    status->flags = sc->flags;
    status->count = sc->packet_count;
    return (0);

case MYDRV_SETCONFIG:
    /* Apply configuration */
    config = (struct mydrv_config *)data;
    if (config->speed > MAX_SPEED)
        return (EINVAL);
    sc->speed = config->speed;
    return (0);

default:
    return (ENOTTY); /* Invalid ioctl */
}
}

```

Best practices:

- Always return **ENOTTY** for unknown commands
- **Validate all input** (ranges, pointers, etc.)
- Use meaningful names for commands
- Document your ioctl interface (man page or header comments)
- Don't assume data pointers are valid (kernel already validated them)

Real example from `/usr/src/sys/dev/usb/misc/uled.c`:

```

static int
uled_ioctl(struct usb_fifo *fifo, u_long cmd, void *addr, int fflags)
{
    struct uled_softc *sc;
    struct uled_color color;
    int error;

```

```
    sc = usb_fifo_softc(fifo);
    error = 0;

    mtx_lock(&sc->sc_mtx);

    switch(cmd) {
    case ULED_GET_COLOR:
        *((struct uled_color *)addr = sc->sc_color;
        break;
    case ULED_SET_COLOR:
        color = *((struct uled_color *)addr;
        uint8_t buf[8];

        sc->sc_color.red = color.red;
        sc->sc_color.green = color.green;
        sc->sc_color.blue = color.blue;

        if (sc->sc_flags & ULED_FLAG_BLINK1) {
            buf[0] = 0x1;
            buf[1] = 'n';
            buf[2] = color.red;
            buf[3] = color.green;
            buf[4] = color.blue;
            buf[5] = buf[6] = buf[7] = 0;
        } else {
            buf[0] = color.red;
            buf[1] = color.green;
            buf[2] = color.blue;
            buf[3] = buf[4] = buf[5] = 0;
            buf[6] = 0x1a;
            buf[7] = 0x05;
        }
        error = uled_ctrl_msg(sc, UT_WRITE_CLASS_INTERFACE,
            UR_SET_REPORT, 0x200, 0, buf, sizeof(buf));
        break;
    default:
        error = ENOTTY;
        break;
    }

    mtx_unlock(&sc->sc_mtx);
    return (error);
}
```

6.8.5 poll/kqfilter: Readiness Notifications

Poll and kqfilter support **event-driven I/O**, allowing programs to wait efficiently for your device to be ready for reading or writing.

When you need these:

- Your device may not be ready immediately (hardware buffer empty/full)
- You want to support `select()`, `poll()`, or `kqueue()` system calls
- Non-blocking I/O makes sense for your device

The `d_poll` signature:

```
typedef int d_poll_t(struct cdev *dev, int events, struct thread *td);
```

Basic implementation:

```
static int
mydriver_poll(struct cdev *dev, int events, struct thread *td)
{
    struct mydriver_softc *sc = dev->si_drv1;
    int revents = 0;

    if (events & (POLLIN | POLLRDNORM)) {
        /* Check if data available for reading */
        if (sc->rx_ready)
            revents |= events & (POLLIN | POLLRDNORM);
        else
            selrecord(td, &sc->rsl); /* Register for notification */
    }

    if (events & (POLLOUT | POLLWRNORM)) {
        /* Check if ready for writing */
        if (sc->tx_ready)
            revents |= events & (POLLOUT | POLLWRNORM);
        else
            selrecord(td, &sc->wsl);
    }

    return (revents);
}
```

When hardware becomes ready, wake up waiters:

```
/* In your interrupt handler or completion routine: */
selwakeup(&sc->rsel); /* Wake readers */
selwakeup(&sc->wsel); /* Wake writers */
```

The `d_kqfilter` signature (kqueue support):

```
typedef int d_kqfilter_t(struct cdev *dev, struct knote *kn);
```

Kqueue is more complex. For beginners, **implementing poll is sufficient**. Kqueue details belong in advanced chapters.

6.8.6 mmap: When Mapping Makes Sense

Mmap allows user programs to **map device memory directly into their address space**. This is useful but advanced.

When to support mmap:

- Hardware has a large memory region (framebuffer, DMA buffers)
- Performance is critical (avoid copy overhead)
- User space needs direct access to hardware registers (dangerous!)

When NOT to support mmap:

- Security concerns (exposing kernel or hardware memory)
- Synchronization complexity (cache coherency, DMA ordering)
- It's overkill for simple devices

The `d_mmap` signature:

```
typedef int d_mmap_t(struct cdev *dev, vm_ooffset_t offset, vm_paddr_t *paddr,
                    int nprot, vm_memattr_t *memattr);
```

Basic implementation:

```
static int
mydriver_mmap(struct cdev *dev, vm_ooffset_t offset, vm_paddr_t *paddr,
             int nprot, vm_memattr_t *memattr)
{
    struct mydriver_softc *sc = dev->si_drv1;

    /* Only allow mapping hardware memory region */
    if (offset >= sc->mem_size)
        return (EINVAL);
}
```

```

    *paddr = rman_get_start(sc->mem_res) + offset;
    *memattr = VM_MEMATTR_UNCACHEABLE; /* Uncached device memory */

    return (0);
}

```

For beginners: Defer mmap implementation until you actually need it. Most drivers don't.

6.8.7 Back-pointers (si_drv1, etc.)

You've seen `dev->si_drv1` throughout this section. This is how you **store your softc pointer** in the `cdev` so you can retrieve it later.

Setting the back-pointer (in `attach`):

```

sc->cdev = make_dev(&mydriver_cdevsw, unit, UID_ROOT, GID_WHEEL,
                  0600, "mydriver%d", unit);
sc->cdev->si_drv1 = sc; /* Store our softc */

```

Retrieving it (in every entry point):

```

struct mydriver_softc *sc = dev->si_drv1;

```

Available back-pointers:

- `si_drv1` - Primary driver data (typically your `softc`)
- `si_drv2` - Secondary data (if needed)

Why not just `device_get_softc()`?

Because `cdev` entry points receive a `struct cdev *`, not a `device_t`. The `si_drv1` field is the bridge.

6.8.8 Permissions and Ownership

When creating device nodes, set appropriate permissions to balance usability and security.

make_dev parameters:

```

struct cdev *
make_dev(struct cdevsw *devsw, int unit, uid_t uid, gid_t gid,
         int perms, const char *fmt, ...);

```

Common permission patterns:

Root-only device (hardware control, dangerous operations):

```
make_dev(&mydrv_cdevsw, unit, UID_ROOT, GID_WHEEL, 0600, "mydriver%d", unit);
```

Permissions: `rw-----` (owner=root)

User-accessible read-only:

```
make_dev(&mydrv_cdevsw, unit, UID_ROOT, GID_WHEEL, 0444, "mysensor%d", unit);
```

Permissions: `r--r--r--` (everyone can read)

Group-accessible device (e.g., audio):

```
make_dev(&mydrv_cdevsw, unit, UID_ROOT, GID_OPERATOR, 0660, "myaudio%d", unit);
```

Permissions: `rw-rw----` (root and operator group)

Public device (like `/dev/null`):

```
make_dev(&mydrv_cdevsw, unit, UID_ROOT, GID_WHEEL, 0666, "mynull", unit);
```

Permissions: `rw-rw-rw-` (everyone)

Security principle: Start restrictive (0600) and only open up when necessary and safe.

Summary

Character device entry points route user-space I/O to your driver:

- **cdevsw**: Routing table mapping system calls to your functions
- **open/close**: Initialize and clean up per-session state
- **read/write**: Transfer data using `uiomove()` and `struct uio`
- **ioctl**: Configuration and control commands
- **poll/kqfilter**: Event-driven readiness notifications (advanced)
- **mmap**: Direct memory mapping (advanced, security-sensitive)
- **si_drv1**: Back-pointer to retrieve your `softc`
- **Permissions**: Set appropriate access controls with `make_dev()`

Next, we'll look at **alternative surfaces** for network and storage drivers, which present very different interfaces.

If you need a pause, this is a good place. You have just crossed the halfway point of the chapter. Everything up to here, the big picture, the driver families, the `softc` and `kobj` method tables, the Newbus lifecycle, and the full character-device I/O surface, is enough foundation to revisit later as a single unit. The sections that follow shift focus: alternative surfaces for network and storage, a safe preview of resources and registers, device-node creation and destruction, module packaging, logging, and a guided tour of real tiny drivers. If your attention is still fresh, continue straight on. If it is flagging, close the book, write one or two sentences in your lab logbook about what clicked, and come back to this marker tomorrow. Neither

choice is wrong.

6.9 Alternative Surfaces: Network and Storage (Fast Orientation)

Character devices use `/dev` and `cdevsw`. But not all drivers fit that model. Network and storage drivers integrate with different kernel subsystems, presenting alternative “surfaces” to the rest of the system. This section provides a **fast orientation**, just enough to recognize these patterns when you see them.

6.9.1 A First Look at `ifnet`

Network drivers don’t create `/dev` entries. Instead, they register **network interfaces** that appear in `ifconfig` and integrate with the network stack.

The `ifnet` structure (simplified view):

```
struct ifnet {
    char        if_xname[IFNAMSIZ]; /* Interface name (e.g., "em0") */
    u_int       if_flags;           /* Flags (UP, RUNNING, etc.) */
    int         if_mtu;             /* Maximum transmission unit */
    uint64_t    if_baudrate;        /* Link speed */
    u_char      if_addr[ETHER_ADDR_LEN]; /* Hardware address */

    /* Driver-provided methods */
    if_init_fn_t if_init;          /* Initialize interface */
    if_ioctl_fn_t if_ioctl;        /* Handle ioctl commands */
    if_transmit_fn_t if_transmit; /* Transmit a packet */
    if_qflush_fn_t if_qflush;      /* Flush transmit queue */
    /* ... many more fields ... */
};
```

Registering a network interface (in `attach`):

```
if_t ifp;

/* Allocate interface structure */
ifp = if_alloc(IFT_ETHER);
if (ifp == NULL)
    return (ENOSPC);

/* Set driver data */
if_setsoftc(ifp, sc);
if_initname(ifp, device_get_name(dev), device_get_unit(dev));

/* Set capabilities and flags */
```



```
if_setflags(ifp, IFF_BROADCAST | IFF_SIMPLEX | IFF_MULTICAST);
if_setcapabilities(ifp, IFCAP_VLAN_MTU | IFCAP_HWCSUM);

/* Provide driver methods */
if_setinitfn(ifp, mydriver_init);
if_setioctlfn(ifp, mydriver_ioctl);
if_settransmitfn(ifp, mydriver_transmit);
if_setqflushfn(ifp, mydriver_qflush);

/* Attach as Ethernet interface */
ether_ifattach(ifp, sc->mac_addr);
```

What the driver must implement:

if_init - Initialize hardware and bring interface up:

```
static void
mydriver_init(void *arg)
{
    struct mydriver_softc *sc = arg;

    /* Reset hardware */
    /* Configure MAC address */
    /* Enable interrupts */
    /* Mark interface running */

    if_setdrvflagbits(sc->ifp, IFF_DRV_RUNNING, 0);
}
```

if_transmit - Transmit a packet:

```
static int
mydriver_transmit(if_t ifp, struct mbuf *m)
{
    struct mydriver_softc *sc = if_getsoftc(ifp);

    /* Queue packet for transmission */
    /* Program DMA descriptor */
    /* Notify hardware */

    return (0);
}
```

if_ioctl - Handle configuration changes:

```
static int
mydriver_ioctl(if_t ifp, u_long command, caddr_t data)
{
    struct mydriver_softc *sc = if_getsoftc(ifp);

    switch (command) {
    case SIOCSIFFLAGS:    /* Interface flags changed */
        /* Handle up/down, promisc, etc. */
        break;
    case SIOCSIFMEDIA:    /* Media selection changed */
        /* Handle speed/duplex changes */
        break;
    /* ... many more ... */
    }
    return (0);
}
```

Receiving packets (typically in interrupt handler):

```
/* In interrupt handler when packet arrives: */
struct mbuf *m;

m = mydriver_rx_packet(sc); /* Get packet from hardware */
if (m != NULL) {
    (*ifp->if_input)(ifp, m); /* Pass to network stack */
}
```

Key difference from character devices:

- No open/close/read/write
- Packets, not byte streams
- Asynchronous transmit/receive model
- Integration with routing, firewalls, protocols

Where to learn more: Chapter 28 covers network driver implementation in depth.

6.9.2 A First Look at GEOM

Storage drivers integrate with FreeBSD's **GEOM (GEOMetry Management)** layer, a modular framework for storage transformations.

GEOM conceptual model:

```

File System (UFS/ZFS)
    ->
GEOM Consumer
    ->
GEOM Class (partition, mirror, encryption)
    ->
GEOM Provider
    ->
Disk Driver (CAM)
    ->
Hardware (AHCI, NVMe)

```

Providers and Consumers:

- **Provider:** Supplies storage (e.g., a disk: ada0)
- **Consumer:** Uses storage (e.g., a filesystem)
- **GEOM Class:** Transformation layer (partitioning, RAID, encryption)

Creating a GEOM provider:

```

struct g_provider *pp;

pp = g_new_providerf(gp, "%s", name);
pp->mediasize = disk_size;
pp->sectorsize = 512;
g_error_provider(pp, 0); /* Mark available */

```

Handling I/O requests (bio structure):

```

static void
mygeom_start(struct bio *bp)
{
    struct mygeom_softc *sc;

    sc = bp->bio_to->geom->softc;

    switch (bp->bio_cmd) {
    case BIO_READ:
        mygeom_read(sc, bp);
        break;
    case BIO_WRITE:
        mygeom_write(sc, bp);
        break;
    case BIO_DELETE: /* TRIM command */

```

```
    mygeom_delete(sc, bp);
    break;
default:
    g_io_deliver(bp, EOPNOTSUPP);
    return;
}
}
```

Completing I/O:

```
bp->bio_completed = bp->bio_length;
bp->bio_resid = 0;
g_io_deliver(bp, 0); /* Success */
```

Key differences from character devices:

- Block-oriented (not byte streams)
- Asynchronous I/O model (bio requests)
- Layered architecture (transformations stack)
- Integration with filesystems and storage stack

Where to learn more: Chapter 27 covers cover GEOM and CAM in depth.

6.9.3 Mixed Models (tun as a Bridge)

Some drivers expose **both** a control plane (character device) and a data plane (network interface or storage). This “bridge” pattern provides flexibility.

Example: tun/tap device

The tun device (network tunnel) presents:

1. **Character device** (/dev/tun0) for control and packet I/O
2. **Network interface** (tun0 in ifconfig) for kernel routing

User-space view:

```
/* Open control interface */
int fd = open("/dev/tun0", O_RDWR);

/* Configure via ioctl */
struct tuninfo info = { ... };
ioctl(fd, TUNSIFINFO, &info);

/* Read packets from network stack */
char packet[2048];
```

```
read(fd, packet, sizeof(packet));

/* Write packets to network stack */
write(fd, packet, packet_len);
```

Kernel view

The tun driver:

- Creates a /dev/tunX node (cdevsw)
- Creates a tunX network interface (ifnet)
- Routes packets between them

When the network stack has a packet for tun0:

1. Packet goes to tun driver's `if_transmit`
2. Driver queues it
3. User's `read()` on /dev/tun0 retrieves it

When user writes to /dev/tun0:

1. Driver receives data in `d_write`
2. Driver wraps it in mbuf
3. Calls `(*ifp->if_input)()` to inject into network stack

Why this pattern

- **Control plane:** Configuration, setup, teardown
- **Data plane:** High-performance packet/block transfer
- **Separation:** Clean interface boundaries

Other examples

- BPF (Berkeley Packet Filter): /dev/bpf for control, sniffs network interfaces
- TAP: Similar to TUN but operates at Ethernet layer

6.9.4 What Comes Later

This section provided **recognition-level** understanding of alternative surfaces. Full implementation comes in dedicated chapters:

Network drivers - Chapter 28

- mbuf management and packet queues
- DMA descriptor rings
- Interrupt moderation and NAPI-like polling
- Hardware offload (checksums, TSO, RSS)
- Link state management
- Media selection (speed/duplex negotiation)

Storage drivers - Chapter 27

- CAM (Common Access Method) architecture
- SCSI/ATA command handling
- DMA and scatter-gather for block I/O
- Error recovery and retries
- NCQ (Native Command Queuing)
- GEOM class implementation

For now: Just recognize that not all drivers use `cdevsw`. Some integrate with specialized kernel subsystems (network stack, storage layer) and present domain-specific interfaces.

Summary

Alternative driver surfaces:

- **Network interfaces (ifnet):** Integrate with network stack, appear in `ifconfig`
- **Storage (GEOM):** Block-oriented, layered transformations, filesystem integration
- **Mixed models:** Combine character device control plane with network/storage data plane

Key takeaway: The driver family (character, network, storage) determines which kernel subsystem you integrate with. All still follow the same Newbus lifecycle (probe/attach/detach).

Next, we'll preview **resources and registers**, the vocabulary for hardware access.

6.10 Resources and Registers: A Safe Preview

Drivers don't just manage data structures, they **talk to hardware**. This means claiming resources (memory regions, IRQs), reading/writing registers, setting up interrupts, and potentially using DMA. This section provides just enough vocabulary to recognize these patterns without drowning in implementation details. Think of it as learning to recognize tools in a workshop before you learn to use them.

6.10.1 Claiming Resources (`bus_alloc_resource_*`)

Hardware devices use **resources**: memory-mapped I/O regions, I/O ports, IRQ lines, DMA channels. Before you can use them, you must **ask the bus to allocate** them.

The allocation function:

```
struct resource *
bus_alloc_resource_any(device_t dev, int type, int *rid, u_int flags);
```

Resource types (from `/usr/src/sys/amd64/include/resource.h`, `/usr/src/sys/arm64/include/resource.h`, etc):

- `SYS_RES_MEMORY` - Memory-mapped I/O region
- `SYS_RES_IOPORT` - I/O port region (x86)
- `SYS_RES_IRQ` - Interrupt line

- SYS_RES_DRQ - DMA channel (legacy)

Example: Allocating PCI BAR 0 (memory region):

```
sc->mem_rid = PCIR_BAR(0); /* Base Address Register 0 */
sc->mem_res = bus_alloc_resource_any(dev, SYS_RES_MEMORY,
                                     &sc->mem_rid, RF_ACTIVE);

if (sc->mem_res == NULL) {
    device_printf(dev, "Could not allocate memory resource\\n");
    return (ENXIO);
}
```

Example: Allocating IRQ:

```
sc->irq_rid = 0;
sc->irq_res = bus_alloc_resource_any(dev, SYS_RES_IRQ,
                                     &sc->irq_rid, RF_ACTIVE | RF_SHAREABLE);

if (sc->irq_res == NULL) {
    device_printf(dev, "Could not allocate IRQ\\n");
    return (ENXIO);
}
```

Releasing resources (in detach):

```
if (sc->mem_res != NULL) {
    bus_release_resource(dev, SYS_RES_MEMORY, sc->mem_rid, sc->mem_res);
    sc->mem_res = NULL;
}
```

What you need to know now:

- Hardware resources must be allocated before use
- Always release them in detach
- Allocation can fail (always check return value)

Full details: Chapter 18 covers resource management, PCI configuration, and memory mapping.

6.10.2 Talking to Hardware with `bus_space`

Once you've allocated a memory resource, you need to **read and write hardware registers**. FreeBSD provides `bus_space` functions for portable MMIO (Memory-Mapped I/O) and PIO (Port I/O) access.

Why not just pointer dereferences?

Direct memory access like `*(uint32_t *)addr` doesn't work reliably because:

- Endianness varies across architectures

- Memory barriers and ordering matter
- Some architectures need special instructions

bus_space abstractions:

```
bus_space_tag_t    bst;    /* Bus space tag (method table) */
bus_space_handle_t bsh;    /* Bus space handle (mapped address) */
```

Getting bus_space handles from resource:

```
sc->bst = rman_get_bustag(sc->mem_res);
sc->bsh = rman_get_bushandle(sc->mem_res);
```

Reading registers:

```
uint32_t value;

value = bus_space_read_4(sc->bst, sc->bsh, offset);
/* _4 means 4 bytes (32 bits), offset is byte offset into region */
```

Writing registers:

```
bus_space_write_4(sc->bst, sc->bsh, offset, value);
```

Common width variants:

- bus_space_read_1 / bus_space_write_1 - 8-bit (byte)
- bus_space_read_2 / bus_space_write_2 - 16-bit (word)
- bus_space_read_4 / bus_space_write_4 - 32-bit (dword)
- bus_space_read_8 / bus_space_write_8 - 64-bit (qword)

Example: Reading hardware status register:

```
#define MY_STATUS_REG  0x00
#define MY_CONTROL_REG 0x04

/* Read status */
uint32_t status = bus_space_read_4(sc->bst, sc->bsh, MY_STATUS_REG);

/* Check a flag */
if (status & STATUS_READY) {
    /* Hardware is ready */
}

/* Write control register */
bus_space_write_4(sc->bst, sc->bsh, MY_CONTROL_REG, CTRL_START);
```


What you need to know now:

- Use `bus_space_read/write` for hardware access
- Never dereference hardware addresses directly
- Offsets are in bytes

Full details: Chapter 16 covers `bus_space` patterns, memory barriers, and register access strategies.

6.10.3 Interrupts in Two Sentences

When hardware needs attention (packet arrived, transfer complete, error occurred), it raises an **interrupt**. Your driver registers an **interrupt handler** that the kernel calls asynchronously when the interrupt fires.

Setting up an interrupt handler (implementation details in Chapter 19):

```
// Placeholder - full interrupt programming covered in Chapter 19
error = bus_setup_intr(dev, sc->irq_res,
                      INTR_TYPE_NET | INTR_MPSAFE,
                      NULL, mydriver_intr, sc, &sc->irq_hand);
```

Your interrupt handler:

```
static void
mydriver_intr(void *arg)
{
    struct mydriver_softc *sc = arg;

    /* Read interrupt status */
    /* Handle the event */
    /* Acknowledge interrupt to hardware */
}
```

Golden rule: Keep interrupt handlers **short and fast**. Defer heavy work to a taskqueue or thread.

What you need to know now:

- Interrupts are asynchronous hardware notifications
- You register a handler function
- Handler runs in interrupt context (limited what you can do)

Full details: Chapter 19 covers interrupt handling, filter vs. thread handlers, interrupt moderation, and taskqueues.

6.10.4 DMA in Two Sentences

For high-performance data transfer, hardware uses **DMA (Direct Memory Access)** to move data between memory and device without CPU involvement. FreeBSD provides `bus_dma` for safe, portable DMA setup including bounce buffers for architectures with IOMMU or DMA limitations.

Typical DMA pattern:

1. Allocate DMA-capable memory with `bus_dmamem_alloc`
2. Load buffer addresses into hardware descriptors
3. Tell hardware to start DMA
4. Hardware interrupts when done
5. Unload and free when driver detaches

What you need to know now:

- DMA = zero-copy data transfer
- Requires special memory allocation
- Architecture-dependent (`bus_dma` handles portability)

Full details: Chapter 21 covers DMA architecture, descriptor rings, scatter-gather, synchronization, and bounce buffers.

6.10.5 Concurrency Note

The kernel is **multithreaded** and **preemptible**. Your driver can be called simultaneously from:

- Multiple user processes (different threads opening your device)
- Interrupt context (hardware events)
- System threads (taskqueues, timers)

This means you need locks to protect shared state:

```
/* In softc: */
struct mtx mtx;

/* In attach: */
mtx_init(&sc->mtx, "mydriver", NULL, MTX_DEF);

/* In your functions: */
mtx_lock(&sc->mtx);
/* ... access shared state ... */
mtx_unlock(&sc->mtx);

/* In detach: */
mtx_destroy(&sc->mtx);
```

What you need to know now:

- Shared data needs protection
- Use mutexes (`MTX_DEF` for most cases)
- Acquire lock, do work, release lock
- Interrupt handlers may need special lock types

Full details: Chapter 11 covers locking strategies, lock types (mutex, sx, rm), lock ordering, deadlock prevention, and lockless algorithms.

Summary

This section previewed the vocabulary for hardware access:

- **Resources:** Allocate with `bus_alloc_resource_any()`, release in detach
- **Registers:** Access with `bus_space_read/write_N()`, never direct pointers
- **Interrupts:** Register handlers with `bus_setup_intr()`, keep them short
- **DMA:** Use `bus_dma` for zero-copy transfers (complex, covered later)
- **Locking:** Protect shared state with mutexes

Remember: This chapter is about **recognition**, not mastery. When you see these patterns in driver code, you'll know what they are. Implementation details come in dedicated chapters.

Next, we'll look at **creating and removing device nodes** in `/dev`.

6.11 Creating and Removing Device Nodes

Character devices need to appear in `/dev` so user programs can open them. This section shows you the minimal API for creating and destroying device nodes using FreeBSD's devfs.

6.11.1 `make_dev/make_dev_s`: Creating `/dev/foo`

The core function for creating device nodes is `make_dev()`:

```
struct cdev *
make_dev(struct cdevsw *devsw, int unit, uid_t uid, gid_t gid,
         int perms, const char *fmt, ...);
```

Parameters:

- `devsw` - Your character device switch (`cdevsw`)
- `unit` - Unit number (minor number)
- `uid` - Owner user ID (typically `UID_ROOT`)
- `gid` - Owner group ID (typically `GID_WHEEL`)
- `perms` - Permissions (octal, like `0600` or `0666`)
- `fmt, ...` - Printf-style device name

Example (creating `/dev/mydriver0`):

```
sc->cdev = make_dev(&mydriver_cdevsw, unit,
                  UID_ROOT, GID_WHEEL, 0600, "mydriver%d", unit);
if (sc->cdev == NULL) {
    device_printf(dev, "Failed to create device node\\n");
    return (ENOMEM);
}
```

```

}

/* Store softc pointer for retrieval in entry points */
sc->cdev->si_drv1 = sc;

```

The safer variant: `make_dev_s()`

`make_dev_s()` handles race conditions better and returns error codes:

```

struct make_dev_args mda;
int error;

make_dev_args_init(&mda);
mda.mda_devsw = &mydriver_cdevsw;
mda.mda_uid = UID_ROOT;
mda.mda_gid = GID_WHEEL;
mda.mda_mode = 0600;
mda.mda_si_drv1 = sc; /* Set back-pointer directly */

error = make_dev_s(&mda, &sc->cdev, "mydriver%d", unit);
if (error != 0) {
    device_printf(dev, "Failed to create device node: %d\\n", error);
    return (error);
}

```

When to create device nodes: Typically in your **attach** function, after hardware initialization succeeds.

6.11.2 Minors and Naming Conventions

Minor numbers identify which instance of your driver a device node represents. The kernel assigns them automatically based on the `unit` parameter you pass to `make_dev()`.

Naming conventions:

- **Single instance:** `mydriver` (no number)
- **Multiple instances:** `mydriver0`, `mydriver1`, etc.
- **Sub-devices:** `mydriver0.ctl`, `mydriver0a`, `mydriver0b`
- **Subdirectories:** Use `/` in name: `"led/%s"` creates `/dev/led/foo`

Examples from FreeBSD:

- `/dev/null`, `/dev/zero` - Single, unnumbered
- `/dev/cuau0`, `/dev/cuau1` - Serial ports, numbered
- `/dev/ada0`, `/dev/ada1` - Disks, numbered
- `/dev/pts/0` - Pseudo-terminal in subdirectory

Best practices:

- Use device number from `device_get_unit()` for consistency
- Follow established naming patterns (users expect them)
- Use descriptive names (not just `/dev/dev0`)

6.11.3 `destroy_dev`: Cleaning Up

When your driver detaches, you must remove device nodes to prevent stale entries in `/dev`.

Simple cleanup:

```
if (sc->cdev != NULL) {
    destroy_dev(sc->cdev);
    sc->cdev = NULL;
}
```

What `destroy_dev()` actually does: It removes the node from `/dev`, blocks new callers from entering any of your `cdevsw` methods, and then **waits for threads currently executing inside your `d_open`, `d_read`, `d_write`, `d_ioctl`, and friends to leave**. Open file descriptors may still exist after it returns, but the kernel guarantees none of your methods are running or will ever run again for that `cdev`. Because it can sleep, `destroy_dev()` must be called from a sleepable context and **never from inside a `d_close` handler or while holding a mutex**.

When you cannot call `destroy_dev()` directly: `destroy_dev_scheduled()`

If you need to tear the node down from a context where you cannot sleep, or from inside a `cdev` method itself, schedule the destruction instead:

```
if (sc->cdev != NULL) {
    destroy_dev_scheduled(sc->cdev); /* Schedule for destruction in a safe context */
    sc->cdev = NULL;
}
```

`destroy_dev_scheduled()` returns immediately; the kernel calls `destroy_dev()` on your behalf from a safe worker thread. For ordinary `DEVICE_DETACH` paths the plain `destroy_dev()` is the right choice and what you will use most often.

When to destroy: Always in your **detach** function, before releasing other resources that the `cdev` methods might still touch.

Complete example pattern:

```
static int
mydriver_detach(device_t dev)
{
    struct mydriver_softc *sc = device_get_softc(dev);

    /* Destroy device node first: no new or in-flight cdev methods
     * can run after this returns. */
```

```

if (sc->cdev != NULL) {
    destroy_dev(sc->cdev);
    sc->cdev = NULL;
}

/* Then release other resources */
if (sc->irq_hand != NULL)
    bus_teardown_intr(dev, sc->irq_res, sc->irq_hand);
if (sc->irq_res != NULL)
    bus_release_resource(dev, SYS_RES_IRQ, sc->irq_rid, sc->irq_res);
if (sc->mem_res != NULL)
    bus_release_resource(dev, SYS_RES_MEMORY, sc->mem_rid, sc->mem_res);

return (0);
}

```

6.11.4 devctl/devmatch: Runtime Events

FreeBSD provides **devctl** and **devmatch** for monitoring device events and matching drivers to hardware.

devctl: Event notification system

Programs can listen to `/dev/devctl` for device events:

```

% sudo service devd stop
% cat /dev/devctl
!system=DEVFS subsystem=CDEV type=CREATE cdev=mydriver0
!system=DEVFS subsystem=CDEV type=DESTROY cdev=mydriver0
...
...
--- press CTRL+C to cancel / exit , remember to restart devd ---
% sudo service devd start

```

Events your driver generates:

- Device node creation (automatically when you call `make_dev`)
- Device node destruction (automatically when you call `destroy_dev`)
- Attach/detach (via `devctl_notify`)

Manual notification (optional):

```

#include <sys/devctl.h>

/* Notify that device attached */
devctl_notify("DEVICE", "ATTACH", device_get_name(dev), device_get_nameunit(dev));

```

```
/* Notify of custom event */
char buf[128];
snprintf(buf, sizeof(buf), "status=%d", sc->status);
devctl_notify("MYDRIVER", "STATUS", device_get_nameunit(dev), buf);
```

devmatch: Automatic driver loading

The devmatch utility scans unattached devices and suggests (or loads) appropriate drivers:

```
% devmatch
kldload -n if_em
kldload -n snd_hda
```

Your driver participates automatically when you use DRIVER_MODULE correctly. The kernel's device database (generated at build time) tracks which drivers match which hardware IDs.

Summary

Creating device nodes:

- Use make_dev() or make_dev_s() in attach
- Set ownership and permissions appropriately
- Store softc back-pointer in si_drv1

Destroying device nodes:

- Use destroy_dev_sched() in detach for safety
- Always destroy before releasing other resources

Device events:

- devctl monitors create/destroy events
- devmatch auto-loads drivers for unattached devices

Next, we'll explore module packaging and the load/unload lifecycle.

6.12 Module Packaging & Lifecycle (Load, Init, Unload)

Your driver doesn't just exist in source form, it's compiled into a **kernel module** (.ko file) that can be dynamically loaded and unloaded. This section explains what a module is, how the lifecycle works, and how to handle load/unload events gracefully.

6.12.1 What a Kernel Module (.ko) Is

A **kernel module** is compiled, relocatable code that the kernel can load at runtime without rebooting. Think of it as a plugin for the kernel.

File extension: `.ko` (kernel object)

Example: `mydriver.ko`

What's inside:

- Your driver code (probe, attach, detach, entry points)
- Module metadata (name, version, dependencies)
- Symbol table (for linking with kernel symbols)
- Relocation information

How it's built:

```
% cd mydriver
% make
```

FreeBSD's build system (`/usr/src/share/mk/bsd.kmod.mk`) compiles your source and links it into a `.ko` file. When you run `make`, the installed copy at `/usr/share/mk/bsd.kmod.mk` is the one actually consulted; the two files are kept in sync by the FreeBSD build.

Why modules matter:

- **No reboot needed:** Load/unload drivers without restarting
- **Smaller kernel:** Only load drivers for hardware you have
- **Development speed:** Test changes quickly
- **Modularity:** Each driver is independent

Built-in vs. module: Drivers can be compiled directly into the kernel (monolithic) or as modules. For development and learning, **always use modules**.

6.12.2 The Module Event Handler

When a module is loaded or unloaded, the kernel calls your **module event handler** to give you a chance to initialize or clean up.

The module event handler signature:

```
typedef int (*modeventhand_t)(module_t mod, int /*modeventtype_t*/ type, void *data);
```

Event types:

- `MOD_LOAD` - Module is being loaded
- `MOD_UNLOAD` - Module is being unloaded
- `MOD_QUIESCE` - Kernel is checking if unload is safe
- `MOD_SHUTDOWN` - System is shutting down

Typical module event handler:


```
static int
mydriver_modevent(module_t mod, int type, void *data)
{
    int error = 0;

    switch (type) {
    case MOD_LOAD:
        /* Module is being loaded */
        printf("mydriver: Module loaded\\n");
        /* Initialize global state if needed */
        break;

    case MOD_UNLOAD:
        /* Module is being unloaded */
        printf("mydriver: Module unloaded\\n");
        /* Clean up global state if needed */
        break;

    case MOD_QUIESCE:
        /* Check if it's safe to unload */
        if (driver_is_busy()) {
            error = EBUSY;
        }
        break;

    case MOD_SHUTDOWN:
        /* System is shutting down */
        break;

    default:
        error = EOPNOTSUPP;
        break;
    }

    return (error);
}
```

Registering the handler (for pseudo-devices without Newbus):

```
static moduledata_t mydriver_mod = {
    "mydriver",          /* Module name */
    mydriver_modevent,   /* Event handler */
    NULL                 /* Extra data */
};
```

```
DECLARE_MODULE(mydriver, mydriver_mod, SI_SUB_DRIVERS, SI_ORDER_MIDDLE);  
MODULE_VERSION(mydriver, 1);
```

DECLARE_MODULE is the lowest-level of these macros and works for any kernel module. For character-device pseudo-drivers, the kernel also provides DEV_MODULE, a thin wrapper that expands to DECLARE_MODULE with the right subsystem and order preset. You will see DEV_MODULE(null, null_modevent, NULL); in /usr/src/sys/dev/null/null.c, for example.

For Newbus drivers: The DRIVER_MODULE macro handles most of this automatically. You typically don't need a separate module event handler unless you have global initialization beyond per-device state.

Example: Pseudo-device with module event handler

From /usr/src/sys/dev/null/null.c (simplified):

```
static int  
null_modevent(module_t mod __unused, int type, void *data __unused)  
{  
    switch(type) {  
        case MOD_LOAD:  
            if (bootverbose)  
                printf("null: <full device, null device, zero device>\n");  
            full_dev = make_dev_credf(MAKEDEV_ETERNAL_KLD, &full_cdevsw, 0,  
                NULL, UID_ROOT, GID_WHEEL, 0666, "full");  
            null_dev = make_dev_credf(MAKEDEV_ETERNAL_KLD, &null_cdevsw, 0,  
                NULL, UID_ROOT, GID_WHEEL, 0666, "null");  
            zero_dev = make_dev_credf(MAKEDEV_ETERNAL_KLD, &zero_cdevsw, 0,  
                NULL, UID_ROOT, GID_WHEEL, 0666, "zero");  
            break;  
  
        case MOD_UNLOAD:  
            destroy_dev(full_dev);  
            destroy_dev(null_dev);  
            destroy_dev(zero_dev);  
            break;  
  
        case MOD_SHUTDOWN:  
            break;  
  
        default:  
            return (EOPNOTSUPP);  
    }  
  
    return (0);  
}
```

```
...  
...  
  
DEV_MODULE(null, null_modevent, NULL);  
MODULE_VERSION(null, 1);
```

This creates `/dev/full`, `/dev/null`, and `/dev/zero` when loaded, and destroys all three when unloaded.

6.12.3 Declaring Dependencies & Versions

If your driver depends on other kernel modules, declare those dependencies explicitly so the kernel loads them in the correct order.

MODULE_DEPEND macro:

```
MODULE_DEPEND(mydriver, usb, 1, 1, 1);  
MODULE_DEPEND(mydriver, netgraph, 5, 7, 9);
```

Parameters:

- `mydriver` - Your module name
- `usb` - Module you depend on
- `1` - Minimum acceptable version
- `1` - Preferred version
- `1` - Maximum acceptable version

Why this matters

If you try to load `mydriver` without `usb` being loaded, the kernel will either:

- Auto-load `usb` first (if available)
- Refuse to load `mydriver` with an error

MODULE_VERSION macro:

```
MODULE_VERSION(mydriver, 1);
```

This declares your module's version. Increment it when you make breaking changes to interfaces that other modules might depend on.

Dependency examples:

```
/* USB device driver */  
MODULE_DEPEND(umass, usb, 1, 1, 1);  
MODULE_DEPEND(umass, cam, 1, 1, 1);
```

```
/* Network driver using Netgraph */
MODULE_DEPEND(ng_ether, netgraph, NG_ABI_VERSION, NG_ABI_VERSION, NG_ABI_VERSION);
```

When to declare dependencies:

- You call functions from another module
- You use data structures defined in another module
- Your driver won't work without another subsystem

Common dependencies:

- usb - USB subsystem
- pci - PCI bus support
- cam - Storage subsystem (CAM)
- netgraph - Network graph framework
- sound - Sound subsystem

6.12.4 kldload/kldunload Flow and Logs

Let's trace what happens when you load and unload a module.

Loading a module:

```
% sudo kldload mydriver
```

Kernel flow:

1. Reads `mydriver.ko` from filesystem
2. Verifies ELF format and signature
3. Resolves symbol dependencies
4. Links module into kernel
5. Calls module event handler with `MOD_LOAD`
6. For Newbus drivers: immediately probes for matching devices
7. If devices match: calls `attach` for each
8. Module is now active

Check if loaded:

```
% kldstat
Id Refs Address                Size Name
 1   23 0xffffffff80200000 1c6e230 kernel
 2    1 0xffffffff81e6f000   5000 mydriver.ko
```

View kernel messages:

```
% dmesg | tail -5
mydriver0: <My Awesome Driver> mem 0xf0000000-0xf0001fff irq 16 at device 2.0 on pci0
mydriver0: Hardware version 1.2
mydriver0: Attached successfully
```

Unloading a module:

```
% sudo kldunload mydriver
```

Kernel flow:

1. Calls module event handler with MOD_QUIESCE (optional check)
2. If EBUSY returned: refuses to unload
3. For Newbus drivers: calls detach for all attached devices
4. Calls module event handler with MOD_UNLOAD
5. Unlinks module from kernel
6. Frees module memory

Common unload failures:

```
% sudo kldunload mydriver
kldunload: can't unload file: Device busy
```

Why:

- Device nodes still open
- Module is depended on by other modules
- Driver returned EBUSY from detach

Force unload (dangerous, only for testing):

```
% sudo kldunload -f mydriver
```

This skips safety checks. Use only in a VM when testing!

6.12.5 Troubleshooting Loads

Problem: Module won't load

Check 1: Missing symbols

```
% sudo kldload ./mydriver.ko
link_elf: symbol usb_ifconfig undefined
```

Solution: Add `MODULE_DEPEND(mydriver, usb, 1, 1, 1)` and ensure USB module is loaded.

Check 2: Module not found

```
% sudo kldload mydriver
kldload: can't load mydriver: No such file or directory
```

Solution: Either provide full path (`./mydriver.ko`) or copy to `/boot/modules/`.

Check 3: Permission denied

```
% kldload mydriver.ko
kldload: Operation not permitted
```

Solution: Use `sudo` or become root.

Check 4: Version mismatch

```
% sudo kldload mydriver.ko
kldload: can't load mydriver: Exec format error
```

Solution: Module was compiled for different FreeBSD version. Rebuild against your running kernel.

Check 5: Duplicate symbols

```
% sudo kldload mydriver.ko
link_elf: symbol mydriver_probe defined in both mydriver.ko and olddriver.ko
```

Solution: Name collision. Unload conflicting module or rename your functions.

Debugging tips:

1. Verbose loading:

```
% sudo kldload -v mydriver.ko
```

2. Check module metadata:

```
% kldstat -v | grep mydriver
```

3. View symbols:

```
% nm mydriver.ko | grep mydriver_probe
```

4. Test in VM:

Always test new drivers in a VM, never on your main system. Crashes are expected during development!

5. Watch kernel log in real-time:

```
% tail -f /var/log/messages
```

Summary

Kernel modules:

- .ko files containing driver code
- Can be loaded/unloaded dynamically
- No reboot needed for testing

Module event handler:

- Handles MOD_LOAD, MOD_UNLOAD events
- Initialize/cleanup global state
- Can refuse unload with EBUSY

Dependencies:

- Declare with MODULE_DEPEND
- Version with MODULE_VERSION
- Kernel enforces load order

Troubleshooting:

- Missing symbols -> add dependencies
- Can't unload -> check for open devices or dependencies
- Always test in VM during development

Next, we'll discuss logging, errors, and user-facing behavior.

6.13 Logging, Errors, and User-Facing Behaviour

Your driver isn't just code, it's part of the user experience. Clear logging, consistent error reporting, and useful diagnostics separate professional drivers from amateur ones. This section covers how to be a good citizen of the FreeBSD kernel.

6.13.1 Logging Etiquette (device_printf, rate-limiting hints)

The cardinal rule: Log enough to be useful, but not so much that you spam the console or fill logs.

Use device_printf() for device-related messages:

```
device_printf(dev, "Attached successfully\\n");  
device_printf(dev, "Hardware error: status=0x%x\\n", status);
```

Output:

```
mydriver0: Attached successfully
mydriver0: Hardware error: status=0x42
```

When to log:

Attach: ONE line summarizing successful attachment

```
device_printf(dev, "Attached (hw ver %d.%d)\n", major, minor);
```

Errors: ALWAYS log failures with context

```
if (error != 0) {
    device_printf(dev, "Could not allocate IRQ: error=%d\n", error);
    return (error);
}
```

Configuration changes: Log significant state changes

```
device_printf(dev, "Link up: 1000 Mbps full-duplex\n");
device_printf(dev, "Entering power-save mode\n");
```

When NOT to log:

Per-packet/per-I/O: NEVER log on every packet or read/write

```
/* BAD: This will flood the log */
device_printf(dev, "Received packet, length=%d\n", len);
```

Verbose debugging info: Not in production code

```
/* BAD: Too verbose */
device_printf(dev, "Step 1\n");
device_printf(dev, "Step 2\n");
device_printf(dev, "Reading register 0x%x\n", reg);
```

Rate-limiting for repetitive events:

If an error can occur repeatedly (hardware timeout, overflow), rate-limit:

```
static struct timeval last_overflow_msg;

if (ppsratecheck(&last_overflow_msg, NULL, 1)) {
    /* Max once per second */
    device_printf(dev, "RX overflow (message rate-limited)\n");
}
```


Using printf vs. device_printf:

- **device_printf**: For messages about a specific device
- **printf**: For messages about module or subsystem

```
/* On module load */
printf("mydriver: version 1.2 loaded\\n");

/* On device attach */
device_printf(dev, "Attached successfully\\n");
```

Log levels (for future reference)

FreeBSD kernel doesn't have explicit log levels like syslog, but conventions exist:

- Critical errors: Always log
- Warnings: Log with “warning:” prefix
- Info: Log major state changes
- Debug: Compile-time conditional (MYDRV_DEBUG)

Example from real driver (/usr/src/sys/dev/uart/uart_core.c):

```
static void
uart_pps_print_mode(struct uart_softc *sc)
{

    device_printf(sc->sc_dev, "PPS capture mode: ");
    switch(sc->sc_pps_mode & UART_PPS_SIGNAL_MASK) {
    case UART_PPS_DISABLED:
        printf("disabled");
        break;
    case UART_PPS_CTS:
        printf("CTS");
        break;
    case UART_PPS_DCD:
        printf("DCD");
        break;
    default:
        printf("invalid");
        break;
    }
    if (sc->sc_pps_mode & UART_PPS_INVERT_PULSE)
        printf("-Inverted");
    if (sc->sc_pps_mode & UART_PPS_NARROW_PULSE)
        printf("-NarrowPulse");
```

```
printf("\n");
}
```

6.13.2 Return Codes and Conventions

FreeBSD uses standard **errno** codes for error reporting. Using them consistently makes your driver predictable and debuggable.

Common errno codes (from `<sys/errno.h>`):

Code	Value	Meaning	When to Use
0	0	Success	Operation succeeded
ENOMEM	12	Out of memory	malloc/bus_alloc_resource failed
ENODEV	19	No such device	Hardware not present/responding
EINVAL	22	Invalid argument	Bad parameter from user
EIO	5	Input/output error	Hardware communication failed
EBUSY	16	Device busy	Can't detach, resource in use
ETIMEDOUT	60	Timeout	Hardware didn't respond
ENOTTY	25	Not a typewriter	Invalid ioctl command
ENXIO	6	No such device/address	Probe rejected device

In probe:

```
if (vendor_id == MY_VENDOR && device_id == MY_DEVICE)
    return (BUS_PROBE_DEFAULT); /* Success, with priority */
else
    return (ENXIO); /* Not my device */
```

In attach:

```
sc->mem_res = bus_alloc_resource_any(...);
if (sc->mem_res == NULL)
    return (ENOMEM); /* Resource allocation failed */

error = mydriver_hw_init(sc);
if (error != 0)
    return (EIO); /* Hardware initialization failed */

return (0); /* Success */
```

In entry points (read/write/ioctl):

```
/* Invalid parameter */
if (len > MAX_LEN)
    return (EINVAL);

/* Hardware not ready */
if (!(sc->flags & FLAG_READY))
    return (ENODEV);

/* I/O error */
if (timeout)
    return (ETIMEDOUT);

/* Success */
return (0);
```

In ioctl:

```
switch (cmd) {
case MYDRV_SETSPEED:
    if (speed > MAX_SPEED)
        return (EINVAL); /* Bad parameter */
    sc->speed = speed;
    return (0);

default:
    return (ENOTTY); /* Unknown ioctl command */
}
```

Summary:

- 0 = success (always)
- Positive errno = failure
- Negative values = special meanings in some contexts (like probe priorities)

User-space sees these:

```
int fd = open("/dev/mydriver0", O_RDWR);
if (fd < 0) {
    perror("open"); /* Prints: "open: No such device" if ENODEV returned */
}
```

6.13.3 Lightweight Observability with sysctl

sysctl provides a way to expose driver state and statistics **without requiring a debugger or special tools**. It's invaluable for troubleshooting and monitoring.

Why sysctl is useful:

- Users can check driver state from shell
- Monitoring tools can scrape values
- No device open required
- Zero overhead when not accessed

Example: Exposing statistics

In softc:

```
struct mydriver_softc {
    /* ... */
    uint64_t stat_packets_rx;
    uint64_t stat_packets_tx;
    uint64_t stat_errors;
    uint32_t current_speed;
};
```

In attach, create sysctl nodes:

```
struct sysctl_ctx_list *ctx;
struct sysctl_oid *tree;

/* Get device's sysctl context */
ctx = device_get_sysctl_ctx(dev);
tree = device_get_sysctl_tree(dev);

/* Add statistics */
SYSCTL_ADD_U64(ctx, SYSCTL_CHILDREN(tree), OID_AUTO,
    "packets_rx", CTLFLAG_RD, &sc->stat_packets_rx, 0,
    "Packets received");

SYSCTL_ADD_U64(ctx, SYSCTL_CHILDREN(tree), OID_AUTO,
    "packets_tx", CTLFLAG_RD, &sc->stat_packets_tx, 0,
    "Packets transmitted");

SYSCTL_ADD_U64(ctx, SYSCTL_CHILDREN(tree), OID_AUTO,
    "errors", CTLFLAG_RD, &sc->stat_errors, 0,
    "Error count");

SYSCTL_ADD_U32(ctx, SYSCTL_CHILDREN(tree), OID_AUTO,
```

```
"speed", CTLFLAG_RD, &sc->current_speed, 0,
"Current link speed (Mbps)");
```

User access:

```
% sysctl dev.mydriver.0
dev.mydriver.0.packets_rx: 1234567
dev.mydriver.0.packets_tx: 987654
dev.mydriver.0.errors: 5
dev.mydriver.0.speed: 1000
```

Read-write sysctl (for configuration):

```
static int
mydriver_sysctl_debug(SYSCTL_HANDLER_ARGS)
{
    struct mydriver_softc *sc = arg1;
    int error, value;

    value = sc->debug_level;
    error = sysctl_handle_int(oidp, &value, 0, req);
    if (error || !req->newptr)
        return (error);

    /* Validate new value */
    if (value < 0 || value > 9)
        return (EINVAL);

    sc->debug_level = value;
    device_printf(sc->dev, "Debug level set to %d\\n", value);

    return (0);
}

/* In attach: */
SYSCTL_ADD_PROC(ctx, SYSCTL_CHILDREN(tree), OID_AUTO,
    "debug", CTLTYPE_INT | CTLFLAG_RW, sc, 0,
    mydriver_sysctl_debug, "I", "Debug level (0-9)");
```

User can change it:

```
% sysctl dev.mydriver.0.debug=3
dev.mydriver.0.debug: 0 -> 3
```

Best practices:

- Expose counters and state (read-only)
- Use clear, descriptive names
- Add description strings
- Group related sysctls under subtrees
- Don't expose sensitive data (keys, passwords)
- Don't make sysctls for every variable (only useful ones)

Cleanup: Sysctl nodes are automatically cleaned up when device detaches (if you used `device_get_sysctl_ctx()`).

Summary**Logging etiquette:**

- One line on attach, always log errors
- Never log per-packet/per-I/O
- Rate-limit repetitive messages
- Use `device_printf` for device messages

Return codes:

- 0 = success
- Standard errno codes (ENOMEM, EINVAL, EIO, etc.)
- Be consistent and predictable

sysctl observability:

- Expose statistics and state for monitoring
- Read-only for counters, read-write for config
- Zero overhead when not used
- Auto-cleanup on detach

Next, we'll take a **read-only tour of tiny real drivers** to see these patterns in practice.

6.14 Read-Only Tour of Tiny Real Drivers (FreeBSD 14.3)

Now that you understand driver structure conceptually, let's tour **real FreeBSD drivers** to see these patterns in practice. We'll examine four small, clean examples, pointing out exactly where probe, attach, entry points, and other structures live. This is **read-only**, you'll implement your own in Chapter 7. For now, **recognize and understand**.

6.14.1 Tour 1 - The canonical character trio `/dev/null`, `/dev/zero`, and `/dev/full`

Open the file:

```
% cd /usr/src/sys/dev/null
% less null.c
```

We'll walk top-to-bottom: headers -> globals -> cdevsw -> write/read/ioctl paths -> module event that creates and destroys the devfs nodes.

6.14.1.1 1) Includes + minimal globals (we'll be creating devfs nodes)

```
32: #include <sys/cdefs.h>
33: #include <sys/param.h>
34: #include <sys/system.h>
35: #include <sys/conf.h>
36: #include <sys/uio.h>
37: #include <sys/kernel.h>
38: #include <sys/malloc.h>
39: #include <sys/module.h>
40: #include <sys/disk.h>
41: #include <sys/bus.h>
42: #include <sys/filio.h>
43:
44: #include <machine/bus.h>
45: #include <machine/vmparam.h>
46:
47: /* For use with destroy_dev(9). */
48: static struct cdev *full_dev;
49: static struct cdev *null_dev;
50: static struct cdev *zero_dev;
51:
52: static d_write_t full_write;
53: static d_write_t null_write;
54: static d_ioctl_t null_ioctl;
55: static d_ioctl_t zero_ioctl;
56: static d_read_t zero_read;
57:
```

6.14.1.1.1 Headers and Global Device Pointers The null driver begins with standard kernel headers and forward declarations that establish the foundation for three related but distinct character devices.

```
#include <sys/cdefs.h>
#include <sys/param.h>
#include <sys/system.h>
#include <sys/conf.h>
#include <sys/uio.h>
#include <sys/kernel.h>
```

```
#include <sys/malloc.h>
#include <sys/module.h>
#include <sys/disk.h>
#include <sys/bus.h>
#include <sys/filio.h>

#include <machine/bus.h>
#include <machine/vmparam.h>
```

6.14.1.1.2 Header Inclusions These headers provide the kernel infrastructure needed for character device drivers:

<sys/cdefs.h> and **<sys/param.h>**: Fundamental system definitions including compiler directives, basic types, and system-wide constants. Every kernel source file includes these first.

<sys/system.h>: Core kernel functions like `printf()`, `panic()`, and `bzero()`. This is the kernel's equivalent of `<stdio.h>` in userspace.

<sys/conf.h>: Character and block device configuration structures, particularly `cdevsw` (character device switch table) and related types. This header defines the `d_open_t`, `d_read_t`, `d_write_t` function pointer types used throughout the driver.

<sys/uio.h>: User I/O operations. The `struct uio` type describes data transfers between kernel and userspace, tracking buffer location, size, and direction. The `uiomove()` function declared here performs the actual data copying.

<sys/kernel.h>: Kernel startup and module infrastructure, including module event types (`MOD_LOAD`, `MOD_UNLOAD`) and the `SYSINIT` framework for initialization ordering.

<sys/malloc.h>: Kernel memory allocation. Though this driver doesn't dynamically allocate memory, the header is included for completeness.

<sys/module.h>: Module loading and unloading infrastructure. Provides `DEV_MODULE` and related macros for registering loadable kernel modules.

<sys/disk.h> and **<sys/bus.h>**: Disk and bus subsystem interfaces. The null driver includes these for kernel dump (`DIOCSKERNELDUMP`) ioctl support.

<sys/filio.h>: File I/O control commands. Defines `FIONBIO` (set non-blocking I/O) and `FIOASYNC` (set asynchronous I/O) ioctls that the driver must handle.

<machine/bus.h> and **<machine/vmparam.h>**: Architecture-specific definitions. The `vmparam.h` header provides `ZERO_REGION_SIZE` and `zero_region`, a kernel virtual memory region pre-filled with zeros that `/dev/zero` uses for efficient reads.

```
/* For use with destroy_dev(9). */
static struct cdev *full_dev;
static struct cdev *null_dev;
static struct cdev *zero_dev;
```


6.14.1.1.3 Device Structure Pointers These three global pointers store references to the character device structures created during module load. Each pointer represents one device node in `/dev`:

full_dev: Points to the `/dev/full` device structure. This device simulates a full disk, reads succeed but writes always fail with `ENOSPC` (no space left on device).

null_dev: Points to the `/dev/null` device structure, the classic “bit bucket” that discards all written data and returns immediate end-of-file on reads.

zero_dev: Points to the `/dev/zero` device structure, which returns an infinite stream of zero bytes when read and discards writes like `/dev/null`.

The comment references `destroy_dev(9)`, indicating these pointers are needed for cleanup during module unload. The `make_dev_credf()` function called during `MOD_LOAD` returns `struct cdev *` values stored here, and `destroy_dev()` called during `MOD_UNLOAD` uses these pointers to remove the device nodes.

The static storage class limits these variables to this source file, no other kernel code can access them directly. This encapsulation prevents unintended external modification.

```
static d_write_t full_write;
static d_write_t null_write;
static d_ioctl_t null_ioctl;
static d_ioctl_t zero_ioctl;
static d_read_t zero_read;
```

6.14.1.1.4 Function Forward Declarations These forward declarations establish function signatures before the `cdevsw` structures that reference them. Each declaration uses a typedef from `<sys/conf.h>`:

d_write_t: Write operation signature: `int (*d_write)(struct cdev *dev, struct uio *uio, int ioflag)`

d_ioctl_t: Ioctl operation signature: `int (*d_ioctl)(struct cdev *dev, u_long cmd, caddr_t data, int fflag, struct thread *td)`

d_read_t: Read operation signature: `int (*d_read)(struct cdev *dev, struct uio *uio, int ioflag)`

Notice the declarations needed:

- Two write functions (`full_write`, `null_write`) because `/dev/full` and `/dev/null` behave differently on write
- Two ioctl functions (`null_ioctl`, `zero_ioctl`) because they handle slightly different ioctl commands
- One read function (`zero_read`) used by both `/dev/zero` and `/dev/full` (both return zeros)

Notably absent: no `d_open_t` or `d_close_t` declarations. These devices don’t need open or close handlers, they have no per-file-descriptor state to initialize or clean up. Opening `/dev/null` requires no setup; closing it requires no teardown. The kernel’s default handlers suffice.

Also absent: `/dev/null` doesn’t need a read function. The `cdevsw` for `/dev/null` uses `(d_read_t *)nullop`, a kernel-provided function that immediately returns success with zero bytes read, signaling end-of-file.

6.14.1.1.5 Design Simplicity This header section's simplicity reflects the devices' conceptual simplicity. Three device pointers and five function declarations are sufficient because these devices:

- Maintain no state (no per-device data structures needed)
- Perform trivial operations (reads return zeros, writes succeed or fail immediately)
- Don't interact with complex kernel subsystems

This minimal complexity makes `null.c` an ideal starting point for understanding character device drivers, the concepts are clear without excessive infrastructure.

6.14.1.2 2) `cdevsw`: wiring system calls to your driver functions

```
58: static struct cdevsw full_cdevsw = {
59:     .d_version =    D_VERSION,
60:     .d_read =     zero_read,
61:     .d_write =     full_write,
62:     .d_ioctl =     zero_ioctl,
63:     .d_name =      "full",
64: };
66: static struct cdevsw null_cdevsw = {
67:     .d_version =    D_VERSION,
68:     .d_read =       (d_read_t *)nullop,
69:     .d_write =       null_write,
70:     .d_ioctl =       null_ioctl,
71:     .d_name =        "null",
72: };
74: static struct cdevsw zero_cdevsw = {
75:     .d_version =    D_VERSION,
76:     .d_read =       zero_read,
77:     .d_write =       null_write,
78:     .d_ioctl =       zero_ioctl,
79:     .d_name =        "zero",
80:     .d_flags =       D_MMAP_ANON,
81: };
```

6.14.1.2.1 Character Device Switch Tables The `cdevsw` (character device switch) structures are the kernel's dispatch tables for character device operations. Each structure maps system call operation, `read(2)`, `write(2)` and `ioctl(2)` to driver-specific functions. The null driver defines three separate `cdevsw` structures, one for each device, allowing them to share some implementations while differing where their behavior diverges.

```
static struct cdevsw full_cdevsw = {
    .d_version =    D_VERSION,
```

```
.d_read =      zero_read,  
.d_write =     full_write,  
.d_ioctl =     zero_ioctl,  
.d_name =      "full",  
};
```

6.14.1.2.2 The /dev/full Device Switch The /dev/full device simulates a filesystem that's completely full. Its cdevsw establishes this behavior through function pointer assignments:

d_version = D_VERSION: Every cdevsw must specify this version constant, ensuring binary compatibility between the driver and the kernel's device framework. The kernel checks this field during device creation and rejects mismatched versions.

d_read = zero_read: Read operations return an infinite stream of zero bytes, identical to /dev/zero. The same function serves both devices since their read behavior is identical.

d_write = full_write: Write operations always fail with ENOSPC (no space left on device), simulating a full disk. This is the distinguishing characteristic of /dev/full.

d_ioctl = zero_ioctl: The ioctl handler processes control operations like FIONBIO (non-blocking mode) and FIOASYNC (async I/O).

d_name = "full": The device name string appears in kernel messages and identifies the device in system accounting. This string determines the device node name created in /dev.

Fields not specified (like d_open, d_close, d_poll) default to NULL, causing the kernel to use built-in default handlers. For simple devices with no state, these defaults are sufficient.

```
static struct cdevsw null_cdevsw = {  
    .d_version =    D_VERSION,  
    .d_read =      (d_read_t *)nullop,  
    .d_write =     null_write,  
    .d_ioctl =     null_ioctl,  
    .d_name =      "null",  
};
```

6.14.1.2.3 The /dev/null Device Switch The /dev/null device is the classic Unix bit bucket that discards writes and immediately signals end-of-file on reads:

d_read = (d_read_t *)nullop: The nullop function is a kernel-provided no-op that returns zero immediately, signaling end-of-file to the application. Any read(2) on /dev/null returns 0 bytes without blocking. The cast to (d_read_t *) satisfies the type checker, nullop has a generic signature that works for any device operation.

d_write = null_write: Write operations succeed immediately, updating the uio structure to indicate all data was consumed, but the data is discarded. Applications see successful writes, but nothing is stored or transmitted.

d_ioctl = null_ioctl: A separate ioctl handler from `/dev/full` and `/dev/zero` because `/dev/null` supports the `DIOCSKERNELDUMP` ioctl for kernel crash dump configuration. This ioctl removes all kernel dump devices, effectively disabling crash dumps.

```
static struct cdevsw zero_cdevsw = {
    .d_version =    D_VERSION,
    .d_read =      zero_read,
    .d_write =     null_write,
    .d_ioctl =     zero_ioctl,
    .d_name =      "zero",
    .d_flags =     D_MMAP_ANON,
};
```

6.14.1.2.4 The `/dev/zero` Device Switch The `/dev/zero` device provides an infinite source of zero bytes and discards writes:

d_read = zero_read: Returns zero bytes as fast as the application can read them. The implementation uses a pre-zeroed kernel memory region for efficiency rather than zeroing a buffer on every read.

d_write = null_write: Shares the write implementation with `/dev/null`, writes are discarded, allowing applications to measure write performance or discard unwanted output.

d_ioctl = zero_ioctl: Handles standard terminal ioctls like `FIONBIO` and `FIOASYNC`, rejecting others with `ENOIOCTL`.

d_flags = D_MMAP_ANON: This flag enables a critical optimization for memory mapping. When an application calls `mmap(2)` on `/dev/zero`, the kernel doesn't actually map the device; instead, it creates anonymous memory (memory not backed by any file or device). This behavior allows applications to use `/dev/zero` for portable anonymous memory allocation:

```
void *mem = mmap(NULL, size, PROT_READ|PROT_WRITE, MAP_PRIVATE,
    open("/dev/zero", O_RDWR), 0);
```

The `D_MMAP_ANON` flag tells the kernel to substitute anonymous memory allocation for the mapping, providing zero-filled pages without involving the device driver. This pattern was historically important before `MAP_ANON` was standardized, and remains supported for compatibility.

6.14.1.2.5 Function Sharing and Reuse Notice the strategic sharing of implementations:

zero_read: Used by both `/dev/full` and `/dev/zero` because both devices return zeros when read.

null_write: Used by both `/dev/null` and `/dev/zero` because both discard written data.

zero_ioctl: Used by both `/dev/full` and `/dev/zero` because they support the same basic ioctl operations.

null_ioctl: Used only by `/dev/null` because it alone supports kernel dump configuration.

full_write: Used only by `/dev/full` because it alone fails writes with `ENOSPC`.

This sharing eliminates code duplication while preserving behavioral differences. The three devices require only five functions total (two write, two ioctl, one read) despite having three complete `cdevsw` structures.

6.14.1.2.6 The `cdevsw` as Contract Each `cdevsw` structure defines a contract between the kernel and the driver. When userspace calls `read(fd, buf, len)` on `/dev/zero`:

1. The kernel identifies the file descriptor's associated device
2. Looks up the `cdevsw` for that device (`zero_cdevsw`)
3. Calls the function pointer in `d_read` (`zero_read`)
4. Returns the result to userspace

This indirection through function pointers enables polymorphism in C: the same system call interface invokes different implementations based on which device is accessed. The kernel doesn't need to know the specifics of `/dev/zero`, it just calls the function registered in the switch table.

6.14.1.2.7 Static Storage and Encapsulation All three `cdevsw` structures use static storage class, limiting their visibility to this source file. The structures are referenced by address during device creation (`make_dev_credf(&full_cdevsw, ...)`), but external code cannot modify them. This encapsulation ensures behavioral consistency, no other driver can accidentally override `/dev/null`'s write behavior.

6.14.1.3 3) Write paths: “discard everything” vs “no space left”

```
83: /* ARGSUSED */
84: static int
85: full_write(struct cdev *dev __unused, struct uio *uio __unused, int flags __unused)
86: {
87:
88:     return (ENOSPC);
89: }
91: /* ARGSUSED */
92: static int
93: null_write(struct cdev *dev __unused, struct uio *uio, int flags __unused)
94: {
95:     uio->uio_resid = 0;
96:
97:     return (0);
98: }
```

6.14.1.3.1 Write Operation Implementations The write functions demonstrate two contrasting approaches to handling output: unconditional failure and unconditional success with data discard. These simple implementations reveal fundamental patterns in device driver design.

```
/* ARGSUSED */
static int
full_write(struct cdev *dev __unused, struct uio *uio __unused, int flags __unused)
{

    return (ENOSPC);
}
```

6.14.1.3.2 The /dev/full Write: Simulating No Space The /dev/full write function is deliberately trivial, it immediately returns ENOSPC (error number 28, “No space left on device”) without examining its arguments or performing any operations.

Function signature: All d_write_t functions receive three parameters:

- struct cdev *dev - the device being written to
- struct uio *uio - describes the user’s write buffer (location, size, offset)
- int flags - I/O flags like O_NONBLOCK or O_DIRECT

The __unused attribute: Each parameter is marked __unused, a compiler directive indicating the parameter is intentionally ignored. This prevents “unused parameter” warnings during compilation. The directive documents that the function’s behavior doesn’t depend on which device instance is accessed, what data the user provided, or what flags were specified.

The /* ARGSUSED */ comment: This traditional lint directive predates modern compiler attributes, serving the same purpose for older static analysis tools. It signals “arguments unused by design, not by mistake.” The comment and __unused attributes are redundant but maintain compatibility with multiple code analysis tools.

Return value ENOSPC: This errno value tells userspace that the write failed because no space remains. To the application, /dev/full appears as a storage device that’s completely full. This behavior is useful for testing how programs handle write failures, many applications don’t properly check write return values, leading to silent data loss when disks fill. Testing against /dev/full exposes these bugs.

Why not process the uio?: Normal device drivers would call uiomove() to consume data from the user’s buffer and update uio->uio_resid to reflect bytes written. The /dev/full driver skips this entirely because it’s simulating a failure condition where no bytes were written. Returning an error without touching uio signals “zero bytes written, operation failed.”

Applications see:

```
ssize_t n = write(fd, buf, 100);
// n == -1, errno == ENOSPC
```

```
/* ARGSUSED */
static int
null_write(struct cdev *dev __unused, struct uio *uio, int flags __unused)
```

```
{
    uio->uio_resid = 0;

    return (0);
}
```

6.14.1.3.3 The `/dev/null` and `/dev/zero` Write: Discarding Data The `null_write` function (used by both `/dev/null` and `/dev/zero`) implements the classic bit bucket behavior: accept all data, discard everything, report success.

Marking data consumed: The single operation `uio->uio_resid = 0` is the key to this function’s behavior. The `uio_resid` field tracks how many bytes remain to be transferred. Setting it to zero tells the kernel “all requested bytes were successfully written,” even though the driver never actually accessed the user’s buffer.

Why this works: The kernel’s write system call implementation checks `uio_resid` to determine how many bytes were written. If a driver sets `uio_resid` to zero and returns success (0), the kernel calculates:

```
bytes_written = original_resid - current_resid
               = original_resid - 0
               = original_resid // all bytes written
```

The application’s `write(2)` call returns the full byte count requested, indicating complete success.

No actual data transfer: Unlike normal drivers that call `uiomove()` to copy data from userspace, `null_write` never accesses the user’s buffer. The data remains in userspace, untouched and unread. The driver simply lies about having consumed it. This is safe because the data is being discarded anyway, there’s no point copying data into kernel memory just to throw it away.

Return value zero: Returning 0 signals success. Combined with `uio_resid = 0`, this creates the illusion of a perfectly functioning write operation that accepted all data.

Why `uio` isn’t marked `__unused`: The function modifies `uio->uio_resid`, so the parameter is actively used. Only `dev` and `flags` are ignored and marked `__unused`.

Applications see:

```
ssize_t n = write(fd, buf, 100);
// n == 100, all bytes "written"
```

6.14.1.3.4 Performance Implications The `null_write` optimization is significant for performance-sensitive applications. Consider a program redirecting gigabytes of unwanted output to `/dev/null`:

```
% ./generate_logs > /dev/null
```

If the driver actually copied data from userspace (via `uiomove()`), this would waste CPU cycles and memory bandwidth copying data that’s immediately discarded. By setting `uio_resid = 0` without touching the buffer, the driver eliminates this overhead entirely. The application fills its userspace buffer, calls `write(2)`, the kernel immediately returns success, and the CPU never accesses the buffer content.

6.14.1.3.5 Contrast in Error Handling Philosophy These two functions embody different design philosophies:

full_write: Simulate a failure condition for testing purposes. Real error, immediate rejection.

null_write: Maximize performance by doing nothing. Fake success, instant return.

Both are correct implementations of their respective device semantics. The simplicity of these functions, five lines combined, demonstrates that device drivers don't need to be complex to be useful. Sometimes the best implementation is the one that does the least work necessary to satisfy the interface contract.

6.14.1.3.6 Interface Contract Satisfaction Both functions satisfy the `d_write_t` contract:

- Accept a device pointer, uio descriptor, and flags
- Return 0 for success or `errno` for failure
- Update `uio_resid` to reflect bytes consumed (or leave it unchanged if none were consumed)

The `cdevsw` function pointers enforce this contract at compile time. Any function not matching the `d_write_t` signature would cause a compilation error when assigned to `d_write` in the `cdevsw` structure. This type safety ensures all write implementations follow the same calling convention, allowing the kernel to invoke them uniformly.

6.14.1.4 4) IOCTLs: accept a tiny, sensible subset; reject the rest

```

100: /* ARGSUSED */
101: static int
102: null_ioctl(struct cdev *dev __unused, u_long cmd, caddr_t data __unused,
103:     int flags __unused, struct thread *td)
104: {
105:     struct diocskerneldump_arg kda;
106:     int error;
107:
108:     error = 0;
109:     switch (cmd) {
110:     case DIOCSKERNELDUMP:
111:         bzero(&kda, sizeof(kda));
112:         kda.kda_index = KDA_REMOVE_ALL;
113:         error = dumper_remove(NULL, &kda);
114:         break;
115:     case FIONBIO:
116:         break;
117:     case FIOASYNC:
118:         if (*(int *)data != 0)
119:             error = EINVAL;
120:         break;
121:     default:

```



```

122:     error = ENOIOCTL;
123: }
124: return (error);
125: }
127: /* ARGSUSED */
128: static int
129: zero_ioctl(struct cdev *dev __unused, u_long cmd, caddr_t data __unused,
130:     int flags __unused, struct thread *td)
131: {
132:     int error;
133:     error = 0;
134:
135:     switch (cmd) {
136:     case FIONBIO:
137:         break;
138:     case FIOASYNC:
139:         if (*(int *)data != 0)
140:             error = EINVAL;
141:         break;
142:     default:
143:         error = ENOIOCTL;
144:     }
145:     return (error);
146: }

```

6.14.1.4.1 Ioctl Operation Implementations The ioctl (I/O control) functions handle device-specific control operations beyond standard read and write. While read and write transfer data, ioctl performs configuration, status queries, and special operations. The null driver implements two ioctl handlers that differ only in their support for kernel crash dump configuration.

```

/* ARGSUSED */
static int
null_ioctl(struct cdev *dev __unused, u_long cmd, caddr_t data __unused,
    int flags __unused, struct thread *td)
{
    struct diocskerneldump_arg kda;
    int error;

    error = 0;
    switch (cmd) {
    case DIOCSKERNELDUMP:
        bzero(&kda, sizeof(kda));

```

```

        kda.kda_index = KDA_REMOVE_ALL;
        error = dumper_remove(NULL, &kda);
        break;
    case FIONBIO:
        break;
    case FIOASYNC:
        if (*(int *)data != 0)
            error = EINVAL;
        break;
    default:
        error = ENOIOCTL;
    }
    return (error);
}

```

6.14.1.4.2 The `/dev/null` Ioctl Handler **Function signature:** The `d_ioctl_t` type requires five parameters:

- `struct cdev *dev` - the device being controlled
- `u_long cmd` - the ioctl command number
- `caddr_t data` - pointer to command-specific data (in/out parameter)
- `int flags` - file descriptor flags from the original `open(2)`
- `struct thread *td` - the calling thread (for credential checks, signal delivery)

Most parameters are marked `__unused` because this simple device doesn't need per-instance state (`dev`), doesn't examine most command data (`data` for some commands), and doesn't check flags or thread credentials.

Command dispatch via switch: The function uses a `switch` statement to handle different ioctl commands, each identified by a unique constant. The pattern `switch (cmd)` followed by case labels is universal in ioctl handlers.

```

case DIOCSKERNELDUMP:
    bzero(&kda, sizeof(kda));
    kda.kda_index = KDA_REMOVE_ALL;
    error = dumper_remove(NULL, &kda);
    break;

```

6.14.1.4.3 Kernel Dump Configuration: `DIOCSKERNELDUMP` This case handles kernel crash dump configuration. When the system crashes, the kernel writes diagnostic information (memory contents, register state, stack traces) to a designated dump device, typically a disk partition or swap space. The `DIOCSKERNELDUMP` ioctl configures this dump device.

Why `/dev/null` for crash dumps?: The idiom `ioctl(fd, DIOCSKERNELDUMP, &args)` on `/dev/null` serves a specific purpose: disabling all kernel dumps. By directing dumps to the bit bucket, administrators

can prevent crash dump collection entirely (useful for security-sensitive systems or when disk space is constrained).

Preparing the argument structure: `bzero(&kda, sizeof(kda))` zeros the `diocskernel_dump_arg` structure, ensuring all fields start in a known state. This is defensive programming, uninitialized stack memory might contain random values that could confuse the dump subsystem.

Removing all dump devices: `kda.kda_index = KDA_REMOVE_ALL` sets the magic index value indicating “remove all configured dump devices, don’t add a new one.” The constant `KDA_REMOVE_ALL` signals special semantics distinct from specifying a particular device index.

Calling the dump subsystem: `dumper_remove(NULL, &kda)` invokes the kernel’s dump management function. The first parameter (`NULL`) indicates no specific device is being removed, the `kda_index` field provides the directive. The function returns 0 on success or an error code on failure.

```
case FIONBIO:
    break;
```

6.14.1.4.4 Non-Blocking I/O: FIONBIO The `FIONBIO` ioctl sets or clears non-blocking mode on the file descriptor. The data parameter points to an integer: non-zero enables non-blocking mode, zero disables it.

Why do nothing?: The handler simply breaks without performing any operation. This is correct because `/dev/null` operations never block:

- Reads immediately return end-of-file (0 bytes)
- Writes immediately succeed (all bytes consumed)

There’s no condition under which a `/dev/null` operation would block, so non-blocking mode is meaningless. The ioctl succeeds (returns 0) but has no effect, maintaining compatibility with applications that configure non-blocking mode without causing errors.

```
case FIOASYNC:
    if (*(int *)data != 0)
        error = EINVAL;
    break;
```

6.14.1.4.5 Asynchronous I/O: FIOASYNC The `FIOASYNC` ioctl enables or disables asynchronous I/O notification. When enabled, the kernel sends `SIGIO` signals to the process when the device becomes readable or writable.

Parameter interpretation: The data parameter points to an integer. Zero means disable async I/O, non-zero means enable it.

Rejecting async I/O: The handler checks if the application is trying to enable async I/O (`*(int *)data != 0`). If so, it returns `EINVAL` (invalid argument), rejecting the request.

Why reject async I/O?: Asynchronous I/O only makes sense for devices that can block. Applications enable it to receive notification when a previously-blocked operation can proceed. Since `/dev/null` never blocks, async I/O is meaningless and potentially confusing. Rather than silently accepting a nonsensical configuration, the driver returns an error, alerting the application to the logical error.

Disabling async I/O succeeds: If `*(int *)data == 0`, the condition is false, error remains 0, and the function returns success. Disabling a feature that was never enabled is harmless.

```
default:
    error = ENOIOCTL;
```

6.14.1.4.6 Unknown Commands: Default Case Any ioctl command not explicitly handled falls through to the default case, which returns `ENOIOCTL`. This special error code means “this ioctl is not supported by this device.” It’s distinct from `EINVAL` (invalid argument to a supported ioctl) and `ENOTTY` (inappropriate ioctl for device type, used for terminal operations on non-terminals).

The kernel’s ioctl infrastructure may retry the operation through other layers when receiving `ENOIOCTL`, allowing generic handlers to process common commands.

```
/* ARGSUSED */
static int
zero_ioctl(struct cdev *dev __unused, u_long cmd, caddr_t data __unused,
           int flags __unused, struct thread *td)
{
    int error;
    error = 0;

    switch (cmd) {
    case FIONBIO:
        break;
    case FIOASYNC:
        if (*(int *)data != 0)
            error = EINVAL;
        break;
    default:
        error = ENOIOCTL;
    }
    return (error);
}
```

6.14.1.4.7 The `/dev/zero` Ioctl Handler The `zero_ioctl` function is nearly identical to `null_ioctl`, with one critical difference: it doesn’t handle `DIOCSKERNELDUMP`. The `/dev/zero` device cannot serve as a kernel dump device (dumps must be stored, not discarded), so the ioctl isn’t supported.

The FIONBIO and FIOASYNC handling is identical, these are standard file descriptor ioctls that all character devices should handle consistently, even if the operations are no-ops.

6.14.1.4.8 Ioctl Design Patterns Several patterns emerge from these implementations:

Explicit handling of no-op operations: Rather than returning errors for meaningless operations like FIONBIO on /dev/null, the handlers succeed silently. This maintains compatibility with applications that unconditionally configure file descriptors without checking device type.

Rejecting nonsensical configurations: Async I/O makes no sense for these devices, so the handlers return errors when applications try to enable it. This is a design choice, the handlers could succeed silently, but explicit errors help developers identify logic bugs.

Standard error codes: EINVAL for invalid arguments, ENOIOCTL for unsupported commands. These conventions allow userspace to distinguish different failure modes.

Minimal data validation: The handlers cast data pointers and dereference them without extensive validation. This is safe because the kernel's ioctl infrastructure has already verified the pointer is accessible to userspace. Device drivers trust the kernel's argument validation.

6.14.1.4.9 Why Two Ioctl Functions? The /dev/full device uses zero_ioctl (not shown using it in the cdevsw, but by examining the structures we saw earlier). Only /dev/null needs the special dump device handling, so only null_ioctl includes the DIOCSKERNELDUMP case. This separation avoids polluting the simpler zero_ioctl with functionality that only one device needs.

The code reuse strategy: write the minimal handler (zero_ioctl), then extend it for special cases (null_ioctl). This keeps each function focused and avoids conditional logic like “if this is /dev/null, handle dumps.”

6.14.1.5 5) Read path: a simple loop driven by uio->uio_resid

```

148: /* ARGSUSED */
149: static int
150: zero_read(struct cdev *dev __unused, struct uio *uio, int flags __unused)
151: {
152:     void *zbuf;
153:     ssize_t len;
154:     int error = 0;
155:
156:     KASSERT(uio->uio_rw == UIO_READ,
157:         ("Can't be in %s for write", __func__));
158:     zbuf = __DECONST(void *, zero_region);
159:     while (uio->uio_resid > 0 && error == 0) {
160:         len = uio->uio_resid;
161:         if (len > ZERO_REGION_SIZE)
162:             len = ZERO_REGION_SIZE;
163:         error = uiomove(zbuf, len, uio);

```

```

164:     }
165:
166:     return (error);
167: }

```

6.14.1.5.1 Read Operation: Infinite Zeros The `zero_read` function provides an endless stream of zero bytes, serving both `/dev/zero` and `/dev/full`. This implementation demonstrates efficient data transfer using a pre-allocated kernel buffer and the `uiomove()` function for kernel-to-userspace copying.

```

/* ARGSUSED */
static int
zero_read(struct cdev *dev __unused, struct uio *uio, int flags __unused)
{
    void *zbuf;
    ssize_t len;
    int error = 0;

    KASSERT(uio->uio_rw == UIO_READ,
        ("Can't be in %s for write", __func__));
}

```

6.14.1.5.2 Function Structure and Safety Assertion **Function signature:** The `d_read_t` type requires the same parameters as `d_write_t`:

- `struct cdev *dev` - the device being read (unused, marked `__unused`)
- `struct uio *uio` - describes the user's read buffer and tracks transfer progress
- `int flags` - I/O flags (unused for this simple device)

Local variables: The function needs minimal state:

- `zbuf` - pointer to the source of zero bytes
- `len` - number of bytes to transfer in each iteration
- `error` - tracks success or failure of transfer operations

Sanity check with KASSERT: The assertion verifies that `uio->uio_rw` equals `UIO_READ`, confirming this is actually a read operation. The `uio` structure serves both read and write operations, with the `uio_rw` field indicating direction.

This assertion catches programming errors during development. If somehow a write operation called this read function, the assertion would trigger a kernel panic with the message “Can’t be in `zero_read` for write.” The `__func__` preprocessor macro expands to the current function name, making the error message precise.

In production kernels compiled without debugging, `KASSERT` compiles to nothing, eliminating any runtime overhead. This pattern, defensive checks during development, zero cost in production, is common throughout FreeBSD’s kernel.

```
zbuf = __DECONST(void *, zero_region);
```

6.14.1.5.3 Accessing the Pre-Zeroed Buffer The `zero_region` variable (declared in `<machine/vm-param.h>`) points to a region of kernel virtual memory that's permanently filled with zeros. The kernel allocates this region during boot and never modifies it, providing an efficient source of zero bytes without repeatedly zeroing temporary buffers.

The `__DECONST` macro: The `zero_region` is declared `const` to prevent accidental modification. However, `uiomove()` expects a non-`const` pointer because it's a generic function that handles both read (kernel to user) and write (user to kernel) operations. The `__DECONST` macro removes the `const` qualifier, essentially telling the compiler "I know this is `const`, but I need to pass it to a function expecting non-`const`. Trust me, it won't be modified."

This is safe because `uiomove()` with a read-direction `uio` only copies data from the kernel buffer to userspace, it never writes to the buffer. The `const`-cast is a necessary workaround for C's type system limitations.

```
while (uio->uio_resid > 0 && error == 0) {
    len = uio->uio_resid;
    if (len > ZERO_REGION_SIZE)
        len = ZERO_REGION_SIZE;
    error = uiomove(zbuf, len, uio);
}

return (error);
```

6.14.1.5.4 The Transfer Loop The loop continues until either the entire read request is satisfied (`uio->uio_resid == 0`) or an error occurs (`error != 0`).

Checking remaining bytes: `uio->uio_resid` tracks how many bytes the application requested but haven't yet been transferred. Initially, this equals the original read size. After each successful transfer, `uiomove()` decrements it.

Limiting transfer size: The code calculates how many bytes to transfer in this iteration:

```
len = uio->uio_resid;
if (len > ZERO_REGION_SIZE)
    len = ZERO_REGION_SIZE;
```

If the remaining request exceeds the zero region's size, the transfer is capped at `ZERO_REGION_SIZE`. This limitation exists because the kernel only pre-allocated a finite zero buffer. Typical values for `ZERO_REGION_SIZE` are 64KB or 256KB, large enough for efficiency but small enough not to waste kernel memory.

Why this matters: If an application reads 1MB from `/dev/zero`, the loop executes multiple times, each iteration transferring up to `ZERO_REGION_SIZE` bytes. The same zero buffer is reused for each iteration, eliminating the need to allocate and zero 1MB of kernel memory.

Performing the transfer: `uiomove(zbuf, len, uio)` is the kernel's workhorse function for moving data between kernel and userspace. It:

1. Copies `len` bytes from `zbuf` (kernel memory) to the user's buffer (described by `uio`)
2. Updates `uio->uio_resid` by subtracting `len` (fewer bytes remaining)
3. Advances `uio->uio_offset` by `len` (file position moves forward, though meaningless for `/dev/zero`)
4. Returns 0 on success or an error code on failure (typically `EFAULT` if the user's buffer address is invalid)

If `uiomove()` returns an error, the loop exits immediately and returns the error to the caller. The application receives whatever data was successfully transferred before the error occurred.

Loop termination: The loop exits when:

- **Success:** `uio->uio_resid` reaches zero, meaning all requested bytes were transferred
- **Error:** `uiomove()` failed, typically because the user's buffer pointer was invalid or the process received a signal

6.14.1.5.5 Infinite Stream Semantics Notice what's missing from this function: no end-of-file check. Most file reads eventually return 0 bytes, signaling EOF. The `/dev/zero` read function never does this, it always transfers the full requested amount (or fails with an error).

From userspace perspective:

```
char buf[4096];
ssize_t n = read(zero_fd, buf, sizeof(buf));
// n always equals 4096, never 0 (unless error)
```

This infinite stream property makes `/dev/zero` useful for:

- Allocating zero-initialized memory (pre-MAP_ANON)
- Generating arbitrary amounts of zero bytes for testing
- Overwriting disk blocks with zeros for data sanitization

6.14.1.5.6 Performance Optimization The pre-allocated `zero_region` is a significant optimization. Consider the alternative implementation:

```
// Inefficient approach
char zeros[4096];
bzero(zeros, sizeof(zeros));
while (uio->uio_resid > 0) {
    len = min(uio->uio_resid, sizeof(zeros));
    error = uiomove(zeros, len, uio);
}
```

This approach would zero a buffer on every function call, wasting CPU cycles. The production implementation zeros the buffer once at boot and reuses it forever, eliminating repeated zeroing overhead.

For applications reading gigabytes from `/dev/zero`, this optimization eliminates billions of store instructions, making reads essentially free (bounded only by memory copy speed).

6.14.1.5.7 Shared Between Devices Recall from the `cdevsw` structures that both `/dev/zero` and `/dev/full` use `zero_read`. This sharing is correct because both devices should return zeros when read. The device identity (dev parameter) is ignored because the behavior is identical regardless of which device is accessed.

This implementation demonstrates a key principle: when multiple devices share behavior, implement it once and reference it from multiple switch tables. Code reuse eliminates duplication and ensures consistent behavior across related devices.

6.14.1.5.8 Error Propagation If `uiomove()` fails partway through a large read, the function returns the error immediately. The userspace `read(2)` system call sees a short read followed by an error on the next call. For example:

```
// Reading 128KB when process receives signal after 64KB
char buf[128 * 1024];
ssize_t n = read(zero_fd, buf, sizeof(buf));
// n might equal 65536 (successful partial read)
// errno unset (partial success)

n = read(zero_fd, buf, sizeof(buf));
// n equals -1, errno equals EINTR (interrupted system call)
```

This error handling is automatic, `uiomove()` detects signals and returns `EINTR`, which the `read` function propagates to userspace. The driver doesn't need explicit signal handling logic.

6.14.1.6 6) Module event: create device nodes on load, destroy on unload

```
169: /* ARGSUSED */
170: static int
171: null_modevent(module_t mod __unused, int type, void *data __unused)
172: {
173:     switch(type) {
174:     case MOD_LOAD:
175:         if (bootverbose)
176:             printf("null: <full device, null device, zero device>\n");
177:         full_dev = make_dev_credf(MAKEDEV_ETERNAL_KLD, &full_cdevsw, 0,
178:             NULL, UID_ROOT, GID_WHEEL, 0666, "full");
179:         null_dev = make_dev_credf(MAKEDEV_ETERNAL_KLD, &null_cdevsw, 0,
180:             NULL, UID_ROOT, GID_WHEEL, 0666, "null");
181:         zero_dev = make_dev_credf(MAKEDEV_ETERNAL_KLD, &zero_cdevsw, 0,
182:             NULL, UID_ROOT, GID_WHEEL, 0666, "zero");
183:         break;
184:
185:     case MOD_UNLOAD:
```

```

186:     destroy_dev(full_dev);
187:     destroy_dev(null_dev);
188:     destroy_dev(zero_dev);
189:     break;
190:
191:     case MOD_SHUTDOWN:
192:         break;
193:
194:     default:
195:         return (EOPNOTSUPP);
196:     }
197:
198:     return (0);
199: }
201: DEV_MODULE(null, null_modevent, NULL);
202: MODULE_VERSION(null, 1);

```

6.14.1.6.1 Module Lifecycle and Registration The final section of the null driver handles module loading, unloading, and registration with the kernel's module system. This code executes when the module is loaded at boot or via `kldload`, and when it's unloaded via `kldunload`.

```

/* ARGSUSED */
static int
null_modevent(module_t mod __unused, int type, void *data __unused)
{
    switch(type) {

```

6.14.1.6.2 The Module Event Handler **Function signature:** Module event handlers receive three parameters:

- `module_t mod` - a handle to the module itself (unused here)
- `int type` - the event type: `MOD_LOAD`, `MOD_UNLOAD`, `MOD_SHUTDOWN`, etc.
- `void *data` - event-specific data (unused for this driver)

The function returns 0 for success or an `errno` value for failure. A failed `MOD_LOAD` prevents the module from loading; a failed `MOD_UNLOAD` keeps the module loaded.

```

case MOD_LOAD:
    if (bootverbose)
        printf("null: <full device, null device, zero device>\n");

```

```

full_dev = make_dev_credf(MAKEDEV_ETERNAL_KLD, &full_cdevsw, 0,
    NULL, UID_ROOT, GID_WHEEL, 0666, "full");
null_dev = make_dev_credf(MAKEDEV_ETERNAL_KLD, &null_cdevsw, 0,
    NULL, UID_ROOT, GID_WHEEL, 0666, "null");
zero_dev = make_dev_credf(MAKEDEV_ETERNAL_KLD, &zero_cdevsw, 0,
    NULL, UID_ROOT, GID_WHEEL, 0666, "zero");
break;

```

6.14.1.6.3 Module Load: Creating Devices The `MOD_LOAD` case executes when the module is first loaded, either during boot or when an administrator runs `kldload null`.

Boot message: The `if (bootverbose)` check controls whether a message appears during boot. The `bootverbose` variable is set when the system boots with verbose output enabled (via boot loader configuration or kernel option). When true, the driver prints an informational message identifying the devices it provides.

This conditional prevents cluttering the boot output in normal operation while allowing administrators to see driver initialization during diagnostic boots. The message format follows FreeBSD convention: driver name, colon, angle-bracketed device list.

Device creation with `make_dev_credf`: This function creates character device nodes in `/dev`. Each call requires several parameters that control the device's properties:

`MAKEDEV_ETERNAL_KLD`: A flag indicating this device should persist until explicitly destroyed. The `ETERNAL` part means the device won't be automatically removed if all references are closed, and `KLD` indicates it's part of a kernel loadable module (as opposed to a statically compiled driver). This flag combination ensures the device nodes remain available as long as the module is loaded, regardless of whether any process has them open.

`&full_cdevsw` (and similarly for `null/zero`): Pointer to the character device switch table that defines the device's behavior. This connects the device node to the driver's function implementations.

`0`: The device unit number. Since these are singleton devices (only one `/dev/null` exists system-wide), unit 0 is used. Multi-instance devices like `/dev/tty0`, `/dev/tty1` would use different unit numbers.

`NULL`: Credential pointer for permission checks. `NULL` means no special credentials are required beyond the standard file permissions.

`UID_ROOT`: The device file owner (root, UID 0). This determines who can change the device's permissions or delete it.

`GID_WHEEL`: The device file group (wheel, GID 0). The wheel group traditionally has administrative privileges.

`0666`: The permission mode in octal. This value (readable and writable by owner, group, and others) allows any process to open these devices. Breaking it down:

- Owner (root): read (4) + write (2) = 6
- Group (wheel): read (4) + write (2) = 6
- Others: read (4) + write (2) = 6

Unlike typical files where world-writable permissions are dangerous, these devices are designed for universal access, any process should be able to write to `/dev/null` or read from `/dev/zero`.

"full" (and similarly "null", "zero"): The device name string. This creates /dev/full, /dev/null, and /dev/zero respectively. The `make_dev_credf` function automatically prepends /dev/ to the name.

Return value storage: Each `make_dev_credf` call returns a `struct cdev *` pointer stored in the global variables (`full_dev`, `null_dev`, `zero_dev`). These pointers are essential for the unload handler to remove the devices later.

```
case MOD_UNLOAD:
    destroy_dev(full_dev);
    destroy_dev(null_dev);
    destroy_dev(zero_dev);
    break;
```

6.14.1.6.4 Module Unload: Destroying Devices The `MOD_UNLOAD` case executes when an administrator runs `kldunload null` to remove the module from the kernel. The module system only calls this handler if the module is eligible for unload (no other code references it).

Device destruction: The `destroy_dev` function removes a device node from /dev and deallocates associated kernel structures. Each call uses the pointer saved during `MOD_LOAD`.

The function handles several cleanup tasks automatically:

- Removes the /dev entry so new opens fail with `ENOENT`
- Waits for existing opens to close (or forcibly closes them)
- Frees the `struct cdev` and related memory
- Unregisters the device from kernel accounting

The order of destruction doesn't matter for these independent devices. If they had dependencies (like one device routing operations to another), destruction order would be critical.

What if devices are open?: By default, `destroy_dev` blocks until all file descriptors referring to the device are closed. An administrator attempting `kldunload null` while a process has /dev/null open would experience a delay. In practice, /dev/null is frequently open (many daemons redirect output there), so unloading this module is rare.

```
case MOD_SHUTDOWN:
    break;
```

6.14.1.6.5 System Shutdown: No-Op The `MOD_SHUTDOWN` event fires during system shutdown or reboot. The handler does nothing because these devices don't need special shutdown handling:

- No hardware to disable or park in a safe state
- No data buffers to flush
- No network connections to close gracefully

Simply breaking (falling through to `return (0)`) indicates successful shutdown handling. The devices will cease to exist when the kernel halts; no explicit cleanup is necessary.

```
default:  
    return (EOPNOTSUPP);
```

6.14.1.6.6 Unsupported Events: Error Return The default case catches any module event types not explicitly handled. Returning `EOPNOTSUPP` (operation not supported) informs the module system that this event isn't applicable to this driver.

Other possible event types include `MOD_QUIESCE` (prepare for unload, used to check if unload is safe) and driver-specific custom events. This driver doesn't support those, so the default handler rejects them.

Why not panic?: An unknown event type isn't a driver bug, the kernel might introduce new event types in future versions. Returning an error is more robust than crashing.

```
return (0);
```

6.14.1.6.7 Success Return After handling any supported event (load, unload, shutdown), the function returns 0 to signal success. This allows the module operation to complete normally.

```
DEV_MODULE(null, null_modevent, NULL);  
MODULE_VERSION(null, 1);
```

6.14.1.6.8 Module Registration Macros These macros register the module with the kernel's module system.

DEV_MODULE(null, null_modevent, NULL): Declares a device driver module with three arguments:

- `null` - the module name, appearing in `kldstat` output and used with `kldload/kldunload` commands
- `null_modevent` - pointer to the event handler function
- `NULL` - optional additional data passed to the event handler (unused here)

The macro expands to generate data structures that the kernel's linker and module loader recognize. When the module loads, the kernel calls `null_modevent` with `type = MOD_LOAD`. When unloading, it calls with `type = MOD_UNLOAD`.

MODULE_VERSION(null, 1): Declares the module's version number. The arguments are:

- `null` - module name (must match `DEV_MODULE`)
- `1` - version number (integer)

Version numbers enable dependency checking. If another module depended on this one, it could specify "requires null version >= 1" to ensure compatibility. For this simple driver, versioning is primarily documentation, it signals that this is the first (and likely only) version of the interface.

6.14.1.6.9 Complete Module Lifecycle

The complete lifecycle for this driver:

At boot or `kldload null`:

1. Kernel loads the module into memory
2. Processes `DEV_MODULE` registration
3. Calls `null_modevent(mod, MOD_LOAD, NULL)`
4. Handler creates `/dev/full`, `/dev/null`, `/dev/zero`
5. Devices are now available to userspace

During operation:

- Applications open, read, write, `ioctl` the devices
- The `cdevsw` function pointers route operations to driver code
- No module events occur during normal operation

At `kldunload null`:

1. Kernel checks if unload is safe (no dependencies)
2. Calls `null_modevent(mod, MOD_UNLOAD, NULL)`
3. Handler destroys the three devices
4. Kernel removes module from memory
5. Attempts to open `/dev/null` now fail with `ENOENT`

At system shutdown:

1. Kernel calls `null_modevent(mod, MOD_SHUTDOWN, NULL)`
2. Handler does nothing (returns success)
3. System continues shutdown sequence
4. Module ceases to exist when kernel halts

This lifecycle management, explicit load and unload handlers, registration macros, is the standard pattern for all FreeBSD kernel modules. Device drivers, filesystem implementations, network protocols, and system call additions all use the same module event mechanism.

6.14.1.7 Interactive Exercises - `/dev/null`, `/dev/zero`, and `/dev/full`

Goal: Confirm you can read a real driver, map user-visible behavior to kernel code, and explain the minimal character device skeleton.

6.14.1.7.1 A) Map System Calls to `cdevsw` (Warm-up)

1. Which function handles writes to `/dev/full`, and what `errno` value does it return? Quote the function name and the return statement. What does this error code mean to userspace applications? *Hint:* look at `full_write`.
2. Which function handles reads from both `/dev/zero` and `/dev/full`? Quote the relevant `.d_read` assignments from both `cdevsw` structures. Why is it correct for both devices to share the same read handler, what behavior do they have in common? *Hint:* compare the `full_cdevsw` and `zero_cdevsw` structures and read `zero_read`.

3. Create a table listing each `cdevsw`'s name and its read/write function assignments:

<code>cdevsw</code>	<code>.d_name</code>	<code>.d_read</code>	<code>.d_write</code>
<code>full_cdevsw</code>	?	?	?
<code>null_cdevsw</code>	?	?	?
<code>zero_cdevsw</code>	?	?	?

Quote each structure. **Hint:* search for the three ``*_cdevsw`` definitions at the top of the file.

6.14.1.7.2 B) Read Path Reasoning with `uiomove()`

1. Locate the `KASSERT` that verifies this is a read operation. Quote the line and explain what would happen if this assertion failed. What does the `__func__` macro provide in the error message? *Hint:* look at the top of `zero_read`.
2. Explain the role of `uio->uio_resid` in the while loop condition. What does this field represent, and how does it change during the loop? Quote the while condition. *Hint:* inside `zero_read`.
3. Why does the code limit each transfer to `ZERO_REGION_SIZE` rather than copying all requested bytes at once? What would be the problem with transferring 1MB in a single `uiomove()` call? Quote the if statement that implements this limit. *Hint:* the clamp is the first thing inside the `zero_read` loop body.
4. The code references two pre-allocated kernel resources: `zero_region` (a pointer) and `ZERO_REGION_SIZE` (a constant). Quote the lines where each is used. Then use `grep` to find where `ZERO_REGION_SIZE` is defined:

```
% grep -r "define.*ZERO_REGION_SIZE" /usr/src/sys/amd64/include/
```

What is the value on your system? **Hint:* ``zero_region`` is used inside ``zero_read``, and ``ZERO_REGION_SIZE``

6.14.1.7.3 C) Write Path Contrasts

1. Compare the implementations of `null_write` and `full_write`. For each function, answer:
 - What does it do with `uio->uio_resid`?
 - What value does it return?
 - What will a userspace `write(2)` call return?

Now verify from userspace:

```
# This should succeed, reporting bytes written:
% dd if=/dev/zero of=/dev/null bs=64k count=8 2>&1 | grep copied

# This should fail with "No space left on device":
% dd if=/dev/zero of=/dev/full bs=1k count=1 2>&1 | grep -i "space"
```

For each test, identify which write handler was called and quote the specific line that caused the observed behavior.

6.14.1.7.4 D) Minimal ioctl Shape

1. Create a comparison table of ioctl handling. For `null_ioctl` and `zero_ioctl`, fill in:

Command	<code>null_ioctl</code> behavior	<code>zero_ioctl</code> behavior
<code>DIOCSKERNELDUMP??</code>		
<code>FIONBIO??</code>		
<code>FIOASYNC??</code>		
Unknown command??		

For each entry, quote the relevant case statement and explain the behavior.

2. The `FIOASYNC` case has special handling when enabling async I/O. Quote the conditional check and explain why these devices reject async I/O mode. *Hint*: look at the `FIOASYNC` case in both `null_ioctl` and `zero_ioctl`.

6.14.1.7.5 E) Device Node Lifecycle

1. During `MOD_LOAD`, three device nodes are created via `make_dev_credf()`. For each call (in the `MOD_LOAD` arm of `null_modevent`), identify:
 - The device name (what appears in `/dev/`)
 - The `cdevsw` pointer (which function table)
 - The permission mode (what does `0666` mean?)
 - The owner and group (`UID_ROOT`, `GID_WHEEL`)

Quote one complete `make_dev_credf()` call and label each parameter.

2. During `MOD_UNLOAD`, `destroy_dev()` is called three times (in the `MOD_UNLOAD` arm of `null_modevent`). Quote these calls and explain:
 - Why do we need the global pointers (`full_dev`, `null_dev`, `zero_dev`)?
 - What would happen if we forgot to call `destroy_dev()` during unload?
 - Why must the `MOD_LOAD` and `MOD_UNLOAD` operations be symmetric?

6.14.1.7.6 F) Trace from Userspace

1. Verify that `/dev/zero` produces zeros and `/dev/null` consumes data:

```
% dd if=/dev/zero bs=1k count=1 2>/dev/null | hexdump -C | head -n 2
# Expected: all zeros (00 00 00 00...)

% printf 'test data' | dd of=/dev/null 2>/dev/null ; echo "Exit code: $?"
# Expected: Exit code: 0
```

Explain these results by tracing through:

- `zero_read`: Which lines produce the zeros? How does the loop work?
- `null_write`: Which line makes the write “succeed”? What happens to the data?

Quote the specific lines responsible for each behavior.

2. Read from `/dev/full` and examine what you get:

```
% dd if=/dev/full bs=16 count=1 2>/dev/null | hexdump -C
```

What output do you see? Look at the `full_cdevsw` structure: which `.d_read` function does it use?

Why does `/dev/full` return zeros instead of an error?

6.14.1.7.7 G) Module Lifecycle

1. Look at the `null_modevent` switch statement. List all the case labels and what each one does. Which cases actually perform work versus just returning success?
2. Find the two macros at the end of the file that register this module. Quote them and explain:
 - What does `DEV_MODULE` do?
 - What does `MODULE_VERSION` do?
 - Why do both use the name “null”?
3. The `MAKEDEV_ETERNAL_KLD` flag is used in all three `make_dev_credf()` calls. What does this flag mean, and why is it appropriate for these devices? *Hint*: look at the `make_dev_credf()` calls inside `null_modevent`, and consider what happens if a process has `/dev/null` open when you try to unload the module.

6.14.1.8 Stretch (thought experiment)

Stretch 1: Examine `null_write`. The function does two things: sets `uio->uio_resid = 0` and returns 0.

Thought experiment: If we changed the `return (0);` to `return (EIO);` but kept the `uio->uio_resid = 0;` assignment unchanged, what would happen?

- What would the kernel think about bytes written?
- What would `write(2)` return to userspace?
- What would `errno` be set to?

Quote the lines involved and explain the interaction between `uio_resid` and the return value.

Stretch 2: In `zero_read`, the code limits each transfer to `ZERO_REGION_SIZE`. Quote the if statement where this limit is enforced.

Thought experiment: Suppose we removed this check and always did:

```
len = uio->uio_resid; // No limit!
error = uiomove(zbuf, len, uio);
```

If a user requests 10MB from `/dev/zero`:

- What invariant would make this “work” (not crash)?
- What resource constraint would we be ignoring?
- Why does the current code use a pre-allocated buffer of limited size?

Hint: The `zero_region` is only `ZERO_REGION_SIZE` bytes. What happens if we try to copy more than that from this fixed-size buffer?

6.14.1.9 Bridge to the next tour

Before moving on: if you can match each user-visible behavior to the right function in `null.c`, you’ve internalized the **character-device skeleton** we’ll keep meeting. Next we’ll look at **led(4)**, which remains small but adds a user-visible **control surface** (writes that change state). Keep watching for three things: **how the device node is created**, **how operations are routed**, and **how the driver declines unsupported actions cleanly**.

6.14.2 Tour 2 - A tiny write-only control surface with timers: **led(4)**

Open the file:

```
% cd /usr/src/sys/dev/led
% less led.c
```

In one file we get a practical pattern for **write-driven device control** backed by a **timer** and per-device state. You’ll see: a per-LED softc, global bookkeeping, a periodic **callout** that advances blink patterns, a parser that converts human-friendly commands into compact sequences, a `write(2)` entry point, and minimal create/destroy helpers.

6.14.2.1 1.0) Includes

```
12: #include <sys/cdefs.h>
13: #include <sys/param.h>
14: #include <sys/conf.h>
15: #include <sys/ctype.h>
16: #include <sys/kernel.h>
17: #include <sys/limits.h>
18: #include <sys/lock.h>
19: #include <sys/malloc.h>
20: #include <sys/mutex.h>
```

```
21: #include <sys/queue.h>
22: #include <sys/sbuf.h>
23: #include <sys/sx.h>
24: #include <sys/system.h>
25: #include <sys/uio.h>
27: #include <dev/led/led.h>
```

6.14.2.1.1 Headers and Subsystem Interface The LED driver begins with kernel headers and a subsystem header that establishes its role as an infrastructure component used by other drivers. Unlike the null driver which stands alone, the LED driver provides services to hardware drivers that need to expose status indicators.

```
#include <sys/cdefs.h>
#include <sys/param.h>
#include <sys/conf.h>
#include <sys/ctype.h>
#include <sys/kernel.h>
#include <sys/limits.h>
#include <sys/lock.h>
#include <sys/malloc.h>
#include <sys/mutex.h>
#include <sys/queue.h>
#include <sys/sbuf.h>
#include <sys/sx.h>
#include <sys/system.h>
#include <sys/uio.h>
```

6.14.2.1.2 Standard Kernel Headers These headers provide the infrastructure for a stateful, timer-driven device driver:

<sys/cdefs.h>, **<sys/param.h>**, **<sys/system.h>**: Fundamental system definitions identical to those in `null.c`. Every kernel source file begins with these.

<sys/conf.h>: Character device configuration, providing `cdevsw` and `make_dev()`. The LED driver uses these to create device nodes dynamically as hardware drivers register LEDs.

<sys/ctype.h>: Character classification functions like `isdigit()`. The LED driver parses user-supplied strings to control blink patterns, requiring character type checking.

<sys/kernel.h>: Kernel initialization infrastructure. This driver uses `SYSINIT` to perform one-time initialization during boot, setting up global resources before any LEDs are registered.

<sys/limits.h>: System limits like `INT_MAX`. The LED driver uses this to configure its unit number allocator with maximum range.

<sys/lock.h> and **<sys/mutex.h>**: Locking primitives for protecting shared data structures. The driver uses a mutex to protect the LED list and blinker state from concurrent access by timer callbacks and user writes.

<sys/queue.h>: BSD linked list macros (`LIST_HEAD`, `LIST_FOREACH`, `LIST_INSERT_HEAD`, `LIST_REMOVE`). The driver maintains a global list of all registered LEDs, allowing timer callbacks to iterate and update each one.

<sys/sbuf.h>: Safe string buffer manipulation. The driver uses `sbuf` to build blink pattern strings from user input, avoiding fixed-size buffer overflows. String buffers automatically grow as needed and provide bounds checking.

<sys/sx.h>: Shared/exclusive locks (reader/writer locks). The driver uses an `sx` lock to protect device creation and destruction, allowing concurrent reads of the LED list while serializing structural modifications.

<sys/uio.h>: User I/O operations. Like `null.c`, this driver needs `struct uio` and `uiomove()` to transfer data between kernel and userspace.

<sys/malloc.h>: Kernel memory allocation. Unlike `null.c` which had no dynamic memory, the LED driver allocates per-LED state structures and duplicates strings for LED names and blink patterns.

```
#include <dev/led/led.h>
```

6.14.2.1.3 Subsystem Interface Header This header defines the LED subsystem's public API, the interface that other kernel drivers use to register and control LEDs. While the specific contents aren't shown in this source file, typical declarations would include:

led_t typedef: A function pointer type for LED control callbacks. Hardware drivers provide a function matching this signature that turns their physical LED on or off:

```
typedef void led_t(void *priv, int onoff);
```

Public functions: The API that hardware drivers call:

- `led_create()` - register a new LED, creating a `/dev/led/name` device node
- `led_create_state()` - register an LED with initial state
- `led_destroy()` - unregister an LED when hardware is removed
- `led_set()` - programmatically control an LED from kernel code

Example usage by a hardware driver:

```
// In a disk driver's attach function:
struct cdev *led_dev;
led_dev = led_create(disk_led_control, sc, "disk0");

// Later, in the LED control callback:
static void
disk_led_control(void *priv, int onoff)
{
    struct disk_softc *sc = priv;
```

```

if (onoff)
    /* Turn on LED via hardware register write */
else
    /* Turn off LED via hardware register write */
}

```

6.14.2.1.4 Architectural Role The header organization reveals the LED driver’s dual nature:

As a character device driver: It includes standard device driver headers (`<sys/conf.h>`, `<sys/uio.h>`) to create `/dev/led/*` nodes that userspace can write to.

As a subsystem: It includes `<dev/led/led.h>` to export an API that other drivers consume. Hardware drivers don’t manipulate `/dev/led/*` directly, they call `led_create()` and provide callbacks.

This pattern, a driver that both exposes user-facing devices and provides kernel-facing APIs, appears throughout FreeBSD. Examples include:

- The `devctl` driver: creates `/dev/devctl` while providing `devctl_notify()` for kernel event reporting
- The `random` driver: creates `/dev/random` while providing `read_random()` for kernel consumers
- The `mem` driver: creates `/dev/mem` while providing direct memory access functions

The LED driver sits between hardware-specific drivers (which know how to control physical LEDs) and userspace (which wants to control LED patterns). It provides abstraction, hardware drivers implement simple on/off control; the LED subsystem handles complex blink patterns, timing, and user interface.

6.14.2.2 1.1) Per-LED State (softc)

```

30: struct ledsc {
31:     LIST_ENTRY(ledsc)  list;
32:     char                *name;
33:     void                *private;
34:     int                 unit;
35:     led_t               *func;
36:     struct cdev *dev;
37:     struct sbuf          *spec;
38:     char                *str;
39:     char                *ptr;
40:     int                 count;
41:     time_t              last_second;
42: };

```

6.14.2.2.1 Per-LED State Structure The `ledsc` structure (LED `softc`, following FreeBSD naming convention for “software context”) contains all per-device state for one registered LED. Unlike the null driver which had no per-device state, the LED driver creates one of these structures for each LED registered in the system, tracking both device identity and current blink pattern execution state.

```

struct ledsc {
    LIST_ENTRY(ledsc)    list;
    char                 *name;
    void                 *private;
    int                  unit;
    led_t                *func;
    struct cdev *dev;
    struct sbuf          *spec;
    char                 *str;
    char                 *ptr;
    int                  count;
    time_t               last_second;
};

```

6.14.2.2.2 Structure Definition and Fields **LIST_ENTRY(ledsc) list**: Linkage for the global LED list. The **LIST_ENTRY** macro (from `<sys/queue.h>`) embeds forward and backward pointers directly in the structure, allowing this LED to be part of a doubly-linked list without separate allocation. The global `led_list` chains together all registered LEDs, enabling timer callbacks to iterate and update each one.

char *name: The LED's name string, duplicated from the hardware driver's registration call. This name appears in the device path `/dev/led/name` and identifies the LED in kernel API calls to `led_set()`. Examples: "disk0", "power", "heartbeat". The string is dynamically allocated and must be freed when the LED is destroyed.

void *private: An opaque pointer passed back to the hardware driver's control function. The hardware driver provides this during `led_create()`, typically pointing to its own device context structure. When the LED subsystem needs to turn the LED on or off, it calls the hardware driver's callback with this pointer, allowing the driver to locate the relevant hardware registers.

int unit: A unique unit number for this LED, used to construct the device minor number. Allocated from a unit number pool to prevent conflicts when multiple LEDs are registered. Unlike the null driver's fixed unit numbers (0 for all devices), the LED driver dynamically assigns units as LEDs are created.

led_t *func: Function pointer to the hardware driver's LED control callback. This function has the signature `void (*led_t)(void *priv, int onoff)` where `priv` is the private pointer above and `onoff` is non-zero for "on", zero for "off". This callback is the hardware-specific part, it knows how to manipulate GPIO pins, write to hardware registers, or send USB control transfers to actually light or extinguish the LED.

struct cdev *dev: Pointer to the character device structure representing `/dev/led/name`. This is what `make_dev()` returns during LED creation. The device node allows userspace to write blink patterns to the LED. The pointer is needed later to call `destroy_dev()` when the LED is removed.

6.14.2.2.3 Blink Pattern Execution State The remaining fields track blink pattern execution by the timer callback:

struct sbuf *spec: The parsed blink specification string buffer. When a user writes a pattern like "f" (flash) or "m...—..." (morse code), the parser converts it to a sequence of timing codes and stores it in this `sbuf`. The string persists as long as the pattern is active, allowing the timer to repeatedly traverse it.

char *str: Pointer to the beginning of the pattern string (extracted from `spec` via `sbuf_data()`). This is where pattern execution starts and where it loops back after reaching the end. If `NULL`, no pattern is active and the LED is in static on/off state.

char *ptr: The current position in the pattern string. The timer callback examines this character to determine what to do next (turn LED on/off, delay for *N* tenths of a second). After processing each character, `ptr` advances. When it reaches the string terminator, it wraps back to `str` for continuous repetition.

int count: A countdown timer for delay characters. Pattern codes like ‘a’ through ‘j’ mean “wait for 1-10 tenths of a second”. When the timer encounters such a code, it sets `count` to the delay value and decrements it on each timer tick. While `count > 0`, the timer skips pattern advancement, implementing the delay.

time_t last_second: Timestamp tracking the last second boundary, used for ‘U’/‘u’ pattern codes that toggle the LED once per second (creating a 1Hz heartbeat pattern). The timer compares `time_second` (kernel’s current time) to this field, only updating the LED when the second changes. This prevents multiple updates within the same second if the timer fires faster than 1Hz.

6.14.2.2.4 Memory Management and Lifecycle Several fields point to dynamically allocated memory:

- `name` - allocated with `strdup(name, M_LED)` during creation
- `spec` - created with `sbuf_new_auto()` when a pattern is set
- The structure itself is allocated with `malloc(sizeof *sc, M_LED, M_WAITOK | M_ZERO)`

All must be freed during `led_destroy()` to prevent memory leaks. The structure’s lifetime spans from `led_create()` to `led_destroy()`, potentially lasting the entire system uptime if the hardware driver never unregisters the LED.

6.14.2.2.5 Relationship to Device Node The `ledsc` structure and the `/dev/led/name` device node are bidirectionally linked:

```
struct cdev (device node)
    ->  si_drv1
struct ledsc
    ->  dev
struct cdev (same device node)
```

This bidirectional linkage allows:

- The write handler to find the LED state: `sc = dev->si_drv1`
- The destroy function to remove the device: `destroy_dev(sc->dev)`

6.14.2.2.6 Contrast with null.c The null driver had no equivalent structure because its devices were stateless. The LED driver needs per-device state because:

Identity: Each LED has a unique name and device node

Callback: Each LED has hardware-specific control logic

Pattern state: Each LED may be executing a different blink pattern at different positions

Timing: Each LED's delay counters and timestamps are independent

This per-device state structure is typical of drivers managing multiple instances of similar hardware. The pattern is universal: one structure per managed entity, containing identity, configuration, and operational state.

6.14.2.3 1.2) Globals

```
44: static struct unrhdr *led_unit;  
45: static struct mtx led_mtx;  
46: static struct sx led_sx;  
47: static LIST_HEAD(, ledsc) led_list = LIST_HEAD_INITIALIZER(led_list);  
48: static struct callout led_ch;  
49: static int blinkers = 0;  
51: static MALLOC_DEFINE(M_LED, "LED", "LED driver");
```

6.14.2.3.1 Global State and Synchronization The LED driver maintains several global variables that coordinate all registered LEDs. These globals provide resource allocation, synchronization, timer management, and a registry of active LEDs, infrastructure shared across all LED instances.

```
static struct unrhdr *led_unit;
```

6.14.2.3.2 Resource Allocator The unit number handler allocates unique unit numbers for LED devices. Each registered LED receives a distinct unit number used to construct its device minor number, ensuring `/dev/led/disk0` and `/dev/led/power` don't collide even if created simultaneously.

The `unrhdr` (unit number handler) provides thread-safe allocation and deallocation of integers from a range. During driver initialization, `new_unrhdr(0, INT_MAX, NULL)` creates a pool spanning the entire positive integer range. When hardware drivers call `led_create()`, the code calls `alloc_unr(led_unit)` to obtain the next available unit. When an LED is destroyed, `free_unr(led_unit, sc->unit)` returns the unit to the pool for reuse.

This dynamic allocation contrasts with the null driver's fixed units (always 0). The LED driver must handle arbitrary numbers of LEDs appearing and disappearing as hardware is added and removed.

```
static struct mtx led_mtx;  
static struct sx led_sx;
```

6.14.2.3.3 Synchronization Primitives The driver uses two locks with distinct purposes:

led_mtx (mutex): Protects the LED list and blink pattern execution state. This lock guards:

- The `led_list` linked list as LEDs are added and removed

- The `blinkers` counter tracking active patterns
- Individual `ledsc` fields modified by timer callbacks (`ptr`, `count`, `last_second`)

The mutex uses `MTX_DEF` semantics (default, can sleep while held). Timer callbacks acquire this mutex briefly to examine and update LED states. Write operations acquire it to install new blink patterns.

led_sx (shared/exclusive lock): Protects device creation and destruction. This lock serializes:

- Calls to `make_dev()` and `destroy_dev()`
- Unit number allocation and deallocation
- String duplication for LED names

Shared/exclusive locks allow multiple readers (threads examining which LEDs exist) to proceed concurrently while writers (threads creating or destroying LEDs) have exclusive access. For the LED driver, creation and destruction are infrequent operations that benefit from being fully serialized with an exclusive lock.

Why two locks?: The separation enables concurrency. Timer callbacks need fast access to LED states protected by the mutex, while device creation/destruction requires the heavier `sx` lock. If a single lock protected everything, timer callbacks would block waiting for slow device operations. The split allows timers to run freely while device management proceeds independently.

```
static LIST_HEAD(, ledsc) led_list = LIST_HEAD_INITIALIZER(led_list);
```

6.14.2.3.4 LED Registry The global LED list maintains all registered LEDs in a doubly-linked list. The `LIST_HEAD` macro (from `<sys/queue.h>`) declares a list head structure and `LIST_HEAD_INITIALIZER` sets its initial empty state.

This list serves multiple purposes:

Timer iteration: The timer callback walks the list with `LIST_FOREACH(sc, &led_list, list)` to update each active LED's blink pattern. Without this registry, the timer wouldn't know which LEDs exist.

Name lookup: The `led_set()` function searches the list to find an LED by name when kernel code wants to control an LED programmatically.

Cleanup verification: When the last LED is removed (`LIST_EMPTY(&led_list)`), the driver can stop the timer callback, conserving CPU cycles when no LEDs need servicing.

The list is protected by `led_mtx` since both timer callbacks and device operations modify it.

```
static struct callout led_ch;
static int blinkers = 0;
```

6.14.2.3.5 Timer Callback Infrastructure **led_ch (callout):** A kernel timer that fires periodically to advance blink patterns. When any LED has an active pattern, the timer is scheduled to fire 10 times per second (`hz / 10`, where `hz` is timer ticks per second, typically 1000). Each timer firing calls `led_timeout()` which walks the LED list and updates pattern states.

The callout remains idle (not scheduled) when no LEDs are blinking, conserving resources. The first LED to receive a blink pattern schedules the timer with `callout_reset(&led_ch, hz / 10, led_timeout, NULL)`. Subsequent patterns don't reschedule, the single timer services all LEDs.

blinkers counter: Tracks how many LEDs currently have active blink patterns. When a pattern is assigned, `blinkers++`. When a pattern completes or is replaced with static on/off, `blinkers--`. When the counter reaches zero, the timer callback doesn't reschedule itself, stopping the periodic wakeups.

This reference counting is critical for performance. Without it, the timer would fire continuously even with no work to do. The counter gates timer activity: schedule when transitioning 0 -> 1, stop when transitioning 1 -> 0.

```
static MALLOC_DEFINE(M_LED, "LED", "LED driver");
```

6.14.2.3.6 Memory Type Declaration The `MALLOC_DEFINE` macro registers a memory allocation type for the LED subsystem. All LED-related allocations specify `M_LED`:

- `malloc(sizeof *sc, M_LED, ...)` for softc structures
- `strdup(name, M_LED)` for LED name strings

Memory types enable kernel accounting and debugging:

- `vmstat -m` shows memory consumption per type
- Developers can track whether the LED driver is leaking memory
- Kernel memory debuggers can filter allocations by type

The three arguments are:

1. `M_LED` - the C identifier used in `malloc()` calls
2. "LED" - short name appearing in accounting output
3. "LED driver" - descriptive text for documentation

6.14.2.3.7 Initialization Coordination These globals are initialized in a specific sequence during boot:

1. **Static initialization:** `led_list` and `blinkers` get compile-time initial values
2. **`led_drvinit()` (via `SYSINIT`):** Allocates `led_unit`, initializes `led_mtx` and `led_sx`, prepares the callout
3. **Runtime:** Hardware drivers call `led_create()` to register LEDs, incrementing `blinkers` and populating `led_list`

The static storage class on all globals limits their visibility to this source file. No other kernel code can directly access these variables, all interactions go through the public API (`led_create()`, `led_destroy()`, `led_set()`). This encapsulation prevents external code from corrupting the LED subsystem's internal state.

6.14.2.3.8 Contrast with null.c The null driver had minimal global state: three device pointers for its fixed devices. The LED driver's globals reflect its dynamic nature:

- **Resource allocation:** Unit numbers for arbitrary device counts
- **Concurrency:** Two locks for different access patterns
- **Registry:** A list tracking all active LEDs
- **Scheduling:** Timer infrastructure for pattern execution
- **Accounting:** Memory type for allocation tracking

This richer global infrastructure supports the LED driver's role as a subsystem managing multiple dynamically-created devices with time-based behaviors, rather than a simple driver exposing fixed stateless devices.

6.14.2.4 2) The heartbeat: `led_timeout()` advances the pattern

This **periodic callout** walks all LEDs and advances each one's pattern. Patterns are encoded in ASCII, so the parser and state machine stay tiny.

```

54: static void
55: led_timeout(void *p)
56: {
57:     struct ledsc    *sc;
58:     LIST_FOREACH(sc, &led_list, list) {
59:         if (sc->ptr == NULL)
60:             continue;
61:         if (sc->count > 0) {
62:             sc->count--;
63:             continue;
64:         }
65:         if (*sc->ptr == '.') {
66:             sc->ptr = NULL;
67:             blinkers--;
68:             continue;
69:         } else if (*sc->ptr == 'U' || *sc->ptr == 'u') {
70:             if (sc->last_second == time_second)
71:                 continue;
72:             sc->last_second = time_second;
73:             sc->func(sc->private, *sc->ptr == 'U');
74:         } else if (*sc->ptr >= 'a' && *sc->ptr <= 'j') {
75:             sc->func(sc->private, 0);
76:             sc->count = (*sc->ptr & 0xf) - 1;
77:         } else if (*sc->ptr >= 'A' && *sc->ptr <= 'J') {
78:             sc->func(sc->private, 1);
79:             sc->count = (*sc->ptr & 0xf) - 1;
80:         }
81:         sc->ptr++;

```

```

82:         if (*sc->ptr == '\0')
83:             sc->ptr = sc->str;
84:     }
85:     if (blinkers > 0)
86:         callout_reset(&led_ch, hz / 10, led_timeout, p);
87: }

```

6.14.2.4.1 Timer Callback: Pattern Execution Engine The `led_timeout` function is the heart of the LED subsystem's blink pattern execution. Called by the kernel's timer subsystem approximately 10 times per second, it walks the global LED list and advances each active pattern by one step, interpreting a simple pattern language to control LED timing and state.

```

static void
led_timeout(void *p)
{
    struct ledsc    *sc;
    LIST_FOREACH(sc, &led_list, list) {

```

6.14.2.4.2 Function Entry and List Iteration **Function signature:** Timer callbacks receive a single `void *` argument passed during timer scheduling. This driver doesn't use the argument (it's typically `NULL`), relying instead on the global LED list to find work.

Iterating all LEDs: The `LIST_FOREACH` macro walks the doubly-linked `led_list`, visiting each registered LED. This allows one timer to service multiple independent LEDs, each potentially executing a different blink pattern at a different position. The iteration is safe because the list is protected by `led_mtx` (the callout was initialized with this mutex via `callout_init_mtx()`).

```

if (sc->ptr == NULL)
    continue;

```

6.14.2.4.3 Skipping Inactive LEDs The `ptr` field indicates whether this LED has an active blink pattern. When `NULL`, the LED is in static on/off state and needs no timer processing. The callback skips to the next LED immediately.

This check is the first filter: LEDs without patterns don't consume CPU time. Only LEDs actively blinking require processing on each timer tick.

```
if (sc->count > 0) {
    sc->count--;
    continue;
}
```

6.14.2.4.4 Handling Delay States The count field implements delays in blink patterns. When the pattern interpreter encounters timing codes like ‘a’ through ‘j’ (meaning “wait 1-10 tenths of a second”), it sets count to the delay value. On subsequent timer ticks, the callback decrements count without advancing through the pattern.

Example: Pattern code ‘c’ (wait 3 tenths of a second) sets count = 2 (the value is 1 less than the intended delay). The next two timer ticks decrement count to 1, then 0. On the third tick, count is already 0, so this check fails and pattern execution proceeds.

This mechanism creates precise timing: at 10Hz, each count represents 0.1 seconds. Pattern ‘AcAc’ produces: LED on, wait 0.3s, LED on again, wait 0.3s, repeat.

```
if (*sc->ptr == '.') {
    sc->ptr = NULL;
    blinkers--;
    continue;
}
```

6.14.2.4.5 Pattern Termination The period character ‘.’ signals pattern end. Unlike most patterns which loop indefinitely, some user specifications include an explicit terminator. When encountered:

Stop pattern execution: Setting ptr = NULL marks this LED as inactive. Future timer ticks will skip it at the first check.

Decrement blinker count: Reducing blinkers tracks that one fewer LED needs servicing. When this counter reaches zero (checked at function end), the timer stops scheduling itself.

Skip remaining code: The continue jumps to the next LED in the list. The pattern-advance and wrap-around code at the tail of led_timeout (the sc->ptr++ step and the *sc->ptr == '\0' rewind) does not execute for terminated patterns.

```
else if (*sc->ptr == 'U' || *sc->ptr == 'u') {
    if (sc->last_second == time_second)
        continue;
    sc->last_second = time_second;
    sc->func(sc->private, *sc->ptr == 'U');
}
```

6.14.2.4.6 Heartbeat Pattern: Second-Based Toggle The ‘U’ and ‘u’ codes create once-per-second toggles, useful for heartbeat indicators showing the system is alive.

Second boundary detection: The kernel variable `time_second` holds the current Unix timestamp. Comparing it to `last_second` detects when a second boundary has passed. If the values match, we’re still within the same second and the callback skips processing with `continue`.

Recording the transition: `sc->last_second = time_second` remembers this second, preventing multiple updates if the timer fires multiple times per second (which it does, 10 times per second).

Updating the LED: The callback invokes the hardware driver’s control function. The second parameter determines LED state:

- `*sc->ptr == 'U' -> true (1) -> LED on`
- `*sc->ptr == 'u' -> false (0) -> LED off`

Pattern “Uu” creates a 1Hz toggle: on for one second, off for one second. Pattern “U” alone keeps the LED on but only updates at second boundaries, which may be used for synchronization purposes.

```
else if (*sc->ptr >= 'a' && *sc->ptr <= 'j') {
    sc->func(sc->private, 0);
    sc->count = (*sc->ptr & 0xf) - 1;
}
```

6.14.2.4.7 Off Delay Pattern Lowercase letters ‘a’ through ‘j’ mean “turn LED off and wait.” This combines two operations: immediate state change plus delay setup.

Turning off the LED: `sc->func(sc->private, 0)` calls the hardware driver’s control function with the off command (second parameter is 0).

Computing the delay: The expression `(*sc->ptr & 0xf) - 1` extracts the delay duration from the character code. In ASCII:

- ‘a’ is 0x61, `0x61 & 0x0f = 1`, minus 1 = 0 (wait 0.1 seconds)
- ‘b’ is 0x62, `0x62 & 0x0f = 2`, minus 1 = 1 (wait 0.2 seconds)
- ‘c’ is 0x63, `0x63 & 0x0f = 3`, minus 1 = 2 (wait 0.3 seconds)
- ...
- ‘j’ is 0x6A, `0x6A & 0x0f = 10`, minus 1 = 9 (wait 1.0 seconds)

The mask `& 0xf` isolates the low 4 bits, which conveniently map ‘a’-‘j’ to values 1-10. Subtracting 1 converts to the countdown format (timer ticks remaining minus one).

```
else if (*sc->ptr >= 'A' && *sc->ptr <= 'J') {
    sc->func(sc->private, 1);
    sc->count = (*sc->ptr & 0xf) - 1;
}
```

6.14.2.4.8 On Delay Pattern Uppercase letters ‘A’ through ‘J’ work identically to lowercase, except the LED is turned on instead of off. The delay calculation is the same:

- ‘A’ -> on for 0.1 seconds
- ‘B’ -> on for 0.2 seconds
- ...
- ‘J’ -> on for 1.0 seconds

Pattern “AaBb” creates: on 0.1s, off 0.1s, on 0.2s, off 0.2s, repeat. Pattern “Aa” is a standard fast blink at ~2.5Hz.

```
sc->ptr++;
if (*sc->ptr == '\0')
    sc->ptr = sc->str;
```

6.14.2.4.9 Pattern Advancement and Looping After processing the current pattern character (whether it was a heartbeat code or a delay code), the pointer advances to the next character.

Detecting pattern end: If the new position is the null terminator, the pattern has been fully executed once. Rather than stopping (as the ‘.’ terminator does), most patterns loop indefinitely.

Looping back: `sc->ptr = sc->str` resets to the pattern’s beginning. The next timer tick will start over from the first character, creating a repeating cycle.

Example: Pattern “AjBj” becomes on-1s, on-1s, repeat continuously. The pattern never stops unless replaced by a new write or the LED is destroyed.

```
if (blinkers > 0)
    callout_reset(&led_ch, hz / 10, led_timeout, p);
}
```

6.14.2.4.10 Timer Rescheduling After processing all LEDs, the callback decides whether to reschedule itself. If any LED still has an active pattern (`blinkers > 0`), the timer is reset to fire again in `hz / 10` ticks (0.1 seconds).

Self-perpetuating timer: Each invocation schedules the next invocation, creating a continuous loop as long as work remains. This is different from a periodic timer that fires unconditionally, the LED timer is work-driven.

Automatic shutdown: When the last active pattern terminates (either via ‘.’ or being replaced with static state), `blinkers` drops to 0 and the timer doesn’t reschedule. The callback exits and won’t run again until a new pattern activates, conserving CPU when all LEDs are static.

The hz variable: The kernel constant `hz` represents timer ticks per second (typically 1000 on modern systems). Dividing by 10 gives the delay in ticks for one-tenth of a second, matching the pattern language’s resolution.

6.14.2.4.11 Pattern Language Summary The timer interprets a simple language embedded in pattern strings:

Code	Meaning	Duration
'a'-'j'	LED off	0.1-1.0 seconds
'A'-'J'	LED on	0.1-1.0 seconds
'U'	LED on	At second boundary
'u'	LED off	At second boundary
'.'	End pattern	-

Example patterns and their effects:

- “Aa” -> blink at ~2.5Hz (0.1s on, 0.1s off)
- “AjAj” -> slow blink at 0.5Hz (1s on, 1s off)
- “AaAaBjBj” -> fast double blink, long pause
- “U” -> steady on, synced to seconds
- “Uu” -> 1Hz toggle

This compact encoding allows complex blink behaviors from short strings, all interpreted by this one timer callback serving all LEDs in the system.

6.14.2.5 3) Apply a new state/pattern: `led_state()`

Given a compiled pattern (sbuf) or a simple on/off flag, this function updates the softc and starts or stops the periodic timer.

```

88: static int
89: led_state(struct ledsc *sc, struct sbuf **sb, int state)
90: {
91:     struct sbuf *sb2 = NULL;
92:
93:     sb2 = sc->spec;
94:     sc->spec = *sb;
95:     if (*sb != NULL) {
96:         if (sc->str != NULL)
97:             free(sc->str, M_LED);
98:         sc->str = strdup(sbuf_data(*sb), M_LED);
99:         if (sc->ptr == NULL)
100:             blinkers++;
101:         sc->ptr = sc->str;
102:     } else {
103:         sc->str = NULL;
104:         if (sc->ptr != NULL)
105:             blinkers--;

```



```

106:      sc->ptr = NULL;
107:      sc->func(sc->private, state);
108:  }
109:  sc->count = 0;
110:  *sb = sb2;
111:  return(0);
112: }

```

6.14.2.5.1 LED State Management: Installing Patterns The `led_state` function installs a new blink pattern or static state for an LED. It handles the transition between different LED modes, managing memory for pattern strings, updating the blinker counter for timer control, and invoking hardware callbacks when needed. This function is the central state change coordinator called by both the write handler and the kernel API.

```

static int
led_state(struct ledsc *sc, struct sbuf **sb, int state)
{
    struct sbuf *sb2 = NULL;

    sb2 = sc->spec;
    sc->spec = *sb;

```

6.14.2.5.2 Function Signature and Pattern Swap **Parameters:** The function receives three values:

- `sc` - the LED whose state is being changed
- `sb` - pointer to a pointer to a string buffer containing the new pattern (or `NULL` for static state)
- `state` - the desired static state (0 or 1) if no pattern is provided

The double pointer pattern: The `sb` parameter is `struct sbuf **`, allowing the function to swap buffers with the caller. The function takes ownership of the caller's buffer and returns the old buffer for cleanup. This swap avoids copying pattern strings and ensures proper memory management.

Preserving the old pattern: `sb2 = sc->spec` saves the current pattern buffer before installing the new one. At function end, this old buffer is returned to the caller via `*sb = sb2`. The caller becomes responsible for freeing it with `sbuff_delete()`.

```

if (*sb != NULL) {
    if (sc->str != NULL)
        free(sc->str, M_LED);
    sc->str = strdup(sbuff_data(*sb), M_LED);
    if (sc->ptr == NULL)
        blinkers++;
    sc->ptr = sc->str;

```

6.14.2.5.3 Installing a Blink Pattern When the caller provides a pattern (non-NULL `sb`), the function activates pattern mode.

Freeing the old string: If `sc->str` is non-NULL, a previous pattern's string exists and must be freed. The `free(sc->str, M_LED)` call releases this memory back to the kernel heap. The `M_LED` tag matches the allocation type used during `strdup()`, maintaining accounting consistency.

Duplicating the new pattern: `sbuf_data(*sb)` extracts the null-terminated string from the string buffer, and `strdup(name, M_LED)` allocates memory and copies it. The pattern string must persist because the timer callback will traverse it repeatedly, the string buffer itself may be deleted by the caller, so a separate copy is needed.

Activating the timer: The check `if (sc->ptr == NULL)` detects whether this LED was previously inactive. If so, incrementing `blinkers++` records that one more LED now needs timer servicing. The timer callback checks this counter at the end of each run; transitioning from 0 to 1 causes the timer to be rescheduled.

Starting pattern execution: `sc->ptr = sc->str` sets the pattern position to the beginning. On the next timer tick, `led_timeout` will process this LED's first pattern character.

Why not start the timer here?: The timer might already be running if other LEDs have active patterns. The `blinkers` counter tracks this: if it was already non-zero, the timer is already scheduled and will process this LED on its next tick. Only when `blinkers` transitions from 0 to 1 (detected in the write handler or `led_set()`) does the timer need explicit scheduling.

```
} else {
    sc->str = NULL;
    if (sc->ptr != NULL)
        blinkers--;
    sc->ptr = NULL;
    sc->func(sc->private, state);
}
```

6.14.2.5.4 Installing Static State When the caller passes NULL for `sb`, the LED should be set to a static on/off state without blinking.

Clearing pattern state: Setting `sc->str = NULL` marks that no pattern string exists. This field is checked during cleanup to determine if memory needs freeing.

Deactivating the timer: The check `if (sc->ptr != NULL)` detects whether this LED was previously executing a pattern. If so, decrementing `blinkers--` records that one fewer LED needs timer servicing. If this was the last active LED, `blinkers` drops to zero and the timer callback won't reschedule itself, stopping timer firings.

Setting to NULL: `sc->ptr = NULL` marks this LED as inactive. The timer callback's first check (`if (sc->ptr == NULL) continue;`) will skip this LED on all future ticks.

Immediate hardware update: `sc->func(sc->private, state)` invokes the hardware driver's control callback to set the LED to the requested state (0 for off, 1 for on). Unlike pattern mode where the timer controls LED changes, static mode requires immediate hardware update since no timer is involved.

```
sc->count = 0;
```

6.14.2.5.5 Resetting Delay Counter The delay counter is zeroed regardless of which path was taken. If a pattern is being installed, starting with `count = 0` ensures the first pattern character executes immediately without inherited delay. If static state is being set, zeroing is harmless since the field isn't used when `ptr` is `NULL`.

```
*sb = sb2;
return(0);
```

6.14.2.5.6 Returning the Old Pattern The function returns the previous pattern buffer through the double pointer. The caller receives either:

- `NULL` if no previous pattern existed
- The old `sbuf` if a pattern is being replaced

The caller must check this returned value and call `sbuf_delete()` if non-`NULL` to free the buffer's memory. This ownership transfer pattern prevents memory leaks while avoiding unnecessary copying.

The return value `0` signals success. This function currently cannot fail, but returning an error code provides future extensibility if validation or resource allocation were added.

6.14.2.5.7 State Transition Examples **Setting initial pattern on inactive LED:**

```
Before: sc->ptr = NULL, sc->spec = NULL, blinkers = 0
Call:   led_state(sc, &pattern_sb, 0)
After:  sc->ptr = sc->str, sc->spec = pattern_sb, blinkers = 1
        Old NULL returned to caller
```

Replacing one pattern with another:

```
Before: sc->ptr = old_str, sc->spec = old_sb, blinkers = 3
Call:   led_state(sc, &new_sb, 0)
After:  sc->ptr = new_str, sc->spec = new_sb, blinkers = 3
        Old old_sb returned to caller for deletion
```

Changing from pattern to static:

```
Before: sc->ptr = pattern_str, sc->spec = pattern_sb, blinkers = 1
Call:   led_state(sc, &NULL_ptr, 1)
After:  sc->ptr = NULL, sc->spec = NULL, blinkers = 0
        Hardware callback invoked with state=1 (on)
        Old pattern_sb returned to caller for deletion
```

Setting static state on already-static LED:

```

Before: sc->ptr = NULL, sc->spec = NULL, blinkers = 0
Call:   led_state(sc, &NULL_ptr, 0)
After:  sc->ptr = NULL, sc->spec = NULL, blinkers = 0
        Hardware callback invoked with state=0 (off)
        Old NULL returned to caller

```

6.14.2.5.8 Thread Safety Considerations This function operates under the protection of `led_mtx`, acquired by the caller (write handler or `led_set()`). The mutex serializes state changes and protects:

- The `blinkers` counter from races when multiple LEDs change state simultaneously
- Individual LED fields (`ptr`, `str`, `spec`, `count`) from corruption
- The relationship between `blinkers` count and actual active patterns

Without the mutex, two simultaneous writes could both increment `blinkers`, creating an incorrect count. Or one thread could free `sc->str` while the timer callback traverses it, causing a use-after-free crash.

6.14.2.5.9 Memory Management Discipline The function demonstrates careful memory management:

Ownership transfer: The caller gives up the new `sbuf` and receives the old one, establishing clear ownership at all times.

Paired allocation/free: Every `strdup()` has a corresponding `free()`, preventing leaks even when patterns are repeatedly replaced.

NULL tolerance: All checks handle NULL pointers gracefully, allowing transitions to/from uninitialized state without special cases.

This discipline prevents the common pattern-replacement bug where updating state leaks the old pattern's memory.

6.14.2.6 4) Parse user commands into patterns: `led_parse()`

```

116: static int
117: led_parse(const char *s, struct sbuf **sb, int *state)
118: {
119:     int i, error;
121:     /* '0' or '1' means immediate steady off/on (no pattern). */
124:     if (*s == '0' || *s == '1') {
125:         *state = *s & 1;
126:         return (0);
127:     }
129:     *state = 0;
130:     *sb = sbuf_new_auto();

```

```

131:     if (*sb == NULL)
132:         return (ENOMEM);
133:     switch(s[0]) {
135:     case 'f': /* blink (default 100/100ms); 'f2' => 200/200ms */
136:         if (s[1] >= '1' && s[1] <= '9') i = s[1] - '1'; else i = 0;
137:         sbuf_printf(*sb, "%c%c", 'A' + i, 'a' + i);
138:         break;
149:     case 'd': /* "digits": flash out numbers 0..9 */
150:         for(s++; *s; s++) {
151:             if (!isdigit(*s)) continue;
152:             i = *s - '0'; if (i == 0) i = 10;
156:             for (; i > 1; i--) sbuf_cat(*sb, "Aa");
158:             sbuf_cat(*sb, "Aj");
159:         }
160:         sbuf_cat(*sb, "jj");
161:         break;
162:     /* other small patterns elided for brevity in this excerpt ... */
187:     case 'm': /* Morse: '.' -> short, '-' -> long, ' ' -> space */
188:         for(s++; *s; s++) {
189:             if (*s == '.') sbuf_cat(*sb, "aA");
190:             else if (*s == '-') sbuf_cat(*sb, "aC");
191:             else if (*s == ' ') sbuf_cat(*sb, "b");
192:             else if (*s == '\n') sbuf_cat(*sb, "d");
193:         }
198:         sbuf_cat(*sb, "j");
199:         break;
200:     default:
201:         sbuf_delete(*sb);
202:         return (EINVAL);
203:     }
204:     error = sbuf_finish(*sb);
205:     if (error != 0 || sbuf_len(*sb) == 0) {
206:         *sb = NULL;
207:         return (error);
208:     }
209:     return (0);
210: }

```

6.14.2.6.1 Pattern Parser: User Commands to Internal Codes The `led_parse` function translates human-friendly pattern specifications from userspace into the internal timing code language that the timer callback interprets. This parser allows users to write simple commands like “f” for flashing or “m...—...” for morse code, which are expanded into sequences of timing codes like “AaAa” or “aAaAaCaCaC”.

```
static int
led_parse(const char *s, struct sbuf **sb, int *state)
{
    int i, error;

    /* '0' or '1' means immediate steady off/on (no pattern). */
    if (*s == '0' || *s == '1') {
        *state = *s & 1;
        return (0);
    }
}
```

6.14.2.6.2 Function Signature and Quick Static Path **Parameters:** The parser receives three values:

- `s` - the user's input string from the write operation
- `sb` - pointer to a pointer where the allocated string buffer will be returned
- `state` - pointer where static state (0 or 1) is returned for non-pattern commands

Fast path for static state: Commands “0” and “1” request static off and on respectively. The expression `*s & 1` extracts the low bit of the ASCII character: ‘0’ (0x30) & 1 = 0, ‘1’ (0x31) & 1 = 1. This value is written to `*state` and the function returns immediately without allocating a string buffer. The caller receives `*sb = NULL` (never assigned) and knows to use `led_state()` with static mode.

This fast path handles the most common case efficiently, toggling LEDs on or off without complex timing.

```
*state = 0;
*sb = sbuf_new_auto();
if (*sb == NULL)
    return (ENOMEM);
```

6.14.2.6.3 String Buffer Allocation For pattern commands, a string buffer is needed to build the internal code sequence.

Default state: Setting `*state = 0` provides a default in case the pattern is used, though this value is ignored when `*sb` is non-NULL.

Creating auto-sizing buffer: `sbuf_new_auto()` allocates a string buffer that automatically grows as data is appended. This eliminates the need to pre-calculate pattern length. Morse code for a long message might produce a very long code sequence, but the buffer expands as needed.

Handling allocation failure: If memory is exhausted, the function returns `ENOMEM` immediately. The caller checks this error and propagates it to userspace, where the write operation fails with “Cannot allocate memory.”

```
switch(s[0]) {
```

6.14.2.6.4 Pattern Dispatch The first character determines the pattern type. Each case implements a different pattern language, expanding user input into timing codes.

```
case 'f': /* blink (default 100/100ms); 'f2' => 200/200ms */
    if (s[1] >= '1' && s[1] <= '9') i = s[1] - '1'; else i = 0;
    sbuf_printf(*sb, "%c%c", 'A' + i, 'a' + i);
    break;
```

6.14.2.6.5 Flash Pattern: Simple Blinking The ‘f’ command creates a symmetric blink pattern, with equal on and off times.

Speed modifier: If a digit follows ‘f’, it specifies the blink speed:

- “f” or “f1” -> i = 0 -> pattern “Aa” -> 0.1s on, 0.1s off (~2.5Hz)
- “f2” -> i = 1 -> pattern “Bb” -> 0.2s on, 0.2s off (~1.25Hz)
- “f3” -> i = 2 -> pattern “Cc” -> 0.3s on, 0.3s off (~0.83Hz)
- ...
- “f9” -> i = 8 -> pattern “Ii” -> 0.9s on, 0.9s off (~0.56Hz)

Pattern construction: `sbuf_printf(*sb, "%c%c", 'A' + i, 'a' + i)` generates two characters: an uppercase letter (on state) followed by the corresponding lowercase letter (off state). Both use the same duration, creating symmetric blinking.

This simple two-character pattern repeats indefinitely, providing the classic blink indicator effect.

```
case 'd': /* "digits": flash out numbers 0..9 */
    for(s++; *s; s++) {
        if (!isdigit(*s)) continue;
        i = *s - '0'; if (i == 0) i = 10;
        for (; i > 1; i--) sbuf_cat(*sb, "Aa");
        sbuf_cat(*sb, "Aj");
    }
    sbuf_cat(*sb, "jj");
    break;
```

6.14.2.6.6 Digit Flash Pattern: Counting Blinks The ‘d’ command followed by digits creates patterns that visually “count” by flashing the LED.

Parsing digits: The loop advances past the ‘d’ command character (`s++`) and examines each subsequent character. Non-digits are silently skipped with `continue`, allowing “d1x2y3” to be interpreted as “d123”.

Digit mapping: `i = *s - '0'` converts ASCII digit to numeric value. The special case `if (i == 0) i = 10` treats zero as ten flashes rather than no flashes, making it distinguishable from the inter-digit pause.

Flash generation: For digit value `i`:

- Generate `i-1` quick flashes: `for (; i > 1; i--) sbuf_cat(*sb, "Aa")`
- Add one longer flash: `sbuf_cat(*sb, "Aj")`

Example for digit 3: two quick flashes “AaAa” plus one 1-second flash “Aj”.

Digit separation: After all digits are processed, `sbuf_cat(*sb, "jj")` appends a 2-second pause before the pattern repeats, clearly separating repetitions.

Result: Command “d12” generates pattern “AjAjAaAjjj” meaning: 1-second flash (digit 1), pause, quick flash then 1-second flash (digit 2), long pause, repeat. This allows reading numbers from LED blinks, valid for diagnostic codes.

```
case 'm': /* Morse: '.' -> short, '-' -> long, ' ' -> space */
    for(s++; *s; s++) {
        if (*s == '.') sbuf_cat(*sb, "aA");
        else if (*s == '-') sbuf_cat(*sb, "aC");
        else if (*s == ' ') sbuf_cat(*sb, "b");
        else if (*s == '\n') sbuf_cat(*sb, "d");
    }
    sbuf_cat(*sb, "jj");
    break;
```

6.14.2.6.7 Morse Code Pattern The ‘m’ command interprets the following characters as morse code elements.

Morse element mapping:

- ‘.’ (dot) -> “aA” -> 0.1s off, 0.1s on (short flash)
- ‘-’ (dash) -> “aC” -> 0.1s off, 0.3s on (long flash)
- ‘ ’ (space) -> “b” -> 0.2s off (word separator)
- ‘\n’ (newline) -> “d” -> 0.4s off (long pause between messages)

Standard morse timing: International morse code specifies:

- Dot: 1 unit
- Dash: 3 units
- Gap between elements: 1 unit
- Gap between letters: 3 units (approximated by the trailing pause in each letter)
- Gap between words: 7 units (space character)

The pattern “aA” gives dot (1 unit off, 1 unit on), “aC” gives dash (1 unit off, 3 units on), with each unit being 0.1 seconds.

Pattern termination: `sbuf_cat(*sb, "j")` adds a 1-second pause before the message repeats, separating consecutive transmissions.

Example: Command “m... —” (SOS) generates “aAaAaCaCaC” meaning: dot-dot-dot, dash-dash-dash, repeat.

```
default:
    sbuf_delete(*sb);
    return (EINVAL);
}
```

6.14.2.6.8 Error Handling for Unknown Commands If the first character doesn’t match any known pattern type, the function rejects the command. The allocated string buffer is freed with `sbuf_delete()` to prevent leaks, and `EINVAL` (invalid argument) is returned to indicate bad user input.

The write operation will fail and return -1 to userspace with `errno = EINVAL`, informing the user that their command syntax is incorrect.

```
error = sbuf_finish(*sb);
if (error != 0 || sbuf_len(*sb) == 0) {
    *sb = NULL;
    return (error);
}
return (0);
```

6.14.2.6.9 Finalizing the Pattern String Sealing the buffer: `sbuf_finish()` finalizes the string buffer, null-terminating it and marking it read-only. After this call, the buffer’s contents can be extracted with `sbuf_data()` but no further appends are allowed.

Validation: Two error conditions are checked:

- `error != 0` - `sbuf_finish()` failed, typically due to memory exhaustion during a buffer resize
- `sbuf_len(*sb) == 0` - the pattern is empty, which shouldn’t happen but is checked defensively

If either condition holds, the buffer is unusable. Setting `*sb = NULL` signals the caller that no pattern was generated, and the error code is returned. The caller must not attempt to use or free the buffer; it was already freed by `sbuf_finish()` on error.

Success: Returning 0 with `*sb` pointing to a valid buffer signals successful parsing. The caller now owns the buffer and must eventually free it with `sbuf_delete()`.

6.14.2.6.10 Pattern Language Summary The parser supports several pattern languages, each optimized for different use cases:

Command	Purpose	Example	Result
0, 1	Static state	“1”	LED on steady
f[1-9]	Symmetric blink	“f”	Fast blink
d[digits]	Count by flashes	“d42”	4 flashes, 2 flashes
m[morse]	Morse code	“msos”	... — ...

This variety allows users to express intent naturally without memorizing timing code syntax. The write handler accepts simple commands; the parser expands them to precise timing sequences; the timer executes those sequences.

6.14.2.7 5.1) The write entry point: `echo "cmd" > /dev/led/<name>`

User space **writes a command string** to the device. The driver parses it and updates the LED’s state. The **shape** is exactly what you’ll write later: `uiomove()` the user buffer, parse, then update the softc under a lock.

```

212: static int
213: led_write(struct cdev *dev, struct uio *uio, int ioflag)
214: {
215:     struct ledsc    *sc;
216:     char *s;
217:     struct sbuf *sb = NULL;
218:     int error, state = 0;
220:     if (uio->uio_resid > 512)
221:         return (EINVAL);
222:     s = malloc(uio->uio_resid + 1, M_DEVBUF, M_WAITOK);
223:     s[uio->uio_resid] = '\0';
224:     error = uiomove(s, uio->uio_resid, uio);
225:     if (error) { free(s, M_DEVBUF); return (error); }
226:     /* parse -> (sb pattern) or (state only) */
227:     error = led_parse(s, &sb, &state);
228:     free(s, M_DEVBUF);
229:     if (error) return (error);
230:     mtx_lock(&led_mtx);
231:     sc = dev->si_drv1;
232:     if (sc != NULL)
233:         error = led_state(sc, &sb, state);
234:     mtx_unlock(&led_mtx);
235:     if (sb != NULL) sbuf_delete(sb);
236:     return (error);
237: }
```

6.14.2.7.1 Write Handler: User Command Interface The `led_write` function implements the character device write operation for `/dev/led/*` devices. When a user writes a pattern command like “f” or “m...—...” to an LED device node, this function copies the data from userspace, parses it into an internal format, and installs the new LED pattern.

```
static int
led_write(struct cdev *dev, struct uio *uio, int ioflag)
{
    struct ledsc    *sc;
    char *s;
    struct sbuf *sb = NULL;
    int error, state = 0;

    if (uio->uio_resid > 512)
        return (EINVAL);
    s = malloc(uio->uio_resid + 1, M_DEVBUF, M_WAITOK);
    s[uio->uio_resid] = '\0';
```

6.14.2.7.2 Size Validation and Buffer Allocation **Size limit enforcement:** The check `uio->uio_resid > 512` rejects writes larger than 512 bytes. LED patterns are short text commands, even complex morse code messages rarely exceed a few dozen characters. This limit prevents memory exhaustion from malicious or buggy programs attempting multi-megabyte writes.

Returning `EINVAL` signals invalid argument to userspace. The write fails immediately without allocating memory or touching the LED state.

Temporary buffer allocation: Unlike the null driver’s `null_write`, which never accesses user data, the LED driver must examine the written bytes to parse commands. The allocation reserves `uio->uio_resid + 1` bytes, the exact write size plus one byte for null termination.

The `M_DEVBUF` allocation type is generic for device driver temporary buffers. The `M_WAITOK` flag allows the allocation to sleep if memory is temporarily unavailable, which is acceptable since this is a blocking write operation with no stringent latency requirements.

Null termination: Setting `s[uio->uio_resid] = '\0'` ensures the buffer is a proper C string. The `uiomove` call will fill the first `uio->uio_resid` bytes with user data, and this assignment adds the terminator immediately after. String functions like those used in parsing require null-terminated strings.

```
error = uiomove(s, uio->uio_resid, uio);
if (error) { free(s, M_DEVBUF); return (error); }
```

6.14.2.7.3 Copying Data from Userspace The `uiomove` function transfers `uio->uio_resid` bytes from the user’s buffer (described by `uio`) into the kernel buffer `s`. This is the same function used in the null and zero drivers for data transfer between address spaces.

Error handling: If `uiomove` fails (typically `EFAULT` for an invalid user pointer), the allocated buffer is freed immediately with `free(s, M_DEVBUF)` and the error propagates to userspace. The write fails without modifying LED state, and the temporary buffer doesn't leak.

This cleanup discipline is critical, kernel code must free allocated memory on all error paths, not just success paths.

```
/* parse -> (sb pattern) or (state only) */
error = led_parse(s, &sb, &state);
free(s, M_DEVBUF);
if (error) return (error);
```

6.14.2.7.4 Parsing the Command Translation to internal format: The `led_parse` function interprets the user's command string, producing either:

- A string buffer (`sb`) containing timing codes for pattern mode
- A state value (0 or 1) for static on/off mode

The parser determines which mode based on the command's first character. Commands like "f", "d", "m" generate patterns; commands "0" and "1" set static state.

Immediate cleanup: The temporary buffer `s` is no longer needed after parsing, whether parsing succeeded or failed, the original command string is no longer required. Freeing it immediately rather than waiting until function end reduces memory consumption in the common case where parsing succeeds and additional processing follows.

Error propagation: If parsing fails (unrecognized command, memory exhaustion, empty pattern), the error is returned to userspace. The write operation fails before acquiring locks or modifying LED state. Users see the write fail with `errno` set to the parser's error code (typically `EINVAL` for bad syntax or `ENOMEM` for resource exhaustion).

```
mtx_lock(&led_mtx);
sc = dev->si_drv1;
if (sc != NULL)
    error = led_state(sc, &sb, state);
mtx_unlock(&led_mtx);
```

6.14.2.7.5 Installing the New State Acquiring the lock: The `led_mtx` mutex protects the LED list and per-LED state from concurrent modification. Multiple threads might write to different LEDs simultaneously, or a write might race with timer callbacks updating blink patterns. The mutex serializes these operations.

Retrieving the LED context: `dev->si_drv1` provides the `ledsc` structure for this device, established during `led_create()`. This pointer links the character device node to its LED state.

Defensive NULL check: The condition `if (sc != NULL)` guards against a race where the LED is being destroyed while a write is in progress. If `led_destroy()` has cleared `si_drv1` but the write handler is still executing, this check prevents dereferencing NULL. In practice, proper reference counting makes this unlikely, but defensive checks prevent kernel panics.

State installation: `led_state(sc, &sb, state)` installs the new pattern or static state. This function:

- Swaps the new pattern buffer with the old one
- Updates blinkers counter if the LED transitions between active and inactive
- Calls the hardware driver's callback for static state changes
- Returns the old pattern buffer via the `sb` pointer

Lock release: After state installation completes, the mutex is released. Other threads blocked on LED operations can now proceed. Lock hold time is minimal, only the state swap and counter update, not the potentially slow parsing that happened earlier.

```
if (sb != NULL) sbuf_delete(sb);  
return (error);
```

6.14.2.7.6 Cleanup and Return Freeing the old pattern: After `led_state` returns, `sb` points to the old pattern buffer (or NULL if no previous pattern existed). The code must free this buffer to prevent memory leaks. Each pattern installation generates one buffer to free from the previous pattern.

The check `if (sb != NULL)` handles both initial pattern installation (no previous pattern) and static state commands (parser never allocated a buffer). Only actual pattern buffers need deletion.

Success return: Returning error (typically 0 for success) completes the write operation. The userspace `write(2)` call returns the number of bytes written (the original `uio->uio_resid`), indicating success.

6.14.2.7.7 Complete Write Flow The whole sequence from userspace write to LED state change (the flow below uses a theoretical device, to illustrate the flow):

```
User: echo "f" > /dev/led/disk0  
->  
led_write() called by kernel  
->  
Validate size (< 512 bytes)  
->  
Allocate temporary buffer  
->  
Copy "f\n" from userspace  
->  
Parse "f" -> timing code "Aa"  
->  
Free temporary buffer
```

```
->
Lock led_mtx
->
Find LED via dev->si_drv1
->
Install new pattern "Aa"
->
Increment blinkers (0 -> 1)
->
Schedule timer if needed
->
Unlock led_mtx
->
Free old pattern (NULL)
->
Return success
->
User: write() returns 2 bytes
```

On the next timer tick (0.1 seconds later), the LED begins blinking at ~2.5Hz, alternating on and off every 0.1 seconds.

6.14.2.7.8 Error Handling Paths The function has multiple error exits, each with proper cleanup:

Size validation failure:

```
Check uio_resid > 512 -> return EINVAL
(nothing allocated yet, no cleanup needed)
```

Allocation failure:

```
malloc() returns NULL -> kernel panics (M_WAITOK)
(M_WAITOK means "wait for memory, never fail")
```

Copyin failure:

```
uiomove() fails -> free(s) -> return EFAULT
(temporary buffer freed, no other resources allocated)
```

Parse failure:

```
led_parse() fails -> free(s) -> return EINVAL
(temporary buffer freed, no string buffer created)
```

State installation success:

```
led_state() succeeds -> sbuf_delete(old) -> return 0
(old pattern freed, new pattern installed)
```

Every error path frees all allocated resources, preventing memory leaks regardless of where failure occurs.

6.14.2.7.9 Contrast with null.c The null driver’s `null_write` was trivial: set `uio_resid = 0` and return. The LED driver’s write handler is substantially more complex because:

User input requires parsing: Commands like “f” and “m...” must be interpreted, not just discarded.

State must be modified: New patterns affect LED behavior, requiring coordination with timer callbacks.

Memory must be managed: Buffers are allocated, swapped, and freed across function boundaries.

Synchronization is required: Multiple writers and timer callbacks must coordinate via mutexes.

This increased complexity reflects the LED driver’s role as infrastructure supporting rich user interaction with physical hardware, not just a simple data sink.

6.14.2.8 5.2) Kernel API: Programmatic LED Control

```
240: int
241: led_set(char const *name, char const *cmd)
...
247:     error = led_parse(cmd, &sb, &state);
...
251:     LIST_FOREACH(sc, &led_list, list) {
252:         if (strcmp(sc->name, name) == 0) break;
253:     }
254:     if (sc != NULL) error = led_state(sc, &sb, state);
255:     else error = ENOENT;
```

The `led_set` function provides a kernel-facing API that allows other kernel code to control LEDs without going through the character device interface. This enables drivers, kernel subsystems, and system event handlers to manipulate LEDs directly using the same pattern language available to userspace.

```
int
led_set(char const *name, char const *cmd)
```

6.14.2.8.1 Function Signature and Purpose **Parameters:** The function receives two strings:

- `name` - the LED identifier, matching the name used during `led_create()` (e.g., “disk0”, “power”)
- `cmd` - the pattern command string, same syntax as userspace writes (e.g., “P”, “1”, “m...—...”)

Return value: Zero for success, or an `errno` value for failure (`EINVAL` for parse errors, `ENOENT` for unknown LED name, `ENOMEM` for allocation failure).

Use cases: Kernel code can call this function to:

- Indicate disk activity: `led_set("disk0", "f")` to blink during I/O
- Show system status: `led_set("power", "1")` to turn on power LED after boot completes
- Signal error conditions: `led_set("status", "m...- - - . . .")` to flash SOS pattern
- Implement heartbeat: `led_set("heartbeat", "Uu")` for 1Hz toggle showing system liveness

```
error = led_parse(cmd, &sb, &state);
```

6.14.2.8.2 Parsing the Command The function reuses the same parser as the write handler. Pattern strings are interpreted identically whether coming from userspace via `write(2)` or from kernel code via `led_set()`.

This code reuse ensures consistency; a command that works in one context works in the other. The parser handles all the complexity of expanding “P” to “Aa” or “m...” to “aA”, so kernel callers don’t need to understand the internal timing code format.

If parsing fails (bad command syntax, memory exhaustion), the error is recorded in the `error` variable and checked later. The function continues to acquire the lock even on parse failure because the lock must be held to safely return without leaking the buffer.

```
LIST_FOREACH(sc, &led_list, list) {  
    if (strcmp(sc->name, name) == 0) break;  
}  
if (sc != NULL) error = led_state(sc, &sb, state);  
else error = ENOENT;
```

6.14.2.8.3 Finding the Named LED **Linear search:** The `LIST_FOREACH` macro walks the global LED list, comparing each LED’s name to the requested name with `strcmp()`. The loop terminates early with `break` when a match is found, leaving `sc` pointing to the matching LED.

Why linear search?: For small lists (typically 5-20 LEDs per system), linear search is faster than hash table overhead. The code simplicity and cache-friendly sequential access outweigh the $O(n)$ complexity. Systems with hundreds of LEDs would benefit from a hash table, but such systems are rare.

Handling not found: If the loop completes without breaking, no LED matched the name and `sc` remains `NULL` (from the `LIST_FOREACH` initialization). Setting `error = ENOENT` (no such file or directory) signals that the named LED doesn’t exist.

Installing state: When a match is found (`sc != NULL`), `led_state()` is called to install the new pattern or static state, using the same state installation function as the write handler. The return value overwrites any parse error, if parsing succeeded but state installation failed, the installation error takes precedence.

6.14.2.8.4 Critical Code Omitted in Fragment The provided fragment omits several critical lines visible in the complete function:

Lock acquisition (before the `LIST_FOREACH` loop in `led_set`):

```
mtx_lock(&led_mtx);
```

The LED list must be locked before traversal to prevent concurrent modifications. If one thread is searching the list while another thread destroys an LED, the search might access freed memory. The mutex serializes list access.

Lock release and cleanup (after the state-install call in `led_set`):

```
mtx_unlock(&led_mtx);
if (sb != NULL)
    sbuf_delete(sb);
return (error);
```

After the state installation attempt, the mutex is released and the old pattern buffer (returned via `sb` by `led_state()`) is freed. This cleanup mirrors the write handler's buffer management.

6.14.2.8.5 Comparison with Write Handler Both `led_write` and `led_set` follow the same pattern:

```
Parse command -> Acquire lock -> Find LED -> Install state -> Release lock ->
↳ Cleanup
```

The key differences:

Aspect	<code>led_write</code>	<code>led_set</code>
Caller	Userspace via <code>write(2)</code>	Kernel code
Input source	<code>uio</code> structure	Direct string pointers
LED identification	<code>dev->si_drv1</code>	Name lookup
Size validation	Limit 512 bytes	No explicit limit (caller responsible)
Error reporting	<code>errno</code> to userspace	Return value to caller

The write handler uses the device pointer to find the LED directly (single device, single LED). The kernel API uses name lookup to support arbitrary LED selection from any kernel context.

6.14.2.8.6 Example Usage Patterns Disk driver indicating activity:

```

void
disk_start_io(struct disk_softc *sc)
{
    /* Begin I/O operation */
    led_set(sc->led_name, "f"); // Start blinking
}

void
disk_complete_io(struct disk_softc *sc)
{
    /* I/O completed */
    led_set(sc->led_name, "0"); // Turn off
}

```

System initialization sequence:

```

void
system_boot_complete(void)
{
    led_set("power", "1"); // Solid on: system ready
    led_set("status", "0"); // Off: no errors
    led_set("heartbeat", "Uu"); // 1Hz toggle: alive
}

```

Error indication:

```

void
critical_error_handler(int error_code)
{
    char pattern[16];
    snprintf(pattern, sizeof(pattern), "d%d", error_code);
    led_set("status", pattern); // Flash error code
}

```

6.14.2.8.7 Thread Safety The function is thread-safe through mutex protection. Multiple threads can call `led_set()` concurrently:

Scenario: Thread A sets “disk0” to “f” while Thread B sets “power” to “1”.

Thread A	Thread B
Parse "f" -> "Aa"	Parse "1" -> state=1
Lock led_mtx	(blocks on lock)

```

Find "disk0"      ...
Install pattern   ...
Unlock led_mtx    Acquire lock
Delete old buffer Find "power"
Return           Install state
                  Unlock led_mtx
                  Delete old buffer
                  Return

```

The mutex serializes list traversal and state modification, preventing corruption. Both operations complete successfully without interference.

6.14.2.8.8 Error Handling The function can fail in several ways:

Parse error:

```
led_set("disk0", "invalid") // Returns EINVAL
```

LED not found:

```
led_set("nonexistent", "f") // Returns ENOENT
```

Memory exhaustion:

```
led_set("disk0", "m..." /* very long morse */) // Returns ENOMEM
```

Kernel callers should check the return value and handle errors appropriately, though in practice LED control failures are rarely fatal, the system continues operating, just without visual indicators.

6.14.2.8.9 Why Both APIs Exist The dual interface (character device + kernel API) serves different needs:

Character device (/dev/led/*):

- User scripts and programs
- System administrators
- Testing and debugging
- Interactive control

Kernel API (led_set()):

- Automated responses to events
- Driver-integrated indicators
- System state visualization
- Performance-critical paths (no system call overhead)

This pattern, exposing functionality through both userspace devices and kernel APIs, appears throughout FreeBSD. The LED subsystem provides a clean example of how to structure such dual-interface services.

6.14.2.9 6) Hook into devfs and export the write method

```
272: static struct cdevsw led_cdevsw = {
273:     .d_version =    D_VERSION,
274:     .d_write =    led_write,
275:     .d_name =    "LED",
276: };
```

6.14.2.9.1 Character Device Switch Table The `led_cdevsw` structure defines the character device operations for all LED device nodes. Unlike the null driver which had three separate `cdevsw` structures for three devices, the LED driver uses a single `cdevsw` shared by all dynamically created `/dev/led/*` devices.

```
static struct cdevsw led_cdevsw = {
    .d_version =    D_VERSION,
    .d_write =    led_write,
    .d_name =    "LED",
};
```

6.14.2.9.2 Structure Definition `d_version = D_VERSION`: The mandatory version field ensures binary compatibility between the driver and the kernel's device framework. All `cdevsw` structures must include this field.

`d_write = led_write`: The only operation explicitly defined. When userspace calls `write(2)` on any `/dev/led/*` device, the kernel invokes this function. The `led_write` handler parses pattern commands and updates LED state.

`d_name = "LED"`: The device class name appearing in kernel messages and accounting. This string identifies the driver type, though individual devices have their own specific names (like “disk0” or “power”).

6.14.2.9.3 Minimal Operation Set Notice what's **not** defined:

No `d_read`: LEDs are output-only devices. Reading from `/dev/led/disk0` is meaningless, there's no state to query, no data to retrieve. Omitting `d_read` causes read attempts to fail with `ENODEV` (operation not supported by device).

No `d_open` / `d_close`: LED devices require no per-open initialization or cleanup. Multiple processes can write to the same LED simultaneously (serialized by the mutex), and closing the device requires no state teardown. The kernel's default handlers suffice.

No `d_ioctl`: Unlike the null driver which supported terminal ioctls, LED devices have no control operations beyond writing patterns. All configuration happens through the write interface.

No `d_poll` / `d_kqfilter`: LEDs are write-only, so there's no condition to wait for. Polling for writability would always return “ready” since writes never block (beyond mutex acquisition), making poll support useless.

This minimalism contrasts with the null driver's more complete interface (which included ioctl handlers) and demonstrates that `cdevsw` structures need only provide operations that make sense for the device type.

6.14.2.9.4 Shared Across Devices A critical distinction from the null driver: this **single** `cdevsw` serves **all** LED devices. When the system has three LEDs registered:

```
/dev/led/disk0    ->  led_cdevsw
/dev/led/power    ->  led_cdevsw
/dev/led/status   ->  led_cdevsw
```

All three device nodes share the same function pointer table. The `led_write` function determines which LED is being written to by examining `dev->si_drv1`, which points to the specific LED's `ledsc` structure.

This sharing is possible because:

- All LEDs support the same operations (write pattern commands)
- Per-device state is accessed through `si_drv1`, not through different functions
- The same parsing and state installation logic applies to every LED

6.14.2.9.5 Contrast with null.c The null driver defined three separate `cdevsw` structures:

```
static struct cdevsw full_cdevsw = { ... };
static struct cdevsw null_cdevsw = { ... };
static struct cdevsw zero_cdevsw = { ... };
```

Each had different function assignments because the devices had different behavior (`full_write` vs. `null_write`, `nullop` vs. `zero_read`). The devices were fundamentally different types.

The LED driver's devices are all the same type, they're LEDs that accept pattern commands. The only differences are:

- Device name ("disk0" vs. "power")
- Hardware control callback (different for each physical LED)
- Current pattern state (independent per LED)

These differences are stored in per-device `ledsc` structures, not encoded in separate function tables. This design scales elegantly: registering 100 LEDs doesn't require 100 `cdevsw` structures, just 100 `ledsc` instances sharing one `cdevsw`.

6.14.2.9.6 Usage in Device Creation When a hardware driver calls `led_create()`, the code creates a device node:

```
sc->dev = make_dev(&led_cdevsw, sc->unit,
    UID_ROOT, GID_WHEEL, 0600, "led/%s", name);
```

The `&led_cdevsw` parameter provides the function dispatch table. All created devices reference the same structure, `make_dev()` doesn't copy it, just stores the pointer. This means:

- Zero memory overhead per device for the function table
- Changes to `led_write` (during development) automatically affect all devices
- The `cdevsw` must remain valid for the system's lifetime (hence `static` storage)

6.14.2.9.7 Device Identification With all devices sharing one `cdevsw`, how does `led_write` distinguish which LED is being written to? The device linkage:

```
// In led_create():
sc->dev = make_dev(&led_cdevsw, ...);
sc->dev->si_drv1 = sc; // Link device to its ledsc

// In led_write():
sc = dev->si_drv1; // Retrieve the ledsc
```

The `si_drv1` field (set during `led_create()`) creates a per-device pointer to the unique `ledsc` structure. Though all devices share the same `cdevsw` and thus the same `led_write` function, each invocation receives a different `dev` parameter, which provides access to device-specific state through `si_drv1`.

This pattern, shared function table, per-device state pointer, is the standard approach for drivers managing multiple similar devices. It combines efficiency (one function table) with flexibility (device-specific behavior through the state pointer).

6.14.2.10 7) Create per-LED device nodes

```
278: struct cdev *
279: led_create(led_t *func, void *priv, char const *name)
280: {
281:     return (led_create_state(func, priv, name, 0));
282: }
283:
284: struct cdev *
285: led_create_state(led_t *func, void *priv, char const *name, int state)
286: {
287:     struct ledsc *sc;
288:     sc = malloc(sizeof *sc, M_LED, M_WAITOK | M_ZERO);
289:     sx_xlock(&led_sx);
290:     sc->name = strdup(name, M_LED);
291:     sc->unit = alloc_unr(led_unit);
292:     sc->private = priv;
293:     sc->func = func;
294:     sc->dev = make_dev(&led_cdevsw, sc->unit,
295:         UID_ROOT, GID_WHEEL, 0600, "led/%s", name);
296:     sx_xunlock(&led_sx);
297:     mtx_lock(&led_mtx);
298:     sc->dev->si_drv1 = sc;
299:     LIST_INSERT_HEAD(&led_list, sc, list);
300:     if (state != -1)
301:         sc->func(sc->private, state != 0);
302:     mtx_unlock(&led_mtx);
```

```
308:     return (sc->dev);
309: }
```

6.14.2.10.1 LED Registration: Creating Dynamic Devices The `led_create` and `led_create_state` functions form the public API that hardware drivers use to register LEDs with the subsystem. These functions allocate resources, create device nodes, and integrate the LED into the global registry, making it accessible to both userspace and kernel code.

```
struct cdev *
led_create(led_t *func, void *priv, char const *name)
{
    return (led_create_state(func, priv, name, 0));
}
```

6.14.2.10.2 Simple Registration Wrapper The `led_create` function provides a simplified interface for the common case where the LED's initial state doesn't matter. It delegates to `led_create_state` with an initial state of 0 (off), allowing hardware drivers to register LEDs with minimal code:

```
struct cdev *led;
led = led_create(my_led_callback, my_softc, "disk0");
```

This convenience wrapper follows the FreeBSD pattern of providing both simple and feature-complete versions of the same API.

```
struct cdev *
led_create_state(led_t *func, void *priv, char const *name, int state)
{
    struct ledsc      *sc;
```

6.14.2.10.3 Full Registration Function **Parameters:** The function receives four values:

- `func` - callback function that controls the physical LED hardware
- `priv` - opaque pointer passed to the callback, typically the driver's `softc`
- `name` - string identifying the LED, becomes part of `/dev/led/name`
- `state` - initial LED state: 0 (off), 1 (on), or -1 (don't initialize)

Return value: Pointer to the created `struct cdev`, which the hardware driver should store for later use with `led_destroy()`. If creation fails, the function panics (due to `M_WAITOK` allocation) rather than returning `NULL`.

```
sc = malloc(sizeof *sc, M_LED, M_WAITOK | M_ZERO);
```

6.14.2.10.4 Allocating LED State The `softc` structure is allocated to track this LED's state. The `M_ZERO` flag zeroes all fields, providing safe defaults:

- Pointer fields (`name`, `dev`, `spec`, `str`, `ptr`) are `NULL`
- Numeric fields (`unit`, `count`) are zero
- The `list` entry is zeroed (will be initialized by `LIST_INSERT_HEAD`)

The `M_WAITOK` flag means the allocation can sleep waiting for memory, which is acceptable since LED registration happens during driver attach (a blocking context). If memory is truly exhausted, the kernel panics, LED registration is considered essential enough that failure is not recoverable.

```
sx_xlock(&led_sx);
sc->name = strdup(name, M_LED);
sc->unit = alloc_unr(led_unit);
sc->private = priv;
sc->func = func;
sc->dev = make_dev(&led_cdevsw, sc->unit,
    UID_ROOT, GID_WHEEL, 0600, "led/%s", name);
sx_xunlock(&led_sx);
```

6.14.2.10.5 Device Creation Under Exclusive Lock **Exclusive lock acquisition:** The `sx_xlock` call acquires the shared/exclusive lock in exclusive (write) mode. This serializes all device creation and destruction operations, preventing races where two threads simultaneously create devices with the same name or allocate the same unit number.

Name duplication: `strdup(name, M_LED)` allocates a copy of the name string. The caller's string may be temporary (stack buffer or string literal), so a persistent copy is needed for the LED's lifetime. This copy will be freed in `led_destroy()`.

Unit number allocation: `alloc_unr(led_unit)` obtains a unique unit number from the global pool. This number becomes the device's minor number, ensuring `/dev/led/disk0` and `/dev/led/power` have distinct device identifiers even though they share the same major number.

Callback registration: The `private` and `func` fields are copied from parameters, establishing the connection to the hardware driver's control function. When the LED state changes (via pattern execution or static state command), `sc->func(sc->private, onoff)` will be called to manipulate the physical hardware.

Device node creation: `make_dev` creates `/dev/led/name` with the following properties:

- `&led_cdevsw` - shared character device operations (write handler)
- `sc->unit` - unique minor number for this LED
- `UID_ROOT, GID_WHEEL` - owned by root:wheel
- `0600` - read/write for owner only (root), no access for others

- "led/%s", name - device path, automatically prepends /dev/

The restrictive permissions (0600) prevent unprivileged users from controlling LEDs, which could be a security concern (information leakage through LED patterns) or nuisance (making the power LED blink rapidly).

Lock release: After device creation completes, the exclusive lock is released. Other threads can now create or destroy LEDs. The lock hold time is minimal, just the core allocation and registration, not including the earlier softc allocation which didn't need protection.

```
mtx_lock(&led_mtx);
sc->dev->si_drv1 = sc;
LIST_INSERT_HEAD(&led_list, sc, list);
if (state != -1)
    sc->func(sc->private, state != 0);
mtx_unlock(&led_mtx);
```

6.14.2.10.6 Integration Under Mutex **Mutex acquisition:** The `led_mtx` mutex protects the LED list and timer-related state. It's acquired after device creation because multiple locks with different purposes reduces contention, threads creating devices don't block threads modifying LED states.

Bidirectional linkage: Setting `sc->dev->si_drv1 = sc` creates the critical link from device node to softc. When `led_write` is called with this device, it can retrieve the softc via `dev->si_drv1`. This linkage must be established before the device is usable.

List insertion: `LIST_INSERT_HEAD(&led_list, sc, list)` adds the LED to the global registry at the head of the list. The `list` field in the softc was zeroed during allocation, and this macro initializes it properly while linking into the existing list.

Using `LIST_INSERT_HEAD` rather than `LIST_INSERT_TAIL` is arbitrary; order doesn't matter for LED list iteration. Head insertion is slightly faster (no need to find the tail), but the performance difference is negligible.

Optional initial state: If `state != -1`, the hardware callback is invoked immediately to set the LED's initial state:

- `state != 0` converts any non-zero value to boolean true (LED on)
- `state == 0` means LED off

The special value -1 means "don't initialize," leaving the LED in whatever state the hardware defaults to. This is useful when the hardware driver has already configured the LED before registration.

Lock release: After list insertion and optional initialization, the mutex is released. The LED is now fully operational; userspace can write to its device node, kernel code can call `led_set()` with its name, and timer callbacks will process any patterns.

```
return (sc->dev);
}
```

6.14.2.10.7 Return Value and Ownership The function returns the cdev pointer, which the hardware driver should store:

```
struct my_driver_softc {
    struct cdev *led_dev;
    /* other fields */
};

void
my_driver_attach(device_t dev)
{
    struct my_driver_softc *sc = device_get_softc(dev);
    /* other initialization */
    sc->led_dev = led_create(my_led_callback, sc, "disk0");
}
```

The hardware driver needs this pointer to call `led_destroy()` during detach. Without storing it, the LED would leak, its device node and resources would persist even after the hardware driver unloads.

6.14.2.10.8 Resource Allocation Summary A successful LED registration allocates:

- Softc structure (freed in `led_destroy`)
- Name string copy (freed in `led_destroy`)
- Unit number (returned to pool in `led_destroy`)
- Device node (destroyed in `led_destroy`)

All resources are cleaned up symmetrically during destruction, preventing leaks when hardware is removed.

6.14.2.10.9 Thread Safety The two-lock design enables safe concurrent operations:

Scenario: Thread A creates “disk0” while Thread B creates “power”.

Thread A	Thread B
Allocate sc1	Allocate sc2
Lock led_sx (exclusive)	(blocks on led_sx)
Create /dev/led/disk0	...
Unlock led_sx	Acquire led_sx
Lock led_mtx	Create /dev/led/power
Insert sc1 to list	Unlock led_sx
Unlock led_mtx	Lock led_mtx
	Insert sc2 to list
	Unlock led_mtx

The exclusive lock serializes device creation (preventing name/unit conflicts), while the mutex serializes list modification (preventing list corruption). Both threads complete successfully with two working LEDs.

6.14.2.10.10 Contrast with null.c The null driver's device creation happened in `null_modevent` during module load:

```
// null.c: static devices created once
full_dev = make_dev_credf(..., "full");
null_dev = make_dev_credf(..., "null");
zero_dev = make_dev_credf(..., "zero");
```

The LED driver's device creation happens dynamically on demand:

```
// led.c: devices created whenever hardware drivers request
led_create(func, priv, "disk0"); // called by disk driver
led_create(func, priv, "power"); // called by power driver
led_create(func, priv, "status"); // called by GPIO driver
```

This dynamic approach scales naturally: the system can have any number of LEDs (zero to hundreds), with devices appearing and disappearing as hardware is added and removed. The subsystem provides infrastructure, but doesn't dictate what LEDs exist, that's determined by which hardware drivers are loaded and what hardware is present.

6.14.2.11 8) Destroy per-LED device nodes

```
306: void
307: led_destroy(struct cdev *dev)
308: {
309:     struct ledsc *sc;
310:     mtx_lock(&led_mtx);
311:     sc = dev->si_drv1;
312:     dev->si_drv1 = NULL;
313:     if (sc->ptr != NULL)
314:         blinkers--;
315:     LIST_REMOVE(sc, list);
316:     if (LIST_EMPTY(&led_list))
317:         callout_stop(&led_ch);
318:     mtx_unlock(&led_mtx);
319:     sx_xlock(&led_sx);
320:     free_unr(led_unit, sc->unit);
321:     destroy_dev(dev);
322:     if (sc->spec != NULL)
323:         sbuf_delete(sc->spec);
324:     free(sc->name, M_LED);
325:     free(sc, M_LED);
```

```
328:    sx_xunlock(&led_sx);
329: }
```

6.14.2.11.1 LED Deregistration: Cleanup and Resource Release The `led_destroy` function unregisters an LED from the subsystem, reversing all operations performed during `led_create`. Hardware drivers call this function during detach to cleanly remove LEDs before the underlying hardware disappears, ensuring no dangling references or resource leaks remain.

```
void
led_destroy(struct cdev *dev)
{
    struct ledsc *sc;

    mtx_lock(&led_mtx);
    sc = dev->si_drv1;
    dev->si_drv1 = NULL;
```

6.14.2.11.2 Function Entry and Softc Retrieval **Parameter:** The function receives the `cdev` pointer returned by `led_create`. Hardware drivers typically store this pointer in their own `softc` and pass it during cleanup:

```
void
my_driver_detach(device_t dev)
{
    struct my_driver_softc *sc = device_get_softc(dev);
    led_destroy(sc->led_dev);
    /* other cleanup */
}
```

Mutex acquisition: The `led_mtx` mutex is acquired first to protect the LED list and timer state. This serializes destruction with ongoing timer callbacks and write operations.

Breaking the linkage: Setting `dev->si_drv1 = NULL` immediately severs the connection between device node and `softc`. Any write operation that started before this function was called but hasn't yet acquired the mutex will see `NULL` when it checks `dev->si_drv1` and safely fail rather than accessing freed memory. This defensive programming prevents use-after-free bugs during concurrent operations.

```
if (sc->ptr != NULL)
    blinkers--;
```

6.14.2.11.3 Deactivating Pattern Execution If this LED has an active blink pattern (`ptr != NULL`), the global `blinkers` counter must be decremented. This counter tracks how many LEDs need timer servicing, and removing an active LED reduces that count.

Timer shutdown logic: When the counter reaches zero (this was the last blinking LED), the timer callback will notice and stop rescheduling itself. However, there's no explicit timer stop here; the counter update is sufficient. The timer callback checks `blinkers > 0` before each reschedule.

```
LIST_REMOVE(sc, list);
if (LIST_EMPTY(&led_list))
    callout_stop(&led_ch);
```

6.14.2.11.4 Removing from Global Registry **List removal:** `LIST_REMOVE(sc, list)` unlinks this LED from the global list. The macro updates neighboring list entries to skip this node, and future timer callbacks won't see this LED when iterating.

Explicit timer stop: If the list becomes empty after removal, `callout_stop(&led_ch)` explicitly stops the timer. This is an optimization, waiting for the timer to notice `blinkers == 0` would work, but stopping immediately when all LEDs are gone is more efficient.

The `callout_stop` function is safe to call on an already-stopped timer (it does nothing), so the check for empty list is just an optimization to avoid the function call when unnecessary.

Lock release: After list modification and timer management, the mutex is released:

```
mtx_unlock(&led_mtx);
```

The remaining cleanup doesn't require mutex protection since this LED is now invisible to timer callbacks and write operations.

```
sx_xlock(&led_sx);
free_unr(led_unit, sc->unit);
destroy_dev(dev);
if (sc->spec != NULL)
    sbuf_delete(sc->spec);
free(sc->name, M_LED);
free(sc, M_LED);
sx_xunlock(&led_sx);
```

6.14.2.11.5 Resource Deallocation Under Exclusive Lock **Exclusive lock acquisition:** The `led_sx` lock serializes device creation and destruction. Acquiring it exclusively prevents new devices from being created while this one is being destroyed, avoiding races where the freed unit number or name might be immediately reused.

Unit number return: `free_unr(led_unit, sc->unit)` returns the unit number to the pool, making it available for future LED registrations. Without this, unit numbers would leak and eventually exhaust the available range.

Device node destruction: `destroy_dev(dev)` removes `/dev/led/name` from the filesystem and deallocates the `cdev` structure. This function blocks until all open file descriptors to the device are closed, ensuring no write operations are in progress.

After `destroy_dev` returns, the device no longer exists in `/dev`, and any future attempts to open it will fail with `ENOENT` (no such file or directory).

Pattern buffer cleanup: If an active pattern exists (`sc->spec != NULL`), its string buffer is freed with `sbuf_delete`. This handles the case where an LED is destroyed while a blink pattern is running.

Name string cleanup: `free(sc->name, M_LED)` releases the duplicated name string allocated during `led_create`. The `M_LED` type tag matches the allocation, maintaining accounting consistency.

Softc deallocation: `free(sc, M_LED)` releases the LED state structure itself. After this call, the `sc` pointer is invalid and must not be accessed.

Lock release: The exclusive lock is released, allowing other device operations to proceed. All resources associated with this LED have been freed.

6.14.2.11.6 Symmetric Cleanup The destruction sequence precisely reverses creation:

Creation Step	Destruction Step
Allocate softc	Free softc
Duplicate name	Free name
Allocate unit	Free unit
Create device node	Destroy device node
Insert into list	Remove from list
Increment blinkers (if pattern)	Decrement blinkers (if pattern)

This symmetry ensures complete cleanup with no resource leaks. Every allocation has a corresponding deallocation, every list insertion has a removal, every increment has a decrement.

6.14.2.11.7 Handling Active LEDs If an LED is destroyed while actively blinking, the function handles this cleanly:

Before destruction:

```
LED state: ptr = "AaAa", spec = sbuf, blinkers = 1
Timer: scheduled, will fire in 0.1s
```

During destruction:

```

Mutex locked
dev->si_drv1 = NULL (breaks write path)
blinkers-- (now 0)
LIST_REMOVE (invisible to timer)
Mutex unlocked
Timer fires, sees empty list, doesn't reschedule
sbuf_delete (frees pattern)

```

After destruction:

```

LED state: freed
Timer: stopped
Device: removed from /dev

```

The LED's pattern is interrupted mid-execution, but no crashes or leaks occur. The hardware LED is left in whatever state it was in at destruction time, turning it off explicitly is the hardware driver's responsibility if desired.

6.14.2.11.8 Thread Safety Considerations The two-phase locking (mutex then exclusive lock) prevents several race conditions:

Race 1: Write vs. Destroy

Thread A (write)	Thread B (destroy)
Begin led_write()	Begin led_destroy()
	Lock led_mtx
	dev->si_drv1 = NULL
	Remove from list
	Unlock led_mtx
Lock led_mtx	Lock led_sx
sc = dev->si_drv1 (NULL)	destroy_dev() blocks
if (sc != NULL) ... (skipped)	...
Unlock led_mtx	[write returns]
Return error	destroy_dev() completes

The write operation safely detects the destroyed LED via the NULL check and returns an error without accessing freed memory.

Race 2: Timer vs. Destroy

Timer callback running	led_destroy() called
Iterating LED list	Lock led_mtx (blocks)
Process this LED	...
	Acquire lock

```

                                Remove from list
                                Unlock
Move to next LED                [timer continues]
                                Free softc

```

The timer finishes processing the LED before it's removed from the list. The mutex ensures the LED isn't freed while the timer is accessing it.

6.14.2.11.9 Contrast with null.c The null driver's cleanup in `MOD_UNLOAD` was simple:

```

destroy_dev(full_dev);
destroy_dev(null_dev);
destroy_dev(zero_dev);

```

Three fixed devices, three destroy calls, done. The LED driver's cleanup is more complex because:

Dynamic lifecycle: LEDs are created and destroyed individually as hardware appears and disappears, not all at once during module unload.

Active state: LEDs may have running timers and allocated patterns that need cleanup.

Reference counting: The blinkers counter must be maintained correctly for timer management.

List management: Removal from the global registry requires proper list manipulation.

This additional complexity is the cost of supporting dynamic device creation, the subsystem must handle arbitrary sequences of create/destroy operations without leaking resources or corrupting state.

6.14.2.11.10 Usage Example A complete hardware driver lifecycle:

```

// During attach
sc->led_dev = led_create(my_led_control, sc, "disk0");

// During normal operation
// LED blinks, patterns execute, writes succeed

// During detach
led_destroy(sc->led_dev);
// LED is gone, /dev/led/disk0 removed
// All resources freed

```

After `led_destroy` returns, the hardware driver can safely unload without leaving orphaned LED state in the kernel.

6.14.2.12 9) Driver init: set up bookkeeping and the callout

```
331: static void
332: led_drvinit(void *unused)
333: {
335:     led_unit = new_unrhdr(0, INT_MAX, NULL);
336:     mtx_init(&led_mtx, "LED mtx", NULL, MTX_DEF);
337:     sx_init(&led_sx, "LED sx");
338:     callout_init_mtx(&led_ch, &led_mtx, 0);
339: }
341: SYSINIT(leddev, SI_SUB_DRIVERS, SI_ORDER_MIDDLE, led_drvinit, NULL);
```

6.14.2.12.1 Driver Initialization and Registration The final section of the LED driver handles one-time initialization during system boot. This code sets up the global infrastructure needed before any LEDs can be registered, establishing the foundation that all subsequent operations rely on.

```
static void
led_drvinit(void *unused)
{
    led_unit = new_unrhdr(0, INT_MAX, NULL);
    mtx_init(&led_mtx, "LED mtx", NULL, MTX_DEF);
    sx_init(&led_sx, "LED sx");
    callout_init_mtx(&led_ch, &led_mtx, 0);
}
```

6.14.2.12.2 Initialization Function **Function signature:** Initialization functions registered with SYSINIT receive a single void * argument for optional data. The LED driver doesn't need any initialization parameters, so the argument is unused and named accordingly.

Unit number allocator creation: new_unrhdr(0, INT_MAX, NULL) creates a unit number pool that can allocate integers from 0 to INT_MAX (typically 2,147,483,647). Each LED registered will receive a unique number from this range, used as the device minor number. The NULL parameter indicates no mutex protects this allocator; external locking (via led_sx) will serialize access instead.

Mutex initialization: mtx_init(&led_mtx, "LED mtx", NULL, MTX_DEF) initializes the mutex that protects:

- The LED list during insertions, removals, and traversals
- The blinkers counter
- Per-LED pattern execution state

The parameters specify:

- &led_mtx - the mutex structure to initialize

- "LED_mtx" - name appearing in lock debugging and analysis tools
- NULL - no witness data (advanced lock-order checking not needed)
- MTX_DEF - default mutex type (can sleep while held, standard recursion rules)

Shared/exclusive lock initialization: `sx_init(&led_sx, "LED_sx")` initializes the lock that protects device creation and destruction. The simpler parameter list reflects that sx locks have fewer options than mutexes; they're always sleepable and non-recursive.

Timer initialization: `callout_init_mtx(&led_ch, &led_mtx, 0)` prepares the timer callback infrastructure. The parameters specify:

- `&led_ch` - the callout structure to initialize
- `&led_mtx` - the mutex held when timer callbacks execute
- `0` - flags (none needed)

This initialization associates the timer with the mutex, so timer callbacks automatically hold `led_mtx` while executing. This simplifies locking in `led_timeout`, it doesn't need to acquire the mutex explicitly because the callout infrastructure does it automatically.

```
SYNINIT(leddev, SI_SUB_DRIVERS, SI_ORDER_MIDDLE, led_drvinit, NULL);
```

6.14.2.12.3 Boot-Time Registration The `SYNINIT` macro registers the initialization function with the kernel's boot sequence. The kernel calls registered functions in order during startup, ensuring dependencies are satisfied.

Macro parameters:

leddev: A unique identifier for this initialization. Must be unique across the entire kernel to prevent collisions. The name doesn't affect behavior, it's purely for identification in debugging.

SI_SUB_DRIVERS: The subsystem level. The kernel initialization happens in phases (we will see a simplified list, the . . . in the list below means that we have skipped some phases):

- `SI_SUB_TUNABLES` - system tunables
- `SI_SUB_COPYRIGHT` - display copyright
- `SI_SUB_VM` - virtual memory
- `SI_SUB_KMEM` - kernel memory allocator
- . . .
- `SI_SUB_DRIVERS` - device drivers
- . . .
- `SI_SUB_RUN_SCHEDULER` - start scheduler

The LED driver initializes during the driver phase, after core kernel services (memory allocation, locking primitives) are available but before devices start attaching.

SI_ORDER_MIDDLE: The order within the subsystem. Multiple initializers in the same subsystem execute in order from `SI_ORDER_FIRST` through `SI_ORDER_ANY` to `SI_ORDER_LAST`. Using `MIDDLE` places the LED driver in the middle of the driver initialization phase, not critical to go first, but not dependent on everything else either.

led_drvinit: Pointer to the initialization function.

NULL: No argument data to pass to the function.

6.14.2.12.4 Initialization Ordering The SYSINIT mechanism ensures proper initialization order:

Before LED init:

```
Memory allocator running (malloc works)
Lock primitives available (mtx_init, sx_init work)
Timer subsystem operational (callout_init works)
Device filesystem ready (make_dev will work later)
```

During LED init:

```
led_drvinit() called
->
Create unit allocator
Initialize locks
Prepare timer infrastructure
```

After LED init:

```
Hardware drivers attach
->
Call led_create()
->
Use the already-initialized infrastructure
```

Without SYSINIT, hardware drivers that call `led_create()` during their attach functions would crash attempting to use uninitialized locks or allocate from a NULL unit number pool.

6.14.2.12.5 Contrast with null.c Module Load The null driver used module event handlers:

```
static int
null_modevent(module_t mod, int type, void *data)
{
    switch(type) {
        case MOD_LOAD:
            /* Create devices */
            break;
        case MOD_UNLOAD:
            /* Destroy devices */
            break;
    }
}
```

```

    }
}

DEV_MODULE(null, null_modevent, NULL);

```

Module events fire when loadable modules are loaded or unloaded. The LED driver uses `SYSINIT` instead because:

Always needed: The LED subsystem is infrastructure that other drivers depend on. It should initialize early during boot, not wait for explicit module loading.

No unload: The LED subsystem doesn't provide a module unload handler. Once initialized, it remains available for the system's lifetime. Unloading would be complex, all registered LEDs would need to be destroyed, which requires coordinating with potentially many hardware drivers.

Separate concerns: `SYSINIT` handles initialization, while individual LEDs are created/destroyed dynamically as hardware appears/disappears. The null driver conflated initialization with device creation (both happened in `MOD_LOAD`), while the LED driver separates them.

6.14.2.12.6 What's Not Initialized Notice what this function **doesn't** do:

No LED creation: Unlike the null driver which created its three devices during initialization, the LED driver creates no devices here. Device creation is demand-driven via `led_create()` calls from hardware drivers.

No list initialization: The global `led_list` was statically initialized:

```
static LIST_HEAD(, ledsc) led_list = LIST_HEAD_INITIALIZER(led_list);
```

Static initialization suffices for list heads, they're just pointer structures that start empty.

No blinkers initialization: The blinkers counter was declared `static int`, giving it an initial value of 0 automatically. No explicit initialization needed.

No timer scheduling: The timer callback starts inactive. It's only scheduled when the first LED receives a blink pattern, not during driver initialization.

This minimal initialization reflects good design: do the minimum necessary work at boot, defer everything else until actually needed.

6.14.2.12.7 Complete Boot Sequence The full sequence from power-on to working LEDs:

1. Kernel starts
2. Early boot (memory, interrupts, etc.)
3. `SYSINIT` runs:
 - `led_drvinit()` initializes LED infrastructure
4. Device enumeration and driver attachment:
 - Disk driver attaches
 - Calls `led_create(..., "disk0")`

- /dev/led/disk0 appears
 - GPIO driver attaches
 - Calls `led_create(..., "power")`
 - /dev/led/power appears
5. System running:
- User scripts write patterns
 - Drivers call `led_set()`
 - LEDs blink and indicate status

The LED subsystem is ready before hardware drivers need it, and hardware drivers can register LEDs at any point during or after boot without worrying about initialization order.

6.14.2.12.8 Why This Matters This initialization pattern, early infrastructure setup via `SYSINIT`, late device creation on demand, is fundamental to FreeBSD's modular architecture. It allows:

Flexibility: Hardware drivers don't need to coordinate initialization order. The LED subsystem is always ready when they need it.

Scalability: The subsystem doesn't pre-allocate resources for devices that might not exist. Memory usage scales with actual hardware.

Modularity: Hardware drivers depend only on the LED API, not on implementation details. The subsystem can change internally without affecting drivers.

Reliability: Initialization failures (like memory exhaustion during `new_unrhdr`) are fatal panics rather than obscure later crashes, making problems immediately visible during boot.

This design philosophy, initialize infrastructure early, create instances lazily, appears throughout the FreeBSD kernel and is worth understanding for anyone implementing subsystems or drivers.

6.14.2.13 Interactive Exercises for `led(4)`

Goal: Understand dynamic device creation, timer-based state machines, and pattern parsing. This driver builds on concepts from the null driver but adds stateful pattern execution and kernel API design.

6.14.2.13.1 A) Structure and Global State

1. Examine the `struct ledsc` definition near the top of `led.c`. This structure contains both device identity and pattern execution state. Create a table categorizing the fields:

Field	Purpose	Category
<code>list</code>	?	Linkage
<code>name</code>	?	Identity
<code>ptr</code>	?	Pattern execution
...	...	...

Quote the fields related to pattern execution (``str``, ``ptr``, ``count``, ``last_second``) and explain the r

2. Locate the file-scope statics that follow struct `ledsc` (`led_unit`, `led_mtx`, `led_sx`, `led_list`, `led_ch`, `blinkers`, and the `M_LED_MALLOC_DEFINE`). For each one, explain its purpose:

- `led_unit` - what does this allocate?
- `led_mtx` vs. `led_sx` - why two locks? What does each protect?
- `led_list` - who iterates this and when?
- `led_ch` - what triggers this?
- `blinkers` - what happens when this reaches 0?

Quote the declaration lines.

3. Examine the `led_cdevsw` structure. Which operation is defined? Which operations are notably absent (compare to `null.c`)? What appears under `/dev` when LEDs are created?

6.14.2.13.2 B) Write-to-Blink Path

1. Trace the data flow in `led_write()`:
 - Find the size check - what's the limit and why?
 - Find the buffer allocation - why `uio_resid + 1`?
 - Find the `uiomove()` call - what's being copied?
 - Find the parse call - what does it produce?
 - Find the state update - what lock is held?

Quote each step and write one sentence explaining its purpose.

2. In `led_state()`, trace two paths:

Path 1 - Installing a pattern (`sb != NULL`):

- Which fields change in the softc?
- When is `blinkers` incremented?
- What does `sc->ptr = sc->str` accomplish?

Path 2 - Setting static state (`sb == NULL`):

- Which fields change?
- When is `blinkers` decremented?
- Why call `sc->func()` here but not in Path 1?

Quote the key lines for each path.

3. Explain the timer-to-pattern connection:
 - When `blinkers` goes from 0 -> 1, what must happen? (Hint: who schedules the timer?)
 - When `blinkers` goes from 1 -> 0, what must happen? (Hint: look for `LIST_EMPTY(&led_list)` and the adjacent `callout_stop(&led_ch)` call in `led_destroy`.)
 - Why doesn't `led_state()` directly schedule the timer?

6.14.2.13.3 C) Timer Callback State Machine

1. In `led_timeout()`, explain the pattern interpreter:

Create a table showing what each code does:

Code	LED Action	Duration Setup	Example
'A'-'J'	?	count = ?	'C' means?
'a'-'j'	?	count = ?	'c' means?
'U'/'u'	?	Special timing	What's checked?
'.'	?	N/A	What happens?

Quote the lines implementing each case.

2. The count field implements delays:

- When is count set to non-zero? Quote the line.
- When is count decremented? Quote the line.
- Why does the pattern advance skip when `count > 0`?

Trace pattern "Ac" (on 0.1s, off 0.3s) through three timer ticks:

- Tick 1: What happens?
- Tick 2: What happens?
- Tick 3: What happens?

3. Find the timer rescheduling logic at the tail of `led_timeout` (the `if (blinkers > 0)` guard followed by `callout_reset(&led_ch, hz / 10, led_timeout, p)`):

- What condition must be true for rescheduling?
- What's the delay (`hz / 10` means what in seconds)?
- Why doesn't the timer reschedule when `blinkers == 0`?

6.14.2.13.4 D) Pattern Parsing DSL

1. For the flash command "f2" (the case 'f': arm inside `led_parse`):

- What does the digit '2' map to (`i = ?`)?
- What two-character string is generated?
- How long is each phase in timer ticks?
- What frequency does this produce?

Quote the lines and calculate the blink rate.

2. For the Morse command "m...—..." (the case 'm': arm inside `led_parse`):

- What string is generated for '.' (dot)?
- What string is generated for '-' (dash)?

- What string is generated for ' ' (space)?
- What string is generated for '\n' (newline)?

Quote the `sbuf_cat()` calls and explain how this implements standard Morse timing (dot = 1 unit, dash = 3 units).

3. For the digit command “d12” (the case 'd': arm inside `led_parse`):

- How is digit '1' represented in flashes?
- How is digit '2' represented in flashes?
- Why is '0' treated as 10 instead of 0?
- What separates repetitions of the pattern?

Quote the loop and explain the formula for flash count.

6.14.2.13.5 E) Dynamic Device Lifecycle

1. In `led_create_state()`, identify the initialization sequence:

- What's allocated first and with what flags?
- Which lock is acquired for device creation? Why exclusive?
- What parameters does `make_dev()` receive? What path is created?
- Which lock protects list insertion? Why different from device creation?
- When is the hardware callback invoked, and what does `state != -1` mean?

Quote each phase and explain the lock separation.

2. In `led_destroy()`, trace the cleanup:

- Why is `dev->si_drv1` set to NULL immediately?
- When is `blinkers` decremented?
- Why call `callout_stop()` only when list becomes empty?
- Which resources are freed under which lock?

Create a table mapping each `led_create()` allocation to its corresponding `led_destroy()` deallocation.

3. Explain the two-phase locking:

- Why acquire `led_mtx` first, then release it before acquiring `led_sx`?
- What would happen if we held `led_mtx` during `destroy_dev()`?
- Could we use just one lock for everything? What would be the downsides?

6.14.2.13.6 F) Kernel API vs. Device Write

1. Compare `led_write()` and `led_set()`:

- Both call `led_parse()` and `led_state()` - what's different about how they find the LED?
- `led_write()` has size limits - does `led_set()` need them? Why or why not?
- Who typically calls each function? Give examples.

Quote the LED lookup logic in both functions.

2. Find the `led_cdevsw` declaration and explain why it's shared:

- How many `cdevsw` structures exist for N LEDs?
- How does `led_write()` know which LED it's writing to?
- Compare this to `null.c` which had three separate `cdevsw` structures.

6.14.2.13.7 G) System Integration

1. Examine the initialization (`led_drvinit` and its `SYSINIT` registration):

- What does `SYSINIT` do and when does it run?
- What are the four resources initialized in `led_drvinit()`?
- Why is the callout associated with `led_mtx`?
- What's NOT initialized here (compare to `null.c`'s `MOD_LOAD`)?

2. Find where the driver registers with `SYSINIT`:

- What's the subsystem level (`SI_SUB_DRIVERS`)?
- Why not use `DEV_MODULE` like `null.c` did?
- Can this driver be unloaded? Why or why not?

6.14.2.13.8 H) Safe Experiments (optional, only if you have a system with physical LEDs)

1. If your system has LEDs in `/dev/led`, try these (as root in a VM):

```
# List available LEDs
ls -l /dev/led/

# Fast blink
echo "f" > /dev/led/SOME_LED_NAME

# Slow blink
echo "f5" > /dev/led/SOME_LED_NAME

# Morse code SOS
echo "m...-..." > /dev/led/SOME_LED_NAME

# Static on
echo "1" > /dev/led/SOME_LED_NAME

# Static off
echo "0" > /dev/led/SOME_LED_NAME
```

For each test:

- Which parse case handles the command?
- What internal pattern string is generated?

- Estimate the timing you observe and verify against the code.
2. Try invalid commands and explain the errors:

```
# Too long
perl -e 'print "f" x 600' > /dev/led/SOME_LED_NAME
# What error? Which line checks this?

# Invalid syntax
echo "xyz" > /dev/led/SOME_LED_NAME
# What error? Which case handles this?
```

6.14.2.14 Stretch (thought experiments)

1. The timer self-rescheduling logic (the `if (blinkers > 0)` guard plus `callout_reset(&led_ch, hz / 10, led_timeout, p)` at the tail of `led_timeout`):

Suppose we removed the `if (blinkers > 0)` check and always called:

```
callout_reset(&led_ch, hz / 10, led_timeout, p);
```

Trace what happens when:

- User writes “f” to an LED (timer starts)
- Pattern runs for 5 seconds
- User writes “0” to stop blinking (`blinkers -> 0`)

What’s the symptom? Where’s the wasted resource? Why does the current check prevent this?

2. The write size limit (the `if (uio->uio_resid > 512) return (EINVAL);` check in `led_write`):

The code rejects writes over 512 bytes. Consider removing this check:

- What’s the immediate risk with `malloc(uio->uio_resid, ...)`?
- The parser then allocates an sbuf - what’s the risk there?
- Could an attacker cause a denial of service? How?
- Why is 512 bytes plenty for any legitimate LED pattern?

Point to the current guard and explain the defense-in-depth principle.

3. The two-lock design:

Suppose we replaced both `led_mtx` and `led_sx` with a single mutex. What would break?

Scenario 1: `led_create()` calls `make_dev()` while holding the lock, and `make_dev()` blocks. What happens to timer callbacks during this time?

Scenario 2: A write operation holds the lock while parsing a complex pattern. What happens to other LEDs’ timer updates?

Explain why separating device structure operations (`led_sx`) from state operations (`led_mtx`) improves concurrency.

Note: If your system doesn't have physical LEDs, you can still trace through the code and understand the patterns. The mental model of “timer walks list -> interprets codes -> calls callbacks” is the key lesson, not seeing actual lights blink.

6.14.2.15 Bridge to the next tour

If you can walk the path from **user `write()`** to a **timer-driven state machine** and back to **device tear-down**, you've internalized the write-centric character-device shape with timers and sbuf-powered parsing. Next we'll look at a slightly different shape: a **network interface pseudo-device** that binds into the **ifnet** stack (`if_tuntap.c`). Keep your eyes on three things: how the driver **registers** with a larger subsystem, how **I/O is routed** through that subsystem's callbacks, and how **open/close/lifecycle** differs from the small `/dev` patterns you've just mastered.

Checkpoint. You have now walked through the full shape of a simple driver: the Newbus lifecycle, `cdevsw` entry points, `make_dev()` and `devctl`, module packaging with `bsd.kmod.mk`, and two real character drivers, the null/zero/full trio and `led(4)`. The rest of the chapter turns to drivers that plug into larger subsystems: the `tun(4)/tap(4)` pseudo-NIC that binds into the ifnet stack, the PCI-backed `uart(4)` glue driver, the synthesis that pulls four tours into one mental model, and the blueprints and labs that turn reading into practice. If you want to close the book and come back, this is a natural pause.

6.14.3 Tour 3 - A pseudo-NIC that is also a character device: `tun(4)/tap(4)`:

Open the file:

```
% cd /usr/src/sys/net
% less if_tuntap.c
```

This driver is a perfect “small but real” example of integrating a simple character device with a larger kernel **subsystem** (the network stack). It exposes `/dev/tunN`, `/dev/tapN`, and `/dev/vmnetN` character devices, while also registering **ifnet** interfaces that you can `ifconfig`.

As you read, keep these “anchors” in mind:

- **Character device surface:** `cdevsw` + `open/read/write/ioctl/poll/kqueue`;
- **Network surface:** `ifnet` + `if_attach` + `bpfattach`;
- **Cloning:** on-demand creation of `/dev/tunN` and the corresponding `ifnet`;
- how a `cdevsw` maps `open/read/write/ioctl` into driver code for three related device names;
- how opening `/dev/tun0` et al. lines up with creating/configuring an `ifnet`;
- how data **flows** both ways: packets from kernel -> user via `read(2)`, and user -> kernel via `write(2)`.

Note

To keep this manageable, code examples below are excerpts from the 2071-line source file. Lines marked with ... have been omitted.

6.14.3.1 1) Where the character device surface is declared (the cdevsw)

```
270: static struct tuntap_driver {
271:     struct cdevsw      cdevsw;
272:     int                ident_flags;
273:     struct unrhdr      *unrhdr;
274:     struct clonedevs    *clones;
275:     ifc_match_f        *clone_match_fn;
276:     ifc_create_f        *clone_create_fn;
277:     ifc_destroy_f       *clone_destroy_fn;
278: } tuntap_drivers[] = {
279:     {
280:         .ident_flags = 0,
281:         .cdevsw = {
282:             .d_version = D_VERSION,
283:             .d_flags = D_NEEDMINOR,
284:             .d_open = tunopen,
285:             .d_read = tunread,
286:             .d_write = tunwrite,
287:             .d_ioctl = tunioctl,
288:             .d_poll = tunpoll,
289:             .d_kqfilter = tunkqfilter,
290:             .d_name = tunname,
291:         },
292:         .clone_match_fn = tun_clone_match,
293:         .clone_create_fn = tun_clone_create,
294:         .clone_destroy_fn = tun_clone_destroy,
295:     },
296:     {
297:         .ident_flags = TUN_L2,
298:         .cdevsw = {
299:             .d_version = D_VERSION,
300:             .d_flags = D_NEEDMINOR,
301:             .d_open = tunopen,
302:             .d_read = tunread,
303:             .d_write = tunwrite,
304:             .d_ioctl = tunioctl,
305:             .d_poll = tunpoll,
306:             .d_kqfilter = tunkqfilter,
```

```

307:         .d_name =      tapname,
308:     },
309:     .clone_match_fn =   tap_clone_match,
310:     .clone_create_fn =  tun_clone_create,
311:     .clone_destroy_fn = tun_clone_destroy,
312: },
313: {
314:     .ident_flags = TUN_L2 | TUN_VMNET,
315:     .cdevsw = {
316:         .d_version =    D_VERSION,
317:         .d_flags =      D_NEEDMINOR,
318:         .d_open =       tunopen,
319:         .d_read =       tunread,
320:         .d_write =      tunwrite,
321:         .d_ioctl =      tunioctl,
322:         .d_poll =       tunpoll,
323:         .d_kqfilter =   tunkqfilter,
324:         .d_name =       vmnetname,
325:     },
326:     .clone_match_fn =   vmnet_clone_match,
327:     .clone_create_fn =  tun_clone_create,
328:     .clone_destroy_fn = tun_clone_destroy,
329: },
330: };

```

This initial fragment demonstrates a clever design pattern: **one driver implementation serving three related but distinct device types** (tun, tap, and vmnet).

Let's see how it works:

```

struct tuntap_driver {
    struct cdevsw      cdevsw;           // Character device switch table
    int                ident_flags;      // Identity flags (TUN_L2, TUN_VMNET)
    struct unrhdr      *unrhdr;          // Unit number allocator
    struct clonedevs    *clones;         // Cloning infrastructure
    ifc_match_f        *clone_match_fn;  // Network interface clone matching
    ifc_create_f        *clone_create_fn; // Network interface creation
    ifc_destroy_f       *clone_destroy_fn; // Network interface destruction
};

```

6.14.3.1.1 The tuntap_driver Structure This structure combines **two kernel subsystems**:

1. **Character device operations** (cdevsw) - how userspace interacts with /dev/tunN, /dev/tapN, /dev/vmnetN

2. Network interface cloning (clone_*_fn) - how the corresponding ifnet structures get created

6.14.3.1.2 The Critical cdevsw Structure The cdevsw (character device switch) is FreeBSD's **function dispatch table** for character devices. Think of it as a vtable or interface:

```
.d_version    = D_VERSION      // ABI version check
.d_flags      = D_NEEDMINOR    // Device needs minor number tracking
.d_open       = tunopen        // Called on open(2)
.d_read       = tunread        // Called on read(2)
.d_write      = tunwrite       // Called on write(2)
.d_ioctl      = tunioctl       // Called on ioctl(2)
.d_poll       = tunpoll        // Called on poll(2)/select(2)
.d_kqfilter   = tunkqfilter    // Called for kqueue event registration
.d_name       = tunname        // Device name ("tun", "tap", "vmnet")
```

Key insight: All three device types share the **same function implementations** (tunopen, tunread, etc.), but behave differently based on ident_flags.

6.14.3.1.3 The Three Driver Instances

```
.ident_flags = 0                // No flags = plain TUN device
.d_name      = tunname          // "tun" -> /dev/tun0, /dev/tun1, ...
```

6.14.3.1.4 1. TUN - Layer 3 (IP) tunnel

- Point-to-point IP tunnel
- Packets are raw IP (no Ethernet headers)
- Used by VPNs like OpenVPN in TUN mode

```
.ident_flags = TUN_L2          // Layer 2 flag
.d_name      = tapname         // "tap" -> /dev/tap0, /dev/tap1, ...
```

6.14.3.1.5 2. TAP - Layer 2 (Ethernet) tunnel

- Ethernet-level tunnel
- Packets include full Ethernet frames
- Used by VMs, bridges, OpenVPN in TAP mode

```
.ident_flags = TUN_L2 | TUN_VMNET // Layer 2 + VMware semantics
.d_name = vmnetname                // "vmnet" -> /dev/vmnet0, ...
```

6.14.3.1.6 3. VMNET - VMware compatibility

- Like TAP but with VMware-specific behavior
- Different lifecycle rules (survives interface down)

6.14.3.1.7 How This Achieves Code Reuse Notice that **all three entries use identical function pointers**:

- tunopen handles opening all three device types
- tunread/tunwrite handle I/O for all three
- The functions check `tp->tun_flags` (derived from `ident_flags`) to determine behavior

For example, in `tunopen`, you'll see:

```
if ((tp->tun_flags & TUN_L2) != 0) {
    // TAP/VMNET-specific setup
} else {
    // TUN-specific setup
}
```

6.14.3.1.8 The Cloning Functions Each driver has **different clone match functions** but shares create/destroy:

- `tun_clone_match` - matches "tun" or "tunN"
- `tap_clone_match` - matches "tap" or "tapN"
- `vmnet_clone_match` - matches "vmnet" or "vmnetN"
- All use `tun_clone_create` - shared creation logic
- All use `tun_clone_destroy` - shared destruction logic

This lets the kernel automatically create `/dev/tun0` when someone opens it, even if it doesn't exist yet.

6.14.3.2 2) From clone request -> cdev creation -> ifnet attach

6.14.3.3 2.1 Clone creation (`tun_clone_create`): pick name/unit, ensure cdev, then hand off to `tuncreate`

```
520: tun_clone_create(struct if_clone *ifc, char *name, size_t len,
521:     struct ifc_data *ifd, struct ifnet **ifpp)
522: {
523:     struct tuntap_driver *drv;
524:     struct cdev *dev;
```

```
525: int err, i, tunflags, unit;
526:
527: tunflags = 0;
528: /* The name here tells us exactly what we're creating */
529: err = tuntap_name2info(name, &unit, &tunflags);
530: if (err != 0)
531:     return (err);
532:
533: drv = tuntap_driver_from_flags(tunflags);
534: if (drv == NULL)
535:     return (ENXIO);
536:
537: if (unit != -1) {
538:     /* If this unit number is still available that's okay. */
539:     if (alloc_unr_specific(drv->unrhdr, unit) == -1)
540:         return (EEXIST);
541: } else {
542:     unit = alloc_unr(drv->unrhdr);
543: }
544:
545: snprintf(name, IFNAMSIZ, "%s%d", drv->cdevsw.d_name, unit);
546:
547: /* find any existing device, or allocate new unit number */
548: dev = NULL;
549: i = clone_create(&drv->clones, &drv->cdevsw, &unit, &dev, 0);
550: /* No preexisting struct cdev *, create one */
551: if (i != 0)
552:     i = tun_create_device(drv, unit, NULL, &dev, name);
553: if (i == 0) {
554:     dev_ref(dev);
555:     tuncreate(dev);
556:     struct tuntap_softc *tp = dev->si_drv1;
557:     *ifpp = tp->tun_ifp;
558: }
559: return (i);
560: }
```

The `tun_clone_create` function serves as the bridge between FreeBSD's network interface cloning subsystem and character device creation. This function is invoked when a user executes commands like `ifconfig tun0 create` or `ifconfig tap1 create`, and its responsibility is to create both a character device (`/dev/tun0`) and its corresponding network interface.


```
static int
tun_clone_create(struct if_clone *ifc, char *name, size_t len,
                struct ifc_data *ifd, struct ifnet **ifpp)
```

6.14.3.3.1 Function Signature and Purpose The function receives an interface name (like “tun0” or “tap3”) and must return a pointer to a newly created ifnet structure through the ifpp parameter. Success returns 0; errors return the appropriate errno values, such as EEXIST or ENXIO.

6.14.3.3.2 Parsing the Interface Name The first step extracts meaning from the interface name:

```
tunflags = 0;
err = tuntap_name2info(name, &unit, &tunflags);
if (err != 0)
    return (err);
```

The tuntap_name2info helper function parses strings like “tap3” or “vmnet1” to extract:

- The **unit number** (3, 1, etc.)
- The **type flags** that determine device behavior (0 for tun, TUN_L2 for tap, TUN_L2|TUN_VMNET for vmnet)

If the name contains no unit number (e.g., just “tun”), the function returns -1 for the unit, signaling that any available unit should be allocated.

```
drv = tuntap_driver_from_flags(tunflags);
if (drv == NULL)
    return (ENXIO);
```

6.14.3.3.3 Locating the Appropriate Driver The extracted flags determine which entry from the tuntap_drivers[] array will handle this device. This lookup returns the tuntap_driver structure containing the correct cdevsw and device name (“tun”, “tap”, or “vmnet”).

6.14.3.3.4 Unit Number Allocation The driver maintains a unit number allocator (unrhdr) to prevent conflicts:

```
if (unit != -1) {
    /* User requested specific unit number */
    if (alloc_unr_specific(drv->unrhdr, unit) == -1)
        return (EEXIST);
} else {
    /* Allocate any available unit */
```

```
    unit = alloc_unr(drv->unrhdr);  
}
```

The `unrhdr` (unit number handler) ensures thread-safe allocation of device minor numbers. When a user requests a specific unit (e.g., “`tun3`”), `alloc_unr_specific` either reserves that number or returns failure if already allocated. When no specific unit is requested, `alloc_unr` selects the next available number.

This mechanism prevents race conditions where multiple processes simultaneously attempt to create the same device unit, as the allocation is serialized by the global `tunmtx` mutex.

6.14.3.3.5 Name Normalization After unit allocation, the function normalizes the interface name:

```
snprintf(name, IFNAMSIZ, "%s%d", drv->cdevsw.d_name, unit);
```

If the user specified `ifconfig tun create` without a unit number, this formats the name with the newly allocated unit, producing strings like “`tun0`” or “`tun1`”. The name parameter serves as both input and output, the caller’s buffer receives the finalized name.

```
dev = NULL;  
i = clone_create(&drv->clones, &drv->cdevsw, &unit, &dev, 0);  
if (i != 0)  
    i = tun_create_device(drv, unit, NULL, &dev, name);
```

6.14.3.3.6 Character Device Creation This section handles an important subtlety: the character device may already exist. The `clone_create` call searches for an existing `/dev/tun0` device node, which might have been created earlier through `devfs` cloning when a process opened the device path.

When `clone_create` returns non-zero (device not found), the code calls `tun_create_device` to construct a new `struct cdev`. This dual-path approach accommodates two creation scenarios:

1. A process opens `/dev/tun0` before any network configuration, triggering `devfs` cloning
2. A user runs `ifconfig tun0 create`, explicitly requesting interface creation

6.14.3.3.7 Network Interface Instantiation The final step connects the character device to the network subsystem:

```
if (i == 0) {  
    dev_ref(dev);  
    tuncreate(dev);  
    struct tuntap_softc *tp = dev->si_drv1;  
    *ifpp = tp->tun_ifp;  
}
```

After successful device creation or lookup:

- `dev_ref(dev)` increments the device's reference count, preventing premature destruction during initialization
- `tuncreate(dev)` allocates and initializes the `ifnet` structure, registering it with the network stack
- `dev->si_drv1` provides the critical linkage, this field points to the `tuntap_softc` structure, which contains both character device state and the `ifnet` pointer
- `*ifpp = tp->tun_ifp` returns the newly created network interface to the `if_clone` subsystem

6.14.3.3.8 Coordination Architecture The `tun_clone_create` function exemplifies a coordination pattern common in kernel drivers. It performs no heavy lifting itself, instead orchestrating several subsystems:

1. Name parsing determines device type and unit
2. Driver lookup selects the appropriate `cdevsw` dispatch table
3. Unit allocation ensures uniqueness
4. Device lookup or creation establishes the character device presence
5. Interface creation registers with the network stack

This separation allows two independent creation paths, character device access and network configuration, to converge correctly regardless of invocation order.

The `si_drv1` field serves as the architectural keystone, linking the character device world (`struct cdev`, file operations, `/dev` namespace) with the network world (`struct ifnet`, packet processing, `ifconfig` visibility). Every subsequent operation, whether a `read(2)` system call or packet transmission, will traverse this link to access the shared `tuntap_softc` state.

6.14.3.4 2.2 Create the `cdev` and wire `si_drv1` (`tun_create_device`)

```

807: static int
808: tun_create_device(struct tuntap_driver *drv, int unit, struct ucred *cr,
809:     struct cdev **dev, const char *name)
810: {
811:     struct make_dev_args args;
812:     struct tuntap_softc *tp;
813:     int error;
814:
815:     tp = malloc(sizeof(*tp), M_TUN, M_WAITOK | M_ZERO);
816:     mtx_init(&tp->tun_mtx, "tun_mtx", NULL, MTX_DEF);
817:     cv_init(&tp->tun_cv, "tun_condvar");
818:     tp->tun_flags = drv->ident_flags;
819:     tp->tun_drv = drv;
820:
821:     make_dev_args_init(&args);
822:     if (cr != NULL)
823:         args.mda_flags = MAKEDEV_REF | MAKEDEV_CHECKNAME;

```

```

824:  args.mda_devsw = &drv->cdevsw;
825:  args.mda_cr = cr;
826:  args.mda_uid = UID_UUCP;
827:  args.mda_gid = GID_DIALER;
828:  args.mda_mode = 0600;
829:  args.mda_unit = unit;
830:  args.mda_si_drv1 = tp;
831:  error = make_dev_s(&args, dev, "%s", name);
832:  if (error != 0) {
833:      free(tp, M_TUN);
834:      return (error);
835:  }
836:
837:  KASSERT((*dev)->si_drv1 != NULL,
838:      ("Failed to set si_drv1 at %s creation", name));
839:  tp->tun_dev = *dev;
840:  knlist_init_mtx(&tp->tun_rsel.si_note, &tp->tun_mtx);
841:  mtx_lock(&tunmtx);
842:  TAILQ_INSERT_TAIL(&tunhead, tp, tun_list);
843:  mtx_unlock(&tunmtx);
844:  return (0);
845: }

```

The `tun_create_device` function constructs the character device node and its associated driver state. This is the point where `/dev/tun0`, `/dev/tap0`, or `/dev/vmnet0` actually come into existence in the device filesystem.

```

static int
tun_create_device(struct tuntap_driver *drv, int unit, struct ucred *cr,
    struct cdev **dev, const char *name)

```

6.14.3.4.1 Function Parameters The function accepts:

- `drv` - pointer to the appropriate entry in `tuntap_drivers[]`
- `unit` - the allocated device unit number (0, 1, 2, etc.)
- `cr` - credential context (NULL for kernel-initiated creation, non-NULL for user-initiated)
- `dev` - output parameter receiving the created `struct cdev` pointer
- `name` - the complete device name string (“tun0”, “tap3”, etc.)

```

tp = malloc(sizeof(*tp), M_TUN, M_WAITOK | M_ZERO);
mtx_init(&tp->tun_mtx, "tun_mtx", NULL, MTX_DEF);

```

```
cv_init(&tp->tun_cv, "tun_condvar");
tp->tun_flags = drv->ident_flags;
tp->tun_drv = drv;
```

6.14.3.4.2 Allocating the Softc Structure Every tun/tap/vmnet device instance requires a `tuntap_softc` structure to maintain its state. This structure contains everything needed to operate the device: flags, the associated network interface pointer, I/O synchronization primitives, and references back to the driver.

The allocation uses `M_WAITOK`, allowing the function to sleep if memory is temporarily unavailable. The `M_ZERO` flag ensures all fields initialize to zero, providing safe defaults for pointers and counters.

Two synchronization primitives are initialized:

- `tun_mtx` - a mutex protecting the `softc`'s mutable fields
- `tun_cv` - a condition variable used during device destruction to wait for all operations to complete

The `tun_flags` field receives the driver's identity flags (0, `TUN_L2`, or `TUN_L2|TUN_VMNET`), establishing whether this instance behaves as a tun, tap, or vmnet device. The `tun_drv` backpointer allows the `softc` to access its parent driver's resources like the unit number allocator.

6.14.3.4.3 Preparing Device Creation Arguments FreeBSD's modern device creation API uses a structure to pass parameters rather than a long argument list:

```
make_dev_args_init(&args);
if (cr != NULL)
    args.mda_flags = MAKEDEV_REF | MAKEDEV_CHECKNAME;
args.mda_devsw = &drv->cdevsw;
args.mda_cr = cr;
args.mda_uid = UID_UUCP;
args.mda_gid = GID_DIALER;
args.mda_mode = 0600;
args.mda_unit = unit;
args.mda_si_drv1 = tp;
```

The `make_dev_args` structure configures every aspect of the device node:

Flags: When `cr` is non-NULL (user-initiated creation), two flags are set:

- `MAKEDEV_REF` - automatically add a reference to prevent immediate destruction
- `MAKEDEV_CHECKNAME` - validate the name doesn't conflict with existing devices

Dispatch table: `mda_devsw` points to the `cdevsw` containing function pointers for `open`, `read`, `write`, `ioctl`, etc. This is how the kernel knows which functions to call when userspace performs operations on this device.

Credentials: `mda_cr` associates the creating user's credentials with the device, used for permission checks.

Ownership and permissions: The device node will be owned by the `uucp` user and `dialer` group with mode `0600` (read/write for owner only). These historical Unix conventions reflect the original use of serial devices for

dial-up networking. In practice, administrators often adjust these permissions via `devfs.rules` or by having privileged daemons open the devices.

Unit number: `mda_unit` embeds the unit number into the device's minor number, allowing the kernel to distinguish `/dev/tun0` from `/dev/tun1`.

Private data: `mda_si_drv1` matters here: this field will become the `si_drv1` member of the created `struct cdev`, establishing the link from character device to driver state. Every subsequent operation on the device will retrieve the `softc` via this field.

```
error = make_dev_s(&args, dev, "%s", name);
if (error != 0) {
    free(tp, M_TUN);
    return (error);
}
```

6.14.3.4.4 Creating the Device Node The `make_dev_s` call creates the `struct cdev` and registers it with `devfs`. If successful, `*dev` receives a pointer to the new device structure. The `"%s"` format string and `name` argument specify the device node path within `/dev`.

Common failure modes include:

- Name conflicts (a device with that name already exists)
- Resource exhaustion (out of kernel memory)
- `Devfs` subsystem errors

On failure, the function immediately deallocates the `softc` and returns the error to the caller. This prevents resource leaks.

```
KASSERT((*dev)->si_drv1 != NULL,
    ("Failed to set si_drv1 at %s creation", name));
tp->tun_dev = *dev;
knlist_init_mtx(&tp->tun_rsel.si_note, &tp->tun_mtx);
```

6.14.3.4.5 Finalizing Device State The `KASSERT` is a development-time sanity check verifying that `make_dev_s` correctly populated `si_drv1` from `mda_si_drv1`. This assertion would fire during kernel development if the device creation logic broke, but it compiles away in release builds.

The `tp->tun_dev` assignment creates the reverse link: while `si_drv1` points from `cdev` to `softc`, `tun_dev` points from `softc` to `cdev`. This bidirectional linkage allows code to traverse in either direction.

The `knlist_init_mtx` call initializes the `kqueue` notification list protected by the `softc`'s mutex. This infrastructure supports `kqueue(2)` event monitoring, allowing userspace applications to efficiently wait for readable/writable conditions on the device.

```
mtx_lock(&tunmtx);
TAILQ_INSERT_TAIL(&tunhead, tp, tun_list);
mtx_unlock(&tunmtx);
return (0);
```

6.14.3.4.6 Global Registration Finally, the new device registers itself in the global tunhead list. This list allows the driver to enumerate all active tun/tap/vmnet instances, which is necessary during module unload or system-wide operations.

The tunmtx mutex protects the list from concurrent modification. Multiple threads might simultaneously create devices, so this lock ensures list consistency.

6.14.3.4.7 The Created Device State At function completion, several kernel objects exist and are properly linked:

```
/dev/tun0 (struct cdev)
    -> si_drv1
tuntap_softc
    -> tun_dev
/dev/tun0 (struct cdev)
    -> tun_drv
tuntap_drivers[0]
```

The_softc is registered in the global device list, ready for both character device operations and network interface attachment. However, the network interface (ifnet) does not yet exist, that will be created by the tuncreate function.

This separation of concerns, character device creation versus network interface creation, allows the two subsystems to initialize independently and in flexible order.

6.14.3.5 2.3 Build & attach the ifnet (tuncreate): L2 (tap) vs L3 (tun)

```
950: static void
951: tuncreate(struct cdev *dev)
952: {
953:     struct tuntap_driver *drv;
954:     struct tuntap_softc *tp;
955:     struct ifnet *ifp;
956:     struct ether_addr eaddr;
957:     int iflags;
958:     u_char type;
959:
```

```
960:  tp = dev->si_drv1;
961:  KASSERT(tp != NULL,
962:    ("si_drv1 should have been initialized at creation"));
963:
964:  drv = tp->tun_drv;
965:  iflags = IFF_MULTICAST;
966:  if ((tp->tun_flags & TUN_L2) != 0) {
967:    type = IFT_ETHER;
968:    iflags |= IFF_BROADCAST | IFF_SIMPLEX;
969:  } else {
970:    type = IFT_PPP;
971:    iflags |= IFF_POINTOPOINT;
972:  }
973:  ifp = tp->tun_ifp = if_alloc(type);
974:  ifp->if_softc = tp;
975:  if_initname(ifp, drv->cdevsw.d_name, dev2unit(dev));
976:  ifp->if_ioctl = tunifioctl;
977:  ifp->if_flags = iflags;
978:  IFQ_SET_MAXLEN(&ifp->if_snd, ifqmaxlen);
979:  ifp->if_capabilities |= IFCAP_LINKSTATE | IFCAP_MEXTPG;
980:  if ((tp->tun_flags & TUN_L2) != 0)
981:    ifp->if_capabilities |=
982:      IFCAP_RXCSUM | IFCAP_RXCSUM_IPV6 | IFCAP_LRO;
983:  ifp->if_capenable |= IFCAP_LINKSTATE | IFCAP_MEXTPG;
984:
985:  if ((tp->tun_flags & TUN_L2) != 0) {
986:    ifp->if_init = tunifinit;
987:    ifp->if_start = tunstart_l2;
988:    ifp->if_transmit = tap_transmit;
989:    ifp->if_qflush = if_qflush;
990:
991:    ether_gen_addr(ifp, &eaddr);
992:    ether_ifattach(ifp, eaddr.octet);
993:  } else {
994:    ifp->if_mtu = TUNMTU;
995:    ifp->if_start = tunstart;
996:    ifp->if_output = tunoutput;
997:
998:    ifp->if_snd.ifq_drv_maxlen = 0;
999:    IFQ_SET_READY(&ifp->if_snd);
1000:
1001:    if_attach(ifp);
1002:    bpfattach(ifp, DLT_NULL, sizeof(u_int32_t));
1003:  }
```



```

1004:
1005:     TUN_LOCK(tp);
1006:     tp->tun_flags |= TUN_INITED;
1007:     TUN_UNLOCK(tp);
1008:
1009:     TUNDEBUG(ifp, "interface %s is created, minor = %#x\n",
1010:             ifp->if_xname, dev2unit(dev));
1011: }

```

The `tuncreate` function constructs and registers the network interface (`ifnet`) corresponding to a character device. After this function completes, the device appears in `ifconfig` output and can participate in network operations. This is where the character device world and the network stack converge.

```

tp = dev->si_drv1;
KASSERT(tp != NULL,
        ("si_drv1 should have been initialized at creation"));

drv = tp->tun_drv;

```

6.14.3.5.1 Retrieving Driver Context The function begins by traversing the link from struct `cdev` to `tuntap_softc` established during device creation. The assertion verifies this fundamental invariant, every device must have an associated `softc`. The `tun_drv` field provides access to the driver-level resources and configuration.

```

iflags = IFF_MULTICAST;
if ((tp->tun_flags & TUN_L2) != 0) {
    type = IFT_ETHER;
    iflags |= IFF_BROADCAST | IFF_SIMPLEX;
} else {
    type = IFT_PPP;
    iflags |= IFF_POINTOPOINT;
}

```

6.14.3.5.2 Determining Interface Type and Flags The interface type and behavior flags depend on whether this is a layer 2 (Ethernet) or layer 3 (IP) tunnel:

Layer 2 devices (tap/vmnet with `TUN_L2` set):

- `IFT_ETHER` - declares this as an Ethernet interface
- `IFF_BROADCAST` - supports broadcast transmission
- `IFF_SIMPLEX` - cannot receive its own transmissions (standard for Ethernet)

- `IFF_MULTICAST` - supports multicast groups

Layer 3 devices (tun without `TUN_L2`):

- `IFT_PPP` - declares this as a point-to-point protocol interface
- `IFF_POINTOPOINT` - has exactly one peer (no broadcast domain)
- `IFF_MULTICAST` - supports multicast (though less meaningful for point-to-point)

These flags control how the network stack treats the interface. For example, routing code uses `IFF_POINTOPOINT` to determine whether a route needs a gateway address or just a destination.

```
ifp = tp->tun_ifp = if_alloc(type);
ifp->if_softc = tp;
if_initname(ifp, drv->cdevsw.d_name, dev2unit(dev));
```

6.14.3.5.3 Allocating and Initializing the Interface The `if_alloc` function allocates a struct `ifnet` of the specified type. This structure is the network stack's representation of the interface, containing packet queues, statistics counters, capability flags, and function pointers.

Three critical linkages are established:

1. `tp->tun_ifp = if_alloc(type)` - `softc` points to `ifnet`
2. `ifp->if_softc = tp` - `ifnet` points back to `softc`
3. `if_initname(ifp, drv->cdevsw.d_name, dev2unit(dev))` - associates the interface name ("tun0") with the `ifnet`

The bidirectional linkage allows code working with either representation to access the other. Network code receiving a packet can find the character device state; character device operations can access network statistics.

```
ifp->if_ioctl = tunifioctl;
ifp->if_flags = iflags;
IFQ_SET_MAXLEN(&ifp->if_snd, ifqmaxlen);
```

6.14.3.5.4 Configuring Interface Operations The `if_ioctl` function pointer handles interface configuration requests like `SIOCSIFADDR` (set address), `SIOCSIFMTU` (set MTU), and `SIOCSIFFLAGS` (set flags). This is distinct from the character device's `ioctl` handler, which processes device-specific commands.

The interface flags are copied from the previously determined `iflags` value. The send queue's maximum length is set to `ifqmaxlen` (typically 50), limiting how many packets can await transmission to userspace.

```

ifp->if_capabilities |= IFCAP_LINKSTATE | IFCAP_MEXTPG;
if ((tp->tun_flags & TUN_L2) != 0)
    ifp->if_capabilities |=
        IFCAP_RXCSUM | IFCAP_RXCSUM_IPV6 | IFCAP_LRO;
ifp->if_capenable |= IFCAP_LINKSTATE | IFCAP_MEXTPG;

```

6.14.3.5.5 Setting Interface Capabilities Interface capabilities declare what hardware offload features the device supports. Two sets of flags exist:

- `if_capabilities` - features the interface can support
- `if_capenable` - features currently enabled

All interfaces support:

- `IFCAP_LINKSTATE` - can report link up/down state changes
- `IFCAP_MEXTPG` - supports multi-page external mbufs (zero-copy optimization)

Layer 2 interfaces additionally support:

- `IFCAP_RXCSUM` - receive checksum offload for IPv4
- `IFCAP_RXCSUM_IPV6` - receive checksum offload for IPv6
- `IFCAP_LRO` - Large Receive Offload (TCP segment coalescing)

These capabilities are initially disabled for tap/vmnet devices. When userspace enables virtio-net header mode via the `TAPSVNETHDR` ioctl, additional transmit capabilities become available, and the code updates these flags accordingly.

```

if ((tp->tun_flags & TUN_L2) != 0) {
    ifp->if_init = tunifinit;
    ifp->if_start = tunstart_l2;
    ifp->if_transmit = tap_transmit;
    ifp->if_qflush = if_qflush;

    ether_gen_addr(ifp, &eaddr);
    ether_ifattach(ifp, eaddr.octet);
}

```

6.14.3.5.6 Layer 2 Interface Registration For Ethernet interfaces, four function pointers configure packet processing:

- `if_init` - called when the interface transitions to the up state
- `if_start` - legacy packet transmission (called by the send queue)
- `if_transmit` - modern packet transmission (bypasses send queue when possible)
- `if_qflush` - discards queued packets

The `ether_gen_addr` function generates a random MAC address for the local side of the tunnel. The address uses the locally-administered bit pattern, ensuring it doesn't conflict with real hardware addresses.

`ether_ifattach` performs Ethernet-specific registration:

- Registers the interface with the network stack
- Attaches BPF (Berkeley Packet Filter) with `DLT_EN10MB` (Ethernet) link type
- Initializes the interface's link-layer address structure
- Sets up multicast filter management

After `ether_ifattach`, the interface is fully operational and visible to userspace tools.

```
} else {
    ifp->if_mtu = TUNMTU;
    ifp->if_start = tunstart;
    ifp->if_output = tunoutput;

    ifp->if_snd.ifq_drv_maxlen = 0;
    IFQ_SET_READY(&ifp->if_snd);

    if_attach(ifp);
    bpfattach(ifp, DLT_NULL, sizeof(u_int32_t));
}
```

6.14.3.5.7 Layer 3 Interface Registration

Point-to-point interfaces follow a simpler path:

The MTU is set to `TUNMTU` (typically 1500), and two packet transmission functions are installed:

- `if_start` - handles packets from the send queue
- `if_output` - called directly by the routing code

The `if_snd.ifq_drv_maxlen = 0` setting is significant, it prevents the legacy send queue from holding packets, as the modern path uses `if_transmit` semantics even though the function pointer isn't set. `IFQ_SET_READY` marks the queue as operational.

`if_attach` registers the interface with the network stack, making it visible to routing and configuration tools.

`bpfattach` enables packet capture with `DLT_NULL` link type. This link type prepends a 4-byte address family field (`AF_INET` or `AF_INET6`) to each packet, allowing tools like `tcpdump` to distinguish IPv4 from IPv6 traffic without examining packet contents.

```
TUN_LOCK(tp);
tp->tun_flags |= TUN_INITED;
TUN_UNLOCK(tp);
```

6.14.3.5.8 Marking Initialization Complete The `TUN_INITED` flag signals that the interface is fully constructed. Other code paths check this flag before performing operations. For example, the device open function verifies that both `TUN_INITED` and `TUN_OPEN` are set before allowing I/O.

The mutex protects this flag from races where one thread checks the state while another is still initializing.

6.14.3.5.9 The Completed Interface After `tuncreate` returns, both the character device and network interface exist and are cross-linked:

```
/dev/tun0 (struct cdev)
    <-> si_drv1 / tun_dev
tuntap_softc
    <-> if_softc / tun_ifp
tun0 (struct ifnet)
```

Opening `/dev/tun0` with `open(2)` allows userspace to read and write packets. Transmitting packets to the `tun0` interface via `sendto(2)` or routing queues them for userspace to read. This bidirectional connection enables userspace VPN and virtualization software to implement custom network protocols while plugging into the kernel's network stack.

6.14.3.6 3) `open(2)`: vnet context, mark open, link up

```
1064: static int
1065: tunopen(struct cdev *dev, int flag, int mode, struct thread *td)
1066: {
1067:     struct ifnet *ifp;
1068:     struct tuntap_softc *tp;
1069:     int error __diagused, tunflags;
1070:
1071:     tunflags = 0;
1072:     CURVNET_SET(TD_TO_VNET(td));
1073:     error = tuntap_name2info(dev->si_name, NULL, &tunflags);
1074:     if (error != 0) {
1075:         CURVNET_RESTORE();
1076:         return (error); /* Shouldn't happen */
1077:     }
1078:
1079:     tp = dev->si_drv1;
1080:     KASSERT(tp != NULL,
1081:         ("si_drv1 should have been initialized at creation"));
1082:
1083:     TUN_LOCK(tp);
1084:     if ((tp->tun_flags & TUN_INITED) == 0) {
```

```
1085:     TUN_UNLOCK(tp);
1086:     CURVNET_RESTORE();
1087:     return (ENXIO);
1088: }
1089: if ((tp->tun_flags & (TUN_OPEN | TUN_DYING)) != 0) {
1090:     TUN_UNLOCK(tp);
1091:     CURVNET_RESTORE();
1092:     return (EBUSY);
1093: }
1094:
1095: error = tun_busy_locked(tp);
1096: KASSERT(error == 0, ("Must be able to busy an unopen tunnel"));
1097: ifp = TUN2IFP(tp);
1098:
1099: if ((tp->tun_flags & TUN_L2) != 0) {
1100:     bcopy(IF_LLADDR(ifp), tp->tun_ether.octet,
1101:         sizeof(tp->tun_ether.octet));
1102:
1103:     ifp->if_drv_flags |= IFF_DRV_RUNNING;
1104:     ifp->if_drv_flags &= ~IFF_DRV_OACTIVE;
1105:
1106:     if (tapuponopen)
1107:         ifp->if_flags |= IFF_UP;
1108: }
1109:
1110: tp->tun_pid = td->td_proc->p_pid;
1111: tp->tun_flags |= TUN_OPEN;
1112:
1113: if_link_state_change(ifp, LINK_STATE_UP);
1114: TUNDEBUG(ifp, "open\n");
1115: TUN_UNLOCK(tp);
1116: /* ... cdevpriv setup ... */
1117: (void)devfs_set_cdevpriv(tp, tundtor);
1118: CURVNET_RESTORE();
1119: return (0);
1120: }
```

The `tunopen` function handles the `open(2)` system call on `tun/tap/vmnet` character devices. This is the entry point where userspace applications like VPN daemons or virtual machine monitors gain control over a network interface. Opening the device transitions it from an initialized but inactive state to an operational state ready for packet I/O.

```
static int
tunopen(struct cdev *dev, int flag, int mode, struct thread *td)
{
    CURVNET_SET(TD_TO_VNET(td));
```

6.14.3.6.1 Function Signature and Virtual Network Context The function receives the standard character device open parameters: the device being opened, flags from the open(2) call, mode bits, and the thread performing the operation.

The CURVNET_SET macro is critical for FreeBSD’s VNET (virtual network stack) support. In systems using jails or virtualization, multiple independent network stacks may exist. This macro switches to the network context associated with the opening thread’s jail or vnet, ensuring all subsequent network operations affect the correct stack. Every function that touches network interfaces or routing tables must bracket its work between CURVNET_SET and CURVNET_RESTORE.

```
tunflags = 0;
error = tuntap_name2info(dev->si_name, NULL, &tunflags);
if (error != 0) {
    CURVNET_RESTORE();
    return (error);
}
```

6.14.3.6.2 Device Type Validation Although the device should already exist and be properly typed, this code validates that the device name still corresponds to a known tun/tap/vmnet variant. The check should always succeed, as indicated by the comment “Shouldn’t happen”. The validation guards against corrupted kernel state or race conditions during device destruction.

```
tp = dev->si_drv1;
KASSERT(tp != NULL,
    ("si_drv1 should have been initialized at creation"));

TUN_LOCK(tp);
if ((tp->tun_flags & TUN_INITED) == 0) {
    TUN_UNLOCK(tp);
    CURVNET_RESTORE();
    return (ENXIO);
}
```

6.14.3.6.3 Retrieving and Validating Device State The softc is retrieved via the si_drv1 link established during device creation. The assertion verifies this fundamental invariant.

The softc mutex is acquired before checking state flags, preventing race conditions. The TUN_INITED flag check ensures the network interface was successfully created. If initialization failed or hasn't completed yet, the open fails with ENXIO (device not configured).

```
if ((tp->tun_flags & (TUN_OPEN | TUN_DYING)) != 0) {
    TUN_UNLOCK(tp);
    CURVNET_RESTORE();
    return (EBUSY);
}
```

6.14.3.6.4 Enforcing Exclusive Access Tun/tap devices enforce exclusive access, only one process may have a device open at a time. This design simplifies packet routing: there's always exactly one userspace consumer for packets arriving at the interface.

The check examines two flags:

- TUN_OPEN - device is already open by another process
- TUN_DYING - device is being destroyed

Either condition returns EBUSY, informing userspace that the device is unavailable. This prevents scenarios where multiple VPN daemons fight over the same tunnel or where a process opens a device mid-destruction.

```
error = tun_busy_locked(tp);
KASSERT(error == 0, ("Must be able to busy an unopen tunnel"));
ifp = TUN2IFP(tp);
```

6.14.3.6.5 Marking the Device Busy The busy mechanism prevents device destruction while operations are in progress. The tun_busy_locked function increments the tun_busy counter and fails if TUN_DYING is set.

The assertion verifies that marking the device busy must succeed, since we hold the lock and already checked that neither TUN_OPEN nor TUN_DYING is set, no concurrent destruction can be occurring.

The TUN2IFP macro extracts the ifnet pointer from the softc, providing access to the network interface for subsequent configuration.

```
if ((tp->tun_flags & TUN_L2) != 0) {
    bcopy(IF_LLADDR(ifp), tp->tun_ether.octet,
        sizeof(tp->tun_ether.octet));

    ifp->if_drv_flags |= IFF_DRV_RUNNING;
    ifp->if_drv_flags &= ~IFF_DRV_OACTIVE;
```



```
if (tapuponopen)
    ifp->if_flags |= IFF_UP;
}
```

6.14.3.6.6 Layer 2 Interface Activation For Ethernet interfaces (tap/vmnet), opening the device activates several features:

The MAC address is copied from the interface to `tp->tun_ether`. This snapshot preserves the “remote” MAC address that userspace might need. While the interface itself knows its local MAC address, the softc stores this copy for symmetric access patterns.

Two driver flags are updated:

- `IFF_DRV_RUNNING` - signals that the driver is ready to transmit and receive
- `IFF_DRV_OACTIVE` - cleared to indicate output is not blocked

These “driver flags” (`if_drv_flags`) are distinct from interface flags (`if_flags`). Driver flags reflect the device driver’s internal state, while interface flags reflect administratively configured properties.

The `tapuponopen` sysctl controls whether opening the device automatically marks the interface administratively up. When enabled, `ifp->if_flags |= IFF_UP` brings the interface up without requiring a separate `ifconfig tap0 up` command. This convenience feature is disabled by default to maintain traditional Unix semantics where device availability and interface state are orthogonal.

```
tp->tun_pid = td->td_proc->p_pid;
tp->tun_flags |= TUN_OPEN;
```

6.14.3.6.7 Recording Ownership The controlling process’s PID is recorded in `tun_pid`. This information appears in `ifconfig` output and helps administrators identify which process owns each tunnel. While not used for access control (the file descriptor provides that), it’s valuable for debugging and monitoring.

The `TUN_OPEN` flag is set, transitioning the device into the open state. Subsequent open attempts will now fail with `EBUSY` until this process closes the device.

```
if_link_state_change(ifp, LINK_STATE_UP);
TUNDEBUG(ifp, "open\n");
TUN_UNLOCK(tp);
```

6.14.3.6.8 Signaling Link State The `if_link_state_change` call notifies the network stack that the interface’s link is now up. This generates routing socket messages that daemons like `devd` can monitor, and it updates the interface’s link state visible in `ifconfig` output.

For physical Ethernet interfaces, link state reflects cable connection status. For tun/tap devices, link state reflects whether userspace has the device open. This semantic mapping allows routing protocols and management tools to treat virtual interfaces consistently with physical ones.

The debug message logs the open event, and the mutex is released before the final setup step.

```
(void)devfs_set_cdevpriv(tp, tundtor);  
CURVNET_RESTORE();  
return (0);
```

6.14.3.6.9 Establishing Close Notification The `devfs_set_cdevpriv` call associates the `softc` with this file descriptor and registers `tundtor` (tunnel destructor) as the cleanup function. When the file descriptor is closed, whether explicitly via `close(2)` or implicitly via process termination, the kernel automatically invokes `tundtor` to tear down the device state.

This mechanism provides robust cleanup semantics. Even if a process crashes or is killed, the kernel ensures proper device shutdown. The function pointer and data association are per-file-descriptor, allowing the same device to be opened multiple times in succession (though not concurrently) with correct cleanup for each instance.

The return value 0 signals successful open. At this point, userspace can begin reading packets transmitted to the interface and writing packets to inject into the network stack.

6.14.3.6.10 State Transitions The open operation transitions the device through several states:

```
Device created -> TUN_INITED set  
->  
tunopen() called  
->  
Check exclusive access  
->  
Mark busy (prevent destruction)  
->  
Configure interface (L2: set RUNNING, optionally set UP)  
->  
Record owner PID  
->  
Set TUN_OPEN flag  
->  
Signal link state UP  
->  
Register close handler  
->  
Device ready for I/O
```

After successful open, the device exists in three interlinked representations:

- Character device node (/dev/tun0) with an open file descriptor
- Network interface (tun0) with link state UP
- Softc structure binding them with TUN_OPEN set

Packets can now flow bidirectionally: the network stack queues outbound packets for userspace to read, and userspace writes inbound packets for the network stack to process.

6.14.3.7 4) read(2): userspace receives a whole packet (or EWOULDBLOCK)

```

1706: /*
1707:  * The cdevsw read interface - reads a packet at a time, or at
1708:  * least as much of a packet as can be read.
1709:  */
1710: static int
1711: tunread(struct cdev *dev, struct uio *uio, int flag)
1712: {
1713:     struct tuntap_softc *tp = dev->si_drv1;
1714:     struct ifnet *ifp = TUN2IFP(tp);
1715:     struct mbuf *m;
1716:     size_t len;
1717:     int error = 0;
1718:
1719:     TUNDEBUG (ifp, "read\n");
1720:     TUN_LOCK(tp);
1721:     if ((tp->tun_flags & TUN_READY) != TUN_READY) {
1722:         TUN_UNLOCK(tp);
1723:         TUNDEBUG (ifp, "not ready 0x%x\n", tp->tun_flags);
1724:         return (EHOSTDOWN);
1725:     }
1726:
1727:     tp->tun_flags &= ~TUN_RWAIT;
1728:
1729:     for (;;) {
1730:         IFQ_DEQUEUE(&ifp->if_snd, m);
1731:         if (m != NULL)
1732:             break;
1733:         if (flag & O_NONBLOCK) {
1734:             TUN_UNLOCK(tp);
1735:             return (EWOULDBLOCK);
1736:         }
1737:         tp->tun_flags |= TUN_RWAIT;
1738:         error = mtx_sleep(tp, &tp->tun_mtx, PCATCH | (PZERO + 1),

```

```

1739:         "tunread", 0);
1740:     if (error != 0) {
1741:         TUN_UNLOCK(tp);
1742:         return (error);
1743:     }
1744: }
1745: TUN_UNLOCK(tp);
1746:
1747: len = min(tp->tun_vhdrlen, uio->uio_resid);
1748: if (len > 0) {
1749:     struct virtio_net_hdr_mrg_rxbuf vhdr;
1750:
1751:     bzero(&vhdr, sizeof(vhdr));
1752:     if (m->m_pkthdr.csum_flags & TAP_ALL_OFFLOAD) {
1753:         m = virtio_net_tx_offload(ifp, m, false, &vhdr.hdr);
1754:     }
1755:
1756:     TUNDEBUG(ifp, "txvhdr: f %u, gt %u, hl %u, "
1757:         "gs %u, cs %u, co %u\n", vhdr.hdr.flags,
1758:         vhdr.hdr.gso_type, vhdr.hdr.hdr_len,
1759:         vhdr.hdr.gso_size, vhdr.hdr.csum_start,
1760:         vhdr.hdr.csum_offset);
1761:     error = uiomove(&vhdr, len, uio);
1762: }
1763: if (error == 0)
1764:     error = m_mbuftouio(uio, m, 0);
1765: m_freem(m);
1766: return (error);
1767: }

```

The `tunread` function implements the `read(2)` system call for `tun/tap` devices, transferring packets from the kernel's network stack to userspace. This is the critical path where packets destined for transmission on the virtual network interface become available to VPN daemons, virtual machine monitors, or other userspace networking applications.

```

static int
tunread(struct cdev *dev, struct uio *uio, int flag)
{
    struct tuntap_softc *tp = dev->si_drv1;
    struct ifnet *ifp = TUN2IFP(tp);
    struct mbuf *m;
    size_t len;

```

```
int error = 0;
```

6.14.3.7.1 Function Overview and Context Retrieval The function receives the standard `read(2)` parameters: the device being read, a `uio` (user I/O) structure describing the userspace buffer, and flags from the `open(2)` call (particularly `O_NONBLOCK`).

The `softc` and interface pointers are retrieved via the established linkages. The `mbuf` pointer `m` will hold the packet being transferred, while `len` tracks how much data to copy.

```
TUNDEBUG(ifp, "read\n");
TUN_LOCK(tp);
if ((tp->tun_flags & TUN_READY) != TUN_READY) {
    TUN_UNLOCK(tp);
    TUNDEBUG(ifp, "not ready 0x%x\n", tp->tun_flags);
    return (EHOSTDOWN);
}
```

6.14.3.7.2 Device Readiness Check The `TUN_READY` macro combines two flags: `TUN_OPEN` | `TUN_INITED`. Both must be set for I/O to proceed:

- `TUN_INITED` - the network interface was successfully created
- `TUN_OPEN` - a process has opened the device

If either condition fails, the read returns `EHOSTDOWN`, signaling that the network path is unavailable. This error code is semantically appropriate, from the kernel's perspective, packets are being sent to a "host" (userspace), but that host is down.

```
tp->tun_flags &= ~TUN_RWAIT;
```

6.14.3.7.3 Preparing for Packet Retrieval The `TUN_RWAIT` flag tracks whether a reader is blocked waiting for packets. Clearing it before entering the loop ensures correct state regardless of how the previous read completed, whether it retrieved a packet, timed out, or was interrupted.

```
for (;;) {
    IFQ_DEQUEUE(&ifp->if_snd, m);
    if (m != NULL)
        break;
    if (flag & O_NONBLOCK) {
        TUN_UNLOCK(tp);
    }
}
```

```

        return (EWOULDBLOCK);
    }
    tp->tun_flags |= TUN_RWAIT;
    error = mtx_sleep(tp, &tp->tun_mtx, PCATCH | (PZERO + 1),
        "tunread", 0);
    if (error != 0) {
        TUN_UNLOCK(tp);
        return (error);
    }
}
TUN_UNLOCK(tp);

```

6.14.3.7.4 The Packet Dequeue Loop This loop implements the standard kernel pattern for blocking I/O with non-blocking mode support.

Packet retrieval: `IFQ_DEQUEUE` atomically removes the head packet from the interface's send queue. This macro handles queue locking internally and returns `NULL` if the queue is empty.

Success path: When `m != NULL`, a packet was successfully dequeued, and the loop exits.

Non-blocking path: If the queue is empty and `O_NONBLOCK` was specified during `open(2)`, the read immediately returns `EWOULDBLOCK` (also known as `EAGAIN`). This allows userspace to use `poll(2)`, `select(2)`, or `kqueue(2)` to wait efficiently for readable conditions without blocking the thread.

Blocking path: For blocking reads, the code:

1. Sets `TUN_RWAIT` to indicate a reader is waiting
2. Calls `mtx_sleep` to block the thread atomically

The `mtx_sleep` function atomically releases `tp->tun_mtx` and puts the thread to sleep. When woken (by `tunstart` or `tunstart_l2` when packets arrive), it reacquires the mutex before returning.

The sleep parameters specify:

- `tp` - the wait channel (arbitrary unique pointer, using the softc)
- `&tp->tun_mtx` - mutex to release/reacquire atomically
- `PCATCH | (PZERO + 1)` - allow signal interruption, priority just above normal
- `"tunread"` - name for debugging (shows in `ps` or `top`)
- `0` - no timeout (sleep indefinitely)

Signal handling: If interrupted by a signal (like `SIGINT`), `mtx_sleep` returns an error (typically `EINTR` or `ERESTART`), and the function propagates this to userspace. This allows `Ctrl+C` to interrupt a blocked read.

After successfully dequeuing a packet, the mutex is released. The remainder of the function operates on the mbuf without holding locks, avoiding contention with packet transmission threads.

```

len = min(tp->tun_vhdrlen, uio->uio_resid);
if (len > 0) {

```

```

struct virtio_net_hdr_mrg_rxbuf vhdr;

bzero(&vhdr, sizeof(vhdr));
if (m->m_pkthdr.csum_flags & TAP_ALL_OFFLOAD) {
    m = virtio_net_tx_offload(ifp, m, false, &vhdr.hdr);
}
/* ... debug output ... */
error = uiomove(&vhdr, len, uio);
}

```

6.14.3.7.5 Virtio-Net Header Processing For tap devices configured with virtio-net header mode (via the TAPSVNETHDR ioctl), packets are prefixed with a metadata header describing offload features. This optimization allows userspace (particularly QEMU/KVM) to use hardware offload capabilities:

The `tun_vhdr.len` field is zero for standard mode and non-zero (typically 10 or 12 bytes) when virtio headers are enabled. The code only processes headers if both the header is enabled (`len > 0`) and the userspace buffer has room (`uio->uio_resid`).

The `vhdr` structure is zero-initialized to provide safe defaults. If the mbuf has offload flags set (TAP_ALL_OFFLOAD includes TCP/UDP checksum offload and TSO), `virtio_net_tx_offload` populates the header with:

- Checksum computation parameters (where to start, where to insert)
- Segmentation parameters (MSS, header length)
- Generic flags (whether header is valid)

The `uiomove(&vhdr, len, uio)` call copies the header to userspace. This function handles the kernel-to-user memory transfer, updating `uio` to reflect consumed buffer space. If this copy fails (typically due to invalid userspace pointer), the error is recorded but processing continues to free the mbuf.

```

if (error == 0)
    error = m_mbuftouio(uio, m, 0);
m_freem(m);
return (error);

```

6.14.3.7.6 Packet Data Transfer Assuming the header transfer succeeded (or no header was required), `m_mbuftouio` copies the packet data from the mbuf chain to the userspace buffer. This function: - Walks the mbuf chain (packets may be fragmented across multiple mbufs) - Copies each segment to userspace via `uiomove` - Updates `uio->uio_resid` to reflect remaining buffer space - Returns an error if the buffer is too small or pointers are invalid

The `m_freem` call releases the mbuf back to the kernel's memory pool. This must always execute, even if earlier operations failed, to prevent memory leaks. The mbuf is freed regardless of whether the copy succeeded, once dequeued from the send queue, the packet's fate is sealed.

6.14.3.7.7 Data Flow Summary The complete path from network transmission to userspace read:

```
Application calls send()/sendto()
->
Kernel routing selects tun0 interface
->
tunoutput() or tap_transmit() enqueues mbuf
->
tunstart()/tunstart_l2() wakes blocked reader
->
tunread() dequeues mbuf from if_snd
->
Optional: Generate virtio-net header
->
Copy header to userspace (if enabled)
->
Copy packet data to userspace
->
Free mbuf
->
Userspace receives packet data
```

6.14.3.7.8 Error Handling Semantics The function returns several distinct error codes with specific meanings:

- EHOSTDOWN - device not ready (not open or not initialized)
- EWOULDBLOCK - non-blocking read, no packets available
- EINTR/ERESTART - interrupted by signal while waiting
- EFAULT - userspace buffer pointer invalid
- 0 - success, packet transferred

These error codes allow userspace to distinguish between transient conditions (like EWOULDBLOCK requiring retry) and permanent failures (like EHOSTDOWN requiring device reopen).

6.14.3.7.9 Blocking and Wakeup Coordination The TUN_RWAIT flag and mtx_sleep coordination ensure efficient resource usage. When no packets are available:

1. Reader blocks in `mtx_sleep`, consuming no CPU
2. When the network stack transmits a packet, `tunstart` or `tunstart_l2` executes
3. Those functions check TUN_RWAIT and call `wakeup(tp)` if set
4. The sleeping thread wakes, loops, and dequeues the packet

This pattern avoids polling loops while ensuring prompt packet delivery. The mutex protects against races where packets arrive between the empty queue check and the sleep call.

6.14.3.8 5) write(2): userspace injects a packet (L2 vs L3 path)

6.14.3.9 5.1 Main write dispatcher (tunwrite)

```

1896: /*
1897:  * the cdevsw write interface - an atomic write is a packet - or else!
1898:  */
1899: static int
1900: tunwrite(struct cdev *dev, struct uio *uio, int flag)
1901: {
1902:     struct virtio_net_hdr_mrg_rxbuf vhdr;
1903:     struct tuntap_softc *tp;
1904:     struct ifnet *ifp;
1905:     struct mbuf *m;
1906:     uint32_t mru;
1907:     int align, vhdrlen, error;
1908:     bool l2tun;
1909:
1910:     tp = dev->si_drv1;
1911:     ifp = TUN2IFP(tp);
1912:     TUNDEBUG(ifp, "tunwrite\n");
1913:     if ((ifp->if_flags & IFF_UP) != IFF_UP)
1914:         /* ignore silently */
1915:         return (0);
1916:
1917:     if (uio->uio_resid == 0)
1918:         return (0);
1919:
1920:     l2tun = (tp->tun_flags & TUN_L2) != 0;
1921:     mru = l2tun ? TAPMRU : TUNMRU;
1922:     vhdrlen = tp->tun_vhdrlen;
1923:     align = 0;
1924:     if (l2tun) {
1925:         align = ETHER_ALIGN;
1926:         mru += vhdrlen;
1927:     } else if ((tp->tun_flags & TUN_IFHEAD) != 0)
1928:         mru += sizeof(uint32_t); /* family */
1929:     if (uio->uio_resid < 0 || uio->uio_resid > mru) {
1930:         TUNDEBUG(ifp, "len=%zd!\n", uio->uio_resid);
1931:         return (EIO);
1932:     }
1933:
1934:     if (vhdrlen > 0) {
1935:         error = uiomove(&vhdr, vhdrlen, uio);

```

```

1936:     if (error != 0)
1937:         return (error);
1938:     TUNDEBBUG(ifp, "txvhdr: f %u, gt %u, hl %u, "
1939:         "gs %u, cs %u, co %u\n", vhdr.hdr.flags,
1940:         vhdr.hdr.gso_type, vhdr.hdr.hdr_len,
1941:         vhdr.hdr.gso_size, vhdr.hdr.csum_start,
1942:         vhdr.hdr.csum_offset);
1943: }
1944:
1945: if ((m = m_uiotombuf(uio, M_NOWAIT, 0, align, M_PKTHDR)) == NULL) {
1946:     if_inc_counter(ifp, IFCOUNTER_IERRORS, 1);
1947:     return (ENOBUFS);
1948: }
1949:
1950: m->m_pkthdr.rcvif = ifp;
1951: #ifdef MAC
1952:     mac_ifnet_create_mbuf(ifp, m);
1953: #endif
1954:
1955: if (l2tun)
1956:     return (tunwrite_l2(tp, m, vhdrlen > 0 ? &vhdr : NULL));
1957:
1958: return (tunwrite_l3(tp, m));
1959: }

```

The `tunwrite` function implements the `write(2)` system call for `tun/tap` devices, injecting packets from userspace into the kernel's network stack. This is the complementary operation to `tunread`, where `tunread` delivers kernel-originated packets to userspace, `tunwrite` accepts userspace packets for kernel processing. The comment “an atomic write is a packet - or else!” emphasizes a critical design principle: each `write(2)` call must contain exactly one complete packet.

```

static int
tunwrite(struct cdev *dev, struct uio *uio, int flag)
{
    struct virtio_net_hdr_mrg_rxbuf vhdr;
    struct tuntap_softc *tp;
    struct ifnet *ifp;
    struct mbuf *m;
    uint32_t mru;
    int align, vhdrlen, error;
    bool l2tun;

```

```
tp = dev->si_drv1;
ifp = TUN2IFP(tp);
```

6.14.3.9.1 Function Initialization and Context The function retrieves the device context through the standard `si_drv1` linkage. Local variables track the maximum receive unit, alignment requirements, virtio header length, and whether this is a layer 2 interface.

```
TUNDEBBUG(ifp, "tunwrite\n");
if ((ifp->if_flags & IFF_UP) != IFF_UP)
    /* ignore silently */
    return (0);

if (uio->uio_resid == 0)
    return (0);
```

6.14.3.9.2 Interface State Validation Two early checks filter out invalid operations:

Interface down check: If the interface is administratively down (not marked `IFF_UP`), the write succeeds immediately without processing the packet. This silent discard behavior differs from the read path, which returns `EHOSTDOWN` when not ready. The asymmetry makes sense: applications writing packets shouldn't fail when the interface is temporarily down, packets are simply dropped, mimicking what would happen on a real network interface with no carrier.

Zero-length write: Writing zero bytes is treated as a no-op success. This handles edge cases like `write(fd, buf, 0)` without error.

```
l2tun = (tp->tun_flags & TUN_L2) != 0;
mru = l2tun ? TAPMRU : TUNMRU;
vhdrlen = tp->tun_vhdrlen;
align = 0;
if (l2tun) {
    align = ETHER_ALIGN;
    mru += vhdrlen;
} else if ((tp->tun_flags & TUN_IFHEAD) != 0)
    mru += sizeof(uint32_t);
```

6.14.3.9.3 Determining Packet Size Limits The Maximum Receive Unit (MRU) depends on the interface type:

- Layer 3 (tun): `TUNMRU` (typically 1500 bytes, standard IPv4 MTU)
- Layer 2 (tap/vmnet): `TAPMRU` (typically 1518 bytes, Ethernet frame size)

Alignment requirements: Layer 2 devices set `align = ETHER_ALIGN` (usually 2 bytes). This ensures the IP header following the 14-byte Ethernet header lands on a 4-byte boundary, which improves performance on architectures with alignment restrictions or cache line efficiency concerns.

Header adjustments: The MRU increases to accommodate:

- Virtio-net headers for tap devices (`vhdr.len` bytes)
- Address family indicator for tun devices in IFHEAD mode (4 bytes)

These headers precede the actual packet data in the userspace buffer but are not part of the on-wire packet format.

```
if (uio->uio_resid < 0 || uio->uio_resid > mru) {
    TUNDEBBUG(ifp, "len=%zd!\n", uio->uio_resid);
    return (EIO);
}
```

6.14.3.9.4 Validating Write Size The write size (`uio->uio_resid`) must fall within valid bounds. Negative sizes are impossible in correct operation but checked for safety. Oversized writes indicate either:

- Application bugs (trying to write jumbo frames without configuration)
- Protocol violations (incorrect packet framing)
- Malicious behavior

The EIO return signals a generic I/O error, appropriate for data that cannot be processed.

```
if (vhdr.len > 0) {
    error = uiomove(&vhdr, vhdr.len, uio);
    if (error != 0)
        return (error);
    TUNDEBBUG(ifp, "txvhdr: f %u, gt %u, hl %u, "
        "gs %u, cs %u, co %u\n", vhdr.hdr.flags,
        vhdr.hdr.gso_type, vhdr.hdr.hdr_len,
        vhdr.hdr.gso_size, vhdr.hdr.csum_start,
        vhdr.hdr.csum_offset);
}
```

6.14.3.9.5 Processing Virtio-Net Headers When virtio-net header mode is enabled (common for VM networking), userspace prepends a small header to each packet describing offload operations:

- **Checksum offload:** Instructs the kernel where to compute and insert checksums
- **Segmentation offload:** For large packets (TSO/GSO), describes how to segment into MTU-sized chunks
- **Receive offload hints:** Indicates checksums already validated by VM guest

The `uiomove` call copies the header from userspace, consuming `vhdr.len` bytes from the user buffer and advancing `uio`. If the copy fails (invalid pointer), the error propagates immediately, a corrupted header cannot be safely processed.

The debug output logs header fields for troubleshooting offload issues. In production builds with `tundebug = 0`, these statements compile away.

```
if ((m = m_uiotombuf(uio, M_NOWAIT, 0, align, M_PKTHDR)) == NULL) {
    if_inc_counter(ifp, IFCOUNTER_IERRORS, 1);
    return (ENOBUFS);
}
```

6.14.3.9.6 Constructing the Mbuf The `m_uiotombuf` function is the kernel’s utility for converting userspace data into the network stack’s native packet format (mbuf chains). Its parameters specify:

- `uio` - source data from userspace
- `M_NOWAIT` - don’t sleep for memory (return `NULL` immediately if allocation fails)
- `0` - no maximum length (use all remaining `uio_resid` bytes)
- `align` - start packet data this many bytes into the first mbuf
- `M_PKTHDR` - allocate an mbuf with packet header (required for network packets)

Memory allocation failure: If `m_uiotombuf` returns `NULL`, the system is out of mbuf memory. The `IFCOUNTER_IERRORS` counter increments (visible in `netstat -i`), and `ENOBUFS` informs userspace of temporary resource exhaustion. Applications should generally retry after a brief delay.

The `M_NOWAIT` policy: Using `M_NOWAIT` rather than `M_WAITOK` prevents userspace writes from blocking indefinitely when memory is low. This is appropriate for the write path, if memory isn’t available now, failing quickly allows the application to handle backpressure.

```
m->m_pkthdr.rcvif = ifp;
#ifdef MAC
mac_ifnet_create_mbuf(ifp, m);
#endif
```

6.14.3.9.7 Setting Packet Metadata Two pieces of metadata are attached to the packet:

Receive interface: `m_pkthdr.rcvif` records which interface received the packet. This seems counterintuitive; we’re injecting a packet, not receiving one, but from the kernel’s perspective, packets written to `/dev/tun0` are “received” on the `tun0` interface. This field is used for:

- Firewall rules (ipfw, pf) that filter based on incoming interface
- Routing decisions that consider packet source
- Accounting that attributes traffic to specific interfaces

MAC Framework labeling: If the Mandatory Access Control framework is enabled, `mac_ifnet_create_mbuf` applies security labels to the packet based on the interface's policy. This supports systems using TrustedBSD MAC for fine-grained network security.

```
if (l2tun)
    return (tunwrite_l2(tp, m, vhdrlen > 0 ? &vhdr : NULL));

return (tunwrite_l3(tp, m));
```

6.14.3.9.8 Dispatching by Layer The final step delegates to layer-specific processing functions:

Layer 2 path (`tunwrite_l2`): For tap/vmnet devices, the mbuf contains a complete Ethernet frame. The function: - Validates the Ethernet header - Applies virtio-net offload hints if present - Injects the frame into the Ethernet processing path - Potentially processes through LRO (Large Receive Offload)

Layer 3 path (`tunwrite_l3`): For tun devices, the mbuf contains a raw IP packet (possibly preceded by an address family indicator in IFHEAD mode). The function: - Extracts the protocol family (IPv4 vs IPv6) - Dispatches to the appropriate network layer protocol handler - Bypasses link-layer processing entirely

Both functions assume ownership of the mbuf, they will either successfully inject it into the network stack or free it on error. The caller should not access the mbuf after these calls return.

6.14.3.9.9 Data Flow Summary The complete path from userspace write to kernel network processing:

```
Application calls write(fd, packet, len)
->
tunwrite() validates interface state and size
->
Extract virtio-net header (if enabled)
->
Copy packet data from userspace to mbuf
->
Set mbuf metadata (rcvif, MAC labels)
->
Layer 2: tunwrite_l2()          Layer 3: tunwrite_l3()
->                             ->
Validate Ethernet header       Extract address family
->                             ->
Apply offload hints            Dispatch to IP/IPv6
->                             ->
ether_input() / LRO            netisr_dispatch()
->                             ->
Network stack processes packet
->
Routing, firewall, socket delivery
```

6.14.3.9.10 Atomic Write Semantics The opening comment, “an atomic write is a packet - or else!”, highlights a critical contract: userspace must write complete packets in single `write(2)` calls. The driver provides no buffering or packet assembly:

- Writing 1000 bytes, then 500 bytes creates **two** packets (1000-byte and 500-byte)
- Not “one 1500-byte packet assembled from two writes”

This design simplifies the driver and matches the semantics of real network interfaces, which receive complete frames. Applications that need to construct packets piece-by-piece must buffer in userspace before writing.

6.14.3.9.11 Error Handling and Resource Management The function’s error handling demonstrates defensive programming patterns:

- **Early validation** prevents resource allocation for invalid requests
- **Immediate cleanup** on `m_uio` to mbuf failure (increment error counter, return `ENOBUFS`)
- **Ownership transfer** to layer-specific functions eliminates double-free risks

The only resource allocated (the mbuf) has clear ownership transfer semantics. After calling `tunwrite_l2` or `tunwrite_l3`, the write function never touches it again.

6.14.3.10 5.2 L3 (tun) dispatch to the network stack (netisr)

```
1845: static int
1846: tunwrite_l3(struct tuntap_softc *tp, struct mbuf *m)
1847: {
1848:     struct epoch_tracker et;
1849:     struct ifnet *ifp;
1850:     int family, isr;
1851:
1852:     ifp = TUN2IFP(tp);
1853:     /* Could be unlocked read? */
1854:     TUN_LOCK(tp);
1855:     if (tp->tun_flags & TUN_IFHEAD) {
1856:         TUN_UNLOCK(tp);
1857:         if (m->m_len < sizeof(family) &&
1858:             (m = m_pullup(m, sizeof(family))) == NULL)
1859:             return (ENOBUFS);
1860:         family = ntohl(*mtod(m, u_int32_t *));
1861:         m_adj(m, sizeof(family));
1862:     } else {
1863:         TUN_UNLOCK(tp);
1864:         family = AF_INET;
```

```
1865:     }
1866:
1867:     BPF_MTAP2(ifp, &family, sizeof(family), m);
1868:
1869:     switch (family) {
1870: #ifdef INET
1871:     case AF_INET:
1872:         isr = NETISR_IP;
1873:         break;
1874: #endif
1875: #ifdef INET6
1876:     case AF_INET6:
1877:         isr = NETISR_IPV6;
1878:         break;
1879: #endif
1880:     default:
1881:         m_freem(m);
1882:         return (EAFNOSUPPORT);
1883:     }
1884:     random_harvest_queue(m, sizeof(*m), RANDOM_NET_TUN);
1885:     if_inc_counter(ifp, IFCOUNTER_IBYTES, m->m_pkthdr.len);
1886:     if_inc_counter(ifp, IFCOUNTER_IPACKETS, 1);
1887:     CURVNET_SET(ifp->if_vnet);
1888:     M_SETFIB(m, ifp->if_fib);
1889:     NET_EPOCH_ENTER(et);
1890:     netisr_dispatch(isr, m);
1891:     NET_EPOCH_EXIT(et);
1892:     CURVNET_RESTORE();
1893:     return (0);
1894: }
```

The `tunwrite_l3` function handles packets written to layer 3 (tun) devices, injecting raw IP packets directly into the kernel's network protocol handlers. Unlike layer 2 (tap) devices that process complete Ethernet frames, tun devices work with IP packets that have no link-layer headers, making them ideal for VPN implementations and IP tunneling protocols.

```
static int
tunwrite_l3(struct tuntap_softc *tp, struct mbuf *m)
{
    struct epoch_tracker et;
    struct ifnet *ifp;
    int family, isr;
```



```
ifp = TUN2IFP(tp);
```

6.14.3.10.1 Function Context and Protocol Family Extraction The function receives the `softc` and an `mbuf` containing the packet. The `epoch_tracker` will be used later to ensure safe concurrent access to routing structures. The `family` variable will hold the protocol family (`AF_INET` or `AF_INET6`), and `isr` will identify the appropriate network interrupt service routine.

```
TUN_LOCK(tp);
if (tp->tun_flags & TUN_IFHEAD) {
    TUN_UNLOCK(tp);
    if (m->m_len < sizeof(family) &&
        (m = m_pullup(m, sizeof(family))) == NULL)
        return (ENOBUFS);
    family = ntohl(*mtod(m, u_int32_t *));
    m_adj(m, sizeof(family));
} else {
    TUN_UNLOCK(tp);
    family = AF_INET;
}
```

6.14.3.10.2 Determining Protocol Family Tun devices support two modes for indicating packet protocol:

IFHEAD mode (`TUN_IFHEAD` flag set): Each packet begins with a 4-byte address family indicator in network byte order. This mode, enabled via the `TUNSIFHEAD` ioctl, allows a single tun device to carry both IPv4 and IPv6 traffic. The code:

1. Checks if the first `mbuf` contains at least 4 bytes using `m->m_len`
2. If not, calls `m_pullup` to consolidate the header into the first `mbuf`
3. Extracts the family using `mtod` (`mbuf-to-data` pointer) and converts from network to host byte order with `ntohl`
4. Strips the family indicator with `m_adj`, which advances the data pointer by 4 bytes

The `m_pullup` call can fail if memory is exhausted, returning `NULL`. In this case, the original `mbuf` has already been freed by `m_pullup`, so the function simply returns `ENOBUFS` without calling `m_freem`.

Non-IFHEAD mode (default): All packets are assumed to be IPv4. This legacy mode simplifies applications that only handle IPv4, but prevents multiplexing protocols over one device.

The mutex is held only while reading `tun_flags`, minimizing lock contention. The comment “Could be unlocked read?” questions whether the lock is even necessary, since flags rarely change after initialization, an unlocked read would likely be safe. However, the conservative approach prevents theoretical races.

```
BPF_MTAP2(ifp, &family, sizeof(family), m);
```

6.14.3.10.3 Berkeley Packet Filter Tap The BPF_MTAP2 macro passes the packet to any attached BPF (Berkeley Packet Filter) listeners, typically packet capture tools like tcpdump. The macro name breaks down as:

- **BPF** - Berkeley Packet Filter subsystem
- **MTAP** - tap into the packet stream from an mbuf
- **2** - two-argument variant that prepends metadata

The call prepends the 4-byte family value before the packet data, allowing capture tools to distinguish IPv4 from IPv6 without packet inspection. This matches the link-layer type DLT_NULL configured during interface creation; captured packets have a 4-byte address family header even if the wire format doesn't.

BPF operates efficiently: if no listeners are attached, the macro expands to a simple conditional check that costs only a few instructions. This design allows pervasive instrumentation points throughout the network stack without performance impact when not actively debugging.

```
switch (family) {  
#ifdef INET  
case AF_INET:  
    isr = NETISR_IP;  
    break;  
#endif  
#ifdef INET6  
case AF_INET6:  
    isr = NETISR_IPV6;  
    break;  
#endif  
default:  
    m_freem(m);  
    return (EAFNOSUPPORT);  
}
```

6.14.3.10.4 Protocol Validation and Dispatch Setup The protocol family determines which network layer interrupt service routine (netisr) will process the packet:

- **AF_INET** -> NETISR_IP - IPv4 processing
- **AF_INET6** -> NETISR_IPV6 - IPv6 processing

The `#ifdef` guards are required: if the kernel was compiled without IPv4 or IPv6 support, those cases don't exist, and attempting to inject such packets results in EAFNOSUPPORT (address family not supported).

Unsupported protocol families trigger immediate mbuf deallocation via `m_freem` and return an error. This prevents packets from leaking into the network stack with incorrect metadata that could cause crashes or security issues.

```
random_harvest_queue(m, sizeof(*m), RANDOM_NET_TUN);
```

6.14.3.10.5 Entropy Collection This call contributes entropy to the kernel’s random number generator. Network packet arrival timing is unpredictable and difficult for attackers to manipulate, making it a valuable entropy source. The function samples metadata about the mbuf structure (not packet contents) to seed the random pool.

The `RANDOM_NET_TUN` flag tags the entropy source, allowing the random subsystem to track entropy diversity. Systems relying on `/dev/random` for cryptographic operations benefit from accumulating entropy from multiple independent sources.

```
if_inc_counter(ifp, IFCOUNTER_IBYTES, m->m_pkthdr.len);  
if_inc_counter(ifp, IFCOUNTER_IPACKETS, 1);
```

6.14.3.10.6 Interface Statistics These calls update interface statistics visible via `netstat -i` or `ifconfig`:

- `IFCOUNTER_IBYTES` - total bytes received
- `IFCOUNTER_IPACKETS` - total packets received

From the kernel’s perspective, packets written by userspace are “input” to the interface, hence the use of input counters rather than output counters. This matches the semantic established by setting `m_pkthdr.rcvif` earlier, the packet is being received from userspace.

The `if_inc_counter` function handles atomic updates, ensuring accurate counts even with concurrent packet processing on multiprocessor systems.

```
CURVNET_SET(ifp->if_vnet);  
M_SETFIB(m, ifp->if_fib);
```

6.14.3.10.7 Network Stack Context Setup Two pieces of context are established before injecting the packet:

Virtual network stack: `CURVNET_SET` switches to the network context (vnet) associated with the interface. In systems using jails or network stack virtualization, multiple independent network stacks coexist. This macro ensures routing tables, firewall rules, and socket lookups operate in the correct namespace.

Forwarding Information Base (FIB): `M_SETFIB` tags the packet with the interface’s FIB number. FreeBSD supports multiple routing tables (FIBs), allowing policy-based routing where different applications or interfaces use distinct routing policies. The packet inherits the interface’s FIB, ensuring routes are looked up in the appropriate table.

These settings affect all subsequent packet processing: firewall rules, routing decisions, and socket delivery.

```

NET_EPOCH_ENTER(et);
netisr_dispatch(isr, m);
NET_EPOCH_EXIT(et);
CURVNET_RESTORE();
return (0);

```

6.14.3.10.8 Epoch-Protected Dispatch The critical packet injection occurs within an epoch section:

Network epoch: FreeBSD's network stack uses epoch-based reclamation (a form of read-copy-update) to protect data structures from concurrent access without heavy locking. `NET_EPOCH_ENTER` registers this thread as active in the network epoch, preventing routing entries, interface structures, and other network objects from being deallocated until `NET_EPOCH_EXIT`.

This mechanism enables lock-free reads of routing tables and interface lists, dramatically improving multicore scalability. The epoch tracker `et` maintains the context needed to exit cleanly.

Netisr dispatch: `netisr_dispatch(isr, m)` hands the packet to the network interrupt service routine subsystem. This asynchronous dispatch model decouples packet injection from protocol processing:

1. The packet is queued to the appropriate netisr thread (typically one per CPU core)
2. The calling thread (handling the `write(2)`) returns immediately
3. The netisr thread dequeues and processes the packet asynchronously

This design prevents userspace writes from blocking in complex protocol processing (IP forwarding, firewall evaluation, TCP reassembly). The netisr thread will: - Validate IP headers (checksum, length, version) - Process IP options - Consult routing tables - Apply firewall rules - Deliver to local sockets or forward to other interfaces

Context restoration: `CURVNET_RESTORE` switches back to the calling thread's original network context. This is essential for correctness, without restoration, subsequent operations in the thread would execute in the wrong network namespace.

6.14.3.10.9 Ownership and Lifecycle After `netisr_dispatch`, the function returns success but no longer owns the mbuf. The netisr subsystem assumes responsibility for either: - Delivering the packet to its destination and freeing the mbuf - Dropping the packet (for policy, routing, or validation reasons) and freeing the mbuf

The function never needs to call `m_freem` in the success path, ownership has transferred to the network stack.

6.14.3.10.10 Data Flow Through the Network Stack The complete path after dispatch:

```

tunwrite_l3() injects packet
->
netisr_dispatch() queues to NETISR_IP/NETISR_IPV6
->
Netisr thread dequeues packet
->

```

```

ip_input() / ip6_input() processes
    ->
Routing table lookup
    ->
Firewall evaluation (ipfw, pf)
    ->
    | -> Local delivery: socket input queue
    | -> Forward: ip_forward() -> output interface
    | -> Drop: m_freem()

```

6.14.3.10.11 Error Paths and Resource Management The function has three possible outcomes:

1. **Success** (return 0): Packet dispatched to network stack, mbuf ownership transferred
2. **Pullup failure** (return ENOBUFS): m_pullup freed the mbuf, no further cleanup needed
3. **Unsupported protocol** (return EAFNOSUPPORT): Mbuf explicitly freed with m_freem

All paths correctly manage mbuf ownership, preventing both leaks and double-frees. This careful resource management is characteristic of well-designed kernel code.

6.14.3.11 6) Readiness: poll(2) and kqueue

```

1965:  */
1966:  static      int
1967:  tunpoll(struct cdev *dev, int events, struct thread *td)
1968:  {
1969:      struct tuntap_softc *tp = dev->si_drv1;
1970:      struct ifnet      *ifp = TUN2IFP(tp);
1971:      int      revents = 0;
1972:
1973:      TUNDEBUG(ifp, "tunpoll\n");
1974:
1975:      if (events & (POLLIN | POLLRDNORM)) {
1976:          IFQ_LOCK(&ifp->if_snd);
1977:          if (!IFQ_IS_EMPTY(&ifp->if_snd)) {
1978:              TUNDEBUG(ifp, "tunpoll q=%d\n", ifp->if_snd.ifq_len);
1979:              revents |= events & (POLLIN | POLLRDNORM);
1980:          } else {
1981:              TUNDEBUG(ifp, "tunpoll waiting\n");
1982:              selrecord(td, &tp->tun_rsel);
1983:          }
1984:          IFQ_UNLOCK(&ifp->if_snd);
1985:      }
1986:      revents |= events & (POLLOUT | POLLWRNORM);

```

```

1987:
1988:     return (revents);
1989: }
1990:
1991: /*
1992:  * tunkqfilter - support for the kevent() system call.
1993:  */
1994: static int
1995: tunkqfilter(struct cdev *dev, struct knote *kn)
1996: {
1997:     struct tuntap_softc *tp = dev->si_drv1;
1998:     struct ifnet *ifp = TUN2IFP(tp);
1999:
2000:     switch(kn->kn_filter) {
2001:     case EVFILT_READ:
2002:         TUNDEBUG(ifp, "%s kqfilter: EVFILT_READ, minor = %#x\n",
2003:             ifp->if_xname, dev2unit(dev));
2004:         kn->kn_fop = &tun_read_filterops;
2005:         break;
2006:
2007:     case EVFILT_WRITE:
2008:         TUNDEBUG(ifp, "%s kqfilter: EVFILT_WRITE, minor = %#x\n",
2009:             ifp->if_xname, dev2unit(dev));
2010:         kn->kn_fop = &tun_write_filterops;
2011:         break;
2012:
2013:     default:
2014:         return (EINVAL);
2015:     }
2016:
2017:     kn->kn_hook = tp;
2018:     knlist_add(&tp->tun_rsel.si_note, kn, 0);
2019:
2020:     return (0);
2021: }

```

The `tunpoll` function implements support for `poll(2)` and `select(2)`, which allow applications to monitor multiple file descriptors for I/O readiness:

```

static int
tunpoll(struct cdev *dev, int events, struct thread *td)
{
    struct tuntap_softc *tp = dev->si_drv1;
    struct ifnet *ifp = TUN2IFP(tp);

```

```
int revents = 0;
```

The function receives:

- `dev` - the character device being polled
- `events` - bitmask of events the application wants to monitor
- `td` - the calling thread context

The return value `revents` indicates which requested events are currently ready. The function builds this bitmask by checking actual device conditions.

6.14.3.11.1 Event Notification Mechanisms: `tunpoll` and `tunkqfilter` Efficient I/O multiplexing is essential for applications managing multiple tun/tap devices or integrating tunnel I/O with other event sources. FreeBSD provides two interfaces for this: the traditional `poll(2)`/`select(2)` system calls and the more scalable `kqueue(2)` mechanism. The `tunpoll` and `tunkqfilter` functions implement these interfaces, allowing applications to wait efficiently for readable or writable conditions without busy-polling.

```
if (events & (POLLIN | POLLRDNORM)) {
    IFQ_LOCK(&ifp->if_snd);
    if (!IFQ_IS_EMPTY(&ifp->if_snd)) {
        TUNDEBUG(ifp, "tunpoll q=%d\n", ifp->if_snd.ifq_len);
        revents |= events & (POLLIN | POLLRDNORM);
    } else {
        TUNDEBUG(ifp, "tunpoll waiting\n");
        selrecord(td, &tp->tun_rsel);
    }
    IFQ_UNLOCK(&ifp->if_snd);
}
```

6.14.3.11.2 Read Readiness When the application requests read events (`POLLIN` or `POLLRDNORM`, which are synonymous for devices):

Queue check: The send queue lock is acquired, and `IFQ_IS_EMPTY` tests whether packets await reading. If packets are present:

- The requested read events are added to `revents`
- The application will be notified that `read(2)` can proceed without blocking

Registration for notification: If the queue is empty:

- `selrecord` registers this thread's interest in the device becoming readable
- The thread's context is added to `tp->tun_rsel`, a per-device selection list
- When packets arrive later (in `tunstart` or `tunstart_l2`), the code calls `selwakeup(&tp->tun_rsel)` to notify all registered threads

The `selrecord` mechanism is the key to efficient waiting. Instead of the application repeatedly polling, the kernel maintains a list of interested threads and wakes them when conditions change. This pattern appears throughout the FreeBSD kernel for any device supporting `poll(2)`.

The send queue lock protects against races where packets arrive between checking the queue and registering interest. The lock ensures atomicity: if the queue is empty during the check, registration completes before any packet arrival can call `selwakeup`.

```
revents |= events & (POLLOUT | POLLWRNORM);
```

6.14.3.11.3 Write Readiness Writes are always ready for tun/tap devices. The device has no internal buffering that could fill, `write(2)` either succeeds immediately (allocating an mbuf and dispatching to the network stack) or fails immediately (if mbuf allocation fails). There's no condition where writing would block waiting for buffer space to become available.

This unconditional write readiness is common for network devices. Unlike pipes or sockets with limited buffer space, tun/tap devices accept writes as fast as the application can generate them, relying on the mbuf allocator's dynamic memory management.

6.14.3.11.4 Kqueue Interface: `tunkqfilter` The `tunkqfilter` function implements support for `kqueue(2)`, FreeBSD's scalable event notification mechanism. Kqueue offers several advantages over `poll(2)`:

- Edge-triggered semantics (notifications only on state changes)
- Better performance with thousands of file descriptors
- User data can be attached to events
- More flexible event types (not just read/write)

```
static int
tunkqfilter(struct cdev *dev, struct knote *kn)
{
    struct tuntap_softc *tp = dev->si_drv1;
    struct ifnet *ifp = TUN2IFP(tp);
```

The function receives a `knote` (kernel note) structure representing the event registration. The `knote` persists across multiple event deliveries, unlike `poll(2)` which requires re-registration on every call.

```
switch(kn->kn_filter) {
case EVFILT_READ:
    TUNDEBUG(ifp, "%s kqfilter: EVFILT_READ, minor = %#x\n",
        ifp->if_xname, dev2unit(dev));
    kn->kn_fop = &tun_read_filterops;
```



```

    break;

case EVFILT_WRITE:
    TUNDEBLOG(ifp, "%s kqfilter: EVFILT_WRITE, minor = %#x\n",
        ifp->if_xname, dev2unit(dev));
    kn->kn_fop = &tun_write_filterops;
    break;

default:
    return (EINVAL);
}

```

6.14.3.11.5 Filter Type Validation The application specifies which event type to monitor via `kn->kn_filter`:

- `EVFILT_READ` - monitor for readable condition
- `EVFILT_WRITE` - monitor for writable condition

For each filter type, the code assigns a function table (`kn_fop`) that implements the filter's semantics. These tables were defined earlier in the source:

```

static const struct filterops tun_read_filterops = {
    .f_isfd = 1,
    .f_attach = NULL,
    .f_detach = tunkqdetach,
    .f_event = tunkqread,
};

static const struct filterops tun_write_filterops = {
    .f_isfd = 1,
    .f_attach = NULL,
    .f_detach = tunkqdetach,
    .f_event = tunkqwrite,
};

```

The `filterops` structure defines callbacks:

- `f_isfd` - flag indicating this filter operates on file descriptors
- `f_attach` - called when the filter is registered (NULL here, no special setup needed)
- `f_detach` - called when the filter is removed (`tunkqdetach` cleanup)
- `f_event` - called to test event condition (`tunkqread` or `tunkqwrite`)

Unsupported filter types (like `EVFILT_SIGNAL` or `EVFILT_TIMER`) return `EINVAL`, as they don't make sense for `tun/tap` devices.

```
kn->kn_hook = tp;
knlist_add(&tp->tun_rsel.si_note, kn, 0);

return (0);
}
```

6.14.3.11.6 Registering the Event

Two steps complete registration:

Attach context: `kn->kn_hook` stores the softc pointer. This allows the filter operation functions (`tunkqread`, `tunkqwrite`) to access device state without global lookups. When the event fires, the callback receives the knote, extracts `kn_hook`, and casts it back to `tuntap_softc *`.

Add to notification list: `knlist_add` inserts the knote into the device's kernel note list (`tp->tun_rsel.si_note`). This list is shared between `poll(2)` and `kqueue(2)` infrastructure, the `si_note` field within `tun_rsel` handles `kqueue` events, while other `tun_rsel` fields handle `poll/select` events.

When packets arrive (in `tunstart` or `tunstart_12`), the code calls `KNOTE_LOCKED(&tp->tun_rsel.si_note, 0)`, which iterates the knote list and invokes each filter's `f_event` callback. If the callback returns true (readable/writable condition met), the `kqueue` subsystem delivers the event to userspace.

The third argument to `knlist_add(0)` indicates no special flags, the knote is added unconditionally without requiring specific locking state.

6.14.3.11.7 Filter Operation Callbacks

Though not shown in this fragment, the filter operations are worth understanding:

tunkqread: Called to test read readiness

```
static int
tunkqread(struct knote *kn, long hint)
{
    struct tuntap_softc *tp = kn->kn_hook;
    struct ifnet *ifp = TUN2IFP(tp);

    if ((kn->kn_data = ifp->if_snd.ifq_len) > 0) {
        return (1); // Readable
    }
    return (0); // Not readable
}
```

The callback checks the send queue length and stores it in `kn->kn_data`, making the count available to userspace via the `kevent` structure. Returning 1 signals the event should fire; returning 0 means the condition is not yet met.

tunkqwrite: Called to test write readiness

```
static int
tunkqwrite(struct knote *kn, long hint)
{
    struct tuntap_softc *tp = kn->kn_hook;
    struct ifnet *ifp = TUN2IFP(tp);

    kn->kn_data = ifp->if_mtu;
    return (1); // Always writable
}
```

Since writes are always possible, this always returns 1. The `kn_data` field is set to the interface MTU, giving userspace information about maximum write size.

tunkqdetach: Called when removing the event

```
static void
tunkqdetach(struct knote *kn)
{
    struct tuntap_softc *tp = kn->kn_hook;

    knlist_remove(&tp->tun_rsel.si_note, kn, 0);
}
```

This removes the knote from the device's notification list, ensuring no further events are delivered for this registration.

6.14.3.11.8 Comparison: Poll vs. Kqueue The two mechanisms serve similar purposes but with different characteristics:

Poll/Select: - Level-triggered: reports readiness state on every call - Requires kernel scanning of all file descriptors on each call - Simple API, widely portable - $O(n)$ complexity in number of file descriptors

Kqueue: - Edge-triggered: reports changes in readiness state - Kernel maintains active event list, only reports changes - More complex API, FreeBSD/macOS specific - $O(1)$ complexity for event delivery

For applications monitoring a single tun/tap device, the difference is negligible. For VPN concentrators or network simulators managing hundreds of virtual interfaces, kqueue's scalability advantages become significant.

6.14.3.11.9 Notification Flow When a packet arrives for transmission, the complete notification sequence:

```
Network stack routes packet to tun0
->
tunoutput() / tap_transmit() enqueues mbuf
->
tunstart() / tunstart_l2() wakes waiters:
```

```
| -> wakeup(tp) - wakes blocked read()
| -> selwakeup(&tp->tun_rsel) - wakes poll()/select()
| -> KNOTE_LOCKED(&tp->tun_rsel.si_note, 0) - delivers kqueue events
->
```

Application receives notification

```
->
```

Application calls read() to retrieve packet

This multi-mechanism notification ensures applications using any waiting strategy, blocking reads, poll/select loops, or kqueue event loops, receive prompt packet delivery notification.

6.14.3.12 Interactive Exercises for tun(4)/tap(4)

Goal: Trace both directions of data flow and map user-space operations to the exact kernel lines.

6.14.3.12.1 A) Device personalities and cloning (warm-up)

1. In the `tuntap_drivers[]` array, list the three `.d_name` values and identify which function pointers (`.d_open`, `.d_read`, `.d_write`, etc.) are assigned for each. Note: are they the same or different functions? Quote the initializer lines you used. (Tip: examine lines around 280-291 and the subsequent entries for `tap/vmnet`.)
2. In `tun_clone_create()`, find where the driver:
 - computes the final name with unit,
 - calls `clone_create()`,
 - falls back to `tun_create_device()`, and
 - calls `tuncreate()` to attach the `ifnet`.

Quote those lines and explain the sequence.

3. In `tun_create_device()`, record the mode used for the `cdev` and which field points `si_drv1` to the `softc`. Quote the lines. (Hint: look for `mda_mode` and `mda_si_drv1`.)

6.14.3.12.2 B) Interface bring-up path

1. In `tuncreate()`, point to the `if_alloc()`, `if_initname()`, and `if_attach()` calls. Why is `bpfattach()` called for L3 mode **with** `DLT_NULL` instead of `DLT_EN10MB`? Quote the lines you used.
2. In `tunopen()`, identify where link state is marked UP on open. Quote the line(s).
3. In `tunopen()`, what prevents two processes from opening the same device simultaneously? Quote the check and explain the flags involved. (Hint: look for `TUN_OPEN` and `EBUSY`.)

6.14.3.12.3 C) Read a packet from user space (kernel -> user)

1. In `tunread()`, explain the blocking and non-blocking behaviors. Which flag forces `EWouldBlock`? Where is the sleep done? Quote the lines.

2. Where is the optional virtio header copied to user space, and how is the payload then delivered? Quote those lines.
3. Where are readers woken when output arrives from the stack? Trace the wakeups in `tunstart_l2()` (or the L3 start path): `wakeup`, `selwakeuppri`, and `KNOTE`. Quote the lines.

6.14.3.12.4 D) Write a packet from user space (user -> kernel)

1. In `tunwrite()`, find the guard that silently ignores writes if the interface is down, and the check that bounds the maximum write size (MRU + headers). Quote the lines.
2. Still in `tunwrite()`, where is the user buffer turned into an mbuf? Quote the call and explain the `align` parameter for L2.
3. Follow the L3 path into `tunwrite_l3()`: where is the address family read (when `TUN_IFHEAD` is set), where is BPF tapped, and where is the netisr dispatch called? Quote those lines.
4. Follow the L2 path into `tunwrite_l2()`: where does it drop frames whose destination MAC address doesn't match the interface's MAC (unless promiscuous mode is set)? This simulates what real Ethernet hardware wouldn't deliver. Quote those lines.

6.14.3.12.5 E) Quick user-space validations (safe experiments) These checks assume you created a `tun0` (L3) or `tap0` (L2) and brought it up in a private VM.

```
# L3: read a packet the kernel queued for us
% ifconfig tun0 10.0.0.1/24 up
% ( ping -c1 10.0.0.2 >/dev/null & ) &
% dd if=/dev/tun0 bs=4096 count=1 2>/dev/null | hexdump -C | head -n2
# Expected: You should see an ICMP echo request (type 8)
# with destination IP 10.0.0.2 starting around offset 0x14

# L3: inject an IPv4 echo request (requires crafting a full frame)
# (later in the book we'll show a tiny C sender using write())
```

For each command you run, point to the exact lines in `tunread()` or `tunwrite_l3()` that explain the behavior you observe.

6.14.3.13 Stretch (thought experiments)

1. If `tunwrite()` returned `EIO` when the interface is down, instead of ignoring writes, how would tools that rely on blind writes behave? Point to the current “ignore if down” line and explain the design choice.
2. Suppose `tunstart_l2()` called `wakeup(tp)` but **not** `selwakeuppri(&tp->tun_rsel, ...)`. What would happen to an application using `poll(2)` to wait for packets? Would blocking `read(2)` still work? Point to both notification mechanisms and explain why each is necessary.

6.14.3.14 Bridging to the next tour

The `if_tuntap` driver demonstrates how character devices and network interfaces integrate, with userspace acting as the “hardware” endpoint. Our next driver explores fundamentally different territory: **uart_bus_pci** shows how real hardware devices are discovered and bound to kernel drivers through FreeBSD’s layered bus architecture.

This shift from character device operations to bus attachment represents a critical architectural pattern: the separation between **bus-specific glue code** and **device-agnostic core functionality**. The `uart_bus_pci` driver is intentionally minimal, with under 300 lines of code, focusing solely on device identification (matching PCI vendor/device IDs), resource negotiation (claiming I/O ports and interrupts), and handoff to the generic UART subsystem via `uart_bus_probe()` and `uart_bus_attach()`.

6.14.4 Tour 4 - The PCI glue: `uart(4)`

Open the file:

```
% cd /usr/src/sys/dev/uart
% less uart_bus_pci.c
```

This file is the **PCI “bus glue”** for the generic UART core. It matches hardware via a PCI ID table, picks a UART **class**, calls the **shared uart bus probe/attach**, and adds a tiny bit of bus-specific logic (MSI preference, unique console matching). The actual UART register shuffling lives in the common UART code; this file is about **matching and wiring**.

6.14.4.1 1) Method table + driver object (what Newbus calls)

```
52: static device_method_t uart_pci_methods[] = {
53:     /* Device interface */
54:     DEVMETHOD(device_probe,      uart_pci_probe),
55:     DEVMETHOD(device_attach,     uart_pci_attach),
56:     DEVMETHOD(device_detach,     uart_pci_detach),
57:     DEVMETHOD(device_resume,     uart_bus_resume),
58:     DEVMETHOD_END
59: };
60:
61: static driver_t uart_pci_driver = {
62:     uart_driver_name,
63:     uart_pci_methods,
64:     sizeof(struct uart_softc),
65: };
```

Map this mentally to the Newbus lifecycle: `probe` -> `attach` -> `detach` (+ `resume`).

6.14.4.1.1 Device Methods and Driver Structure FreeBSD’s device driver framework uses an object-oriented approach where drivers declare which operations they support through method tables. The `uart_pci_methods` array and `uart_pci_driver` structure establish this driver’s interface to the kernel’s device management subsystem.

```
static device_method_t uart_pci_methods[] = {  
    /* Device interface */  
    DEVMETHOD(device_probe,      uart_pci_probe),  
    DEVMETHOD(device_attach,     uart_pci_attach),  
    DEVMETHOD(device_detach,     uart_pci_detach),  
    DEVMETHOD(device_resume,     uart_bus_resume),  
    DEVMETHOD_END  
};
```

6.14.4.1.2 The Device Method Table The `device_method_t` array maps generic device operations to driver-specific implementations. Each `DEVMETHOD` entry binds a method identifier to a function pointer:

device_probe -> `uart_pci_probe`: Called by the PCI bus driver during device enumeration to ask “can you drive this device?” The function examines the device’s PCI vendor and device IDs, returning a priority value indicating how well it matches. Lower values mean better matches; returning `ENXIO` means “not my device.”

device_attach -> `uart_pci_attach`: Called after a successful probe to initialize the device. This function allocates resources (I/O ports, interrupts), configures the hardware, and makes the device operational. If attachment fails, the driver should release any allocated resources.

device_detach -> `uart_pci_detach`: Called when the device is being removed from the system (hot-unplug, driver unload, or system shutdown). Must release all resources claimed during attach and ensure the hardware is left in a safe state.

device_resume -> `uart_bus_resume`: Called when the system resumes from a suspend state. Note this points to `uart_bus_resume`, not a PCI-specific function, the generic UART layer handles power management uniformly across all bus types.

DEVMETHOD_END: A sentinel marking the array’s end. The kernel iterates this table until reaching this terminator.

```
static driver_t uart_pci_driver = {  
    uart_driver_name,  
    uart_pci_methods,  
    sizeof(struct uart_softc),  
};
```

6.14.4.1.3 The Driver Declaration The `driver_t` structure packages the method table with metadata:

uart_driver_name: A string identifying this driver, typically “uart”. This name appears in kernel messages, device tree output, and administrative tools. The name is defined in the generic uart code and shared across all bus attachments (PCI, ISA, ACPI), ensuring consistent device naming regardless of how the UART was discovered.

uart_pci_methods: Pointer to the method table defined above. When the kernel needs to perform an operation on a uart_pci device, it looks up the appropriate method in this table and calls the corresponding function.

sizeof(struct uart_softc): The size of the driver’s per-device state structure. The kernel allocates this much memory when creating a device instance, accessible via `device_get_softc()`. Importantly, this uses `uart_softc` from the generic UART layer, not a PCI-specific structure, the core UART state is bus-agnostic.

6.14.4.1.4 Architectural Significance This simple structure embodies FreeBSD’s layered driver model. The method table contains four functions:

- Two are PCI-specific (`uart_pci_probe`, `uart_pci_attach`, `uart_pci_detach`)
- One is bus-agnostic (`uart_bus_resume`)

The PCI-specific functions handle only bus-related concerns: matching device IDs, claiming PCI resources, and managing MSI interrupts. All UART-specific logic, baud rate configuration, FIFO management, character I/O, lives in the generic `uart_bus.c` code that these functions call.

This separation means the same UART hardware logic works whether the device appears on the PCI bus, the ISA bus, or as an ACPI-enumerated device. Only the probe/attach glue changes. This pattern, thin bus-specific wrappers around substantial generic cores, reduces code duplication and simplifies porting to new bus types or architectures.

The method table mechanism also enables runtime polymorphism. If a UART appears on different buses (a 16550 on both PCI and ISA, for example), the kernel loads different driver modules (`uart_pci`, `uart_isa`), each with its own method table, but both share the underlying `uart_softc` structure and call the same generic functions for actual device operation.

6.14.4.2 2) Local structs + flags we’ll use

```
67: struct pci_id {
68:     uint16_t    vendor;
69:     uint16_t    device;
70:     uint16_t    subven;
71:     uint16_t    subdev;
72:     const char  *desc;
73:     int         rid;
74:     int         rclk;
75:     int         regshft;
76: };
77:
78: struct pci_unique_id {
79:     uint16_t    vendor;
```



```

80:     uint16_t    device;
81: };
83: #define PCI_NO_MSI 0x40000000
84: #define PCI_RID_MASK    0x0000ffff

```

What matters later: `rid` (which BAR/IRQ to use), optional `rclock` and `regshft`, and the `PCI_NO_MSI` hint.

6.14.4.2.1 Device Identification Structures Hardware drivers must identify which specific devices they can manage. For PCI devices, this identification relies on vendor and device ID codes burned into the hardware’s configuration space. The `pci_id` and `pci_unique_id` structures encode this matching logic along with device-specific configuration parameters.

```

struct pci_id {
    uint16_t    vendor;
    uint16_t    device;
    uint16_t    subven;
    uint16_t    subdev;
    const char  *desc;
    int         rid;
    int         rclk;
    int         regshft;
};

```

6.14.4.2.2 The Primary Identification Structure Each `pci_id` entry describes one UART variant and how to configure it:

vendor and device: The primary identification pair. Every PCI device has a 16-bit vendor ID (assigned by the PCI Special Interest Group) and a 16-bit device ID (assigned by the vendor). For example, Intel is vendor `0x8086`, and their AMT Serial-over-LAN controller is device `0x108f`. These IDs are read from the device’s configuration space at bus enumeration time.

subven and subdev: Secondary identification for OEM customization. Many manufacturers build cards using reference designs from chipset vendors, then assign their own subsystem vendor and device IDs. A value of `0xffff` in these fields acts as a wildcard, meaning “match any subsystem IDs.” This allows matching either specific OEM variants or entire chipset families.

The four-level matching hierarchy enables precise identification:

1. Match only specific OEM cards: all four IDs must match exactly
2. Match all cards using a chipset: vendor/device match, subven/subdev are `0xffff`
3. Match specific OEM customization: vendor/device plus exact subven/subdev

desc: Human-readable device description displayed in boot messages and `dmesg` output. Examples: “Intel AMT - SOL” or “Oxford Semiconductor OXCB950 Cardbus 16950 UART”. This string helps administrators identify which physical device corresponds to which `/dev/cuaU*` entry.

rid: Resource ID specifying which PCI Base Address Register (BAR) contains the UART's registers. PCI devices can have up to six BARs (numbered 0x10, 0x14, 0x18, 0x1c, 0x20, 0x24). Most UARTs use BAR 0 (0x10), but some multi-function cards place the UART at alternate BARs. This field may also encode flags via the high bits.

clk: Reference clock frequency in Hz. The UART's baud rate generator divides this clock to produce serial bit timing. Standard PC UARTs use 1843200 Hz (1.8432 MHz), but embedded UARTs and specialized cards often use different frequencies. Some Intel devices use 24x the standard clock for high-speed operation. An incorrect `clk` causes garbled serial communication due to baud rate mismatch.

regshft: Register address shift value. Most UARTs place consecutive registers at consecutive byte addresses (shift = 0), but some embed the UART in larger register spaces with registers at every 4th byte (shift = 2) or other intervals. The driver shifts register offsets by this amount when accessing hardware. This accommodates SoC designs where the UART shares address space with other peripherals.

```
struct pci_unique_id {
    uint16_t    vendor;
    uint16_t    device;
};
```

6.14.4.2.3 The Simplified Identification Structure This smaller structure identifies devices guaranteed to exist only once per system. Certain hardware, particularly server management controllers and embedded SoC UARTs, is designed as singleton devices. For these, vendor and device IDs alone suffice for matching against system consoles, without needing subsystem IDs or configuration parameters.

The distinction matters for console matching: if a UART serves as the system console (configured in firmware or boot loader), the kernel must identify which enumerated device corresponds to the pre-configured console. For unique devices, a simple vendor/device match provides certainty.

```
#define PCI_NO_MSI        0x40000000
#define PCI_RID_MASK      0x0000ffff
```

6.14.4.2.4 Resource ID Encoding The `rid` field serves double duty through bit packing:

PCI_RID_MASK (0x0000ffff): The lower 16 bits contain the actual BAR number (0x10, 0x14, etc.). Masking with this value extracts the resource ID for bus allocation functions.

PCI_NO_MSI (0x40000000): The high bit flags devices with broken or unreliable Message Signaled Interrupt (MSI) support. Some UART implementations don't correctly implement MSI, causing interrupt delivery failures or system hangs. This flag tells the attach function to use traditional line-based interrupts instead of attempting MSI allocation.

This encoding scheme avoids enlarging the `pci_id` structure with an additional boolean field. Since BAR numbers only use the low byte, the high bits are available for flags. The driver extracts the actual RID with `id->rid & PCI_RID_MASK` and checks MSI capability with `(id->rid & PCI_NO_MSI) == 0`.

6.14.4.2.5 Purpose in Device Matching These structures populate a large static array (examined in the next fragment) that the probe function searches during device enumeration. When the PCI bus driver discovers a device with class “Simple Communications” (modems and UARTs), it calls this driver’s probe function. The probe function walks the array comparing the device’s IDs against each entry, looking for a match. Upon finding one, it uses the associated desc, rid, rclk, and regshft values to configure the device correctly.

This table-driven approach simplifies adding new hardware support: most new UART variants require only adding a table entry with the correct IDs and clock frequency, without modifying code.

6.14.4.3 3) The PCI ID table (ns8250-ish parts)

Below is the **contiguous** table used to match vendor/device(/subvendor/subdevice), plus per-device hints (RID, reference clock, register shift). The 0xffff row terminates the list.

```

86: static const struct pci_id pci_ns8250_ids[] = {
87: { 0x1028, 0x0008, 0xffff, 0, "Dell Remote Access Card III", 0x14,
88:   128 * DEFAULT_RCLK },
89: { 0x1028, 0x0012, 0xffff, 0, "Dell RAC 4 Daughter Card Virtual UART", 0x14,
90:   128 * DEFAULT_RCLK },
91: { 0x1033, 0x0074, 0x1033, 0x8014, "NEC RCV56ACF 56k Voice Modem", 0x10 },
92: { 0x1033, 0x007d, 0x1033, 0x8012, "NEC RS232C", 0x10 },
93: { 0x103c, 0x1048, 0x103c, 0x1227, "HP Diva Serial [GSP] UART - Powerbar SP2",
94:   0x10 },
95: { 0x103c, 0x1048, 0x103c, 0x1301, "HP Diva RMP3", 0x14 },
96: { 0x103c, 0x1290, 0xffff, 0, "HP Auxiliary Diva Serial Port", 0x18 },
97: { 0x103c, 0x3301, 0xffff, 0, "HP iLO serial port", 0x10 },
98: { 0x11c1, 0x0480, 0xffff, 0, "Agere Systems Venus Modem (V90, 56KFlex)", 0x14 },
99: { 0x115d, 0x0103, 0xffff, 0, "Xircom Cardbus Ethernet + 56k Modem", 0x10 },
100: { 0x125b, 0x9100, 0xa000, 0x1000,
101:   "ASIX AX99100 PCIe 1/2/3/4-port RS-232/422/485", 0x10 },
102: { 0x1282, 0x6585, 0xffff, 0, "Davicom 56PDV PCI Modem", 0x10 },
103: { 0x12b9, 0x1008, 0xffff, 0, "3Com 56K FaxModem Model 5610", 0x10 },
104: { 0x131f, 0x1000, 0xffff, 0, "Siig CyberSerial (1-port) 16550", 0x18 },
105: { 0x131f, 0x1001, 0xffff, 0, "Siig CyberSerial (1-port) 16650", 0x18 },
106: { 0x131f, 0x1002, 0xffff, 0, "Siig CyberSerial (1-port) 16850", 0x18 },
107: { 0x131f, 0x2000, 0xffff, 0, "Siig CyberSerial (1-port) 16550", 0x10 },
108: { 0x131f, 0x2001, 0xffff, 0, "Siig CyberSerial (1-port) 16650", 0x10 },
109: { 0x131f, 0x2002, 0xffff, 0, "Siig CyberSerial (1-port) 16850", 0x10 },
110: { 0x135a, 0x0a61, 0xffff, 0, "Brainboxes UC-324", 0x18 },
111: { 0x135a, 0x0aa1, 0xffff, 0, "Brainboxes UC-246", 0x18 },
112: { 0x135a, 0x0aa2, 0xffff, 0, "Brainboxes UC-246", 0x18 },
113: { 0x135a, 0x0d60, 0xffff, 0, "Intashield IS-100", 0x18 },
114: { 0x135a, 0x0da0, 0xffff, 0, "Intashield IS-300", 0x18 },
115: { 0x135a, 0x4000, 0xffff, 0, "Brainboxes PX-420", 0x10 },
116: { 0x135a, 0x4001, 0xffff, 0, "Brainboxes PX-431", 0x10 },

```

```

117: { 0x135a, 0x4002, 0xffff, 0, "Brainboxes PX-820", 0x10 },
118: { 0x135a, 0x4003, 0xffff, 0, "Brainboxes PX-831", 0x10 },
119: { 0x135a, 0x4004, 0xffff, 0, "Brainboxes PX-246", 0x10 },
120: { 0x135a, 0x4005, 0xffff, 0, "Brainboxes PX-101", 0x10 },
121: { 0x135a, 0x4006, 0xffff, 0, "Brainboxes PX-257", 0x10 },
122: { 0x135a, 0x4008, 0xffff, 0, "Brainboxes PX-846", 0x10 },
123: { 0x135a, 0x4009, 0xffff, 0, "Brainboxes PX-857", 0x10 },
124: { 0x135c, 0x0190, 0xffff, 0, "Quatech SSCLP-100", 0x18 },
125: { 0x135c, 0x01c0, 0xffff, 0, "Quatech SSCLP-200/300", 0x18 },
126: { 0x135e, 0x7101, 0xffff, 0, "Sealevel Systems Single Port RS-232/422/485/530",
127:     0x18 },
128: { 0x1407, 0x0110, 0xffff, 0, "Lava Computer mfg DSerial-PCI Port A", 0x10 },
129: { 0x1407, 0x0111, 0xffff, 0, "Lava Computer mfg DSerial-PCI Port B", 0x10 },
130: { 0x1407, 0x0510, 0xffff, 0, "Lava SP Serial 550 PCI", 0x10 },
131: { 0x1409, 0x7168, 0x1409, 0x4025, "Timedia Technology Serial Port", 0x10,
132:     8 * DEFAULT_RCLK },
133: { 0x1409, 0x7168, 0x1409, 0x4027, "Timedia Technology Serial Port", 0x10,
134:     8 * DEFAULT_RCLK },
135: { 0x1409, 0x7168, 0x1409, 0x4028, "Timedia Technology Serial Port", 0x10,
136:     8 * DEFAULT_RCLK },
137: { 0x1409, 0x7168, 0x1409, 0x5025, "Timedia Technology Serial Port", 0x10,
138:     8 * DEFAULT_RCLK },
139: { 0x1409, 0x7168, 0x1409, 0x5027, "Timedia Technology Serial Port", 0x10,
140:     8 * DEFAULT_RCLK },
141: { 0x1415, 0x950b, 0xffff, 0, "Oxford Semiconductor OXCB950 Cardbus 16950 UART",
142:     0x10, 16384000 },
143: { 0x1415, 0xc120, 0xffff, 0, "Oxford Semiconductor OXPCIE952 PCIe 16950 UART",
144:     0x10 },
145: { 0x14e4, 0x160a, 0xffff, 0, "Broadcom TruManage UART", 0x10,
146:     128 * DEFAULT_RCLK, 2 },
147: { 0x14e4, 0x4344, 0xffff, 0, "Sony Ericsson GC89 PC Card", 0x10 },
148: { 0x151f, 0x0000, 0xffff, 0, "TOPIC Semiconductor TP560 56k modem", 0x10 },
149: { 0x1d0f, 0x8250, 0x0000, 0, "Amazon PCI serial device", 0x10 },
150: { 0x1d0f, 0x8250, 0x1d0f, 0, "Amazon PCI serial device", 0x10 },
151: { 0x1fd4, 0x1999, 0x1fd4, 0x0001, "Sunix SER5xxxx Serial Port", 0x10,
152:     8 * DEFAULT_RCLK },
153: { 0x8086, 0x0c5f, 0xffff, 0, "Atom Processor S1200 UART",
154:     0x10 | PCI_NO_MSI },
155: { 0x8086, 0x0f0a, 0xffff, 0, "Intel ValleyView LPI01 HSUART#1", 0x10,
156:     24 * DEFAULT_RCLK, 2 },
157: { 0x8086, 0x0f0c, 0xffff, 0, "Intel ValleyView LPI01 HSUART#2", 0x10,
158:     24 * DEFAULT_RCLK, 2 },
159: { 0x8086, 0x108f, 0xffff, 0, "Intel AMT - SOL", 0x10 },
160: { 0x8086, 0x19d8, 0xffff, 0, "Intel Denverton UART", 0x10 },

```

```

161: { 0x8086, 0x1c3d, 0xffff, 0, "Intel AMT - KT Controller", 0x10 },
162: { 0x8086, 0x1d3d, 0xffff, 0, "Intel C600/X79 Series Chipset KT Controller",
163:   0x10 },
164: { 0x8086, 0x1e3d, 0xffff, 0, "Intel Panther Point KT Controller", 0x10 },
165: { 0x8086, 0x228a, 0xffff, 0, "Intel Cherryview SIO HSUART#1", 0x10,
166:   24 * DEFAULT_RCLK, 2 },
167: { 0x8086, 0x228c, 0xffff, 0, "Intel Cherryview SIO HSUART#2", 0x10,
168:   24 * DEFAULT_RCLK, 2 },
169: { 0x8086, 0x2a07, 0xffff, 0, "Intel AMT - PM965/GM965 KT Controller", 0x10 },
170: { 0x8086, 0x2a47, 0xffff, 0, "Mobile 4 Series Chipset KT Controller", 0x10 },
171: { 0x8086, 0x2e17, 0xffff, 0, "4 Series Chipset Serial KT Controller", 0x10 },
172: { 0x8086, 0x31bc, 0xffff, 0, "Intel Gemini Lake SIO/LPSS UART 0", 0x10,
173:   24 * DEFAULT_RCLK, 2 },
174: { 0x8086, 0x31be, 0xffff, 0, "Intel Gemini Lake SIO/LPSS UART 1", 0x10,
175:   24 * DEFAULT_RCLK, 2 },
176: { 0x8086, 0x31c0, 0xffff, 0, "Intel Gemini Lake SIO/LPSS UART 2", 0x10,
177:   24 * DEFAULT_RCLK, 2 },
178: { 0x8086, 0x31ee, 0xffff, 0, "Intel Gemini Lake SIO/LPSS UART 3", 0x10,
179:   24 * DEFAULT_RCLK, 2 },
180: { 0x8086, 0x3b67, 0xffff, 0, "5 Series/3400 Series Chipset KT Controller",
181:   0x10 },
182: { 0x8086, 0x5abc, 0xffff, 0, "Intel Apollo Lake SIO/LPSS UART 0", 0x10,
183:   24 * DEFAULT_RCLK, 2 },
184: { 0x8086, 0x5abe, 0xffff, 0, "Intel Apollo Lake SIO/LPSS UART 1", 0x10,
185:   24 * DEFAULT_RCLK, 2 },
186: { 0x8086, 0x5ac0, 0xffff, 0, "Intel Apollo Lake SIO/LPSS UART 2", 0x10,
187:   24 * DEFAULT_RCLK, 2 },
188: { 0x8086, 0x5aee, 0xffff, 0, "Intel Apollo Lake SIO/LPSS UART 3", 0x10,
189:   24 * DEFAULT_RCLK, 2 },
190: { 0x8086, 0x8811, 0xffff, 0, "Intel EG20T Serial Port 0", 0x10 },
191: { 0x8086, 0x8812, 0xffff, 0, "Intel EG20T Serial Port 1", 0x10 },
192: { 0x8086, 0x8813, 0xffff, 0, "Intel EG20T Serial Port 2", 0x10 },
193: { 0x8086, 0x8814, 0xffff, 0, "Intel EG20T Serial Port 3", 0x10 },
194: { 0x8086, 0x8c3d, 0xffff, 0, "Intel Lynx Point KT Controller", 0x10 },
195: { 0x8086, 0x8cbd, 0xffff, 0, "Intel Wildcat Point KT Controller", 0x10 },
196: { 0x8086, 0x8d3d, 0xffff, 0,
197:   "Intel Corporation C610/X99 series chipset KT Controller", 0x10 },
198: { 0x8086, 0x9c3d, 0xffff, 0, "Intel Lynx Point-LP HECI KT", 0x10 },
199: { 0x8086, 0xa13d, 0xffff, 0,
200:   "100 Series/C230 Series Chipset Family KT Redirection",
201:   0x10 | PCI_NO_MSI },
202: { 0x9710, 0x9820, 0x1000, 1, "NetMos NM9820 Serial Port", 0x10 },
203: { 0x9710, 0x9835, 0x1000, 1, "NetMos NM9835 Serial Port", 0x10 },
204: { 0x9710, 0x9865, 0xa000, 0x1000, "NetMos NM9865 Serial Port", 0x10 },

```

```

205: { 0x9710, 0x9900, 0xa000, 0x1000,
206:     "MosChip MCS9900 PCIe to Peripheral Controller", 0x10 },
207: { 0x9710, 0x9901, 0xa000, 0x1000,
208:     "MosChip MCS9901 PCIe to Peripheral Controller", 0x10 },
209: { 0x9710, 0x9904, 0xa000, 0x1000,
210:     "MosChip MCS9904 PCIe to Peripheral Controller", 0x10 },
211: { 0x9710, 0x9922, 0xa000, 0x1000,
212:     "MosChip MCS9922 PCIe to Peripheral Controller", 0x10 },
213: { 0xdeaf, 0x9051, 0xffff, 0, "Middle Digital PC Weasel Serial Port", 0x10 },
214: { 0xffff, 0, 0xffff, 0, NULL, 0, 0}
215: };

```

Notice per-device **RID** (which BAR/IRQ), frequency hints (*rclk* like 24 * *DEFAULT_RCLK*), and optional *regshft*.

6.14.4.3.1 The Device Identification Table The `pci_ns8250_ids` array is the heart of the driver's device recognition logic. This table lists every known PCI UART variant compatible with the NS8250/16550 register interface, along with the configuration parameters needed to operate each correctly. During system boot, the PCI bus driver walks all discovered devices and calls this driver's probe function for potential matches; the probe function searches this table to determine compatibility.

```
static const struct pci_id pci_ns8250_ids[] = {
```

6.14.4.3.2 Table Structure and Purpose The array name, `pci_ns8250_ids`, reflects that all listed devices implement the National Semiconductor 8250 (or compatible 16450/16550/16650/16750/16850/16950) register interface. Despite coming from dozens of manufacturers, these UARTs share a common programming model dating back to the original IBM PC's serial port design. This compatibility allows a single driver to support disparate hardware through a unified register abstraction.

The `static const` qualifiers indicate this data is read-only and internal to this compilation unit. The table resides in read-only memory, preventing accidental modification and allowing the kernel to share one copy across all CPU cores.

6.14.4.3.3 Entry Analysis: Understanding the Patterns Examining representative entries reveals the matching hierarchy and configuration diversity:

Simple wildcard match (Intel AMT SOL entry in `pci_ns8250_ids`):

```
{ 0x8086, 0x108f, 0xffff, 0, "Intel AMT - SOL", 0x10 },
```

- Vendor 0x8086 (Intel), device 0x108f (AMT Serial-over-LAN)
- Subsystem IDs 0xffff (wildcard) match all OEM variants

- Description for boot messages and device listings
- RID 0x10 (BAR0), standard clock rate (implied DEFAULT_RCLK), no register shift

This pattern matches Intel's AMT SOL controller regardless of which motherboard manufacturer integrated it.

OEM-specific match (adjacent HP Diva entries in `pci_ns8250_ids`):

```
{ 0x103c, 0x1048, 0x103c, 0x1227, "HP Diva Serial [GSP] UART - Powerbar SP2", 0x10 },
{ 0x103c, 0x1048, 0x103c, 0x1301, "HP Diva RMP3", 0x14 },
```

- Same chipset (HP vendor 0x103c, device 0x1048) used in multiple products
- Different subsystem device IDs (0x1227, 0x1301) distinguish variants
- Different BARs (0x10 vs 0x14) indicate the UART appears at different addresses in each card's configuration space

This illustrates how one chipset spawns multiple table entries when OEMs configure it differently across product lines.

Non-standard clock frequency (Dell Remote Access Card III entry in `pci_ns8250_ids`):

```
{ 0x1028, 0x0008, 0xffff, 0, "Dell Remote Access Card III", 0x14,
  128 * DEFAULT_RCLK },
```

- Dell (0x1028) RAC III uses 128x the standard 1.8432 MHz clock = 235.9296 MHz
- This extremely high frequency supports baud rates far beyond standard serial ports
- Without the correct `rclk` value, all baud rate calculations would be wrong by 128x, producing gibberish

Server management cards often use high clocks to support fast console redirection over network links.

Register address shifting (Intel ValleyView LPIO1 HSUART entry in `pci_ns8250_ids`):

```
{ 0x8086, 0x0f0a, 0xffff, 0, "Intel ValleyView LPIO1 HSUART#1", 0x10,
  24 * DEFAULT_RCLK, 2 },
```

- Intel SoC UART with 24x standard clock for high-speed operation
- `regshift = 2` means registers appear at 4-byte intervals (addresses 0, 4, 8, 12, ...)
- The generic UART code shifts all register offsets left by 2 bits: `address << 2`

This accommodates SoC designs where the UART shares a large memory-mapped region with other peripherals, often with registers aligned to 32-bit boundaries for bus efficiency.

MSI incompatibility (Atom Processor S1200 entry in `pci_ns8250_ids`, combined with the `PCI_NO_MSI` handling in `uart_pci_attach`):

```
{ 0x8086, 0x0c5f, 0xffff, 0, "Atom Processor S1200 UART",
  0x10 | PCI_NO_MSI },
```

- The `PCI_NO_MSI` flag in the RID field indicates broken MSI support

- The attach function will detect this flag and use legacy line-based interrupts instead
- These devices claim MSI capability in their PCI configuration space but don't deliver interrupts correctly

Such quirks typically arise from silicon errata or incomplete MSI implementation in integrated peripherals.

Multiple subsystem variants (Timedia Technology entries in `pci_ns8250_ids`):

```
{ 0x1409, 0x7168, 0x1409, 0x4025, "Timedia Technology Serial Port", 0x10,
  8 * DEFAULT_RCLK },
{ 0x1409, 0x7168, 0x1409, 0x4027, "Timedia Technology Serial Port", 0x10,
  8 * DEFAULT_RCLK },
```

- Same base chipset (vendor 0x1409, device 0x7168) used across a product family
- Each subsystem device ID represents a different card model or port count variant
- All share the same clock (8x standard) and BAR configuration
- The probe function matches the first entry with compatible subsystem IDs

This repetition is unavoidable when a manufacturer uses one chipset across many SKUs, each with unique subsystem identification.

```
{ 0xffff, 0, 0xffff, 0, NULL, 0, 0 }
```

6.14.4.3.4 The Sentinel Entry The final entry marks the table's end. The matching function walks entries until finding `vendor == 0xffff`, indicating no more devices to check. Using 0xffff (an invalid vendor ID; no such vendor exists) ensures the sentinel can't accidentally match real hardware.

6.14.4.3.5 Table Maintenance and Evolution This table grows continuously as new UART hardware appears. Adding support for a new device typically requires:

1. Determining the vendor/device/subsystem IDs (via `pciconf -lv` on FreeBSD)
2. Finding the correct BAR where the UART registers reside (often documented, sometimes discovered via trial)
3. Identifying the clock frequency (from datasheets or experimentation)
4. Testing that standard NS8250 register access works

Most entries use default values (standard clock, no shift, BAR0), requiring only IDs and a description. Complex entries like those with unusual clocks or MSI quirks often emerge from bug reports or hardware donations to developers.

The table-driven approach keeps the code maintainable: adding a new UART rarely requires code changes, just a new table entry. This is critical for a subsystem supporting dozens of manufacturers and hundreds of product variants accumulated over decades of PC hardware evolution.

6.14.4.3.6 Architectural Note This table documents only NS8250-compatible UARTs. Non-compatible serial controllers (like USB serial adapters, IEEE 1394 serial, or proprietary designs) use different drivers. The probe function verifies NS8250 compatibility before accepting a device, ensuring this table's assumptions hold for all matched hardware.

6.14.4.4 4) Matching function: from PCI IDs to a hit

```

218: const static struct pci_id *
219: uart_pci_match(device_t dev, const struct pci_id *id)
220: {
221:     uint16_t device, subdev, subven, vendor;
222:
223:     vendor = pci_get_vendor(dev);
224:     device = pci_get_device(dev);
225:     while (id->vendor != 0xffff &&
226:         (id->vendor != vendor || id->device != device))
227:         id++;
228:     if (id->vendor == 0xffff)
229:         return (NULL);
230:     if (id->subven == 0xffff)
231:         return (id);
232:     subven = pci_get_subvendor(dev);
233:     subdev = pci_get_subdevice(dev);
234:     while (id->vendor == vendor && id->device == device &&
235:         (id->subven != subven || id->subdev != subdev))
236:         id++;
237:     return ((id->vendor == vendor && id->device == device) ? id : NULL);

```

First match vendor/device; if the entry has specific sub-IDs, check those too; otherwise accept the wildcard.

6.14.4.4.1 Device Matching Logic: `uart_pci_match` The `uart_pci_match` function implements a two-phase search algorithm that efficiently matches PCI devices against the identification table while respecting the vendor/device/subsystem hierarchy. This function is the core of device recognition, called during probe to determine if a discovered PCI device is a supported UART.

```

const static struct pci_id *
uart_pci_match(device_t dev, const struct pci_id *id)
{
    uint16_t device, subdev, subven, vendor;

```

6.14.4.4.2 Function Signature and Context The function accepts a `device_t` representing the PCI device being probed and a pointer to the start of the identification table. It returns either a pointer to the matching `pci_id` entry (containing configuration parameters) or `NULL` if no match exists.

The return type is `const struct pci_id *` because the function returns a pointer into the read-only table, the caller must not modify the returned entry.

```
vendor = pci_get_vendor(dev);
device = pci_get_device(dev);
while (id->vendor != 0xffff &&
       (id->vendor != vendor || id->device != device))
    id++;
if (id->vendor == 0xffff)
    return (NULL);
```

6.14.4.4.3 Phase One: Primary ID Matching The function begins by reading the device’s primary identification from PCI configuration space. The `pci_get_vendor()` and `pci_get_device()` functions access configuration space registers 0x00 and 0x02, which every PCI device must implement.

The search loop: The while condition has two termination criteria:

1. `id->vendor != 0xffff` - haven’t reached the sentinel entry
2. `(id->vendor != vendor || id->device != device)` - current entry doesn’t match

The loop advances through the table until finding either a matching vendor/device pair or the sentinel. This linear search is acceptable because:

- The table has fewer than 100 entries (fast even with linear search)
- Probe happens once per device at boot (not performance-critical)
- The table is in cache-friendly sequential memory

Sentinel detection: If the loop exits with `id->vendor == 0xffff`, no entry matched the device’s primary IDs. Returning NULL signals “not my device” to the probe function, which will return ENXIO to allow other drivers a chance.

```
if (id->subven == 0xffff)
    return (id);
```

6.14.4.4.4 Wildcard Subsystem Handling This is the fast-path exit for entries with wildcard subsystem IDs. When `subven == 0xffff`, the entry matches all variants of this chipset regardless of OEM customization. The function returns immediately without reading subsystem IDs from configuration space.

This optimization avoids unnecessary PCI configuration reads for the common case where the driver accepts all OEM variants of a chipset (e.g., “Intel AMT - SOL” matches Intel’s chipset in any motherboard).

```
subven = pci_get_subvendor(dev);
subdev = pci_get_subdevice(dev);
while (id->vendor == vendor && id->device == device &&
       (id->subven != subven || id->subdev != subdev))
    id++;
```

6.14.4.4.5 Phase Two: Subsystem ID Matching For entries requiring specific subsystem matches, the function reads the subsystem vendor and device IDs from PCI configuration space registers 0x2C and 0x2E.

The refinement loop: This second search advances through consecutive table entries with the same primary IDs, looking for a subsystem match. The loop continues while:

1. `id->vendor == vendor && id->device == device` - still examining entries for this chipset
2. `(id->subven != subven || id->subdev != subdev)` - subsystem IDs don't match

This handles tables with multiple entries for one chipset, each specifying different OEM variants:

c

```
{ 0x103c, 0x1048, 0x103c, 0x1227, "HP Diva Serial - Powerbar SP2", 0x10 },
{ 0x103c, 0x1048, 0x103c, 0x1301, "HP Diva RMP3", 0x14 },
```

Both entries have vendor 0x103c and device 0x1048, but different subsystem device IDs. The loop examines each until finding the correct variant.

```
return ((id->vendor == vendor && id->device == device) ? id : NULL);
```

6.14.4.4.6 Final Validation After the refinement loop exits, one of two conditions holds:

1. The loop found a matching entry (all four IDs match) -> return it
2. The loop exhausted entries for this chipset without matching subsystems -> return NULL

The ternary expression performs a final sanity check: even though the loop condition guarantees `id` points to an entry with matching primary IDs (or past the last such entry), explicitly verifying ensures correct behavior if the loop walked past all entries for this device without finding a subsystem match.

This covers the case where:

- Primary IDs match (phase one succeeded)
- Table has entries with specific subsystem requirements
- None of those subsystem entries match the device
- The loop advanced until finding a different primary ID or the sentinel

6.14.4.4.7 Matching Examples **Example 1: Simple wildcard match**

- Device: Intel AMT SOL (vendor 0x8086, device 0x108f)
- Phase one: finds { 0x8086, 0x108f, 0xffff, 0, ... }
- Wildcard check: `subven == 0xffff`, return immediately
- Result: match without reading subsystem IDs

Example 2: OEM-specific match

- Device: HP Diva RMP3 (vendor 0x103c, device 0x1048, subven 0x103c, subdev 0x1301)

- Phase one: finds first entry with vendor 0x103c, device 0x1048
- Wildcard check: `subven != 0xffff`, read subsystem IDs
- Phase two: first entry has subdev 0x1227 (no match), advance
- Phase two: second entry has subdev 0x1301 (match!), return
- Result: returns second entry with BAR 0x14 and correct description

Example 3: No match

- Device: Unknown UART (vendor 0x1234, device 0x5678)
- Phase one: walks entire table without finding matching primary IDs
- Sentinel detection: returns NULL
- Result: probe function returns `ENXIO`

6.14.4.4.8 Efficiency Considerations The two-phase approach optimizes the common case:

- Most table entries use wildcard subsystems (require only primary ID match)
- Reading PCI configuration space is slower than memory access
- Deferring subsystem ID reads until necessary reduces probe latency

For devices with wildcard entries, the function performs two configuration space reads (vendor, device) and returns. Only devices requiring subsystem matching incur four reads.

The linear search is justified because:

- Table size is bounded and small (< 100 entries)
- Modern CPUs prefetch sequential memory efficiently
- Probe happens once per device lifetime, not in I/O paths
- Code simplicity outweighs marginal speedup from binary search or hash tables

6.14.4.4.9 Integration with Probe Function The probe function calls `uart_pci_match` with the table base pointer:

```
id = uart_pci_match(dev, pci_ns8250_ids);
if (id != NULL) {
    sc->sc_class = &uart_ns8250_class;
    goto match;
}
```

A non-NULL return provides both confirmation that the device is supported and access to its configuration parameters (`id->rid`, `id->rclk`, `id->regshft`). The probe function uses these values to initialize the generic UART layer correctly for this hardware variant.

6.14.4.5 5) Console uniqueness helper (rare but educational)

```

239: extern SLIST_HEAD(uart_devinfo_list, uart_devinfo) uart_sysdevs;
242: static const struct pci_unique_id pci_unique_devices[] = {
243: { 0x1d0f, 0x8250 } /* Amazon PCI serial device */
244: };
248: static void
249: uart_pci_unique_console_match(device_t dev)
250: {
251:     struct uart_softc *sc;
252:     struct uart_devinfo * sysdev;
253:     const struct pci_unique_id * id;
254:     uint16_t vendor, device;
255:
256:     sc = device_get_softc(dev);
257:     vendor = pci_get_vendor(dev);
258:     device = pci_get_device(dev);
259:
260:     /* Is this a device known to exist only once in a system? */
261:     for (id = pci_unique_devices; ; id++) {
262:         if (id == &pci_unique_devices[nitems(pci_unique_devices)])
263:             return;
264:         if (id->vendor == vendor && id->device == device)
265:             break;
266:     }
267:
268:     /* If it matches a console, it must be the same device. */
269:     SLIST_FOREACH(sysdev, &uart_sysdevs, next) {
270:         if (sysdev->pci_info.vendor == vendor &&
271:             sysdev->pci_info.device == device) {
272:             sc->sc_sysdev = sysdev;
273:             sysdev->bas.rclk = sc->sc_bas.rclk;
274:         }
275:     }

```

If a PCI UART is known to be **unique** in a system, tie it to the console instance automatically.

6.14.4.5.1 Console Device Matching: `uart_pci_unique_console_match` FreeBSD must identify which UART serves as the system console, the device where boot messages appear and where single-user mode login occurs. For most systems, firmware or the boot loader configures the console before the kernel starts, but the kernel must later match this pre-configured console to the correct driver instance during PCI enumeration. The `uart_pci_unique_console_match` function solves this matching problem for devices guaranteed to exist only once per system.

6.14.4.5.2 The Console Matching Problem When the kernel boots, early console output may use a UART initialized by firmware (BIOS/UEFI) or the boot loader. This “system device” (sysdev) has register addresses and basic configuration but no association with a PCI device tree entry. Later, during normal device enumeration, the PCI bus driver discovers UARTs and attaches driver instances. The kernel must determine which enumerated device corresponds to the pre-configured console.

The challenge: PCI enumeration order is not guaranteed. The device at PCI address 0:1f:3 (bus 0, device 31, function 3) might enumerate as `uart0` on one boot and `uart1` after adding a card. Matching by device tree position would be unreliable.

```
extern SLIST_HEAD(uart_devinfo_list, uart_devinfo) uart_sysdevs;

static const struct pci_unique_id pci_unique_devices[] = {
{ 0x1d0f, 0x8250 } /* Amazon PCI serial device */
};
```

6.14.4.5.3 The Unique Device Approach The solution for certain hardware: some devices are architecturally guaranteed to exist only once. Server management controllers, SoC-integrated UARTs, and cloud instance serial ports fall into this category. For these devices, vendor and device IDs alone suffice for matching.

The `uart_sysdevs` list contains pre-configured console devices recorded during early boot. Each `uart_devinfo` structure captures the console’s register base address, baud rate, and (if known) PCI identification.

The `pci_unique_devices` array lists devices meeting the uniqueness criterion. Currently it contains only Amazon’s EC2 serial device (vendor 0x1d0f, device 0x8250), which exists exactly once in EC2 instances and serves as the console for serial console access.

```
static void
uart_pci_unique_console_match(device_t dev)
{
    struct uart_softc *sc;
    struct uart_devinfo *sysdev;
    const struct pci_unique_id *id;
    uint16_t vendor, device;

    sc = device_get_softc(dev);
    vendor = pci_get_vendor(dev);
    device = pci_get_device(dev);
```

6.14.4.5.4 Function Entry and Device Identification The function is called from `uart_pci_probe` after successful device identification but before final probe completion. It receives the device being probed and retrieves:

- The softc (driver instance state) via `device_get_softc()`
- The device's vendor and device IDs from PCI configuration space

The softc at this point has been partially initialized by `uart_bus_probe()` with register access methods and clock rates, but `sc->sc_sysdev` is NULL unless console matching succeeds.

```
/* Is this a device known to exist only once in a system? */
for (id = pci_unique_devices; ; id++) {
    if (id == &pci_unique_devices[nitems(pci_unique_devices)])
        return;
    if (id->vendor == vendor && id->device == device)
        break;
}
```

6.14.4.5.5 Uniqueness Verification The loop searches the unique device table for a match. Two exit conditions:

Not unique: If the loop walks past the last entry without matching, this device isn't guaranteed unique. The function returns immediately; console matching requires stricter identification (likely including subsystem IDs or base address comparison), which this function doesn't attempt.

Is unique: If vendor and device IDs match an entry, the device is guaranteed unique in the system. The loop breaks, and matching proceeds.

The array bounds check uses `nitems(pci_unique_devices)`, a macro computing array element count. This pointer comparison detects when `id` has advanced past the array's end:

```
if (id == &pci_unique_devices[nitems(pci_unique_devices)])
```

This is equivalent to `id == pci_unique_devices + array_length`, checking if the pointer equals the address just beyond the last valid element.

```
/* If it matches a console, it must be the same device. */
SLIST_FOREACH(sysdev, &uart_sysdevs, next) {
    if (sysdev->pci_info.vendor == vendor &&
        sysdev->pci_info.device == device) {
        sc->sc_sysdev = sysdev;
        sysdev->bas.rclk = sc->sc_bas.rclk;
    }
}
```

6.14.4.5.6 Console Device Matching The `SLIST_FOREACH` macro iterates the system device list, checking each pre-configured console for matching PCI IDs. The list typically contains zero or one entry (systems without serial consoles or with one console), but the code correctly handles multiple consoles.

Match confirmation: When `sysdev->pci_info` matches the device's vendor and device IDs, the uniqueness guarantee ensures this enumerated device is the same physical hardware the firmware configured as a console. No ambiguity exists; there's only one device with these IDs in the system.

Linking the instances: `sc->sc_sysdev = sysdev` creates a bidirectional association:

- The driver instance (`sc`) now knows it's managing a console device
- Console-specific behaviors activate: special character handling, kernel message output, debugger entry

Clock synchronization: `sysdev->bas.rclk = sc->sc_bas.rclk` updates the system device's clock rate to match the value from the identification table. Early boot initialization might not know the precise clock frequency, using a default or probe-detected value. The PCI driver, having matched the device against the table, knows the correct frequency and updates the system device record.

This clock update is critical: if early boot used an incorrect clock, baud rate calculations would be wrong. The console might have worked by luck (if firmware configured the UART's divisor latch directly) but would fail when the driver reconfigures it. Synchronizing `rclk` ensures subsequent operations use correct values.

6.14.4.5.7 Why This Function Exists Traditional console matching compares base addresses: the system device's physical register address matches the PCI BAR of one enumerated device. This works reliably but requires reading BARs for all UARTs and handling complications like I/O port vs. memory-mapped registers.

For unique devices, vendor/device ID matching is simpler and equally reliable. The uniqueness guarantee eliminates ambiguity: if a unique device exists as a console and that device is enumerated, they must be the same.

6.14.4.5.8 Limitations and Scope This function only handles devices in `pci_unique_devices`. Most UARTs don't qualify:

- Multi-port cards have identical vendor/device IDs for all ports
- Generic chipsets appear in multiple products
- Motherboard UARTs from one vendor may use the same chipset across product lines

For non-unique devices, the probe function falls back to other matching methods (typically base address comparison in `uart_bus_probe`), or the console association might be established through hints or device tree properties.

The function is called opportunistically: it attempts to match for all probed devices but only succeeds for unique devices that also happen to be consoles. Failure is not an error; it simply means this device is either not unique or not a console.

6.14.4.5.9 Integration Context The probe function calls this after initial device identification:

```
result = uart_bus_probe(dev, ...);
if (sc->sc_sysdev == NULL)
    uart_pci_unique_console_match(dev);
```


The check `sc->sc_sysdev == NULL` ensures this function runs only if `uart_bus_probe` didn't already establish a console association through other means. This ordering provides a fallback: try precise matching first (base address comparison), then try unique device matching.

If matching succeeds, subsequent driver operations recognize the console status and enable special handling: synchronous output for panic messages, debugger break character detection, and kernel message routing.

6.14.4.6 6) probe: choose the class and call the shared bus probe

```

277: static int
278: uart_pci_probe(device_t dev)
279: {
280:     struct uart_softc *sc;
281:     const struct pci_id *id;
282:     struct pci_id cid = {
283:         .regshift = 0,
284:         .rclk = 0,
285:         .rid = 0x10 | PCI_NO_MSI,
286:         .desc = "Generic SimpleComm PCI device",
287:     };
288:     int result;
289:
290:     sc = device_get_softc(dev);
291:
292:     id = uart_pci_match(dev, pci_ns8250_ids);
293:     if (id != NULL) {
294:         sc->sc_class = &uart_ns8250_class;
295:         goto match;
296:     }
297:     if (pci_get_class(dev) == PCIC_SIMPLECOMM &&
298:         pci_get_subclass(dev) == PCIS_SIMPLECOMM_UART &&
299:         pci_get_progif(dev) < PCIP_SIMPLECOMM_UART_16550A) {
300:         /* XXX rclk what to do */
301:         id = &cid;
302:         sc->sc_class = &uart_ns8250_class;
303:         goto match;
304:     }
305:     /* Add checks for non-ns8250 IDs here. */
306:     return (ENXIO);
307:
308: match:
309:     result = uart_bus_probe(dev, id->regshift, 0, id->rclk,
310:         id->rid & PCI_RID_MASK, 0, 0);
311:     /* Bail out on error. */

```

```

312:     if (result > 0)
313:         return (result);
314:     /*
315:      * If we haven't already matched this to a console, check if it's a
316:      * PCI device which is known to only exist once in any given system
317:      * and we can match it that way.
318:      */
319:     if (sc->sc_sysdev == NULL)
320:         uart_pci_unique_console_match(dev);
321:     /* Set/override the device description. */
322:     if (id->desc)
323:         device_set_desc(dev, id->desc);
324:     return (result);
325: }

```

Two routes to a match: explicit table hit or class/subclass fallback. Then call the **UART bus probe** with *regshft*, *rclk*, and *rid*.

6.14.4.6.1 Device Probe Function: `uart_pci_probe` The probe function is the kernel’s first interaction with a potential device during enumeration. When the PCI bus driver discovers a device, it calls the probe function of every registered driver, asking “can you manage this device?” The probe function examines the hardware’s identification and configuration, returning a priority value indicating match quality or an error signaling “not my device.”

```

static int
uart_pci_probe(device_t dev)
{
    struct uart_softc *sc;
    const struct pci_id *id;
    int result;

    sc = device_get_softc(dev);

```

6.14.4.6.2 Function Purpose and Contract The probe function receives a `device_t` representing the hardware being examined. It must determine compatibility without modifying device state or allocating resources; those operations belong in the attach function.

The return value encodes probe results:

- Negative values or zero indicate success, with lower values representing better matches
- Positive values (particularly `ENXIO`) indicate “this driver cannot manage this device”
- The kernel selects the driver returning the lowest (best) value

The `softc` is retrieved via `device_get_softc()`, which returns a zeroed structure of the size specified in the driver declaration (`sizeof(struct uart_softc)`). The probe function initializes critical fields like `sc_class` before delegating to generic code.

```
id = uart_pci_match(dev, pci_ns8250_ids);
if (id != NULL) {
    sc->sc_class = &uart_ns8250_class;
    goto match;
}
```

6.14.4.6.3 Explicit Device Table Matching The primary matching path searches the explicit device table. If `uart_pci_match` returns non-NULL, the device is explicitly supported with known configuration parameters.

Setting the UART class: `sc->sc_class = &uart_ns8250_class` assigns the function table for NS8250-compatible register access. The `uart_class` structure (defined in the generic UART layer) contains function pointers for operations like:

- Reading/writing registers
- Configuring baud rates
- Managing FIFOs and flow control
- Handling interrupts

Different UART families (NS8250/16550, SAB82532, Z8530) would assign different class pointers. This driver only handles NS8250 variants, so the class assignment is unconditional.

The `goto match` bypasses subsequent checks, once explicitly identified, no further heuristics are needed.

```
if (pci_get_class(dev) == PCIC_SIMPLECOMM &&
    pci_get_subclass(dev) == PCIS_SIMPLECOMM_UART &&
    pci_get_progif(dev) < PCIP_SIMPLECOMM_UART_16550A) {
    /* XXX rclk what to do */
    id = &cid;
    sc->sc_class = &uart_ns8250_class;
    goto match;
}
```

6.14.4.6.4 Generic SimpleComm Device Fallback This fallback handles devices not in the explicit table but advertising themselves as generic UARTs through PCI class codes. The PCI specification defines a class/-subclass/programming interface hierarchy for device categorization:

Class check: `PCIC_SIMPLECOMM (0x07)` identifies “Simple Communication Controllers,” which includes serial ports, parallel ports, and modems.

Subclass check: PCIS_SIMPLECOMM_UART (0x00) narrows this to serial controllers specifically.

Programming interface check: `pci_get_progif(dev) < PCIP_SIMPLECOMM_UART_16550A` accepts devices claiming 8250-compatible (ProgIF 0x00) or 16450-compatible (ProgIF 0x01) programming interfaces, but rejects devices claiming 16550A compatibility (ProgIF 0x02) or higher.

This seemingly backwards logic exists because early 16550A implementations had broken FIFOs. The PCI specification allowed devices to claim “16550-compatible” without specifying whether FIFOs worked. Rejecting 16550A+ ProgIF values forces these devices through explicit table matching, where quirks can be documented. Only conservative 8250/16450 claims are trusted.

Fallback configuration: The `cid` structure (declared at function entry) provides default parameters:

```
struct pci_id cid = {
    .regshft = 0,          /* Standard register spacing */
    .rclk = 0,             /* Use default clock */
    .rid = 0x10 | PCI_NO_MSI, /* BAR0, no MSI */
    .desc = "Generic SimpleComm PCI device",
};
```

The comment `/* xxx rclk what to do */` highlights uncertainty: without explicit table entry, the correct clock frequency is unknown. The generic code defaults to 1.8432 MHz (standard PC UART clock), which works for most hardware but fails for devices with non-standard clocks.

The `PCI_NO_MSI` flag in the default RID disables MSI for generic devices. Since quirks aren’t known, conservative interrupt handling prevents potential MSI-related hangs or interrupt storms.

Setting `id = &cid` makes this local structure visible to the match path below, treating the generic configuration as if it came from the table.

```
/* Add checks for non-ns8250 IDs here. */
return (ENXIO);
```

6.14.4.6.5 Non-Match Exit If neither explicit matching nor generic class matching succeeds, the device isn’t a supported UART. Returning `ENXIO` (“Device not configured”) tells the kernel to try other drivers.

The comment indicates an extension point: drivers for other UART families (Exar, Oxford, Sunix with proprietary registers) would add their checks here before the final `ENXIO`.

```
match:
result = uart_bus_probe(dev, id->regshft, 0, id->rclk,
    id->rid & PCI_RID_MASK, 0, 0);
/* Bail out on error. */
if (result > 0)
    return (result);
```

6.14.4.6.6 Delegating to Generic Probe Logic The match label unifies both identification paths (explicit table and generic class). All subsequent code operates on `id`, which points either to a table entry or the `cid` structure.

Calling the generic layer: `uart_bus_probe()` lives in `uart_bus.c` and handles bus-agnostic initialization:

- Allocates and maps the I/O resource (BAR indicated by `id->rid`)
- Configures register access using `id->regshft`
- Sets the reference clock to `id->rclock` (or default if zero)
- Probes the hardware to verify UART presence and identify the FIFO depth
- Establishes register base address

The additional parameters (three zeros) specify:

- Flags controlling probe behavior
- Device unit number hint (0 = auto-assign)
- Reserved for future use

Error handling: If `uart_bus_probe` returns a positive value (error), that value propagates to the caller. Typical errors include:

- `ENOMEM` - couldn't allocate resources
- `ENXIO` - registers don't respond correctly (not a UART or disabled)
- `EIO` - hardware access failures

Successful probe returns zero or a negative priority value.

```
if (sc->sc_sysdev == NULL)
    uart_pci_unique_console_match(dev);
```

6.14.4.6.7 Console Device Association After successful generic probe, the driver attempts console matching. The check `sc->sc_sysdev == NULL` ensures this runs only if `uart_bus_probe` didn't already identify the device as a console (which it might have done via base address comparison).

Console association is opportunistic; failure doesn't prevent device attachment, it just means this UART won't receive kernel messages or serve as a login prompt.

```
/* Set/override the device description. */
if (id->desc)
    device_set_desc(dev, id->desc);
return (result);
```

6.14.4.6.8 Setting Device Description The device description appears in boot messages, `dmesg`, and `pci-conf -lv` output. It helps administrators identify hardware: "Intel AMT - SOL" is more meaningful than "PCI device 8086:108f."

For explicitly matched devices, `id->desc` contains the table-specified string. For generic devices, it's "Generic SimpleComm PCI device." The description is set unconditionally if present; even if a generic probe set one, the PCI-specific driver overrides it with more accurate information.

Finally, the function returns the result from `uart_bus_probe`, which the kernel uses to select among competing drivers. For UARTs, this is typically `BUS_PROBE_DEFAULT` (-20), the standard priority for base-OS drivers, since NS8250 drivers are the only ones claiming these devices.

6.14.4.6.9 Probe Priority and Driver Selection The probe priority mechanism handles hardware claimed by multiple drivers. Consider a multi-function card with serial ports and network interfaces: - `uart_pci` might probe it (matches PCI class, returning `BUS_PROBE_DEFAULT` = -20) - A vendor-specific driver might also probe it (matching vendor/device ID exactly)

The vendor driver should return a higher value (closer to zero), such as `BUS_PROBE_VENDOR` (-10) or `BUS_PROBE_SPECIFIC` (0), and Newbus will select it because its priority is **greater** than `BUS_PROBE_DEFAULT`. Remember: closer to zero wins.

For most serial hardware, only `uart_pci` probes successfully, making priority moot. But the mechanism allows graceful coexistence with specialised drivers.

```
PCI bus discovers device
    ->
Calls uart_pci_probe(dev)
    ->
Check explicit table -> uart_pci_match()
    -> (if matched)
Set NS8250 class, jump to match label
    ->
Check PCI class codes
    -> (if generic UART)
Use default config, jump to match label
    -> (if neither)
Return ENXIO (not my device)

match:
    ->
Call uart_bus_probe() for generic init
    -> (on error)
Return error code
    -> (on success)
Attempt console matching (if needed)
    ->
Set device description
    ->
Return success (0 or priority)
```

6.14.4.6.10 The Complete Probe Flow After successful probe, the kernel records this driver as the handler for this device and will later call `uart_pci_attach` to complete initialization.

6.14.4.7 7) attach: prefer single-vector MSI, then defer to the core

```

327: static int
328: uart_pci_attach(device_t dev)
329: {
330:     struct uart_softc *sc;
331:     const struct pci_id *id;
332:     int count;
333:
334:     sc = device_get_softc(dev);
335:
336:     /*
337:      * Use MSI in preference to legacy IRQ if available. However, experience
338:      * suggests this is only reliable when one MSI vector is advertised.
339:      */
340:     id = uart_pci_match(dev, pci_ns8250_ids);
341:     if ((id == NULL || (id->rid & PCI_NO_MSI) == 0) &&
342:         pci_msi_count(dev) == 1) {
343:         count = 1;
344:         if (pci_alloc_msi(dev, &count) == 0) {
345:             sc->sc_irid = 1;
346:             device_printf(dev, "Using %d MSI message\n", count);
347:         }
348:     }
349:
350:     return (uart_bus_attach(dev));
351: }
```

Small bus-specific policy (prefer 1-vector MSI) and then *delegate* to `uart_bus_attach()`.

6.14.4.7.1 Device Attach Function: `uart_pci_attach` The attach function is called after successful probe to make the device operational. While probe merely identifies the device and verifies compatibility, attach allocates resources, configures hardware, and integrates the device into the system. For `uart_pci`, attach focuses on one PCI-specific concern, interrupt configuration, before delegating to the generic UART initialization code.

```

static int
uart_pci_attach(device_t dev)
{
```

```

struct uart_softc *sc;
const struct pci_id *id;
int count;

sc = device_get_softc(dev);

```

6.14.4.7.2 Function Entry and Context The attach function receives the same `device_t` passed to probe. The `softc` retrieved here contains initialization performed during probe: the UART class assignment, base address configuration, and any console association.

Unlike probe (which must be idempotent and non-destructive), attach may modify device state, allocate resources, and fail destructively. If attach fails, the device becomes unavailable and typically requires reboot or manual intervention to recover.

6.14.4.7.3 Message Signaled Interrupts: Background Traditional PCI interrupts use dedicated physical signal lines (INTx: INTA#, INTB#, INTC#, INTD#) shared among multiple devices. This sharing causes several problems:

- Interrupt storms when devices don't properly acknowledge interrupts
- Latency from iterating handlers until finding the interrupting device
- Limited routing flexibility in complex systems

Message Signaled Interrupts (MSI) replace physical signals with memory writes to special addresses. When a device needs service, it writes to a CPU-specific address, triggering an interrupt on that CPU. MSI advantages:

- No sharing, each device gets dedicated interrupt vectors
- Lower latency, direct CPU targeting
- Better scalability, thousands of vectors available vs. four INTx lines

However, MSI implementation quality varies, particularly in UARTs (simple devices often getting minimal validation). Some UART MSI implementations suffer from lost interrupts, spurious interrupts, or system hangs.

```

/*
 * Use MSI in preference to legacy IRQ if available. However, experience
 * suggests this is only reliable when one MSI vector is advertised.
 */
id = uart_pci_match(dev, pci_ns8250_ids);
if ((id == NULL || (id->rid & PCI_NO_MSI) == 0) &&
    pci_msi_count(dev) == 1) {

```

6.14.4.7.4 MSI Eligibility Check The driver attempts MSI allocation only when three conditions hold:

Device not in table OR MSI not explicitly disabled: The condition `(id == NULL || (id->rid & PCI_NO_MSI) == 0)` evaluates true in two cases:

1. `id == NULL` - device matched via generic class codes, not explicit table entry (no known quirks)
2. `(id->rid & PCI_NO_MSI) == 0` - device in table, but MSI flag is clear (MSI known working)

If the device has `PCI_NO_MSI` set in its table entry, this condition fails and MSI allocation is skipped entirely. Legacy line-based interrupts will be used instead.

Single MSI vector advertised: `pci_msi_count(dev) == 1` queries the device's MSI capability structure to determine how many interrupt vectors it supports. UARTs only need one interrupt (serial events: received character, transmit buffer empty, modem status change), so multi-vector support is unnecessary.

The comment captures hard-won experience: devices advertising multiple MSI vectors (even though they only use one) often have buggy implementations. Restricting allocation to single-vector devices avoids these problems. A device advertising eight vectors for a simple UART likely received minimal MSI testing.

```
count = 1;
if (pci_alloc_msi(dev, &count) == 0) {
    sc->sc_irid = 1;
    device_printf(dev, "Using %d MSI message\n", count);
}
```

6.14.4.7.5 MSI Allocation Requesting allocation: `pci_alloc_msi(dev, &count)` asks the PCI subsystem to allocate MSI vectors for this device. The count parameter is both input and output: - Input: requested number of vectors (1) - Output: actual allocated count (might be less if resources exhausted)

The function returns zero on success, non-zero on failure. Failure reasons include: - System doesn't support MSI (old chipsets, disabled in BIOS) - MSI resources exhausted (too many devices already using MSI) - Device MSI capability structure is malformed

Recording interrupt resource ID: On successful allocation, `sc->sc_irid = 1` records that interrupt resource ID 1 will be used. The significance: - RID 0 typically represents the legacy INTx interrupt - RID 1+ represent MSI vectors - The generic UART attach code will allocate the interrupt resource using this RID

Without this assignment, the default RID (0) would be used, causing the driver to allocate the legacy interrupt instead of the newly-allocated MSI vector.

User notification: `device_printf` logs the MSI allocation to the console and system message buffer. This information helps administrators debug interrupt-related issues. Output appears as:

```
uart0: <Intel AMT - SOL> port 0xf0e0-0xf0e7 mem 0xfebff000-0xfebff0ff irq 16 at device
↪ 22.0 on pci0
uart0: Using 1 MSI message
```

Silent fallback: If `pci_alloc_msi` fails, the conditional body doesn't execute. The `sc->sc_irid` field remains at its default value (0), and no message is printed. The attach function proceeds to generic initialization, which will allocate the legacy interrupt. This silent fallback ensures device functionality even when MSI is unavailable, legacy interrupts work universally.

```
return (uart_bus_attach(dev));
```

6.14.4.7.6 Delegating to Generic Attach After PCI-specific interrupt configuration, the function calls `uart_bus_attach()` to complete initialization. This generic function (shared across all bus types: PCI, ISA, ACPI, USB) performs:

Resource allocation: - I/O ports or memory-mapped registers (already mapped during probe) - Interrupt resource (using `sc->sc_irid` to select MSI or legacy) - Possibly DMA resources (not used by most UARTs)

Hardware initialization: - Reset the UART - Configure default parameters (8 data bits, no parity, 1 stop bit) - Enable and size the FIFO - Set up modem control signals

Character device creation: - Allocate TTY structures - Create device nodes (`/dev/cuaU0`, `/dev/ttyU0`) - Register with the TTY layer for line discipline support

Console integration: - If `sc->sc_sysdev` is set, configure as system console - Enable console output through this UART - Handle kernel debugger entry via break signals

Return value propagation: The return value from `uart_bus_attach()` passes directly to the kernel. Success (0) indicates the device is operational; errors (positive `errno` values) indicate failure.

6.14.4.7.7 Attach Failure Handling If `uart_bus_attach()` fails, the device remains unusable. The PCI subsystem notes the failure and won't call device methods (`read`, `write`, `ioctl`) on this instance. However, resources already allocated by `attach` (like MSI vectors) may leak unless the driver's `detach` function is called.

Proper error handling in the generic `attach` code ensures:

- Failed interrupt allocation triggers resource cleanup
- Partial initialization is rolled back
- The device remains in a safe state for retry or removal

```
Kernel calls uart_pci_attach(dev)
->
Check MSI eligibility
| -> Device has PCI_NO_MSI flag -> skip MSI
| -> Device advertises multiple vectors -> skip MSI
| -> Device advertises one vector -> attempt MSI
->
Allocate MSI vector via pci_alloc_msi()
| -> Success: set sc->sc_irid = 1, log message
| -> Failure: silent, sc->sc_irid remains 0
->
Call uart_bus_attach(dev)
->
Generic code allocates interrupt using sc->sc_irid
| -> RID 1: MSI vector
| -> RID 0: legacy INTx
->
```

```
Complete UART initialization
->
Create device nodes (/dev/cuaU*, /dev/ttyU*)
->
Return success/failure
```

6.14.4.7.8 The Complete Attach Flow After successful attach, the UART is fully operational. Applications can open `/dev/cuaU0` for serial communication, kernel messages flow to the console (if configured), and interrupt-driven I/O handles character transmission and reception.

6.14.4.7.9 Architectural Simplicity The attached function's brevity, twenty-three lines including comments, demonstrates the layered architecture's power. PCI-specific concerns (MSI allocation) are handled here in minimal code, while complex UART initialization lives in the generic layer where it's shared across all bus types.

This separation means:

- ISA-attached UARTs skip MSI logic but reuse all UART initialization
- ACPI-attached UARTs might handle power management differently but share character device creation
- USB serial adapters use completely different interrupt delivery but share TTY integration

The `uart_pci` driver is thin glue connecting PCI resource management to generic UART functionality, exactly as intended.

6.14.4.8 8) detach and module registration

```
353: static int
354: uart_pci_detach(device_t dev)
355: {
356:     struct uart_softc *sc;
357:
358:     sc = device_get_softc(dev);
359:
360:     if (sc->sc_irid != 0)
361:         pci_release_msi(dev);
362:
363:     return (uart_bus_detach(dev));
364: }
366: DRIVER_MODULE(uart, pci, uart_pci_driver, NULL, NULL);
```

*Release MSI if we took it, then let the UART core unwind. Finally, register this driver on the **pci** bus.*

6.14.4.8.1 Device Detach Function and Driver Registration The detach function is called when a device must be removed from the system, either due to hot-unplug, driver unload, or system shutdown. It must reverse all operations performed during attach, releasing resources and ensuring the hardware is left in a safe state. The final `DRIVER_MODULE` macro registers the driver with the kernel's device framework.

```
static int
uart_pci_detach(device_t dev)
{
    struct uart_softc *sc;

    sc = device_get_softc(dev);
```

6.14.4.8.2 Device Detach Function: `uart_pci_detach` Detach receives the device being removed and retrieves its `softc` containing the current configuration. The function must be prepared to handle partial initialization states, if attach failed midway, detach might be called to clean up whatever succeeded.

```
if (sc->sc_irid != 0)
    pci_release_msi(dev);
```

6.14.4.8.3 MSI Resource Release The conditional checks whether MSI was allocated during attach. Recall that `sc->sc_irid = 1` signals successful MSI allocation; the default value (0) indicates legacy interrupts were used.

Releasing MSI vectors: `pci_release_msi(dev)` returns the MSI interrupt vector to the system pool, making it available for other devices. This call must be made before the generic detach, which will deallocate the interrupt resource itself. The sequence matters:

1. Release MSI allocation (returns vector to system)
2. Generic detach deallocates the interrupt resource (frees kernel structures)

Reversing this order would leak MSI vectors, the kernel would consider them allocated even after the device is gone.

Why check `sc_irid`? Calling `pci_release_msi` when MSI wasn't allocated is harmless but wastes cycles. More importantly, it documents the code's intent: "if we allocated MSI during attach, release it during detach." This symmetry aids understanding.

The lack of error handling is intentional, `pci_release_msi` cannot meaningfully fail during detach. The device is being removed regardless; if MSI release fails (due to corrupted kernel state), proceeding with detach is still correct.

```
return (uart_bus_detach(dev));
```

6.14.4.8.4 Delegating to Generic Detach After PCI-specific resource cleanup, the function calls `uart_bus_attach()` to handle generic UART teardown. This mirrors the attach sequence: PCI-specific code wraps generic code.

Generic detach operations:

Character device removal: Close any open file descriptors, destroy `/dev/cuaU*` and `/dev/ttyU*` nodes, and deregister from the TTY layer.

Hardware shutdown: Disable interrupts at the UART, flush FIFOs, and deassert modem control signals. This prevents the hardware from generating spurious interrupts or asserting control lines after the driver is gone.

Resource deallocation: Free the interrupt resource (the kernel structure, not the MSI vector, that was already released above), unmap I/O ports or memory regions, and release any allocated kernel memory.

Console disconnection: If this device was the system console, redirect console output to an alternative device or disable console output entirely. The system must remain bootable even if the console UART is removed.

Return value: `uart_bus_detach()` returns zero on success or an error code on failure. In practice, detach rarely fails, the device is being removed whether or not software cleanup succeeds gracefully.

6.14.4.8.5 Detach Failure Consequences If detach returns an error, the kernel's response depends on context:

Driver unload: If attempting to unload the driver module (`kldunload uart_pci`), the operation fails and the module remains loaded. The device stays attached, preventing resource leaks.

Device hot-removal: If physical removal triggered detach (PCIe hot-unplug), the hardware is already gone. Detach failure is logged but the device tree entry is removed anyway. Resource leaks may occur, but system stability is preserved.

System shutdown: During shutdown, detach failures are ignored. The system is halting regardless, so resource leaks are irrelevant.

Well-designed detach functions should never fail. The `uart_pci` implementation achieves this by:

- Performing only infallible operations (resource release)
- Delegating complex logic to generic code that handles edge cases
- Not requiring hardware responses (hardware might already be disconnected)

```
DRIVER_MODULE(uart, pci, uart_pci_driver, NULL, NULL);
```

6.14.4.8.6 Driver Registration: DRIVER_MODULE This macro registers the driver with FreeBSD's device framework, making it available for device matching during boot and module load. The macro expands to considerable infrastructure code, but its parameters are straightforward:

uart: The driver name, matching the string in `uart_driver_name`. This name appears in kernel messages, device tree paths, and administrative commands. Multiple drivers can share the same name if they attach to different buses, `uart_pci`, `uart_isa`, and `uart_acpi` all use “uart”, distinguishing themselves by the bus they attach to.

pci: The parent bus name. This driver attaches to the PCI bus, so it specifies “pci”. The kernel’s bus framework uses this to determine when to call the driver’s probe function, only PCI devices are offered to `uart_pci`.

uart_pci_driver: Pointer to the `driver_t` structure defined earlier, containing the method table and softc size. The kernel uses this to invoke driver methods and allocate per-device state.

NULL, NULL: Two reserved parameters for module initialization hooks. Most drivers don’t need these, passing NULL for both. The hooks allow running code when the module loads (before any device attach) or unloads (after all devices detach). Uses include:

- Allocating global resources (memory pools, worker threads)
- Registering with subsystems (like the network stack)
- Performing one-time hardware initialization

For `uart_pci`, no module-level initialization is needed, all work happens in probe/attach on a per-device basis.

6.14.4.8.7 The Module Lifecycle The `DRIVER_MODULE` macro makes the driver participates in FreeBSD’s modular kernel architecture:

Static compilation: If compiled into the kernel (options `UART` in kernel config), the driver is available at boot. The linker includes `uart_pci_driver` in the kernel’s driver table, and PCI enumeration during boot calls its probe function.

Dynamic loading: If compiled as a module (`kldload uart_pci.ko`), the module loader processes the `DRIVER_MODULE` registration, adding the driver to the active table. Existing devices are reprobbed; new matches trigger attach.

Dynamic unloading: `kldunload uart_pci` attempts to detach all devices managed by this driver. If any detach fails or devices are in use (open file descriptors), unload fails and the module remains. Successful unload removes the driver from the active table.

6.14.4.8.8 Relationship to Other UART Drivers The FreeBSD UART subsystem includes multiple bus-specific drivers all sharing generic code:

- `uart_pci.c` - PCI-attached UARTs (this driver)
- `uart_isa.c` - ISA bus UARTs (legacy COM ports)
- `uart_acpi.c` - ACPI-enumerated UARTs (modern laptops/servers)
- `uart_fdt.c` - Flattened Device Tree UARTs (embedded systems, ARM)

Each uses `DRIVER_MODULE` to register with its respective bus:

```
DRIVER_MODULE(uart, pci, uart_pci_driver, NULL, NULL); // PCI bus
DRIVER_MODULE(uart, isa, uart_isa_driver, NULL, NULL); // ISA bus
DRIVER_MODULE(uart, acpi, uart_acpi_driver, NULL, NULL); // ACPI bus
```

All share the name “uart” but attach to different buses. A system might load all four modules simultaneously, with each handling UARTs discovered on its bus. A desktop might have: - Two ISA COM ports (COM1/COM2 via `uart_isa`) - One PCI management controller (IPMI via `uart_pci`) - Zero ACPI UARTs (not present)

Each device gets an independent driver instance, all sharing the generic UART code in `uart_bus.c` and `uart_core.c`.

6.14.4.8.9 Complete Driver Structure With all pieces explained, the complete driver structure is:

```
uart_pci_methods[] -> Method table (probe/attach/detach/resume)

uart_pci_driver -> Driver declaration (name, methods, softc size)

DRIVER_MODULE() -> Registration (uart, pci, uart_pci_driver)
```

At runtime, the PCI bus driver discovers devices and consults the registered driver table. For each device, it calls probe functions of matching drivers. The `uart_pci` probe function examines device IDs against its table, returning success for matches. The kernel then calls attach to initialize the device. Later, detach cleans up when the device is removed.

This architecture, method tables, layered initialization, bus-independent core logic, repeats throughout FreeBSD’s device driver framework. Understanding it in the `uart_pci` context prepares you for more complex drivers: network cards, storage controllers, and graphics adapters all follow similar patterns at larger scale.

6.14.4.9 Interactive Exercises for `uart` (4)

Goal: Cement the PCI driver pattern: device identification tables -> probe -> attach -> generic core, with MSI as a bus-specific variation.

6.14.4.9.1 A) Driver Skeleton & Registration

1. Point to the `device_method_t` array and the `driver_t` structure. For each, identify what it declares and how they connect to each other. Quote the relevant lines. Which field in `driver_t` points to the method table? *Hint:* look for `uart_pci_methods[]` and the `uart_pci_driver` definition near the top of the file.
2. Where is the `DRIVER_MODULE` macro and which bus does it target? What are the five parameters it receives? Quote it and explain each parameter. *Hint:* `DRIVER_MODULE(uart, pci, ...)` sits at the bottom of the file.

6.14.4.9.2 B) Device Identification and Matching

1. In the `pci_ns8250_ids[]` table, find at least two Intel entries (vendor 0x8086) that demonstrate special handling: one with the `PCI_NO_MSI` flag and one with a non-standard clock frequency (`rc1k`). Quote both complete entries and explain what each special parameter means for the hardware. *Hint:* grep the table for 0x8086 and look near the Atom and ValleyView HSUART rows.

2. In `uart_pci_match()`, trace the two-phase matching logic. Where does the first loop match primary IDs (vendor/device)? Where does the second loop match subsystem IDs? What happens if an entry has `subven == 0xffff`? Quote the relevant lines (3-5 lines total). *Hint:* work through the two for loops in `uart_pci_match` and note the `subven == 0xffff` wildcard check.
3. Find an example in `pci_ns8250_ids[]` where the same vendor/device pair appears multiple times with different subsystem IDs. Quote 2-3 consecutive entries and explain why this duplication exists. *Hint:* the HP Diva block (vendor 0x103c, device 0x1048) and the Timedia 0x1409/0x7168 block in `pci_ns8250_ids`.

6.14.4.9.3 C) Probe Flow

1. In `uart_pci_probe()`, show where the code sets `sc->sc_class` to `&uart_ns8250_class` after successful table matching, and where it then calls `uart_bus_probe()`. Quote both spots (2-3 lines each). *Hint:* the class assignment sits on the success path after `uart_pci_match`, and the `uart_bus_probe` call is the final step before `uart_pci_probe` returns.
2. What does `uart_pci_unique_console_match()` do when it finds a unique device that matches a console? Quote the assignment to `sc->sc_sysdev` and the `rclock` synchronization line. Why is clock synchronization necessary? *Hint:* focus on the tail of `uart_pci_unique_console_match`, where `sc->sc_sysdev` is set and `sc->sc_sysdev->bas.rclk` is copied into `sc->sc_bas.rclk`.
3. In `uart_pci_probe()`, explain the fallback path for “Generic SimpleComm” devices. What PCI class, subclass, and progif values trigger this path? Why does the comment say “XXX rclk what to do”? Quote the conditional check and note what configuration is used. *Hint:* look for the local `cid` structure at the top of `uart_pci_probe` and the `pci_get_class/subclass/progif` check further down.

6.14.4.9.4 D) Attach and Detach

1. In `uart_pci_attach()`, why does the function re-match the device against the ID table when probe already did matching? Quote the line. *Hint:* look for the `uart_pci_match` call near the top of `uart_pci_attach`.
2. Quote the exact conditional that checks MSI eligibility (must prefer single-vector MSI) and the call that allocates it. What happens if MSI allocation fails? Quote 5-7 lines. *Hint:* the `pci_msi_count/pci_alloc_msi` block sits just after the `uart_pci_match` call in `uart_pci_attach`.
3. In `uart_pci_detach()`, quote the two critical operations: MSI release and delegation to generic detach. Why must MSI be released before calling `uart_bus_detach()`? Explain the order dependency. *Hint:* both the `pci_release_msi` call and the `uart_bus_detach` call appear in sequence inside `uart_pci_detach`.

6.14.4.9.5 E) Integration: Tracing Complete Flow

1. Starting from boot, trace how a Dell RAC 4 (vendor 0x1028, device 0x0012) becomes `/dev/cuaU0`. For each step, quote the relevant line:
 - Which table entry matches?
 - What clock frequency does it specify?

- What happens in probe? (which class is set? which function is called?)
 - What happens in attach? (will it use MSI?)
 - Which generic function creates the device node?
2. A device has vendor 0x8086, device 0xa13d (100 Series Chipset KT). Will it use MSI? Trace through the logic:
- Find and quote the table entry
 - Check the `rid` field, what flag is present?
 - Quote the conditional in `uart_pci_attach()` that checks this flag
 - What interrupt mechanism will be used instead?

6.14.4.9.6 F) Architecture and Design Patterns

1. Compare `if_tuntap.c` (from the previous section) with `uart_bus_pci.c`:
- `if_tuntap` had ~2200 lines; `uart_bus_pci` has ~370. Why such a difference in size?
 - `if_tuntap` contained complete device logic; `uart_bus_pci` is mostly glue code. Where does the actual UART register access, baud rate configuration, and TTY integration happen? (Hint: what function does `attach` call?)
 - Which design approach, monolithic like `if_tuntap` or layered like `uart_bus_pci`, makes it easier to support the same hardware on multiple buses (PCI, ISA, USB)?
2. Imagine you need to add support for:
- A new PCI UART: vendor 0xABCD, device 0x1234, standard clock, BAR 0x10
 - An ISA-attached version of the same UART chipset

For the PCI variant, what would you modify in `uart_bus_pci.c`? (Quote the structure and location) For the ISA variant, would you modify `uart_bus_pci.c` at all, or work in a different file? How many lines of UART register access code would you need to write/duplicate?

6.14.4.10 Stretch (thought experiments)

Examine the MSI allocation logic in `uart_pci_attach()`.

The comment says “experience suggests this is only reliable when one MSI vector is advertised.”

1. Why would a simple UART (which only needs one interrupt) ever advertise multiple MSI vectors?
2. What problems might occur with multi-vector MSI that the driver avoids by checking `pci_msi_count(dev) == 1`?
3. If MSI allocation fails silently (the `if` condition is false), the driver continues. Where in the generic `attach` code will the interrupt resource be allocated instead? What type of interrupt will be used?

6.14.4.11 Why this matters in your “anatomy” chapter

You’ve just walked a **tiny PCI glue** driver end-to-end. It **matches** devices, chooses a UART **class**, calls a **shared probe/attach** in the subsystem core, and sprinkles light PCI policy (MSI/console). This is the same

shape you'll reuse for other buses: **match** -> **probe** -> **attach** -> **core**, plus **resources/IRQs** and **clean detach**. Keep this pattern in mind when you move from pseudo-devices to **real hardware** in later chapters.

6.15 From Four Drivers to One Mental Model

You've now walked through four complete drivers, each demonstrating different aspects of FreeBSD's device driver architecture. These weren't arbitrary examples; they form a deliberate progression that reveals the patterns underlying all kernel drivers.

6.15.1 The Progression You've Completed

Tour 1: `/dev/null`, `/dev/zero`, `/dev/full` (`null.c`)

- Simplest possible character devices
- Static device creation during module load
- Trivial operations: discard writes, return zeros, simulate errors
- No per-device state, no timers, no complexity
- **Key lesson:** The `cdevsw` function dispatch table and basic I/O with `uiomove()`

Tour 2: LED Subsystem (`led.c`)

- Dynamic device creation on demand
- Subsystem providing both userspace interface and kernel API
- Timer-driven state machine for pattern execution
- Pattern parsing DSL converting user commands to internal codes
- **Key lesson:** Stateful devices, infrastructure drivers, lock separation (`mtx` vs. `sx`)

Tour 3: TUN/TAP Network Tunnels (`if_tuntap.c`)

- Dual character device + network interface
- Bidirectional data flow: kernel <-> userspace packet exchange
- Network stack integration (`ifnet`, BPF, routing)
- Blocking I/O with proper wakeups (`poll/select/kqueue` support)
- **Key lesson:** Complex integration bridging two kernel subsystems

Tour 4: PCI UART Driver (`uart_bus_pci.c`)

- Hardware bus attachment (PCI enumeration)
- Layered architecture: thin bus glue + thick generic core
- Device identification via vendor/device ID tables
- Resource management (BARs, interrupts, MSI)
- **Key lesson:** The probe-attach-detach lifecycle, code reuse through layering

6.15.2 Patterns That Emerged

As you progressed through these drivers, certain patterns appeared repeatedly:

6.15.2.1 1. The Character Device Pattern

Every character device follows the same structure, whether it's `/dev/null` or `/dev/tun0`:

- A `cdevsw` structure mapping system calls to functions
- `make_dev()` creating the `/dev` entry
- `si_drv1` linking the device node to per-device state
- `destroy_dev()` cleaning up on removal

The complexity varies, `null.c` has no state, `led.c` tracks patterns, `tuntap` tracks network interface, but the skeleton is identical.

6.15.2.2 2. The Dynamic vs. Static Device Pattern

`null.c` creates three fixed devices at module load. `led.c` and `tuntap` create devices on demand as hardware registers or users open device nodes. This flexibility comes with complexity:

- Unit number allocation (`unrhdr`)
- Global registries (linked lists)
- More sophisticated locking

6.15.2.3 3. The Subsystem API Pattern

`led.c` demonstrates infrastructure design: it's both a device driver (exposing `/dev/led/*`) and a service provider (exporting `led_create()` for other drivers). This dual role appears throughout FreeBSD drivers that are libraries for other drivers.

6.15.2.4 4. The Layered Architecture Pattern

`uart_bus_pci.c` is minimal because most logic lives in `uart_bus.c`. The pattern:

- Bus-specific code handles: device identification, resource claiming, interrupt setup
- Generic code handles: device initialization, protocol implementation, user interface

This separation means the same UART logic works on PCI, ISA, USB, and device-tree platforms.

6.15.2.5 5. The Data Movement Patterns

You've seen three approaches to transferring data:

- **Simple**: `null_write` sets `uio_resid = 0` and returns (discard data)
- **Buffered**: `zero_read` loops calling `uiomove()` from a kernel buffer
- **Zero-copy**: `tuntap` uses `mbufs` for efficient packet handling

6.15.2.6 6. The Synchronization Patterns

Each driver's locking reflects its needs:

- `null.c`: none (stateless devices)
- `led.c`: two locks (mtx for fast state, sx for slow structure changes)
- `tuntap`: per-device mutex protecting queues and ifnet state
- `uart_pci`: minimal (most locking in generic `uart_bus` layer)

6.15.2.7 7. The Lifecycle Patterns

All drivers follow create-operate-destroy, but with variations:

- **Module lifecycle**: `null.c`'s `MOD_LOAD`/`MOD_UNLOAD` events
- **Dynamic lifecycle**: `led.c`'s `led_create()`/`led_destroy()` API
- **Clone lifecycle**: `tuntap`'s on-demand device creation
- **Hardware lifecycle**: `uart_pci`'s probe-attach-detach sequence

6.15.3 What You Can Now Recognize

After these four tours, when you encounter any FreeBSD driver, you should immediately identify:

What kind of driver is this?

- Character device only? (like `null.c`)
- Infrastructure/subsystem? (like `led.c`)
- Dual device/network? (like `tuntap`)
- Hardware bus attachment? (like `uart_pci`)

Where's the state?

- Global only? (`led.c`'s global list and timer)
- Per-device? (`tuntap`'s `softc` with queues and ifnet)
- Split? (`uart_pci`'s minimal state + `uart_bus`'s rich state)

How's it locked?

- One mutex for everything?
- Multiple locks for different data/access patterns?
- Handed off to generic code?

What's the data path?

- Copying with `uiomove()`?
- Using mbufs?
- Zero-copy techniques?

What's the lifecycle?

- Fixed (created once at load)?
- Dynamic (created on demand)?
- Hardware-driven (appears/disappears with physical devices)?

6.15.4 The Blueprint Ahead

The document that follows distills these patterns into a quick-reference guide, a checklist and template collection you can use when writing or analyzing drivers. It's organized by integration point (character device, network interface, bus attachment) and captures the critical decisions and invariants you must maintain.

Think of the four drivers you've studied as worked examples, and the blueprint as the extracted principles. Together, they form your foundation for understanding FreeBSD's driver architecture. The drivers showed you *how* things work in context; the blueprint reminds you *what* you must do to make your own drivers work correctly.

When you're ready to write your own driver or modify an existing one, start with the blueprint's self-check questions. Then refer back to the appropriate tour (null.c for basic devices, led.c for timers and APIs, tuntap for networking, uart_pci for hardware) to see those patterns in complete implementations.

You're now equipped to navigate the kernel's device drivers, not as intimidating black boxes, but as variations on patterns you've internalized through hands-on study.

6.16 Driver Anatomy Blueprint (FreeBSD 14.3)

This is your quick-reference map for FreeBSD drivers. It captures the shape (the moving parts and where they live), the contract (what the kernel expects from you), and the pitfalls (what breaks under load). Use it as the checklist before and after you code.

6.16.1 Core Skeleton: What Every Driver Needs

Identify your integration point:

Character device (devfs) -> struct cdevsw + make_dev*()/destroy_dev()

- Entry points: open/read/write/ioctl/poll/kqfilter
- Example: null.c, led.c

Network interface (ifnet) -> if_alloc()/if_attach()/if_free() + optional cdev

- Callbacks: if_transmit or if_start, input via netisr_dispatch()
- Example: if_tuntap.c

Bus-attached (e.g., PCI) -> device_method_t[] + driver_t + DRIVER_MODULE()

- Lifecycle: probe/attach/detach (+ suspend/resume if needed)
- Example: uart_bus_pci.c

Minimal invariants (commit these to memory):

- Every object you create (cdev, ifnet, callout, taskqueue, resource) has a symmetric destroy/free on error paths and during detach/unload
- Concurrency is explicit: if you touch state from multiple contexts (syscall path, timeout, rx/tx, interrupt), you hold the right lock or design for lock-free with strict rules
- Resource cleanup must happen in reverse order of allocation

6.16.2 Character Device Blueprint

Shape:

- `static struct cdevsw` with only what you implement; leave others `NULL` or omit
- Module or init hook creates nodes: `make_dev_credf()`/`make_dev_s()`
- Keep a `struct cdev *` to tear down later

Entry points:

read: Loop while `uio->uio_resid > 0`; move bytes with `uiomove()`; return early on error

- Example: `zero_read` loops copying from pre-zeroed kernel buffer

write: Either consume (`uio_resid = 0; return 0;`) or fail (`return ENOSPC/EIO/...`)

- No partial writes unless you mean it
- Example: `null_write` consumes all; `full_write` always fails

ioctl: Small `switch(cmd)`; return 0, specific `errno`, or `ENOIOCTL`

- Handle standard terminal `ioctls` (`FIONBIO`, `FIOASYNC`) even if they're no-ops
- Example: `null_ioctl` handles kernel dump configuration

poll/kqueue (optional): Wire readiness + notifications if userspace blocks

- Example: `tuntap`'s `poll` checks queue and registers via `selrecord()`

Concurrency & timers:

- If you have periodic work (e.g., LED blink), use a callout bound to the right mutex
- Arm/re-arm responsibly; stop it in teardown when the last user goes away
- Example: `led.c`'s `callout_init_mtx(&led_ch, &led_mtx, 0)`

Teardown:

- `destroy_dev()`, stop callouts/taskqueues, free buffers
- Clear pointers (e.g., `si_drv1 = NULL`) under lock before freeing
- Example: `led_destroy`'s two-phase cleanup (mtx then sx)

Check before lab:

- Can you match each user-visible behavior to the exact entry point?
- Are all allocations paired with frees on every error path?

6.16.3 Network Pseudo-Interface Blueprint

Two faces:

- Character device side (`/dev/tunN`, `/dev/tapN`) with `open/read/write/ioctl/poll`
- `ifnet` side (`ifconfig tun0 ...`) with `attach`, flags, link state, and BPF hooks

Data flow:**Kernel -> user (read):**

- Dequeue packet (mbuf) from your queue
- Block until available unless `O_NONBLOCK` (then `EWOULDBLOCK`)
- Copy optional headers first (virtio/ifhead), then payload via `m_mbufoutio()`
- Free mbuf with `m_freem()`
- Example: `tunread`'s loop with `mtx_sleep()` for blocking

User -> kernel (write):

- Build mbuf with `m_uiofrombuf()`
- Decide L2 vs L3 path
- For L3: pick AF and `netisr_dispatch()`
- For L2: validate destination (drop frames real NIC wouldn't receive unless promise)
- Example: `tunwrite_l3` dispatches via `NETISR_IP/NETISR_IPV6`

Lifecycle:

- Clone or first open creates `cdev` and `softc`
- Then `if_alloc()/if_attach()` and `bpfattach()`
- Open can raise link up; close can drop it
- Example: `tuncreate` builds `ifnet`, `tunopen` marks link UP

Notify readers:

- `wakeup()`, `selwakeuppri()`, `KNOTE()` when packets arrive
- Example: `tunstart`'s triple notification when packet enqueued

Check before lab:

- Do you know which paths block and which return immediately?
- Is your maximum I/O size bounded (MRU + headers)?
- Are wakeups fired on every packet enqueue?

6.16.4 PCI Glue Blueprint**Match -> Probe -> Attach -> Detach:**

Match: Vendor/device(/subvendor/subdevice) table; fall back to class/subclass when needed

- Example: `uart_pci_match`'s two-phase search (primary IDs then subsystem)

Probe: Choose driver class, compute parameters (reg shift, rclk, BAR RID), then call shared bus probe

- Example: `uart_pci_probe` sets `sc->sc_class = &uart_ns8250_class`

Attach: Allocate interrupts (prefer single-vector MSI if supported), then delegate to subsystem

- Example: `uart_pci_attach`'s conditional MSI allocation

Detach: Release MSI/IRQ, then delegate to subsystem detach

- Example: `uart_pci_detach` checks `sc_irid` and releases MSI if allocated

Resources:

- Map BARs, allocate IRQs, hand resources to core
- Track IDs so you can release them symmetrically
- Example: `id->rid & PCI_RID_MASK` extracts BAR number

Check before lab:

- Do you handle the “no match” path cleanly (`ENXIO`)?
- Are you leak-free across any mid-attach failure?
- Do you check for quirks (like `PCI_NO_MSI` flag)?

6.16.5 Locking & Concurrency Cheatsheet

Fast path data movement (read/write, rx/tx):

- Protect queues and state with a mutex
- Minimize hold time; never sleep while holding if avoidable
- Example: `tuntap`’s `tun_mtx` protecting send queue

Configuration / topology (create/destroy, link up/down):

- Typically an `sx` lock or higher-level serialization
- Example: `led.c`’s `led_sx` for device creation/destruction

Timer/callout:

- Use `callout_init_mtx(&callout, &mtx, flags)` so timeout runs with your mutex held
- Example: `led.c`’s timer automatically holds `led_mtx`

User-space notifications:

- After enqueueing: `wakeup(tp), selwakeuppri(&sel, PRIIO), KNOTE(&klist, NOTE_*)`
- Example: `tunstart`’s triple notification pattern

Lock ordering rules:

- Never acquire locks in inconsistent order
- Document your lock hierarchy
- Example: `led.c` acquires `led_mtx` then releases before taking `led_sx`

6.16.6 Data Movement Patterns

`uiomove()` loop for cdev read/write:

- Cap chunk size to a safe buffer (avoid giant copies)
- Check and handle errors on each iteration
- Example: `zero_read` limits to `ZERO_REGION_SIZE` per iteration

mbuf path for networking:**User -> kernel:**

```
m = m_uiotombuf(uio, M_NOWAIT, 0, align, M_PKTHDR);  
// set metadata (AF/virtio)  
netisr_dispatch(isr, m);
```

Kernel -> user:

```
// optional header to user (uiomove())  
m_mbuftouio(uio, m, 0);  
m_freem(m);
```

Example: tunwrite builds mbuf; tunread extracts to userspace

6.16.7 Common Patterns From the Tours

Pattern: Shared cdevsw, per-device state via si_drv1

- One function table, many device instances
- Example: led.c shares led_cdevsw across all LEDs
- State accessed via `sc = dev->si_drv1`

Pattern: Subsystem providing both APIs

- Userspace interface (character device)
- Kernel API (function calls)
- Example: led.c's `led_write()` vs. `led_set()`

Pattern: Timer-driven state machine

- Reference counter tracks active items
- Timer reschedules only when work remains
- Example: led.c's `blinkers` counter gates timer

Pattern: Two-phase cleanup

- Phase 1: Make invisible (clear pointers, remove from lists)
- Phase 2: Free resources
- Example: `led_destroy` clears `si_drv1` before destroying device

Pattern: Unit number allocation

- Use `unrhdr` for dynamic assignment
- Prevents conflicts in multi-instance devices
- Example: led.c's `led_unit` pool

6.16.8 Errors, Edge Cases, and User Experience

Error handling:

- Prefer clear errno over silent behavior unless silence is part of the contract
- Example: tunwrite silently ignores writes when interface down (expected behavior)
- Example: led_write returns EINVAL for bad commands (error condition)

Bound inputs:

- Always validate sizes, counts, indices
- Example: led_write rejects commands over 512 bytes
- Example: tuntap checks against MRU + headers

Default to fail fast:

- Unsupported ioctl -> ENOIOCTL
- Invalid flags -> EINVAL
- Malformed frames -> drop and increment error counter

Module unload:

- Think about impact on active users
- Don't yank foundational devices from busy systems
- Example: null.c can be unloaded; led.c cannot (no unload handler)

6.16.9 Minimal Templates

6.16.9.1 Character Device (Read/Write/Ioctl Only)

```
static d_read_t  foo_read;
static d_write_t foo_write;
static d_ioctl_t foo_ioctl;

static struct cdevsw foo_cdevsw = {
    .d_version = D_VERSION,
    .d_read    = foo_read,
    .d_write   = foo_write,
    .d_ioctl   = foo_ioctl,
    .d_name    = "foo",
};

static struct cdev *foo_dev;

static int
foo_read(struct cdev *dev, struct uio *uio, int flags)
{
```

```

    while (uio->uio_resid > 0) {
        size_t n = MIN(uio->uio_resid, CHUNK);
        int err = uiomove(srcbuf, n, uio);
        if (err) return err;
    }
    return 0;
}

static int
foo_write(struct cdev *dev, struct uio *uio, int flags)
{
    /* Consume all (bit bucket pattern) */
    uio->uio_resid = 0;
    return 0;
}

static int
foo_ioctl(struct cdev *dev, u_long cmd, caddr_t data,
          int fflag, struct thread *td)
{
    switch (cmd) {
    case FIONBIO:
        return 0; /* Non-blocking always OK */
    default:
        return ENOIOCTL;
    }
}

```

6.16.9.2 Dynamic Device Registration

```

static struct unrhdr *foo_units;
static struct mtx foo_mtx;
static LIST_HEAD(, foo_softc) foo_list;

struct cdev *
foo_create(void *priv, const char *name)
{
    struct foo_softc *sc;

    sc = malloc(sizeof(*sc), M_FOO, M_WAITOK | M_ZERO);
    sc->unit = alloc_unr(foo_units);
    sc->private = priv;
}

```

```

    sc->dev = make_dev(&foo_cdevsw, sc->unit,
        UID_ROOT, GID_WHEEL, 0600, "foo/%s", name);
    sc->dev->si_drv1 = sc;

    mtx_lock(&foo_mtx);
    LIST_INSERT_HEAD(&foo_list, sc, list);
    mtx_unlock(&foo_mtx);

    return sc->dev;
}

void
foo_destroy(struct cdev *dev)
{
    struct foo_softc *sc;

    mtx_lock(&foo_mtx);
    sc = dev->si_drv1;
    dev->si_drv1 = NULL;
    LIST_REMOVE(sc, list);
    mtx_unlock(&foo_mtx);

    free_unr(foo_units, sc->unit);
    destroy_dev(dev);
    free(sc, M_F00);
}

```

6.16.9.3 PCI Glue (Probe/Attach/Detach)

```

static int foo_probe(device_t dev)
{
    /* Table match -> pick class */
    id = foo_pci_match(dev, foo_ids);
    if (id == NULL)
        return ENXIO;

    sc->sc_class = &foo_device_class;
    return foo_bus_probe(dev, id->regshift, id->rclk,
        id->rid & RID_MASK);
}

static int foo_attach(device_t dev)
{

```

```

    /* Maybe allocate single-vector MSI */
    if (pci_msi_count(dev) == 1) {
        count = 1;
        if (pci_alloc_msi(dev, &count) == 0)
            sc->sc_irid = 1;
    }
    return foo_bus_attach(dev);
}

static int foo_detach(device_t dev)
{
    /* Release MSI if used */
    if (sc->sc_irid != 0)
        pci_release_msi(dev);

    return foo_bus_detach(dev);
}

static device_method_t foo_methods[] = {
    DEVMETHOD(device_probe,  foo_probe),
    DEVMETHOD(device_attach, foo_attach),
    DEVMETHOD(device_detach, foo_detach),
    DEVMETHOD_END
};

static driver_t foo_driver = {
    "foo",
    foo_methods,
    sizeof(struct foo_softc)
};

DRIVER_MODULE(foo, pci, foo_driver, NULL, NULL);

```

6.16.10 Pre-Lab Self-Check (2 Minutes)

Ask yourself these questions before writing code:

1. Which integration point am I targeting (devfs, ifnet, PCI)?
2. Do I know my entry points and what each must return on success/failure?
3. What are my locks and which contexts touch each field?
4. Can I list every resource I allocate and where I free it on:
 - Success path

- Mid-attach failure
- Detach/unload

5. Have I studied a similar driver from the tours?

- `null.c` for simple character devices
- `led.c` for dynamic devices and timers
- `tuntap` for network integration
- `uart_pci` for hardware attachment

6.16.11 After-Lab Reflection (5 minutes)

After writing or modifying code, verify:

1. Did I leak anything on an early return?
2. Did I block in a context that shouldn't sleep?
3. Did I notify userspace/kernel peers after enqueueing work?
4. Can I point from a user-visible behavior back to the specific source lines?
5. Does my locking follow a consistent hierarchy?
6. Are my error messages helpful for debugging?

6.16.12 Common Pitfalls and How to Avoid Them

This section catalogs the mistakes that cause the most pain in driver development, silent corruption, deadlocks, panics, and resource leaks. Each pitfall includes the symptom, the root cause, and the correct pattern to follow.

6.16.12.1 Data Movement Errors

6.16.12.1.1 Pitfall: Forgetting to update `uio_resid` **Symptom:** Infinite loops in read/write handlers, or userspace receiving wrong byte counts.

Root cause: The kernel uses `uio_resid` to track remaining bytes. If you don't decrement it, the kernel thinks no progress was made.

Wrong:

```
static int
bad_write(struct cdev *dev, struct uio *uio, int flags)
{
    /* Data is discarded but uio_resid never changes! */
    return 0; /* Kernel sees 0 bytes written, retries infinitely */
}
```

Correct:

```
static int
good_write(struct cdev *dev, struct uio *uio, int flags)
{
    uio->uio_resid = 0; /* Mark all bytes consumed */
    return 0;
}
```

How to avoid: Always ask “how many bytes did I actually process?” and update `uio_resid` accordingly. Even if you discard data (like `/dev/null`), you must mark it consumed.

Related: Partial transfers are dangerous. If you process some bytes but then fail, you must update `uio_resid` to reflect what was actually transferred before returning the error, or userspace will retry with the wrong offset.

6.16.12.1.2 Pitfall: Not bounding chunk sizes in `uiomove()` loops **Symptom:** Stack overflow if copying to stack buffer, kernel panic on huge allocations.

Root cause: User requests can be arbitrarily large. Copying multi-megabyte transfers in one shot exhausts resources.

Wrong:

```
static int
bad_read(struct cdev *dev, struct uio *uio, int flags)
{
    char buf[uio->uio_resid]; /* Stack overflow if user requests 1MB! */
    memset(buf, 0, sizeof(buf));
    return uiomove(buf, uio->uio_resid, uio);
}
```

Correct:

```
#define CHUNK_SIZE 4096

static int
good_read(struct cdev *dev, struct uio *uio, int flags)
{
    char buf[CHUNK_SIZE];
    int error;

    memset(buf, 0, sizeof(buf));

    while (uio->uio_resid > 0) {
        size_t len = MIN(uio->uio_resid, CHUNK_SIZE);
        error = uiomove(buf, len, uio);
        if (error)
```

```
        return error;
    }
    return 0;
}
```

How to avoid: Always loop with a reasonable chunk size (typically 4KB-64KB). Study `zero_read` in `null.c`, it limits transfers to `ZERO_REGION_SIZE` per iteration.

6.16.12.1.3 Pitfall: Accessing user memory directly from kernel **Symptom:** Security vulnerabilities, kernel crashes on invalid pointers.

Root cause: Kernel and user memory spaces are separate. Dereferencing user pointers directly bypasses protection.

Wrong:

```
static int
bad_ioctl(struct cdev *dev, u_long cmd, caddr_t data, int flag, struct thread *td)
{
    char *user_ptr = *(char **)data;
    strcpy(kernel_buf, user_ptr); /* DANGER: user_ptr not validated! */
}
```

Correct:

```
static int
good_ioctl(struct cdev *dev, u_long cmd, caddr_t data, int flag, struct thread *td)
{
    char *user_ptr = *(char **)data;
    char kernel_buf[256];
    int error;

    error = copyinstr(user_ptr, kernel_buf, sizeof(kernel_buf), NULL);
    if (error)
        return error;
    /* Now safe to use kernel_buf */
}
```

How to avoid: Never dereference pointers received from userspace. Use `copyin()`, `copyout()`, `copyinstr()`, or `uiomove()` for all user <-> kernel transfers. These functions validate addresses and handle page faults safely.

6.16.12.2 Locking Disasters

6.16.12.2.1 Pitfall: Holding locks across uiomove() **Symptom:** System deadlock when user memory is paged out.

Root cause: uiomove() can page fault, which may need to acquire VM locks. If you hold another lock during the fault, and that lock is needed by the paging path, deadlock results.

Wrong:

```
static int
bad_read(struct cdev *dev, struct uio *uio, int flags)
{
    mtx_lock(&my_mtx);
    /* Build response in kernel buffer */
    uiomove(kernel_buf, len, uio); /* DEADLOCK RISK: uiomove while locked */
    mtx_unlock(&my_mtx);
    return 0;
}
```

Correct:

```
static int
good_read(struct cdev *dev, struct uio *uio, int flags)
{
    char *local_buf;
    size_t len;

    mtx_lock(&my_mtx);
    /* Copy data to private buffer while locked */
    len = MIN(uio->uio_resid, bufsize);
    local_buf = malloc(len, M_TEMP, M_WAITOK);
    memcpy(local_buf, sc->data, len);
    mtx_unlock(&my_mtx);

    /* Transfer to user without holding lock */
    error = uiomove(local_buf, len, uio);
    free(local_buf, M_TEMP);
    return error;
}
```

How to avoid: Always release locks before uiomove(), copyin(), copyout(). Snapshot the data you need while locked, then transfer it to userspace unlocked.

Exception: Some sleep-capable locks (sx locks with SX_DUPOK) can be held across user memory access if carefully designed, but mutexes never can.

6.16.12.2.2 Pitfall: Inconsistent lock ordering **Symptom:** Deadlock when two threads acquire the same locks in opposite orders.

Root cause: Lock ordering violations create circular wait conditions.

Wrong:

```
/* Thread A */
mtx_lock(&lock_a);
mtx_lock(&lock_b); /* Order: A then B */

/* Thread B */
mtx_lock(&lock_b);
mtx_lock(&lock_a); /* Order: B then A - DEADLOCK! */
```

Correct:

```
/* Establish hierarchy: always lock_a before lock_b */

/* Thread A */
mtx_lock(&lock_a);
mtx_lock(&lock_b);

/* Thread B */
mtx_lock(&lock_a); /* Same order everywhere */
mtx_lock(&lock_b);
```

How to avoid:

1. Document your lock hierarchy in comments at the top of the file
2. Always acquire locks in the same order throughout the driver
3. Use WITNESS kernel option during development to detect violations
4. Study `led.c`: it acquires `led_mtx` first, releases it, then acquires `led_sx`, never holds both simultaneously

6.16.12.2.3 Pitfall: Forgetting to initialize locks **Symptom:** Kernel panic with “lock not initialized” or immediate hang on first lock acquisition.

Root cause: Lock structures must be explicitly initialized before use.

Wrong:

```
static struct mtx my_lock; /* Declared but not initialized */

static int
foo_attach(device_t dev)
{
```

```
mtx_lock(&my_lock); /* PANIC: uninitialized lock */
}
```

Correct:

```
static struct mtx my_lock;

static void
foo_init(void)
{
    mtx_init(&my_lock, "my lock", NULL, MTX_DEF);
}

SYSINIT(foo, SI_SUB_DRIVERS, SI_ORDER_FIRST, foo_init, NULL);
```

How to avoid:

- Initialize locks in module load handler, SYSINIT, or attach function
- Use `mtx_init()`, `sx_init()`, `rw_init()` as appropriate
- For callouts: `callout_init_mtx()` associates timer with lock
- Study `led.c`'s `led_drvinit()`: initializes all locks before any devices are created

6.16.12.2.4 Pitfall: Destroying locks while threads still hold them **Symptom:** Kernel panic during module unload or device detach.

Root cause: Lock structures must remain valid until all users are done.

Wrong:

```
static int
bad_detach(device_t dev)
{
    mtx_destroy(&sc->mtx); /* Destroy lock */
    destroy_dev(sc->dev);  /* But device write handler may still run! */
    return 0;
}
```

Correct:

```
static int
good_detach(device_t dev)
{
    destroy_dev(sc->dev); /* Wait for all users to finish */
    /* Now safe - no threads can be in device operations */
}
```

```
mtx_destroy(&sc->mtx);
return 0;
}
```

How to avoid:

- `destroy_dev()` blocks until all open file descriptors close and in-progress operations complete
- Destroy locks only after devices/resources are gone
- For global locks: destroy in module unload or never (if module can't unload)

6.16.12.3 Resource Management Failures

6.16.12.3.1 Pitfall: Leaking resources on error paths **Symptom:** Memory leaks, device node leaks, eventual resource exhaustion.

Root cause: Early returns skip cleanup code.

Wrong:

```
static int
bad_attach(device_t dev)
{
    sc = malloc(sizeof(*sc), M_DEV, M_WAITOK);

    sc->res = bus_alloc_resource_any(dev, SYS_RES_MEMORY, &rid, RF_ACTIVE);
    if (sc->res == NULL)
        return ENXIO; /* LEAK: sc not freed! */

    error = setup_irq(dev);
    if (error)
        return error; /* LEAK: sc and sc->res not freed! */

    return 0;
}
```

Correct:

```
static int
good_attach(device_t dev)
{
    sc = malloc(sizeof(*sc), M_DEV, M_WAITOK | M_ZERO);

    sc->res = bus_alloc_resource_any(dev, SYS_RES_MEMORY, &rid, RF_ACTIVE);
    if (sc->res == NULL) {
        error = ENXIO;
    }
}
```

```

        goto fail;
    }

    error = setup_irq(dev);
    if (error)
        goto fail;

    return 0;

fail:
    if (sc->res != NULL)
        bus_release_resource(dev, SYS_RES_MEMORY, rid, sc->res);
    free(sc, M_DEV);
    return error;
}

```

How to avoid:

- Use a single fail: label at the end of the function
- Check which resources were allocated and free only those
- Initialize pointers to NULL so you can check them
- Consider: every malloc() needs a free(), every make_dev() needs a destroy_dev()

6.16.12.3.2 Pitfall: Use-after-free in concurrent cleanup **Symptom:** Kernel panic with “page fault in kernel mode”, often intermittent.

Root cause: One thread frees memory while another thread still accesses it.

Wrong:

```

void
bad_destroy(struct cdev *dev)
{
    struct foo_softc *sc = dev->si_drv1;

    free(sc, M_F00);           /* Free immediately */
    /* Another thread's foo_write may still be using sc! */
}

```

Correct:

```

void
good_destroy(struct cdev *dev)
{
    struct foo_softc *sc;

```

```

mtx_lock(&foo_mtx);
sc = dev->si_drv1;
dev->si_drv1 = NULL;      /* Break link first */
LIST_REMOVE(sc, list);    /* Remove from searchable lists */
mtx_unlock(&foo_mtx);

destroy_dev(dev);          /* Wait for operations to drain */

/* Now safe - no threads can find or access sc */
free(sc, M_F00);
}

```

How to avoid:

- Make objects invisible before freeing (clear pointers, remove from lists)
- Use `destroy_dev()` which waits for in-progress operations
- Study `led_destroy`: clears `si_drv1` first, removes from list, then frees

6.16.12.3.3 Pitfall: Not checking for allocation failures with `M_NOWAIT` **Symptom:** Kernel panic dereferencing NULL pointer.

Root cause: `M_NOWAIT` allocations can fail, but code assumes success.

Wrong:

```

static int
bad_write(struct cdev *dev, struct uio *uio, int flags)
{
    char *buf = malloc(uio->uio_resid, M_TEMP, M_NOWAIT);
    /* PANIC if malloc returns NULL and we dereference buf! */
    uiomove(buf, uio->uio_resid, uio);
    free(buf, M_TEMP);
}

```

Correct:

```

static int
good_write(struct cdev *dev, struct uio *uio, int flags)
{
    char *buf = malloc(uio->uio_resid, M_TEMP, M_NOWAIT);
    if (buf == NULL)
        return ENOMEM;

    error = uiomove(buf, uio->uio_resid, uio);
}

```

```
    free(buf, M_TEMP);
    return error;
}
```

Better: Use M_WAITOK when safe:

```
static int
better_write(struct cdev *dev, struct uio *uio, int flags)
{
    /* M_WAITOK can sleep but never returns NULL */
    char *buf = malloc(uio->uio_resid, M_TEMP, M_WAITOK);
    error = uiomove(buf, uio->uio_resid, uio);
    free(buf, M_TEMP);
    return error;
}
```

How to avoid:

- Use M_WAITOK unless in interrupt context or holding spin locks
- Always check M_NOWAIT allocations for NULL
- Study `led_write`: uses M_WAITOK since write operations can sleep

6.16.12.4 Timer and Asynchronous Operation Errors

6.16.12.4.1 Pitfall: Timer callback accessing freed memory **Symptom:** Panic in timer callback, memory corruption.

Root cause: Device destroyed but timer still scheduled.

Wrong:

```
void
bad_destroy(struct cdev *dev)
{
    struct foo_softc *sc = dev->si_drv1;

    destroy_dev(dev);
    free(sc, M_F00);           /* Free softc */
    /* Timer may fire and access sc! */
}
```

Correct:

```
void
good_destroy(struct cdev *dev)
{
    struct foo_softc *sc = dev->si_drv1;

    callout_drain(&sc->callout); /* Wait for callback to complete */
    destroy_dev(dev);
    free(sc, M_F00);             /* Now safe */
}
```

How to avoid:

- Use `callout_drain()` before freeing structures accessed by callback
- Or use `callout_stop()` and ensure no callback is running
- Initialize callouts with `callout_init_mtx()` to automatically hold your lock
- Study `led_destroy`: stops timer when list becomes empty

6.16.12.4.2 Pitfall: Timer rescheduling unconditionally **Symptom:** CPU waste, system slowdown, unnecessary wakeups.

Root cause: Timer fires even when there's no work to do.

Wrong:

```
static void
bad_timeout(void *arg)
{
    /* Process items */
    LIST_FOREACH(item, &list, entries) {
        if (item->active)
            process_item(item);
    }

    /* Always reschedule - wastes CPU even when list empty! */
    callout_reset(&timer, hz / 10, bad_timeout, arg);
}
```

Correct:

```
static void
good_timeout(void *arg)
{
    int active_count = 0;

    LIST_FOREACH(item, &list, entries) {
```



```

        if (item->active) {
            process_item(item);
            active_count++;
        }
    }

    /* Only reschedule if there's work */
    if (active_count > 0)
        callout_reset(&timer, hz / 10, good_timeout, arg);
}

```

How to avoid:

- Maintain a counter of items needing service
- Only schedule timer when counter > 0
- Study led.c: blinkers counter gates timer rescheduling

6.16.12.5 Network Driver Specific Issues

6.16.12.5.1 Pitfall: Not freeing mbufs on error paths **Symptom:** mbuf exhaustion, “network buffers exhausted” messages.

Root cause: Mbufs are a limited resource that must be explicitly freed.

Wrong:

```

static int
bad_transmit(struct ifnet *ifp, struct mbuf *m)
{
    if (validate_packet(m) < 0)
        return EINVAL; /* LEAK: m not freed! */

    if (queue_full())
        return ENOBUFS; /* LEAK: m not freed! */

    enqueue_packet(m);
    return 0;
}

```

Correct:

```

static int
good_transmit(struct ifnet *ifp, struct mbuf *m)
{
    if (validate_packet(m) < 0) {

```

```

        m_freem(m);
        return EINVAL;
    }

    if (queue_full()) {
        m_freem(m);
        return ENOBUFS;
    }

    enqueue_packet(m); /* Queue now owns mbuf */
    return 0;
}

```

How to avoid:

- Whoever has the mbuf pointer is responsible for freeing it
- On error: `m_freem(m)` before returning
- On success: ensure someone else took ownership (queued, transmitted, etc.)

6.16.12.5.2 Pitfall: Forgetting to notify blocked readers/writers **Symptom:** Processes hang in read/write/poll even though data is available.

Root cause: Data arrives but waiters aren't woken.

Wrong:

```

static void
bad_rx_handler(struct foo_softc *sc, struct mbuf *m)
{
    TAILQ_INSERT_TAIL(&sc->rxq, m, list);
    /* Reader blocked in read() never wakes up! */
}

```

Correct:

```

static void
good_rx_handler(struct foo_softc *sc, struct mbuf *m)
{
    TAILQ_INSERT_TAIL(&sc->rxq, m, list);

    /* Triple notification pattern */
    wakeup(sc); /* Wake sleeping threads */
    selwakeuppri(&sc->rsl, PZERO + 1); /* Wake poll/select */
    KNOTE_LOCKED(&sc->rsl.si_note, 0); /* Wake kqueue */
}

```

How to avoid:

- After enqueueing data: call `wakeup()`, `selwakeuppri()`, `KNOTE()`
- Study `tunstart` in `if_tun.c`: triple notification pattern
- For write: notify after dequeueing (when space becomes available)

6.16.12.6 Input Validation Failures

6.16.12.6.1 Pitfall: Not bounding input sizes **Symptom:** Denial of service, kernel memory exhaustion.

Root cause: Attacker can request huge allocations or cause huge copies.

Wrong:

```
static int
bad_write(struct cdev *dev, struct uio *uio, int flags)
{
    char *buf = malloc(uio->uio_resid, M_TEMP, M_WAITOK);
    /* Attacker writes 1GB, kernel allocates 1GB! */
    uiomove(buf, uio->uio_resid, uio);
    process(buf);
    free(buf, M_TEMP);
}
```

Correct:

```
#define MAX_CMD_SIZE 4096

static int
good_write(struct cdev *dev, struct uio *uio, int flags)
{
    char *buf;

    if (uio->uio_resid > MAX_CMD_SIZE)
        return EINVAL; /* Reject excessive requests */

    buf = malloc(uio->uio_resid, M_TEMP, M_WAITOK);
    uiomove(buf, uio->uio_resid, uio);
    process(buf);
    free(buf, M_TEMP);
}
```

How to avoid:

- Define maximum sizes for all inputs (commands, packets, buffers)
- Check limits before allocation
- Study `led_write`: rejects commands over 512 bytes

6.16.12.6.2 Pitfall: Trusting user-provided lengths and offsets **Symptom:** Buffer overruns, reading uninitialized memory, information leaks.

Root cause: User controls length fields in ioctl structures.

Wrong:

```
struct user_request {
    void *buf;
    size_t len;
};

static int
bad_ioctl(struct cdev *dev, u_long cmd, caddr_t data, int flag, struct thread *td)
{
    struct user_request *req = (struct user_request *)data;
    char kernel_buf[256];

    /* User can set len > 256! */
    copyin(req->buf, kernel_buf, req->len); /* Buffer overrun! */
}
```

Correct:

```
static int
good_ioctl(struct cdev *dev, u_long cmd, caddr_t data, int flag, struct thread *td)
{
    struct user_request *req = (struct user_request *)data;
    char kernel_buf[256];

    if (req->len > sizeof(kernel_buf))
        return EINVAL;

    return copyin(req->buf, kernel_buf, req->len);
}
```

How to avoid:

- Validate all length fields against buffer sizes
- Validate offsets are within valid ranges
- Use MIN() to cap lengths: len = MIN(user_len, MAX_LEN)

6.16.12.7 Race Conditions and Timing Issues

6.16.12.7.1 Pitfall: Check-then-use races **Symptom:** Intermittent crashes, security vulnerabilities (TOC-TOU bugs).

Root cause: State changes between check and use.

Wrong:

```
static int
bad_write(struct cdev *dev, struct uio *uio, int flags)
{
    struct foo_softc *sc = dev->si_drv1;

    if (sc == NULL)           /* Check */
        return ENXIO;

    /* Another thread destroys device here! */

    process_data(sc->buf);    /* Use - sc may be freed! */
}
```

Correct:

```
static int
good_write(struct cdev *dev, struct uio *uio, int flags)
{
    struct foo_softc *sc;
    int error;

    mtx_lock(&foo_mtx);
    sc = dev->si_drv1;
    if (sc == NULL) {
        mtx_unlock(&foo_mtx);
        return ENXIO;
    }

    /* Process while holding lock */
    error = process_data_locked(sc->buf);
    mtx_unlock(&foo_mtx);
    return error;
}
```

How to avoid:

- Hold appropriate lock from check through use
- Make checks and uses atomic with respect to each other
- Or use reference counting to keep objects alive

6.16.12.7.2 Pitfall: Missing memory barriers on lock-free code **Symptom:** Rare corruption on multi-core systems, works fine in single-core.

Root cause: CPU reordering of memory operations.

Wrong:

```
/* Producer */
sc->data = new_value;    /* Write data */
sc->ready = 1;           /* Set flag - may be reordered before data write! */

/* Consumer */
if (sc->ready)           /* Check flag */
    use(sc->data);        /* May see old data! */
```

Correct with explicit barriers:

```
/* Producer */
sc->data = new_value;
atomic_store_rel_int(&sc->ready, 1); /* Release barrier */

/* Consumer */
if (atomic_load_acq_int(&sc->ready)) /* Acquire barrier */
    use(sc->data);
```

Better: Just use locks:

```
/* Much simpler and correct */
mtx_lock(&sc->mtx);
sc->data = new_value;
sc->ready = 1;
mtx_unlock(&sc->mtx);
```

How to avoid:

- Avoid lock-free programming unless you're an expert
- Use locks for correctness, optimize only if profiling shows need
- If you must go lock-free: use atomic operations with explicit barriers

6.16.12.8 Module Lifecycle Issues

6.16.12.8.1 Pitfall: Device operations racing with module unload **Symptom:** Crash during `kldunload`, jumps to invalid memory.

Root cause: Functions unloaded while still in use.

Wrong:

```
static int
bad_unload(module_t mod, int type, void *data)
{
    switch(type) {
    case MOD_UNLOAD:
        destroy_dev(my_dev);
        return 0; /* Module text may be unloaded while write() in progress! */
    }
}
```

Correct:

```
static int
good_unload(module_t mod, int type, void *data)
{
    switch(type) {
    case MOD_UNLOAD:
        /* destroy_dev() waits for all operations to complete */
        destroy_dev(my_dev);
        /* Now safe - no code paths reference module functions */
        return 0;
    }
}
```

How to avoid:

- `destroy_dev()` automatically prevents this by waiting
- For infrastructure modules (like `led.c`): don't provide unload handler
- Test unload under load: `while true; do cat /dev/foo; done & sleep 1; kldunload foo`

6.16.12.8.2 Pitfall: Unload leaving dangling references **Symptom:** Crashes in seemingly unrelated code after module unload.

Root cause: Other code holds pointers to unloaded module's data/functions.

Wrong:

```
/* Your module */
void my_callback(void *arg) { /* ... */ }

static int
bad_load(module_t mod, int type, void *data)
{
    register_callback(my_callback); /* Register with another subsystem */
    return 0;
}
```

```
}

static int
bad_unload(module_t mod, int type, void *data)
{
    return 0; /* Forgot to unregister - subsystem will call invalid function! */
}
```

Correct:

```
static int
good_unload(module_t mod, int type, void *data)
{
    unregister_callback(my_callback); /* Clean up registrations */
    /* Wait for any in-progress callbacks to complete */
    return 0;
}
```

How to avoid:

- Every registration needs corresponding deregistration
- Every callback installation needs removal
- Every “register with subsystem” needs “unregister from subsystem”

6.16.13 Debugging Pitfall Patterns

6.16.13.1 How to detect these bugs:

For locking issues:

```
# In kernel config or loader.conf
options WITNESS
options WITNESS_SKIPSPIN
options INVARIANTS
options INVARIANT_SUPPORT
```

WITNESS detects lock order violations and reports them in dmesg.

For memory issues:

```
# Track allocations
vmstat -m | grep M_YOURTYPE

# Enable kernel malloc debugging
options MALLOC_DEBUG_MAXZONES=8
```


For race conditions:

- Run stress tests on multi-core systems
- Use stress2 test suite
- Concurrent operations: multiple threads opening/closing/reading/writing

For leak detection:

- Before load: note resource counts (`vmstat -m, devfs, ifconfig -a`)
- Load module, exercise it heavily
- Unload module
- Check resource counts - should return to baseline

6.16.14 Prevention Checklist

Before committing code, verify:

Data movement

- All `uiomove()` calls properly update `uio_resid`
- Chunk sizes bounded to reasonable limits
- No direct dereferencing of user pointers

Locking

- No locks held across `uiomove()/copyin()/copyout()`
- Consistent lock ordering documented and followed
- All locks initialized before use
- Locks destroyed only after last user done

Resources

- Every allocation has matching free on all paths
- Error paths tested and leak-free
- Objects made invisible before freeing
- NULL checks after `M_NOWAIT` allocations

Timers

- `callout_drain()` before freeing structures
- Timer rescheduling gated by work counter
- Callout initialized with associated mutex

Network (if applicable)

- Mbufs freed on all error paths
- Triple notification after enqueue
- Input sizes validated against MRU

Input validation

- Maximum sizes defined and enforced

- User-provided lengths checked
- Offsets validated before use

Races

- No check-then-use patterns without locks
- Critical sections properly protected
- Lock-free code avoided unless necessary

Lifecycle

- `destroy_dev()` before freeing softc
- All registrations have deregistrations
- Unload tested under concurrent use

6.16.15 When Things Go Wrong

If you see “sleeping with lock held”:

- Likely holding mutex across `uiomove()` or allocation with `M_WAITOK`
- Solution: Release lock before blocking operation

If you see “lock order reversal”:

- Two locks acquired in different orders in different code paths
- Solution: Establish and document hierarchy, fix violating code

If you see “page fault in kernel mode”:

- Usually use-after-free or NULL dereference
- Check: Are you accessing memory after freeing? Is `si_drv1` cleared first?

If processes hang forever:

- Missing `wakeup()` or notification
- Check: Does every enqueue call `wakeup/selwakeup/KNOTE`?

If resources leak:

- Error path missing cleanup
- Check: Does every early return free what was allocated?

6.16.16 You’re Ready: From Patterns to Practice

By studying these pitfalls and their solutions in the context of the four driver tours, you develop the instincts to avoid them. The patterns repeat: check before use, lock appropriately, free what you allocate, notify when you enqueue, validate user input. Master these, and your drivers will be robust.

You now have a compact mental model: the same few patterns repeat with different applications. Keep this blueprint open while you tackle hands-on labs, it’s the shortest path from “I think I get it” to “I can ship a driver that behaves correctly.”

When in doubt, return to the four driver tours. They're your worked examples showing these patterns in complete, working code.

Next, it's time to get hands-on with four practical labs.

6.17 Hands-On Labs: From Reading to Building (Beginner-Safe)

You've read about driver structure; now **experience it**. These four carefully designed labs take you from reading code to building working kernel modules, each one validating your understanding before moving forward.

6.17.1 Lab Design Philosophy

These labs are:

- **Safe:** Run in your lab VM, isolated from your main system
- **Incremental:** Each builds on the last with clear checkpoints
- **Self-validating:** You'll know immediately if you've succeeded
- **Explanatory:** Code includes comments explaining the "why" behind the "what"
- **Complete:** All code is tested on FreeBSD 14.3 and ready to use

6.17.2 Prerequisites for All Labs

Before starting, ensure you have:

1. **FreeBSD 14.3** running (VM or physical machine)
2. **Source code installed:** `/usr/src` must exist

```
# If /usr/src is missing, install it:
% sudo pkg install git
% sudo git clone --branch releng/14.3 --depth 1 https://git.FreeBSD.org/src.git
↪ src /usr/src
```

3. **Build tools installed:**

```
% sudo pkg install llvm
```

4. **Root access** via `sudo` or `su`
5. **Your lab logbook** for notes and observations

6.17.3 Time Commitment

- **Lab 1** (Scavenger Hunt): 30-40 minutes
- **Lab 2** (Hello Module): 40-50 minutes
- **Lab 3** (Device Node): 60-75 minutes

- **Lab 4** (Error Handling): 30-40 minutes

Total: 2.5 - 3.5 hours for all labs

Recommendation: Complete Lab 1 and Lab 2 in one session, take a break, then tackle Lab 3 and 4 in a second session.

6.18 Lab 1: Explore the Driver Map (Read-Only Scavenger Hunt)

6.18.1 Goal

Locate and identify key driver structures in real FreeBSD source code. Build navigation confidence and pattern recognition skills.

6.18.2 What You'll Learn

- How to find and read FreeBSD driver source files
- How to recognize common patterns (cdevsw, probe/attach, DRIVER_MODULE)
- Where different types of drivers live in the source tree
- How to use `less` and `grep` effectively for driver exploration

6.18.3 Prerequisites

- FreeBSD 14.3 with `/usr/src` installed
- Text editor or `less` for viewing files
- Terminal with your favorite shell

6.18.4 Time Estimate

30-40 minutes (questions only)

+10 minutes if you want to explore beyond the questions

6.18.5 Instructions

6.18.5.1 Part 1: Character Device Driver - The Null Driver

Step 1: Navigate to the null driver

```
% cd /usr/src/sys/dev/null
% ls -l
total 8
-rw-r--r--  1 root  wheel  4127  Oct 14 10:15 null.c
```

Step 2: Open the file with `less`

```
% less null.c
```

Navigation tips for less:

- Press / to search (example: /cdevsw to find cdevsw structure)
- Press n to find next occurrence
- Press q to quit
- Press g to go to top, G to go to bottom

Step 3: Answer these questions (write in your lab logbook):

Q1: What line number defines the `null_cdevsw` structure?

Hint: Search for `/cdevsw` in `less`

Q2: Which function handles writes to `/dev/null`?

Hint: Look at the `.d_write =` line in the `cdevsw` structure

Q3: What does the write function return?

Hint: Look at the function implementation

Q4: Where is the module event handler? What's its name?

Hint: Search for `modevent`

Q5: What macro registers the module with the kernel?

Hint: Look near the end of the file, search for `DECLARE_MODULE`

Q6: How many device nodes does this module create in `/dev`?

Hint: Count the `make_dev_credf` calls in the load handler

Q7: What are the device node names?

Hint: Look at the last parameter in each `make_dev_credf` call

6.18.5.2 Part 2: Infrastructure Driver - The LED Driver

Step 4: Navigate to the LED driver

```
% cd /usr/src/sys/dev/led
```

```
% less led.c
```

Step 5: Answer these questions:

Q8: Find the `softc` structure. What's it called?

Hint: Search for `_softc {` to find structure definitions

Q9: Where is `led_create()` defined?

Hint: Search for `^led_create` (^ means start of line)

Q10: What subdirectory in `/dev` do LED device nodes appear under?

Hint: Look at the `make_dev` call in `led_create()`, check the path

Q11: Find the `led_write` function. What does it do with user input?

Hint: Look for the function definition, read the code

Q12: Is there a probe/attach pair, or does this use a module event handler?

Hint: Search for probe and attach vs modevent

Q13: Can you find where the driver allocates memory for the softc?

Hint: Look in led_create() for malloc calls

6.18.5.3 Part 3: Network Driver - The Tun/Tap Driver

Step 6: Navigate to the tun/tap driver

```
% cd /usr/src/sys/net
% less if_tuntap.c
```

Note: This is a larger, more complex driver. Don't try to understand everything, just find the specific patterns.

Step 7: Answer these questions:

Q14: Find the softc structure for tun. What's it called?

Hint: Search for tun_softc {

Q15: Does the softc contain both a struct cdev * and network interface pointer?

Hint: Look at the members of the softc structure

Q16: Where is the tun_cdevsw structure defined?

Hint: Search for tun_cdevsw =

Q17: What function is called when you open /dev/tun?

Hint: Look at the .d_open = line in the cdevsw

Q18: Where does the driver create the network interface?

Hint: Search for if_alloc in the source

6.18.5.4 Part 4: Bus-Attached Driver - A PCI UART

Step 8: Navigate to a PCI driver

```
% cd /usr/src/sys/dev/uart
% less uart_bus_pci.c
```

Step 9: Answer these questions:

Q19: Find the probe function. What's it called?

Hint: Look for a function ending in _probe

Q20: What does the probe function check to identify compatible hardware?

Hint: Look inside the probe function for ID comparisons

Q21: Where is DRIVER_MODULE declared?

Hint: Search for DRIVER_MODULE - should be near end of file

Q22: What bus does this driver attach to?

Hint: Look at the second parameter to DRIVER_MODULE macro

Q23: Find the device method table. What's it called?

Hint: Search for device_method_t - should be an array

Q24: How many methods are defined in the method table?

Hint: Count entries between the declaration and DEVMETHOD_END

6.18.6 Check Your Answers

After completing all questions, compare with the answer key below. Don't peek before attempting!

6.18.6.1 Part 1: Null Driver

A1: The null_cdevsw definition (the character-device switch table for /dev/null)

A2: null_write function

A3: Sets uio->uio_resid = 0 to mark all bytes consumed, then returns 0 (success). The data is discarded.

A4: null_modevent(), defined near the bottom of null.c just before the DEV_MODULE registration

A5: DEV_MODULE(null, null_modevent, NULL); followed by MODULE_VERSION(null, 1);

A6: Three device nodes: /dev/null, /dev/zero and /dev/full

A7: "null", "zero", "full"

6.18.6.2 Part 2: LED Driver

A8: struct ledsc (note the compact "LED softc" name; not led_softc)

A9: led_create() is a thin wrapper around led_create_state(); both live together in led.c, just after the led_cdevsw definition

A10: /dev/led/ (LEDs appear as /dev/led/name, created with make_dev(..., "led/%s", name))

A11: led_write() reads the user's buffer via uiomove(), passes it through led_parse() to turn a human-readable string like "f3" or "m-.-" into a compact pattern, then installs the pattern with led_state().

A12: Neither. led.c is an infrastructure subsystem (no probe/attach, no module event handler). It initializes at boot via SYSINIT(leddev, SI_SUB_DRIVERS, SI_ORDER_MIDDLE, led_drvinit, NULL) near the end of the file and has no separate load/unload handler; hardware drivers call led_create()/led_destroy() to register their LEDs at runtime.

A13: Yes, in led_create_state(): sc = malloc(sizeof *sc, M_LED, M_WAITOK | M_ZERO);

6.18.6.3 Part 3: Tun/Tap Driver

A14: struct tuntap_softc

A15: Yes. The `softc` embeds an `ifnet` pointer (`tun_ifp`) and is linked to a `cdev` via `dev->si_drv1` and the `softc`'s back-pointer.

A16: There's no single `tun_cdevsw` variable. Three `struct cdevsw` definitions live inside the `tun-tap_drivers[]` array (one each for `tun`, `tap`, and `vmnet`). They share the same handlers (`tunopen`, `tunread`, `tunwrite`, `tunioctl`, `tunpoll`, `tunkqfilter`) and differ only in their `.d_name` and `flags`.

A17: `tunopen()` is assigned to `.d_open` in each `cdevsw` inside `tuntap_drivers[]`.

A18: In `tuncreate()`, the interface is created with `if_alloc(type)` where `type` is `IFT_ETHER` for `tap` and `IFT_PPP` for `tun`.

6.18.6.4 Part 4: PCI UART Driver

A19: `uart_pci_probe()`

A20: It calls `uart_pci_match()` against the `pci_ns8250_ids` table to match known UART vendor/device IDs, and falls back to the PCI class code (`PCIC_SIMPLECOMM` with subclass `PCIS_SIMPLECOMM_UART`) for generic 16550-class parts.

A21: At the end of the file: `DRIVER_MODULE(uart, pci, uart_pci_driver, NULL, NULL);`

A22: `pci` (the second argument to `DRIVER_MODULE`).

A23: `uart_pci_methods[]`

A24: Four entries plus `DEVMETHOD_END`: `device_probe`, `device_attach`, `device_detach`, and `device_resume`.

If your answers differ significantly:

1. Don't worry! FreeBSD code evolves between versions
2. The important part is **finding** the structures, not exact line numbers
3. If you found similar patterns in different locations, that's success

6.18.7 Success Criteria

- Found all major structures in each driver
- Understand the pattern: entry points (`cdevsw/ifnet`), lifecycle (`probe/attach/detach`), registration (`DRIVER_MODULE/DECLARE_MODULE`)
- Can navigate driver source confidently
- Recognize differences between driver types (`character` vs `network` vs `bus-attached`)

6.18.8 What You Learned

- **Character devices** use `cdevsw` structures with entry point functions
- **Network devices** combine character devices (`cdev`) with network interfaces (`ifnet`)
- **Bus-attached drivers** use `newbus` (`probe/attach/detach`) and method tables
- **Infrastructure modules** may skip `probe/attach` if they're not hardware drivers
- **softc structures** hold per-device state
- **Module registration** varies (`DECLARE_MODULE` vs `DRIVER_MODULE`) depending on driver type

6.18.9 Lab Logbook Entry Template

Lab 1 Complete: [Date]

Time taken: ____ minutes

Questions answered: 24/24

Most interesting discovery:

[What surprised you most about real driver code?]

Challenging aspects:

[What was hard to find? Any patterns you didn't expect?]

Key insight:

[What "clicked" for you during this exploration?]

Next steps:

[Ready for Lab 2 where you'll build your first module]

6.19 Lab 2: Minimal Module with Logs Only

6.19.1 Goal

Build, load, and unload your first kernel module. Confirm your toolchain works and understand the module lifecycle through direct observation.

6.19.2 What You'll Learn

- How to write a minimal kernel module
- How to create a Makefile for kernel module builds
- How to load and unload modules safely
- How to observe kernel messages in dmesg
- The module event handler lifecycle (load/unload)
- How to troubleshoot common build errors

6.19.3 Prerequisites

- FreeBSD 14.3 with /usr/src installed
- Build tools installed (clang, make)
- sudo/root access
- Completed Lab 1 (recommended but not required)

6.19.4 Time Estimate

40-50 minutes (including build, test, and documentation)

6.19.5 Instructions

6.19.5.1 Step 1: Create Working Directory

```
% mkdir -p ~/drivers/hello
% cd ~/drivers/hello
```

Why this location?: Your home directory keeps driver experiments separate from system files and survives reboots.

6.19.5.2 Step 2: Create the Minimal Driver

Create a file named `hello.c`:

```
% vi hello.c    # or nano, emacs, your choice
```

Enter the following code (explanation follows):

```
/*
 * hello.c - Minimal FreeBSD kernel module for testing
 *
 * This is the simplest possible kernel module: it does nothing except
 * print messages when loaded and unloaded. Perfect for verifying that
 * your build environment works correctly.
 *
 * FreeBSD 14.3 compatible
 */

#include <sys/param.h>    /* System parameter definitions */
#include <sys/module.h>    /* Kernel module definitions */
#include <sys/kernel.h>    /* Kernel types and macros */
#include <sys/system.h>    /* System functions (printf) */

/*
 * Module event handler
 *
 * This function is called whenever something happens to the module:
 * - MOD_LOAD: Module is being loaded into the kernel
 * - MOD_UNLOAD: Module is being removed from the kernel
 */
```

```
* - MOD_SHUTDOWN: System is shutting down (rare, usually not implemented)
* - MOD QUIESCE: Module should prepare for unload (advanced, not shown here)
*
* Parameters:
*   mod: Module identifier (handle to this module)
*   event: What's happening (MOD_LOAD, MOD_UNLOAD, etc.)
*   arg: Extra data (usually NULL, not used here)
*
* Returns:
*   0 on success
*   Error code (like EOPNOTSUPP) on failure
*/
static int
hello_modevent(module_t mod __unused, int event, void *arg __unused)
{
    int error = 0;

    /*
     * The __unused attribute tells the compiler "I know these parameters
     * aren't used, don't warn me about it." It's good practice to mark
     * intentionally unused parameters.
     */

    switch (event) {
    case MOD_LOAD:
        /*
         * This runs when someone does 'kldload hello.ko'
         *
         * printf() in kernel code goes to the kernel message buffer,
         * which you can see with 'dmesg' or in /var/log/messages.
         *
         * Notice we say "Hello:" at the start - this helps identify
         * which module printed the message when reading logs.
         */
        printf("Hello: Module loaded successfully!\n");
        printf("Hello: This message appears in dmesg\n");
        printf("Hello: Module address: %p\n", (void *)&hello_modevent);
        break;

    case MOD_UNLOAD:
        /*
         * This runs when someone does 'kldunload hello'
         *
         * This is where you'd clean up resources if this module
```

```

        * had allocated anything. Our minimal module has nothing
        * to clean up.
        */
    printf("Hello: Module unloaded. Goodbye!\n");
    break;

default:
    /*
     * We don't handle other events (like MOD_SHUTDOWN).
     * Return EOPNOTSUPP ("operation not supported").
     */
    error = EOPNOTSUPP;
    break;
}

return (error);
}

/*
 * Module declaration structure
 *
 * This tells the kernel about our module:
 * - name: "hello" (how it appears in kldstat)
 * - evhand: pointer to our event handler
 * - priv: private data (NULL for us, we have none)
 */
static moduledata_t hello_mod = {
    "hello",          /* module name */
    hello_modevent,    /* event handler function */
    NULL              /* extra data (not used) */
};

/*
 * DECLARE_MODULE macro
 *
 * This is the magic that registers our module with FreeBSD.
 *
 * Parameters:
 * 1. hello: Unique module identifier (matches name in moduledata_t)
 * 2. hello_mod: Our moduledata_t structure
 * 3. SI_SUB_DRIVERS: Subsystem order (we're a "driver" subsystem)
 * 4. SI_ORDER_MIDDLE: Load order within subsystem (middle of the pack)
 *
 * Load order matters when modules depend on each other. SI_SUB_DRIVERS

```

```
* and SI_ORDER_MIDDLE are safe defaults for simple modules.
*/
DECLARE_MODULE(hello, hello_mod, SI_SUB_DRIVERS, SI_ORDER_MIDDLE);

/*
* MODULE_VERSION macro
*
* Declares the version of this module. Version numbers help the kernel
* manage module dependencies and compatibility.
*
* Format: MODULE_VERSION(name, version_number)
* Version 1 is fine for new modules.
*/
MODULE_VERSION(hello, 1);
```

Code explanation summary:

- **Includes:** Bring in kernel headers (unlike userspace, we can't use `<stdio.h>`)
- **Event handler:** Function called when module loads/unloads
- **moduledata_t:** Connects the module name to its event handler
- **DECLARE_MODULE:** Registers everything with the kernel
- **MODULE_VERSION:** Declares version for dependency tracking

6.19.5.3 Step 3: Create the Makefile

Create a file named `Makefile` (exact name, capital M):

```
% vi Makefile
```

Enter this content:

```
# Makefile for hello kernel module
#
# This Makefile uses FreeBSD's kernel module build infrastructure.
# The .include at the end does all the heavy lifting.

# KMOD: Kernel module name (will produce hello.ko)
KMOD=    hello

# SRCS: Source files to compile (just hello.c)
SRCS=    hello.c

# Include FreeBSD's kernel module build rules
# This single line gives you:
```

```
# - 'make' or 'make all': Build the module
# - 'make clean': Remove build artifacts
# - 'make install': Install to /boot/modules (don't use in lab!)
# - 'make load': Load the module (requires root)
# - 'make unload': Unload the module (requires root)
.include <bsd.kmod.mk>
```

Makefile notes:

- **Must be named “Makefile”** (or “makefile”, but “Makefile” is convention)
- **Tabs matter:** If you get errors, check that indentation uses TABS not spaces
- **KMOD** determines the output filename (hello.ko)
- **bsd.kmod.mk** is FreeBSD’s kernel module build infrastructure (does the complex stuff)

6.19.5.4 Step 4: Build the Module

```
% make clean
rm -f hello.ko hello.o ... [various cleanup]

% make
cc -O2 -pipe -fno-strict-aliasing -Werror -D_KERNEL -DKLD_MODULE ... -c hello.c
ld -d -warn-common -r -d -o hello.ko hello.o
```

What’s happening:

1. **make clean:** Removes old build artifacts (always safe to run)
2. **make:** Compiles hello.c to hello.o, then links to create hello.ko
3. The compiler flags (-D_KERNEL -DKLD_MODULE) tell the code it’s in kernel mode

Expected output: You should see compilation commands but **no errors**.

Common error messages:

```
Error: "implicit declaration of function 'printf'"
Fix: Check your includes - you need <sys/systm.h>
```

```
Error: "expected ';' before '}'"
Fix: Check for missing semicolons in your code
```

```
Error: "undefined reference to __something"
Fix: Usually means wrong includes or typo in function name
```

6.19.5.5 Step 5: Verify Build Success

```
% ls -lh hello.ko
-rwxr-xr-x  1 youruser  youruser   14K Nov 14 15:30 hello.ko
```

What to look for:

- **File exists:** `hello.ko` is present
- **Size is reasonable:** 10-20 KB is typical for minimal modules
- **Executable bit set:** `-rwxr-xr-x` (the 'x' means executable)

6.19.5.6 Step 6: Load the Module

```
% sudo kldload ./hello.ko
```

Important notes:

- **Must use sudo** (or be root): Only root can load kernel modules
- **Use `./hello.ko`:** The `./` tells `kldload` to use the local file, not search system paths
- **No output is normal:** If it loads successfully, `kldload` prints nothing

If you get an error:

```
kldload: can't load ./hello.ko: module already loaded or in kernel
Solution: The module is already loaded. Unload it first: sudo kldunload hello
```

```
kldload: can't load ./hello.ko: Exec format error
Solution: Module was built for different FreeBSD version. Rebuild on target system.
```

```
kldload: an error occurred. Please check dmesg(8) for more details.
Solution: Run 'dmesg | tail' to see what went wrong
```

6.19.5.7 Step 7: Verify Module Is Loaded

```
% kldstat | grep hello
 5      1 0xffffffff82500000      3000 hello.ko
```

Column meanings:

- **5:** Module ID (your number may differ)
- **1:** Reference count (how many things depend on it)
- **0xffffffff82500000:** Kernel memory address where module is loaded
- **3000:** Size in hex ($0x3000 = 12288 \text{ bytes} = 12 \text{ KB}$)
- **hello.ko:** Module filename

6.19.5.8 Step 8: View Kernel Messages

```
% dmesg | tail -5
Hello: Module loaded successfully!
Hello: This message appears in dmesg
Hello: Module address: 0xffffffff82500000
```

What's dmesg?: The kernel message buffer. Everything printed with `printf()` in kernel code goes here.

Alternative ways to view:

```
% dmesg | grep Hello
% tail -f /var/log/messages    # Watch in real-time (Ctrl+C to stop)
```

6.19.5.9 Step 9: Unload the Module

```
% sudo kldunload hello
```

What happens:

1. Kernel calls your `hello_modevent()` with `MOD_UNLOAD`
2. Your handler prints “Goodbye!” and returns 0 (success)
3. Kernel removes the module from memory

6.19.5.10 Step 10: Verify Unload Messages

```
% dmesg | tail -3
Hello: This message appears in dmesg
Hello: Module address: 0xffffffff82500000
Hello: Module unloaded. Goodbye!
```

6.19.5.11 Step 11: Confirm Module Is Gone

```
% kldstat | grep hello
[no output - module is unloaded]

% ls -l /dev/ | grep hello
[no output - this module doesn't create devices]
```


6.19.6 Behind the Scenes: What Just Happened?

Let's trace the **complete life cycle** of your module:

6.19.6.1 When you ran `kldload ./hello.ko`:

1. **Kernel loads file**: Read `hello.ko` from disk into kernel memory
2. **Relocation**: Adjust memory addresses in the code to work at the loaded address
3. **Symbol resolution**: Connect function calls to their implementations
4. **Initialization**: Call your `hello_modevent()` with `MOD_LOAD`
5. **Registration**: Add "hello" to the kernel's module list
6. **Complete**: `kldload` returns success (exit code 0)

Your `printf()` calls in `MOD_LOAD` happened during step 4.

6.19.6.2 When you ran `kldunload hello`:

1. **Lookup**: Find the "hello" module in kernel's module list
2. **Reference check**: Ensure nothing is using the module (ref count = 1)
3. **Shutdown**: Call your `hello_modevent()` with `MOD_UNLOAD`
4. **Cleanup**: Remove from module list
5. **Unmap**: Free the kernel memory that held the module code
6. **Complete**: `kldunload` returns success

Your `printf()` in `MOD_UNLOAD` happened during step 3.

6.19.6.3 Why `DECLARE_MODULE` and `MODULE_VERSION` matter:

```
DECLARE_MODULE(hello, hello_mod, SI_SUB_DRIVERS, SI_ORDER_MIDDLE);
```

This macro expands to code that creates a special data structure in a special ELF section (`.set` section) of the `hello.ko` file. When the kernel loads the module, it scans for these structures and knows:

- **Name**: "hello"
- **Handler**: `hello_modevent`
- **When to initialize**: `SI_SUB_DRIVERS` phase, `SI_ORDER_MIDDLE` position

Without this macro, the kernel wouldn't know your module exists!

6.19.7 Troubleshooting Guide

6.19.7.1 Problem: Module won't compile

Symptom: `make` shows errors

Common causes:

1. **Typo in code:** Carefully compare with example above
2. **Wrong includes:** Check that all four `#include` lines are present
3. **Tabs vs spaces in Makefile:** Makefiles require TABS for indentation
4. **Missing `/usr/src`:** Build needs kernel headers from `/usr/src`

Debug steps:

```
# Check if /usr/src exists
% ls /usr/src/sys/sys/param.h
[should exist]

# Try compiling manually to see better errors
% cc -c -D_KERNEL -I/usr/src/sys hello.c
```

6.19.7.2 Problem: “Operation not permitted” when loading

Symptom: `kldload: can't load ./hello.ko: Operation not permitted`

Cause: Not running as root

Fix:

```
% sudo kldload ./hello.ko
# OR
% su
# kldload ./hello.ko
```

6.19.7.3 Problem: “module already loaded”

Symptom: `kldload: can't load ./hello.ko: module already loaded`

Cause: Module is already in the kernel

Fix:

```
% sudo kldunload hello
% sudo kldload ./hello.ko
```

6.19.7.4 Problem: No messages in `dmesg`

Symptom: `kldload` succeeds but `dmesg` shows nothing

Possible causes:

1. **Messages scrolled away:** Use `dmesg | tail -20` to see recent messages
2. **Wrong module loaded:** Check `kldstat` to verify your module is there
3. **Event handler not called:** Check that `DECLARE_MODULE` matches `moduledata_t` name

6.19.7.5 Problem: Kernel panic

Symptom: System crashes, shows panic message

Unlikely with this minimal module, but if it happens:

1. **Don't panic** (no pun intended): Your VM can be rebooted
2. **Check the code:** Probably a typo in the `DECLARE_MODULE` macro
3. **Start fresh:** Reboot VM, compare your code character-by-character with example

6.19.8 Success Criteria

- Module compiles without errors or warnings
- `hello.ko` file is created (10-20 KB)
- Module loads without errors
- Messages appear in `dmesg` showing load
- Module appears in `kldstat` output
- Module unloads successfully
- Unload message appears in `dmesg`
- No kernel panics or crashes

6.19.9 What You Learned

Technical skills:

- Writing a minimal kernel module structure
- Using FreeBSD's kernel module build system
- Loading and unloading kernel modules safely
- Observing kernel messages with `dmesg`

Concepts:

- Module event handlers (`MOD_LOAD`/`MOD_UNLOAD` lifecycle)
- `DECLARE_MODULE` and `MODULE_VERSION` macros
- Kernel `printf` vs userspace `printf`
- Why root access is required for module operations

Confidence:

- Your build environment works correctly
- You can compile and load kernel code

- You understand the basic module lifecycle
- You're ready to add actual functionality (Lab 3)

6.19.10 Lab Logbook Entry Template

Lab 2 Complete: [Date]

Time taken: ____ minutes

Build results:

- First attempt: ☐ Success ☐ Errors (describe: ____)
- After fixes: ☐ Success

Module operations:

- Load: ☐ Success ☐ Errors
- Visible in kldstat: ☐ Yes ☐ No
- Messages in dmesg: ☐ Yes ☐ No
- Unload: ☐ Success ☐ Errors

Key insight:

[What did you learn about the kernel module lifecycle?]

Challenges faced:

[What went wrong? How did you fix it?]

Next steps:

[Ready for Lab 3: adding real functionality with device nodes]

6.19.11 Optional Experiment: Module Load Order

Want to see why SI_SUB and SI_ORDER matter?

1. Check current boot order:

```
% kldstat -v | less
```

2. Try different subsystem orders: Edit hello.c and change:

```
DECLARE_MODULE(hello, hello_mod, SI_SUB_DRIVERS, SI_ORDER_MIDDLE);
```

to:

```
DECLARE_MODULE(hello, hello_mod, SI_SUB_PSEUDO, SI_ORDER_FIRST);
```

Rebuild and reload. Module still works! The order only matters when modules depend on each other.

6.20 Lab 3: Create and Remove a Device Node

6.20.1 Goal

Extend the minimal module to create a `/dev` entry that users can interact with. Implement basic read and write operations.

6.20.2 What You'll Learn

- How to create a character device node in `/dev`
- How to implement `cdevsw` (character device switch) entry points
- How to safely copy data between user and kernel space with `uiomove()`
- How `open/close/read/write` syscalls connect to your driver functions
- The relationship between `struct cdev`, `cdevsw`, and device operations
- Proper resource cleanup and `NULL` pointer safety

6.20.3 Prerequisites

- Completed Lab 2 (Hello Module)
- Understanding of file operations (`open`, `read`, `write`, `close`)
- Basic C string handling knowledge

6.20.4 Time Estimate

60-75 minutes (including code understanding, building, and thorough testing)

6.20.5 Instructions

6.20.5.1 Step 1: Create New Working Directory

```
% mkdir -p ~/drivers/demo  
% cd ~/drivers/demo
```

Why a new directory?: Keep each lab self-contained for easy reference later.

6.20.5.2 Step 2: Create the Driver Source

Create `demo.c` with the following complete code:

```
/*
 * demo.c - Simple character device with /dev node
 *
 * This driver demonstrates:
 * - Creating a device node in /dev
 * - Implementing open/close/read/write operations
 * - Safe data transfer between kernel and user space
 * - Proper resource management and cleanup
 *
 * Compatible with FreeBSD 14.3
 */

#include <sys/param.h>      /* System parameters and limits */
#include <sys/module.h>      /* Kernel module support */
#include <sys/kernel.h>      /* Kernel types */
#include <sys/system.h>      /* System functions like printf */
#include <sys/conf.h>        /* Character device configuration */
#include <sys/uio.h>         /* User I/O structures and uiomove() */
#include <sys/malloc.h>      /* Kernel memory allocation */

/*
 * Global device node pointer
 *
 * This holds the handle to our /dev/demo entry. We need to keep this
 * so we can destroy the device when the module unloads.
 *
 * NULL when module is not loaded.
 */
static struct cdev *demo_dev = NULL;

/*
 * Open handler - called when someone opens /dev/demo
 *
 * This is called every time a process opens the device file:
 *   open("/dev/demo", O_RDWR);
 *   cat /dev/demo
 *   echo "hello" > /dev/demo
 *
 * Parameters:
 *   dev: Device being opened (our cdev structure)
 *   oflags: Open flags (O_RDONLY, O_WRONLY, O_RDWR, O_NONBLOCK, etc.)
 */
```

```

*  devtype: Device type (usually S_IFCHR for character devices)
*  td: Thread opening the device (process context)
*
* Returns:
*  0 on success
*  Error code (like EBUSY, ENOMEM) on failure
*
* Note: The __unused attribute marks parameters we don't use, avoiding
*       compiler warnings.
*/
static int
demo_open(struct cdev *dev __unused, int oflags __unused,
          int devtype __unused, struct thread *td __unused)
{
    /*
     * In a real driver, you might:
     * - Check if exclusive access is required
     * - Allocate per-open state
     * - Initialize hardware
     * - Check device readiness
     *
     * Our simple demo just logs that open happened.
     */
    printf("demo: Device opened (pid=%d, comm=%s)\n",
          td->td_proc->p_pid, td->td_proc->p_comm);

    return (0); /* Success */
}

/*
* Close handler - called when last reference is closed
*
* Important: This is called when the LAST file descriptor referring to
* this device is closed. If a process opens /dev/demo twice, close is
* called only after both fds are closed.
*
* Parameters:
*  dev: Device being closed
*  fflag: File flags from the open call
*  devtype: Device type
*  td: Thread closing the device
*
* Returns:
*  0 on success

```

```

* Error code on failure
*/
static int
demo_close(struct cdev *dev __unused, int fflag __unused,
           int devtype __unused, struct thread *td __unused)
{
    /*
    * In a real driver, you might:
    * - Free per-open state
    * - Flush buffers
    * - Update hardware state
    * - Cancel pending operations
    */
    printf("demo: Device closed (pid=%d)\n", td->td_proc->p_pid);

    return (0); /* Success */
}

/*
* Read handler - transfer data from kernel to user space
*
* This is called when someone reads from the device:
* cat /dev/demo
* dd if=/dev/demo of=output.txt bs=1024 count=1
* read(fd, buffer, size);
*
* Parameters:
* dev: Device being read from
* uio: User I/O structure describing the read request
* ioflag: I/O flags (IO_NDELAY for non-blocking, etc.)
*
* The 'uio' structure contains:
* uio_resid: Bytes remaining to transfer (initially = read size)
* uio_offset: Current position in the "file" (we ignore this)
* uio_rw: Direction (UIO_READ for read operations)
* uio_td: Thread performing the I/O
* [internal]: Scatter-gather list describing user buffer(s)
*
* Returns:
* 0 on success
* Error code (like EFAULT if user buffer is invalid)
*/
static int
demo_read(struct cdev *dev __unused, struct uio *uio, int ioflag __unused)

```



```
{
    /*
     * Our message to return to user space.
     * Could be device data, sensor readings, status info, etc.
     */
    char message[] = "Hello from demo driver!\n";
    size_t len;
    int error;

    /*
     * Log the read request details.
     * uio_resid tells us how many bytes the user wants to read.
     */
    printf("demo: Read called, uio_resid=%zd bytes requested\n",
        uio->uio_resid);

    /*
     * Calculate how many bytes to actually transfer.
     *
     * We use MIN() to transfer the smaller of:
     * 1. What the user requested (uio_resid)
     * 2. What we have available (sizeof(message)-1, excluding null terminator)
     *
     * Why -1? The null terminator '\0' is for C string handling in the
     * kernel, but we don't send it to user space. Text files don't have
     * null terminators between lines.
     */
    len = MIN(uio->uio_resid, sizeof(message) - 1);

    /*
     * uiomove() - The safe way to copy data to user space
     *
     * This function:
     * 1. Verifies the user's buffer is valid and writable
     * 2. Copies 'len' bytes from 'message' to the user's buffer
     * 3. Automatically updates uio->uio_resid (subtracts len)
     * 4. Handles scatter-gather buffers (if user buffer is non-contiguous)
     * 5. Returns error if user buffer is invalid (EFAULT)
     *
     * CRITICAL SAFETY RULE:
     * Never use memcpy(), bcopy(), or direct pointer access for user data!
     * User pointers are in user space, not accessible in kernel space.
     * uiomove() safely bridges this gap.
     */
}
```

```

    * Parameters:
    *   message: Source data (kernel space)
    *   len: Bytes to copy
    *   uio: Destination description (user space)
    *
    * After uiomove() succeeds:
    *   uio->uio_resid is decreased by len
    *   uio->uio_offset is increased by len (for seekable devices)
    */
error = uiomove(message, len, uio);

if (error != 0) {
    printf("demo: Read failed, error=%d\n", error);
    return (error);
}

printf("demo: Read completed, transferred %zu bytes\n", len);

/*
 * Return 0 for success.
 * The caller knows how much we transferred by checking how much
 * uio_resid decreased.
 */
return (0);
}

/*
 * Write handler - receive data from user space
 *
 * This is called when someone writes to the device:
 *   echo "hello" > /dev/demo
 *   dd if=input.txt of=/dev/demo bs=1024
 *   write(fd, buffer, size);
 *
 * Parameters:
 *   dev: Device being written to
 *   uio: User I/O structure describing the write request
 *   ioflag: I/O flags
 *
 * Returns:
 *   0 on success (usually - see note below)
 *   Error code on failure
 *
 * IMPORTANT WRITE SEMANTICS:

```

```
* Unlike read(), write() is expected to consume ALL the data.
* If you don't consume everything (uio_resid > 0 after return),
* the kernel will call write() again with the remaining data.
* This can cause infinite loops if you always return 0 with resid > 0!
*/
static int
demo_write(struct cdev *dev __unused, struct uio *uio, int ioflag __unused)
{
    char buffer[128]; /* Temporary buffer for incoming data */
    size_t len;
    int error;

    /*
     * Limit transfer size to our buffer size.
     *
     * We use sizeof(buffer)-1 to reserve space for null terminator
     * (so we can safely print the string).
     *
     * Note: Real drivers might:
     * - Accept unlimited data (loop calling uiomove)
     * - Have larger buffers
     * - Queue data for processing
     * - Return EFBIG if data exceeds device capacity
    */
    len = MIN(uio->uio_resid, sizeof(buffer) - 1);

    /*
     * uiomove() for write: Copy FROM user space TO kernel buffer
     *
     * Same function, but now we're the destination.
     * The direction is determined by uio->uio_rw internally.
    */
    error = uiomove(buffer, len, uio);
    if (error != 0) {
        printf("demo: Write failed during uiomove, error=%d\n", error);
        return (error);
    }

    /*
     * Add null terminator so we can safely use printf.
     *
     * SECURITY NOTE: In a real driver, you must validate data!
     * - Check for null bytes if expecting text
     * - Validate ranges for numeric data
    */
```

```

    * - Sanitize before using in format strings
    * - Never trust user input
    */
    buffer[len] = '\0';

    /*
     * Do something with the data.
     *
     * Real drivers might:
     * - Send to hardware (network packet, disk write, etc.)
     * - Process commands (like LED control strings)
     * - Update device state
     * - Queue for async processing
     *
     * We just log it.
     */
    printf("demo: User wrote %zu bytes: \"%s\"\n", len, buffer);

    /*
     * Return success.
     *
     * At this point, uio->uio_resid should be 0 (we consumed everything).
     * If not, the kernel will call us again with the remainder.
     */
    return (0);
}

/*
 * Character device switch (cdevsw) structure
 *
 * This is the "method table" that connects system calls to your functions.
 * When a user process calls open(), read(), write(), etc. on /dev/demo,
 * the kernel looks up this table to find which function to call.
 *
 * Think of it as a virtual function table (vtable) in OOP terms.
 */
static struct cdevsw demo_cdevsw = {
    .d_version =    D_VERSION,      /* ABI version - always required */
    .d_open =      demo_open,      /* open() syscall handler */
    .d_close =     demo_close,     /* close() syscall handler */
    .d_read =      demo_read,      /* read() syscall handler */
    .d_write =     demo_write,     /* write() syscall handler */
    .d_name =      "demo",         /* Device name for identification */

```

```

/*
 * Other possible entries (not used here):
 *
 * .d_ioctl =    demo_ioctl,    // ioctl() for configuration/control
 * .d_poll =     demo_poll,     // poll()/select() for readiness
 * .d_mmap =     demo_mmap,     // mmap() for direct memory access
 * .d_strategy=  demo_strategy, // For block devices (legacy)
 * .d_kqfilter=  demo_kqfilter, // kqueue event notification
 *
 * Unimplemented entries default to NULL and return ENODEV.
 */
};

/*
 * Module event handler
 *
 * This is called on module load and unload.
 * We create our device node on load, destroy it on unload.
 */
static int
demo_mdevent(module_t mod __unused, int event, void *arg __unused)
{
    int error = 0;

    switch (event) {
    case MOD_LOAD:
        /*
         * make_dev() - Create a device node in /dev
         *
         * This is the key function that makes your driver visible
         * to user space. It creates an entry in the devfs filesystem.
         *
         * Parameters:
         *   &demo_cdevsw: Pointer to our method table
         *   0: Unit number (minor number) - use 0 for single-instance devices
         *   UID_ROOT: Owner user ID (0 = root)
         *   GID_WHEEL: Owner group ID (0 = wheel group)
         *   0666: Permissions (rw-rw-rw- = world read/write)
         *   "demo": Device name (appears as /dev/demo)
         *
         * Returns:
         *   Pointer to cdev structure on success
         *   NULL on failure (rare - usually only if name collision)
         */

```

```

    * The returned cdev is an opaque handle representing the device.
    */
demo_dev = make_dev(&demo_cdevsw,
                    0,           /* unit number */
                    UID_ROOT,   /* owner UID */
                    GID_WHEEL,  /* owner GID */
                    0666,       /* permissions: rw-rw-rw- */
                    "demo");    /* device name */

/*
* Always check if make_dev() succeeded.
* Failure is rare but possible.
*/
if (demo_dev == NULL) {
    printf("demo: Failed to create device node\n");
    return (ENXIO); /* "Device not configured" */
}

printf("demo: Device /dev/demo created successfully\n");
printf("demo: Permissions: 0666 (world readable/writable)\n");
printf("demo: Try: cat /dev/demo\n");
printf("demo: Try: echo \"test\" > /dev/demo\n");
break;

case MOD_UNLOAD:
    /*
    * Cleanup on module unload.
    *
    * CRITICAL ORDERING:
    * 1. Make device invisible (destroy_dev)
    * 2. Wait for all operations to complete
    * 3. Free resources
    *
    * destroy_dev() does steps 1 and 2 automatically!
    */

    /*
    * Always check for NULL before destroying.
    * This protects against:
    * - MOD_LOAD failure (demo_dev never created)
    * - Double-unload attempts
    * - Corrupted state
    */
    if (demo_dev != NULL) {

```

```
/*
 * destroy_dev() - Remove device node and clean up
 *
 * This function:
 * 1. Removes /dev/demo from the filesystem
 * 2. Marks device as "going away"
 * 3. WAITS for all in-progress operations to complete
 * 4. Ensures no new operations can start
 * 5. Frees associated kernel resources
 *
 * SYNCHRONIZATION GUARANTEE:
 * After destroy_dev() returns, no threads are executing
 * your open/close/read/write functions. This makes cleanup
 * safe - no race conditions with active I/O.
 *
 * This is why you can safely unload modules while they're
 * in use (e.g., someone has the device open). The unload
 * will wait until they close it.
 */
destroy_dev(demo_dev);

/*
 * Set pointer to NULL for safety.
 *
 * Defense in depth: If something accidentally tries to
 * use demo_dev after unload, NULL pointer dereference
 * is much easier to debug than use-after-free.
 */
demo_dev = NULL;

printf("demo: Device /dev/demo destroyed\n");
}
break;

default:
/*
 * We don't handle MOD_SHUTDOWN or other events.
 */
error = EOPNOTSUPP;
break;
}

return (error);
}
```

```
/*
 * Module declaration - connects everything together
 */
static moduledata_t demo_mod = {
    "demo",          /* Module name */
    demo_modevent,    /* Event handler */
    NULL              /* Extra data */
};

/*
 * Register module with kernel
 */
DECLARE_MODULE(demo, demo_mod, SI_SUB_DRIVERS, SI_ORDER_MIDDLE);

/*
 * Declare module version
 */
MODULE_VERSION(demo, 1);
```

Key concepts in this code:

1. **cdevsw structure**: The dispatch table connecting syscalls to your functions
2. **uiomove()**: Safe kernel <-> user data transfer (never use memcpy!)
3. **make_dev()**: Creates visible /dev entry
4. **destroy_dev()**: Removes device and waits for operations to complete
5. **NULL safety**: Always check pointers before use, set to NULL after free

6.20.5.3 Step 3: Create the Makefile

Create Makefile:

```
# Makefile for demo character device driver

KMOD=    demo
SRCS=    demo.c

.include <bsd.kmod.mk>
```

6.20.5.4 Step 4: Build the Driver


```
% make clean
rm -f demo.ko demo.o ...

% make
cc -O2 -pipe -fno-strict-aliasing -Werror -D_KERNEL ... -c demo.c
ld -d -warn-common -r -d -o demo.ko demo.o
```

Expected: Clean build with no errors.

If you see warnings about unused parameters: This is fine - we marked them `__unused` but some compiler versions still warn.

6.20.5.5 Step 5: Load the Driver

```
% sudo kldload ./demo.ko

% dmesg | tail -5
demo: Device /dev/demo created successfully
demo: Permissions: 0666 (world readable/writable)
demo: Try: cat /dev/demo
demo: Try: echo "test" > /dev/demo
```

6.20.5.6 Step 6: Verify Device Node Creation

```
% ls -l /dev/demo
crw-rw-rw-  1 root  wheel  0x5e Nov 14 16:00 /dev/demo
```

What you're seeing:

- **c:** Character device (not block device or regular file)
- **rw-rw-rw-:** Permissions 0666 (anyone can read/write)
- **root wheel:** Owned by root, group wheel
- **0x5e:** Device number (major/minor combined - your value may differ)
- **/dev/demo:** The device path

6.20.5.7 Step 7: Test Reading

```
% cat /dev/demo
Hello from demo driver!
```

What happened:

1. cat opened /dev/demo -> demo_open() called
2. cat called read() -> demo_read() called
3. Driver copied “Hello from demo driver!\n” to cat’s buffer via uiomove()
4. cat printed the received data to stdout
5. cat closed the file -> demo_close() called

Check kernel log:

```
% dmesg | tail -5
demo: Device opened (pid=1234, comm=cat)
demo: Read called, uio_resid=65536 bytes requested
demo: Read completed, transferred 25 bytes
demo: Device closed (pid=1234)
```

Note: uio_resid=65536 means cat requested 64 KB (its default buffer). We only sent 25 bytes, which is fine - read() returns how much was actually transferred.

6.20.5.8 Step 8: Test Writing

```
% echo "Test message" > /dev/demo

% dmesg | tail -4
demo: Device opened (pid=1235, comm=sh)
demo: User wrote 13 bytes: "Test message
"
demo: Device closed (pid=1235)
```

What happened:

1. Shell opened /dev/demo for writing
2. echo wrote “Test message\n” (13 bytes including newline)
3. Driver received it via uiomove() and logged it
4. Shell closed the device

6.20.5.9 Step 9: Test Multiple Operations

```
% (cat /dev/demo; echo "Another test" > /dev/demo; cat /dev/demo)
Hello from demo driver!
Hello from demo driver!
```

Watch dmesg in another terminal:

```
% dmesg -w      # Watch mode - updates in real-time
...
demo: Device opened (pid=1236, comm=sh)
demo: Read called, uio_resid=65536 bytes requested
demo: Read completed, transferred 25 bytes
demo: Device closed (pid=1236)
demo: Device opened (pid=1237, comm=sh)
demo: User wrote 13 bytes: "Another test
"
demo: Device closed (pid=1237)
demo: Device opened (pid=1238, comm=sh)
demo: Read called, uio_resid=65536 bytes requested
demo: Read completed, transferred 25 bytes
demo: Device closed (pid=1238)
```

6.20.5.10 Step 10: Test With dd (Controlled I/O)

```
% dd if=/dev/demo bs=10 count=1 2>/dev/null
Hello from

% dd if=/dev/demo bs=100 count=1 2>/dev/null
Hello from demo driver!
```

What this shows:

- First dd: Requested 10 bytes, got 10 bytes (“Hello from”)
- Second dd: Requested 100 bytes, got 25 bytes (our full message)
- The driver respects the requested size via `uio_resid`

6.20.5.11 Step 11: Verify Unload Protection

Open the device and keep it open:

```
% (sleep 30; echo "Done") > /dev/demo &
[1] 1240
```

Now try to unload (in the same 30-second window):

```
% sudo kldunload demo
[hangs... waiting...]
```

After 30 seconds:

```
Done
demo: Device closed (pid=1240)
demo: Device /dev/demo destroyed
[kldunload completes]
```

What happened: `destroy_dev()` waited for the write operation to complete before allowing the unload. This is a CRITICAL safety feature - it prevents crashes from unloading code that's still executing.

6.20.5.12 Step 12: Final Cleanup

```
% sudo kldunload demo    # If still loaded
% ls -l /dev/demo
ls: /dev/demo: No such file or directory # Good - it's gone
```

6.20.6 Behind the Scenes: The Complete Path

Let's trace `cat /dev/demo` from shell to driver and back:

6.20.6.1 1. Shell executes `cat`

User space:
Shell forks, execs `/bin/cat` with argument `"/dev/demo"`

6.20.6.2 2. `cat` opens the file

```
User space:
cat: fd = open("/dev/demo", O_RDONLY);

Kernel:
-> VFS layer: Lookup "/dev/demo" in devfs
-> devfs: Find cdev structure (created by make_dev)
-> devfs: Allocate file descriptor, file structure
-> devfs: Call cdev->si_devsw->d_open (demo_open)

Kernel (in demo_open):
-> printf("Device opened...")
-> return 0 (success)

Kernel:
```

```
-> Return file descriptor to cat
```

User space:

```
cat: fd = 3 (success)
```

6.20.6.3 3. cat reads data

User space:

```
cat: n = read(fd, buffer, 65536);
```

Kernel:

```
-> VFS: Lookup file descriptor 3
-> VFS: Find associated cdev
-> VFS: Allocate and initialize uio structure:
    uio_rw = UIO_READ
    uio_resid = 65536 (requested size)
    uio_offset = 0
    [iovec array pointing to cat's buffer]
-> VFS: Call cdev->si_devsw->d_read (demo_read)
```

Kernel (in demo_read):

```
-> printf("Read called, uio_resid=65536...")
-> len = MIN(65536, 24) # We have 25 bytes (24 + null)
-> uiomove("Hello from demo driver!\n", 24, uio)
    -> Copy 24 bytes from kernel message[] to cat's buffer
    -> Update uio_resid: 65536 - 24 = 65512
-> printf("Read completed, transferred 24 bytes")
-> return 0
```

Kernel:

```
-> Calculate transferred = (original resid - final resid) = 24
-> Return 24 to cat
```

User space:

```
cat: n = 24 (got 24 bytes)
```

6.20.6.4 4. cat processes data

User space:

```
cat: write(STDOUT_FILENO, buffer, 24);
[Your terminal shows: Hello from demo driver!]
```

6.20.6.5 5. cat tries to read more

User space:

```
cat: n = read(fd, buffer, 65536); # Try to read more
```

Kernel:

```
-> Call demo_read again  
-> uiomove returns 24 bytes again (we always return same message)
```

User space:

```
cat: n = 24  
cat: write(STDOUT_FILENO, buffer, 24);  
[Would print again, but cat knows this is a device not a file]
```

Actually, cat will keep reading until it gets 0 bytes (EOF). Our driver never returns 0, so cat would hang! But typically cat times out or you hit Ctrl+C.

Better read() implementation for file-like behavior:

```
static size_t bytes_sent = 0; /* Track position */  
  
static int  
demo_read(struct cdev *dev __unused, struct uio *uio, int ioflag __unused)  
{  
    char message[] = "Hello from demo driver!\n";  
    size_t len;  
  
    /* If we already sent the message, return 0 (EOF) */  
    if (bytes_sent >= sizeof(message) - 1) {  
        bytes_sent = 0; /* Reset for next open */  
        return (0); /* EOF */  
    }  
  
    len = MIN(uio->uio_resid, sizeof(message) - 1 - bytes_sent);  
    uiomove(message + bytes_sent, len, uio);  
    bytes_sent += len;  
  
    return (0);  
}
```

But for our demo, the simple version is fine.

6.20.6.6 6. cat closes file

User space:

```
cat: close(fd);
```

Kernel:

```
-> VFS: Decrement file reference count
-> VFS: If last reference, call cdev->si_devsw->d_close (demo_close)
```

Kernel (in demo_close):

```
-> printf("Device closed...")
-> return 0
```

Kernel:

```
-> Free file descriptor
-> Return to cat
```

User space:

```
cat: exit(0)
```

6.20.7 Concept Deep-Dive: Why uiomove()?

Question: Why can't we just use `memcpy()` or direct pointer access?

Answer: User space and kernel space have **separate address spaces**.

6.20.7.1 Address space separation:

User space (cat process):

```
Address 0x1000: cat's buffer[0]
Address 0x1001: cat's buffer[1]
...
```

Kernel space:

```
Address 0x1000: DIFFERENT memory (maybe page tables)
Address 0x1001: DIFFERENT memory
```

A pointer that's valid in user space (like cat's buffer at 0x1000) is **meaningless** in kernel space. If you try:

```
/* WRONG - WILL CRASH */
char *user_buf = (char *)0x1000; /* User's buffer address */
strcpy(user_buf, "data"); /* KERNEL PANIC! */
```

The kernel will try to write to address 0x1000 in *kernel* address space, which is completely different memory. At best, you corrupt kernel data. At worst, immediate panic.

6.20.7.2 What uiomove() does:

1. **Validates:** Checks that user addresses are actually in user space
2. **Maps:** Temporarily maps user pages into kernel address space
3. **Copies:** Performs the copy using valid kernel addresses
4. **Unmaps:** Cleans up the temporary mapping
5. **Handles faults:** If user buffer is invalid, returns EFAULT

This is why **every driver must use uiomove(), copyin(), or copyout()** for user data transfer. Direct access is always wrong and dangerous.

6.20.8 Success Criteria

- Driver compiles without errors
- Module loads successfully
- Device node /dev/demo appears with correct permissions
- Can read from device (get message)
- Can write to device (message logged in dmesg)
- Operations appear in dmesg with correct PIDs
- Module can be unloaded cleanly
- Device node disappears after unload
- Unload waits for operations to complete (tested with sleep experiment)
- No kernel panics or crashes

6.20.9 What You Learned

Technical skills:

- Creating character device nodes with `make_dev()`
- Implementing `cdevsw` method table
- Safe user-kernel data transfer with `uiomove()`
- Proper resource cleanup with `destroy_dev()`
- Debugging with `printf()` and `dmesg`

Concepts:

- How syscalls (open/read/write/close) map to driver functions
- The role of `cdevsw` as a dispatch table
- Why `uiomove()` is necessary (address space separation)
- How `destroy_dev()` provides synchronization
- The relationship between `cdev`, `devfs`, and `/dev` entries

Best practices:

- Always check make_dev() return value
- Always check for NULL before destroy_dev()
- Set pointers to NULL after freeing
- Use MIN() to prevent buffer overruns
- Log operations for debugging

6.20.10 Common Mistakes and How to Avoid Them

6.20.10.1 Mistake 1: Using memcpy() instead of uiomove()

Wrong:

```
memcpy(user_buffer, kernel_data, size); /* CRASH! */
```

Right:

```
uiomove(kernel_data, size, uio); /* Safe */
```

6.20.10.2 Mistake 2: Not consuming all write data

Wrong:

```
demo_write(...) {  
    /* Only process part of the data */  
    uiomove(buffer, 10, uio);  
    return (0); /* BUG: uio_resid is not 0! */  
}
```

Result: Kernel calls demo_write() again with remaining data -> infinite loop

Right:

```
demo_write(...) {  
    /* Process ALL data */  
    len = MIN(uio->uio_resid, buffer_size);  
    uiomove(buffer, len, uio);  
    /* Now uio_resid = 0, or we return EFBIG if too much */  
    return (0);  
}
```

6.20.10.3 Mistake 3: Forgetting NULL check before destroy_dev()

Wrong:

```
MOD_UNLOAD:
    destroy_dev(demo_dev); /* What if make_dev failed? */
```

Right:

```
MOD_UNLOAD:
    if (demo_dev != NULL) {
        destroy_dev(demo_dev);
        demo_dev = NULL;
    }
```

6.20.10.4 Mistake 4: Wrong permissions on device node

If you use 0600 permissions:

```
make_dev(&demo_cdevsw, 0, UID_ROOT, GID_WHEEL, 0600, "demo");
```

Regular users can't access it:

```
% cat /dev/demo
cat: /dev/demo: Permission denied
```

Use 0666 for world-accessible devices (appropriate for learning/testing).

6.20.11 Lab Logbook Entry Template

Lab 3 Complete: [Date]

Time taken: ____ minutes

Build results:

- Compilation: [] Success [] Errors
- Module size: ____ KB

Testing results:

- Device node created: [] Yes [] No
- Permissions correct: [] Yes [] No (expected: crw-rw-rw-)
- Read test: [] Success [] Failed
- Write test: [] Success [] Failed
- Multiple operations: [] Success [] Failed
- Unload protection: [] Tested [] Not tested

Key insight:

[What did you learn about user-kernel data transfer?]

Most interesting discovery:

[What surprised you? Maybe how `destroy_dev` waits?]

Challenges faced:

[Any build errors? Runtime issues? How did you resolve them?]

Code understanding:

- `uiomove()` purpose: [Explain in your own words]
- `cdevsw` role: [Explain in your own words]
- Why NULL checks matter: [Explain in your own words]

Next steps:

[Ready for Lab 4: deliberate bugs and error handling]

6.21 Lab 4: Error Handling and Defensive Programming

6.21.1 Goal

Learn error handling by deliberately introducing bugs, observing symptoms, and fixing them properly. Develop defensive programming instincts for driver development.

6.21.2 What You'll Learn

- What happens when cleanup is incomplete
- How to detect resource leaks
- The importance of cleanup order
- How to handle allocation failures
- Defensive programming techniques (NULL checks, pointer clearing)
- How to debug driver issues using kernel logs and system tools

6.21.3 Prerequisites

- Completed Lab 3 (Demo Device)
- Understanding of `demo.c` code structure
- Ability to edit C code and rebuild

6.21.4 Time Estimate

30-40 minutes (deliberate breaking, observing, and fixing)

6.21.5 Important Safety Note

These experiments involve **deliberately crashing** your driver (not the kernel, just the driver). This is safe in your lab VM but demonstrates real bugs you must avoid in production code.

Always:

- Use your lab VM, never your host system
- Take a VM snapshot before starting
- Be prepared to reboot if something hangs

6.21.6 Part 1: The Resource Leak Bug

6.21.6.1 Experiment 1A: Forget to destroy_dev()

Goal: See what happens when you forget to clean up device nodes.

Step 1: Edit demo.c, comment out destroy_dev():

```
case MOD_UNLOAD:
    if (demo_dev != NULL) {
        /* destroy_dev(demo_dev); */ /* COMMENTED OUT - BUG! */
        demo_dev = NULL;
        printf("demo: Device /dev/demo destroyed\n"); /* LIE! */
    }
    break;
```

Step 2: Rebuild and load:

```
% make clean && make
% sudo kldload ./demo.ko
% ls -l /dev/demo
crw-rw-rw-  1 root  wheel  0x5e Nov 14 17:00 /dev/demo
```

Step 3: Unload the module:

```
% sudo kldunload demo
% dmesg | tail -1
demo: Device /dev/demo destroyed  # Lied!
```

Step 4: Check if device still exists:

```
% ls -l /dev/demo
crw-rw-rw-  1 root  wheel  0x5e Nov 14 17:00 /dev/demo  # STILL THERE!
```

Step 5: Try to use the orphaned device:

```
% cat /dev/demo
```

Symptoms you might see:

- Hang (cat blocks forever)
- Kernel panic (jumps to unmapped memory)
- Error message about invalid device

Step 6: Check for leaks:

```
% vmstat -m | grep cdev
cdev      10      15K      -      1442      16,32,64
```

The count may be higher than before you started.

Step 7: Reboot to clean up:

```
% sudo reboot
```

What you learned:

- **Orphaned device nodes** persist in /dev even when driver unloads
- Trying to use orphaned devices causes **undefined behavior** (crash, hang, or errors)
- This is a **resource leak** - the cdev structure and device node are never freed
- **Always call `destroy_dev()`** in cleanup path

6.21.6.2 Experiment 1B: Fix it properly

Step 1: Restore the `destroy_dev()` call:

```
case MOD_UNLOAD:
    if (demo_dev != NULL) {
        destroy_dev(demo_dev); /* RESTORED */
        demo_dev = NULL;
        printf("demo: Device /dev/demo destroyed\n");
    }
    break;
```

Step 2: Rebuild, load, test, unload:

```
% make clean && make
% sudo kldload ./demo.ko
% ls -l /dev/demo          # Exists
% cat /dev/demo            # Works
% sudo kldunload demo
% ls -l /dev/demo          # GONE - correct!
```

Success: Device node properly cleaned up.

6.21.7 Part 2: The Wrong Order Bug

6.21.7.1 Experiment 2A: Free before destroying

Goal: See why cleanup order matters.

Step 1: Add a malloc'd buffer to demo.c:

After `static struct cdev *demo_dev = NULL;`, add:

```
static char *demo_buffer = NULL;
```

Step 2: Allocate in MOD_LOAD:

```
case MOD_LOAD:
    /* Allocate a buffer */
    demo_buffer = malloc(128, M_TEMP, M_WAITOK | M_ZERO);
    printf("demo: Allocated buffer at %p\n", demo_buffer);

    demo_dev = make_dev(...);
    /* ... rest of load code ... */
    break;
```

Step 3: WRONG CLEANUP - free before destroy:

```
case MOD_UNLOAD:
    /* BUG: Free while device is still accessible! */
    if (demo_buffer != NULL) {
        free(demo_buffer, M_TEMP);
        demo_buffer = NULL;
        printf("demo: Freed buffer\n");
    }

    /* Device is still alive and can be opened! */
    if (demo_dev != NULL) {
```

```
    destroy_dev(demo_dev);  
    demo_dev = NULL;  
}  
break;
```

Step 4: Rebuild and test:

```
% make clean && make  
% sudo kldload ./demo.ko
```

Step 5: While module is loaded, in another terminal:

```
% ( sleep 2; cat /dev/demo ) & # Start delayed cat  
% sudo kldunload demo          # Try to unload
```

Race condition:

1. kldunload starts
2. Your code frees demo_buffer
3. destroy_dev() called
4. Meanwhile, cat opened /dev/demo (device still existed!)
5. demo_read() tries to use freed demo_buffer
6. **Use-after-free crash** or corrupted data

Symptoms:

- Kernel panic: “page fault in kernel mode”
- Corrupted output
- Hang

Step 6: Reboot to recover.

6.21.7.2 Experiment 2B: Fix the ordering

Correct order: Make device invisible FIRST, then free resources.

```
case MOD_UNLOAD:  
    /* CORRECT: Destroy device first */  
    if (demo_dev != NULL) {  
        destroy_dev(demo_dev); /* Waits for all operations */  
        demo_dev = NULL;  
    }  
  
    /* Now safe - no one can call our functions */
```

```

if (demo_buffer != NULL) {
    free(demo_buffer, M_TEMP);
    demo_buffer = NULL;
    printf("demo: Freed buffer\n");
}
break;

```

Why this works:

1. `destroy_dev()` removes `/dev/demo` from filesystem
2. `destroy_dev()` **waits** for any in-progress operations (like active reads)
3. After `destroy_dev()` returns, **no new operations can start**
4. **Now** it's safe to free `demo_buffer` - nothing can access it

Step 7: Rebuild and test:

```

% make clean && make
% sudo kldload ./demo.ko
% ( sleep 2; cat /dev/demo ) &
% sudo kldunload demo
# Works safely - no crash

```

What you learned:

- **Cleanup order is critical:** Device invisible -> wait for operations -> free resources
- `destroy_dev()` provides synchronization (waits for operations)
- **Reverse order** of initialization: Last allocated, first freed

6.21.8 Part 3: The NULL Pointer Bug

6.21.8.1 Experiment 3A: Missing NULL check

Goal: See why NULL checks matter.

Step 1: Make `make_dev()` fail by using an existing name:

Load demo module, then try to load again in `MOD_LOAD`:

```

case MOD_LOAD:
    demo_dev = make_dev(&demo_cdevsw, 0, UID_ROOT, GID_WHEEL, 0666, "demo");

    /* BUG: Don't check for NULL! */
    printf("demo: Device created at %p\n", demo_dev); /* Might print NULL! */
    /* Continuing even though make_dev failed... */
    break;

```


Or simulate failure:

```
case MOD_LOAD:
    demo_dev = NULL; /* Simulate make_dev failure */
    /* BUG: No check! */
    printf("demo: Device created at %p\n", demo_dev);
    break;
```

Step 2: Try to unload without NULL check:

```
case MOD_UNLOAD:
    /* BUG: No NULL check! */
    destroy_dev(demo_dev); /* Passing NULL to destroy_dev! */
    break;
```

Step 3: Test:

```
% make clean && make
% sudo kldload ./demo.ko
# Module "loads" but device wasn't created
% sudo kldunload demo
# Might panic or crash
```

Symptoms:

- Kernel panic in `destroy_dev`
- “panic: bad address”
- System hang

6.21.8.2 Experiment 3B: Proper NULL checking

```
case MOD_LOAD:
    demo_dev = make_dev(&demo_cdevsw, 0, UID_ROOT, GID_WHEEL, 0666, "demo");

    /* ALWAYS check return value! */
    if (demo_dev == NULL) {
        printf("demo: Failed to create device node\n");
        return (ENXIO); /* Abort load */
    }

    printf("demo: Device /dev/demo created successfully\n");
    break;
```

```

case MOD_UNLOAD:
    /* ALWAYS check for NULL before using pointer! */
    if (demo_dev != NULL) {
        destroy_dev(demo_dev);
        demo_dev = NULL; /* Clear pointer for safety */
    }
    break;

```

Defensive programming rules:

1. **Check every allocation:** `if (ptr == NULL) handle_error();`
2. **Check before freeing:** `if (ptr != NULL) free(ptr);`
3. **Clear after freeing:** `ptr = NULL;` (defense against use-after-free)

6.21.9 Part 4: The Allocation Failure Bug

6.21.9.1 Experiment 4: Handling malloc failures

Goal: Learn to handle `M_NOWAIT` allocation failures.

Step 1: Add allocation to attach:

```

case MOD_LOAD:
    /* Allocate with M_NOWAIT - can fail! */
    demo_buffer = malloc(128, M_TEMP, M_NOWAIT | M_ZERO);

    /* BUG: Don't check for NULL */
    strcpy(demo_buffer, "Hello"); /* CRASH if malloc failed! */

    demo_dev = make_dev(&demo_cdevsw, 0, UID_ROOT, GID_WHEEL, 0666, "demo");
    /* ... */
    break;

```

If malloc fails (rare but possible):

```

panic: page fault while in kernel mode
fault virtual address = 0x0
fault code = supervisor write data
instruction pointer = 0x8:0xffffffff12345678
current process = 1234 (kldload)

```

Step 2: Fix with proper error handling:

```

case MOD_LOAD:
    /* Allocate with M_NOWAIT */
    demo_buffer = malloc(128, M_TEMP, M_NOWAIT | M_ZERO);
    if (demo_buffer == NULL) {
        printf("demo: Failed to allocate buffer\n");
        return (ENOMEM); /* Out of memory */
    }

    /* Now safe to use */
    strcpy(demo_buffer, "Hello");

    /* Create device */
    demo_dev = make_dev(&demo_cdevsw, 0, UID_ROOT, GID_WHEEL, 0666, "demo");
    if (demo_dev == NULL) {
        printf("demo: Failed to create device node\n");
        /* BUG: Forgot to free demo_buffer! */
        return (ENXIO);
    }

    printf("demo: Device created successfully\n");
    break;

```

Wait, there's still a bug! If `make_dev()` fails, we return without freeing `demo_buffer`.

Step 3: Fix with complete error unwinding:

```

case MOD_LOAD:
    int error = 0;

    /* Allocate buffer */
    demo_buffer = malloc(128, M_TEMP, M_NOWAIT | M_ZERO);
    if (demo_buffer == NULL) {
        printf("demo: Failed to allocate buffer\n");
        return (ENOMEM);
    }

    strcpy(demo_buffer, "Hello");

    /* Create device */
    demo_dev = make_dev(&demo_cdevsw, 0, UID_ROOT, GID_WHEEL, 0666, "demo");
    if (demo_dev == NULL) {
        printf("demo: Failed to create device node\n");
        error = ENXIO;
        goto fail;
    }

```

```
printf("demo: Device created successfully\n");
return (0); /* Success */
```

fail:

```
/* Error cleanup - undo everything we did */
if (demo_buffer != NULL) {
    free(demo_buffer, M_TEMP);
    demo_buffer = NULL;
}
return (error);
```

Error unwinding pattern:

1. Each allocation step can fail
2. On failure, **undo everything done before**
3. Common pattern: use `goto fail` to centralize cleanup
4. Free in reverse order of allocation

6.21.10 Part 5: Complete Example with Full Error Handling

Here's a template showing all best practices:

```
case MOD_LOAD:
    int error = 0;

    /* Step 1: Allocate buffer */
    demo_buffer = malloc(128, M_TEMP, M_WAITOK | M_ZERO);
    if (demo_buffer == NULL) { /* Paranoid - M_WAITOK shouldn't fail */
        error = ENOMEM;
        goto fail_0; /* Nothing to clean up yet */
    }

    strcpy(demo_buffer, "Initialized");

    /* Step 2: Create device node */
    demo_dev = make_dev(&demo_cdevsw, 0, UID_ROOT, GID_WHEEL, 0666, "demo");
    if (demo_dev == NULL) {
        printf("demo: Failed to create device node\n");
        error = ENXIO;
        goto fail_1; /* Need to free buffer */
    }

    /* Success! */
```

```
    printf("demo: Module loaded successfully\n");
    return (0);

/* Error unwinding - labels in reverse order of operations */
fail_1:
    /* Failed after allocating buffer */
    free(demo_buffer, M_TEMP);
    demo_buffer = NULL;
fail_0:
    /* Failed before allocating anything */
    return (error);
```

Why this pattern works:

- Each fail_N label knows exactly what was allocated up to that point
- Cleanup happens in reverse order (last allocated, first freed)
- Single return point for errors makes debugging easier
- All error paths properly clean up

6.21.11 Debugging Checklist: Finding Driver Bugs

When your driver misbehaves, check these systematically:

6.21.11.1 1. Check dmesg for kernel messages

```
% dmesg | tail -20
% dmesg | grep -i panic
% dmesg | grep -i "page fault"
```

Look for:

- Panic messages
- “sleeping with lock held”
- “lock order reversal”
- Your driver’s printf messages

6.21.11.2 2. Check for resource leaks

Before loading module:

```
% vmstat -m | grep cdev > before.txt
```

After load + unload:

```
% vmstat -m | grep cdev > after.txt  
% diff before.txt after.txt
```

If counts increased, you have a leak.

6.21.11.3 3. Check for orphaned devices

```
% ls -l /dev/ | grep demo
```

If /dev/demo exists after unload, you forgot `destroy_dev()`.

6.21.11.4 4. Test unload under load

```
% ( sleep 10; cat /dev/demo ) &  
% sudo kldunload demo
```

Should wait for cat to finish. If it crashes, you have a race condition.

6.21.11.5 5. Check module state

```
% kldstat -v | grep demo
```

Shows dependencies and references.

6.21.12 Success Criteria

- Observed orphaned device node (Experiment 1A)
- Fixed with proper `destroy_dev()` (Experiment 1B)
- Observed use-after-free crash (Experiment 2A)
- Fixed with correct cleanup order (Experiment 2B)
- Understood NULL pointer dangers (Experiment 3)
- Implemented proper NULL checking (Experiment 3B)
- Learned error unwinding pattern (Experiment 4)
- Can identify resource leaks with `vmstat`
- Can debug with `dmesg`

6.21.13 What You Learned

Bug types:

- Resource leaks (forgotten `destroy_dev`)
- Use-after-free (wrong cleanup order)
- NULL pointer dereference (missing checks)
- Memory leaks (failed error unwinding)

Defensive programming:

- Always check return values
- Always NULL-check before using pointers
- Clean up in reverse order of initialization
- Clear pointers after freeing (`ptr = NULL`)
- Use `goto` for error unwinding

Debugging techniques:

- Using `dmesg` to track operations
- Using `vmstat` to detect leaks
- Testing unload under load
- Deliberately introducing bugs to understand symptoms

Patterns to follow:

```
/* Allocation */
ptr = malloc(size, type, M_WAITOK);
if (ptr == NULL) {
    error = ENOMEM;
    goto fail;
}

/* Device creation */
dev = make_dev(...);
if (dev == NULL) {
    error = ENXIO;
    goto fail_after_malloc;
}

/* Success */
return (0);

/* Error cleanup */
fail_after_malloc:
    free(ptr, type);
    ptr = NULL;
```

```
fail:
    return (error);
```

6.21.14 Lab Logbook Entry Template

Lab 4 Complete: [Date]

Time taken: ____ minutes

Experiments conducted:

- Orphaned device: ☐ Observed ☐ Fixed
- Wrong cleanup order: ☐ Observed crash ☐ Fixed
- NULL pointer bug: ☐ Observed ☐ Fixed
- Error unwinding: ☐ Implemented ☐ Tested

Most valuable insight:

[What "clicked" about error handling?]

Bugs I've seen before:

[Have you made similar mistakes in userspace code?]

Defensive programming rules I'll remember:

1. [e.g., "Always check malloc return"]
2. [e.g., "Cleanup in reverse order"]
3. [e.g., "Set pointers to NULL after free"]

Debugging techniques learned:

[Which debugging method was most useful?]

Ready for Chapter 7:

- ☐ Yes - I understand error handling
- ☐ Need more practice - I'll review the error patterns again

6.22 Labs Summary and Next Steps

Congratulations! You've completed all four labs. Here's what you've accomplished:

6.22.1 Lab Progression Summary

Lab	What You Built	Key Skill
Lab 1	Navigation skills	Read and understand driver code
Lab 2	Minimal module	Build and load kernel modules
Lab 3	Character device	Create /dev nodes, implement I/O
Lab 4	Error handling	Defensive programming, debugging

6.22.2 Key Concepts Mastered

Module lifecycle:

- MOD_LOAD -> initialize
- MOD_UNLOAD -> cleanup
- DECLARE_MODULE registration

Device framework:

- cdevsw as method dispatch table
- make_dev() to create /dev entries
- destroy_dev() for cleanup + synchronization

Data transfer:

- uiomove() for safe user-kernel copying
- uio structure for I/O requests
- uio_resid tracking

Error handling:

- NULL checking all allocations
- Reverse-order cleanup
- Error unwinding with goto
- Resource leak prevention

Debugging:

- Using dmesg for kernel logs
- vmstat for resource tracking
- Testing under load

6.22.3 Your Driver Development Toolkit

You now have a solid foundation of:

1. **Pattern recognition:** You can look at any FreeBSD driver and identify its structure
2. **Practical skills:** You can build, load, test, and debug kernel modules
3. **Safety knowledge:** You understand common bugs and how to avoid them
4. **Debugging ability:** You can diagnose problems using system tools

6.22.4 Celebrate Your Achievement!

You've completed hands-on labs that many developers skip. You didn't just read about drivers, you **built** them, **broke** them, and **fixed** them. This experiential learning is invaluable.

6.23 Wrapping Up

Congratulations! You've completed a comprehensive tour of FreeBSD driver anatomy. Let's recap what you've learned and where we're heading next.

6.23.1 What You Now Know

Vocabulary - You can speak the language of FreeBSD drivers:

- **newbus**: The device framework (probe/attach/detach)
- **devclass**: Grouping of related devices
- **softc**: Per-device private data structure
- **cdevsw**: Character device switch (entry point table)
- **ifnet**: Network interface structure
- **GEOM**: Storage layer architecture
- **devfs**: Dynamic device filesystem

Structure - You recognize driver patterns instantly:

- Probe functions check device IDs and return priority
- Attach functions initialize hardware and create device nodes
- Detach functions clean up in reverse order
- Method tables map kernel calls to your functions
- Module declarations register with the kernel

Lifecycle - You understand the flow:

1. Bus enumeration discovers hardware
2. Probe functions compete for devices
3. Attach functions initialize winners
4. Devices operate (read/write, transmit/receive)
5. Detach functions clean up on unload

Entry points - You know how user programs reach your driver:

- Character devices: open/close/read/write/ioctl via `/dev`
- Network interfaces: transmit/receive via network stack
- Storage devices: bio requests via GEOM/CAM

6.23.2 What You Can Now Do

- Navigate the FreeBSD kernel source tree confidently

- Recognize common driver patterns (probe/attach/detach, cdevsw)
- Understand probe/attach/detach lifecycle
- Build kernel modules with proper Makefiles
- Load and unload modules safely
- Create character device nodes with appropriate permissions
- Implement basic I/O operations (open/close/read/write)
- Use `uiomove()` correctly for user-kernel data transfer
- Handle errors and clean up resources properly
- Debug with `dmesg` and system tools
- Avoid common pitfalls (resource leaks, wrong cleanup order, NULL pointers)

6.23.3 Mindset Shift

Notice the shift in this chapter:

- **Chapter 1-5:** Foundations (UNIX, C, kernel C)
- **Chapter 6** (this one): Structure and patterns (recognition)
- **Chapter 7+:** Implementation (building)

You’ve crossed a threshold. You’re no longer just learning concepts, you’re ready to write real kernel code. This is exciting and a little intimidating, and that’s exactly right.

6.23.4 Final Thoughts

Driver development is like learning a musical instrument. At first, the patterns feel foreign and complex. But with practice, they become second nature. You’ll start to see probe/attach/detach everywhere you look. You’ll recognize `cdevsw` instantly. You’ll know what “allocate resources, check for errors, clean up on failure” means without thinking.

Trust the process. The labs were just the beginning. In Chapter 7, you’ll write more code, make mistakes, debug them, and build confidence. By Chapter 8, driver structure will feel natural.

6.23.5 Before You Move On

Take a moment to:

- **Review your lab logbook** - What surprised you? What clicked?
- **Revisit any confusing sections** - Now that you’ve done the labs, re-reading makes more sense
- **Browse one more driver** - Pick any from `/usr/src/sys/dev` and see how much you recognize

6.23.6 Looking Ahead

Chapter 6 was the last foundational chapter of Part 1. You now have a complete mental model of how a FreeBSD driver is shaped, from the moment the bus enumerates a device, through probe, attach, operation, and detach, all the way out to `/dev` and `ifconfig`.

The next chapter, **Chapter 7: Writing Your First Driver**, puts that model to work. You will build a pseudo-device called `myfirst`, attach it cleanly through Newbus, create a `/dev/myfirst0` node, expose a read-only `sysctl`, log lifecycle events, and detach without leaks. The goal is not a fancy driver, it is a disciplined one, the kind of skeleton every production driver grows from.

Everything you practised in this chapter, the `cdevsw` shape, the probe/attach/detach rhythm, the unwinding pattern, the rule to always release resources in reverse order, will show up again in Chapter 7 as code you type yourself. Keep your lab logbook close, keep `/usr/src/sys/dev/null/null.c` bookmarked as a reference skeleton, and when you turn the page, you will already know most of what you are about to build.

6.24 Part 1 Checkpoint

Part 1 has carried you from “what even is UNIX” to “I can read a small driver and name its pieces.” Before Chapter 7 asks you to type and load a real module, pause and confirm that the foundation feels steady under your feet. Part 2 builds directly on every skill that the first six chapters gathered.

By the end of Part 1 you should be able to install, configure, and snapshot a FreeBSD working lab, track its source tree under version control, and keep a disciplined logbook of what you changed and why. You should be able to drive the FreeBSD command line for ordinary development work, which means moving around the filesystem, inspecting processes, reading and adjusting permissions, installing packages, following logs, and writing short shell scripts that survive unusual filenames. You should also be able to read and write kernel-style C without flinching at its dialect, including types and qualifiers, bit flags, the preprocessor, pointers and arrays, function pointers, bounded strings, and the kernel-side allocators and logging helpers that replace `malloc(3)` and `printf(3)`. And you should be able to look at any driver under `/usr/src/sys/dev` and name its pieces: which function is the probe, which is the attach, which is the detach, where the softc lives, which entry points the character switch provides, and what resources the attach path is acquiring.

If any of those still feels like a lookup rather than a habit, the labs that anchor them are worth a second pass:

- Lab discipline and source navigation: the hands-on labs across Chapter 2 (shell, files, processes, scripting) and the install-and-snapshot walk-through in Chapter 3.
- C for the kernel: Chapter 4 Lab 4 (Function Pointer Dispatch, a mini `cdevsw`) and Lab 5 (Fixed-Size Circular Buffer), both of which preview patterns you will meet again in every driver.
- Kernel C dialect: Chapter 5 Lab 1 (Safe Memory Allocation and Cleanup) and Lab 2 (User-Kernel Data Exchange), which teach the two boundaries every driver crosses.
- Driver anatomy: Chapter 6 Lab 1 (Explore the Driver Map), Lab 2 (Minimal Module with Logs Only), and Lab 3 (Create and Remove a Device Node).

Part 2 will expect a working FreeBSD lab with `/usr/src` installed, a kernel you can build and boot, and the habit of reverting to a clean snapshot after each experiment. It will expect enough comfort with kernel C that a `struct cdevsw`, a `d_read` handler signature, or a labelled-goto cleanup pattern does not stop you. It will also expect the probe/attach/detach rhythm to be held firmly in mind, so that Chapter 7 can turn that rhythm into code you type yourself. If those three hold, you are ready to cross from recognition to authorship. If one wobbles, the quiet hour spent now saves a bewildering afternoon later.

6.25 Challenge Exercises (Optional)

These optional exercises deepen your understanding and build confidence. They're more open-ended than labs but still safe for beginners. Complete as many as you like before moving to Chapter 7.

6.25.1 Challenge 1: Trace a Lifecycle in dmesg

Goal: Capture and annotate real driver lifecycle messages.

Instructions:

1. Choose a driver that's loadable as a module (e.g., `if_em`, `snd_hda`, `usb`)
2. Set up logging:

```
% tail -f /var/log/messages > ~/driver_lifecycle.log &
```

3. Load the driver:

```
% sudo kldload if_em
```

4. Watch the attach sequence in real time

5. Unload the driver:

```
% sudo kldunload if_em
```

6. Stop logging (kill the tail process)

7. Annotate the log file:

- Mark where probe was called
- Mark where attach happened
- Mark resource allocations
- Mark where detach cleaned up

8. Write a one-page summary explaining the lifecycle you observed

Success criteria: Your annotated log shows clear understanding of when each lifecycle phase occurred.

6.25.2 Challenge 2: Map the Entry Points

Goal: Completely document a driver's `cdevsw` structure.

Instructions:

1. Open `/usr/src/sys/dev/null/null.c`
2. Create a table:

Entry Point	Function Name	Present?	What It Does
d_open	?	?	?
d_close	?	?	?
d_read	?	?	?
d_write	?	?	?
d_ioctl	?	?	?
d_poll	?	?	?
d_mmap	?	?	?

3. Fill in the table
4. For missing entry points, explain why they're not needed
5. For present entry points, describe what they do in 1-2 sentences
6. Repeat for `/usr/src/sys/dev/led/led.c`
7. Compare the two tables: What's similar? What's different? Why?

Success criteria: Your tables are accurate and your explanations demonstrate understanding.

6.25.3 Challenge 3: Classification Drill

Goal: Practice identifying driver families by examining source code.

Instructions:

1. Choose **five random drivers** from `/usr/src/sys/dev/`

```
% ls /usr/src/sys/dev | shuf | head -5
```

2. For each driver, create an entry in your logbook:
 - Driver name
 - Primary source file
 - Classification (character, network, storage, bus, or mixed)
 - Evidence (how did you determine the classification?)
 - Purpose (what does this driver do?)
3. Verification: Use `man 4 <drivername>` to confirm your classification

Example entry:

```
Driver: led
File: sys/dev/led/led.c
Classification: Character device
Evidence: Has cdevsw structure, creates /dev/led/* nodes, no ifnet or GEOM
Purpose: Control system LEDs (keyboard lights, chassis indicators)
Man page: man 4 led (confirmed)
```

Success criteria: Correctly classified all five, with clear evidence for each.

6.25.4 Challenge 4: Error Code Audit

Goal: Understand error handling patterns in real drivers.

Instructions:

1. Open `/usr/src/sys/dev/uart/uart_core.c`
2. Find the `uart_bus_attach()` function
3. List every error code returned (ENOMEM, ENXIO, EIO, etc.)
4. For each, note:
 - What condition triggered it
 - What resources were freed before returning
 - Whether cleanup was complete
5. Repeat for `/usr/src/sys/dev/ahci/ahci.c` (`ahci_attach` function)
6. Write a short essay (1-2 pages):
 - Common error handling patterns you observed
 - How drivers ensure no resource leaks
 - Best practices you can apply to your own code

Success criteria: Your essay demonstrates understanding of proper error unwinding.

6.25.5 Challenge 5: Dependency Detective

Goal: Understand module dependencies and load order.

Instructions:

1. Find a driver that declares `MODULE_DEPEND`

```
% grep -r "MODULE_DEPEND" /usr/src/sys/dev/usb | head -5
```
2. Pick one example (e.g., a USB driver)
3. Open the source file and find all `MODULE_DEPEND` declarations
4. For each dependency:
 - What module does it depend on?
 - Why is this dependency needed? (What functions/types from that module are used?)
 - What would happen if you tried to load without the dependency?
5. Test it:

```
% sudo kldload <dependency_module>
% sudo kldload <your_driver>
% kldstat
```

6. Try to unload the dependency while your driver is loaded:

```
% sudo kldunload <dependency_module>
```

What happens? Why?

7. Document your findings: Draw a dependency graph showing the relationships.

Success criteria: You can explain why each dependency exists and predict load order.

Summary

These challenges develop:

- **Challenge 1:** Real-world lifecycle observation
- **Challenge 2:** Entry point mastery
- **Challenge 3:** Pattern recognition across drivers
- **Challenge 4:** Error handling discipline
- **Challenge 5:** Dependency understanding

Optional: Share your challenge results in the FreeBSD forums or mailing lists. The community loves seeing newcomers take on harder problems.

6.26 Summary Reference Table - Driver Building Blocks at a Glance

This one-screen cheat sheet maps concepts to implementations. Bookmark this page for quick reference while working on Chapter 7 and beyond.

Concept	What It Is	Typical API/Structure	Where in Tree	When You'll Use It
device_t	Opaque device handle	device_t dev	<sys/bus.h>	Every driver function (probe/attach/detach)
softc	Per-device private data	struct mydriver_softc	You define it	Store state, resources, locks
devclass	Device class grouping	devclass_t	<sys/bus.h>	Auto-managed by DRIVER_MODULE
cdevsw	Character device switch	struct cdevsw	<sys/conf.h>	Character device entry points
d_open	Open handler	d_open_t	In your cdevsw	Initialize per-session state
d_close	Close handler	d_close_t	In your cdevsw	Clean up per-session state

Concept	What It Is	Typical API/Structure	Where in Tree	When You'll Use It
d_read	Read handler	d_read_t	In your cdevsw	Transfer data to user
d_write	Write handler	d_write_t	In your cdevsw	Accept data from user
d_ioctl	Ioctl handler	d_ioctl_t	In your cdevsw	Configuration and control
uiomove	Copy to/from user	int uiomove(...)	<sys/uio.h>	In read/write handlers
make_dev	Create device node	struct cdev *make_dev(...)	<sys/conf.h>	In attach (character devices)
destroy_dev	Remove device node	void destroy_dev(...)	<sys/conf.h>	In detach
ifnet (if_t)	Network interface	if_t	<net/if_var.h>	Network drivers
ether_ifattach	Register Ethernet if	void ether_ifattach(...)	<net/ethernet.h>	Network driver attach
ether_ifdetach	Unregister Ethernet if	void ether_ifdetach(...)	<net/ethernet.h>	Network driver detach
GEOM provider	Storage provider	struct g_provider	<geom/geom.h>	Storage drivers
bio	Block I/O request	struct bio	<sys/bio.h>	Storage I/O handling
bus_alloc_resource	Allocate resource	struct resource *	<sys/bus.h>	Attach (memory, IRQ, etc.)
bus_release_resource	Release resource	void	<sys/bus.h>	Detach cleanup
bus_space_read	Read register	uint32_t bus_space_read_4(...)	<machine/bus.h>	Hardware register access
bus_space_write	Write register	void bus_space_write_4(...)	<machine/bus.h>	Hardware register access
bus_setup_intr	Register interrupt	int bus_setup_intr(...)	<sys/bus.h>	Attach (interrupt setup)
bus_teardown_intr	Unregister interrupt	int bus_teardown_intr(...)	<sys/bus.h>	Detach cleanup
device_printf	Device-specific log	void device_printf(...)	<sys/bus.h>	All driver functions
device_get_softc	Retrieve softc	void *device_get_softc(device_t)	<sys/bus.h>	First line of most functions

Concept	What It Is	Typical API/Structure	Where in Tree	When You'll Use It
device_set_desc	Set device description	void device_set_desc(...)	<sys/bus.h>	In probe function
DRIVER_MODULE	Module driver	Macro	<sys/module.h>	Once per driver (end of file)
MODULE_VERSION	version	Macro	<sys/module.h>	Once per driver
MODULE_DEPEND	dependency	Macro	<sys/module.h>	If you depend on other modules
DEVMETHOD	Map method to function	Macro	<sys/bus.h>	In method table
DEVMETHOD_END	End method table	Macro	<sys/bus.h>	Last entry in method table
mtx	Mutex lock	struct mtx	<sys/mutex.h>	Protect shared state
mtx_init	Initialize mutex	void mtx_init(...)	<sys/mutex.h>	In attach
mtx_destroy	Destroy mutex	void mtx_destroy(...)	<sys/mutex.h>	In detach
mtx_lock	Acquire lock	void mtx_lock(...)	<sys/mutex.h>	Before accessing shared data
mtx_unlock	Release lock	void mtx_unlock(...)	<sys/mutex.h>	After accessing shared data
malloc	Allocate memory	void *malloc(...)	<sys/malloc.h>	Dynamic allocation
free	Free memory	void free(...)	<sys/malloc.h>	Cleanup
M_WAITOK	Wait for memory	Flag	<sys/malloc.h>	malloc flag (can sleep)
M_NOWAIT	Don't wait	Flag	<sys/malloc.h>	malloc flag (returns NULL if unavailable)

6.26.1 Quick Lookup by Task

Need to...	Use This	Man Page
Create a character device	make_dev()	make_dev(9)
Read/write hardware registers	bus_space_read/write_N()	bus_space(9)
Allocate hardware resources	bus_alloc_resource()	bus_alloc_resource(9)
Set up interrupts	bus_setup_intr()	bus_setup_intr(9)

Need to...	Use This	Man Page
Copy data to/from user	uiomove()	uio(9)
Log a message	device_printf()	device(9)
Protect shared data	mtx_lock() / mtx_unlock()	mutex(9)
Register a driver	DRIVER_MODULE()	DRIVER_MODULE(9)

6.26.2 Probe/Attach/Detach Quick Reference

```

/* Probe - Check if we can handle this device */
static int mydrv_probe(device_t dev) {
    /* Check IDs, return BUS_PROBE_DEFAULT or ENXIO */
}

/* Attach - Initialize device */
static int mydrv_attach(device_t dev) {
    sc = device_get_softc(dev);
    /* Allocate resources */
    /* Initialize hardware */
    /* Create device node or register interface */
    return (0); /* or error code */
}

/* Detach - Clean up */
static int mydrv_detach(device_t dev) {
    sc = device_get_softc(dev);
    /* Reverse order of attach */
    /* Check pointers before freeing */
    /* Set pointers to NULL after freeing */
    return (0); /* or EBUSY if can't detach */
}

```

7 Writing Your First Driver

7.1 Reader Guidance & Outcomes

Welcome to Part 2. If Part 1 was your foundation, learning the environment, the language, and the architecture, **Part 2 is where you build**. This chapter marks the moment you stop reading about drivers and start writing one.

But let's be clear about what we're building and, just as importantly, what we're not building yet. This chapter follows a **discipline-first** approach: you'll write a minimal driver that attaches cleanly, logs properly, creates a simple user surface, and detaches without leaks. No fancy I/O, no hardware register access, no interrupt handling. Those come later, once the discipline is second nature.

7.1.1 What You'll Build

By the time you move on to the next chapter, you will have a working FreeBSD 14.3 driver called `myfirst` that:

- **Attaches as a pseudo-device** using the Newbus framework
- **Creates a `/dev/myfirst0` node** (stubbed, read-only preview)
- **Exposes a read-only `sysctl`** showing basic runtime state
- **Logs lifecycle events** cleanly with `device_printf()`
- **Handles errors** with a single-label unwind pattern
- **Detaches cleanly** with no resource leaks or dangling pointers

This driver won't do anything exciting yet. It won't read from hardware, it won't handle interrupts, and it won't process packets or blocks. What it *will* do is demonstrate **lifecycle discipline** the foundation every production driver depends on.

7.1.2 What You'll Not Build (Yet)

This chapter deliberately defers several important topics so you can master structure before complexity:

- **Full I/O semantics:** `read(2)` and `write(2)` will be stubbed. Real read and write paths arrive in Chapter 9, after Chapter 8 has covered device-file policy and userland visibility.
- **Hardware interaction:** No register access, no DMA, no interrupts. Those are covered in **Part 4** when you have a solid foundation.
- **PCI/USB/ACPI specifics:** We use a pseudo-device (no bus dependency) in this chapter. Bus-specific attachment patterns appear in Part 4 (PCI, interrupts, DMA) and Part 6 (USB, storage, network).
- **Locking and concurrency:** You'll see a mutex in the softc, but we won't exercise complex concurrent paths until **Part 3**.

- **Advanced sysctls:** Just one read-only node for now. Larger sysctl trees, write handlers, and tunables come back in Part 5.

Why this matters: Trying to learn everything at once leads to confusion. By keeping the scope narrow, you'll understand *why* each piece exists before adding the next layer.

7.1.3 Estimated Time Investment

- **Reading only:** 2-3 hours to absorb concepts and code walkthroughs
- **Reading + typing examples:** 4-5 hours if you type the driver code yourself
- **Reading + all four labs:** 5-7 hours, including build, test, and verification cycles
- **Optional challenges:** Add 2-3 hours for deep-dive exercises

Recommended pace: Break this into two or three sessions. Session 1 through the scaffold and Newbus basics, session 2 through logging and error handling, session 3 for labs and smoke tests.

7.1.4 Prerequisites

Before starting, ensure you have:

- **FreeBSD 14.3** running in your lab (VM or bare metal)
- **Chapters 1-6 completed** (especially Chapter 2's lab setup and Chapter 6's anatomy tour)
- **/usr/src installed** with FreeBSD 14.3 sources matching your running kernel
- **Basic C fluency** from Chapter 4
- **Kernel programming awareness** from Chapter 5

Check your kernel version:

```
% freebsd-version -k
14.3-RELEASE
```

If this doesn't match, revisit Chapter 2's setup guidance.

7.1.5 Learning Outcomes

By completing this chapter, you will be able to:

- Scaffold a minimal FreeBSD driver from scratch
- Implement and explain probe/attach/detach lifecycle methods
- Define and use a driver softc structure safely
- Create and destroy /dev nodes using `make_dev_s()`
- Add a read-only sysctl for observability
- Handle errors with disciplined unwinding (single fail: label pattern)
- Build, load, test, and unload your driver reliably
- Identify and fix common beginner mistakes (resource leaks, null dereferences, missing cleanup)

7.1.6 Success Criteria

You'll know you've succeeded when:

- `kldload ./myfirst.ko` completes without errors
- `dmesg -a` shows your attach message
- `ls -l /dev/myfirst0` displays your device node
- `sysctl dev.myfirst.0` returns your driver's state
- `kldunload myfirst` cleans up without leaks or panics
- You can repeat the load/unload cycle reliably
- A simulated attach failure unwinds cleanly (negative-path test)

7.1.7 Where This Chapter Fits

You're entering **Part 2 - Building Your First Driver**, the bridge from theory to practice:

- **Chapter 7 (this chapter)**: scaffold a minimal driver with clean attach/detach
- **Chapter 8**: wire up real `open()`, `close()`, and device-file semantics
- **Chapter 9**: implement basic `read()` and `write()` paths
- **Chapter 10**: handle buffering, blocking, and poll/select

Each chapter builds on the previous, adding one layer of functionality at a time.

7.1.8 A Note on “Hello World” Versus “Hello Production”

You've probably seen “hello world” kernel modules before: a `MOD_LOAD` event handler that prints a message. That's fine for checking if the build system works, but it's not a driver. It doesn't attach to anything, doesn't create user surfaces, and teaches almost nothing about lifecycle discipline.

This chapter's `myfirst` driver is different. It's still minimal, but it follows the patterns you'll see in every production FreeBSD driver:

- Registers with Newbus
- Implements probe/attach/detach correctly
- Manages resources (even if they're trivial)
- Cleans up reliably

Think of `myfirst` as **hello production**, not hello world. The jump from toy to tool starts here.

7.1.9 How to Use This Chapter

1. **Read sequentially**: Each section builds on the previous. Don't skip ahead.
2. **Type the code yourself**: Muscle memory matters. Copying snippets is fine, but typing cements patterns.
3. **Complete the labs**: They're checkpoints, not optional extras. Each lab validates understanding before moving forward.
4. **Use the capstone checklist**: Before declaring victory, run through the smoke-test checklist (near the end of this chapter). It catches common mistakes.
5. **Keep a log**: Record what worked, what failed, and what you learned. Future-you will thank you.

7.1.10 A Word About Mistakes

You *will* encounter errors. You'll forget to initialize a pointer, you'll skip a cleanup step, you'll typo a function name. That's expected and **healthy**. Every error is a chance to practice debugging, reading logs, and understanding cause and effect.

When something breaks:

- Read the full error message. FreeBSD's kernel messages are detailed.
- Check `dmesg -a` for lifecycle events.
- Use the troubleshooting decision tree (section later in this chapter).
- Revisit the relevant section and compare your code to the examples.

Don't rush past errors. They're teaching moments.

7.1.11 Let's Begin

You've completed the foundation. You've walked through real drivers in Chapter 6. Now it's time to **build your own**. Let's start with the project scaffold.

7.2 Project Scaffold (KLD skeleton)

Every driver begins with a scaffold, a bare-bones structure that compiles, loads, and unloads without doing much of anything. Think of this as the frame of a house: walls, doors, and furniture come later. Right now, we're building the foundation and skeleton that holds everything together.

In this section, you'll create a minimal FreeBSD 14.3 driver project from scratch. By the end, you'll have:

- A clean directory structure
- A simple Makefile
- A `.c` file with the absolute minimum lifecycle code
- A working build that produces a `myfirst.ko` module

This scaffold is **intentionally boring**. It won't create `/dev` nodes yet, it won't implement `sysctls`, and it won't do any real work. But it will teach you the build cycle, the basic structure, and the discipline of clean entry and exit. Master this, and everything else is just adding layers.

7.2.1 Directory Layout

Let's create a workspace for your driver. Convention in the FreeBSD source tree is to keep drivers under `/usr/src/sys/dev/<drivename>`, but for your first driver, we'll work in your home directory. This keeps your experiments isolated and makes rebuilding simple.

Create the structure:

```
% mkdir -p ~/drivers/myfirst
% cd ~/drivers/myfirst
```

Your working directory will contain:

```
~/drivers/myfirst/  
├─ myfirst.c      # Driver source code  
└─ Makefile       # Build instructions
```

That's it. FreeBSD's kernel module build system (`bsd.kmod.mk`) handles all the complexity (compiler flags, include paths, linking, etc.) for you.

Why this organization?

- **Single directory:** Keeps everything together, easy to clean (`rm -rf ~/drivers/myfirst`).
- **Named after the driver:** When you have multiple projects, you know what `~/drivers/myfirst` contains.
- **Matches tree patterns:** Real FreeBSD drivers in `/usr/src/sys/dev/` follow the same “one directory per driver” approach.

7.2.2 The Minimal Makefile

FreeBSD's build system is remarkably simple for kernel modules. Create `Makefile` with these three lines:

```
# Makefile for myfirst driver  
  
KMOD=    myfirst  
SRCS=    myfirst.c  
  
.include <bsd.kmod.mk>
```

Line by line:

- `KMOD= myfirst` - Declares the module name. This will produce `myfirst.ko`.
- `SRCS= myfirst.c` - Lists source files. We only have one for now.
- `.include <bsd.kmod.mk>` - Pulls in FreeBSD's kernel module build rules. This single line replaces hundreds of lines of manual makefile logic.

Important: The indentation before `.include` is a **tab character**, not spaces. If you use spaces, `make` will fail with a cryptic error. (Most editors can be configured to insert tabs when you press the Tab key.)

What `bsd.kmod.mk` provides:

- Correct compiler flags for kernel code (`-D_KERNEL`, `-ffreestanding`, etc.)
- Include paths (`-I/usr/src/sys`, `-I/usr/src/sys/dev`, etc.)
- Linking rules for creating `.ko` files
- Standard targets: `make`, `make clean`, `make install`, etc.

You don't need to understand the internal details. Just know that `.include <bsd.kmod.mk>` gives you a working build system for free.

Test the Makefile:

Before writing any code, test the build setup:

```
% make clean
% ls
Makefile
```

Right now, `make clean` does almost nothing (no files to delete yet), but it confirms the Makefile syntax is valid.

7.2.3 The Bare-Bones `myfirst.c`

Now create `myfirst.c`, the actual driver source. This first version is **minimal by design**: it compiles, loads, and unloads, but doesn't create devices, doesn't handle I/O, and doesn't allocate resources.

Here's the skeleton:

```
/* -
 * SPDX-License-Identifier: BSD-2-Clause
 *
 * Copyright (c) 2025 Your Name
 * All rights reserved.
 */

#include <sys/param.h>
#include <sys/module.h>
#include <sys/kernel.h>
#include <sys/systm.h>

/*
 * Module load/unload event handler.
 *
 * This function is called when the module is loaded (MOD_LOAD)
 * and unloaded (MOD_UNLOAD). For now, we just print messages.
 */
static int
myfirst_loader(module_t mod, int what, void *arg)
{
    int error = 0;

    switch (what) {
    case MOD_LOAD:
        printf("myfirst: driver loaded\n");
        break;
```

```

        case MOD_UNLOAD:
            printf("myfirst: driver unloaded\n");
            break;
        default:
            error = EOPNOTSUPP;
            break;
    }

    return (error);
}

/*
 * Module declaration.
 *
 * This ties the module name "myfirst" to the loader function above.
 */
static moduledata_t myfirst_mod = {
    "myfirst",           /* module name */
    myfirst_loader,      /* event handler */
    NULL                 /* extra arg (unused here) */
};

/*
 * DECLARE_MODULE registers this module with the kernel.
 *
 * Parameters:
 * - module name: myfirst
 * - moduledata: myfirst_mod
 * - subsystem: SI_SUB_DRIVERS (driver subsystem)
 * - order: SI_ORDER_MIDDLE (standard priority)
 */
DECLARE_MODULE(myfirst, myfirst_mod, SI_SUB_DRIVERS, SI_ORDER_MIDDLE);
MODULE_VERSION(myfirst, 1);

```

What this code does:

- **Includes:** Pull in kernel headers for module infrastructure and logging.
- **myfirst_loader():** Handles module lifecycle events. Right now, just MOD_LOAD and MOD_UNLOAD.
- **moduledata_t:** Connects the module name to the loader function.
- **DECLARE_MODULE():** Registers the module with the kernel. This is what makes kldload recognize your module.
- **MODULE_VERSION():** Stamps the module with version 1 (increment this if you change the exported ABI in the future).

What this code doesn't do (yet):

- Doesn't create any devices
- Doesn't call `make_dev()`
- Doesn't register with Newbus
- Doesn't allocate memory or resources

This is **load/unload only**, the absolute minimum to prove the build system works.

7.2.4 Build and Test the Scaffold

Let's compile and load this minimal module:

1. Build:

```
% make
machine -> /usr/src/sys/amd64/include
x86 -> /usr/src/sys/x86/include
i386 -> /usr/src/sys/i386/include
touch opt_global.h
Warning: Object directory not changed from original /usr/home/youruser/project/myfirst
cc -O2 -pipe -fno-strict-aliasing -Werror -D_KERNEL -DKLD_MODULE -nostdinc -include
↳ /usr/home/youruser/project/myfirst/opt_global.h -I. -I/usr/src/sys
↳ -I/usr/src/sys/contrib/ck/include -fno-common -fno-omit-frame-pointer
↳ -mno-omit-leaf-frame-pointer
↳ -fdebug-prefix-map=./machine=/usr/src/sys/amd64/include
↳ -fdebug-prefix-map=./x86=/usr/src/sys/x86/include
↳ -fdebug-prefix-map=./i386=/usr/src/sys/i386/include -MD -MF.depend.myfirst.o
↳ -MTmyfirst.o -mmodel=kernel -mno-red-zone -mno-mmx -mno-sse -msoft-float
↳ -fno-asynchronous-unwind-tables -ffreestanding -fwrapv -fstack-protector -Wall
↳ -Wstrict-prototypes -Wmissing-prototypes -Wpointer-arith -Wcast-qual -Wundef
↳ -Wno-pointer-sign -D__printf__=__freebsd_kprintf__ -Wmissing-include-dirs
↳ -fdiagnostics-show-option -Wno-unknown-pragmas -Wswitch
↳ -Wno-error=tautological-compare -Wno-error=empty-body
↳ -Wno-error=parentheses-equality -Wno-error=unused-function -Wno-error=pointer-sign
↳ -Wno-error=shift-negative-value -Wno-address-of-packed-member
↳ -Wno-format-zero-length -mno-aes -mno-avx -std=gnu17 -c myfirst.c -o myfirst.o
ld -m elf_x86_64_fbsd -warn-common --build-id=sha1 -T
↳ /usr/src/sys/conf/ldscript.kmod.amd64 -r -o myfirst.ko myfirst.o
:> export_syms
awk -f /usr/src/sys/conf/kmod_syms.awk myfirst.ko export_syms | xargs -J % objcopy %
↳ myfirst.ko
objcopy --strip-debug myfirst.ko
```

You'll see compiler output. As long as it ends with `myfirst.ko` being created and no errors, you're good.

2. Verify the build output:

```
% ls -l myfirst.ko
-rw-r--r--  1 youruser youruser 11592 Nov  7 00:15 myfirst.ko
```

(File size will vary based on compiler and architecture.)

3. Load the module:

```
% sudo kldload ./myfirst.ko
% dmesg | tail -n 2
myfirst: driver loaded
```

4. Check it's loaded:

```
% kldstat | grep myfirst
 6      1 0xffffffff82a38000      20b8 myfirst.ko
```

Your module is now part of the running kernel.

5. Unload the module:

```
% sudo kldunload myfirst
% dmesg | tail -n 2
myfirst: driver unloaded
```

6. Confirm it's gone:

```
% kldstat | grep myfirst
(no output)
```

Perfect. Your scaffold works.

7.2.5 What Just Happened?

Let's trace the flow step by step:

1. **Build:** make invoked the FreeBSD kernel module build system, which compiled `myfirst.c` with kernel flags and linked it into `myfirst.ko`.
2. **Load:** `kldload` read `myfirst.ko`, linked it into the running kernel, and called your `myfirst_loader()` function with `MOD_LOAD`.
3. **Log:** Your `printf()` wrote “myfirst: driver loaded” to the kernel message buffer.
4. **Unload:** `kldunload` called your loader with `MOD_UNLOAD`, you printed a message, then the kernel removed your code from memory.

Key insight: This isn't a Newbus driver yet. There's no `probe()`, no `attach()`, no devices. It's just a module that loads and unloads. Think of it as **stage 0**: proving the build system works before adding complexity.

7.2.6 Troubleshooting Common Scaffold Issues

1. Problem: make fails with “missing separator”

Cause: Your Makefile uses spaces instead of tabs before `.include`.

Fix: Replace the leading spaces with a tab character.

2. Problem: `kldload` says “Exec format error”

Cause: Mismatch between your kernel version and `/usr/src` version.

Fix: Verify `freebsd-version -k` matches your source tree. Rebuild your kernel or re-clone `/usr/src` for the correct version.

3. Problem: Module loads but no message in `dmesg`

Cause: Kernel message buffer might have scrolled, or `printf()` is being rate-limited.

Fix: Use `dmesg -a` to see all messages, including older ones. Also check `sysctl kern.msgbuf_show_timestamp`.

4. Problem: `kldunload` says “module busy”

Cause: Something is still using your module (very unlikely with this minimal scaffold).

Fix: Not applicable here, but later you’ll see this if device nodes are still open or resources aren’t released.

7.2.7 Clean Build Practices

As you iterate on your driver, adopt these habits early:

1. Always clean before rebuilding:

```
% make clean
% make
```

This ensures stale object files don’t contaminate your build.

2. Unload before rebuilding:

```
% sudo kldunload myfirst 2>/dev/null || true
% make clean && make
```

If the module isn’t loaded, `kldunload` fails harmlessly. The `|| true` prevents the shell from stopping.

3. Use a rebuild script:

Create `~/drivers/myfirst/rebuild.sh`:

```
#!/bin/sh
#
# FreeBSD kernel module rebuild script
# Usage: ./rebuild_module.sh <module_name>
```

```
#

set -e

# Configuration
MODULE_NAME="${1}"

# Colors for output (if terminal supports it)
if [ -t 1 ]; then
    RED='\033[0;31m'
    GREEN='\033[0;32m'
    YELLOW='\033[1;33m'
    BLUE='\033[0;34m'
    NC='\033[0m' # No Color
else
    RED=''
    GREEN=''
    YELLOW=''
    BLUE=''
    NC=''
fi

# Helper functions
print_step() {
    printf "${BLUE}==>${NC} ${1}\n"
}

print_success() {
    printf "${GREEN}▣${NC} ${1}\n"
}

print_error() {
    printf "${RED}▣${NC} ${1}\n" >&2
}

print_warning() {
    printf "${YELLOW}!${NC} ${1}\n"
}

check_root() {
    if [ "$(id -u)" -ne 0 ]; then
        print_error "This script must be run as root or with sudo"
        exit 1
    fi
}
```

```
}

is_module_loaded() {
    kldstat -q -n "${1}" 2>/dev/null
}

# Validate arguments
if [ -z "${MODULE_NAME}" ]; then
    print_error "Usage: $0 <module_name>"
    exit 1
fi

# Validate source file exists
if [ ! -f "${MODULE_NAME}.c" ]; then
    print_error "Source file '${MODULE_NAME}.c' not found in current directory"
    exit 1
fi

# Check if we have root privileges
check_root

# Check if Makefile exists
if [ ! -f "Makefile" ]; then
    print_error "Makefile not found in current directory"
    exit 1
fi

# Step 1: Unload module if loaded
print_step "Checking if module '${MODULE_NAME}' is loaded..."
if is_module_loaded "${MODULE_NAME}"; then
    print_warning "Module is loaded, unloading..."

    # Capture dmesg state before unload
    DMESG_BEFORE_UNLOAD=$(dmesg | wc -l)

    if kldunload "${MODULE_NAME}" 2>/dev/null; then
        print_success "Module unloaded successfully"
    else
        print_error "Failed to unload module"
        exit 1
    fi

    # Verify unload
    sleep 1
fi
```

```
if is_module_loaded "${MODULE_NAME}"; then
    print_error "Module still loaded after unload attempt"
    exit 1
fi
print_success "Verified: module removed from memory"

# Check dmesg for unload messages
DMESG_AFTER_UNLOAD=$(dmesg | wc -l)
DMESG_UNLOAD_NEW=$((DMESG_AFTER_UNLOAD - DMESG_BEFORE_UNLOAD))

if [ ${DMESG_UNLOAD_NEW} -gt 0 ]; then
    echo
    print_step "Kernel messages from unload:"
    dmesg | tail -n ${DMESG_UNLOAD_NEW}
    echo
fi
else
    print_success "Module not loaded, proceeding..."
fi

# Step 2: Clean build artifacts
print_step "Cleaning build artifacts..."
if make clean; then
    print_success "Clean completed"
else
    print_error "Clean failed"
    exit 1
fi

# Step 3: Build module
print_step "Building module..."
if make; then
    print_success "Build completed"
else
    print_error "Build failed"
    exit 1
fi

# Verify module file exists
if [ ! -f "${MODULE_NAME}.ko" ]; then
    print_error "Module file '${MODULE_NAME}.ko' not found after build"
    exit 1
fi
```



```
# Step 4: Load module
print_step "Loading module..."
DMESG_BEFORE=$(dmesg | wc -l)

if kldload "./${MODULE_NAME}.ko"; then
    print_success "Module load command executed"
else
    print_error "Failed to load module"
    exit 1
fi

# Step 5: Verify module is loaded
sleep 1
print_step "Verifying module load..."

if is_module_loaded "${MODULE_NAME}"; then
    print_success "Module is loaded in kernel"

    # Show module info
    echo
    kldstat | head -n 1
    kldstat | grep "${MODULE_NAME}"
else
    print_error "Module not found in kldstat output"
    exit 1
fi

# Step 6: Check kernel messages
echo
print_step "Recent kernel messages from load:"
DMESG_AFTER=$(dmesg | wc -l)
DMESG_NEW=$((DMESG_AFTER - DMESG_BEFORE))

if [ ${DMESG_NEW} -gt 0 ]; then
    dmesg | tail -n ${DMESG_NEW}
else
    print_warning "No new kernel messages"
    dmesg | tail -n 5
fi

echo
print_success "Module '${MODULE_NAME}' rebuilt and loaded successfully!"
```

Make it executable:

```
% chmod +x rebuild.sh
```

Now you can iterate quickly:

```
% ./rebuild.sh myfirst
```

This script unloads, cleans, builds, loads, and shows recent kernel messages, all in one go. It's a huge time-saver during development.

Note: You might wonder if this script needs to be this complex. For a one-time use, it doesn't. However, kernel module development involves repeated cycles of unload-rebuild-load, often dozens of times per day. Building it with proper error handling and validation creates a tool you'll confidently reuse throughout your development process, saving you countless hours later. More importantly, this is a perfect opportunity to practice defensive programming: validating inputs, checking for errors at each step, and providing clear feedback when something goes wrong. These habits will serve you well in all your future development work.

7.2.8 Version Control Checkpoint

Before moving forward, commit your scaffold to Git (if you're using version control, and you should be):

```
% cd ~/drivers/myfirst
% git init
% git add Makefile myfirst.c
% git commit -m "Initial scaffold: loads and unloads cleanly"
```

This gives you a known-good state to return to if you break something later. If you're using a remote repository (GitHub, GitLab, etc.), you can push these changes with `git push`, but it's not required for local version control benefits.

7.2.9 What's Next?

You now have a working scaffold: a module that builds, loads, and unloads. It's not a Newbus driver yet, and it doesn't create any user-visible surfaces, but it's a solid foundation.

In the next section, we'll add **Newbus integration**, transforming this simple module into a proper pseudo-device driver that registers with the device tree and implements the `probe()` and `attach()` lifecycle methods.

7.3 Newbus: Just Enough to Attach

You've built a scaffold that loads and unloads. Now we'll transform it into a **Newbus driver**, one that registers with FreeBSD's device framework and follows the standard `identify / probe / attach / detach` lifecycle.

This is where your driver stops being a passive module and starts behaving like a real device driver. By the end of this section, you'll have a driver that:

- Registers as a pseudo-device on the nexus bus
- Provides an `identify()` method that creates the `myfirst` device on the bus
- Implements `probe()` to claim the device
- Implements `attach()` to initialize (even though initialization is minimal for now)
- Implements `detach()` to clean up
- Logs lifecycle events properly

We're keeping this **just enough** to show the pattern. No resource allocation yet, no device nodes, no sysctls. Those come in later sections. Right now, focus on understanding **how Newbus calls your code** and **what each method should do**.

7.3.1 Why Newbus?

FreeBSD uses Newbus to manage device discovery, driver matching, and lifecycle. Even for pseudo-devices (software-only devices with no backing hardware), following the Newbus pattern ensures:

- Consistent behavior across all drivers
- Proper integration with the device tree
- Reliable lifecycle management (`attach` / `detach` / `suspend` / `resume`)
- Compatibility with tools like `devinfo` and `kldunload`

Mental model: Newbus is the kernel's HR department. It opens new positions (`identify`), interviews drivers for each position (`probe`), hires the best fit (`attach`), and manages resignations (`detach`). For real hardware, the bus posts the position automatically. For a pseudo-device, your driver also writes the job description, which is what `identify` is for.

7.3.2 The Minimal Newbus Pattern

Every Newbus driver follows this structure:

1. **Define device methods** (`identify`, `probe`, `attach`, `detach`) as functions
2. **Create a method table** mapping Newbus method names to your functions
3. **Declare a driver structure** that includes the method table and softc size
4. **Register the driver** with `DRIVER_MODULE()`

For drivers that attach to a real bus such as `pci` or `usb`, the bus enumerates hardware on its own and asks every registered driver “is this device yours?” through `probe`. A pseudo-device has no hardware to enumerate, so we have to tell the bus that the device exists. That is the job of `identify`. We will introduce it in step 4 below, after `probe` and `attach` are in place, so the role of each method is clear before the file gets crowded.

Let's walk through each piece step by step.

7.3.3 Step 1: Include Newbus Headers

At the top of `myfirst.c`, add these includes (replacing or augmenting the minimal includes from the scaffold):

```
#include <sys/param.h>
#include <sys/module.h>
#include <sys/kernel.h>
#include <sys/systm.h>
#include <sys/bus.h>          /* For device_t, Newbus APIs */
#include <sys/conf.h>         /* For cdevsw (used later) */
```

What these provide:

- `<sys/bus.h>` - Core Newbus types (`device_t`, `device_method_t`) and functions (`device_printf`, `device_get_softc`, etc.)
- `<sys/conf.h>` - Character device switch structures (we'll use this when creating `/dev` nodes)

7.3.4 Step 2: Define Your Softc

The **softc** (software context) is your driver's per-device private data structure. Even though we're not storing anything interesting yet, **every Newbus driver has one**.

Add this near the top of `myfirst.c`, after the includes:

```
/*
 * Driver softc (software context).
 *
 * One instance of this structure exists per device.
 * Newbus allocates and zeroes it for us.
 */
struct myfirst_softc {
    device_t    dev;          /* Back-pointer to device_t */
    uint64_t    attach_time;  /* When we attached (ticks) */
    int         is_ready;     /* Simple flag */
};
```

Why these fields?

- `dev` - Convenient back-pointer. Lets you call `device_printf(sc->dev, ...)` without passing `dev` everywhere.
- `attach_time` - Example state. We'll record when `attach()` ran.
- `is_ready` - Another example flag. Shows how you'd track driver state.

Key insight: You never `malloc()` or `free()` the `softc` yourself. Newbus does it automatically based on the size you declare in the driver structure.

7.3.5 Step 3: Implement Probe

The `probe()` method answers one question: **“Does this driver match this device?”**

For a pseudo-device, the answer is always yes (we're not checking PCI IDs or hardware signatures). But we still implement `probe()` to follow the pattern and set a device description.

Add this function:

```
/*
 * Probe method.
 *
 * Called by Newbus to see if this driver wants to handle this device.
 * For a pseudo-device created by our own identify method, we always accept.
 *
 * The return value is a priority. Higher values win when several drivers
 * are willing to take the same device. ENXIO means "not mine, reject".
 */
static int
myfirst_probe(device_t dev)
{
    device_set_desc(dev, "My First FreeBSD Driver");
    return (BUS_PROBE_DEFAULT);
}
```

Line by line:

- `device_set_desc()` sets a human-readable description. It appears in `devinfo -v` and in attach messages. The string must remain valid for the device's lifetime, so always pass a string literal here. If you ever need a dynamically built description, use `device_set_desc_copy()` instead.
- `return (BUS_PROBE_DEFAULT)` tells Newbus "I will handle this device, with the standard base-OS priority."

Probe discipline:

- **Don't** allocate resources in `probe()`. If another driver wins, your resources would leak.
- **Don't** touch hardware in `probe()` (not relevant here, but essential for real hardware drivers).
- **Do** return quickly. Probe is called frequently during boot and hot-plug events.

A note on probe priority values. When several drivers are willing to take the same device, the kernel picks the one whose `probe()` returned the **highest** value. The constants in `<sys/bus.h>` reflect that ordering, with the more specific bids being numerically larger:

Constant	Value (FreeBSD 14.3)	When to use it
<code>BUS_PROBE_SPECIFIC</code>	0	Only this driver can possibly handle this device
<code>BUS_PROBE_VENDOR</code>	-10	Vendor-supplied driver, beats the generic class driver
<code>BUS_PROBE_DEFAULT</code>	-20	Standard base-OS driver for this class
<code>BUS_PROBE_LOW_PRIORITY</code>	-40	Older or less desirable driver

Constant	Value (FreeBSD 14.3)	When to use it
BUS_PROBE_GENERIC	-100	Generic fallback driver
BUS_PROBE_NOWILDCARD	very large negative	Only match devices created explicitly (e.g. by identify)

BUS_PROBE_DEFAULT is the right choice for a typical driver, including ours: we identify our own device by name in `identify()` so no real competitor exists, and the value is high enough that nothing will beat us.

7.3.6 Step 4: Implement Attach

The `attach()` method is where you **initialize your driver**. Resources get allocated, hardware gets configured, device nodes get created. For now, we'll just log a message and populate the `softc`.

Add this function:

```
/*
 * Attach method.
 *
 * Called after probe succeeds. Initialize the driver here.
 */
static int
myfirst_attach(device_t dev)
{
    struct myfirst_softc *sc;

    sc = device_get_softc(dev);
    sc->dev = dev;
    sc->attach_time = ticks; /* Record when we attached */
    sc->is_ready = 1;

    device_printf(dev, "Attached successfully at tick %lu\n",
        (unsigned long)sc->attach_time);

    return (0);
}
```

What this does:

- `device_get_softc(dev)` - Retrieves the `softc` that Newbus allocated for us (zeroed initially).
- `sc->dev = dev` - Saves the `device_t` back-pointer for convenience.
- `sc->attach_time = ticks` - Records the current kernel tick count (a simple timestamp).
- `sc->is_ready = 1` - Sets a flag (not used yet, but shows how you'd track state).
- `device_printf()` - Logs the attach event with our device name prefix.
- `return (0)` - Success. Non-zero would indicate failure and abort attachment.

Attach discipline:

- **Do** allocate resources here (memory, locks, hardware mappings).
- **Do** create user surfaces (/dev nodes, network interfaces, etc.).
- **Do** handle failures gracefully. If something goes wrong, undo what you started and return an error code.
- **Don't** touch user-space yet. Attach runs during module load or device discovery, before any user program can interact with you.

Error handling preview:

Right now, attach can't fail (we're not doing anything that could go wrong). Later sections will add resource allocation, and you'll see how to unwind on failure.

7.3.7 Step 5: Implement Detach

The detach() method is the inverse of attach(): tear down what you built, release what you claimed, and leave no traces.

Add this function:

```
/*
 * Detach method.
 *
 * Called when the driver is being unloaded or the device is removed.
 * Clean up everything you set up in attach().
 */
static int
myfirst_detach(device_t dev)
{
    struct myfirst_softc *sc;

    sc = device_get_softc(dev);

    device_printf(dev, "Detaching (was attached for %lu ticks)\n",
        (unsigned long)(ticks - sc->attach_time));

    sc->is_ready = 0;

    return (0);
}
```

What this does:

- Retrieves the_softc (we know it exists because attach succeeded).
- Logs how long the driver was attached (current ticks minus attach_time).
- Clears the is_ready flag (not strictly necessary since the_softc will be freed soon, but good practice).
- Returns 0 (success).

Detach discipline:

- **Do** release all resources (locks destroyed, memory freed, device nodes destroyed).
- **Do** ensure no active I/O or callbacks can reach your code after detach returns.
- **Do** return EBUSY if the device is in use and can't be detached yet (e.g., open device nodes).
- **Don't** assume softc is still valid after detach returns. Newbus will free it.

Why detach matters:

Poor detach implementations are the #1 source of kernel panics on unload. If you forget to destroy a lock, free a resource, or remove a callback, you'll crash when that resource is accessed after your code is gone.

7.3.8 Step 6: Implement Identify

We have probe, attach, and detach. They tell the kernel **what to do** when a myfirst device shows up. But there is no myfirst device on the nexus bus yet, and nexus has no way to invent one. We have to create the device ourselves at the moment our driver is registered. That is what an identify method does.

Add this function:

```
/*
 * Identify method.
 *
 * Called by Newbus once, right after the driver is registered with the
 * parent bus. Its job is to create child devices that this driver will
 * then probe and attach.
 *
 * Real hardware drivers usually do not need an identify method, because
 * the bus (PCI, USB, ACPI, ...) enumerates devices on its own. A pseudo
 * device has nothing for the bus to find, so we add our single device
 * here, by name.
 */
static void
myfirst_identify(driver_t *driver, device_t parent)
{
    if (device_find_child(parent, driver->name, -1) != NULL)
        return;
    if (BUS_ADD_CHILD(parent, 0, driver->name, -1) == NULL)
        device_printf(parent, "myfirst: BUS_ADD_CHILD failed\n");
}
```

Line by line:

- `device_find_child(parent, driver->name, -1)` checks whether a myfirst device already exists below parent. If we did not check, reloading the module (or any second pass over the bus) would create duplicate devices.

- `BUS_ADD_CHILD(parent, 0, driver->name, -1)` asks the parent bus to create a new child device named `myfirst`, at order 0, with an automatically chosen unit number. After this call, Newbus will run our probe against the new child, and if probe accepts, attach.
- We log on failure but do not panic. `BUS_ADD_CHILD` can fail under memory pressure, and a missing pseudo-device should not take the system down.

Where this fits. `identify` runs once per driver per bus, when the driver is first attached to that bus. After `identify`, the bus's normal probe and attach machinery takes over. This is the same pattern used by drivers such as `cryptosoft`, `aesni`, and `snd_dummy` in the FreeBSD source tree, which you can browse later as references.

7.3.9 Step 7: Create the Method Table

Now connect your functions to Newbus method names. Add this after your function definitions:

```
/*
 * Device method table.
 *
 * Maps Newbus method names to our functions.
 */
static device_method_t myfirst_methods[] = {
    /* Device interface */
    DEVMETHOD(device_identify,    myfirst_identify),
    DEVMETHOD(device_probe,      myfirst_probe),
    DEVMETHOD(device_attach,     myfirst_attach),
    DEVMETHOD(device_detach,     myfirst_detach),

    DEVMETHOD_END

};
```

What this table means:

- `DEVMETHOD(device_identify, myfirst_identify)` says “when the bus invites every driver to create its devices, run `myfirst_identify()`.”
- `DEVMETHOD(device_probe, myfirst_probe)` says “when the kernel calls `DEVICE_PROBE(dev)`, run `myfirst_probe()`.”
- Same for `attach` and `detach`.
- `DEVMETHOD_END` terminates the table and is required.

Behind the scenes: The `DEVMETHOD()` macro and the `kobj` system (kernel objects) generate the glue code that dispatches to your functions. You don't need to understand the internals; just know that this table is how Newbus finds your code.

7.3.10 Step 8: Declare the Driver

Tie everything together in a `driver_t` structure:

```

/*
 * Driver declaration.
 *
 * Specifies our method table and softc size.
 */
static driver_t myfirst_driver = {
    "myfirst",          /* Driver name */
    myfirst_methods,    /* Method table */
    sizeof(struct myfirst_softc) /* Softc size */
};

```

Parameters:

- "myfirst" - Driver name (shows up in logs and as the device name prefix).
- myfirst_methods - Pointer to the method table you just created.
- sizeof(struct myfirst_softc) - Tells Newbus how much memory to allocate per device.

Why the softc size? Newbus allocates one softc per device instance. By declaring the size here, you never manually allocate or free it, Newbus manages the lifetime.

7.3.11 Step 9: Register with DRIVER_MODULE

Replace the old DECLARE_MODULE() macro from the scaffold with this:

```

/*
 * Driver registration.
 *
 * Attach this driver under the nexus bus. Our identify method will
 * create the actual myfirst child device when the module loads.
 */

DRIVER_MODULE(myfirst, nexus, myfirst_driver, 0, 0);
MODULE_VERSION(myfirst, 1);

```

What this does:

- DRIVER_MODULE(myfirst, nexus, myfirst_driver, 0, 0) registers myfirst as a driver willing to attach below the nexus bus. The two trailing zeros are an optional module event handler and its argument; our minimal driver does not need them.
- MODULE_VERSION(myfirst, 1) stamps the module with version 1, so other modules can declare a dependency on it.

Why nexus?

nexus is FreeBSD's root bus, the top of every architecture's device tree. Chapter 6 advised you, correctly, that nexus is rarely the right parent for a real hardware driver: a PCI driver belongs under pci, a USB driver under

usb, and so on. Pseudo-devices are different. They have no physical bus, so the convention in the FreeBSD source tree is to attach them to nexus and create the child device themselves through an identify method. This is exactly what cryptosoft, aesni, and snd_dummy do, and exactly what we are doing here.

7.3.12 Step 10: Remove the Old Module Loader

You no longer need the `myfirst_loader()` function or `moduledata_t` struct from the scaffold. Newbus now drives the module lifecycle through `identify`, `probe`, `attach`, and `detach`. Remove those old pieces entirely.

Your `myfirst.c` should now have:

- Includes
- Softc structure
- `myfirst_identify()`
- `myfirst_probe()`
- `myfirst_attach()`
- `myfirst_detach()`
- Method table
- Driver structure
- `DRIVER_MODULE()` and `MODULE_VERSION()`

No more `MOD_LOAD` event handler.

7.3.13 Step 11: Adjust the Makefile

Add this line to your Makefile:

```
# Required for Newbus drivers: generates device_if.h and bus_if.h
SRCS+= device_if.h bus_if.h
```

Why this is needed:

FreeBSD's Newbus framework uses a method dispatch system built on top of `kobj`. The `DEVMETHOD()` entries in your method table refer to method identifiers declared in the generated headers `device_if.h` and `bus_if.h`. `bsd.kmod.mk` knows how to build these from `/usr/src/sys/kern/device_if.m` and `/usr/src/sys/kern/bus_if.m`, but it only does so if you list them in `SRCS`. If you forget this line you will get a confusing error about unknown method identifiers when you compile.

7.3.14 Build and Test the Newbus Driver

Let's compile and test:

1. Clean and build:

```
% make clean
% make
```

You should see no errors.

2. Load the module:

```
% sudo kldload ./myfirst.ko
% dmesg | tail -n 3
myfirst0: <My First FreeBSD Driver> on nexus0
myfirst0: Attached successfully at tick 123456
```

Notice:

- The device name is `myfirst0` (driver name + unit number).
- It attached “on nexus0” (the parent bus).
- Your custom attach message appeared.

3. Check the device tree:

```
% devinfo -v | grep myfirst
myfirst0
```

Your driver is now part of the device tree.

4. Unload:

```
% sudo kldunload myfirst
% dmesg | tail -n 2
myfirst0: Detaching (was attached for 5432 ticks)
```

Your detach message shows how long the driver was attached.

5. Verify it's gone:

```
% devinfo -v | grep myfirst
(no output)
```

7.3.15 What Changed?

Compared to the scaffold, your driver now:

- **Registers with Newbus** instead of using a simple module loader.
- **Adds a child device** (`myfirst0`) to the device tree through `identify`.
- **Follows the identify / probe / attach / detach lifecycle** instead of just load/unload.
- **Allocates and manages a `softc`** automatically.

This is the **foundation pattern** for every FreeBSD driver. Master this, and the rest is just adding layers.

7.3.16 Common Newbus Mistakes (and How to Avoid Them)

Mistake 0: Forgetting the identify method on a pseudo-device

Symptom: `kldload` succeeds, but no `myfirst0` device appears, no probe message in `dmesg`, and `devinfo` shows nothing under `nexus0`. The driver compiled and loaded, but it never attached.

Cause: The driver was registered with `DRIVER_MODULE(..., nexus, ...)` but no `device_identify` method was provided. Nexus has nothing to enumerate, so probe and attach are never called.

Fix: Add the `identify` method shown in Step 6 and put `DEVMETHOD(device_identify, myfirst_identify)` in the method table. This is the most common reason a beginner's pseudo-device driver “loads but does nothing.”

Mistake 1: Allocating resources in probe

Wrong:

```
static int
myfirst_probe(device_t dev)
{
    struct myfirst_softc *sc = device_get_softc(dev);
    sc->something = malloc(...); /* BAD! */
    return (BUS_PROBE_DEFAULT);
}
```

Why it's wrong: If probe fails or another driver wins, your allocation leaks.

Right: Allocate in `attach()`, where you know the driver has been selected.

Mistake 2: Forgetting to return 0 from attach

Wrong:

```
static int
myfirst_attach(device_t dev)
{
    /* ... setup ... */
    /* (missing return statement) */
}
```

Why it's wrong: Compiler might warn, but the return value is undefined. You might accidentally return garbage, causing `attach` to fail mysteriously.

Right: Always end `attach` with `return (0)` on success or `return (error_code)` on failure.

Mistake 3: Not cleaning up in detach**Wrong:**

```
static int
myfirst_detach(device_t dev)
{
    device_printf(dev, "Detaching\n");
    return (0);
    /* (forgot to free resources, destroy locks, etc.) */
}
```

Why it's wrong: Resources leak. Locks remain active. Next load might panic.

Right: Detach must undo everything attach did. We'll cover the cleanup pattern in detail in the error handling section.

7.3.17 Newbus Lifecycle Timing Diagram

```
[ Boot or kldload ]
|
v
identify(parent) --> "What devices does this driver provide?"
|                    (Pseudo-devices: BUS_ADD_CHILD here)
|                    (Real hardware: usually omitted)
v
probe(dev) --> "Is this device mine?"
|              (Check IDs, set description)
| (return a probe priority such as BUS_PROBE_DEFAULT)
v
attach(dev) --> "Initialize and prepare for use"
|              (Allocate resources, create surfaces)
|              (If fails, undo what was done, return error)
v
[ Device ready, normal operation ]
|
| (time passes, I/O happens, sysctls read, etc.)
|
v
detach(dev) --> "Shutdown and cleanup"
|              (Destroy surfaces, release resources)
|              (Return EBUSY if still in use)
```

```
v
[ Module unloaded or device gone ]
```

Key insight: Each step is distinct. Identify creates the device, probe claims it, attach initializes, detach cleans up. Don't blur the boundaries.

7.3.18 Quick Self-Check

Before moving forward, ensure you can answer these:

1. **Where do I allocate memory for driver state?** Answer: In `attach()`, or just use the `softc` (Newbus allocates it for you).
2. **What does `device_get_softc()` return?** Answer: A pointer to your driver's per-device private data (`struct myfirst_softc *` in this case).
3. **When is probe called?** Answer: During device enumeration. For a real bus, that happens when the bus discovers a device. For our pseudo-device, it happens right after our `identify` method calls `BUS_ADD_CHILD()` to put a `myfirst` device on the nexus bus.
4. **What must detach do?** Answer: Undo everything `attach` did, release resources, and ensure no code paths can reach the driver afterward.
5. **Why do we use nexus for this driver, and why does it need an `identify` method?** Answer: Because it's a pseudo-device with no physical bus. `nexus` is the conventional parent for software-only devices, but `nexus` has no devices to enumerate, so we create our own through `identify`.

If those answers make sense, you're ready for the next section: adding real state management with the `softc`.

7.4 `softc` and Lifecycle State

You've seen the `softc` structure declared, allocated, and retrieved, but we haven't talked about **why it exists** or **how to use it properly**. In this section, we'll explore the `softc` pattern in depth: what goes in it, how to initialize and access it safely, and how to avoid common pitfalls.

The `softc` is your driver's **memory**. Every resource, every lock, every statistic, and every flag lives here. Getting this right is the difference between a reliable driver and one that panics under load.

7.4.1 What Is the `softc`?

The **`softc`** (software context) is a per-device structure that stores everything your driver needs to operate. Think of it as the driver's "workspace" or "notebook", one instance per device, holding all the state that makes that particular device work.

Key properties:

- **Per-device:** If your driver handles multiple devices (e.g., `myfirst0`, `myfirst1`), each gets its own `softc`.
- **Kernel-allocated:** You declare the structure type and size; Newbus allocates and zeroes the memory.
- **Lifetime:** Exists from device creation (before `attach()`) until device deletion (after `detach()`).
- **Access pattern:** Retrieve with `device_get_softc(dev)` at the start of every method.

Why not global variables?

Global variables can't handle multiple devices. If you stored state in globals, `myfirst0` and `myfirst1` would clobber each other's data. The `softc` pattern solves this elegantly: each device has its own isolated state.

7.4.2 What Belongs in the `softc`?

A well-designed `softc` contains:

1. Identification and housekeeping

- `device_t dev` - Back-pointer to the device (for logging and callbacks)
- `int unit` - Device unit number (often extracted from `dev`, but handy to cache)
- `char name[16]` - Device name string if you need it frequently

2. Resources

- `struct resource *mem_res` - MMIO regions (for hardware drivers)
- `int mem_rid` - Resource ID for memory
- `struct resource *irq_res` - Interrupt resource
- `void *irq_handler` - Interrupt handler cookie
- `bus_dma_tag_t dma_tag` - DMA tag (for drivers that do DMA)

3. Synchronization primitives

- `struct mtx mtx` - Mutex for protecting shared state
- `struct sx sx` - Shared/exclusive lock if needed
- `struct cv cv` - Condition variable for sleeping/waking

4. Device state flags

- `int is_attached` - Set in `attach`, cleared in `detach`
- `int is_open` - Set when `/dev` node is open
- `uint32_t flags` - Bitfield for misc state (running, suspended, error, etc.)

5. Statistics and counters

- `uint64_t tx_packets` - Packets transmitted (network driver example)
- `uint64_t rx_bytes` - Bytes received
- `uint64_t errors` - Error count
- `time_t last_reset` - When stats were last cleared

6. Driver-specific data

- Hardware registers, queues, buffers, work structures, anything unique to your driver's operation.

What doesn't belong:

- **Large buffers:** Softc lives in kernel memory (wired, non-pageable). Big buffers should be allocated separately with `malloc()` or `contigmalloc()` and pointed to from the softc.
- **Constant data:** Use const global arrays or static tables instead.
- **Temporary variables:** Function locals are fine. Don't clutter the softc with per-operation temporaries.

7.4.3 Our myfirst Softc (Minimal Example)

Let's revisit and expand our softc definition:

```
struct myfirst_softc {
    device_t      dev;           /* Back-pointer */
    int           unit;          /* Device unit number */

    struct mtx     mtx;          /* Protects shared state */

    uint64_t      attach_ticks;  /* When attach() ran */
    uint64_t      open_count;    /* How many times opened */
    uint64_t      bytes_read;    /* Bytes read from device */

    int           is_attached;    /* 1 if attach succeeded */
    int           is_open;        /* 1 if /dev node is open */
};
```

Field by field:

- `dev` - Standard back-pointer. Almost every driver includes this.
- `unit` - Cached unit number (from `device_get_unit(dev)`). Optional but convenient.
- `mtx` - Mutex to protect concurrent access. Even though we're not exercising concurrency yet, including it now teaches good habits.
- `attach_ticks` - When we attached (kernel ticks). Simple example state.
- `open_count` / `bytes_read` - Counters. Real drivers track these for stats and observability.
- `is_attached` / `is_open` - Flags for lifecycle state. Useful in error checking.

Why a mutex now?

Even though our minimal driver doesn't need it yet, including the mutex teaches the **pattern**. Every driver will eventually need locking, and it's easier to design it in from the start than retrofit it later.

We don't use the mutex for real shared-data protection yet. Concurrency, lock ordering, and deadlock pitfalls arrive in Part 3. For now, the lock is here to establish the lifecycle pattern, and to make detach ordering safe. See the "destroying locks while threads still hold them" pitfall from Chapter 6.

7.4.4 Initializing the softc in attach()

Newbus zeroes the softc before calling `attach()`, but you still need to initialize certain fields explicitly (locks, back-pointers, flags).

Here's the updated `attach()`:

```
static int
myfirst_attach(device_t dev)
{
    struct myfirst_softc *sc;

    sc = device_get_softc(dev);

    /* Initialize back-pointer and unit */
    sc->dev = dev;
    sc->unit = device_get_unit(dev);

    /* Initialize mutex */
    mtx_init(&sc->mtx, device_get_nameunit(dev), "myfirst", MTX_DEF);

    /* Record attach time */
    sc->attach_ticks = ticks;

    /* Set state flags */
    sc->is_attached = 1;
    sc->is_open = 0;

    /* Initialize counters */
    sc->open_count = 0;
    sc->bytes_read = 0;

    device_printf(dev, "Attached at tick %lu\n",
        (unsigned long)sc->attach_ticks);

    return (0);
}
```

What changed:

- **mtx_init()** - Initializes the mutex. Parameters:
 - `&sc->mtx` - Address of the mutex field in the `softc`.
 - `device_get_nameunit(dev)` - Returns a string like “myfirst0” (used in lock debugging).
 - “myfirst” - Lock type name (appears in lock traces).
 - `MTX_DEF` - Standard mutex (as opposed to spin mutex).
- **sc->is_attached = 1** - Flag that we're now ready.
- **Counter initialization** - Explicitly zero them (even though Newbus zeroed the whole `softc`, being explicit documents intent).

Discipline: Initialize all fields that will be tested later. Don't assume "zero means uninitialized" is always correct semantics (for flags, maybe; for pointers, definitely not).

7.4.5 Destroying the softc in detach()

In `detach()`, you must undo everything `attach()` did. For the softc, that means:

- Destroy locks (mutexes, sx, cv, etc.)
- Free any memory or resources pointed to by the softc
- Clear flags (not strictly necessary, but good hygiene)

Updated `detach()`:

```
static int
myfirst_detach(device_t dev)
{
    struct myfirst_softc *sc;
    uint64_t uptime;

    sc = device_get_softc(dev);

    /* Calculate how long we were attached */
    uptime = ticks - sc->attach_ticks;

    /* Refuse detach if device is open */
    if (sc->is_open) {
        device_printf(dev, "Cannot detach while device is open\n");
        return (EBUSY);
    }

    /* Log stats before shutting down */
    device_printf(dev, "Detaching: uptime %lu ticks, opened %lu times, read %lu
↪ bytes\n",
        (unsigned long)uptime,
        (unsigned long)sc->open_count,
        (unsigned long)sc->bytes_read);

    /* Destroy the mutex */
    mtx_destroy(&sc->mtx);

    /* Clear attached flag */
    sc->is_attached = 0;

    return (0);
}
```

What's new:

- **if (sc->is_open) return (EBUSY)** - Refuse to detach if the /dev node is still open. This prevents crashes from accessing freed resources.
- **Stats logging** - Shows how long the driver was up and what it did.
- **mtx_destroy(&sc->mtx)** - **Critical.** Every `mtx_init()` must have a matching `mtx_destroy()`, or you leak kernel lock resources.
- **Clear flags** - Not strictly required (Newbus will free the softc soon), but good defensive programming.

Common mistake: Forgetting `mtx_destroy()`. This causes lock tracking panics on the next load if you're using WITNESS or INVARIANTS kernels.

7.4.6 Accessing the softc from Other Methods

Every driver method that needs state starts the same way:

```
static int
myfirst_some_method(device_t dev)
{
    struct myfirst_softc *sc;

    sc = device_get_softc(dev);

    /* Now use sc-> to access state */
    ...
}
```

This is the **idiomatic pattern** you'll see in every FreeBSD driver. One line to enter your driver's world.

Why not pass softc directly?

Newbus methods are defined to receive `device_t`. The softc is an implementation detail of your driver. By consistently retrieving it via `device_get_softc()`, your driver stays flexible (you could change the softc structure without changing method signatures).

7.4.7 Using Locks to Protect the softc

Even though we haven't added concurrent operations yet, let's preview the **locking pattern** you'll use when you do.

Basic pattern:

```
static void
myfirst_increment_counter(device_t dev)
{
    struct myfirst_softc *sc;
```

```
    sc = device_get_softc(dev);

    mtx_lock(&sc->mtx);
    sc->open_count++;
    mtx_unlock(&sc->mtx);
}
```

Rules:

- **Lock before modifying shared state.**
- **Unlock as soon as you're done** (don't hold locks longer than necessary).
- **Never return with a lock held** (unless you're doing advanced lock handoff patterns).
- **Document lock order** if you hold multiple locks (to avoid deadlock).

When you'll need this: As soon as you add `open()`, `read()`, `write()`, or any method that could run concurrently (user programs calling your driver from multiple threads, or interrupt handlers updating stats).

For now, the mutex exists but isn't exercised. We'll use it in later sections when we add concurrent entry points.

7.4.8 Softc Best Practices

1. Keep it organized

Group related fields together:

```
struct myfirst_softc {
    /* Identification */
    device_t      dev;
    int           unit;

    /* Synchronization */
    struct mtx     mtx;

    /* Resources */
    struct resource *mem_res;
    int           mem_rid;

    /* Statistics */
    uint64_t      tx_packets;
    uint64_t      rx_bytes;

    /* State flags */
    int           is_attached;
}
```

```

    int            is_open;
};

```

2. Comment non-obvious fields

```

    int            pending_requests;  /* Must hold mtx to access */
    time_t         last_activity;     /* Protected by mtx */

```

3. Use fixed-width types for counters

```

    uint64_t       packets;  /* Not "unsigned long" */
    uint32_t       errors;   /* Not "int" */

```

Why? Portability. `int` and `long` sizes vary by architecture. `uint64_t` is always 64 bits.

4. Avoid padding waste

The compiler inserts padding to align fields. Arrange large fields first, then smaller:

```

/* Good: no wasted padding */
struct example {
    uint64_t       big_counter;  /* 8 bytes */
    uint32_t       medium;       /* 4 bytes */
    uint32_t       medium2;      /* 4 bytes */
    uint16_t       small;        /* 2 bytes */
    uint8_t        tiny;         /* 1 byte */
    uint8_t        tiny2;        /* 1 byte */
};

```

5. Zero fields you'll test

```

    sc->is_open = 0;          /* Explicit, even though Newbus zeroed it */
    sc->bytes_read = 0;

```

Why? Clarity. Someone reading the code knows you *intended* zero, not that you relied on implicit zeroing.

7.4.9 Debugging softc Issues

Problem: Kernel panic “NULL pointer dereference” in your driver.

Likely cause: You forgot to retrieve the `softc`, or retrieved it after a point where `dev` might be invalid.

Fix: Always `sc = device_get_softc(dev);` at the start of every method.

Problem: Mutex panic “already locked” or “not locked.”

Likely cause: Forgot `mtx_init()` in `attach()` or mismatched `mtx_lock()` / `mtx_unlock()` calls.

Fix: Check your init/destroy pairs. Use WITNESS-enabled kernels (options `WITNESS` in kernel config) to catch lock violations.

Problem: Stats or flags seem random/corrupted.

Likely cause: Concurrent access without locking, or accessing softc after `detach()` freed it.

Fix: Ensure all shared state is protected by the mutex. Ensure no code paths (callbacks, timers, threads) can reach the driver after `detach()` returns.

7.4.10 Quick Self-Check

Before moving forward, ensure you understand:

1. **What is the softc?** Answer: Per-device private data structure holding all driver state.
2. **Who allocates the softc?** Answer: Newbus, based on the size declared in the `driver_t` structure.
3. **When must you initialize the mutex?** Answer: In `attach()`, before any code that might use it.
4. **When must you destroy the mutex?** Answer: In `detach()`, before the function returns.
5. **Why do we refuse detach if `is_open` is true?** Answer: To prevent freeing resources while user programs still have the device open, which would cause a crash.

If those answers are clear, you’re ready to add **logging discipline** in the next section.

7.5 Logging Etiquette & dmesg Hygiene

A well-behaved driver **talks when it should** and **stays quiet when it shouldn’t**. Logging too much floods dmesg and makes debugging harder; logging too little leaves users and developers blind when something goes wrong. This section teaches you **when, what, and how to log** in a FreeBSD driver.

By the end, you’ll know:

- Which events **must** be logged (attach, errors, critical state changes)
- Which events **should** be logged (optional, debug-level info)
- Which events **must not** be logged (per-packet/per-operation spam)
- How to use `device_printf()` effectively
- How to create rate-limited logging for hot paths
- How to make your logs readable and actionable

7.5.1 Why Logging Matters

When a driver misbehaves, `dmesg` is often the first place developers and users look. Good logs answer questions like:

- Did the driver attach successfully?
- What hardware did it find?
- Did an error occur? Why?
- Is the device operational or in an error state?

Bad logs spam the console, hide critical messages, or omit important details.

Mental model: Logging is like a doctor taking notes during an exam. Write enough to diagnose problems later, but don't record every heartbeat.

7.5.2 The Golden Rules of Driver Logging

Rule 1: Log lifecycle events

Always log:

- Successful attach (one line per device)
- Attach failures (with reason)
- Successful detach (optional but recommended)
- Detach failures (with reason)

Example:

```
device_printf(dev, "Attached successfully\n");
device_printf(dev, "Failed to allocate memory resource: error %d\n", error);
```

Rule 2: Log errors

When something goes wrong, **always log what and why**. Include:

- What operation failed
- Error code (errno value)
- Context (if relevant)

Example:

```
if (error != 0) {
    device_printf(dev, "Could not allocate IRQ resource: error %d\n", error);
    return (error);
}
```

Bad example:


```
if (error != 0) {
    return (error); /* User sees nothing! */
}
```

Rule 3: Never log in hot paths

“Hot path” = code that runs frequently during normal operation (every packet, every interrupt, every read/write call).

Never do this:

```
static int
myfirst_read(struct cdev *dev, struct uio *uio, int flag)
{
    device_printf(dev, "Read called\n"); /* BAD: spams logs */
    ...
}
```

Why? If a program reads from your device in a loop, you’ll generate thousands of log lines per second, rendering the console unusable.

When to log hot-path events: Only for debugging during development, and guarded by a debug flag or sysctl that’s disabled by default.

Rule 4: Use device_printf() for device-specific messages

device_printf() automatically prefixes your message with the device name:

```
device_printf(dev, "Interrupt timeout\n");
```

Output:

```
myfirst0: Interrupt timeout
```

This makes it immediately clear **which device** is talking, especially when multiple instances exist.

Don’t use plain printf():

```
printf("Interrupt timeout\n"); /* Which device? Unknown. */
```

Rule 5: Rate-limit warnings in repetitive error paths

If an error can repeat rapidly (e.g., DMA timeout on every frame), rate-limit it:

```
static int
myfirst_check_fifo(struct myfirst_softc *sc)
{
    if (fifo_is_full(sc)) {
        if (sc->log_fifo_full == 0) {
            device_printf(sc->dev, "FIFO full, dropping packets\n");
            sc->log_fifo_full = 1; /* Only log once until cleared */
        }
        return (ENOSPC);
    }
    sc->log_fifo_full = 0; /* Clear flag when condition resolves */
    return (0);
}
```

This pattern logs the first occurrence, suppresses repeats, and logs again when the condition changes.

Rule 6: Be concise and actionable

Compare:

Bad:

```
device_printf(dev, "Something went wrong in the code here\n");
```

Good:

```
device_printf(dev, "Failed to map BAR0 MMIO region: error %d\n", error);
```

The good example tells you **what** failed, **where** (BAR0), and **how** (error code).

7.5.3 Logging Patterns for Common Events

Attach success:

```
device_printf(dev, "Attached successfully, hardware rev %d.%d\n",
    hw_major, hw_minor);
```

Attach failure:

```
device_printf(dev, "Attach failed: could not allocate IRQ\n");
goto fail;
```

Detach:

```
device_printf(dev, "Detached, uptime %lu seconds\n",
    (unsigned long)(ticks - sc->attach_ticks) / hz);
```

Resource allocation failure:

```
if (sc->mem_res == NULL) {
    device_printf(dev, "Could not allocate memory resource\n");
    error = ENXIO;
    goto fail;
}
```

Unexpected hardware state:

```
if (status & DEVICE_ERROR_BIT) {
    device_printf(dev, "Hardware reported error 0x%x\n", status);
    /* attempt recovery or fail */
}
```

First open:

```
if (sc->open_count == 0) {
    device_printf(dev, "Device opened for the first time\n");
}
```

(But only if this is unusual or noteworthy; don't log every open in production.)

7.5.4 Rate-Limited Logging Macro (Advanced Preview)

For errors that might repeat rapidly, you can define a rate-limited logging macro:

```
#define MYFIRST_RATELIMIT_HZ 1 /* Max once per second */

static int
myfirst_log_ratelimited(struct myfirst_softc *sc, const char *fmt, ...)
{
    static time_t last_log = 0;
    time_t now;
```

```
    va_list ap;

    now = time_second;
    if (now - last_log < MYFIRST_RATELIMIT_HZ)
        return (0); /* Too soon, skip */

    last_log = now;

    va_start(ap, fmt);
    device_vprintf(sc->dev, fmt, ap);
    va_end(ap);

    return (1);
}
```

Usage:

```
if (error_condition) {
    myfirst_log_ratelimited(sc, "DMA timeout occurred\n");
}
```

This limits the log to **once per second**, even if the condition triggers thousands of times.

When to use: Only for hot-path errors that could spam logs (interrupt storms, queue overflows, etc.). Not needed for attach/detach or rare errors.

7.5.5 What to Log During Development vs Production

Development (verbose):

- Every function entry/exit (guarded by a debug flag)
- Register reads/writes
- State transitions
- Resource allocation/deallocation

Production (quiet):

- Attach/detach lifecycle
- Errors
- Critical state changes (link up/down, device reset)
- First occurrence of repetitive errors

Transition: Start verbose, then pare down as the driver stabilizes. Leave debug logging behind compile-time or sysctl guards for future troubleshooting.

7.5.6 Using Sysctls for Debug Logging

Instead of hardcoding verbosity, expose a sysctl:

```
static int myfirst_debug = 0;
SYSCTL_INT(_hw_myfirst, OID_AUTO, debug, CTLFLAG_RWTUN,
    &myfirst_debug, 0, "Enable debug logging");
```

Then wrap debug logs:

```
if (myfirst_debug) {
    device_printf(dev, "DEBUG: entering attach\n");
}
```

Benefit: Users or developers can enable logging without recompiling:

```
% sysctl hw.myfirst.debug=1
```

We'll cover sysctls in detail in the next section; this is just a preview.

7.5.7 Inspecting Logs

View all kernel messages:

```
% dmesg -a
```

View recent messages:

```
% dmesg | tail -n 20
```

Search for your driver:

```
% dmesg | grep myfirst
```

Clear the message buffer (if testing repeatedly):

```
% sudo dmesg -c > /dev/null
```

(Not always advisable, but useful when you want a clean slate for testing.)

7.5.8 Example: Logging in myfirst

Let's add disciplined logging to our driver. Update `attach()` and `detach()`:

Updated attach:

```
static int
myfirst_attach(device_t dev)
{
    struct myfirst_softc *sc;

    sc = device_get_softc(dev);
    sc->dev = dev;
    sc->unit = device_get_unit(dev);

    /* Initialize mutex */
    mtx_init(&sc->mtx, device_get_nameunit(dev), "myfirst", MTX_DEF);

    /* Record attach time */
    sc->attach_ticks = ticks;
    sc->is_attached = 1;

    /* Log attach success */
    device_printf(dev, "Attached successfully at tick %lu\n",
        (unsigned long)sc->attach_ticks);

    return (0);
}
```

Updated detach:

```
static int
myfirst_detach(device_t dev)
{
    struct myfirst_softc *sc;
    uint64_t uptime_ticks;

    sc = device_get_softc(dev);

    /* Refuse detach if open */
    if (sc->is_open) {
        device_printf(dev, "Cannot detach: device is open\n");
        return (EBUSY);
    }
}
```

```
/* Calculate uptime */
uptime_ticks = ticks - sc->attach_ticks;

/* Log detach */
device_printf(dev, "Detaching: uptime %lu ticks, opened %lu times\n",
    (unsigned long)uptime_ticks,
    (unsigned long)sc->open_count);

/* Cleanup */
mtx_destroy(&sc->mtx);
sc->is_attached = 0;

return (0);
}
```

What we're logging:

- **Attach:** Confirms success and records when.
- **Detach refusal:** If device is open, explain why detach failed.
- **Detach success:** Shows uptime and usage stats.

This gives users and developers clear visibility into lifecycle events.

7.5.9 Common Logging Mistakes

Mistake 1: Logging inside locks

Wrong:

```
mtx_lock(&sc->mtx);
device_printf(dev, "Locked, doing work\n"); /* Can cause priority inversion */
/* ... work ... */
mtx_unlock(&sc->mtx);
```

Why it's wrong: `device_printf()` can block (acquiring internal locks). Calling it while holding your mutex can lead to deadlock or priority inversion.

Right:

```
mtx_lock(&sc->mtx);
/* ... work ... */
mtx_unlock(&sc->mtx);

device_printf(dev, "Work completed\n"); /* Log after releasing lock */
```

Mistake 2: Multi-line logs that can interleave

Wrong:

```
printf("myfirst0: Attach starting\n");
printf("myfirst0: Step 1\n");
printf("myfirst0: Step 2\n");
```

Why it's wrong: If another driver or kernel component logs between your lines, your message gets fragmented.

Right:

```
device_printf(dev, "Attach starting: step 1, step 2 completed\n");
```

Or use a single sbuf (string buffer) and print it once (advanced).

Mistake 3: Logging sensitive data

Don't log:

- User data (packet contents, file data, etc.)
- Cryptographic keys or secrets
- Anything that violates privacy expectations

Always assume logs are public.

7.5.10 Quick Self-Check

Before moving forward, confirm you understand:

1. **When must you log?** Answer: Lifecycle events (attach/detach), errors, critical state changes.
2. **When must you not log?** Answer: Hot paths (interrupts, read/write loops, per-packet operations).
3. **Why use `device_printf()` instead of `printf()`?** Answer: Automatically includes device name, making logs clearer.
4. **How do you rate-limit a log that could repeat rapidly?** Answer: Use a flag or timestamp to track last log time, and suppress repeats.
5. **What should every error log include?** Answer: What failed, why (error code), and enough context to diagnose.

If those are clear, you're ready to add your first user-visible surface: a `/dev` node.

7.6 A Temporary User Surface: /dev (Preview Only)

Every driver needs a way for user programs to interact with it. For character devices, that surface is a **device node** in /dev. In this section, we'll create /dev/myfirst0, but we won't implement full I/O yet, just enough to show the pattern and prove the device is reachable from user-space.

This is a **preview**, not the full implementation. Real read() and write() semantics come in **Chapter 8 and 9**. Here, we're focusing on:

- Creating the /dev node using make_dev_s()
- Defining a cdevsw (character device switch) with stubbed methods
- Handling open() and close() to track device state
- Cleaning up the device node in detach()

Think of this as **wiring the front door** before furnishing the house. The door opens and closes, but the rooms are empty.

7.6.1 What Is a Character Device Switch (cdevsw)?

The **cdevsw** is a structure containing function pointers for character device operations: open, close, read, write, ioctl, mmap, etc. When a user program calls open("/dev/myfirst0", ...), the kernel looks up the cdevsw associated with that device node and calls your d_open function.

Structure definition (abbreviated):

```
struct cdevsw {
    int      d_version;    /* D_VERSION (API version) */
    d_open_t  *d_open;     /* open(2) handler */
    d_close_t  *d_close;   /* close(2) handler */
    d_read_t   *d_read;    /* read(2) handler */
    d_write_t  *d_write;   /* write(2) handler */
    d_ioctl_t  *d_ioctl;   /* ioctl(2) handler */
    const char *d_name;    /* Device name */
    /* ... more fields ... */
};
```

Key insight: You provide implementations for the operations your device supports, and leave others NULL (which the kernel interprets as “not supported” or “default behavior”).

7.6.2 Defining the cdevsw for myfirst

We'll define a minimal cdevsw with open and close handlers, and stub read/write for now.

Add this near the top of myfirst.c, after the softc definition:

```

/* Forward declarations for cdevsw methods */
static d_open_t      myfirst_open;
static d_close_t     myfirst_close;
static d_read_t      myfirst_read;
static d_write_t     myfirst_write;

/*
 * Character device switch.
 *
 * Maps system calls to our driver functions.
 */
static struct cdevsw myfirst_cdevsw = {
    .d_version =    D_VERSION,
    .d_open =      myfirst_open,
    .d_close =     myfirst_close,
    .d_read =      myfirst_read,
    .d_write =     myfirst_write,
    .d_name =      "myfirst",
};

```

What this means:

- `d_version = D_VERSION` - Required API version stamp.
- `d_open = myfirst_open` - When user calls `open("/dev/myfirst0", ...)`, kernel calls `myfirst_open()`.
- Similar for `close`, `read`, `write`.
- `d_name = "myfirst"` - Base name for device nodes (combined with unit number to form `myfirst0`, `myfirst1`, etc.).

7.6.3 Implementing `open()`

The `open()` handler is called when a user program opens the device node. It's your chance to:

- Verify the device is ready
- Track open state (increment counters, set flags)
- Return an error if the device can't be opened (e.g., exclusive access, hardware not ready)

Add this function:

```

/*
 * open() handler.
 *
 * Called when a user program opens /dev/myfirst0.
 */
static int

```

```

myfirst_open(struct cdev *dev, int oflags, int devtype, struct thread *td)
{
    struct myfirst_softc *sc;

    sc = dev->si_drv1; /* Retrieve softc from cdev */

    if (sc == NULL || !sc->is_attached) {
        return (ENXIO); /* Device not ready */
    }

    mtx_lock(&sc->mtx);
    if (sc->is_open) {
        mtx_unlock(&sc->mtx);
        return (EBUSY); /* Only allow one opener (exclusive access) */
    }

    sc->is_open = 1;
    sc->open_count++;
    mtx_unlock(&sc->mtx);

    device_printf(sc->dev, "Device opened (count: %lu)\n",
        (unsigned long)sc->open_count);

    return (0);
}

```

What this does:

- **sc = dev->si_drv1** - Retrieves the softc. When we create the device node, we'll stash the softc pointer here.
- **if (!sc->is_attached)** - Sanity check. If the device isn't attached, refuse to open.
- **if (sc->is_open) return (EBUSY)** - Enforce exclusive access (only one opener at a time). Real devices might allow multiple openers; this is just a simple example.
- **sc->is_open = 1** - Mark device as open.
- **sc->open_count++** - Increment lifetime open counter.
- **device_printf()** - Log the open event (for now; you'd remove this in production).

Lock discipline: We hold the mutex while checking and updating `is_open`, ensuring thread safety.

7.6.4 Implementing close()

The `close()` handler is called when the last reference to the open device is released. Clean up open-specific state here.

Add this function:

```
/*
 * close() handler.
 *
 * Called when the user program closes /dev/myfirst0.
 */
static int
myfirst_close(struct cdev *dev, int fflag, int devtype, struct thread *td)
{
    struct myfirst_softc *sc;

    sc = dev->si_drv1;

    if (sc == NULL) {
        return (ENXIO);
    }

    mtx_lock(&sc->mtx);
    sc->is_open = 0;
    mtx_unlock(&sc->mtx);

    device_printf(sc->dev, "Device closed\n");

    return (0);
}
```

What this does:

- Clears the `is_open` flag.
- Logs the close event.
- Returns 0 (success).

Simple pattern: `open()` sets flags, `close()` clears them.

7.6.5 Stubbing `read()` and `write()`

We'll implement minimal stubs that return success but do nothing. This proves the device node is wired correctly without committing to I/O semantics yet.

Stub `read()`:

```
/*
 * read() handler (stubbed).
 *
 * For now, just return EOF (0 bytes read).
 * Real implementation in Chapter 9.
 */
```

```
*/
static int
myfirst_read(struct cdev *dev, struct uio *uio, int ioflag)
{
    /* Return EOF immediately */
    return (0);
}
```

Stub write():

```
/*
 * write() handler (stubbed).
 *
 * For now, pretend we wrote everything.
 * Real implementation in Chapter 9.
 */
static int
myfirst_write(struct cdev *dev, struct uio *uio, int ioflag)
{
    /* Pretend we consumed all bytes */
    uio->uio_resid = 0;
    return (0);
}
```

What these do:

- **read()** - Returns 0 (EOF), meaning “no data available.”
- **write()** - Sets `uio->uio_resid = 0`, meaning “all bytes written.”

User programs will see this as a device that “accepts writes but discards them” and “reads return EOF immediately.” Not useful yet, but proves the plumbing works.

7.6.6 Creating the Device Node in `attach()`

Now we tie it all together. In `attach()`, create the `/dev` node and associate it with your softc.

Add this to the end of `myfirst_attach()`, just before `return (0)`:

```
/* Create /dev node */
{
    struct make_dev_args args;
    int error;

    make_dev_args_init(&args);
    args.mda_devsw = &myfirst_cdevsw;
```

```

        args.mda_uid = UID_ROOT;
        args.mda_gid = GID_WHEEL;
        args.mda_mode = 0600; /* rw----- (root only) */
        args.mda_si_drv1 = sc; /* Stash softc pointer */

        error = make_dev_s(&args, &sc->cdev, "myfirst%d", sc->unit);
        if (error != 0) {
            device_printf(dev, "Failed to create device node: error %d\n",
↪ error);

            mtx_destroy(&sc->mtx);
            return (error);
        }

        device_printf(dev, "Created /dev/%s\n", devtoname(sc->cdev));

```

What this does:

- **make_dev_args_init(&args)** - Initializes the args structure with defaults.
- **args.mda_devsw = &myfirst_cdevsw** - Associates this cdev with our cdevsw.
- **args.mda_uid / gid / mode** - Sets ownership and permissions. 0600 means root read/write only.
- **args.mda_si_drv1 = sc** - Stores the softc pointer so open() / close() can retrieve it.
- **make_dev_s(&args, &sc->cdev, "myfirst%d", sc->unit)** - Creates /dev/myfirst0 (or myfirst1, etc., based on unit number).
- **Error handling:** If make_dev_s() fails, destroy the mutex and return the error.

Important: We save the struct cdev * in sc->cdev so we can destroy it later in detach().

Add the cdev field to the softc:

Update struct myfirst_softc:

```

struct myfirst_softc {
    device_t      dev;
    int           unit;
    struct mtx     mtx;
    uint64_t      attach_ticks;
    uint64_t      open_count;
    uint64_t      bytes_read;
    int           is_attached;
    int           is_open;

    struct cdev    *cdev; /* /dev node */
};

```

7.6.7 Destroying the Device Node in detach()

In `detach()`, you must remove the `/dev` node **before** the softc is freed.

Add this near the start of `myfirst_detach()`, after the `is_open` check:

```
/* Destroy /dev node */
if (sc->cdev != NULL) {
    destroy_dev(sc->cdev);
    sc->cdev = NULL;
}
```

What this does:

- **destroy_dev(sc->cdev)** - Removes `/dev/myfirst0` from the filesystem. Any open file descriptors are invalidated, and subsequent operations on them return errors.
- **sc->cdev = NULL** - Clear the pointer (defensive programming).

Order matters: Destroy the device node **before** destroying the mutex or freeing other resources. This ensures no user-space operations can reach your driver after detach starts tearing things down.

7.6.8 Build, Test, and Verify

Let's compile and test the new device node:

1. Clean and build:

```
% make clean && make
```

2. Load the driver:

```
% sudo kldload ./myfirst.ko
% dmesg | tail -n 3
myfirst0: <My First FreeBSD Driver> on nexus0
myfirst0: Attached successfully at tick 123456
myfirst0: Created /dev/myfirst0
```

3. Check the device node:

```
% ls -l /dev/myfirst0
crw----- 1 root  wheel  0x5a Nov  6 15:45 /dev/myfirst0
```

Success! The device node exists.

4. Test open and close:

```
% sudo sh -c 'cat < /dev/myfirst0'
(no output, immediate EOF)
```

Check dmesg:

```
% dmesg | tail -n 2
myfirst0: Device opened (count: 1)
myfirst0: Device closed
```

Your `open()` and `close()` handlers ran.

5. Test write:

```
% sudo sh -c 'echo "hello" > /dev/myfirst0'
% dmesg | tail -n 2
myfirst0: Device opened (count: 2)
myfirst0: Device closed
```

Write succeeded (though the data was discarded).

6. Unload the driver:

```
% sudo kldunload myfirst
% ls -l /dev/myfirst0
ls: /dev/myfirst0: No such file or directory
```

The device node was correctly destroyed on unload.

7.6.9 What Just Happened?

- You created a character device switch (`cdevsw`) mapping syscalls to your functions.
- You implemented `open()` and `close()` handlers that track state.
- You stubbed `read()` and `write()` to prove the plumbing works.
- You created `/dev/myfirst0` in `attach()` and destroyed it in `detach()`.
- User programs can now `open("/dev/myfirst0", ...)` and interact with your driver.

7.6.10 Common Device Node Mistakes

Mistake 1: Forgetting to destroy the device node in `detach`

Wrong:


```
static int
myfirst_detach(device_t dev)
{
    struct myfirst_softc *sc = device_get_softc(dev);
    mtx_destroy(&sc->mtx);
    return (0);
    /* Forgot destroy_dev(sc->cdev)! */
}
```

Why it's wrong: The device node persists after unload. Attempting to open it crashes the kernel (code is gone, but the node remains).

Right: Always call `destroy_dev()` in `detach`.

Mistake 2: Accessing `softc` in `open/close` without checking `is_attached`

Wrong:

```
static int
myfirst_open(struct cdev *dev, ...)
{
    struct myfirst_softc *sc = dev->si_drv1;
    /* No check if sc or sc->is_attached is valid */
    mtx_lock(&sc->mtx); /* Might be NULL or freed! */
    ...
}
```

Why it's wrong: If `detach()` runs concurrently, the `softc` might be invalid.

Right: Check `sc != NULL` and `sc->is_attached` before accessing state.

Mistake 3: Using `make_dev()` instead of `make_dev_s()`

Old pattern:

```
sc->cdev = make_dev(&myfirst_cdevsw, sc->unit, UID_ROOT, GID_WHEEL, 0600, "myfirst%d",
    ↪ sc->unit);
if (sc->cdev == NULL) {
    /* Error handling */
}
```

Why it's dated: `make_dev()` can fail and return `NULL`, requiring awkward error checks.

Modern pattern: `make_dev_s()` returns an error code, making error handling cleaner:

```
error = make_dev_s(&args, &sc->cdev, "myfirst%d", sc->unit);
if (error != 0) {
    /* Handle error */
}
```

Prefer `make_dev_s()` in new code.

7.6.11 Quick Self-Check

Before moving forward, confirm:

1. **What is the `cdevsw`?** Answer: A structure mapping syscalls (`open`, `read`, `write`, etc.) to driver functions.
2. **How does `open()` retrieve the `softc`?** Answer: Via `dev->si_drv1`, which we set when creating the device node.
3. **When must you call `destroy_dev()`?** Answer: In `detach()`, before the `softc` is freed.
4. **Why do we check `is_attached` in `open()`?** Answer: To ensure the device hasn't started detaching, which could lead to accessing freed memory.
5. **What does the `stub read()` return?** Answer: 0 (EOF), indicating no data available.

If those are clear, you're ready to add **observability via `sysctl`** in the next section.

7.7 A Tiny Control Plane: Read-Only `sysctl`

Device nodes in `/dev` let user programs send and receive data, but they're not the only way to expose your driver to the outside world. **Sysctls** provide a lightweight control and observability plane, letting users and administrators query driver state, read statistics, and (optionally) tune parameters at runtime.

In this section, we'll add a **read-only `sysctl`** that exposes basic driver statistics. This gives you a taste of FreeBSD's `sysctl` infrastructure without committing to full read-write tunables or complex hierarchies (those return when we look at observability and debugging in Part 5).

By the end, you'll have:

- A `sysctl` node under `dev.myfirst.0.*` showing attach time, open count, and bytes read
- Understanding of static vs dynamic `sysctls`
- A pattern you can extend later for more complex observability

7.7.1 Why Sysctls Matter

Sysctls provide **out-of-band observability**, a way to inspect driver state without opening the device or triggering I/O. They're essential for:

- **Debugging:** “Is the driver actually attached? What’s the current state?”
- **Monitoring:** “How many times has this device been opened? Any errors?”
- **Tuning:** (Read-write sysctls, covered later) “Adjust the buffer size or timeout value.”

Example use case: A network interface might expose `dev.em.0.rx_packets` and `dev.em.0.tx_errors` so monitoring tools can track performance without analyzing packet flows.

Mental model: Sysctls are like a “status dashboard” on the side of your driver, visible via `sysctl` commands without affecting normal operation.

7.7.2 The FreeBSD Sysctl Tree

Sysctls are organized hierarchically, like a filesystem:

```
kern.ostype = "FreeBSD"
hw.ncpu = 8
dev.em.0.rx_packets = 123456
```

Common top-level branches:

- `kern.*` - Kernel parameters
- `hw.*` - Hardware information
- `dev.*` - Device-specific nodes (this is where your driver goes)
- `net.*` - Network stack parameters

Your driver’s namespace: `dev.<drivername>.<unit>.*`

For `myfirst`, that means `dev.myfirst.0.*` for the first instance.

7.7.3 Static vs Dynamic Sysctls

Static sysctls:

- Declared at compile time using `SYSCCTL_*` macros
- Simple to define, but can’t be created/destroyed dynamically
- Good for driver-wide settings or constants

Example:

```
static int myfirst_debug = 0;
SYSCCTL_INT(_hw, OID_AUTO, myfirst_debug, CTLFLAG_RWTUN,
    &myfirst_debug, 0, "Enable debug logging");
```

Dynamic sysctls:

- Created at runtime (usually in `attach()`)
- Can be destroyed in `detach()`
- Good for per-device state (like stats for `myfirst0`, `myfirst1`, etc.)

For this chapter, we'll use **dynamic sysctls** so each device instance has its own nodes.

7.7.4 Adding a Sysctl Context to the softc

Dynamic sysctls require a **sysctl context** (`struct sysctl_ctx_list`) to track nodes you create. This makes cleanup automatic when you free the context.

Add these fields to `struct myfirst_softc`:

```
struct myfirst_softc {
    device_t      dev;
    int           unit;
    struct mtx     mtx;
    uint64_t      attach_ticks;
    uint64_t      open_count;
    uint64_t      bytes_read;
    int           is_attached;
    int           is_open;
    struct cdev    *cdev;

    /* Sysctl context for dynamic nodes */
    struct sysctl_ctx_list sysctl_ctx;
    struct sysctl_oid      *sysctl_tree; /* Root of our subtree */
};
```

What these fields do:

- `sysctl_ctx` - Tracks all sysctl nodes we create. When we call `sysctl_ctx_free()`, all nodes are destroyed automatically.
- `sysctl_tree` - Root OID (Object Identifier) for `dev.myfirst.0.*`. Child nodes attach here.

7.7.5 Creating the Sysctl Tree in `attach()`

Add this code to `myfirst_attach()`, after creating the `/dev` node:

```
/* Initialize sysctl context */
sysctl_ctx_init(&sc->sysctl_ctx);

/* Create device sysctl tree: dev.myfirst.0 */
sc->sysctl_tree = SYSCTL_ADD_NODE(&sc->sysctl_ctx,
    SYSCTL_CHILDREN(device_get_sysctl_tree(dev)),
```

```

    OID_AUTO, "stats", CTLFLAG_RD | CTLFLAG_MPSAFE, 0,
    "Driver statistics");

if (sc->sysctl_tree == NULL) {
    device_printf(dev, "Failed to create sysctl tree\n");
    destroy_dev(sc->cdev);
    mtx_destroy(&sc->mtx);
    return (ENOMEM);
}

/* Add individual sysctl nodes */
SYSCTL_ADD_U64(&sc->sysctl_ctx,
    SYSCTL_CHILDREN(sc->sysctl_tree),
    OID_AUTO, "attach_ticks", CTLFLAG_RD,
    &sc->attach_ticks, 0, "Tick count when driver attached");

SYSCTL_ADD_U64(&sc->sysctl_ctx,
    SYSCTL_CHILDREN(sc->sysctl_tree),
    OID_AUTO, "open_count", CTLFLAG_RD,
    &sc->open_count, 0, "Number of times device was opened");

SYSCTL_ADD_U64(&sc->sysctl_ctx,
    SYSCTL_CHILDREN(sc->sysctl_tree),
    OID_AUTO, "bytes_read", CTLFLAG_RD,
    &sc->bytes_read, 0, "Total bytes read from device");

device_printf(dev, "Sysctl tree created under dev.myfirst.%d.stats\n",
    sc->unit);

```

What this does:

1. **sysctl_ctx_init(&sc->sysctl_ctx)** - Initializes the context (must be first).
2. **SYSCTL_ADD_NODE()** - Creates a subtree node `dev.myfirst.0.stats`. Parameters:
 - `&sc->sysctl_ctx` - Context that owns this node.
 - `SYSCTL_CHILDREN(device_get_sysctl_tree(dev))` - Parent (the device's sysctl tree).
 - `OID_AUTO` - Auto-assign an OID number.
 - `"stats"` - Node name.
 - `CTLFLAG_RD | CTLFLAG_MPSAFE` - Read-only, MP-safe.
 - `0` - Handler function (none for a node).
 - `"Driver statistics"` - Description.
3. **SYSCTL_ADD_U64()** - Adds a 64-bit unsigned integer sysctl. Parameters:
 - `&sc->sysctl_ctx` - Context.
 - `SYSCTL_CHILDREN(sc->sysctl_tree)` - Parent (stats subtree).

- `OID_AUTO` - Auto-assign OID.
- `"attach_ticks"` - Leaf name.
- `CTLFLAG_RD` - Read-only.
- `&sc->attach_ticks` - Pointer to the variable to expose.
- `0` - Format hint (0 = default).
- `"Tick count..."` - Description.

4. **Error handling:** If node creation fails, clean up and return `ENOMEM`.

Result: You now have three sysctls:

- `dev.myfirst.0.stats.attach_ticks`
- `dev.myfirst.0.stats.open_count`
- `dev.myfirst.0.stats.bytes_read`

7.7.6 Destroying the Sysctl Tree in `detach()`

Cleanup is simple: free the context, and all nodes are destroyed automatically.

Add this to `myfirst_detach()`, after destroying the device node:

```
/* Free sysctl context (destroys all nodes) */
sysctl_ctx_free(&sc->sysctl_ctx);
```

That's it. One line cleans up everything.

Why it's safe: `sysctl_ctx_free()` walks the context's list and removes each node. As long as you created them all via the context, cleanup is automatic.

7.7.7 Build, Load, and Test Sysctls

1. Clean and build:

```
% make clean && make
```

2. Load the driver:

```
% sudo kldload ./myfirst.ko
% dmesg | tail -n 4
myfirst0: <My First FreeBSD Driver> on nexus0
myfirst0: Attached successfully at tick 123456
myfirst0: Created /dev/myfirst0
myfirst0: Sysctl tree created under dev.myfirst.0.stats
```

3. Query the sysctls:

```
% sysctl dev.myfirst.0.stats
dev.myfirst.0.stats.attach_ticks: 123456
dev.myfirst.0.stats.open_count: 0
dev.myfirst.0.stats.bytes_read: 0
```

4. Open the device and check again:

```
% sudo sh -c 'cat < /dev/myfirst0'
% sysctl dev.myfirst.0.stats.open_count
dev.myfirst.0.stats.open_count: 1
```

The counter incremented!

5. Unload and verify cleanup:

```
% sudo kldunload myfirst
% sysctl dev.myfirst.0.stats
sysctl: unknown oid 'dev.myfirst.0.stats'
```

Sysctls were correctly destroyed.

7.7.8 Making Sysctls More Useful

Right now, the sysctls just expose raw numbers. Let's make them more user-friendly.

Add a human-readable uptime sysctl:

Instead of exposing raw tick counts, compute uptime in seconds.

Add a handler function:

```
/*
 * Sysctl handler for uptime_seconds.
 *
 * Computes how long the driver has been attached, in seconds.
 */
static int
sysctl_uptime_seconds(SYSCTL_HANDLER_ARGS)
{
    struct myfirst_softc *sc = arg1;
    uint64_t uptime;

    uptime = (ticks - sc->attach_ticks) / hz;

    return (sysctl_handle_64(oidp, &uptime, 0, req));
}
```

Register it in attach():

```

SYSCTL_ADD_PROC(&sc->sysctl_ctx,
    SYSCTL_CHILDREN(sc->sysctl_tree),
    OID_AUTO, "uptime_seconds", CTLTYPE_U64 | CTLFLAG_RD | CTLFLAG_MPSAFE,
    sc, 0, sysctl_uptime_seconds, "QU",
    "Seconds since driver attached");

```

Test:

```

% sysctl dev.myfirst.0.stats.uptime_seconds
dev.myfirst.0.stats.uptime_seconds: 42

```

Much more readable than raw tick counts!

7.7.9 Read-Only vs Read-Write Sysctls

Our sysctls are read-only (CTLFLAG_RD). To make them writable, use CTLFLAG_RW and add a handler that validates input.

Example (preview only, not implemented now):

```

static int
sysctl_set_debug_level(SYSCTL_HANDLER_ARGS)
{
    struct myfirst_softc *sc = arg1;
    int new_level, error;

    new_level = sc->debug_level;
    error = sysctl_handle_int(oidp, &new_level, 0, req);
    if (error != 0 || req->newptr == NULL)
        return (error);

    if (new_level < 0 || new_level > 3)
        return (EINVAL); /* Reject invalid values */

    sc->debug_level = new_level;
    device_printf(sc->dev, "Debug level set to %d\n", new_level);

    return (0);
}

```

We'll return to read-write sysctls in Part 5, when we look at debugging and observability tooling in depth. For now, read-only exposure is enough.

7.7.10 Sysctl Best Practices

1. Expose meaningful metrics

- Counters (packets, errors, opens, closes)
- State flags (attached, open, enabled)
- Derived values (uptime, throughput, utilization)

Don't expose:

- Internal pointers or addresses (security risk)
 - Meaningless raw data (use handlers to format nicely)
-

2. Use descriptive names and descriptions

Good:

```
SYCTL_ADD_U64(..., "rx_packets", ..., "Packets received");
```

Bad:

```
SYCTL_ADD_U64(..., "cnt1", ..., "Counter");
```

3. Group related sysctls under subtrees

```
dev.myfirst.0.stats.*    (statistics)
dev.myfirst.0.config.*   (tunable parameters)
dev.myfirst.0.debug.*    (debug flags and counters)
```

4. Protect concurrent access

If a sysctl reads or writes shared state, hold the appropriate lock:

```
static int
sysctl_read_counter(SYSCTL_HANDLER_ARGS)
{
    struct myfirst_softc *sc = arg1;
    uint64_t value;
```

```
    mtx_lock(&sc->mtx);
    value = sc->some_counter;
    mtx_unlock(&sc->mtx);

    return (sysctl_handle_64(oidp, &value, 0, req));
}
```

5. Clean up in detach

Always call `sysctl_ctx_free(&sc->sysctl_ctx)` in detach, or you leak OIDs.

7.7.11 Common Sysctl Mistakes

Mistake 1: Forgetting `sysctl_ctx_init`

Wrong:

```
SYSCTL_ADD_NODE(&sc->sysctl_ctx, ...); /* Context not initialized! */
```

Why it's wrong: Uninitialized context causes panics or leaks.

Right: Call `sysctl_ctx_init(&sc->sysctl_ctx)` in attach before adding nodes.

Mistake 2: Not freeing the context in detach

Wrong:

```
static int
myfirst_detach(device_t dev)
{
    /* ... destroy other resources ... */
    return (0);
    /* Forgot sysctl_ctx_free! */
}
```

Why it's wrong: Sysctl nodes persist after unload. Next access crashes.

Right: Always `sysctl_ctx_free(&sc->sysctl_ctx)` in detach.

Mistake 3: Exposing raw pointers

Wrong:

```
SYSCTL_ADD_PTR(..., "softc_addr", ..., &sc, ...); /* Security hole! */
```

Why it's wrong: Leaks kernel address space layout (KASLR bypass).

Right: Never expose pointers via sysctls.

7.7.12 Quick Self-Check

Before moving forward, confirm:

1. **What is a sysctl?** Answer: A kernel variable or computed value exposed via the `sysctl` command.
2. **Where do driver sysctls live?** Answer: Under `dev.<drivername>.<unit>.*`.
3. **What must you call in `attach()` before adding nodes?** Answer: `sysctl_ctx_init(&sc->sysctl_ctx)`.
4. **What cleans up all sysctl nodes in `detach()`?** Answer: `sysctl_ctx_free(&sc->sysctl_ctx)`.
5. **Why use a handler instead of exposing a variable directly?** Answer: To compute derived values (like uptime) or validate writes.

If those are clear, you're ready to learn **error handling and clean unwind** in the next section.

7.8 Error Paths & Clean Unwind

So far, we've written `attach()` assuming everything succeeds. But real drivers must handle failures gracefully: if memory allocation fails, if a resource isn't available, or if hardware misbehaves, your driver must **undo what it started** and return an error without leaving partial state behind.

This section teaches the **single-label unwind pattern**, the standard FreeBSD idiom for error cleanup. Master this, and your driver will never leak resources, no matter where failure strikes.

7.8.1 Why Error Handling Matters

Poor error handling causes:

- **Resource leaks** (memory, locks, device nodes)
- **Kernel panics** (accessing freed memory, double-frees)
- **Inconsistent state** (device half-attached, locks initialized but not destroyed)

Real-world impact: A driver with sloppy error paths might work fine during normal operation, then panic the system when it encounters an unusual failure (out of memory, missing hardware, etc.).

Your goal: Ensure `attach()` either fully succeeds or fully fails, with no middle ground.

7.8.2 The Single-Label Unwind Pattern

FreeBSD kernel code uses a **goto-based unwind pattern** for cleanup. It looks like this:

```
static int
myfirst_attach(device_t dev)
{
    struct myfirst_softc *sc;
    int error;

    sc = device_get_softc(dev);
    sc->dev = dev;

    /* Step 1: Initialize mutex */
    mtx_init(&sc->mtx, "myfirst", NULL, MTX_DEF);

    /* Step 2: Allocate memory resource (example) */
    sc->mem_res = bus_alloc_resource_any(dev, SYS_RES_MEMORY,
        &sc->mem_rid, RF_ACTIVE);
    if (sc->mem_res == NULL) {
        device_printf(dev, "Failed to allocate memory resource\n");
        error = ENXIO;
        goto fail_mtx;
    }

    /* Step 3: Create device node */
    error = create_dev_node(sc);
    if (error != 0) {
        device_printf(dev, "Failed to create device node: %d\n", error);
        goto fail_mem;
    }

    /* Step 4: Create sysctls */
    error = create_sysctls(sc);
    if (error != 0) {
        device_printf(dev, "Failed to create sysctls: %d\n", error);
        goto fail_dev;
    }

    device_printf(dev, "Attached successfully\n");
    return (0);

fail_dev:
    destroy_dev(sc->cdev);
fail_mem:
```

```
        bus_release_resource(dev, SYS_RES_MEMORY, sc->mem_rid, sc->mem_res);
fail_mtx:
    mtx_destroy(&sc->mtx);
    return (error);
}
```

How it works:

- Each initialization step has a corresponding cleanup label.
- If a step fails, jump to the label that undoes everything **completed so far**.
- Labels are arranged in **reverse order** of initialization.
- Each label falls through to the next, so a failure at step 4 undoes 3, 2, and 1.

Why this pattern?

- **Centralized cleanup:** All error paths converge on one unwind sequence.
- **Easy to maintain:** Adding a new step means adding one goto and one cleanup label.
- **No duplication:** You don't repeat cleanup code in every error branch.

7.8.3 Applying the Pattern to myfirst

Let's refactor our attach() to handle errors properly.

Before (no error handling):

```
static int
myfirst_attach(device_t dev)
{
    struct myfirst_softc *sc = device_get_softc(dev);

    mtx_init(&sc->mtx, ...);
    create_dev_node(sc);
    create_sysctls(sc);

    return (0); /* What if something failed? */
}
```

After (with clean unwind):

```
static int
myfirst_attach(device_t dev)
{
    struct myfirst_softc *sc;
    struct make_dev_args args;
    int error;
```

```
sc = device_get_softc(dev);
sc->dev = dev;
sc->unit = device_get_unit(dev);

/* Step 1: Initialize mutex */
mtx_init(&sc->mtx, device_get_nameunit(dev), "myfirst", MTX_DEF);

/* Step 2: Record attach time and initialize state */
sc->attach_ticks = ticks;
sc->is_attached = 1;
sc->is_open = 0;
sc->open_count = 0;
sc->bytes_read = 0;

/* Step 3: Create /dev node */
make_dev_args_init(&args);
args.mda_devsw = &myfirst_cdevsw;
args.mda_uid = UID_ROOT;
args.mda_gid = GID_WHEEL;
args.mda_mode = 0600;
args.mda_si_drv1 = sc;

error = make_dev_s(&args, &sc->cdev, "myfirst%d", sc->unit);
if (error != 0) {
    device_printf(dev, "Failed to create device node: %d\n", error);
    goto fail_mtx;
}

/* Step 4: Initialize sysctl context */
sysctl_ctx_init(&sc->sysctl_ctx);

/* Step 5: Create sysctl tree */
sc->sysctl_tree = SYSCTL_ADD_NODE(&sc->sysctl_ctx,
    SYSCTL_CHILDREN(device_get_sysctl_tree(dev)),
    OID_AUTO, "stats", CTLFLAG_RD | CTLFLAG_MPSAFE, 0,
    "Driver statistics");

if (sc->sysctl_tree == NULL) {
    device_printf(dev, "Failed to create sysctl tree\n");
    error = ENOMEM;
    goto fail_dev;
}
```

```

/* Step 6: Add sysctl nodes */
SYSCTL_ADD_U64(&sc->sysctl_ctx,
    SYSCTL_CHILDREN(sc->sysctl_tree),
    OID_AUTO, "attach_ticks", CTLFLAG_RD,
    &sc->attach_ticks, 0, "Tick count when driver attached");

SYSCTL_ADD_U64(&sc->sysctl_ctx,
    SYSCTL_CHILDREN(sc->sysctl_tree),
    OID_AUTO, "open_count", CTLFLAG_RD,
    &sc->open_count, 0, "Number of times device was opened");

SYSCTL_ADD_U64(&sc->sysctl_ctx,
    SYSCTL_CHILDREN(sc->sysctl_tree),
    OID_AUTO, "bytes_read", CTLFLAG_RD,
    &sc->bytes_read, 0, "Total bytes read from device");

device_printf(dev, "Attached successfully\n");
return (0);

/* Error unwinding (in reverse order of initialization) */
fail_dev:
    destroy_dev(sc->cdev);
    sysctl_ctx_free(&sc->sysctl_ctx);
fail_mtx:
    mtx_destroy(&sc->mtx);
    sc->is_attached = 0;
    return (error);
}

```

Key improvements:

- Every operation that can fail is checked.
- Failures jump to the appropriate cleanup label.
- The unwind sequence undoes exactly what succeeded.
- All paths return an error code (never return success after a failure).

7.8.4 Label Naming Convention

Choose label names that indicate what needs undoing:

- `fail_mtx` - Destroy the mutex
- `fail_mem` - Release memory resource
- `fail_dev` - Destroy device node
- `fail_irq` - Release interrupt resource

Or use numbers if you prefer:

- fail1, fail2, fail3

Either works, but descriptive names make the code easier to read.

7.8.5 Testing Error Paths

Simulate a failure to verify your unwind logic works.

Add a deliberate failure after mutex init:

```
mtx_init(&sc->mtx, ...);

/* Simulate allocation failure for testing */
if (1) { /* Change to 0 to disable */
    device_printf(dev, "Simulated failure\n");
    error = ENXIO;
    goto fail_mtx;
}
```

Build and load:

```
% make clean && make
% sudo kldload ./myfirst.ko
```

Check dmesg:

```
% dmesg | tail -n 2
myfirst0: <My First FreeBSD Driver> on nexus0
myfirst0: Simulated failure
```

Verify cleanup:

```
% devinfo | grep myfirst
(no output - device didn't attach)

% ls /dev/myfirst*
ls: cannot access '/dev/myfirst*': No such file or directory
```

Key observation: The driver failed cleanly. No device node, no sysctl leaks, no panic.

Now disable the simulated failure and test normal attach again.

7.8.6 Common Error Handling Mistakes

Mistake 1: Not checking return values

Wrong:

```
make_dev_s(&args, &sc->cdev, "myfirst%d", sc->unit);  
/* Forgot to check error! */
```

Why it's wrong: If `make_dev_s()` fails, `sc->cdev` might be NULL or garbage, and you proceed as if everything's fine.

Right: Always check error and branch accordingly.

Mistake 2: Partial cleanup

Wrong:

```
fail_dev:  
    destroy_dev(sc->cdev);  
    return (error);  
/* Forgot to destroy mutex! */
```

Why it's wrong: Mutex remains initialized. Next load panics on re-init.

Right: Each label must undo **everything** initialized before it.

Mistake 3: Double-cleanup

Wrong:

```
fail_dev:  
    destroy_dev(sc->cdev);  
    mtx_destroy(&sc->mtx);  
    goto fail_mtx;  
  
fail_mtx:  
    mtx_destroy(&sc->mtx); /* Destroyed twice! */  
    return (error);  
}
```

Why it's wrong: Double-free or double-destroy causes panics.

Right: Each resource should be cleaned up exactly once, at its corresponding label.

Mistake 4: Returning success after failure

Wrong:

```
if (error != 0) {
    goto fail_mtx;
}
return (0); /* Even if we jumped to fail_mtx! */
```

Why it's wrong: The `goto` bypasses the `return`, but the pattern implies all error paths must **return an error code**.

Right: Ensure error labels end with `return (error)`.

7.8.7 The Full Picture: Attach and Detach

Attach logic:

1. Initialize resources in order.
2. Check each step for failure.
3. On failure, jump to the unwind label corresponding to the last successful step.
4. Unwind labels fall through in reverse order, cleaning up everything.

Detach logic:

Detach is simpler, undo everything in reverse order of `attach()`, assuming full success:

```
static int
myfirst_detach(device_t dev)
{
    struct myfirst_softc *sc = device_get_softc(dev);

    /* Refuse if device is open */
    if (sc->is_open) {
        device_printf(dev, "Cannot detach: device is open\n");
        return (EBUSY);
    }

    device_printf(dev, "Detaching\n");

    /* Undo in reverse order of attach */
    destroy_dev(sc->cdev);          /* Step 1: Drop user surface first */
    sysctl_ctx_free(&sc->sysctl_ctx); /* Step 2: Free sysctl context */
    mtx_destroy(&sc->mtx);          /* Step 3: Destroy mutex last */
    sc->is_attached = 0;
}
```

```
    return (0);  
}
```

Symmetry: Every `attach()` step has a corresponding `detach()` action, in reverse order.

Ordering note: We destroy the device (`destroy_dev`) before freeing the `sysctl` context and destroying the mutex. This follows the Chapter 6 pitfall guidance: “Destroy device before locks.” The `destroy_dev()` call blocks until all file operations drain, ensuring no code paths can reach our locks after the device is gone.

7.8.8 Defensive Programming Checklist

Before declaring your error paths done, check:

- ☐ Every function that can fail is checked
- ☐ Every error sets error and jumps to a cleanup label
- ☐ Every cleanup label undoes exactly what preceded it
- ☐ Labels are in reverse order of initialization
- ☐ `detach()` undoes everything `attach()` did, in reverse order
- ☐ No resource is freed twice
- ☐ No resource is leaked on failure

7.8.9 Quick Self-Check

Before moving forward, confirm:

1. **What is the single-label unwind pattern?** Answer: A goto-based cleanup sequence where each label undoes progressively more resources, in reverse order of initialization.
2. **Why are cleanup labels in reverse order?** Answer: Because you must undo the most recent step first, then earlier steps, walking backward through initialization.
3. **What must every error path do before returning?** Answer: Jump to the appropriate cleanup label and ensure `return (error)` is executed.
4. **How do you test error paths?** Answer: Simulate failures (e.g., force an allocation to fail) and verify cleanup is correct (no leaks, no panics).
5. **When should detach refuse to proceed?** Answer: When the device is still in use (e.g., `is_open` is true), return `EBUSY`.

If those are clear, you’re ready to explore **real driver examples in the FreeBSD source tree**.

7.9 Look-in-Tree Anchors (Reference Only)

You've built a minimal driver from first principles. Now let's **anchor your understanding** by pointing you to real FreeBSD 14.3 drivers that demonstrate the same patterns you've just learned. This section is a **guided tour**, not an exhaustive walk-through, think of it as a reading list for when you want to see how production code applies the lessons from this chapter.

7.9.1 Why Look at Real Drivers?

Real drivers show you:

- How patterns scale to complex hardware
- What code looks like in context (full file, not just snippets)
- Variations on the patterns (different attach logic, resource types, error handling styles)
- FreeBSD idioms and conventions in practice

You're not expected to understand every line of these drivers right now. The goal is to **recognize the scaffolding** you've already built, and see how it extends to more capable drivers.

7.9.2 Anchor 1: `/usr/src/sys/dev/null/null.c`

What it is: The `/dev/null`, `/dev/zero`, and `/dev/full` pseudo-devices.

Why study it:

- Simplest possible character devices
- No hardware, no resources, just `cdevsw + MOD_LOAD` handler
- Shows how `read()` and `write()` are implemented (even if trivial)
- Good reference for stubbed I/O

What to look for:

- The `cdevsw` structures (use `grep -n cdevsw` to find them)
- The `null_write()` and `zero_read()` handlers
- The module loader (`null_modevent()`, use `grep -n modevent` to find it)
- How `make_dev_credf()` is used (use `grep -n make_dev` to find it)

File location:

```
% less /usr/src/sys/dev/null/null.c
```

Quick scan:

```
% grep -n "cdevsw\|make_dev" /usr/src/sys/dev/null/null.c
```

7.9.3 Anchor 2: /usr/src/sys/dev/led/led.c

What it is: The LED control framework, used by platform-specific drivers to expose LEDs as /dev/led/*.

Why study it:

- Still simple, but shows resource management (callouts, lists)
- Demonstrates dynamic device creation per LED
- Uses locking (struct mtx)
- Shows how drivers manage multiple instances

What to look for:

- The ledsc structure (near the top of the file; use `grep -n ledsc` to find it), analogous to your softc
- The led_create() function (use `grep -n "led_create\|led_destroy"` to find it), which creates device nodes dynamically
- The led_destroy() function, cleanup pattern
- How the global LED list is protected with a mutex

File location:

```
% less /usr/src/sys/dev/led/led.c
```

Quick scan:

```
% grep -n "ledsc\|led_create\|led_destroy" /usr/src/sys/dev/led/led.c
```

7.9.4 Anchor 3: /usr/src/sys/net/if_tuntap.c

What it is: The tun and tap pseudo-network interfaces (tunnel devices).

Why study it:

- Hybrid driver: character device **and** network interface
- Shows how to register with ifnet (the network stack)
- More complex lifecycle (clone devices, per-open state)
- Good example of real-world locking and concurrency

What to look for:

- The struct tuntap_softc (use `grep -n "struct tuntap_softc"` to find it), much richer than yours
- The tun_create() function, which registers the ifnet
- The cdevsw and how it coordinates with the network side
- Use of if_attach() and if_detach() for network integration

File location:

```
% less /usr/src/sys/net/if_tuntap.c
```

Warning: This is a large, complex file (~2000 lines). Don't try to understand everything. Focus on:

```
% grep -n "tuntap_softc\|if_attach\|make_dev" /usr/src/sys/net/if_tuntap.c | head -20
```

7.9.5 Anchor 4: /usr/src/sys/dev/uart/uart_bus_pci.c

What it is: PCI attachment glue for UART (serial port) devices.

Why study it:

- Real hardware driver (PCI bus)
- Shows how `probe()` checks PCI IDs
- Demonstrates resource allocation (I/O ports, IRQs)
- Error unwinding in `attach()`

What to look for:

- The `uart_pci_probe()` function (use `grep -n uart_pci_probe` to find it), PCI ID matching
- The `uart_pci_attach()` function, resource allocation
- Use of `bus_alloc_resource()` and `bus_release_resource()`
- The `device_method_t` table (use `grep -n device_method` to find it)

File location:

```
% less /usr/src/sys/dev/uart/uart_bus_pci.c
```

Quick scan:

```
% grep -n "uart_pci_probe\|uart_pci_attach\|device_method"  
↪ /usr/src/sys/dev/uart/uart_bus_pci.c
```

Tip: This file is small (~250 lines) and very clean. It's a great example of a real Newbus driver.

7.9.6 Anchor 5: DRIVER_MODULE and MODULE_VERSION Patterns

Look for these macros at the bottom of driver files:

```
% grep -rln 'DRIVER_MODULE\|MODULE_VERSION' /usr/src/sys/dev/null/ /usr/src/sys/dev/led/
```

You will see the same registration patterns you used in `myfirst`. For drivers that attach to `nexus` and provide their own `identify` method, the pattern in `/usr/src/sys/crypto/aesni/aesni.c` and `/usr/src/sys/dev/sound/dummy.c` is the closest match to what you wrote.

7.9.7 How to Use These Anchors

1. Start with `null.c`

Read the whole file, it's short (~220 lines). You should recognize almost everything.

2. Skim `led.c`

Focus on the structure and lifecycle (creation/destruction). Don't get lost in the state machine.

3. Preview `if_tuntap.c`

Open it, scroll through, notice the hybrid structure (cdevsw + ifnet). Don't try to understand it all; just see the shape.

4. Study `uart_bus_pci.c`

Read `probe()` and `attach()`. This is your bridge to real hardware drivers (covered in Part 4).

5. Compare to your driver

For each anchor, ask:

- What's similar to my `myfirst` driver?
- What's different?
- What new concepts do I see (callouts, ifnet, PCI resources)?

6. Note what to learn next

When you see something unfamiliar (e.g., `callout_reset`, `if_attach`, `bus_alloc_resource`), jot it down. These are topics for later chapters.

7.9.8 Quick Tour: Common Patterns Across Drivers

Pattern	<code>null.c</code>	<code>led.c</code>	<code>if_tuntap.c</code>	<code>uart_pci.c</code>	<code>myfirst.c</code>
Uses <code>cdevsw</code>	yes	yes	yes	no	yes
Uses <code>ifnet</code>	no	no	yes	no	no
Newbus <code>probe/attach</code>	no	no	no (clone)	yes	yes
Has an <code>identify</code> method	no	no	no	no	yes
Module load handler	yes	yes	yes	no (Newbus)	no (Newbus)
Allocates <code>softc</code>	no	no	yes	yes	yes (Newbus)
Uses locking	no	yes	yes	yes	yes
Allocates bus resources	no	no	no	yes	no
Creates <code>/dev</code> nodes	yes	yes	yes	no	yes

7.9.9 What to Skip (For Now)

When reading these drivers, don't get stuck on:

- Hardware register access (`bus_read_4`, `bus_write_2`)
- Interrupt setup (`bus_setup_intr`, handler registration)
- DMA (`bus_dma_tag_create`, `bus_dmamap_load`)
- Advanced locking (read-mostly locks, lock order)
- Network packet handling (mbuf chains, `if_transmit`)

You'll learn these in their dedicated chapters. Right now, focus on **structure and lifecycle**.

7.9.10 Self-Study Exercise

Pick one anchor (`null.c` recommended for beginners) and:

1. Read the entire file
2. Identify the `cdevsw` structure
3. Find the `open`, `close`, `read`, `write` handlers
4. Trace the module load/unload flow
5. Compare to your `myfirst` driver

Write a paragraph in your lab logbook: “Here’s what I learned from [driver name].”

7.9.11 Quick Self-Check

Before moving to the labs, confirm:

1. **Why look at real drivers?** Answer: To see patterns in context, learn idioms, and bridge from minimal examples to production code.
2. **Which driver is simplest?** Answer: `null.c`, it’s pseudo-only with no state or resources.
3. **Which driver shows hybrid structure?** Answer: `if_tuntap.c`, it’s both a character device and a network interface.
4. **What should you skip when reading complex drivers?** Answer: Hardware-specific details (registers, DMA, interrupts) until later chapters cover them.
5. **How should you use these anchors?** Answer: As reference examples, not detailed study material. Skim, compare, note new concepts, move on.

If those are clear, you’re ready for **hands-on labs** where you’ll build, test, and extend your driver.

7.10 Hands-On Labs

You’ve read about driver structure, seen the patterns, and walked through the code. Now it’s time to **build, test, and validate** your understanding through four practical labs. Each lab is a checkpoint, ensuring you’ve mastered the concepts before moving forward.

7.10.1 Lab Overview

Lab	Focus	Duration	Key Learning
Lab 7.1	Source Hunt	20-30 min	Navigate FreeBSD source, identify patterns
Lab 7.2	Build & Load	30-40 min	Compile, load, verify lifecycle
Lab 7.3	Device Node	30-40 min	Create /dev, test open/close
Lab 7.4	Error Handling	30-45 min	Simulate failures, verify unwinding

Total time: 2-2.5 hours if you complete all labs in one session.

Prerequisites:

- FreeBSD 14.3 lab environment from Chapter 2
- /usr/src installed
- Basic shell and editor skills
- ~/drivers/myfirst project from earlier sections

Let's begin.

7.10.2 Lab 7.1: Source Code Scavenger Hunt

Objective: Build familiarity with the FreeBSD source tree by finding and identifying driver patterns.

Skills practiced:

- Navigating /usr/src/sys
- Using grep and find
- Reading real driver code

Instructions:

1. Locate the null driver:

```
% find /usr/src/sys -name "null.c" -type f
```

Expected output:

```
/usr/src/sys/dev/null/null.c
```

2. Open and scan:

```
% less /usr/src/sys/dev/null/null.c
```

3. Find the cdevsw structures:

Search for cdevsw inside less by typing /cdevsw and pressing Enter.

You should land on lines defining null_cdevsw, zero_cdevsw, and full_cdevsw.

4. Find the module event handler:

Search for modevent:

```
/modevent
```

You should see null_modevent().

5. Identify device creation:

Search for make_dev:

```
/make_dev
```

You should find three calls creating /dev/null, /dev/zero, and /dev/full.

6. Compare to your driver:

Open your myfirst.c and compare:

- How does null.c create device nodes? (Answer: make_dev_credf in the module loader)
- How does your driver create them? (Answer: make_dev_s in attach())

7. Find the LED driver:

```
% find /usr/src/sys -name "led.c" -path "*/dev/led/*"
```

8. Scan for softc:

```
% grep -n "struct ledsc" /usr/src/sys/dev/led/led.c | head -5
```

You should see the struct ledsc definition near the top of the file, right after the #include block.

9. Repeat for if_tuntap:

```
% less /usr/src/sys/net/if_tuntap.c
```

Search for tuntap_softc. Notice how much richer it is than your minimal softc.

10. Record your findings:

In your lab logbook, write:

Lab 7.1 Completed:

- Located `null.c`, `led.c`, `if_tuntap.c`
- Identified `cdevsw`, module loader, and `softc` structures
- Compared patterns to myfirst driver
- Key insight: [your observation]

Success criteria:

- ☐ Found all three driver files
- ☐ Located `cdevsw` and `softc` in each
- ☐ Identified device creation calls
- ☐ Compared to your driver

If stuck: Use `grep -r "DRIVER_MODULE" /usr/src/sys/dev/null/` to find key macros.

7.10.3 Lab 7.2: Build, Load, and Verify Lifecycle

Objective: Compile your driver, load it into the kernel, verify lifecycle events, and unload cleanly.

Skills practiced:

- Building kernel modules
- Loading/unloading with `kldload/kldunload`
- Inspecting `dmesg` and `devinfo`

Instructions:**1. Navigate to your driver:**

```
% cd ~/drivers/myfirst
```

2. Clean and build:

```
% make clean
% make
```

Verify `myfirst.ko` was created:

```
% ls -lh myfirst.ko
-rwxr-xr-x  1 youruser yourgroup  8.5K Nov  6 16:00 myfirst.ko
```

3. Load the module:

```
% sudo kldload ./myfirst.ko
```

4. Check kernel messages:

```
% dmesg | tail -n 5
```

Expected output:

```
myfirst0: <My First FreeBSD Driver> on nexus0
myfirst0: Attached successfully at tick 123456
myfirst0: Created /dev/myfirst0
myfirst0: Sysctl tree created under dev.myfirst.0.stats
```

5. Verify device tree:

```
% devinfo -v | grep myfirst
myfirst0
```

6. Check device node:

```
% ls -l /dev/myfirst0
crw----- 1 root wheel 0x5a Nov 6 16:00 /dev/myfirst0
```

7. Query sysctls:

```
% sysctl dev.myfirst.0.stats
dev.myfirst.0.stats.attach_ticks: 123456
dev.myfirst.0.stats.open_count: 0
dev.myfirst.0.stats.bytes_read: 0
```

8. Unload the module:

```
% sudo kldunload myfirst
```

9. Verify cleanup:

```
% dmesg | tail -n 2
myfirst0: Detaching: uptime 1234 ticks, opened 0 times
```

```
% ls /dev/myfirst0
ls: /dev/myfirst0: No such file or directory
```

```
% sysctl dev.myfirst.0
sysctl: unknown oid 'dev.myfirst.0'
```

10. Reload and verify idempotency:

```
% sudo kldload ./myfirst.ko
% sudo kldunload myfirst
% sudo kldload ./myfirst.ko
% sudo kldunload myfirst
```

All cycles should succeed without errors.

11. Record results:

Lab logbook:

```
Lab 7.2 Completed:
- Built myfirst.ko successfully
- Loaded without errors
- Verified attach messages in dmesg
- Verified /dev node and sysctls
- Unloaded cleanly
- Repeated load/unload cycle 3 times: all succeeded
```

Success criteria:

- ☐ Module builds without errors
- ☐ Loads without kernel panic
- ☐ Attach message appears in dmesg
- ☐ /dev/myfirst0 exists while loaded
- ☐ Sysctls readable
- ☐ Unload removes everything
- ☐ Reload works reliably

Troubleshooting:

- If build fails, check Makefile syntax (tabs, not spaces).
- If load fails with “Exec format error,” check kernel/source version match.
- If unload says “module busy,” check nothing is holding the device open.

7.10.4 Lab 7.3: Test Device Node Open/Close

Objective: Interact with `/dev/myfirst0` from user-space and verify your `open()` and `close()` handlers are called.

Skills practiced:

- User-space device access
- Monitoring driver logs
- Tracking state changes

Instructions:

1. Load the driver:

```
% sudo kldload ./myfirst.ko
```

2. Open the device with cat (read):

```
% sudo sh -c 'cat < /dev/myfirst0'
```

(No output, immediate EOF)

3. Check logs:

```
% dmesg | tail -n 3
myfirst0: Device opened (count: 1)
myfirst0: Device closed
```

4. Write to the device:

```
% sudo sh -c 'echo "test" > /dev/myfirst0'
```

5. Check logs again:

```
% dmesg | tail -n 3
myfirst0: Device opened (count: 2)
myfirst0: Device closed
```

6. Verify sysctl counter:

```
% sysctl dev.myfirst.0.stats.open_count
dev.myfirst.0.stats.open_count: 2
```

7. Test exclusive access:

Open two terminals.

Terminal 1:

```
% sudo sh -c 'exec 3<>/dev/myfirst0; sleep 10'
```

(This holds the device open for 10 seconds)

Terminal 2 (quickly, while terminal 1 is still sleeping):

```
% sudo sh -c 'cat < /dev/myfirst0'
cat: /dev/myfirst0: Device busy
```

Success! Exclusive access is enforced.

8. Try to unload while open:

Terminal 1 (keep the device open):

```
% sudo sh -c 'exec 3<>/dev/myfirst0; sleep 30'
```

Terminal 2:

```
% sudo kldunload myfirst
kldunload: can't unload file: Device busy
```

Check dmesg:

```
% dmesg | tail -n 2
myfirst0: Cannot detach: device is open
```

Perfect! Your detach() correctly refuses to unload while in use.

9. Close and retry unload:

Terminal 1: Wait for sleep 30 to finish (or Ctrl+C to interrupt).

Terminal 2:

```
% sudo kldunload myfirst
(succeeds)
```

10. Record results:

Lab logbook:

Lab 7.3 Completed:

- Opened device with cat, verified open/close logs
- Opened device with echo, verified counter increment
- Exclusive access enforced (second open returned EBUSY)
- Detach refused while device open
- Detach succeeded after close

Success criteria:

- ☐ Open triggers open() handler (logged)
- ☐ Close triggers close() handler (logged)
- ☐ Sysctl counter increments on each open
- ☐ Second open returns EBUSY
- ☐ Detach returns EBUSY while open
- ☐ Detach succeeds after close

Troubleshooting:

- If you don't see "Device opened" logs, check that device_printf() is present in your open() handler.
 - If exclusive access isn't enforced, verify the if (sc->is_open) return (EBUSY) check in open().
-

7.10.5 Lab 7.4: Simulate Attach Failure and Verify Unwinding

Objective: Inject a deliberate failure into attach() and verify cleanup is correct (no leaks, no panics).

Skills practiced:

- Testing error paths
- Debugging attach failures
- Verifying resource cleanup

Instructions:**1. Add a simulated failure:**

Edit myfirst.c and add this after mutex initialization in attach():

```
mtx_init(&sc->mtx, device_get_nameunit(dev), "myfirst", MTX_DEF);

/* SIMULATED FAILURE FOR LAB 7.4 */
device_printf(dev, "Simulating attach failure for testing\n");
error = ENXIO;
goto fail_mtx;

/* (rest of attach continues...) */
```


2. Rebuild:

```
% make clean && make
```

3. Try to load:

```
% sudo kldload ./myfirst.ko
kldload: can't load ./myfirst.ko: Device not configured
```

4. Check dmesg:

```
% dmesg | tail -n 3
myfirst0: <My First FreeBSD Driver> on nexus0
myfirst0: Simulating attach failure for testing
```

5. Verify no leaks:

```
% ls /dev/myfirst0
ls: /dev/myfirst0: No such file or directory
```

```
% sysctl dev.myfirst.0
sysctl: unknown oid 'dev.myfirst.0'
```

```
% devinfo -v | grep myfirst
(no output)
```

Perfect! The device failed to attach, and no resources were left behind.

6. Try to load again:

```
% sudo kldload ./myfirst.ko
kldload: can't load ./myfirst.ko: Device not configured
```

Still fails cleanly (no double-initialization panics).

7. Remove the simulated failure:

Edit `myfirst.c` and delete or comment out the simulated failure block.

8. Rebuild and load normally:

```
% make clean && make
% sudo kldload ./myfirst.ko
% dmesg | tail -n 5
myfirst0: <My First FreeBSD Driver> on nexus0
myfirst0: Attached successfully at tick 123456
myfirst0: Created /dev/myfirst0
myfirst0: Sysctl tree created under dev.myfirst.0.stats
```

Success!

9. Test another failure point:

Inject failure after creating the device node:

```
error = make_dev_s(&args, &sc->cdev, "myfirst%d", sc->unit);
if (error != 0) {
    device_printf(dev, "Failed to create device node: %d\n", error);
    goto fail_mtx;
}

/* SIMULATED FAILURE AFTER DEV NODE CREATION */
device_printf(dev, "Simulating failure after dev node creation\n");
error = ENOMEM;
goto fail_dev;
```

10. Rebuild and test:

```
% make clean && make
% sudo kldload ./myfirst.ko
kldload: can't load ./myfirst.ko: Cannot allocate memory
```

```
% dmesg | tail -n 3
myfirst0: <My First FreeBSD Driver> on nexus0
myfirst0: Simulating failure after dev node creation
```

11. Verify /dev node was destroyed:

```
% ls /dev/myfirst0
ls: /dev/myfirst0: No such file or directory
```

Perfect! Even though the node was created, the error path destroyed it.

12. Remove simulation and restore normal operation:

Delete the second simulated failure, rebuild, and load normally.

13. Record results:

Lab logbook:

Lab 7.4 Completed:

- Simulated failure after mutex init: cleanup correct
- Simulated failure after dev node creation: cleanup correct
- Verified no leaks in either case
- Verified no panics on repeated load attempts
- Restored normal operation

Success criteria:

- ☐ Simulated failure after mutex: no leaks
- ☐ Simulated failure after dev node: node destroyed
- ☐ Multiple load attempts don't panic
- ☐ Normal operation restored after removing simulation

Troubleshooting:

- If you see a panic, your error path has a bug. Check that every goto jumps to the correct label.
 - If resources leak, ensure each cleanup label is reachable and correct.
-

7.10.6 Labs Complete!

You've now:

- Navigated the FreeBSD source tree (Lab 7.1)
- Built, loaded, and verified your driver (Lab 7.2)
- Tested open/close and exclusive access (Lab 7.3)
- Verified error unwinding with simulated failures (Lab 7.4)

Pat yourself on the back. You've moved from reading about drivers to **building and testing one yourself**. This is a major milestone.

7.11 Short Exercises

These exercises reinforce the concepts from this chapter. They're **optional but recommended** if you want to deepen your understanding before moving forward.

7.11.1 Exercise 7.1: Add a Sysctl Flag

Task: Add a new read-only sysctl showing whether the device is currently open.

Steps:

1. In `attach()`, add:

```
SYSCTL_ADD_INT(&sc->sysctl_ctx,  
    SYSCTL_CHILDREN(sc->sysctl_tree),  
    OID_AUTO, "is_open", CTLFLAG_RD,  
    &sc->is_open, 0, "1 if device is currently open");
```

2. Rebuild, load, and test:

```
% sysctl dev.myfirst.0.stats.is_open  
dev.myfirst.0.stats.is_open: 0  
  
% sudo sh -c 'exec 3<>/dev/myfirst0; sysctl dev.myfirst.0.stats.is_open; exec 3<&-'  
dev.myfirst.0.stats.is_open: 1
```

Expected result: The flag shows 1 while open, 0 after close.

7.11.2 Exercise 7.2: Log First and Last Open

Task: Modify `open()` to log only the **first** open and `close()` to log only the **last** close.

Hints:

- Check `sc->open_count` before and after incrementing.
- In `close()`, decrement a counter and check if it reaches zero.

Expected behavior:

```
% sudo sh -c 'cat < /dev/myfirst0'  
myfirst0: Device opened for the first time  
myfirst0: Device closed (no more openers)  
  
% sudo sh -c 'cat < /dev/myfirst0'  
(no log: not the first open)  
myfirst0: Device closed (no more openers)
```

7.11.3 Exercise 7.3: Add a “Reset Stats” Sysctl

Task: Add a write-only sysctl that resets `open_count` and `bytes_read` to zero.

Steps:

1. Define a handler:

```
static int
sysctl_reset_stats(SYSCTL_HANDLER_ARGS)
{
    struct myfirst_softc *sc = arg1;
    int error, val;

    val = 0;
    error = sysctl_handle_int(oidp, &val, 0, req);
    if (error != 0 || req->newptr == NULL)
        return (error);

    mtx_lock(&sc->mtx);
    sc->open_count = 0;
    sc->bytes_read = 0;
    mtx_unlock(&sc->mtx);

    device_printf(sc->dev, "Statistics reset\n");
    return (0);
}
```

2. Register it:

```
SYSCTL_ADD_PROC(&sc->sysctl_ctx,
    SYSCTL_CHILDREN(sc->sysctl_tree),
    OID_AUTO, "reset_stats", CTLTYPE_INT | CTLFLAG_WR | CTLFLAG_MPSAFE,
    sc, 0, sysctl_reset_stats, "I",
    "Write 1 to reset statistics");
```

3. Test:

```
% sysctl dev.myfirst.0.stats.open_count
dev.myfirst.0.stats.open_count: 5

% sudo sysctl dev.myfirst.0.stats.reset_stats=1
dev.myfirst.0.stats.reset_stats: 0 -> 1

% dmesg | tail -n 1
myfirst0: Statistics reset
```

```
% sysctl dev.myfirst.0.stats.open_count
dev.myfirst.0.stats.open_count: 0
```

7.11.4 Exercise 7.4: Test Load/Unload 100 Times

Task: Write a script that loads and unloads your driver 100 times, checking for any failures or leaks.

Script (`~/drivers/myfirst/stress_test.sh`):

```
#!/bin/sh
set -e

for i in $(seq 1 100); do
    echo "Iteration $i"
    sudo kldload ./myfirst.ko
    sleep 0.1
    sudo kldunload myfirst
done

echo "Stress test completed: 100 cycles"
```

Run:

```
% chmod +x stress_test.sh
% ./stress_test.sh
```

Expected result: All iterations succeed without errors.

If it fails: Check `dmesg` for panic messages or leaked resources.

7.11.5 Exercise 7.5: Compare Your Driver to `null.c`

Task: Open `/usr/src/sys/dev/null/null.c` side-by-side with your `myfirst.c`. List 5 similarities and 5 differences.

Example observations:

Similarities:

1. Both use `cdevsw` for character device operations.

2. Both create /dev nodes.
3. Both have open and close handlers.
4. Both return EOF on read.
5. Both log attach/detach events.

Differences:

1. null.c uses MOD_LOAD handler; myfirst uses Newbus.
 2. null.c doesn't have a softc; myfirst does.
 3. null.c creates multiple devices (null, zero, full); myfirst creates one.
 4. null.c doesn't use sysctls; myfirst does.
 5. null.c is stateless; myfirst tracks counters.
-

7.12 Optional Challenges

These are **advanced exercises** for readers who want to push beyond the basics. Don't attempt these until you've completed all labs and exercises.

7.12.1 Challenge 7.1: Implement a Simple Read Buffer

Goal: Instead of returning EOF immediately, return a fixed string on read().

Steps:

1. Add a buffer to the softc:

```
char    read_buffer[64]; /* Data to return on read */
size_t  read_len;        /* Length of valid data */
```

2. In attach(), populate the buffer:

```
snprintf(sc->read_buffer, sizeof(sc->read_buffer),
         "Hello from myfirst driver!\n");
sc->read_len = strlen(sc->read_buffer);
```

3. In myfirst_read(), copy data to user-space:

```
static int
myfirst_read(struct cdev *dev, struct uio *uio, int ioflag)
{
    struct myfirst_softc *sc = dev->si_drv1;
    size_t len;
    int error;
```

```
len = MIN(uio->uio_resid, sc->read_len);
if (len == 0)
    return (0); /* EOF */

error = uiomove(sc->read_buffer, len, uio);
return (error);
}
```

4. Test:

```
% sudo cat /dev/myfirst0
Hello from myfirst driver!
```

Expected behavior: Reading returns the string once, then EOF on subsequent reads.

7.12.2 Challenge 7.2: Allow Multiple Openers

Goal: Remove the exclusive-access check so multiple programs can open the device simultaneously.

Steps:

1. Remove the `if (sc->is_open) return (EBUSY)` check in `open()`.
2. Use a **reference count** instead of a boolean flag:

```
int    open_refcount; /* Number of current openers */
```

3. In `open()`:

```
mtx_lock(&sc->mtx);
sc->open_refcount++;
mtx_unlock(&sc->mtx);
```

4. In `close()`:

```
mtx_lock(&sc->mtx);
sc->open_refcount--;
mtx_unlock(&sc->mtx);
```

5. In `detach()`, refuse if `open_refcount > 0`:


```
if (sc->open_refcount > 0) {
    device_printf(dev, "Cannot detach: device has %d openers\n",
        sc->open_refcount);
    return (EBUSY);
}
```

6. Test with two terminals:

Terminal 1:

```
% sudo sh -c 'exec 3<>/dev/myfirst0; sleep 30'
```

Terminal 2:

```
% sudo sh -c 'cat < /dev/myfirst0'
(succeeds instead of returning EBUSY)
```

7.12.3 Challenge 7.3: Add a Write Counter

Goal: Track how many bytes have been written to the device.

Steps:

1. Add to softc:

```
uint64_t    bytes_written;
```

Note: This is a discarding write handler for Chapter 7 only. We deliberately do not use `uiomove()` or store user data here. Full data movement, buffering, and `uiomove()` live in Chapter 9.

2. In `myfirst_write()`:

```
size_t len = uio->uio_resid;

mtx_lock(&sc->mtx);
sc->bytes_written += len;
mtx_unlock(&sc->mtx);

uio->uio_resid = 0;
return (0);
```

3. Expose via `sysctl`:

```
SYSCALL_ADD_U64(&sc->sysctl_ctx, ..., "bytes_written", ...);
```

4. Test:

```
% sudo sh -c 'echo "test" > /dev/myfirst0'
% sysctl dev.myfirst.0.stats.bytes_written
dev.myfirst.0.stats.bytes_written: 5
```

7.12.4 Challenge 7.4: Create a Second Device (myfirst1)

Goal: Manually create a second device instance to test multi-device support.

Hint: Right now, your driver creates `myfirst0` automatically. To test multiple devices, you'd need to trigger a second probe/attach cycle. This is complex (requires bus-level manipulation or cloning), so consider it **research only**.

Alternate approach: Study `/usr/src/sys/net/if_tuntap.c` to see how it handles clone devices (creating new instances on demand).

7.12.5 Challenge 7.5: Implement Rate-Limited Logging

Goal: Add a rate-limited log for open events (log at most once per second).

Steps:

1. Add to `softc`:

```
time_t last_open_log;
```

2. In `open()`:

```
time_t now = time_second;

if (now - sc->last_open_log >= 1) {
    device_printf(sc->dev, "Device opened (count: %lu)\n",
        (unsigned long)sc->open_count);
    sc->last_open_log = now;
}
```

3. Test by opening rapidly:

```
% for i in $(seq 1 10); do sudo sh -c 'cat < /dev/myfirst0'; done
```

Expected behavior: Only a few log messages appear (rate-limited).

7.13 Pitfalls & Troubleshooting Decision Tree

Even with careful coding, you'll encounter issues. This section provides a **decision tree** to diagnose common problems quickly.

7.13.1 Symptom: Driver Won't Load

Check:

- ☐ Does `freebsd-version -k` match `/usr/src` version?
 - **No:** Rebuild kernel or re-clone `/usr/src` for correct version.
 - **Yes:** Continue.
- ☐ Does `make` complete without errors?
 - **No:** Read compiler error messages. Common causes:
 - * Missing semicolons
 - * Mismatched braces
 - * Undefined functions (missing includes)
 - **Yes:** Continue.
- ☐ Does `kldload` fail with “Exec format error”?
 - **Yes:** Kernel/module mismatch. Rebuild with matching sources.
 - **No:** Continue.
- ☐ Does `kldload` fail with “No such file or directory”?
 - **Yes:** Check module path (`./myfirst.ko` vs `/boot/modules/myfirst.ko`).
 - **No:** Continue.
- ☐ Check `dmesg` for attach errors:

```
% dmesg | tail -n 10
```

Look for error messages from your driver.

7.13.2 Symptom: Device Node Doesn't Appear

Check:

- ☐ Did `attach()` succeed?

```
% dmesg | grep myfirst
```

Look for “Attached successfully” message.

- ☐ If attach failed, did error handling run?

Look for error messages in `dmesg`.

- ☐ Did `make_dev_s()` succeed?

Add logging:

```
device_printf(dev, "make_dev_s returned: %d\n", error);
```

- ☐ Is the device node's name correct?

```
% ls -l /dev/myfirst*
```

Check spelling and unit number.

7.13.3 Symptom: Kernel Panic on Load

Check:

- ☐ Did you forget `mtx_init()`?

Panic during `mtx_lock()` → forgot to initialize mutex.

- ☐ Did you dereference a NULL pointer?

Panic with “NULL pointer dereference” → check `device_get_softc()` return value.

- ☐ Did you corrupt memory?

Enable WITNESS and INVARIANTS in your kernel config, then rebuild:

```
options WITNESS
options INVARIANTS
```

Reboot, reload your driver. WITNESS will catch lock violations.

7.13.4 Symptom: Kernel Panic on Unload

Check:

- ☐ Did you forget `destroy_dev()`?

Unload panics when user tries to access the device node.

- ☐ Did you forget `mtx_destroy()`?

WITNESS-enabled kernels will panic on unload if locks aren't destroyed.

- ☐ Did you forget `sysctl_ctx_free()`?

Sysctl OID leaks can cause panics on reload.

- ☐ Is code still running when you unload?

Check for: - Open device nodes (`sc->is_open` should be false) - Active timers or callbacks (not used in this chapter, but common later)

7.13.5 Symptom: “Device Busy” on Unload

Check:

- ☐ Is the device still open?

```
% fstat | grep myfirst
```

If a process has the device open, unload will fail.

- ☐ Did you return `EBUSY` from `detach()`?

Check `detach()` logic:

```
if (sc->is_open) {  
    return (EBUSY);  
}
```

7.13.6 Symptom: Sysctls Don't Appear

Check:

- ☐ Did `sysctl_ctx_init()` run?
- ☐ Did `SYSCTL_ADD_NODE()` succeed?

Add logging:

```
if (sc->sysctl_tree == NULL) {  
    device_printf(dev, "sysctl tree creation failed\n");  
}
```

- ☐ Is the sysctl path correct?

```
% sysctl dev.myfirst
```

Check for typos in driver name or unit number.

7.13.7 Symptom: Open/Close Not Logging

Check:

- ☐ Did you add `device_printf()` to the handlers?
- ☐ Is the `cdevsw` registered correctly?

Check that `args.mda_devsw = &myfirst_cdevsw` is set before `make_dev_s()`.

- ☐ Is `si_drv1` set correctly?

```
args.mda_si_drv1 = sc;
```

If this is `NULL`, `open()` will fail.

7.13.8 Symptom: Module Loads But Does Nothing

Check:

- ☐ Did `attach()` run?

```
% dmesg | grep myfirst
```

If you see no output at all and your driver attaches to nexus, the most likely cause is a missing `identify` method. Without it, nexus has no `myfirst` device to probe and your code is never called. Re-read the **Step 6: Implement Identify** section above and confirm that `DEVMETHOD(device_identify, myfirst_identify)` is present in your method table.

- ☐ Is the device on the correct bus?

For pseudo-devices, use `nexus` (and remember to provide `identify`). For PCI, use `pci`. For USB, use `usb`.

- ☐ Did `probe()` return a non-error priority?

If `probe()` returns `ENXIO`, the driver will not attach. For our pseudo-device, `BUS_PROBE_DEFAULT` is the right value.

7.13.9 General Debugging Tips

Enable verbose boot:

```
% sudo sysctl boot.verbose=1
```

Reload your driver to see more detailed messages.

Use printf debugging:

Add `device_printf()` statements at key points to trace execution flow.

Check lock state:

If using `WITNESS`, check lock order:

```
% sysctl debug.witness.fullgraph
```

Save dmesg after panic:

```
% sudo dmesg -a > panic.log
```

Analyze the log for clues.

7.14 Self-Assessment Rubric

Use this rubric to evaluate your understanding before moving to Chapter 8.

7.14.1 Core Knowledge

Rate yourself (1-5, where 5 = fully confident):

- ☐ I can explain what the softc is and why it exists. (Score: __/5)
- ☐ I understand probe/attach/detach lifecycle. (Score: __/5)
- ☐ I can create a device node using `make_dev_s()`. (Score: __/5)
- ☐ I can implement basic open/close handlers. (Score: __/5)
- ☐ I can add a read-only sysctl. (Score: __/5)
- ☐ I understand the single-label unwind pattern. (Score: __/5)
- ☐ I can test error paths with simulated failures. (Score: __/5)

****Total Score: __/35****

Interpretation:

- **30-35:** Excellent. You're ready for Chapter 8.
 - **25-29:** Good. Review weak areas before proceeding.
 - **20-24:** Adequate. Revisit labs and exercises.
 - **<20:** Spend more time on this chapter.
-

7.14.2 Practical Skills

Can you do these without looking at notes?

- ☐ Build a kernel module with `make`.
- ☐ Load a module with `kldload`.
- ☐ Check attach messages in `dmesg`.
- ☐ Query sysctls with `sysctl`.
- ☐ Open a device node with `cat` or shell redirection.
- ☐ Unload a module with `kldunload`.
- ☐ Simulate an attach failure.
- ☐ Verify cleanup after failure.

Scoring: 1 point per skill. **Target:** 7/8 or higher.

7.14.3 Code Reading

Can you recognize these patterns in real FreeBSD code?

- ☐ Identify a `cdevsw` structure.
- ☐ Locate `probe()`, `attach()`, `detach()` methods.
- ☐ Find the `softc` structure definition.
- ☐ Identify `DRIVER_MODULE()` macro.

- ☐ Spot error unwinding with goto labels.
- ☐ Find `make_dev()` or `make_dev_s()` calls.
- ☐ Identify `sysctl` creation (`SYSCTL_ADD_*`).

Scoring: 1 point per pattern. **Target:** 6/7 or higher.

7.14.4 Conceptual Understanding

True or False:

1. The `softc` is allocated by the driver in `attach()`. (**False.** Newbus allocates it from the size declared in `driver_t`.)
2. `probe()` should allocate resources. (**False.** `probe()` only inspects the device and decides whether to claim it; `attach()` does the allocation.)
3. `detach()` must undo everything `attach()` did. (**True.**)
4. Error labels should be in reverse order of initialization. (**True.**)
5. You can skip `mtx_destroy()` if the module is unloading. (**False.** Every `mtx_init()` needs a matching `mtx_destroy()`.)
6. `Sysctls` are automatically cleaned up when the module unloads. (**False.** Only if you call `sysctl_ctx_free()` on the per-device context you initialized.)
7. `make_dev_s()` is safer than `make_dev()`. (**True.** It returns an explicit error and avoids the race where `make_dev()` could fail without a clear way to report it.)
8. A pseudo-device that attaches to `nexus` must provide an `identify` method. (**True.** Without it the bus has no device to probe.)

Scoring: 1 point per correct answer. **Target:** 7/8 or higher.

7.14.5 Overall Assessment

Add your scores:

- Core Knowledge: __/35
- Practical Skills: __/8
- Code Reading: __/7
- Conceptual Understanding: __/8

****Total: __/58****

Grade:

- **51-58:** A (Excellent mastery)
- **44-50:** B (Good understanding)
- **35-43:** C (Adequate, but review weak areas)

- **<35:** Revisit chapter before proceeding

7.15 Wrapping Up & Pointers Forward

Congratulations! You’ve completed Chapter 7 and built your first FreeBSD driver from the ground up. Let’s reflect on what you’ve accomplished and preview what’s ahead.

7.15.1 What You Built

Your `myfirst` driver is minimal but complete:

- **Lifecycle discipline:** Clean probe/attach/detach with no leaks.
- **User surface:** `/dev/myfirst0` node that opens and closes reliably.
- **Observability:** Read-only sysctls showing attach time, open count, and bytes read.
- **Error handling:** Single-label unwind pattern that recovers gracefully from failures.
- **Logging:** Proper use of `device_printf()` for lifecycle events and errors.

This isn’t a toy. It’s a **production-quality scaffold**, the same foundation every FreeBSD driver starts with.

7.15.2 What You Learned

You now understand:

- How Newbus discovers and attaches drivers
- The role of the `softc` (per-device state)
- How to create and destroy device nodes
- How to expose metrics via sysctls
- How to handle errors without leaking resources
- How to test lifecycle paths (load/unload, open/close, simulated failures)

These skills are **transferable**. Whether you write a PCI driver, a USB driver, or a network interface, you’ll use these same patterns.

7.15.3 What’s Still Missing (And Why)

Your driver doesn’t do much yet:

- **Read/write semantics:** Stubbed, not implemented. (**Chapter 8 and 9**)
- **Buffering:** No queues, no ring buffers. (**Chapter 10**)
- **Hardware interaction:** No registers, no DMA, no interrupts. (**Part 4**)
- **Concurrency:** Mutex present but not exercised. (**Part 3**)
- **Real-world I/O:** No blocking, no poll/select. (**Chapter 10**)

This was intentional. **Master structure before complexity**. You wouldn’t learn carpentry by building a skyscraper on day one. You’d start with a workbench, just like you did here.

7.15.4 What's Next

Chapter 8, Working with Device Files. devfs permissions and ownership, persistent nodes, and the userland probes you will use to inspect and test your device.

Chapters 9 and 10, Reading and Writing to Devices, plus Handling Input and Output Efficiently. Implement read and write with `uiomove`, introduce buffering and flow control, and define blocking, non-blocking, and `poll` or `kqueue` semantics with proper error handling.

7.15.5 Your Next Steps

Before moving to Chapter 8:

1. **Complete all labs** in this chapter if you haven't already.
2. **Attempt at least two exercises** to reinforce patterns.
3. **Test your driver thoroughly:** Load/unload 10 times, open/close 10 times, simulate one more failure.
4. **Commit your code to Git:** This is a milestone.

```
% cd ~/drivers/myfirst
% git add myfirst.c Makefile
% git commit -m "Chapter 7 complete: minimal driver with lifecycle discipline"
```

5. **Take a break.** You've earned it. Kernel programming is intense, and consolidation time matters.

When you're ready for Chapter 8:

- You'll extend this same driver (no restart from scratch).
- The structure you built here will carry forward.
- Concepts will build incrementally, not reset.

7.15.6 A Final Word

Building a driver from scratch can feel overwhelming at first. But look at what you've accomplished:

- You started with nothing but a Makefile and a blank `.c` file.
- You built a driver that compiles, loads, attaches, operates, detaches, and unloads cleanly.
- You tested error paths and verified cleanup.
- You exposed state via `sysctls` and created a user-accessible device node.

This is not beginner's luck. This is competence. Most general-purpose software developers never touch a kernel module, let alone build one with disciplined attach and detach paths. You just did.

The journey from “hello module” to “production driver” is long, but you've taken the hardest step: **starting**. Every chapter from here adds one more layer of capability, one more tool in your kit.

Keep your lab logbook updated. Keep experimenting. Keep asking “why?” when something doesn't make sense. And most importantly, **keep building**.

Welcome to the world of FreeBSD driver development. You've earned your place here.

7.15.7 See You in Chapter 8

In the next chapter, we'll breathe life into your device node by implementing real file semantics: managing per-open state, handling exclusive vs shared access, and preparing the ground for actual I/O.

Until then, enjoy your success. You've built something real, and that's worth celebrating.

"The expert at anything was once a beginner." - Helen Hayes

8 Working with Device Files

8.1 Reader Guidance & Outcomes

In Chapter 7 you built `myfirst`, a real FreeBSD driver that attaches cleanly, creates `/dev/myfirst0`, opens and closes that node, and unloads without leaks. That was the first win, and it was a real one. You now have a working driver skeleton on disk, a `.ko` file that the kernel will accept and release at your command, and a `/dev` entry that user programs can reach.

This chapter zooms in on the piece of that work that is easiest to take for granted: the **device file itself**. The line of code that created `/dev/myfirst0` in Chapter 7 was compact, but it sits on top of a subsystem called **devfs**, and that subsystem is the bridge between everything your driver does inside the kernel and every tool or program a user ever points at it. Understanding that bridge well now will make Chapters 9 and 10, where real data starts to flow, much less mysterious.

8.1.1 Why This Chapter Earns Its Own Place

Chapter 6 introduced the device-file model at the level of mental pictures, and Chapter 7 used enough of it to get a driver running. Neither chapter stopped to examine the surface itself. That is not an oversight. In a book that teaches driver writing from first principles, the device file is worth a dedicated chapter because the mistakes made on that surface are also the mistakes hardest to undo later.

Consider what the surface has to carry. It carries identity (a path a user program can predict). It carries access policy (who is allowed to open, read, or write). It carries multiplexing (one driver, many instances, many simultaneous openers). It carries lifecycle (when the node appears, when it disappears, and what happens when a user program is mid-call as it goes). It carries compatibility (legacy names alongside modern ones). It carries observability (what operators can see and change from userland). A driver that gets the internals right but the surface wrong will be a driver that operators refuse to deploy, a driver that security reviewers flag, a driver that breaks subtly inside jails, a driver that deadlocks on unload under realistic load.

Chapter 7 gave you just enough of that surface to prove the path worked. This chapter gives you enough to design it on purpose.

8.1.2 Where Chapter 7 Left the Driver

It is worth a short checkpoint on the state of `myfirst` before we extend it. Your Chapter 7 driver ends up with all of the following in place:

- A `device_identify`, `device_probe`, and `device_attach` path that creates exactly one Newbus child of `nexus0` named `myfirst0`.
- A softc allocated by Newbus, reachable through `device_get_softc(dev)`.

- A mutex, a sysctl tree under `dev.myfirst.0.stats`, and three read-only counters.
- A struct `cdevsw` populated with `d_open`, `d_close`, and stubs for `d_read` and `d_write`.
- A `/dev/myfirst0` node created with `make_dev_s(9)` in `attach` and removed with `destroy_dev(9)` in `detach`.
- A single-label error unwind that leaves the kernel in a consistent state if any `attach` step fails.
- An exclusive-open policy that refuses a second `open(2)` with `EBUSY`.

Chapter 8 treats that driver as the starting point and grows it along three axes: **shape** (what the node is called and how it is grouped), **policy** (who is allowed to use it and how that policy stays in place across reboots), and **per-descriptor state** (how two simultaneous openers can own separate book keeping).

8.1.3 What You Will Learn

By the end of this chapter you will be able to:

- Explain what a device file is in FreeBSD, and why `/dev` is not an ordinary directory.
- Describe how struct `cdev`, the devfs `vnode`, and a user file descriptor relate to each other.
- Choose sensible ownership, group, and permission values for a new device node.
- Give a device node a structured name (including a subdirectory under `/dev`).
- Create an alias so a single `cdev` is reachable under more than one path.
- Attach per-open state with `devfs_set_cdevpriv()` and clean it up safely when the file descriptor closes.
- Adjust device-node permissions persistently from userland with `devfs.conf` and `devfs.rules`.
- Exercise your driver from a small userland C program, not just with `cat` and `echo`.

8.1.4 What You Will Build

You will extend the Chapter 7 `myfirst` driver in three small steps:

1. **Stage 0: tidier permissions and a structured name.** The node moves from `/dev/myfirst0` to `/dev/myfirst/0` with a group-accessible variant for lab use.
2. **Stage 1: a user-visible alias.** You add `/dev/myfirst` as an alias for `/dev/myfirst/0` so legacy paths keep working.
3. **Stage 2: per-open state.** Each `open(2)` gets its own small counter with `devfs_set_cdevpriv()`, and you verify from userland that two simultaneous opens see independent values.

You will also write a short userland program, `probe_myfirst.c`, that opens the device, reads a bit, reports what it saw, and closes cleanly. This program will come back in Chapter 9 when real `read(2)` and `write(2)` paths are implemented.

8.1.5 What This Chapter Does Not Cover

Several topics touch `/dev` but are deliberately postponed:

- **Full read and write semantics.** Chapter 7 stubbed these; Chapter 9 implements them properly with `uiomove(9)`. Here we only prepare the ground.
- **Cloning devices** (`clone_create`, the `dev_clone` event handler). These are worth a careful look later, once the basic model is solid.

- **ioctl(2) design.** Inspecting and changing device state through `ioctl` is a topic of its own and belongs later in the book.
- **GEOM and storage devices.** GEOM builds on top of `cdevs` but adds a whole stack of its own. That belongs in Part 6.
- **Network interface nodes and ifnet.** Network drivers do not live under `/dev`. They show up through a different surface, which we will meet in Part 6.

Keeping scope tight here is the point. The surface of a device is small; the discipline around it should be the first thing you master.

8.1.6 Estimated Time Investment

- **Reading only:** about 30 minutes.
- **Reading plus the code changes to myfirst:** around 90 minutes.
- **Reading plus all four labs:** two to three hours, including rebuild cycles and userland testing.

A steady session with breaks works best. The chapter is shorter than Chapter 7, but the ideas here show up in almost every driver you will ever read.

8.1.7 Prerequisites

- A working Chapter 7 `myfirst` driver that loads, attaches, and unloads cleanly.
- FreeBSD 14.3 in your lab with matching `/usr/src`.
- Basic comfort reading `/usr/src` paths such as `/usr/src/sys/dev/null/null.c`.

8.1.8 How to Get the Most Out of This Chapter

Open your Chapter 7 source next to this chapter and edit the same file. You are not starting a new project; you are growing the one you already have. When the chapter asks you to inspect a FreeBSD file, really open it in `less` and scroll around. The device-file model clicks much faster when you have seen a couple of real drivers shape their nodes.

A practical habit that pays off immediately: as you read, keep a second terminal open against a freshly booted lab system and confirm every claim about existing nodes with `ls -l` or `stat(1)`. Typing `ls -l /dev/null` and watching the output match the prose is tiny, but it anchors the abstraction in something you can see. By the time the chapter reaches the labs, you will have a quiet reflex for verifying each claim against the running kernel rather than trusting the text alone.

A second habit: when the chapter names a source file under `/usr/src`, open it side by side with the section. Real FreeBSD drivers are the textbook; this book is only the reading guide. The material inside `/usr/src/sys/dev/null/null.c` and `/usr/src/sys/dev/led/led.c` is short enough to scan in a few minutes each, and each one is shaped by the very decisions this chapter is about to explain. A short tour there is worth more than any amount of prose here.

8.1.9 Roadmap Through the Chapter

If you want a picture of the chapter as one continuous thread, here it is. The sections in order are:

1. What a device file actually is, in theory and in `ls -l` practice.
2. How `devfs`, the filesystem behind `/dev`, came to be and what it does for you.
3. The three kernel objects that line up behind a device file.
4. How ownership, group, and mode shape what `ls -l` shows and who can open the node.
5. How names get chosen, including unit numbers and subdirectories.
6. How one `cdev` can answer to several names through aliases.
7. How per-open state is registered, retrieved, and cleaned up.
8. How the destructor really works once `destroy_dev(9)` is called.
9. How `devfs.conf` and `devfs.rules` shape policy from userland.
10. How to drive the device from small userland programs you can write yourself.
11. How real FreeBSD drivers solve these same problems.
12. Which `errno` values your `d_open` should return, and when.
13. Which tools to reach for when something on this surface looks wrong.
14. Four to eight labs that walk you through every pattern hands-on.
15. Challenges that stretch the patterns into realistic scenarios.
16. A field guide to the pitfalls that waste time if you hit them cold.

Follow the chapter end to end if this is your first time. If you are revising, you can approach each section on its own; the structure is designed to read as a complete survey, not only as a linear tutorial.

8.2 What a Device File Actually Is

Long before FreeBSD existed, UNIX committed to a famous idea: **treat devices as files**. A serial line, a disk, a terminal, a stream of random bytes, each of these could be opened, read, written, and closed with the same handful of system calls. User programs did not need to know whether the bytes they consumed came from a spinning disk, a memory buffer, or an imaginary source like `/dev/null`.

That idea was not marketing. It was a design choice Ken Thompson and Dennis Ritchie made in the first versions of UNIX in the early 1970s, and it shaped the entire operating system that followed. By presenting every device through the same system-call interface as every regular file, they turned every command-line tool that dealt with files into a tool that could also deal with devices. `cat` could copy bytes out of a serial port. `dd` could read from a tape drive. `cp` could stream a stream. That alignment is still the single most useful property of a UNIX-like system, and FreeBSD inherits it in full.

FreeBSD keeps that spirit. From user space, a device file looks like any other entry in the filesystem. It has a path, an owner, a group, a mode, and a type visible to `ls -l`. You can `open(2)` it, pass the returned file descriptor to `read(2)` and `write(2)`, query it with `stat(2)`, and close it when you are done.

8.2.1 A Short History, in Pieces Worth Knowing

The device-file model you will be writing into has gone through a few serious revisions since 1971, and each revision was driven by a real limitation of the previous one. Knowing the broad strokes saves confusion later when you read old books or old manual pages.

In **V7 UNIX** and the BSDs that followed for two decades, the entries under `/dev` were real entries on a real on-disk filesystem. An administrator used `mknod(8)` to create them, passing a device type (character or block) and a pair of small integers called the *major number* and the *minor number*. The kernel used the major number to pick a row in a table of drivers (the `cdevsw` or `bdevsw` array, depending on type), and the minor number to pick which instance of that driver the call was going to. The pair was typed into `/dev` once, by hand or by a shell script called `MAKEDEV`, and then lived on the disk forever.

That model worked for a long time. It stopped working when two things happened at once: hardware started being hot-pluggable, and the space of major numbers got crowded enough that every kernel change required coordination across the tree. A disk plugged in at runtime needed a node that had not existed before boot. A driver that moved from one tree location to another needed its numbers renegotiated. Static `/dev` entries misrepresented both.

In **FreeBSD 5**, released in 2003, the answer was **devfs**, a kernel-managed virtual filesystem that replaces the on-disk `/dev` entirely. When a driver creates a device node through `make_dev(9)`, `devfs` adds an entry to its live tree and user programs can see it immediately. When the driver calls `destroy_dev(9)`, `devfs` removes the entry. Major and minor numbers still exist inside the kernel as an implementation detail, but they are not part of the contract any more. Paths and `cdev` pointers are. That is the model you are writing into today.

A third shift worth naming: **block devices have retreated from userland visibility**. Older UNIX variants exposed certain storage devices as block-special files whose type letter in `ls -l` was `b`. FreeBSD has not shipped block-special device nodes to userland for many years. Storage drivers still exist in the kernel; they just publish themselves through `GEOM` and show up in `/dev` as character devices. The only time you will see a `b` in FreeBSD is on disks mounted from other filesystem types. Your drivers will expose character devices, full stop.

8.2.2 Why the File Abstraction Earned Its Keep

The payoff of the “everything is a file” idea is that every tool in the base system is, without special knowledge, already a tool for talking to your device. That is worth a minute of reflection because it sets the tone for how you should design your driver.

`cat` reads from files. It will also read from `/dev/myfirst/0` once your driver implements `d_read`, with no special build and no awareness that it is talking to a driver. `dd` reads and writes files in arbitrarily sized blocks; it will happily stream to or from a character device, and it offers flags (`bs=`, `count=`, `iflag=nonblock`) that let operators exercise driver behavior without writing new programs. `tee`, `head`, `tail` in follow mode, `od`, `hexdump`, `strings`, all of them already work on your device the day you ship it. Shell redirection works. Pipes work. The kernel’s file-descriptor machinery does not care which side of a pipe is a device and which side is a regular file.

The design guidance that flows from this is simple and strict: **your driver should behave like a file whenever it can**. That means returning `read(2)` lengths that match reality, returning end-of-file as zero bytes read, returning `write(2)` results that count the bytes actually consumed, respecting `errno` conventions when something goes wrong, and not inventing new meanings for existing system calls unless the meaning is unavoidable. The more your device looks like an ordinary file to every tool in `/bin` and `/usr/bin`, the less your users have to learn, and the less brittle your interface becomes when new tools appear years from now.

8.2.3 What a Device File Is Not

The abstraction is generous but it is not unlimited. It is worth naming a few things a device file is explicitly not, so you do not design around the wrong mental model.

A device file is **not an IPC channel** in general. It can behave like one, the same way a named pipe can behave like one, but the classic tools for interprocess communication are `pipe(2)`, `socketpair(2)`, UNIX-domain sockets, and `kqueue(9)`. If two user programs want to exchange messages with each other, they should use those tools rather than opening a device node as a side channel. A driver that lets itself be pressed into service as an ad-hoc IPC bus will find its semantics getting more and more complicated as users invent new uses for it.

A device file is **not a registry of settings** your driver keeps across reboots. `devfs` does not persist anything. Anything you write into `/dev/yournode` is processed by your driver at the moment it is written and is gone unless your driver chose to remember it. If you need persistent configuration, the right tools are `sysctl(8)` tunables, loader environment variables set via `loader.conf(5)`, and configuration files the userland part of your stack reads from `/etc`.

A device file is **not a broadcast medium** by default. If a driver wants to deliver the same byte stream to every open file descriptor, it must implement that explicitly; the kernel does not fan writes across readers automatically, and it does not duplicate reads into multiple files. Chapter 10's discussion of `poll(2)` and `kqueue(9)` touches the edge of this, and several drivers in the tree (for example `/usr/src/sys/net/bpf.c`) solve it on purpose. It is not free.

A device file is **not a substitute for a system call**. If your driver needs a structured, typed, versioned control interface that user programs invoke with named commands, that is what `ioctl(2)` is for. We do not design `ioctl` in this chapter, but the distinction matters today: do not smuggle control commands through `write(2)` strings when `ioctl(2)` would express them more precisely. Chapter 25 revisits `ioctl` design as part of advanced driver practice.

Holding those limits in mind now will keep your design honest later. The device-file surface is a small number of well-shaped primitives. The art is choosing which of them your driver exposes, and wiring each of them carefully.

8.2.4 The Variety You Will See Under `/dev`

A short tour of a freshly booted FreeBSD 14.3 system is a good way to calibrate what this chapter is shaping. Open a terminal and run `ls /dev` on your lab machine. You will see a cross-section of the naming patterns the book will teach you to recognise:

- **Singletons:** `null`, `zero`, `random`, `urandom`, `full`, `klog`, `mem`, `kmem`.
- **Numbered instances:** `bpf0`, `bpf1`, `md0`, `ttvu0`, `ttvv0`, `ttvv1`, `cuaU0`.
- **Subdirectories per driver:** `led/`, `pts/`, `fd/`, `input/`.
- **Standard names with special meaning:** `stdin`, `stdout`, `stderr`, `console`, `tty`.
- **Aliases for conventional paths:** `log`, `sndstat`, `midistat`.

Every entry in that tour is a `cdev` that some driver or subsystem asked `devfs` to present. Some were created by drivers that loaded at boot, some by drivers that were compiled into the kernel itself, and some by the `devfs` mount's own setup. Each of them was shaped by the same decisions you are about to make for `myfirst`.

Take a quick look at the nodes already present on your lab system:

```
% ls -l /dev/null /dev/zero /dev/random
crw-rw-rw-  1 root  wheel      0x17 Apr 17 09:14 /dev/null
crw-r--r--  1 root  wheel      0x14 Apr 17 09:14 /dev/random
crw-rw-rw-  1 root  wheel      0x18 Apr 17 09:14 /dev/zero
```

The leading **c** in each mode field tells you these are **character devices**. On FreeBSD there are no block-device nodes to worry about in this context; storage is served through character devices fronted by GEOM, and the old `block special` distinction is no longer exposed to userland the way it was in older UNIXes. For the work you are doing now, and for most drivers you will write, **character device** is the shape your node will take.

The interesting part of `ls -l` is what is *not* there. There is no backing file on disk that holds the bytes of `/dev/null`. There is no regular file hiding behind `/dev/random` somewhere under `/var`. These nodes are presented by the kernel on demand, and the permissions and ownership you see are whatever the kernel has chosen to advertise. That is the key mental shift for this chapter: in FreeBSD, the entries under `/dev` are not ordinary files that your driver pokes at. They are a **view** of kernel-side objects called `struct cdev`, served by a dedicated filesystem.

8.2.5 Character Devices Are the Common Case

A character device delivers a stream of bytes through `read(2)` and accepts a stream of bytes through `write(2)`. It may or may not support seeking. It may support `ioctl(2)`, `mmap(2)`, `poll(2)`, and `kqueue(9)`, and each of those is opt-in on the driver side. The driver declares which operations it supports by filling in the relevant fields of a `struct cdevsw`, exactly as you saw in Chapter 7.

`myfirst` is a character device. So are `/dev/null`, `/dev/zero`, the terminal nodes under `/dev/tty*`, the BPF packet interfaces under `/dev/bpf*`, and many others. When you are new to driver work, “character device” is almost always the right answer.

If you are coming from a background where block-special nodes were common, the mental adjustment is small. On FreeBSD you never write a driver that exposes a block-special node directly; storage drivers produce character devices that GEOM consumes, and GEOM in turn republishes its own character devices upward. For this chapter and the next several, the phrase “device file” always means a character device.

8.2.6 Reading `ls -l` for a Device Node

It is worth spending a minute on the shape of the output, because every line you will ever inspect with `ls -l` follows the same template.

```
% ls -l /dev/null
crw-rw-rw-  1 root  wheel      0x17 Apr 17 09:14 /dev/null
```

The **c** in the leading position says this is a character device. A regular file would show `-`, a directory would show `d`, a symbolic link would show `l`. The nine permission characters after it read exactly the same way as for

a regular file: three triples of owner, group, and other permissions, in the order read, write, execute. Devices ignore the execute bit, so you will almost never see it set.

The owner is `root` and the group is `wheel`. Those are printed from the values the kernel advertises, which is a mix of whatever the driver asked for and whatever `devfs.conf` or `devfs.rules` applied on top. Change any of them and this column changes immediately; there is no file on disk to rewrite.

The field that looks odd is `0x17`. On a regular file this column holds the size in bytes. On a device file, `devfs` reports a small hexadecimal identifier instead. It is not a major or minor number in the old System V sense, and you will not normally need to interpret it. If you want to confirm two names point at the same underlying `cdev` (for instance a primary node and an alias), `stat -f '%d %i' path` is a more reliable way to compare. We will come back to it in the userland section.

Finally, `ls -l` on a directory under `/dev` works exactly as you would expect, because `devfs` really is a filesystem. `ls -l /dev/myfirst` will list the files inside the `myfirst/` subdirectory if the driver placed one there.

8.3 devfs: Where /dev Comes From

On an old-school UNIX system, `/dev` was a real directory on a real disk, and device nodes were created by a command called `mknod(8)`. If you needed a new node, you ran `mknod` with a type, a major number, and a minor number, and an entry appeared on disk. It was static. It did not care whether the hardware was present. It did not clean up after itself.

FreeBSD moved away from that model. On a modern FreeBSD system, `/dev` is a **devfs** mount, a virtual filesystem maintained entirely by the kernel. You can see it in the output of `mount(8)`:

```
% mount | grep devfs
devfs on /dev (devfs)
```

Inside a jail, you will usually see a second `devfs` mounted at the jail's own `/dev`. The same filesystem type, the same mechanism, just a different view filtered by `devfs.rules`.

The rules of `devfs` are simple, and they are worth internalizing:

1. You do not create files under `/dev` with `touch` or `mknod`. You create them from inside the kernel, by calling `make_dev_s(9)` or one of its relatives.
2. When your driver's `cdev` goes away, the corresponding entry disappears from `/dev` automatically.
3. Ownership, group, and mode on a node start out with whatever your driver asked for, and can be adjusted from userland through `devfs.conf` and `devfs.rules`.
4. Nothing about a `devfs` node is persisted between reboots. It is always a live reflection of the current state of the kernel.

That last point is the one that catches people by surprise the first time. You cannot `chmod /dev/myfirst0` and expect the change to survive the next reboot. If you need a permission to stick, you encode it either in the driver or in one of the `devfs` configuration files. We will do both in this chapter.

If you want to look at `devfs` directly, its implementation lives in `/usr/src/sys/fs/devfs/`. You do not need to read it cover to cover, but being aware of where it is will save you confusion later when someone asks why `/dev/foo` looks the way it does.

8.3.1 Why This Model Replaced `mknod`

The move away from static device nodes in on-disk `/dev` was not a stylistic choice. It was driven by three real problems:

1. **Nodes for hardware that was not present.** Before `devfs`, `/dev` carried entries for every device the system *could* have, whether the kernel currently had a driver for it or not. Users were left guessing which paths were live.
2. **Stale nodes for hardware that had been removed.** Hot-pluggable hardware (USB, FireWire, CardBus back in its day) made static `/dev` trees actively misleading.
3. **Major and minor number exhaustion.** The pair (`major`, `minor`) was a finite resource and a source of allocation politics across the kernel tree. `devfs` sidesteps that problem entirely.

Today `/dev` is a running mirror of what the kernel actually supports and what devices are currently present. A driver that loads makes a node; a driver that unloads retracts it. A disk that is removed vanishes. That is by design, and it is why the word “node” is a better mental model than “file” for the things you see under `/dev`.

8.3.2 What `devfs` Is and Is Not

`devfs` **is** a filesystem. You can `ls` inside it, change into subdirectories, redirect into nodes, and so on. What `devfs` **is not** is a general place to keep user data. Do not try to echo a log file into `/dev`. Do not expect touch to work inside it. `devfs` accepts a small, well-defined set of operations, and anything outside that set returns an error.

This narrow surface is a feature. It means `devfs` never surprises you by persisting state, never competes with a regular filesystem for space, and never misinterprets a regular file as a device. Your driver creates nodes; `devfs` presents them; everything else goes through your `cdevsw` handlers.

8.3.3 `devfs` in the Family of Synthetic Filesystems

FreeBSD has a small family of filesystems that do not store files on any disk. `devfs` is one of them. Others you may have met are `tmpfs(5)`, which serves files from RAM, `nullfs(5)`, which republishes another directory under a new name, and `fdescfs(5)`, which presents each process’s file descriptors as files under `/dev/fd`. They are all real filesystems in the eyes of `mount(8)` and the VFS layer, but each one synthesises its contents from something other than a block device.

Knowing the family helps for two reasons. The first is that `devfs` shares a few ideas with its relatives. All synthetic filesystems lay their tree on demand, they all manage their storage outside any on-disk container, and they all have strong opinions about which operations do and do not make sense inside them. The second reason is that you will see them combined in practice. A jail typically mounts its own `devfs` under its own `/dev`, and it may also mount a `nullfs` view of `/usr/ports` or a `tmpfs` for `/var/run`. Reading the output of `mount(8)` inside a running FreeBSD host or jail is the quickest way to get a feel for how these filesystems cooperate.

A quick look on a typical lab host might show:

```
% mount
/dev/ada0p2 on / (ufs, local, journaled soft-updates)
```

```
devfs on /dev (devfs)
fdescfs on /dev/fd (fdescfs)
tmpfs on /tmp (tmpfs, local)
```

Each of those is a filesystem type with its own semantics. `devfs` is the one we care about in this chapter, and its job description is unique: present the kernel's live collection of `struct cdev` objects as a tree of file-like nodes.

8.3.4 Mount Options Worth Knowing

`devfs` accepts a handful of mount options, defined in the `devfs` code and described in the manual page for `mount_devfs(8)`. You will not often change them from the defaults, but it helps to recognise them when you see them in `/etc/fstab`, in jail configurations, or in the output of `mount -v`.

- **ruleset=N**: applies `devfs` ruleset `N` to the mount. A ruleset is a named list of path patterns and actions defined in `/etc/devfs.rules`. This option is how jails limit what their `/dev` looks like.
- **noauto**: present in `fstab` to mark the filesystem as not mounted automatically at boot.
- **late**: mounts late in the boot sequence, after local filesystems and network. Relevant when combined with `ruleset=`.

The typical host configuration needs none of these; the default `devfs` mount at `/dev` is plain. Where they matter most is in jail configuration, which is why Section 10 of this chapter walks through a complete jail example.

8.3.5 devfs Inside Jails

A FreeBSD jail is a restricted execution environment with its own filesystem view, its own set of processes, and usually its own `/dev`. When `jail(8)` starts a jail with `mount.devfs=1` in its configuration, it mounts a separate `devfs` under the jail's `/dev`. That mount is an instance of the same filesystem type, with one decisive difference: it applies a ruleset that filters what appears inside the jail.

The default ruleset for a jail is `devfsrules_jail`, numbered 4 in `/etc/defaults/devfs.rules`. That is the path readers will actually edit or consult on a running system; the source home from which it is installed is `/usr/src/sbin/devfs/devfs.rules`, for readers who want to see the canonical shipped rules. It starts from `devfsrules_hide_all` (which hides everything) and then selectively unhides exactly the handful of nodes a typical jail needs: `null`, `zero`, `random`, `urandom`, `crypto`, `ptmx`, `pts`, `fd` and the `PTY` nodes. Everything else that exists on the host's `/dev` simply does not exist inside the jail. The jail cannot open it, cannot list it, cannot stat it.

This is the mechanism that keeps jails honest. It is also the mechanism you will interact with if your driver's nodes should, or should not, appear inside a jail. If a lab needs `/dev/myfirst/0` to be reachable from a jail, you write a ruleset that unhides it. If a deployment should keep the node out of jails by default, you do nothing: `devfs` will hide it for you. Chapter 8 revisits this in detail when we reach Section 10.

8.3.6 chroot and devfs

There is one related context worth a short note because it sometimes confuses people reading older documentation. A plain `chroot(8)` environment does **not** automatically get its own `devfs`. If a shell script `chroots` into `/compat/linux` and then tries to open `/dev/null`, the open succeeds only because `/dev/null` exists on the host and is visible under the `chroot`'s path through a bind mount or because a `devfs` was mounted there explicitly. If neither is true, the open fails with `ENOENT`.

Jails are different because jails explicitly build a filesystem view with `devfs` mounted inside. A `chroot` is lower-level and leaves filesystem arrangement entirely to the caller. The practical consequence, for driver authors, is that a regression test that runs under `chroot` may or may not be able to see your device, depending on how the `chroot` was set up. When in doubt, test inside a jail.

8.3.7 /dev/fd and the Standard Symlinks

A few entries under `/dev` are not `cdevs` at all; they are symbolic links maintained as part of the `devfs` mount. `/dev/stdin`, `/dev/stdout`, `/dev/stderr` each resolve through `/dev/fd/0`, `/dev/fd/1`, and `/dev/fd/2`, and those in turn are served by `fdescfs(5)` when it is mounted on `/dev/fd`. The combination gives user programs a reliable path-based way to reference the current file descriptor, which is occasionally useful in shell scripts and `awk` programs that want to read or write “whatever the current program's `stdin` is”.

These entries are worth a mention for two reasons. First, they are examples of symlinks inside `devfs`: `ls -l /dev/stdin` shows `lrwxr-xr-x` and an arrow, not `crw-...`. Second, they are a reminder that the entries under `/dev` are not all `cdevs`. Most are; a few are not. When the chapter later contrasts `make_dev_alias(9)` with the `link` directive in `devfs.conf`, this is the distinction that sits underneath.

8.3.8 Why the “Live Mirror” Property Matters to Drivers

The fact that `devfs` is a live mirror of the kernel's current state has several implications for how you design and debug drivers. Each of these points will come back in the chapter; it helps to state them plainly now.

- **A node that is missing from `/dev` is a node your driver did not create.** If you expected `/dev/my-first/0` to exist and it does not, the first thing to check is whether `attach` ran and whether `make_dev_s` returned zero. `dmesg` usually tells you.
- **A node that lingers after unload is a node your driver did not destroy.** That cannot happen if you used `destroy_dev(9)` correctly, but it is a useful frame: if the path survives the `kldunload`, you missed a call.
- **A permission change you made interactively will not survive a reboot.** `devfs` has no on-disk record of what you did. The persistent tools for expressing policy are `devfs.conf` and `devfs.rules`, and Section 10 of this chapter covers them in depth.
- **A jail sees a filtered subset.** Assume, when reasoning about security or feature exposure, that someone is eventually going to run your driver with a jail active on the same host. If your node is too loose and a ruleset lets jails see it, the loose mode has a larger blast radius.

8.4 cdev, vnode, and File Descriptor

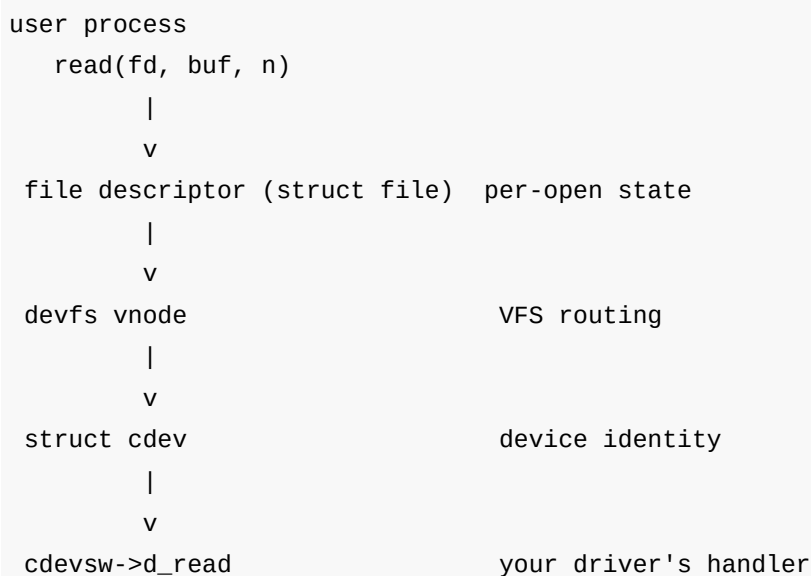
Open a device file and three kernel-side objects quietly line up behind your file descriptor. Understanding this trio is the difference between writing a driver that just happens to work and a driver whose lifecycle you actually control.

The first object is `struct cdev`, the **kernel-side identity of the device**. There is one `struct cdev` per device node, no matter how many programs have it open. Your driver creates it with `make_dev_s(9)` and destroys it with `destroy_dev(9)`. The `cdev` carries the identifying information about the node: its name, its owner, its mode, the `struct cdevsw` that dispatches system calls, and two driver-controlled slots called `si_drv1` and `si_drv2`. Chapter 7 already used `si_drv1` to stash the `softc` pointer, and that is by far the most common use for it.

The second object is a **devfs vnode**. Vnodes are the generic FreeBSD VFS objects that represent open filesystem inodes. Each device node has a vnode underneath it, just as a regular file does, and the VFS layer uses the vnode to route operations toward the correct filesystem. For a device node, that filesystem is `devfs`, and `devfs` forwards the operation to the `cdev`.

The third object is the **file descriptor** itself, represented inside the kernel by a `struct file`. Unlike the `cdev`, there is one `struct file` per open, not one per device. This is where per-open state lives. Two processes that both open `/dev/myfirst0` share the same `cdev` but get separate file structures, and `devfs` knows how to keep those structures cleanly distinct.

Put the three together and the path of a single `read(2)` looks like this:



Every box above exists independently, and each one has a different lifetime. The `cdev` lives for as long as your driver keeps it alive. The vnode lives for as long as anyone has the node resolved on the VFS layer. The `struct file` lives for as long as the user process keeps its descriptor open. When you write a driver, you are filling in only the last row of that diagram, but it helps enormously to know the rows above.

8.4.1 Tracing a Single read(2) Through the Stack

Walk through the story once in prose, with a concrete `read(2)` call as the anchor. A user program has this line:

```
ssize_t n = read(fd, buf, 64);
```

Here is what happens. The kernel receives the `read(2)` syscall and looks up `fd` in the calling process's file-descriptor table. That yields a `struct file`. The kernel sees that the file's type is a vnode-backed file whose vnode lives in `devfs`, so it dispatches through the generic file-operation vector into `devfs`'s read handler.

`devfs` takes a reference on the underlying `struct cdev`, retrieves the pointer to `struct cdevsw` from it, and calls `cdevsw->d_read`. That is **your** function. Inside it you inspect the `struct uio` the kernel prepared, look at the device through the `struct cdev *dev` argument, and optionally recover the per-open structure with `devfs_get_cdevpriv`. When you return, `devfs` releases its reference on the `cdev` and the read call unwinds back to the user program.

A few invariants fall out of that trace and are worth remembering:

- **Your handler never runs if the cdev is gone.** Between `destroy_dev(9)` retiring the node and the last caller dropping its reference, `devfs` simply refuses new operations.
- **Two calls from two processes can reach `d_read` simultaneously.** Neither `devfs` nor the VFS layer serialises callers on your behalf. Concurrency control is your responsibility, and Part 3 of this book is dedicated to it.
- **The `struct file` you are implicitly serving is hidden from your handler.** You do not need to know which descriptor triggered the call; you only need the `cdev`, the `uio`, and (optionally) the `cdevpriv` pointer.

That last point is the one that earns its keep in practice. By hiding the descriptor from the handler, FreeBSD gives you a clean API: all the per-descriptor bookkeeping goes through `devfs_set_cdevpriv` and `devfs_get_cdevpriv`, and your handler code stays small.

8.4.2 Why This Matters for Beginners

Two practical consequences fall out of this model, and both will come back in the next chapter.

First, **pointers stored on the cdev are shared across all opens**. If you store a counter in `si_drv1`, every process that opens the node sees the same counter. That is perfect for driver-wide state such as the `softc`, and terrible for per-session state such as a read position.

Second, **the kernel does not care how many times your device is opened**. Unless you tell it otherwise, every `open(2)` just goes through. If you need exclusive access, as the Chapter 7 code does through its `is_open` flag, you have to enforce it yourself. If you need per-open bookkeeping, you attach that bookkeeping to the file descriptor, not to the `cdev`. We will do both before the end of the chapter.

8.4.3 A Closer Look at struct cdev

You have been using `struct cdev` through a pointer for the whole of Chapter 7. It is time to look inside it. The full definition is in `/usr/src/sys/sys/conf.h`, and the important fields are these:

```

struct cdev {
    void                *si_spare0;
    u_int               si_flags;
    struct timespec      si_atime, si_ctime, si_mtime;
    uid_t               si_uid;
    gid_t               si_gid;
    mode_t              si_mode;
    struct ucred         *si_cred;
    int                 si_drv0;
    int                 si_refcount;
    LIST_ENTRY(cdev)     si_list;
    LIST_ENTRY(cdev)     si_clone;
    LIST_HEAD(, cdev)     si_children;
    LIST_ENTRY(cdev)     si_siblings;
    struct cdev          *si_parent;
    struct mount          *si_mountpt;
    void                 *si_drv1, *si_drv2;
    struct cdevsw         *si_devsw;
    int                  si_iosize_max;
    u_long               si_usecount;
    u_long               si_threadcount;
    union { ... }        __si_u;
    char                 si_name[SPECNAMELEN + 1];
};

```

Not every field matters for a beginner-level driver. A few do, and knowing what they represent saves hours the first time you look at unfamiliar code.

si_name is the null-terminated name of the node as devfs sees it. When you pass "myfirst/%d" and unit 0 to `make_dev_s`, this is the field that ends up containing the string `myfirst/0`. The helper `devtoname(struct cdev *dev)` returns a pointer to this field and is the right tool for logging or debug output.

si_flags is a bit field that carries status flags about the cdev. The flags your driver will touch most often are `SI_NAMED` (set when `make_dev*` has placed the node into devfs) and `SI_ALIAS` (set on aliases created by `make_dev_alias`). The kernel manages them; your code rarely, if ever, writes to this field directly. A useful reading habit: if you see an unfamiliar `SI_*` flag in someone else's driver, look it up in `/usr/src/sys/sys/conf.h` and read the one-line comment.

si_drv1 and **si_drv2** are the two generic driver-controlled slots. Chapter 7 used `si_drv1` to stash the softc pointer, and that is the most common pattern. `si_drv2` is available for a second pointer when you need one. These fields are yours to use; the kernel never touches them.

si_devsw is the pointer to the `struct cdevsw` that dispatches operations on this cdev. It is the link between the node and your handlers.

si_uid, **si_gid**, **si_mode** hold the advertised ownership and mode. They are set from the `mda_uid`, `mda_gid`, `mda_mode` arguments you pass to `make_dev_args_init`. They are mutable in principle, but the right way to change them is through `devfs.conf` or `devfs.rules`, not by assigning into the struct.

si_refcount, **si_usecount**, **si_threadcount** are the three counters devfs uses to keep the cdev alive while anyone might still touch it. **si_refcount** counts long-lived references (the cdev is listed in devfs, other cdevs may alias it). **si_usecount** counts active open file descriptors. **si_threadcount** counts kernel threads currently executing inside a cdevsw handler for this cdev. Your driver almost never reads these directly; the routines **dev_ref**, **dev_rel**, **dev_refthread**, and **dev_relthread** manage them on your behalf. What matters conceptually is that **destroy_dev(9)** will refuse to finish tearing a cdev down while **si_threadcount** is non-zero; it waits, sleeping briefly, until every in-flight handler has returned.

si_parent and **si_children** link a cdev into a parent-child relationship. This is how **make_dev_alias(9)** wires an alias cdev to its primary and how certain clone mechanisms wire per-open nodes to their template. Most of the time you will not interact with these fields; it is enough to know they exist and that they are one of the reasons devfs can unwind an alias cleanly when the primary is destroyed.

si_flags & **SI_ETERNAL** deserves a short note. Some nodes, in particular the ones **null** creates for **/dev/null**, **/dev/zero**, and **/dev/full**, are flagged as eternal with **MAKEDEV_ETERNAL_KLD**. The kernel refuses to destroy them during normal operation. When you start writing modules that expose devices at KLD load time and want the nodes to stay alive across unload attempts, this is the knob. For a driver in active development, leave it alone.

8.4.4 struct cdevsw: The Dispatch Table

Your Chapter 7 cdevsw filled in a handful of fields. The real structure is longer, and the remaining fields are worth at least a recognition pass, because you will meet them in real drivers and sooner or later want to use some of them.

The structure is defined in **/usr/src/sys/sys/conf.h** as:

```
struct cdevsw {
    int            d_version;
    u_int          d_flags;
    const char     *d_name;
    d_open_t       *d_open;
    d_fdopen_t     *d_fdopen;
    d_close_t      *d_close;
    d_read_t       *d_read;
    d_write_t      *d_write;
    d_ioctl_t      *d_ioctl;
    d_poll_t       *d_poll;
    d_mmap_t       *d_mmap;
    d_strategy_t   *d_strategy;
    void           *d_spare0;
    d_kqfilter_t   *d_kqfilter;
    d_purge_t      *d_purge;
    d_mmap_single_t *d_mmap_single;
    /* fields managed by the kernel, not touched by drivers */
};
```

Take the fields one at a time.

d_version is an ABI stamp. It must be set to `D_VERSION`, a value defined a few lines above the structure. The kernel checks this field when registering the `cdevsw` and will refuse to proceed if the stamp does not match. Forgetting to set it is a classic beginner bug: the driver compiles, loads, and then either produces weird errors on first open or crashes the system outright. Always set `d_version = D_VERSION` as the first field in every `cdevsw` you write.

d_flags carries a set of `cdevsw`-wide flags. The flag names are defined with the rest of the structure. The ones worth recognising now are:

- `D_TAPE`, `D_DISK`, `D_TTY`, `D_MEM`: hint to the kernel about the nature of the device. For most drivers you leave this zero.
- `D_TRACKCLOSE`: if set, `devfs` calls your `d_close` for every `close(2)` on a descriptor, not only for the last close. Useful when you want to reliably run per-descriptor teardown even in the face of `dup(2)`.
- `D_MMAP_ANON`: special handling for anonymous memory mappings. `/dev/zero` sets this, which is how `mmap(..., /dev/zero, ...)` yields zero-filled pages.
- `D_NEEDGIANT`: forces dispatch of this `cdevsw`'s handlers under the Giant lock. Modern drivers should not need this; if you see it in code, treat it as a historical marker rather than a model to follow.
- `D_NEEDMINOR`: signals that the driver uses `clone_create` to allocate minor numbers for cloned `cdevs`. You will not need this in Chapter 8.

d_name is the base name string used by the kernel when logging about this `cdevsw`. It also becomes part of the pattern the `clone_create(9)` mechanism uses when it synthesises cloned devices. Set it to a short, human-readable string such as `"myfirst"`.

d_open, **d_close**: session boundaries. Called when a user program calls `open(2)` on the node or releases its last descriptor with `close(2)`. Chapter 7 introduced both, and this chapter refines how you use them.

d_fdopen: an alternative to `d_open` for drivers that want the `struct file *` passed to them directly. Rare in beginner-level drivers. Ignore unless a future chapter introduces it.

d_read, **d_write**: byte-stream operations. Chapter 7 left these stubbed. Chapter 9 implements them with `uiomove(9)`.

d_ioctl: control-path operations. Chapter 25 will treat `ioctl` design in depth. For now, recognise the field and know that it is where structured commands from `ioctl(2)` land.

d_poll: called by `poll(2)` to ask whether the device is currently readable or writable. Chapter 10 handles this as part of the I/O-efficiency story.

d_kqfilter: called by `kqueue(9)` machinery. Same chapter.

d_mmap, **d_mmap_single**: supports mapping the device into a user process's address space. Rare in beginner drivers, covered later when it matters.

d_strategy: called by some kernel layers (notably the old `physio(9)` path) to hand the driver a block of I/O as a `struct bio`. Not relevant for the pseudo-devices you will be writing in Part 2.

d_purge: called by `devfs` during destruction if the `cdev` still has threads running inside its handlers. A well-written `d_purge` wakes those threads and convinces them to return quickly so destruction can proceed. Most simple drivers do not need one; Chapter 10 revisits this in the context of blocking I/O.

When you design your own `cdevsw`, you fill in only the fields that correspond to operations your device actually supports. Every `NULL` field is a polite refusal: the kernel interprets it either as “this operation is not supported on this device” or as “use the default behavior”, depending on which operation it is. Do not reach into the spare fields.

8.4.5 The `D_VERSION` Stamp and Why It Exists

A short aside about `d_version` is useful, because it will save you time the first time your driver mysteriously fails to register.

The kernel interface for `cdevsw` structures has evolved over FreeBSD’s lifetime. Fields have been added, removed, or changed type across major releases. The `d_version` stamp is the kernel’s way of confirming that your module was built against a compatible definition of the structure. The canonical way to set it is:

```
static struct cdevsw myfirst_cdevsw = {
    .d_version = D_VERSION,
    /* ...remaining fields... */
};
```

The macro `D_VERSION` is defined in `/usr/src/sys/sys/conf.h` and will be updated by the kernel team whenever the structure changes in a way that breaks ABI. Modules built against the new header get the new stamp. Modules built against an old header get an old stamp and the kernel refuses them.

This is a small detail that saves large headaches. Set it every time. If you ever see the kernel print a `cdevsw` version mismatch on load, your build environment and your running kernel have drifted; rebuild the module against the headers of the kernel you intend to run.

8.4.6 Reference Counting at the `cdev` Level

The counters you saw on `struct cdev` are the engine that keeps device destruction safe. A simple way to picture them:

- `si_refcount` is the “how many things in the kernel still hold this `cdev` by the neck” count. Aliases, clones, and certain bookkeeping paths bump it. The `cdev` cannot actually be freed while this is non-zero.
- `si_usecount` is the “how many user-space file descriptors have this `cdev` open” count. It is incremented by `devfs` on a successful `open(2)` and decremented on `close(2)`. Your driver never touches it directly.
- `si_threadcount` is the “how many kernel threads are right now executing inside one of my `cdevsw` handlers” count. It is incremented by `dev_refthread(9)` when `devfs` enters a handler on your behalf and decremented by `dev_relthread(9)` when the handler returns. Your driver never touches it directly.

The rule that makes this usable is: `destroy_dev(9)` will block until `si_threadcount` falls to zero and will not return until no more handlers can be entered for this `cdev`. That is how `destroy_dev` is able to guarantee that after it returns, your handlers will not be called again. The section later in this chapter titled “Destroying `cdevs` Safely” revisits this guarantee and the cases where you need its stronger cousin `destroy_dev_drain(9)`.

8.4.7 One More Turn on Lifetimes

With that in hand, the diagram from the previous subsection has a little more meaning than it did the first time. The `cdev` is a long-lived kernel object whose lifetime is under your driver's control. The `vnode` is a VFS-layer object that lives only as long as the filesystem layer has use for it. The `struct file` is a short-lived per-open object that lives only as long as the process keeps the descriptor. And underneath all three, the counters described above keep them honest.

You do not need to memorise any of that. You need to recognise the shape. When you later read a driver and see `dev_refthread` or `si_refcount`, you will remember what they are for. When you watch `destroy_dev` sleep in a debugger, you will recognise that it is waiting for `si_threadcount` to drop. That recognition is what turns kernel code from a puzzle into something you can reason about.

8.5 Permissions, Ownership, and Mode

When your driver calls `make_dev_s(9)`, three fields on the `struct make_dev_args` decide what `ls -l /dev/yournode` will show:

```
args.mda_uid  = UID_ROOT;
args.mda_gid  = GID_WHEEL;
args.mda_mode = 0600;
```

`UID_ROOT`, `UID_BIN`, `UID_UUCP`, `UID_NOBODY`, `GID_WHEEL`, `GID_KMEM`, `GID_TTY`, `GID_OPERATOR`, `GID_DIALER`, and a handful of related names are defined in `/usr/src/sys/sys/conf.h`. Use those constants rather than raw numbers when a well-known identity exists. It makes your driver easier to read and it protects you from silent drift if the numeric value ever changes.

The mode is a classic UNIX permission triple. The meaning of each bit is the same as for a regular file, with the caveat that devices do not care about the execute bit. A few combinations come up often:

- `0600`: owner read and write. The safest default for a driver that is still being developed.
- `0660`: owner and group read and write. Appropriate when you have a well-defined privileged group, such as operator or dialer.
- `0644`: owner read and write, everyone can read. Rare for control devices, sometimes appropriate for read-only status or random-byte style nodes.
- `0666`: everyone can read and write. Used only for intentionally harmless sources like `/dev/null` and `/dev/zero`. Do not reach for this unless you have a real reason.

The rule of thumb is simple: ask “who actually needs to touch this node?” and encode that answer, no more. Opening permissions wider later is easy. Narrowing them after users have come to depend on the wider mode is not.

8.5.1 Where the Mode Comes From

It is worth being explicit about who decides the final mode on the node. Three actors have a say:

1. **Your driver**, through the `mda_uid`, `mda_gid`, and `mda_mode` fields at `make_dev_s()` time. This is the baseline.
2. **/etc/devfs.conf**, which can apply a one-time static adjustment when a node appears. This is the standard way an operator tightens or loosens permissions on a specific path.
3. **/etc/devfs.rules**, which can apply rule-based adjustments, commonly to filter what a jail sees.

If the driver sets `0600` and nothing else is configured, you will see `0600`. If the driver sets `0600` and `devfs.conf` says `perm myfirst/0 0660`, you will see `0660` for that node. The kernel is the mechanism; the operator's configuration is the policy.

8.5.2 Named Groups You Will Meet

FreeBSD ships with a small set of well-known groups that show up repeatedly in device ownership. Each has a matching constant in `/usr/src/sys/sys/conf.h`. A brief field guide helps you choose one quickly:

- **GID_WHEEL** (`wheel`). Trusted administrators. The safest default when you are not sure who should have access beyond root.
- **GID_OPERATOR** (`operator`). Users who run operational tools but are not full administrators. Commonly used for devices that need human supervision but should not require `sudo` every time.
- **GID_DIALER** (`dialer`). Historically for serial-port dial-out access. Still used for TTY nodes that user-space dial programs need.
- **GID_KMEM** (`kmem`). Read access to kernel memory through `/dev/kmem`-style nodes. Very sensitive, rarely the right answer for a new driver.
- **GID_TTY** (`tty`). Ownership for terminal devices.

When a suitable named group exists, use it. When none fits, leave the group as `wheel` and add `devfs.conf` entries for sites that need their own grouping. Inventing a brand-new group in your driver is rarely worth the trouble.

8.5.3 A Worked Example

Suppose the driver baseline is `UID_ROOT`, `GID_WHEEL`, `0600`, and you want to let a specific lab user read and write through a controlled group. The sequence looks like this.

With the driver loaded and no `devfs.conf` entries:

```
% ls -l /dev/myfirst/0
crw----- 1 root  wheel    0x5a Apr 17 10:02 /dev/myfirst/0
```

Add a section to `/etc/devfs.conf`:

```
own    myfirst/0    root:operator
perm   myfirst/0    0660
```

Apply it and inspect again:

```
% sudo service devfs restart
% ls -l /dev/myfirst/0
crw-rw----  1 root  operator   0x5a Apr 17 10:02 /dev/myfirst/0
```

The driver was not reloaded. The cdev in the kernel is the same object. Only the advertised ownership and mode changed, and they changed because a policy file told devfs to change them. This is the layering you want: the driver ships a defensible baseline, and the operator shapes the view.

8.5.4 Case Studies from the Tree

It helps to spend a minute on the permissions that real FreeBSD devices advertise, because the choices those drivers make are not accidental. Each one is a small design decision, and each one is consistent with the threat model for that kind of node.

`/dev/null` and `/dev/zero` ship with mode `0666`, `root:wheel`. Everyone on the system, privileged or not, is allowed to open them and read or write through them. That is the correct choice because the data they carry is trivially inexhaustible (zero bytes out, bytes-disposed-of in, no hardware state, no secrets). Making them any tighter would break a long list of scripts, tools, and programming idioms that depend on their being universally available. The code that creates them is in `/usr/src/sys/dev/null/null.c`, and the arguments to `make_dev_credf(9)` there are worth a glance.

`/dev/random` is typically mode `0644`, readable by anyone, writable only by root. Read access is deliberately broad because many userland programs need entropy. Write access is narrow because feeding the entropy pool is a privileged operation.

`/dev/mem` and `/dev/kmem` are historically mode `0640`, owner `root`, group `kmem`. The group exists precisely so that privileged monitoring tools can link to them without running as root. The mode is tight because the nodes expose raw memory; a casually readable `/dev/mem` would be a disaster. If you ever see a driver default to a mode this loose for a node that carries hardware state or kernel memory, treat it as a defect.

`/dev/pf`, the control node for the packet filter, is mode `0600`, owner `root`, group `wheel`. A user that can write into `/dev/pf` can change firewall rules. There is no acceptable broader mode; the whole point of the interface is to centralise privileged network configuration, and anything looser would turn the firewall into a free-for-all.

`/dev/bpf*`, the Berkeley Packet Filter nodes, are mode `0600`, owner `root`, group `wheel`. A reader of `/dev/bpf*` sees every packet on an attached interface. That is unambiguous privilege, and the permission reflects it.

TTY nodes under `/dev/ttyu*` and similar hardware-serial surfaces are usually mode `0660`, owner `uucp`, group `dialer`. The `dialer` group exists to let a set of trusted users run dial-out programs without `sudo`. The permission set is the narrowest that still lets the intended workflow function.

The pattern is easy to name: **FreeBSD's base system never chooses wide device permissions unless the data on the other side is harmless**. When you design a node of your own, use that pattern as a mental check. If your node carries data that could hurt someone, narrow the mode. If it carries data that is trivially regenerable and trivially discardable, widening is defensible; do it anyway only when there is a reason.

8.5.5 Least Privilege Applied to Device Files

“Least privilege” is an overused phrase, but it is precisely right when it applies to device files. You, the driver author, are choosing who can talk to your code from userland, and you get to set the lower bound. Every choice you make wider than necessary is a choice that invites mistakes later.

A practical checklist for each new node you design:

1. **Name the primary consumer in a sentence.** “The monitor daemon reads status every second.” “The control tool invokes `ioctl` to push configuration.” “Users of the operator group can read raw packet counters.” If you cannot name the consumer, you cannot set permissions; you are guessing.
2. **Derive the mode from the sentence.** A monitor daemon that runs as `root:wheel` and reads once a second wants `0600`. A control tool that a subset of privileged admins run wants `0660` with a dedicated group. A read-only status node consumed by unprivileged dashboards wants `0644`.
3. **Put the reasoning in a comment next to the `mda_mode` line.** Future maintainers will thank you. Future auditors will thank you more.
4. **Default to `UID_ROOT`.** There is almost never a reason for the owner of a node created by a driver to be anything else, unless the driver explicitly models a non-root daemon identity.

The opposite habit, which the book wants to inoculate against, is the “open it up and tighten later” impulse. Permissions on a shipped driver are very hard to tighten, because by the time someone notices, some user’s workflow depends on the loose mode and the tightening breaks their day. Start tight. Widen when you have reviewed a real request.

8.5.6 Transitioning a Loose Mode to a Tight One

Occasionally you will inherit a driver that was wide open and needs to be narrowed. The right approach is three-stage:

Stage 1: Announce. Put the planned change in a release note, in the driver’s kernel log on first attach, and in whatever operator-facing channel your project uses. Invite feedback for at least one release cycle.

Stage 2: Offer a transition path. Either a `devfs.conf` entry that reopens the old mode for people who need it, or a `sysctl` that the driver reads at attach to pick its default mode. The important property is that a site with a legitimate need to keep the old mode can do so without forking the driver.

Stage 3: Flip the default. In the next release after the transition window ends, change the driver’s own `mda_mode` to the narrower value. The `devfs.conf` escape hatch remains for sites that need it; everyone else gets the narrower default.

None of that is specific to FreeBSD; it is how any well-run project handles backward-incompatible interface changes. It is worth naming here because device-file permissions have exactly this property: they are part of your driver’s public interface.

8.5.7 What the `uid` and `gid` Constants Actually Are

The `UID_*` and `GID_*` constants defined in `/usr/src/sys/sys/conf.h` are **not** guaranteed to match the user and group database on every system. The names chosen in the header correspond to identities that the FreeBSD base system reserves in `/etc/passwd` and `/etc/group`, but a locally modified system could in theory renumber

them, or a product built on FreeBSD could add its own. In practice, on every FreeBSD system you will touch, the constants match.

The discipline to hold is simple: use the symbolic name when one exists, and look it up in the header before you invent a new identity. The header currently defines at least these:

- User IDs: `UID_ROOT` (0), `UID_BIN` (3), `UID_UUCP` (66), `UID_NOBODY` (65534).
- Group IDs: `GID_WHEEL` (0), `GID_KMEM` (2), `GID_TTY` (4), `GID_OPERATOR` (5), `GID_BIN` (7), `GID_GAMES` (13), `GID_VIDEO` (44), `GID_RT_PRI0` (47), `GID_ID_PRI0` (48), `GID_DIALER` (68), `GID_NOGROUP` (65533), `GID_NOBODY` (65534).

If you need an identity that is not in the list, the base system probably does not have one reserved. In that case, leave ownership as `UID_ROOT/GID_WHEEL` and let operators map your node to their own local group through `devfs.conf`. Inventing a new group inside your driver is almost always the wrong move.

8.5.8 Three-Layer Policy: Driver, `devfs.conf`, `devfs.rules`

When you combine the driver's baseline with `devfs.conf` and `devfs.rules`, you get a layered policy model that is worth seeing once end-to-end. Consider a device that the driver creates with `root:wheel 0600`. Three layers act on it:

- **Layer 1, the driver itself:** sets the baseline. Every `/dev/myfirst/0` on every `devfs` mount starts at `root:wheel 0600`.
- **Layer 2, `/etc/devfs.conf`:** applied once per host `devfs` mount, typically at boot. Can change ownership, mode, or add a symlink. On the running host, after `service devfs restart`, the node might appear as `root:operator 0660`.
- **Layer 3, `/etc/devfs.rules`:** applied at mount time based on the ruleset attached to the mount. A jail whose `devfs` mount uses ruleset `10` sees the filtered, possibly-modified subset. The same node might be hidden inside the jail, or it might be unhidden with further mode and group adjustments.

The practical consequence of this layering is that **the same cdev can look different in different places at the same time**. On the host it might be `0660` owned by `operator`. In a jail it might be `0640` owned by a jailed user identity. In another jail it might not exist at all.

That is a feature, not a bug. It lets you ship a driver with a strict baseline and lets operators widen per-environment without editing your code. Chapter 8 Section 10 walks through all three layers with a worked example.

8.6 Naming, Unit Numbers, and Subdirectories

The `printf`-style argument to `make_dev_s(9)` chooses where in `/dev` the node appears. In Chapter 7 you used:

```
error = make_dev_s(&args, &sc->cdev, "myfirst%d", sc->unit);
```

That produced `/dev/myfirst0`. Two details hide in there.

The first detail is `sc->unit`. It is the Newbus unit number that FreeBSD assigns to your device instance. With one instance attached, you get 0. If your driver supported multiple instances, you might see `myfirst0`, `myfirst1`, and so on.

The second detail is the format string itself. Device names are paths relative to `/dev`, and they may contain slashes. A name such as `"myfirst/%d"` does not produce a weird filename with a slash in it; `devfs` interprets the slash the way a filesystem does, creates the intermediate directory if needed, and places the node inside. So:

- `"myfirst%d"` with unit 0 gives `/dev/myfirst0`.
- `"myfirst/%d"` with unit 0 gives `/dev/myfirst/0`.
- `"myfirst/control"` gives `/dev/myfirst/control`, with no unit number at all.

Grouping related nodes into a subdirectory is a common pattern for drivers that expose more than one surface. Think of `/dev/led/*` from `/usr/src/sys/dev/led/led.c`, or `/dev/pf`, `/dev/pflog*`, and friends from the packet-filter subsystem. The subdirectory makes the relationship obvious at a glance, keeps the top level of `/dev` tidy, and lets an operator grant or deny access to the whole set with a single `devfs.conf` line.

You will adopt this pattern for `myfirst` in this chapter. The main data path moves from `/dev/myfirst0` to `/dev/myfirst/0`. Then you will add an alias so the old path keeps working for any lab scripts that remember the previous layout.

8.6.1 Names in the Real FreeBSD Tree

Browsing the `/dev` of a running FreeBSD system is educational in its own right, because the naming conventions you see there were shaped by the same pressures your driver will face. A short tour, grouped by theme:

- **Direct device names.** `/dev/null`, `/dev/zero`, `/dev/random`, `/dev/urandom`. One cdev per node, top-level, short stable names. Good for singletons with no hierarchy.
- **Unit-numbered names.** `/dev/bpf0`, `/dev/bpf1`, `/dev/ttyu0`, `/dev/md0`. One cdev per instance, numbered from zero. The format string looks like `"bpf%d"` and the driver manages the unit numbers.
- **Subdirectories per driver.** `/dev/led/*`, `/dev/pts/*`, `/dev/ipmi*` in some configurations. Used when a single driver exposes many related nodes. Makes operator policy simple: one `devfs.conf` or `devfs.rules` entry can cover the whole set.
- **Split data and control nodes.** `/dev/bpf` (the cloning entry point) plus per-open clones, `/dev/fido/*` for FIDO devices, and so on. Used when a driver needs different semantics for discovery versus data.
- **Aliased names for convenience.** `/dev/stdin`, `/dev/stdout`, `/dev/stderr` are symlinks that `devfs` provides for the current process's file descriptors. `/dev/random` and `/dev/urandom` were once aliased; in modern FreeBSD they are separate nodes served by the same random driver but the history is still visible.

You do not need to memorise these patterns. You do need to recognise them, because when you read existing drivers they will all make more sense once the naming convention is named.

8.6.2 Multiple Nodes Per Device

Some drivers expose one node and that is enough. Other drivers expose several, each with different semantics. A common split is:

- A **data node** that carries the bulk payload (reads, writes, mmaps) and is meant for high-throughput use.
- A **control node** that carries management traffic (configuration, status, reset) and is typically group-readable for monitoring tools.

When a driver does this, it calls `make_dev_s(9)` twice in `attach()` and keeps both `cdev` pointers in the softc. In Chapter 8 you will stop at one data node plus one alias, but the pattern is worth knowing now so you recognise it when you see it.

Lab 8.5 builds out a minimal two-node variant of `myfirst` with a data node at `/dev/myfirst/0` and a control node at `/dev/myfirst/0ctl`. Each node has its own `cdevsw` and its own permission mode. The lab is there to show how the pattern looks in code; most of your drivers in later chapters will use it.

8.6.3 The `make_dev` Family in Depth

You have used `make_dev_s(9)` for every node you have created so far. FreeBSD actually provides a small family of `make_dev*` functions, each with slightly different ergonomics. Reading existing drivers will expose you to all of them, and knowing when to use which saves grief later.

The full declarations live in `/usr/src/sys/sys/conf.h`. In order of increasing modernity:

```
struct cdev *make_dev(struct cdevsw *_devsw, int _unit, uid_t _uid, gid_t _gid,
                     int _perms, const char *_fmt, ...);

struct cdev *make_dev_cred(struct cdevsw *_devsw, int _unit,
                           struct ucred *_cr, uid_t _uid, gid_t _gid, int _perms,
                           const char *_fmt, ...);

struct cdev *make_dev_credf(int _flags, struct cdevsw *_devsw, int _unit,
                             struct ucred *_cr, uid_t _uid, gid_t _gid, int _mode,
                             const char *_fmt, ...);

int make_dev_p(int _flags, struct cdev **_cdev, struct cdevsw *_devsw,
               struct ucred *_cr, uid_t _uid, gid_t _gid, int _mode,
               const char *_fmt, ...);

int make_dev_s(struct make_dev_args *_args, struct cdev **_cdev,
               const char *_fmt, ...);
```

Take them one at a time.

make_dev is the original positional-argument form. It returns the new `cdev` pointer directly, or panics on any error. Panicking on error is a strong hint that it is intended for code paths that cannot recover, such as the very early initialization of truly-eternal devices. Avoid it in new drivers. It is still in the tree only because older drivers use it, and because some of those drivers are places where an early panic is genuinely acceptable.

make_dev_cred adds a credential (`struct ucred *`) argument. The credential is used by devfs when applying rules; it tells the system “this `cdev` was created by this credential” for rule-matching purposes. Most drivers

pass `NULL` for the credential and get the default behavior. You will see this form in drivers that clone devices on demand in response to user requests; it is not common elsewhere.

`make_dev_credf` extends `make_dev_cred` with a flags word. This is the first member of the family that lets you say “do not panic if this fails; return `NULL` so I can handle it”.

`make_dev_p` is a functional equivalent to `make_dev_credf` with a cleaner return-value convention: it returns an `errno` value (zero on success) and writes the new `cdev` pointer through an output parameter. This is the form most widely used in modern code bases that were written before `make_dev_s` existed.

`make_dev_s` is the modern recommended form. It accepts a pre-populated `struct make_dev_args` (initialised with `make_dev_args_init_impl` and described below) and writes the `cdev` pointer through an output parameter. It returns an `errno` value, zero on success. The reason the book uses it is simple: it is the easiest form to read, the easiest form to extend (adding a new field to the argument struct is ABI-friendly), and the easiest form to error-check.

The argument structure, also from `/usr/src/sys/sys/conf.h`:

```
struct make_dev_args {
    size_t      mda_size;
    int         mda_flags;
    struct cdevsw *mda_devsw;
    struct ucred *mda_cr;
    uid_t       mda_uid;
    gid_t       mda_gid;
    int         mda_mode;
    int         mda_unit;
    void        *mda_si_drv1;
    void        *mda_si_drv2;
};
```

`mda_size` is set automatically by `make_dev_args_init(a)`; you never touch it. `mda_flags` carries the `MAKEDEV_*` flags described below. `mda_devsw`, `mda_cr`, `mda_uid`, `mda_gid`, `mda_mode`, and `mda_unit` correspond to the positional arguments of the older forms. `mda_si_drv1` and `mda_si_drv2` let you pre-populate the driver pointer slots on the resulting `cdev`, which is how you avoid the window where `si_drv1` could briefly be `NULL` after `make_dev_s` returns but before you assign it. Always populate `mda_si_drv1` before the call.

8.6.4 Which Form Should You Use?

For new drivers, **use `make_dev_s`**. Every example in this book uses it, and every driver you write for yourself should do the same unless a very specific reason forces otherwise.

For reading existing code, recognise all of them. If you find a driver that calls `make_dev(...)` and ignores its return value, you are looking at either a driver that predates the modern APIs or a driver whose authors decided a panic on failure is acceptable. Both are defensible in context; neither is the right default for new code.

8.6.5 The MAKEDEV_* Flags

The flags that can be OR-ed into `mda_flags` (or passed as the first argument to `make_dev_p` and `make_dev_credf`) are defined in `/usr/src/sys/sys/conf.h`. Each one carries a specific meaning:

- **MAKEDEV_REF**: increments the reference count on the resulting `cdev` by one, in addition to the usual reference. Used when the caller plans to hold the `cdev` pointer long-term across events that would ordinarily drop the reference. Rare in beginner-level drivers.
- **MAKEDEV_NOWAIT**: tells the allocator not to wait if memory is tight. On an out-of-memory condition the function returns `ENOMEM` (for `make_dev_s`) or `NULL` (for older forms) instead of blocking. Use this only if your caller cannot afford to sleep.
- **MAKEDEV_WAITOK**: the inverse. Tells the allocator it is safe to sleep for memory. This is the default for `make_dev` and `make_dev_s`, so you rarely spell it out.
- **MAKEDEV_ETERNAL**: marks the `cdev` as never-to-be-destroyed. `devfs` will refuse to honor `destroy_dev(9)` on it during normal operation. Used by the eternal in-kernel devices such as `null`, `zero`, and `full`. Do not set this in a driver you plan to unload.
- **MAKEDEV_CHECKNAME**: asks the function to validate the node name against `devfs`'s rules before creating it. On failure it returns an error rather than creating a badly-named `cdev`. Useful in code paths that synthesise names from user input.
- **MAKEDEV_WHTOUT**: creates a “whiteout” entry, used in conjunction with stacked filesystems to mask an underlying entry. Not something you will encounter in driver work.
- **MAKEDEV_ETERNAL_KLD**: a macro that expands to `MAKEDEV_ETERNAL` when the code is built outside a loadable module and to zero when it is built as a KLD. This lets shared source for a device (like `null`) set the flag when statically compiled and clear it when loaded as a module, so that the module remains unloadable.

For a typical beginner-level driver, the flag field is zero, which is what the `myfirst` examples in the companion tree use. `MAKEDEV_CHECKNAME` is worth reaching for when the node name is built from user input or from a string whose provenance you do not fully control; for a driver that passes a constant format string such as `"myfirst/%d"`, the flag adds nothing useful.

8.6.6 The `cdevsw d_flags`

Separate from the `MAKEDEV_*` flags, the `cdevsw` itself carries a `d_flags` field that shapes how `devfs` and other kernel machinery treat the `cdev`. These flags were listed in the `cdevsw` tour a few sections back; this section is the place to understand when to set them.

D_TRACKCLOSE is the flag you are most likely to want in Chapter 8. By default, `devfs` calls your `d_close` only when the last file descriptor referring to the `cdev` is released. If a process has called `dup(2)` or `fork(2)` and two descriptors share the open, `d_close` fires once, at the very end. That is often what you want. It is not what you want if you need a reliable per-descriptor close hook. Setting `D_TRACKCLOSE` makes `devfs` call `d_close` for every `close(2)` on every descriptor. For a driver that uses `devfs_set_cdevpriv(9)` for per-open state, the destructor is usually the better hook; `D_TRACKCLOSE` remains useful when the semantics of your device genuinely require each close to be observable.

D_MEM tags the `cdev` as a memory-style device; `/dev/mem` itself sets this. It changes how certain kernel paths treat I/O to the node.

D_DISK, **D_TAPE**, **D_TTY** are hints for the category of device. Modern drivers mostly do not set them, because GEOM owns disks, the TTY subsystem owns TTYs, and tape devices route through their own layer. You will see them on legacy drivers.

D_MMAP_ANON alters how mapping the device yields pages. The zero device sets it; mapping `/dev/zero` yields anonymous, zero-filled pages. Useful to recognise; you will not need to set it until you write a driver that wants the same semantics.

D_NEEDGIANT requests that all `cdevsw` handlers for this `cdev` be dispatched under the Giant lock. It exists as a safety blanket for drivers that have not been audited for SMP. A new driver should not set this flag. If you see it in code written after 2010 or so, treat it with suspicion.

D_NEEDMINOR tells `devfs` that the driver uses `clone_create(9)` to allocate minor numbers on demand. You will not encounter this until you write a cloning driver, which is out of scope for this chapter.

The flags you will set in `myfirst` are, in most versions, none. Once Chapter 8 adds per-open state, the driver still does not need **D_TRACKCLOSE** because the `cdevpriv` destructor covers the per-descriptor cleanup need.

8.6.7 Name Length and Name Characters

`make_dev_s` accepts a `printf`-style format and produces a name that `devfs` stores in the `cdev`'s `si_name` field. The size of that field is `SPECNAMELEN + 1`, and `SPECNAMELEN` is currently 255. A name longer than that is an error.

In addition to length, a name must be acceptable as a filesystem path under `devfs`. That means it must not contain null bytes, it must not use `.` or `..` as components, and it should not use characters that shells or scripts interpret specially. The safest set is lowercase ASCII letters, digits, and the three separators `/`, `-`, and `..`. Other characters will sometimes work and sometimes not; if you are ever tempted to use spaces, colons, or non-ASCII characters in a device name, stop and pick a simpler name instead.

8.6.8 Unit Numbers: Where They Come From

Unit numbers are small integers that distinguish instances of the same driver. They appear in the device name (`myfirst0`, `myfirst1`), in `sysctl` branches (`dev.myfirst.0`, `dev.myfirst.1`), and in the `cdev`'s `si_drv0` field.

Two ways are common for assigning them:

Newbus assignment. When your driver attaches to a bus and Newbus instantiates a device, the bus assigns a unit number. You retrieve it with `device_get_unit(9)` and use it as `sc->unit`, exactly as Chapter 7 does. Newbus guarantees the number is unique within the driver's namespace.

Explicit allocation with `unrhdr`. For drivers that create nodes outside the Newbus flow, the `unrhdr(9)` allocator assigns unit numbers from a pool. `/usr/src/sys/dev/led/led.c` uses this: `sc->unit = alloc_unr(led_unit);`. The LED framework does not attach through Newbus for each LED, so it cannot ask Newbus for a unit number; it maintains its own unit pool.

For a beginner driver built on Newbus, the first approach is the one to use. The second becomes relevant once you write a pseudo-device that can be instantiated many times on demand, which is a later-chapter topic.

8.6.9 Naming Conventions in the Tree

Since you are likely to read real FreeBSD drivers as part of learning, it helps to recognise the shapes their names take. A short tour:

- **bpf%d**: one node per BPF instance. Seen in `/usr/src/sys/net/bpf.c`.
- **md%d**: memory disks. `/usr/src/sys/dev/md/md.c`.
- **led/%s**: subdirectory per driver, one node per LED. `/usr/src/sys/dev/led/led.c` uses the name argument as a free-form string, chosen by the caller, e.g. `led/ehci0`.
- **ttyu%d**, **cuaU%d**: hardware serial ports, paired “in” and “out” nodes.
- **ptyp%d**, **ttyp%d**: pseudo-terminal pairs.
- **pts/%d**: modern PTY allocation in a subdirectory.
- **fuse**: singleton entry point for the FUSE subsystem.
- **mem**, **kmem**: singletons for memory inspection.
- **pci**, **pciconf**: PCI bus inspection interfaces.
- **io**: I/O-port access, singleton.
- **audit**: audit subsystem control device.

Notice that in most of these the name encodes the driver’s identity. That is deliberate. When an operator later needs to write a `devfs.conf` rule or a firewall rule or a backup script, they match on paths, and predictable paths make their life easier.

8.6.10 Handling Multiple Units

Your Chapter 7 driver registered exactly one Newbus child in its `device_identify` callback, so there is only one instance and the only unit number is 0. Some drivers want more than one instance, either at boot or on demand.

For a driver instantiated at boot with a fixed count, the pattern is to add more children in `device_identify`:

```
static void
myfirst_identify(driver_t *driver, device_t parent)
{
    int i;

    for (i = 0; i < MYFIRST_INSTANCES; i++) {
        if (device_find_child(parent, driver->name, i) != NULL)
            continue;
        if (BUS_ADD_CHILD(parent, 0, driver->name, i) == NULL)
            device_printf(parent,
                "myfirst%d: BUS_ADD_CHILD failed\n", i);
    }
}
```

Newbus calls `attach` for each child, and each call gets its own `softc` and its own unit number. Your `make_dev_s` format string `"myfirst/%d"` with `sc->unit` then produces `/dev/myfirst/0`, `/dev/myfirst/1`, and so on.

For a driver instantiated on demand, the architecture is very different. You typically expose a single “control” cdev, and when a user performs an operation on it the driver allocates a new instance and a new cdev. The memory disk driver in `/usr/src/sys/dev/md/md.c` is a clear example: `/dev/mdctl` accepts a `MDIOCATTACh` ioctl, and each successful attach produces a new `/dev/mdN` cdev through the GEOM layer. The pseudo-terminal subsystem takes a similar approach: a user opening `/dev/ptmx` is handed a freshly allocated `/dev/pts/N` on the other side. Chapter 8 does not walk you through those machineries; it is enough to know that when you see a driver create cdevs from inside an event handler rather than from `attach`, dynamic instantiation is the pattern you are looking at.

8.6.11 A Small Detour: devtoname and Friends

Three small helpers come up often in driver code and in the book from here on. They are worth collecting:

- **devtoname(cdev)**: returns a pointer to the node’s name. Read-only. Used for logging: `device_printf(dev, "created /dev/%s\n", devtoname(sc->cdev))`.
- **dev2unit(cdev)**: returns the `si_drv0` field, which is conventionally the unit number. Defined as a macro in `conf.h`.
- **device_get_nameunit(dev)**: used on a `device_t`, returns the Newbus-scoped name such as `"my-first0"`. Useful for mutex names.

All three are safe to use in contexts where the cdev or device is known to be alive, which for a driver handler is always.

8.7 Aliases: One cdev, More Than One Name

Sometimes a device needs to be reachable under more than one name. Maybe you renamed a node and want the old name to keep working during a deprecation window. Maybe you want a stable short name that always points at unit 0 without the user having to know which unit is current. Maybe the rest of the system already has a strong convention and you want to play nicely with it.

FreeBSD offers `make_dev_alias(9)` for this. An alias is itself a `struct cdev`, but one that carries the `SI_ALIAS` flag and shares the same underlying dispatch machinery as the primary node. A user program opening the alias lands in the same `cdevsw` handlers as a user program opening the primary name.

Signatures, from `/usr/src/sys/sys/conf.h`:

```
struct cdev *make_dev_alias(struct cdev *_pdev, const char *_fmt, ...);
int         make_dev_alias_p(int _flags, struct cdev **_cdev,
                             struct cdev *_pdev, const char *_fmt, ...);
```

You pass in the primary cdev, a format string, and optional arguments. You get back a new cdev that represents the alias. When you are done, you destroy the alias with `destroy_dev(9)`, the same way you destroy any other cdev.

Here is the shape of the code you will add to `myfirst_attach()`:

```

sc->cdev_alias = make_dev_alias(sc->cdev, "myfirst");
if (sc->cdev_alias == NULL) {
    device_printf(dev, "failed to create /dev/myfirst alias\n");
    /* fall through; the primary node is still usable */
}

```

Two observations about that snippet. First, failing to create an alias is not fatal. The primary path still works, so we log and continue. Second, you only need to keep a pointer to the alias cdev if you plan to destroy it on detach. In most drivers you do, so stash it in the softc right next to cdev.

8.7.1 Aliases vs link in devfs.conf

Readers familiar with UNIX symbolic links sometimes ask why FreeBSD offers two different ways to give a device a second name. The distinction is real and worth stating clearly.

A `make_dev_alias(9)` alias is a **second cdev that shares its dispatch machinery with the primary**. When the user opens it, devfs walks directly to your cdevsw handlers. There is no symlink in the filesystem. `ls -l` on the alias shows another character-special node with its own mode and ownership. The kernel knows the alias is bound to the primary cdev (the `SI_ALIAS` flag and `si_parent` pointer record that relationship) and cleans it up automatically if the primary goes away, provided your driver remembers to call `destroy_dev(9)` on it.

A `link` directive in `/etc/devfs.conf` creates a **symbolic link inside devfs**. `ls -l` shows an `l` in the type field and an arrow pointing at the target. Open it and the kernel first resolves the symlink, then opens the target. The target and the link have independent permission and ownership; the symlink itself carries no access policy beyond its existence.

Which to choose?

- Use `make_dev_alias` when the driver itself has a reason to expose the extra name, for instance a short well-known form or a legacy path that must look identical to the new one at the permission level.
- Use `link` in `devfs.conf` when the operator wants a convenience shortcut and the driver has no opinion. Nothing about that sort of link belongs in kernel code.

Both approaches work. The wrong choice is not dangerous; it is usually just clumsy. Keep driver code lean and let operator policy stay where policy lives.

8.7.2 Comparison Table: Three Ways to Give a Node Two Names

A short comparison pulls the distinctions into one place:

Property	<code>make_dev_alias</code>	<code>devfs.conf link</code>	Symbolic link via <code>ln -s</code>
Lives in kernel code	yes	no	no
Lives in devfs	yes	yes	no (lives in underlying FS)
<code>ls -l</code> shows as c	yes	no (shows as l)	no (shows as l)

Property	make_dev_alias	devfs.conf link	Symbolic link via ln -s
Carries its own mode and owner	yes	inherits target	inherits target
Auto-cleaned on driver unload	yes	yes (next service devfs restart)	no
Survives a reboot	only while driver is loaded	yes, if in devfs.conf	yes, if under /etc or similar
Appropriate for driver-owned name	yes	no	no
Appropriate for operator shortcut	no	yes	sometimes

The pattern is: drivers own their primary names and any aliases that carry policy; operators own convenience links that do not carry policy. Crossing that line is where future maintenance pain comes from.

8.7.3 The make_dev_alias_p Variant

make_dev_alias has a sibling that accepts a flags word and returns an errno, for the same reasons the main make_dev family does. Its declaration in /usr/src/sys/sys/conf.h:

```
int make_dev_alias_p(int _flags, struct cdev **_cdev, struct cdev *_pdev,
                    const char *_fmt, ...);
```

The valid flags are MAKEDEV_WAITOK, MAKEDEV_NOWAIT, and MAKEDEV_CHECKNAME. Behavior is analogous to make_dev_p: zero on success, the new cdev written through the output pointer, a non-zero errno value on failure.

If your alias creation is in a path that cannot sleep, use make_dev_alias_p(MAKEDEV_NOWAIT, ...) and be prepared for ENOMEM. In the ordinary case where your alias is created during attach under regular conditions, make_dev_alias(9) is fine; it uses MAKEDEV_WAITOK internally.

8.7.4 The make_dev_physpath_alias Variant

There is a third alias function, make_dev_physpath_alias, used by drivers that want to publish physical-path aliases in addition to their logical names. It exists to support the hardware-topology paths under /dev/something/by-path/... that certain storage drivers expose. Most beginner drivers never need it.

8.7.5 Reading Uses of make_dev_alias in the Tree

A useful exercise: grep for make_dev_alias across /usr/src/sys and look at the contexts where it is used. You will find it in storage drivers that want to publish a stable name alongside a dynamically-numbered one, in certain pseudo-devices that want a legacy compatibility name, and in a small number of specialised drivers that model a hardware topology.

Most drivers do not use it, and that is fine. When a driver does, the reason is almost always one of three:

1. **Legacy path compatibility.** A driver that was renamed but must keep the old name working.
2. **A well-known shortcut.** A short name that always resolves to instance zero or to the current default, so shell scripts can write one path instead of negotiating the unit number.
3. **Topology exposure.** A name that reflects where the hardware sits, in addition to what the hardware is.

Your `myfirst` driver is using case 1: `/dev/myfirst` as a shortcut for `/dev/myfirst/0` so the Chapter 7 prose still resolves. That is the shape of a typical beginner use.

8.7.6 Alias Lifetimes and Destruction Order

A cdev that is registered as an alias has the `SI_ALIAS` flag set and is linked into the primary cdev's `si_children` list with the `si_parent` back-pointer. This means the kernel knows the relationship and will do the right thing even if you tear the cdev down in a slightly wrong order. It does not mean you can ignore the order; it means destruction is more forgiving than teardown of general kernel objects.

In practice, the rule you should follow in your detach path is: **destroy the alias first, then the primary**. The example drivers in the companion tree do this, and the reason is simple readability. Any other order makes your code harder to reason about, and reviewers will flag it.

If a driver omits the `destroy_dev` call on the alias entirely, the primary's destruction will unwind the alias automatically when the primary goes away; that is what `destroy_dev1` does when it walks `si_children`. But leaving that work to the destructor is wasteful because the primary is holding a reference that keeps it alive longer than needed, and because the operator sees the alias disappear "later" rather than cleanly at unload time. Just destroy both.

8.7.7 When Aliases Start to Smell

A few patterns with aliases are mild code smells worth naming:

- **Chains of aliases.** Aliases of aliases are legal but almost always mean the driver is trying to paper over a naming decision that should have been revisited. If you find yourself wanting to alias an alias, stop and rename the primary.
- **Too many aliases.** One or two is routine. Five or more suggests the driver is not sure what it wants to be called. Revisit the naming.
- **Aliases with wildly different modes.** Two cdevs that point at the same handler set but expose radically different permission modes are indistinguishable from a trap. Make the permissions consistent, or use two separate primaries with two separate `cdevsw` values that enforce different policies in code.

None of these are errors. They are signals that the design is drifting. Notice them early and the driver stays readable; ignore them and the driver becomes a thing reviewers dread.

8.8 Per-Open State with `devfs_set_cdevpriv`

Now we come to the piece of the chapter that prepares the ground for Chapter 9. Your Chapter 7 driver enforced **exclusive open** by setting a flag in the `softc`. That works, but it is the coarsest possible policy. Many real devices

allow several openers and want to keep a small amount of bookkeeping **per file descriptor**, not per device. Think of a log stream, a status feed, or any node where different consumers want their own read positions.

FreeBSD offers three related routines for this, declared in `/usr/src/sys/sys/conf.h` and implemented in `/usr/src/sys/fs/devfs/devfs_vnops.c`:

```
int  devfs_set_cdevpriv(void *priv, d_priv_dtor_t *dtr);
int  devfs_get_cdevpriv(void **datap);
void devfs_clear_cdevpriv(void);
```

The model is simple and pleasant to use:

1. Inside your `d_open` handler, allocate a small per-open structure and call `devfs_set_cdevpriv(priv, dtor)`. The kernel attaches `priv` to the current file descriptor and remembers `dtor` as the function to call when that descriptor eventually closes.
2. In `d_read`, `d_write`, or any other handler, call `devfs_get_cdevpriv(&priv)` to retrieve the pointer.
3. When the process calls `close(2)`, or exits, or otherwise drops its last reference to the descriptor, `devfs` calls your destructor with `priv`. You free whatever you allocated.

You do not need to worry about the order of cleanup with respect to your own `d_close` handler. `Devfs` handles it. The important invariant is that your destructor will be called exactly once per successful `devfs_set_cdevpriv`.

A real example from `/usr/src/sys/net/bpf.c` looks like this:

```
d = malloc(sizeof(*d), M_BPF, M_WAITOK | M_ZERO);
error = devfs_set_cdevpriv(d, bpf_dtor);
if (error != 0) {
    free(d, M_BPF);
    return (error);
}
```

That is essentially the whole pattern. `BPF` allocates a per-open descriptor, registers it, and if the registration fails, frees the allocation and returns the error. The destructor `bpf_dtor` cleans up when the descriptor dies. You will do the same thing for `myfirst`, with a much smaller per-open structure.

8.8.1 A Minimal Per-Open Counter for `myfirst`

You will add one small structure and one destructor. Nothing else in the driver changes shape.

```
struct myfirst_fh {
    struct myfirst_softc *sc;    /* back-pointer to the owning softc */
    uint64_t             reads; /* bytes this descriptor has read */
    uint64_t             writes; /* bytes this descriptor has written */
};
```

```
static void
myfirst_fh_dtor(void *data)
{
    struct myfirst_fh *fh = data;
    struct myfirst_softc *sc = fh->sc;

    mtx_lock(&sc->mtx);
    sc->active_fhs--;
    mtx_unlock(&sc->mtx);

    device_printf(sc->dev, "per-open dtor fh=%p reads=%lu writes=%lu\n",
        fh, (unsigned long)fh->reads, (unsigned long)fh->writes);

    free(fh, M_DEVBUF);
}
```

The destructor does three things worth noticing. It decrements `active_fhs` under the same mutex that guards the other `softc` counters, so the count stays consistent with what `d_open` saw when it opened the descriptor. It logs a line that matches the shape of the `open via ...` message, so every `open` in `dmesg` has a visibly paired destructor. And it frees the allocation last, after everything that might need to read from `fh` has already run.

In your `d_open`, allocate one of these and register it:

```
static int
myfirst_open(struct cdev *dev, int oflags, int devtype, struct thread *td)
{
    struct myfirst_softc *sc;
    struct myfirst_fh *fh;
    int error;

    sc = dev->si_drv1;
    if (sc == NULL || !sc->is_attached)
        return (ENXIO);

    fh = malloc(sizeof(*fh), M_DEVBUF, M_WAITOK | M_ZERO);
    fh->sc = sc;

    error = devfs_set_cdevpriv(fh, myfirst_fh_dtor);
    if (error != 0) {
        free(fh, M_DEVBUF);
        return (error);
    }

    mtx_lock(&sc->mtx);
    sc->open_count++;
}
```

```
    sc->active_fhs++;
    mtx_unlock(&sc->mtx);

    device_printf(sc->dev, "open via %s fh=%p (active=%d)\n",
        devtoname(dev), fh, sc->active_fhs);
    return (0);
}
```

Notice two things. First, the exclusive-open check from Chapter 7 is gone. With per-open state in place, there is no reason to refuse a second opener. If you do want exclusivity later, you can still add it back in; it is a separate decision. Second, the destructor will take care of the free. Your `d_close` does not need to touch `fh` at all.

In a handler that runs later, such as `d_read`, you retrieve the per-open structure:

```
static int
myfirst_read(struct cdev *dev, struct uio *uio, int ioflag)
{
    struct myfirst_fh *fh;
    int error;

    error = devfs_get_cdevpriv((void **)&fh);
    if (error != 0)
        return (error);

    /* Real read logic arrives in Chapter 9. For now, report EOF
     * and leave the counter untouched so userland tests can observe
     * that the descriptor owns its own state.
     */
    (void)fh;
    return (0);
}
```

The `(void)fh` silences the “unused variable” warning until Chapter 9 gives it work to do. That is fine. What matters for now is that your driver has a clean, working, cleanly destroyed per-file structure. From userland you can confirm the wiring by opening the device from two processes and watching the device-printf messages come through with two different `fh=` pointers.

8.8.2 What the Destructor Guarantees

Because the destructor does most of the work, it is worth being precise about when it runs and what state the world is in at that moment. Reading `devfs_destroy_cdevpriv` in `/usr/src/sys/fs/devfs/devfs_vnops.c` confirms the details.

- The destructor runs **exactly once per successful `devfs_set_cdevpriv` call**. If the function returned

EBUSY because the descriptor already had private data, the destructor for *your* data is never invoked; you must free the allocation yourself, as the example code does.

- The destructor runs **when the file descriptor is released**, not when your `d_close` is called. For an ordinary `close(2)`, the two moments are close together. For a process that exits while holding the descriptor, the descriptor is released as part of exit teardown; the destructor still runs. For a descriptor shared through `fork(2)` or passed through a UNIX-domain socket, the destructor runs only when the last reference drops.
- The destructor runs with no kernel lock held on your behalf. If your destructor touches softc state, take whatever lock the softc uses, just as the stage 2 example does when decrementing `active_fhs`.
- The destructor must not block for long. It is not a sleep-forever context, but it is also not an interrupt handler. Treat it like an ordinary kernel function and keep it short.

8.8.3 When `devfs_set_cdevpriv` Returns EBUSY

`devfs_set_cdevpriv` can fail in exactly one interesting way: the descriptor already has private data associated with it. That happens when something, typically your own code in a previous call, has already set a `cdevpriv` and you are trying to set another. The clean fix is to do the set once, early, and then read it back with `devfs_get_cdevpriv` wherever you need it.

Two cautions follow from this. The first is: do not call `devfs_set_cdevpriv` twice from the same open. The second is: when the call fails, free whatever you allocated before trying to set it. The example `myfirst_open` in this chapter follows both rules. When you port the pattern to your own driver, keep them in view.

8.8.4 When Not to Use `devfs_set_cdevpriv`

Per-open state is not the right home for everything. Keep device-wide state in the softc, reachable through `si_drv1`. Keep per-open state in the `cdevpriv` structure, reachable through `devfs_get_cdevpriv`. Mixing the two is the quickest way to write a driver that works in single-opener tests and falls apart when two processes show up at once.

`devfs_clear_cdevpriv(9)` exists, and you may see it in third-party code, but for most drivers the automatic cleanup through the destructor is sufficient. Reach for `devfs_clear_cdevpriv` only when you have a concrete reason, for instance a driver that can cleanly detach per-open state early in response to an `ioctl(2)`. If you are not sure you need it, you do not.

8.8.5 Inside `devfs_set_cdevpriv`: How the Machinery Works

The two functions you call look almost trivial from the outside. The machinery they drive is worth looking at once, because knowing the shape of it makes every edge case easier to reason about.

From `/usr/src/sys/fs/devfs/devfs_vnops.c`:

```
int
devfs_set_cdevpriv(void *priv, d_priv_dtor_t *priv_dtr)
{
    struct file *fp;
    struct cdev_priv *cdp;
```



```

    struct cdev_privdata *p;
    int error;

    fp = curthread->td_fpop;
    if (fp == NULL)
        return (ENOENT);
    cdp = cdev2priv((struct cdev *)fp->f_data);
    p = malloc(sizeof(struct cdev_privdata), M_CDEVpdata, M_WAITOK);
    p->cdpd_data = priv;
    p->cdpd_dtr = priv_dtr;
    p->cdpd_fp = fp;
    mtx_lock(&cdevpriv_mtx);
    if (fp->f_cdevpriv == NULL) {
        LIST_INSERT_HEAD(&cdp->cdp_fdpriv, p, cdpd_list);
        fp->f_cdevpriv = p;
        mtx_unlock(&cdevpriv_mtx);
        error = 0;
    } else {
        mtx_unlock(&cdevpriv_mtx);
        free(p, M_CDEVpdata);
        error = EBUSY;
    }
    return (error);
}

```

A short walk through the important bits:

- `curthread->td_fpop` is the file pointer for the current dispatch. `devfs` sets this up before it calls into your `d_open` and unsets it afterward. If you called `devfs_set_cdevpriv` from a context where no dispatch is active, `fp` would be `NULL` and the function would return `ENOENT`. In practice this only happens if you try to call it from the wrong context, for instance from a timer callback that is not bound to a file.
- A small record, `struct cdev_privdata`, is allocated from a dedicated malloc bucket `M_CDEVpdata`. It carries three fields: your pointer, your destructor, and a back-pointer to the `struct file`.
- Two threads entering this function at the same time for the same descriptor would be a disaster, so a single mutex `cdevpriv_mtx` guards the critical section. The check for `fp->f_cdevpriv == NULL` is what prevents double-registration: if a record is already attached, the new record is freed and `EBUSY` comes back.
- On success the record is inserted onto two lists: the descriptor's own pointer `fp->f_cdevpriv`, and the `cdev`'s list of all its descriptor-private records `cdp->cdp_fdpriv`. The first makes `devfs_get_cdevpriv` a one-pointer lookup. The second makes it possible for `devfs` to iterate every live record when the `cdev` is destroyed.

The destructor path is equally small:

```
void
devfs_destroy_cdevpriv(struct cdev_privdata *p)
{
    mtx_assert(&cdevpriv_mtx, MA_OWNED);
    KASSERT(p->cdpd_fp->f_cdevpriv == p,
        ("devfs_destroy_cdevpriv %p != %p",
            p->cdpd_fp->f_cdevpriv, p));
    p->cdpd_fp->f_cdevpriv = NULL;
    LIST_REMOVE(p, cdpd_list);
    mtx_unlock(&cdevpriv_mtx);
    (p->cdpd_dtr)(p->cdpd_data);
    free(p, M_CDEVPDATA);
}
```

Two things to notice. First, the destructor is called **with the mutex dropped**, so your destructor can take locks of its own without a risk of a deadlock against `cdevpriv_mtx`. Second, the record itself is freed immediately after your destructor returns, so a stale pointer into it would be a use-after-free. If your destructor stashes the pointer somewhere else, stash a copy of the data, not the record.

8.8.6 Interaction with fork, dup, and SCM_RIGHTS

File descriptors in UNIX have three common ways of multiplying: `dup(2)`, `fork(2)`, and passing through a UNIX-domain socket with `SCM_RIGHTS`. Each produces additional references to the same `struct file`. `devfs`'s `cdevpriv` machinery behaves consistently across all three.

After `dup(2)` or `fork(2)`, the new file descriptor refers to the **same** `struct file` as the original. The `cdevpriv` record is keyed on the `struct file`, not the descriptor number, so both descriptors share the record. Your destructor fires exactly once, when the last descriptor pointing at that file is released. That last-release can be an explicit `close(2)`, an implicit `exit(3)` that closes everything, or even a crash that terminates the process.

Passing the descriptor through `SCM_RIGHTS` is the same story from the `cdevpriv`'s point of view. The receiving process gets a new descriptor that points at the same `struct file`. The record stays attached; the destructor still fires only when the last reference drops, which may now be in the process on the other end of the socket.

This is usually exactly what you want, because it matches the user's mental model. One per-open state per conceptual open. If you ever need a different model, for instance a model where every `dup(2)`'d descriptor should have its own state, the solution is to set `D_TRACKCLOSE` on your `cdevsw` and allocate per-descriptor state inside `d_open` itself without using `devfs_set_cdevpriv`. That is unusual; ordinary drivers do not need it.

8.8.7 A Tour of Real Uses in the Tree

To cement the pattern, a short tour of three drivers that use `cdevpriv` in recognisable ways. You do not need to understand what each driver does as a whole; focus only on the device-file shape.

`/usr/src/sys/net/bpf.c` is the canonical example. Its `bpfopen` allocates a per-open descriptor, calls `devfs_set_cdevpriv(d, bpf_dtor)`, and sets up a small pile of counters and state. The destructor `bpf_dtor` tears down all of it: it detaches the descriptor from its BPF interface, frees counters, drains a select list, and drops a reference. The pattern is exactly the one this chapter has described, plus a lot of BPF-specific machinery that Part 6 will revisit.

`/usr/src/sys/fs/fuse/fuse_device.c` takes the same pattern and layers FUSE-specific state on top. The open allocates a struct `fuse_data`, registers it with `devfs_set_cdevpriv`, and every subsequent handler retrieves it with `devfs_get_cdevpriv`. The destructor tears down the FUSE session.

`/usr/src/sys/opencrypto/crypto_dev.c` uses `cdevpriv` for per-open crypto session state. Each open gets its own bookkeeping, and the destructor cleans it up.

These three drivers have almost nothing in common at the subsystem level: one is about packet capture, one about userland filesystems, one about hardware crypto offload. What they share is the device-file shape. The same three steps, in the same order, for the same reasons.

8.8.8 Patterns for What to Put in the Per-Open Structure

Now that you know the mechanics, the design question is what fields your per-open structure should hold. A few patterns recur across real drivers.

Counters. Bytes read, bytes written, calls made, errors reported. Each descriptor owns its own counter. `myfirst` in stage 2 already does this with `reads` and `writes`.

Read positions. If your driver exposes a seekable stream of bytes, the current offset belongs in the per-open structure, not in the `softc`. Two readers at different offsets are the reason.

Subscription handles. If the descriptor is reading events and a `poll` or `kqueue` needs to know whether there are more events pending for this specific descriptor, the subscription record belongs here. Chapter 10 uses this pattern.

Filter state. Drivers like BPF let each descriptor install a filter program. That program's compiled form is per-descriptor. Again, belongs in the per-open structure.

Reservations or tickets. If the driver hands out scarce resources (a hardware slot, a DMA channel, a shared buffer range) and ties them to an open, the record goes into per-open state. When the descriptor closes, the destructor frees the reservation automatically.

Credentials snapshots. Some drivers want to remember who opened the descriptor at the time of open, separately from whoever is currently doing a read or write on it. Capturing a snapshot of `td->td_ucred` at open time is a common pattern. Credentials are reference-counted (`crhold/crfree`), and the destructor is the right place to drop the reference.

Not every driver needs all of these. The list is a menu, not a checklist. When you design a driver, walk through it and ask “what information belongs to this specific open of the node?” The answers go into the per-open structure.

8.8.9 A Warning Against Cross-Referencing From the softc to Per-Open Records

A temptation that crops up with per-open state is for the softc to carry pointers back to per-open records, so that broadcasting an event to every descriptor becomes a simple list walk. The temptation is understandable; the implementation is full of edge cases. Two threads racing to close the last descriptor while a third is trying to broadcast is the scenario that breaks the straightforward code, and fixing it tends to require more locks than you want to add.

FreeBSD's answer is `devfs_foreach_cdevpriv(9)`, a callback-based iterator that walks the per-open records attached to a given cdev under the right lock. If you ever need the pattern, use that function and give it a callback. Do not maintain your own lists.

We will not use `devfs_foreach_cdevpriv` in Chapter 8. It is named here because if you scan the FreeBSD tree for `cdevpriv` you will find it, and you should recognise it as the safe alternative to reinventing the iteration yourself.

8.9 Destroying cdevs Safely

The act of putting a cdev into `devfs` is routine. Taking it out again is the part that needs thought. Chapter 7 taught you `destroy_dev(9)`, and for the simple path of a well-behaved driver it is all you need. Real drivers sometimes need more. This section walks through the family of destruction helpers, explains what they guarantee, and shows where each one is the right tool.

8.9.1 The Draining Model

Let us start from the question destruction has to answer: “when is it safe to free the softc and unload the module?” The naive answer, “after `destroy_dev` returns”, is almost right. The careful answer is, “after `destroy_dev` returns **and** no more kernel threads can be in any of my handlers for this cdev.”

The struct `cdev` counters you met earlier are how the kernel tracks this. `si_threadcount` increments every time `devfs` enters one of your handlers on behalf of a user syscall, and decrements every time the handler returns. `destroy_dev1`, which is the internal function that `destroy_dev` calls, watches that counter. Here is the relevant excerpt from `/usr/src/sys/kern/kern_conf.c`:

```
while (csw != NULL && csw->d_purge != NULL && dev->si_threadcount) {
    csw->d_purge(dev);
    mtx_unlock(&cdp->cdp_threadlock);
    msleep(csw, &devmtx, PRIBIO, "devprg", hz/10);
    mtx_lock(&cdp->cdp_threadlock);
    if (dev->si_threadcount)
        printf("Still %lu threads in %s\n",
            dev->si_threadcount, devtoname(dev));
}
while (dev->si_threadcount != 0) {
    /* Use unique dummy wait ident */
    mtx_unlock(&cdp->cdp_threadlock);
```

```
msleep(&csw, &devmtx, PRIBIO, "devdrn", hz / 10);  
mtx_lock(&cdp->cdp_threadlock);  
}
```

Two loops. The first loop calls `d_purge` if the driver provides one; the second simply waits. In both cases the result is the same: `destroy_dev` does not return until `si_threadcount` is zero. This is the **draining** behavior that makes destruction safe. When the call returns, no thread is inside any handler, and no new thread can enter one, because `si_devsw` has been cleared.

What this means for your code: **after `destroy_dev(sc->cdev)` returns, nothing in user space can trigger a call into your handlers with this `cdev`.** You are free to destroy softc members that those handlers depend on.

8.9.2 The Four Destruction Functions

FreeBSD exposes four related functions for `cdev` destruction. Each handles a slightly different case.

`destroy_dev(struct cdev *dev)`

The ordinary case. Synchronous: waits for in-flight handlers to finish, then unlinks the `cdev` from `devfs` and releases the kernel's primary reference. Used in Chapter 7 and in every single-threaded destruction path in this book. Requires the caller to be sleepable and to not hold any lock that the in-flight handlers might need.

`destroy_dev_sched(struct cdev *dev)`

A deferred form. Schedules the destruction on a taskqueue and returns immediately. Useful when the calling context cannot sleep, for instance from inside a callback that runs under a lock. The actual destruction happens asynchronously, and the caller must not assume it has completed when the function returns.

`destroy_dev_sched_cb(struct cdev *dev, void (*cb)(void *), void *arg)`

The same deferred form, but with a callback that runs after destruction completes. Use when you need to do follow-up work (free the softc, for example) once you know the `cdev` is truly gone.

`destroy_dev_drain(struct cdevsw *csw)`

The sweep. Waits for **every** `cdev` registered against the given `cdevsw` to be fully destroyed, including those scheduled through the deferred forms. Used when you are about to unregister or free the `cdevsw` itself, for instance inside the `MOD_UNLOAD` handler of a module that ships multiple drivers.

8.9.3 The Race That `destroy_dev_drain` Exists to Prevent

Draining is a subtle point, and the best way to explain it is with the scenario it fixes.

Suppose your module exports a `cdevsw`. In `MOD_UNLOAD`, your code calls `destroy_dev(sc->cdev)` and then returns success. The kernel proceeds to tear the module down. Everything looks fine, until a moment later when a deferred task scheduled earlier through `destroy_dev_sched` finally runs. That task dereferences the `struct cdevsw` as part of its cleanup. The `cdevsw` has been unmapped along with the module. The kernel panics.

The race is narrow but real. `destroy_dev_drain` is the fix: call it on your `cdevsw` after you are confident no new `cdevs` will be created, and it will not return until every `cdev` registered against that `cdevsw` has completed its destruction. Only then is it safe to let the module go.

If your driver creates one `cdev` from `attach`, destroys it from `detach`, and never uses the deferred forms, you do not need `destroy_dev_drain`. `myfirst` does not need it. Real drivers that manage cloning `cdevs` or that destroy `cdevs` from event handlers usually do.

8.9.4 The Order of Operations in detach

Given all this, the right order of operations in a `detach` handler for a driver with a primary `cdev`, an alias, and per-open state is:

1. Refuse to detach if any descriptors are still open. Return `EBUSY`. Your `active_fhs` counter is the right thing to check.
2. Destroy the alias `cdev` with `destroy_dev(sc->cdev_alias)`. This unlinks the alias from `devfs` and drains any in-flight calls against it.
3. Destroy the primary `cdev` with `destroy_dev(sc->cdev)`. Same as above for the primary.
4. Tear down the `sysctl` tree with `sysctl_ctx_free(9)`.
5. Destroy the mutex with `mtx_destroy(9)`.
6. Clear the `is_attached` flag in case anything still reads it.
7. Return zero.

Notice that steps 2 and 3 serve two purposes each. They remove the nodes from `devfs` so no new opens can arrive, and they drain in-flight calls so no handler is still running when step 4 tries to free state that a handler would read.

The pattern is simple. The only way to get it wrong is to free something before the draining `destroy_dev` has finished with it. Stick to this order and you will be safe.

8.9.5 Unload Under Load

A healthy intuition-building exercise is to reason about what happens when `kldunload` arrives while a userland program is inside a `read(2)` on your device.

Walk through the timeline:

- The kernel begins unloading the module. It calls your `MOD_UNLOAD` handler, which ultimately calls `device_delete_child` on your Newbus device, which invokes your `detach`.
- Your `detach` reaches `destroy_dev(sc->cdev)`. This call is synchronous and will wait for in-flight handlers to finish.
- The userland `read(2)` is currently executing your `d_read`. `si_threadcount` is 1.
- `destroy_dev` sleeps, watching `si_threadcount`.
- Your `d_read` returns. `si_threadcount` drops to 0.
- `destroy_dev` returns. Your `detach` continues with `sysctl` and mutex teardown.
- The userland `read(2)` has already returned its bytes to user space. The descriptor is still open.
- A subsequent `read(2)` on the same descriptor from the same process now fails cleanly, because the `cdev` is gone.

This is what “destroy the node first, then tear down its dependencies” buys you. The window where userland could observe inconsistent state is made vanishingly small by the kernel’s draining behavior.

8.9.6 When `destroy_dev` Refuses to Proceed Quickly

Sometimes `destroy_dev` will sit in its drain loop for longer than you expect. There are a few common reasons.

- A handler is blocked in a sleep that no wakeup is going to release. For instance, a `d_read` that calls `msleep(9)` on a condition variable that nobody ever signals. In that case your driver has a logic bug. Destruction is doing exactly the right thing by refusing to proceed; your job is to wake the blocked thread, either by writing a `d_purge` handler that nudges it or by ensuring the control flow that eventually wakes it is still functional during unload.
- A userland program is stuck in a `ioctl` that is waiting on hardware. The fix is usually the same: a `d_purge` handler that tells the blocked thread to abandon the wait.
- Two destructions racing, each draining the other. That is the case `destroy_dev_drain` exists for.

If you watch `dmesg` during a stuck unload, you will see the “Still N threads in foo” message printed by the loop above. That is your cue: find out what those threads are doing and convince them to return.

8.9.7 A Minimal `d_purge` Example

For completeness, here is what a simple `d_purge` looks like. Not every driver needs one; it is worth showing so you recognise the shape when you read real code:

```
static void
myfirst_purge(struct cdev *dev)
{
    struct myfirst_softc *sc = dev->si_drv1;

    mtx_lock(&sc->mtx);
    sc->shutting_down = 1;
    wakeup(&sc->rx_queue);    /* nudge any thread waiting on us */
    mtx_unlock(&sc->mtx);
}

static struct cdevsw myfirst_cdevsw = {
    .d_version = D_VERSION,
    .d_purge   = myfirst_purge,
    /* ... */
};
```

The function is called from inside `destroy_dev1` while `si_threadcount` is still non-zero. Its job is to poke any blocked kernel thread inside your handlers so that the thread observes the shutdown and returns. For drivers that do only blocking reads, setting a shutdown flag and issuing a `wakeup(9)` is usually all that is needed. Chapter 10 revisits this when blocking I/O becomes a first-class topic.

8.9.8 Summary

Destruction of a cdev is a rehearsed dance between three participants: your driver, devfs, and any userland threads currently mid-call. The guarantees are strong if you use the right tools:

- Use `destroy_dev` in ordinary paths. Let it drain.
- Use the deferred forms when you cannot sleep or when you need a callback after destruction.
- Use `destroy_dev_drain` when you are about to free or unregister a cdevsw.
- Destroy the node before anything the node's handlers depend on.
- Provide a `d_purge` if your handlers can block indefinitely.

Beyond that, the detail is the kind of thing you look up when you need it. The shape is the part that matters, and the shape is simple.

8.10 Persistent Policy: `devfs.conf` and `devfs.rules`

Your driver fixes the **baseline** mode, owner, and group of each node. Persistent operator-side adjustments belong in `/etc/devfs.conf` and `/etc/devfs.rules`. Both files are standard parts of the FreeBSD base system, and both apply to every devfs mount on the host.

8.10.1 `devfs.conf`: one-time, per-path adjustments

`devfs.conf` is the simplest tool. Each line applies a one-shot adjustment when a matching device node appears. The format is documented in `devfs.conf(5)`. The common directives are `own`, `perm`, and `link`:

```
# /etc/devfs.conf
#
# Adjustments applied once when each node appears.

own      myfirst/0      root:operator
perm     myfirst/0      0660
link     myfirst/0      myfirst-primary
```

Those three lines say: every time `/dev/myfirst/0` shows up, chown it to `root:operator`, set its mode to `0660`, and create a symlink named `/dev/myfirst-primary` that points at it. Restart the devfs service to apply changes on a running system:

```
% sudo service devfs restart
```

`devfs.conf` is fine for small, stable lab setups. It is not a policy engine. If you need conditional rules or jail-specific filtering, reach for `devfs.rules`.

8.10.2 devfs.rules: rule-based, used by jails

`devfs.rules` describes named rulesets; each ruleset is a list of patterns and actions. A jail references a ruleset by name in its `jail.conf(5)`, and when the jail's own `devfs` mount comes up, the kernel walks the matching ruleset and filters the node set accordingly. The format is documented in `devfs(8)` and `devfs.rules(5)`.

A tiny example:

```
# /etc/devfs.rules

[myfirst_lab=10]
add path 'myfirst/*' unhide
add path 'myfirst/*' mode 0660 group operator
```

This defines a ruleset numbered 10, named `myfirst_lab`. It unhides any node under `myfirst/` (jails hide nodes by default), and then sets them group-readable and group-writable by operator. To use the ruleset, name it in `jail.conf`:

```
devfs_ruleset = 10;
```

We are not going to set up a jail in this chapter. The point here is recognition: when you see `devfs_ruleset` in a jail configuration or `service devfs restart` in operator documentation, you are looking at policy that sits on top of whatever your driver exposed, not inside it. Keep your driver honest at the baseline, and let these files shape what the operator allows.

8.10.3 The Full devfs.conf Syntax

`devfs.conf` has a small, stable grammar. Each line is a directive. Blank lines and lines that begin with `#` are ignored. A `#` anywhere on a line starts a comment to end-of-line. Only three directive keywords exist:

- **own path user[:group]**: change the ownership of `path` to `user`, and, if `:group` is given, to that group as well. The user and group can be names that exist in the password database or numeric ids.
- **perm path mode**: change the mode of `path` to the given octal mode. Leading zero is optional but conventional.
- **link path linkname**: create a symbolic link at `/dev/linkname` pointing at `/dev/path`.

Each directive operates on the node whose path is given relative to `/dev`. The path can name a device directly, or it can name a glob that matches a family of devices. Glob characters are `*`, `?`, and character classes in brackets.

The action is applied when the node first appears under `/dev`. For nodes that exist at boot, that means during the early `service devfs start` phase. For nodes that appear later, as when a driver module loads, the action is applied when the matching `cdev` is added to `devfs`.

The effect of `service devfs restart` on a running system is to re-run every directive in `/etc/devfs.conf` against whatever is currently under `/dev`. This is how you apply a newly-added directive to devices that already exist.

8.10.4 Reading the Example devfs.conf

The FreeBSD base system ships a commented-out example at `/etc/defaults/devfs.conf` (or equivalent path; the file is installed by the base system). The upstream source at `/usr/src/sbin/devfs/devfs.conf` is instructive:

```
# Commonly used by many ports
#link    cd0      cdrom
#link    cd0      dvd

# Allow a user in the wheel group to query the smb0 device
#perm    smb0     0660

# Allow members of group operator to cat things to the speaker
#own     speaker  root:operator
#perm    speaker  0660
```

The file is mostly comments, which is the expected state: FreeBSD installs with no `devfs.conf` directives active. Operators who need host-specific changes add them to `/etc/devfs.conf`. This chapter's Lab 8.4 adds entries for `myfirst/0`.

8.10.5 Worked Examples That Recur in Practice

Three `devfs.conf` patterns come up often. Each is worth showing once.

Granting a monitoring tool access to a status node. Suppose a driver exposes `/dev/something/status` as `root:wheel 0600`, and you want a local monitoring tool that runs as a non-root user to read it. The simplest answer is a dedicated group:

```
# /etc/devfs.conf
own     something/status      root:monitor
perm    something/status      0640
```

After service `devfs` restart, the node is readable by members of the `monitor` group. Create the group with `pw groupadd monitor` and add the relevant users.

Providing a stable, convenient name for a renumbered device. Suppose the driver used to create `/dev/old_name` and now creates `/dev/new_name/0`, and you have scripts that still reference the old path. A `link` directive preserves compatibility:

```
link    new_name/0          old_name
```

`/dev/old_name` becomes a symbolic link to `/dev/new_name/0`. The link carries no policy of its own; ownership and mode come from the target.

Widening a narrow default for a lab system. Suppose a driver defaults to `root:wheel 0600` and you want a lab machine where the local admin user can interact with the node without `sudo`. Rather than modifying the driver, give the local admin the operator group and widen the mode in `devfs.conf`:

```
own      myfirst/0      root:operator
perm     myfirst/0      0660
```

This is exactly the Lab 8.4 setup. It stays contained to the lab machine and leaves the driver's shipping default intact.

8.10.6 devfs.rules in Depth

`devfs.rules` is a different beast. Rather than applying one-shot directives to a path, it defines **named rulesets** that a `devfs` mount can reference. Each ruleset is a list of rules; each rule matches paths by pattern and applies actions.

The file lives at `/etc/devfs.rules` and the base system ships defaults at `/etc/defaults/devfs.rules`. The format is documented in `devfs.rules(5)` and `devfs(8)`.

A ruleset is introduced by a bracketed header:

```
[rulesetname=number]
```

`number` is a small integer, the way `devfs` identifies the ruleset internally. `rulesetname` is a human-readable tag for use in jail configuration. Rules following a header belong to that ruleset until the next header.

A rule begins with the `add` keyword and names a path pattern and an action. The common actions are:

- **unhide**: make matching nodes visible. Rulesets that derive from `devfsrules_hide_all` use this to whitelist a specific set of nodes.
- **hide**: make matching nodes invisible. Used to remove something from the default set.
- **group name**: change the group of matching nodes.
- **user name**: change the owner.
- **mode N**: change the mode to octal `N`.
- **include \$name**: include the rules from another ruleset named `$name`.

Include directives are how FreeBSD's shipping rulesets compose. The `devfsrules_jail` ruleset begins with `add include $devfsrules_hide_all` to establish a clean slate, then includes `devfsrules_unhide_basic` for the handful of nodes every reasonable program expects, then `devfsrules_unhide_login` for the PTYs and standard descriptors, then adds a few jail-specific paths on top.

8.10.7 Reading the Default Rulesets

The FreeBSD source ships `/etc/defaults/devfs.rules` (installed from `/usr/src/sbin/devfs/devfs.rules`). Reading it once gives you a model of how rules are layered.

```
[devfsrules_hide_all=1]
add hide
```

Ruleset 1 hides every node in devfs. It is the starting point for jail rulesets that need to whitelist.

```
[devfsrules_unhide_basic=2]
add path null unhide
add path zero unhide
add path crypto unhide
add path random unhide
add path urandom unhide
```

Ruleset 2 unhides the basic pseudo-devices every reasonable process expects.

```
[devfsrules_unhide_login=3]
add path 'ptyp*' unhide
add path 'ptyq*' unhide
/* ... many more PTY paths ... */
add path ptmx unhide
add path pts unhide
add path 'pts/*' unhide
add path fd unhide
add path 'fd/*' unhide
add path stdin unhide
add path stdout unhide
add path stderr unhide
```

Ruleset 3 unhides the TTY and file-descriptor infrastructure that logged-in users depend on.

```
[devfsrules_jail=4]
add include $devfsrules_hide_all
add include $devfsrules_unhide_basic
add include $devfsrules_unhide_login
add path fuse unhide
add path zfs unhide
```

Ruleset 4 composes the three previous rulesets and adds fuse and zfs. This is the default ruleset most jails use.

```
[devfsrules_jail_vnet=5]
add include $devfsrules_hide_all
add include $devfsrules_unhide_basic
```

```
add include $devfsrules_unhide_login
add include $devfsrules_jail
add path pf unhide
```

Ruleset 5 is ruleset 4 plus the packet-filter control node. Used by jails that need pf to be reachable.

Reading these carefully is a good investment. Every pattern you will need for your own rulesets is in one of them.

8.10.8 A Complete Jail Example End to End

To ground the theory, here is a complete example that a reader can apply on a lab system. It assumes you have built and loaded the Chapter 8 stage 2 driver and that `/dev/myfirst/0` exists on the host.

Step 1: define a ruleset in `/etc/devfs.rules`. Add to the end of the file:

```
[myfirst_jail=100]
add include $devfsrules_jail
add path 'myfirst' unhide
add path 'myfirst/*' unhide
add path 'myfirst/*' mode 0660 group operator
```

The ruleset is numbered 100 (any unused small integer works; 100 is safely above the shipped numbers). It includes the default jail ruleset so the jail still has `/dev/null`, `/dev/zero`, the PTYs, and everything else a normal jail needs. Then it unhides the `myfirst/` directory and the nodes inside it, and sets their mode and group.

Step 2: create a jail. A minimal `/etc/jail.conf` entry:

```
myfirstjail {
    path = "/jails/myfirstjail";
    host.hostname = "myfirstjail.example.com";
    mount.devfs;
    devfs_ruleset = 100;
    exec.start = "/bin/sh";
    exec.stop = "/bin/sh -c 'exit'";
    persist;
}
```

Create the jail root:

```
% sudo mkdir -p /jails/myfirstjail
% sudo bsdinstall jail /jails/myfirstjail
```

Replace `bsdinstall` with whatever jail-creation method fits your lab if you already have one.

Step 3: start the jail and inspect.

```
% sudo service devfs restart
% sudo service jail start myfirstjail
% sudo jexec myfirstjail ls -l /dev/myfirst
total 0
crw-rw----  1 root  operator  0x5a Apr 17 10:00 0
```

The node appears inside the jail with the ruleset-specified ownership and mode. If the ruleset did not unhide it, the jail would see no `myfirst` directory at all.

Step 4: prove it. Comment out the `add path 'myfirst/*' unhide` line in `/etc/devfs.rules`, run `sudo service devfs restart`, and re-enter the jail:

```
% sudo jexec myfirstjail ls -l /dev/myfirst
ls: /dev/myfirst: No such file or directory
```

The node is invisible to the jail. The host still sees it. The driver has not been reloaded. Policy in the file completely determines what the jail sees.

This end-to-end exercise is what Lab 8.7 walks through. The point of showing it once in prose is to establish the pattern: **rulesets shape what the jail sees, and the driver does nothing different regardless**. Your driver's job is to expose a sound baseline; the rulesets' job is to filter and adjust per environment.

8.10.9 Debugging Rule Mismatches

When `devfs` does not present a node the way you expect, there are a few tools for diagnosing why.

- **devfs rule show** displays the rulesets currently loaded into the kernel. You can compare them against the file.
- **devfs rule showsets** lists the ruleset numbers currently active.
- **devfs rule ruleset NUM** followed by **devfs rule show** displays rules inside a specific ruleset.
- **service devfs restart** reapplies `/etc/devfs.conf` and reloads all rulesets from `/etc/devfs.rules`. Use this any time you change either file.

Common failure modes:

- A rule uses a path pattern that does not match. Single-quote your glob patterns and remember that `my-first/*` is different from `myfirst` (the directory itself is not covered by the `/*` pattern; you need both rules).
- A ruleset is referenced by a jail but not present in `/etc/devfs.rules`. The jail's `/dev` ends up with no matching rules applied, which often means a hidden-by-default filesystem.
- A ruleset is present but never restarted. After you add a rule, run `service devfs restart` to actually push it into the kernel.

8.10.10 Runtime Manipulation With `devfs(8)`

`devfs(8)` is the low-level administrative tool that `service devfs restart` uses under the hood. You can invoke it directly to apply changes without rebooting or restarting the whole `devfs` subsystem.

```
% sudo devfs rule -s 100 add path 'myfirst/*' unhide
```

This adds a rule to ruleset 100 in the running kernel, without touching the file. Useful for experimentation. Rules added this way do not persist across reboots.

```
% sudo devfs rule showsets
0 1 2 3 4 5 100
```

Shows which ruleset numbers are currently loaded.

```
% sudo devfs rule -s 100 del 1
```

Deletes rule number 1 in ruleset 100.

You will rarely need to touch `devfs(8)` directly in production; the file-based workflow and `service devfs restart` are enough for most needs. During the debugging of a stubborn ruleset, the interactive command is invaluable.

8.10.11 A Caveat About `devfs.conf` Timing

One pattern that bites newcomers is this: you add a `devfs.conf` line, reboot, and discover the line did not take effect. Usually the reason is ordering. `service devfs start` runs early in the boot sequence, before some modules load. Nodes created later by modules loaded later will not be matched by the already-run directives unless you restart the `devfs` service after the module loads.

In practice this means:

1. If your driver is compiled into the kernel, its nodes exist at the time `devfs.conf` is applied. The directives take effect on first boot.
2. If your driver is loaded from `/boot/loader.conf`, its nodes exist before userland starts, so directives are applied normally.
3. If your driver is loaded later (from `rc.d` or by hand), you must run `service devfs restart` after the load so the directives apply to the newly-appeared nodes.

For labs, the last case is the common one. Load the driver, run `service devfs restart`, then verify.

8.11 Exercising Your Device From Userland

Shell tools will get you surprisingly far. You already know these from Chapter 7:

```
% ls -l /dev/myfirst/0
% sudo cat </dev/myfirst/0
% echo "hello" | sudo tee /dev/myfirst/0 >/dev/null
```

They remain useful, especially `ls -l` for confirming that a permission change took effect. But at some point you will want to open the device from a program you wrote yourself, so you can control timing, measure behavior, and simulate realistic user code. The companion files under `examples/part-02/ch08-working-with-device-files/userland/` contain a small probe program that does exactly that. The relevant pieces look like this:

```
#include <err.h>
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>

int
main(int argc, char **argv)
{
    const char *path = (argc > 1) ? argv[1] : "/dev/myfirst/0";
    char buf[64];
    ssize_t n;
    int fd;

    fd = open(path, O_RDWR);
    if (fd < 0)
        err(1, "open %s", path);

    n = read(fd, buf, sizeof(buf));
    if (n < 0)
        err(1, "read %s", path);

    printf("read %zd bytes from %s\n", n, path);

    if (close(fd) != 0)
        err(1, "close %s", path);

    return (0);
}
```

Two things to notice. First, there is nothing device-specific in the code. It is the same `open`, `read`, `close` you would write against a regular file. That is the UNIX tradition paying off. Second, compiling and running this program gives you a repeatable way to drive your driver without shell quoting to worry about. In Chapter 9 you will extend it to write data, measure byte counts, and compare per-open state across descriptors.

Running it once against your Stage 2 driver should produce something like:


```
% cc -Wall -Werror -o probe_myfirst probe_myfirst.c
% sudo ./probe_myfirst
read 0 bytes from /dev/myfirst/0
```

Zero bytes, because `d_read` still returns EOF. The number is boring; the fact that the whole path worked is not.

8.11.1 A Second Probe: Inspecting With `stat(2)`

Reading the metadata of a device node is just as instructive as opening it. FreeBSD's `stat(1)` command and the `stat(2)` system call both report what devfs is advertising. A tiny program built around `stat(2)` makes it easy to compare a primary node and an alias and confirm that they resolve to the same cdev.

The companion source `examples/part-02/ch08-working-with-device-files/userland/stat_myfirst.c` looks like this:

```
#include <err.h>
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>

int
main(int argc, char **argv)
{
    struct stat sb;
    int i;

    if (argc < 2) {
        fprintf(stderr, "usage: %s path [path ...]\n", argv[0]);
        return (1);
    }

    for (i = 1; i < argc; i++) {
        if (stat(argv[i], &sb) != 0)
            err(1, "stat %s", argv[i]);
        printf("%s: mode=%06o uid=%u gid=%u rdev=%#jx\n",
            argv[i],
            (unsigned)(sb.st_mode & 07777),
            (unsigned)sb.st_uid,
            (unsigned)sb.st_gid,
            (uintmax_t)sb.st_rdev);
    }
    return (0);
}
```

Running it against both the primary node and the alias should show the same rdev on both paths:

```
% sudo ./stat_myfirst /dev/myfirst/0 /dev/myfirst
/dev/myfirst/0: mode=020660 uid=0 gid=5 rdev=0x5a
/dev/myfirst:   mode=020660 uid=0 gid=5 rdev=0x5a
```

rdev is the identifier devfs uses to tag the node, and it is the simplest proof that two names really refer to the same underlying cdev. The 020000 high bits in the mode say “character special file”; the low bits are the familiar 0660.

8.11.2 A Third Probe: Parallel Opens

The stage 2 driver lets multiple processes hold the device open simultaneously, and each gets its own per-open structure. A good way to confirm the wiring is to run a program that opens the node several times from within the same process, holds every descriptor for a moment, and reports what happened.

The companion source `examples/part-02/ch08-working-with-device-files/userland/parallel_probe.c` does exactly that:

```
#include <err.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define MAX_FDS 8

int
main(int argc, char **argv)
{
    const char *path = (argc > 1) ? argv[1] : "/dev/myfirst/0";
    int fds[MAX_FDS];
    int i, n;

    n = (argc > 2) ? atoi(argv[2]) : 4;
    if (n < 1 || n > MAX_FDS)
        errx(1, "count must be 1..%d", MAX_FDS);

    for (i = 0; i < n; i++) {
        fds[i] = open(path, O_RDWR);
        if (fds[i] < 0)
            err(1, "open %s (fd %d of %d)", path, i + 1, n);
        printf("opened %s as fd %d\n", path, fds[i]);
    }
}
```

```

    printf("holding %d descriptors; press enter to close\n", n);
    (void) getchar();

    for (i = 0; i < n; i++) {
        if (close(fds[i]) != 0)
            warn("close fd %d", fds[i]);
    }
    return (0);
}

```

Run it and watch dmesg at the same time:

```

% sudo ./parallel_probe /dev/myfirst/0 4
opened /dev/myfirst/0 as fd 3
opened /dev/myfirst/0 as fd 4
opened /dev/myfirst/0 as fd 5
opened /dev/myfirst/0 as fd 6
holding 4 descriptors; press enter to close

```

You should see four open via `myfirst/0 fh=<ptr> (active=N)` lines in dmesg, each with a different pointer. When you press Enter, four per-open dtor `fh=<ptr>` lines follow as each descriptor closes. This is your strongest evidence that per-open state really is per-descriptor.

8.11.3 A Fourth Probe: Stress Testing

A short stress test exercises the destructor path repeatedly and catches leaks that a single-open test would miss. `examples/part-02/ch08-working-with-device-files/userland/stress_probe.c` loops open-and-close:

```

#include <err.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int
main(int argc, char **argv)
{
    const char *path = (argc > 1) ? argv[1] : "/dev/myfirst/0";
    int iters = (argc > 2) ? atoi(argv[2]) : 1000;
    int i, fd;

    for (i = 0; i < iters; i++) {

```

```

        fd = open(path, O_RDWR);
        if (fd < 0)
            err(1, "open (iter %d)", i);
        if (close(fd) != 0)
            err(1, "close (iter %d)", i);
    }
    printf("%d iterations completed\n", iters);
    return (0);
}

```

Run it against the loaded driver and then verify that the active-open counter returns to zero:

```

% sudo ./stress_probe /dev/myfirst/0 10000
10000 iterations completed
% sysctl dev.myfirst.0.stats.active_fhs
dev.myfirst.0.stats.active_fhs: 0
% sysctl dev.myfirst.0.stats.open_count
dev.myfirst.0.stats.open_count: 10000

```

If `active_fhs` settles above zero after the program exits, your destructor is failing to run on some path and you have a real leak to investigate. If `open_count` matches the iteration count, every open was seen. The stress probe is a blunt instrument but it is fast and catches most common mistakes.

8.11.4 Observing devd Events

`devd(8)` is the userland daemon that reacts to device events. Each time a `cdev` appears or disappears, `devd` receives a notification and can run a configured action. You do not need to configure `devd` to see its events; you can subscribe to the event stream directly through the `/var/run/devd.pipe` socket.

A short helper, `examples/part-02/ch08-working-with-device-files/userland/devd_watch.sh`, wires this up:

```

#!/bin/sh
# Print devd events related to the myfirst driver.
nc -U /var/run/devd.segpacket.pipe | grep -i 'myfirst'

```

Run this in one terminal, then load and unload the driver in another:

```

% sudo sh ./devd_watch.sh &
% sudo kldload ./myfirst.ko
!system=DEVFS subsystem=CDEV type=CREATE cdev=myfirst/0
% sudo kldunload myfirst
!system=DEVFS subsystem=CDEV type=DESTROY cdev=myfirst/0

```

You should see CREATE and DESTROY notifications with the cdev name. This is how operators build automatic reactions: a devd rule in `/etc/devd.conf` can match on these events and run a script in response. Challenge 5 at the end of this chapter asks you to write a minimal `devd.conf` rule that logs `myfirst` events.

8.11.5 Handling errno From Device Operations

A good userland probe does more than just call `open` and `exit`. It distinguishes between the different `errno` values the kernel returns and takes sensible action on each. The probes in this chapter all use `err(3)` to print a readable message and `exit`, which is right for a small experimental tool. Production userland code over device nodes usually looks more like this:

```
fd = open(path, O_RDWR);
if (fd < 0) {
    switch (errno) {
        case ENXIO:
            /* Driver not ready yet. Retry or report. */
            warnx("%s: driver not ready", path);
            break;
        case EBUSY:
            /* Node is exclusive and already open. */
            warnx("%s: in use by another process", path);
            break;
        case EACCES:
            /* Permission denied. */
            warnx("%s: permission denied", path);
            break;
        case ENOENT:
            /* Node does not exist. Is the driver loaded? */
            warnx("%s: not present", path);
            break;
        default:
            warn("%s", path);
            break;
    }
    return (1);
}
```

That table of `errno` values is worth internalising. Chapter 8 Section 13 treats it as a first-class topic, because the same values appear on the driver side and deciding which one to return is a design choice.

8.11.6 Driving Devices From Shell Scripts

Before you reach for a userland C program, remember that shell tools cover many cases adequately. For `myfirst`, a handful of one-liners are useful:

```
# Verify the node exists and report its ownership and mode.
ls -l /dev/myfirst/0

# Open and immediately close the device, for probe purposes.
sudo sh -c ': </dev/myfirst/0'

# Read once from the device and pipe the result into hexdump.
sudo cat /dev/myfirst/0 | hexdump -C | head

# Hold the device open for a few seconds with a background shell.
sudo sh -c 'exec 3</dev/myfirst/0; sleep 5; exec 3<&-'

# Show what processes currently have the device open.
sudo fstat /dev/myfirst/0
```

Each of these is a separate debugging move, and together they make up a useful shell toolkit. When you can express your test in shell, shell is often the fastest path.

8.12 Reading Real FreeBSD Drivers Through a Device-File Lens

Nothing cements the device-file model like reading drivers that have had to solve the same problems you are now solving. This section is a guided tour of three drivers in `/usr/src/sys`. The goal is not to understand each driver end to end. The goal is to see how each of them shapes its device file, so you build a library of patterns in your head.

Each walkthrough follows the same shape: open the file, find the `cdevsw`, find the `make_dev` call, find the `destroy_dev` call, notice what is idiomatic and what is unusual.

8.12.1 Walkthrough 1: `/usr/src/sys/dev/null/null.c`

The `null` module is the smallest good example in the tree. Open it in an editor. It is short enough to read in a single pass.

What to notice first: there are **three** `cdevsw` structures in one file.

```
static struct cdevsw full_cdevsw = {
    .d_version =    D_VERSION,
    .d_read =      zero_read,
    .d_write =     full_write,
    .d_ioctl =     zero_ioctl,
    .d_name =      "full",
};

static struct cdevsw null_cdevsw = {
```

```

        .d_version =    D_VERSION,
        .d_read =      (d_read_t *)nullop,
        .d_write =     null_write,
        .d_ioctl =     null_ioctl,
        .d_name =      "null",
};

static struct cdevsw zero_cdevsw = {
        .d_version =    D_VERSION,
        .d_read =      zero_read,
        .d_write =     null_write,
        .d_ioctl =     zero_ioctl,
        .d_name =      "zero",
        .d_flags =     D_MMAP_ANON,
};

```

Three distinct nodes, three distinct cdevsw values, no softc. The module registers three cdevs in its MOD_LOAD handler:

```

full_dev = make_dev_credf(MAKEDEV_ETERNAL_KLD, &full_cdevsw, 0,
        NULL, UID_ROOT, GID_WHEEL, 0666, "full");
null_dev = make_dev_credf(MAKEDEV_ETERNAL_KLD, &null_cdevsw, 0,
        NULL, UID_ROOT, GID_WHEEL, 0666, "null");
zero_dev = make_dev_credf(MAKEDEV_ETERNAL_KLD, &zero_cdevsw, 0,
        NULL, UID_ROOT, GID_WHEEL, 0666, "zero");

```

Notice MAKEDEV_ETERNAL_KLD. When this code is statically compiled into the kernel, the macro expands to MAKEDEV_ETERNAL and marks the cdevs as never-to-be-destroyed. When the same code is built as a loadable module, the macro expands to zero, and the cdevs can be destroyed during unload.

Also notice that mode 0666 and root:wheel. Everything the null module serves is intentionally accessible to everyone.

Unload reads as simply as load:

```

destroy_dev(full_dev);
destroy_dev(null_dev);
destroy_dev(zero_dev);

```

One destroy_dev per cdev. No softc to tear down. No per-open state. No locking beyond what the kernel provides. This is what minimal looks like.

What to copy from null: the habit of setting d_version, the habit of giving each cdevsw its own d_name, the symmetry between load and unload, the willingness to use simple named handlers instead of inventing abstractions.

What to leave alone: `MAKEDEV_ETERNAL_KLD`. Your driver should be unloadable, so you do not want the eternal flag. The `null` module is special because the nodes it creates predate almost every other kernel subsystem and are expected to remain live for the life of the kernel.

8.12.2 Walkthrough 2: `/usr/src/sys/dev/led/led.c`

The LED framework is a step up in structural complexity. It is still small enough to read in a sitting. Where `null` has no softc, `led` has a full per-LED softc. Where `null` creates three singletons, `led` creates one cdev per LED on demand.

Look first at the single cdevsw:

```
static struct cdevsw led_cdevsw = {
    .d_version =    D_VERSION,
    .d_write =      led_write,
    .d_name =       "LED",
};
```

One cdevsw for all LEDs. The framework uses it for every cdev it creates, relying on `si_drv1` to distinguish between them. The minimalism of this definition is itself a lesson: `led` does not implement `d_open`, `d_close`, or `d_read`, because every interaction an operator has with a LED is a pattern string written with `echo`. Reading from the node is not meaningful, and no session state needs to be tracked on open, so the driver simply leaves those fields unset. `devfs` interprets each `NULL` slot as “use the default behaviour”, which for `d_read` is returning zero bytes and for `d_open` and `d_close` is doing nothing. Keep that in mind when you design your own cdevsw values: populate what your device truly needs, and leave the rest alone.

The per-LED softc lives in a `struct ledsc` defined near the top of the file:

```
struct ledsc {
    LIST_ENTRY(ledsc)    list;
    char                 *name;
    void                 *private;
    int                  unit;
    led_t                *func;
    struct cdev           *dev;
    /* ... more state ... */
};
```

It carries a back-pointer to its cdev in the `dev` field, and a unit number allocated from an `unrhdr(9)` pool rather than from `Newbus`:

```
sc->unit = alloc_unr(led_unit);
```

The actual `make_dev` call is right below:


```
sc->dev = make_dev(&led_cdevsw, sc->unit,
    UID_ROOT, GID_WHEEL, 0600, "led/%s", name);
```

Notice the path: "led/%s". Every LED created by the framework lands in the /dev/led/ subdirectory with a free-form name chosen by the calling driver (for instance led/ehci0). This is how the framework keeps its nodes grouped.

Immediately after the make_dev, the framework stashes the_softc pointer:

```
sc->dev->si_drv1 = sc;
```

This is the pre-mda_si_drv1 way of doing it, and it predates make_dev_s. Newer drivers should pass mda_si_drv1 through the args struct instead, so the pointer is set before the cdev becomes reachable.

Destruction is a single call:

```
destroy_dev(dev);
```

Simple. Synchronous. No deferred destruction, no drain loop at the caller level. The framework relies on the kernel's draining behavior in destroy_dev to finish any in-flight handlers.

What to copy from led: the naming convention (subdirectory per framework), the_softc layout (back-pointer plus identity fields plus callback pointer), the clean alloc_unr / free_unr pattern for unit numbers that do not come from Newbus.

What to leave alone: the sc->dev->si_drv1 = sc assignment after make_dev. Use mda_si_drv1 in make_dev_s instead.

8.12.3 Walkthrough 3: /usr/src/sys/dev/md/md.c

The memory disk driver is larger than the previous two, and most of its bulk is not about device files at all. It is about GEOM, about backing store, about swap-backed and vnode-backed instances. For our purposes, we look at one specific thing: the control node /dev/mdctl.

Find the cdevsw declaration near the top of md.c:

```
static struct cdevsw mdctl_cdevsw = {
    .d_version =    D_VERSION,
    .d_ioctl =      mdctl_ioctl,
    .d_name =       MD_NAME,
};
```

Only two fields set. d_version and d_ioctl and a name. No d_open, d_close, d_read, or d_write. The control node is used exclusively through ioctl(2): create an md, attach a backing store, destroy an md. That is the shape of many control interfaces in the tree.

The cdev is created near the bottom of the file:

```
status_dev = make_dev(&mdctl_cdevsw, INT_MAX, UID_ROOT, GID_WHEEL,  
0600, MDCTL_NAME);
```

INT_MAX is a common pattern for singletons when the unit number does not matter: it places the cdev outside any plausible range of driver-instance unit numbers. 0600 and root:wheel are the narrow baseline you would expect for a privileged control node.

Destruction happens in the module unload path:

```
destroy_dev(status_dev);
```

Again, a single call.

What to copy from md: the pattern of exposing a single control cdev for a subsystem whose data path lives elsewhere (in md's case, in GEOM), and the very narrow permissions for a node that has genuine privilege.

What to leave alone: md is a large subsystem; do not try to copy its structure as a template. Copy the control-node idea; leave the GEOM layering for Chapter 27.

8.12.4 A Brief Word About Cloning

The LED framework creates one cdev per LED in response to an API call from other drivers. The md module creates one cdev per memory disk in response to an ioctl on its control node. Both are examples of **drivers that create cdevs dynamically in response to events**.

FreeBSD also has a dedicated cloning mechanism: clone_create(9) and the dev_clone event handler. When a user opens a name that matches a pattern registered by a cloning driver, the kernel synthesises a new cdev, opens it, and returns the descriptor. This was historically a common pattern for subsystems that wanted a new per-session cdev on every open. Modern FreeBSD leans on devfs_set_cdevpriv(9) instead whenever the only reason to clone a cdev was to give each descriptor independent state, because per-open state is simpler, lighter, and does not need a pool of minor numbers. True cloning is still in the tree (the PTY subsystem under /dev/pts/* is the clearest living example), and the API surface that supports it is worth recognising.

Cloning is flexible and we will not cover it in Chapter 8. Several of the API elements we have already met (the D_NEEDMINOR flag, dev_stdclone, the CLONE_* constants) exist to support it. Mentioning them here gives you a vocabulary for the day you encounter a driver that uses them. For now, the takeaway is that FreeBSD has a spectrum of mechanisms for creating cdevs, and make_dev_s from attach is the simplest end of it.

8.12.5 Walkthrough 4: /usr/src/sys/net/bpf.c (Device-File Surface Only)

BPF, the Berkeley Packet Filter, is a large subsystem. We will not try to understand what it does at the network level; Part 6 of this book has a chapter dedicated to network drivers. Here we look at one specific thing: how BPF shapes its device-file surface.

The relevant declarations are near the top of /usr/src/sys/net/bpf.c. The cdevsw is populated with the full set of data-path operations plus poll and kqueue:

```
static struct cdevsw bpf_cdevsw = {
    .d_version =    D_VERSION,
    .d_open =      bpfopen,
    .d_read =      bpfread,
    .d_write =     bpfwrite,
    .d_ioctl =     bpfioctl,
    .d_poll =      bpfpoll,
    .d_name =      "bpf",
    .d_kqfilter =  bpfkqfilter,
};
```

No `d_close`. BPF does everything on close through the `cdevpriv` destructor, which is the pattern this chapter recommends. The dispatch table says “for these operations, call these handlers, and do nothing special on close because the destructor has it”.

The open handler, already quoted in the per-open state section, fits the pattern exactly:

```
d = malloc(sizeof(*d), M_BPF, M_WAITOK | M_ZERO);
error = devfs_set_cdevpriv(d, bpf_dtor);
if (error != 0) {
    free(d, M_BPF);
    return (error);
}
```

Allocate, register with `devfs`, free on registration failure. Nothing else in the open path matters for our purposes here.

The destructor is the interesting part. BPF’s `bpf_dtor` has to unwind a fair amount of state: it stops a callout, disconnects from its BPF interface, drains a select set, and drops references. It does all of that **without ever calling `d_close`** from the `cdevsw`. Destructor-based cleanup is cleaner than close-based cleanup for a driver that supports multiple openers, because the destructor fires exactly once per open, whereas `d_close` without `D_TRACKCLOSE` fires only on the last close of the final shared file.

The `cdev` layout on the BPF side is simpler than a first reading of the subsystem suggests. BPF creates exactly one primary node and one alias:

```
dev = make_dev(&bpf_cdevsw, 0, UID_ROOT, GID_WHEEL, 0600, "bpf");
make_dev_alias(dev, "bpf0");
```

That is the full register-a-device surface. There is no per-instance `/dev/bpfN` `cdev`; the names you may see on a running system such as `/dev/bpf0` and `/dev/bpf` both resolve to the same `cdev`, and every distinct user of BPF is distinguished entirely by its per-open structure, attached through `devfs_set_cdevpriv` at open time. That is exactly the shape you adopt for `myfirst` at the end of this chapter: one `cdev`, an alias, per-descriptor state, and a destructor that handles teardown. BPF is a large subsystem, but its device-file surface is one you now know how to write.

What to copy from BPF: the practice of doing open-time cleanup in the destructor rather than in `d_close`. The practice of allocating per-open state immediately and registering it before any other work. The discipline of freeing the allocation on registration failure. The willingness to keep the `cdev` count small and let the descriptor carry the session.

What to leave alone: the BPF-specific machinery that surrounds the open and close paths, including the interface attach logic, the select-set bookkeeping, and the counter-based statistics. Those belong to the network side of BPF, and Part 6 will revisit them when the book introduces network drivers.

8.12.6 A Synthesis Across All Four Drivers

With four walkthroughs under your belt, it helps to line up the commonalities and the differences in one table. Every row is a driver property; every column is a driver.

Property	<code>null</code>	<code>led</code>	<code>md</code>	<code>bpf</code>
How many <code>cdevsw</code> values	3	1	1 (plus GEOM)	1
<code>cdevs</code> per attach	3 total	1 per LED	1 control + many	1 plus 1 alias
Softc?	no	yes	yes	yes (per-open)
Subdirectory in <code>/dev</code> ?	no	yes (<code>led/*</code>)	no	no
Permission mode	<code>0666</code>	<code>0600</code>	<code>0600</code>	<code>0600</code>
Uses <code>devfs_set_cdevpriv</code> ?	no	no	no	yes
Uses cloning?	no	no	no	no
Uses <code>make_dev_alias</code> ?	no	no	no	yes
<code>d_close</code> populated?	no	no	no	no
Uses <code>destroy_dev_drain</code> ?	no	no	no	no
Primary use-case	pseudo-data	hardware ctl	subsystem ctl	packet capture

Every column is defensible. Each driver has chosen the simplest set of features that fits its job. Your `myfirst` driver by the end of Chapter 8 has a profile closer to `led` than to the others: one `cdevsw`, per-instance `softc`, subdirectory naming, narrow permissions, plus `devfs_set_cdevpriv` for per-open state (which `led` does not need) and an alias (which `led` does not use).

That profile is a good place to be. It is large enough to show that you have engaged with the real mechanisms, and small enough that every line of the driver is there for a reason.

8.12.7 What Four Drivers Taught Us

Four drivers, four different shapes, all of them good examples:

- `null` is minimalism: three `cdevsw` values, three singletons, no `softc`, no per-open state, mode `0666` because the data is harmless.

- `led` is a framework: one `cdevsw`, many `cdevs`, each with its own `softc` and unit number, subdirectory naming, narrow permissions, only `d_write` populated because the device is written-to rather than read-from.
- `md` is a control interface: one `cdevsw` with only `d_ioctl`, a singleton `cdev`, `INT_MAX` as unit number, narrow permissions for privileged operations.
- `bpf` is a per-session driver: one `cdevsw` with the full data-path set, a single primary `cdev` plus an alias, and the whole of per-descriptor state carried through `devfs_set_cdevpriv(9)`.

Your `myfirst` is closest to `bpf` in shape by the end of this chapter: one `cdevsw`, one primary `cdev` with an alias, a `softc` for device-wide state, and per-open state for per-descriptor book keeping. Where it differs is scope. `bpf` implements the full data path, and it does so to serve real packets. `myfirst` stops at the surface. That is fine for now. You are in good company, and the rooms behind the door open in Chapter 9.

Reading other drivers becomes a skill you build over time. The device-file lens is just one of several lenses you will apply. Keep the four walkthroughs above in mind as a starting library.

8.13 Common Scenarios and Recipes

This section is a cookbook. Each entry is a situation you will run into sooner or later, a short recipe for handling it, and a pointer to which part of the chapter explains the underlying machinery. Read them now or bookmark them for when they come up.

8.13.1 Recipe 1: A Driver That Must Never Be Opened Twice

Situation. Your hardware has state that is only sane with a single concurrent user. A second `open(2)` would corrupt the state.

Recipe.

1. In the `softc`, keep a single `int is_open` flag and a mutex.
2. In `d_open`, take the mutex, check the flag, return `EBUSY` if set, set it, drop the mutex.
3. In `d_close`, take the mutex, clear the flag, drop the mutex.
4. In `detach`, check the flag under the mutex and return `EBUSY` if set.
5. Do **not** also use `devfs_set_cdevpriv` for per-open state; there is no “per-open” because there is only one open at a time.

This is the Chapter 7 pattern. `myfirst` used it to force attention on the lifecycle; Chapter 8 stage 2 moves away from it because most drivers want multiple openers. Use the exclusive pattern only when hardware constraints force it.

8.13.2 Recipe 2: A Driver With Per-User Read Offsets

Situation. Your device exposes a seekable stream. Two different user processes should each get their own offset into the stream; neither should see the other’s position change their own view.

Recipe.

1. Define struct `myfirst_fh` with an `off_t read_offset` field.
2. In `d_open`, allocate the struct and call `devfs_set_cdevpriv` with a destructor that frees it.
3. In `d_read`, call `devfs_get_cdevpriv` to retrieve the struct. Use `fh->read_offset` as the starting point. Advance it by the number of bytes actually transferred.
4. In `d_close`, do nothing; the destructor runs automatically when the descriptor releases.

Chapter 9 will fill in the `d_read` body. The skeleton is already in stage 2.

8.13.3 Recipe 3: A Device With a Privileged Control Node

Situation. Your driver has a data surface anyone should be able to read, and a control surface only privileged users should touch.

Recipe.

1. Define two `cdevsw` structures, `something_cdevsw` and `something_ctl_cdevsw`.
2. Create two `cdevs` in `attach`: one for the data node (`0644 root:wheel` or similar), one for the control node (`0600 root:wheel`).
3. Keep both pointers in the `softc`. Destroy the control `cdev` before the data `cdev` in `detach`.
4. Use `priv_check(9)` inside the control node's `d_ioctl` handler if you want enforcement beyond file permissions.

Lab 8.5 walks through this. The `md` walkthrough in the Reading Real Drivers section is a real-world variant.

8.13.4 Recipe 4: A Node That Appears Only When a Condition Is Met

Situation. You want `/dev/myfirst/status` to appear only when the driver's hardware is in a specific state, and disappear otherwise.

Recipe.

1. Keep `sc->cdev_status` as a struct `cdev *` in the `softc`, initialised to `NULL`.
2. In the handler where the condition becomes true, call `make_dev_s` and store the pointer.
3. In the handler where the condition becomes false, call `destroy_dev` on the pointer and set it to `NULL`.
4. Guard both with the `softc` mutex so concurrent transitions do not race.
5. In `detach`, if the pointer is non-`NULL`, destroy it before tearing down the other `cdevs`.

This is the same pattern as the primary data node, just triggered by a different event. Watch for the case where the condition changes repeatedly in quick succession: you will hit the `devfs` allocator harder than expected, and you may want to debounce.

8.13.5 Recipe 5: A Node Whose Ownership Should Change Based on a Runtime Knob

Situation. A lab system sometimes wants the node owned by `operator`, sometimes by `wheel`. You do not want to reload the driver to switch.

Recipe.

1. Leave the driver's `mda_uid` and `mda_gid` at a narrow baseline (`UID_ROOT`, `GID_WHEEL`).

2. Use `devfs.conf` to widen when wanted:

```
own      myfirst/0      root:operator
perm     myfirst/0      0660
```

3. Apply with `service devfs restart`.
4. To switch back, comment out the lines and re-run `service devfs restart`.

Policy lives in userland; the driver stays pristine. Lab 8.4 practices this.

8.13.6 Recipe 6: A Node That Should Be Visible Only Inside a Jail

Situation. A node should appear inside a specific jail but not on the host.

Recipe.

1. In the driver, default to creating the node as usual on the host.
2. In `/etc/devfs.rules`, create a ruleset for the jail that explicitly hides the node:

```
[special_jail=120]
add include $devfsrules_hide_all
add include $devfsrules_unhide_basic
add path myfirst hide
```

3. Apply the ruleset number in the jail's `jail.conf`.

This is a blunter variant of Lab 8.7. For a node that should be visible in the jail and not on the host, the logic inverts: create the node on the host (where it is always created), and use a host-side `devfs.conf` rule to tighten permissions so nothing on the host touches it.

8.13.7 Recipe 7: A Node That Can Survive Unexpected Process Deaths

Situation. Your driver holds resources per open. If a process crashes, you must not leak the resource.

Recipe.

1. Allocate the resource in `d_open`.
2. Register a destructor with `devfs_set_cdevpriv`. The destructor frees the resource.
3. Rely on the kernel to run the destructor when the last reference to the `struct file` drops, regardless of why. `close(2)`, `exit(3)`, or `SIGKILL` all reach the same cleanup path.

The destructor guarantee is the primary reason `devfs_set_cdevpriv` exists. No matter how badly userland misbehaves, your resource gets freed.

8.13.8 Recipe 8: A Device That Must Support Polling

Situation. Your driver produces events, and user programs want to `select(2)` or `poll(2)` on the device to learn when an event is pending.

Recipe. Out of scope for Chapter 8; this is Chapter 10 territory. The short form for recognition: set `.d_poll = myfirst_poll` in your `cdevsw`, implement the handler to return the appropriate mask of `POLLIN`, `POLLOUT`, `POLLERR` bits, and use `selrecord(9)` to register interest for deferred wake-up. Chapter 10 walks through each of these in detail.

8.13.9 Recipe 9: A Device That Needs to Be Mapped Into User Memory

Situation. Your driver has a shared memory region (DMA buffer, hardware register window) that a user process should access directly through `mmap(2)`.

Recipe. Out of scope for Chapter 8; covered in Part 4 when hardware access is introduced. For recognition only: set `.d_mmap = myfirst_mmap` in your `cdevsw`, implement the handler to return the physical page address for each offset and protection mask, and think carefully about what happens when the user-mapped memory is backed by hardware that can vanish. This is one of the more nuanced areas of driver work.

8.13.10 Recipe 10: A Device That Exposes a Log

Situation. Your driver produces log messages that are more voluminous than `device_printf` should handle, and user programs should be able to `read(2)` them in order.

Recipe.

1. Allocate a ring buffer in the softc.
2. In the code paths that produce log events, format them into the ring buffer under a lock.
3. In `d_read`, copy bytes from the ring buffer to userland via `uiomove(9)` (Chapter 9).
4. Use per-open state (a `read_offset` in the `fh`) so each reader drains the buffer at its own pace.
5. Consider setting `D_TRACKCLOSE` if a single reader should be allowed to empty the buffer on close.

This is the shape of several kernel log devices. It is worth knowing the pattern exists; the full implementation is a later-chapter exercise.

8.13.11 When a Recipe Does Not Fit

The cookbook is not exhaustive. When you face a situation that does not match any of the recipes above, a useful habit is to ask three questions in order:

- **Is it about identity?** Then it is a `cdev` question: naming, subdirectories, aliases, creation, destruction.
- **Is it about policy?** Then it is a permissions and policy question: ownership, mode, `devfs.conf`, `devfs.rules`.
- **Is it about state?** Then it is a per-open versus per-device question: `softc` or `devfs_set_cdevpriv`.

Most real-world driver design questions resolve into one of these three. When you have classified the question, the rest of the chapter tells you which tool to reach for.

8.14 Practical Workflows for the Device-File Surface

Knowing the APIs is half the job. Knowing when to reach for which one, and how to spot problems quickly, is the other half. This section collects the workflows that will make the next several chapters go smoothly: the inner loop for editing a driver, the habits that catch bugs early, and the checklists worth running through before a serious change.

8.14.1 The Inner Loop

The “inner loop” is the cycle of edit, build, load, test, unload, edit again. Your Chapter 7 scripts already have a version of this. For Chapter 8 the inner loop gets a little richer because there are more user-visible surfaces to verify.

A useful sequence when you are working on a stage of `myfirst`:

```
% cd ~/drivers/myfirst
% sudo kldunload myfirst 2>/dev/null || true
% make clean && make
% sudo kldload ./myfirst.ko
% dmesg | tail -5
% ls -l /dev/myfirst /dev/myfirst/0 2>/dev/null
% sysctl dev.myfirst.0.stats
% sudo ./probe_myfirst /dev/myfirst/0
% sudo kldunload myfirst
% dmesg | tail -3
```

Each line has a purpose. The first unload is defensive: the previous test left the module loaded, and this clears the slate. The `make clean && make` rebuilds from scratch to avoid a stale object file. The first `dmesg | tail -5` shows the attach messages. The `ls -l` and `sysctl` confirm that the user-visible surface is present and the internal counters are initialised. The probe exercises the data path. The final unload and `dmesg` confirm the detach messages.

If any step produces an unexpected result, you know which step. That is the value of scripting the loop: not to save typing, but to make the failure signal unambiguous.

The `rebuild.sh` helper shipped with Chapter 7’s examples wraps most of this for you. Lab 8 reuses it unchanged.

8.14.2 Reading dmesg Well

`dmesg` is the narrative of what your driver did. Reading it well is a habit worth building early.

The kernel’s default ring buffer can show tens of thousands of lines from earlier boot and runtime activity. When you are developing a specific driver, three techniques make the relevant slice visible:

Clearing before the test. `sudo dmesg -c > /dev/null` clears the buffer. The next load/unload cycle then produces a small, focused log. Use this between experiments.

Filtering by tag. `dmesg | grep myfirst` narrows the view to lines your driver produced, assuming your `device_printf` calls emit the driver name. They do, because `device_printf(9)` prefixes every line with the Newbus name of the device.

Watching in real time. Run `tail -f /var/log/messages` in a second terminal. Every driver message that makes it to `dmesg` also appears there, with timestamps. This is especially useful during long-running tests such as Lab 8.6's parallel-probe exercise.

8.14.3 Watching fstat

For detach problems, `fstat(1)` is your friend. Two idioms come up often:

```
% fstat /dev/myfirst/0
```

Plain lookup; shows all processes holding the node open. The output columns are user, command, pid, fd, mount, inum, mode, rdev, r/w, name.

```
% fstat -p $$ | grep myfirst
```

Restrict the search to the current shell. Useful when you are not sure whether your current shell has a leftover descriptor open from an earlier test.

```
% fstat -u $USER | grep myfirst
```

Restrict to your own user's processes. Similar use case, broader scope.

8.14.4 The sysctl Partner of Every Driver

From Chapter 7 your driver already exposes a `sysctl` tree under `dev.myfirst.0.stats`. Chapter 8 stage 2 adds `active_fhs` to that tree. When you are running experiments, the `sysctls` are the cheapest possible observation tool:

```
% sysctl dev.myfirst.0.stats
dev.myfirst.0.stats.attach_ticks: 123456
dev.myfirst.0.stats.open_count: 42
dev.myfirst.0.stats.active_fhs: 0
dev.myfirst.0.stats.bytes_read: 0
```

Every counter is a check on what the driver thinks is true. Discrepancies between what you expected and what `sysctl` shows are always a signal. If `active_fhs` is non-zero when no descriptor should be open, you have a leak. If `open_count` is smaller than the number of times you opened the device, your attach path is running twice or your counter is racy.

`Sysctls` are cheaper than any other observation mechanism. Prefer them over reading the device itself whenever a numeric or short-string piece of information will do.

8.14.5 A Quick Checklist for Every Code Change

Before you commit a change to the driver, walk through the following. Ten minutes spent here saves hours of debugging later.

1. Does the driver still build from a clean tree?
2. Does it still load and unload cleanly on a system with no open descriptors?
3. Does the user-visible surface (`ls -l /dev/myfirst/...`) match what your code intends?
4. Are the permissions still the narrow defaults they should be?
5. If you changed `attach`, does every error path still unwind completely?
6. If you changed `detach`, does the driver still unload cleanly when descriptors are held open (either with a clean `EBUSY` return or, if you chose a different policy, without leaking)?
7. If you changed per-open state, do all three of `stress_probe`, `parallel_probe`, and `hold_myfirst` still behave as expected?
8. Did you introduce any `device_printf` calls that should be `if (bootverbose)-gated` so they do not flood the log?
9. Did you leave any `#if 0` or debug prints in place? Pull them out now.
10. If you changed ownership or mode, does `devfs.conf` still produce the expected override?

This checklist is deliberately boring. That is the point. A boring, reliable process beats a heroic debugging session every time.

8.14.6 A Quick Checklist Before a Release

When you are preparing a stage of your driver for “done”, the checklist gets a little longer. All of the above, plus:

1. `make clean && make` from a truly clean tree builds without warnings.
2. `kldload ./myfirst.ko; sleep 0.1; kldunload myfirst` completes ten times without issue.
3. `stress_probe 10000` completes without issue and `active_fhs` returns to zero.
4. `parallel_probe 8` opens eight descriptors, holds, and cleanly closes. Kernel log shows eight distinct `fh=` pointers and eight destructors.
5. `kldunload` while a descriptor is open returns `EBUSY` cleanly, not a panic.
6. `devfs.conf` with a widening entry applies on service `devfs` restart.
7. An `ls -l` audit of `/dev/myfirst*` shows no unexpected modes or ownerships.
8. `dmesg` contains exactly the attach and detach messages you expect, with no warnings or errors.
9. Source code is free of commented-out experiments, TODO lines, and debug helpers.
10. The sysctls your driver exposes are descriptive and documented in code comments.

No commit should skip items 1 through 3. They are the cheapest high-value insurance you can buy.

8.14.7 A Workflow for Adding a New Node

Walking through the workflow end-to-end once will anchor the earlier sections. Suppose you decide `myfirst` should expose an additional read-only status node at `/dev/myfirst/status`, distinct from the numbered data nodes. Here is how you would do it.

Step 1: design. Decide on the shape of the node. Does it belong in the same `cdevsw` as the data node, or a different one? A status-only node that answers `read(2)` with a short text summary usually wants its own `cdevsw` with only `d_read` set, because the policy is different from the data node. Decide on the permission mode. Read-only by anyone suggests `0444`; read-only by operators suggests `0440` with an appropriate group.

Step 2: declare. Add the new `cdevsw`, its `d_read` handler, and a `struct cdev *cdev_status` field to the `softc`.

Step 3: implement. Write the `d_read` handler. It formats a short string based on `softc` state and returns it through `uiomove(9)`. For Chapter 8 you might stub this and fill it in after Chapter 9.

Step 4: wire. In `attach`, add the `make_dev_s` call for the status node. In `detach`, add the `destroy_dev` call, before the data node's `destroy`.

Step 5: test. Rebuild, reload, inspect, exercise, unload. Check that `ls -l` shows the status node with the expected mode. Check that `cat /dev/myfirst/status` works and produces sensible output. Check that the whole driver still unloads cleanly.

Step 6: document. Add a comment in the driver source describing the node. Add an entry in the `dev.myfirst.0.stats` `sysctl` if the status is numeric and would fit there too. Note the change in whatever change log you keep.

Six steps, each one small, each one specific. That is the level of granularity at which bugs stay visible.

8.14.8 A Workflow for Diagnosing a Missing Node

The Chapter 8 troubleshooting walkthrough in the Tools section gave you a short checklist. Here is a fuller workflow that fits on an index card.

Phase 1: is it the module?

- `kldstat | grep myfirst` shows the module.
- `dmesg | grep myfirst` shows the attach messages.

If the module is not loaded or the attach did not run, fix that first.

Phase 2: is it Newbus?

- `devinfo -v | grep myfirst` shows the Newbus device.

If Newbus shows nothing, your `device_identify` or `device_probe` is not creating the child. Look there.

Phase 3: is it devfs?

- `ls -l /dev/myfirst` lists the directory (or reports it missing).
- `dmesg | grep 'make_dev'` shows any failure from `make_dev_s`.

If Newbus is fine but devfs shows nothing, `make_dev_s` returned an error. Check your path format string, your `mda_devsw`, your argument structure.

Phase 4: is it policy?

- `devfs rule showsets` lists active rulesets.
- `devfs rule -s N show` lists rules in ruleset N.

If devfs has the cdev but your jail or your local session does not see it, the ruleset is hiding it.

Every failure maps to one of these four phases. Work through them in order and you will almost always identify the cause in under a minute.

8.14.9 A Workflow for Reviewing Someone Else's Driver

When you review a pull request that touches a driver's device-file surface, the useful questions are:

- Does every `make_dev_s` have a matching `destroy_dev`?
- Does every error path after `make_dev_s` call `destroy_dev` before returning?
- Does detach destroy every cdev it created?
- Is `si_drv1` populated through `mda_si_drv1` rather than a post-hoc assignment?
- Is the permission mode defensible for the node's purpose?
- Is the `cdevsw`'s `d_version` set to `D_VERSION`?
- Are all `d_*` handlers present for the operations the node should support, and is each of them consistent about its `errno` returns?
- If the driver uses `devfs_set_cdevpriv`, is there exactly one successful set per open and exactly one destructor?
- If the driver uses aliases, are they destroyed in detach before the primary?
- If the driver has more than one cdev, does it call `destroy_dev_drain` in its unload path?

This is a review checklist, not a tutorial. The review is faster because every question has a yes or no answer and every yes can be checked mechanically.

8.14.10 Keeping a Lab Logbook

A lab logbook is a small notebook or text file where you record what you did, what you saw, and what you learned. The book has recommended this since Chapter 2. In Chapter 8 it pays off in a specific way: you will run the same kinds of experiments many times, and a brief note lets you avoid repeating the same misstep twice.

A useful template for a logbook entry:

```
Date: 2026-04-17
Driver: myfirst stage 2
Goal: verify per-open state is isolated across two processes
Steps:
- loaded stage 2 kmod
- ran parallel_probe with count=4
- observed 4 distinct fh= pointers in dmesg
- observed active_fhs=4 in sysctl
- closed, observed 4 destructor lines, active_fhs=0
Result: as expected
Notes: first run missed destructor lines because dmesg ring buffer
      was full; dmesg -c before the test solved it
```

Two minutes per experiment, no more. The value appears months later when you are tracking down a new issue and a logbook search reveals that the same symptom appeared once before under different circumstances.

8.14.11 Common Design Questions and How to Think About Them

Some questions recur when driver authors reach the device-file stage. Each of these has come up more than once in real review discussions. The answers are short; the reasoning is worth internalising.

Q: Should I create the `cdev` in `device_identify` or in `device_attach`?

In `device_attach`. The `identify` callback runs very early, before the driver instance has a `softc`. The `cdev` wants to reference the `softc` via `mda_si_drv1`, which means the `softc` must already exist. Chapter 7 set this pattern; keep it.

Q: Should I create additional `cdevs` outside `attach` and `detach`?

If they are genuinely per-driver-instance, put them in `attach` and destroy them in `detach`. If they are dynamic, created in response to a user action, create them in whichever handler receives the user's request and destroy them either when a later handler unwinds the request or when the driver detaches. Track them carefully; lost `cdevs` are a common source of leaks.

Q: Should I set `D_TRACKCLOSE`?

Usually no. The per-open state mechanism via `devfs_set_cdevpriv` covers almost every case for which `D_TRACKCLOSE` would be tempting, and it cleans itself up automatically. Set `D_TRACKCLOSE` only when you need your `d_close` to run on every descriptor close, not just the last one. Real use cases are rare; TTY drivers and a handful of others fit.

Q: Should I allow multiple openers?

Default to yes, via per-open state. Exclusive access is sometimes necessary for hardware that can only support one session at a time, but it is a choice, not a default. Chapter 7 forced exclusivity as a teaching move; Chapter 8 stage 2 lifts that restriction precisely because it is not the common case.

Q: Should I return `ENXIO` or `EBUSY` on a failed open?

`ENXIO` when the driver is not ready. `EBUSY` when the device can be opened in principle but not right now. The user-visible messages are different, and an operator who reads your kernel log will thank you for picking the correct one.

Q: Should I `strdup` strings I get from userland?

Not on the open path. If a handler has a legitimate reason to remember a user-supplied string beyond the call, use `malloc(9)` with an explicit size and copy the string in. Never rely on a pointer into userland memory after a handler returns; it may no longer be valid, and even if it is, the kernel should never trust userland-owned memory for long.

Q: Should the `softc` remember which descriptors have it open?

Usually no. Per-open state via `devfs_set_cdevpriv` is the answer. If you need an iteration mechanism, `devfs_foreach_cdevpriv` exists and is correct. Do not maintain your own list of descriptor pointers in the `softc`; the locking is nontrivial and the kernel already provides the right answer.

Q: When should my `detach` refuse with `EBUSY`?

When the driver cannot safely tear itself down with the current state. Open descriptors are the most common reason. Some drivers also refuse if hardware is actively transferring, or if a control operation is in progress. Error out early and cleanly; do not try to coerce the system into a clean state from inside detach.

Q: Can I unload the driver while descriptors are open?

Not if detach refuses. If your detach accepts the situation, the kernel will still drain in-flight handlers, but open descriptors remain on existing file tables until the processes close them, and those descriptors will then return ENXIO (or similar) from further operations. For a teaching driver, refusing EBUSY is the cleaner choice.

These questions are the ones that will come up in your first real driver review. Having seen them once here means you are not seeing them for the first time when the reviewer asks.

8.14.12 A Decision Tree for Common Design Choices

When you sit down to design a new node or change an existing one, the questions tend to fall into a small set of branches. The tree below is a field guide, not an algorithm; real design always involves judgement, but knowing the shape of the tree helps.

Start: I want to expose something through /dev.

Branch 1: What kind of state does the node carry?

- **Session-less, trivial data source or sink** (like /dev/null, /dev/zero): no softc, no per-open state, one cdevsw per behaviour. Use make_dev or make_dev_s in a MOD_LOAD handler. Mode typically 0666.
- **Per-device hardware** (like a serial port, a sensor, an LED): one softc per instance, one cdev per instance. Use attach/detach pattern. Mode typically 0600 or 0660.
- **Subsystem control** (like /dev/pf or /dev/mdctl): one cdev exposing d_ioctl-only operations. Mode 0600.
- **Per-session state** (like BPF, like FUSE): one cdev per session or one cloning entry point. Per-open state via devfs_set_cdevpriv. Mode 0600.

Branch 2: How do users discover the node?

- **Stable fixed name** (like /dev/null): put the name in the make_dev format string and leave it.
- **Per-instance numbered name** (like /dev/myfirst0): use %d in the format string and device_get_unit(9) for the number.
- **Subdirectory grouping** (like /dev/led/foo): use / inside the format string; devfs creates the directory on demand.
- **On-demand per-open instance**: use cloning. Covered later.

Branch 3: Who can touch it?

- **Anyone**: UID_ROOT, GID_WHEEL, mode 0666. Rare; use only for harmless nodes.
- **Root only**: UID_ROOT, GID_WHEEL, mode 0600. The default for anything privileged.
- **Root plus an operator group**: UID_ROOT, GID_OPERATOR, mode 0660. Common for hands-on privileged tools.
- **Root write, anyone read**: UID_ROOT, GID_WHEEL, mode 0644. For status nodes.
- **Custom named group**: define the group in /etc/group, use devfs.conf to adjust ownership at node creation time. Do not invent a group inside your driver.

Branch 4: How many concurrent openers?

- **Exactly one at a time:** exclusive open pattern, flag in `softc`, check under mutex in `d_open`, return `EBUSY` on conflict. No `devfs_set_cdevpriv`.
- **Multiple, each with independent state:** remove the exclusive check, allocate per-open struct in `d_open`, call `devfs_set_cdevpriv`, read back with `devfs_get_cdevpriv`.
- **Multiple, all sharing driver-wide state:** allocate nothing per open; just read and write the `softc` under its mutex.

Branch 5: What happens when the driver unloads with users active?

- **Refuse the unload** with `EBUSY` from `detach` as long as any descriptor is open. This is the clean default.
- **Accept the unload** but invalidate open descriptors. In this case you need a `d_purge` handler to wake any blocked threads and convince them to return promptly. More complex; do this only when the refusal would leave the system in a worse state.

Branch 6: What kind of name adjustments do users and operators need?

- **A second name maintained by the driver itself** (legacy path, well-known shortcut): `make_dev_alias(9)` in `attach`, `destroy_dev(9)` on it in `detach`.
- **A second name maintained by the operator:** link in `/etc/devfs.conf`. Driver does nothing.
- **A permission widening or narrowing per host:** `own` and `perm` in `/etc/devfs.conf`. Driver keeps its baseline.
- **A jail-filtered view:** a ruleset in `/etc/devfs.rules`, referenced in `jail.conf`. Driver has nothing to say.

Branch 7: How do userland programs receive events from the driver?

- **Polling by reading:** drivers that only need to hand out bytes. `d_read` and `d_write`.
- **Blocking reads with signals:** drivers that should unblock on `SIGINT`. Covered in Chapter 10.
- **Poll/select:** `d_poll`. Covered in Chapter 10.
- **Kqueue:** `d_kqfilter`. Covered in Chapter 10.
- **devd notifications:** `devctl_notify(9)` from the driver; operator-side rules in `/etc/devd.conf`.
- **sysctl pulls:** for observability without file-descriptor cost. Always complementary to the `/dev` surface.

This tree does not cover every case. It covers enough that a driver author can navigate the first several design decisions without panicking. When a new question comes up that is not on the tree, write down the question and the answer you settled on; that is how the tree grows for you personally.

8.14.13 A Warning About Over-Engineering

A few design temptations are worth calling out specifically, because they tend to turn simple drivers into complicated drivers with no gain.

- **Inventing your own IPC protocol over read/write.** If the messages are structured, use `ioctl(2)` (Chapter 25).
- **Embedding a tiny language into `ioctl` commands** so users can “script” the driver. This is almost always a sign that the feature belongs in userland.

- **Multiplexing many unrelated subsystems through one cdevsw.** If two surfaces have different semantics, give them two cdevsw values; it costs nothing and reads better.
- **Adding D_NEEDGIANT to silence an SMP warning.** The warning is correct; fix the locking.
- **Handling every possible errno value from every possible userland program.** Pick the right one for your situation and stick with it. The standard err(3) family does the rest.

The discipline of “as simple as it can be, but no simpler” is especially important at this level. Every line of driver code is a line that could have a bug in it under load. A lean driver is easier to review, easier to debug, easier to port, and easier to hand off to the next maintainer.

8.15 Hands-On Labs

These labs extend the Chapter 7 driver in place. You should not need to retype anything from scratch. The companion directory mirrors the stages.

8.15.1 Lab 8.1: Structured Name and Tighter Permissions

Goal. Move the device from /dev/myfirst0 to /dev/myfirst/0, and change the group to operator with mode 0660.

Steps.

1. In myfirst_attach(), change the make_dev_s() format string to "myfirst/%d".
2. Change args.mda_gid from GID_WHEEL to GID_OPERATOR, and args.mda_mode from 0600 to 0660.
3. Rebuild and reload:

```
% make clean && make
% sudo kldload ./myfirst.ko
% ls -l /dev/myfirst
total 0
crw-rw----  1 root  operator  0x5a Apr 17 09:41 0
```

4. Confirm that a normal user in the operator group can now read from the node without sudo. On FreeBSD, you add a user to that group with pw groupmod operator -m yourname, then start a fresh shell.
5. Unload the driver and confirm the /dev/myfirst/ directory disappears along with the node.

Success criteria.

- /dev/myfirst/0 appears on load and disappears on unload.
- ls -l /dev/myfirst/0 shows crw-rw---- root operator.
- A member of the operator group can cat </dev/myfirst/0 without error.

8.15.2 Lab 8.2: Add an Alias

Goal. Expose `/dev/myfirst` as an alias for `/dev/myfirst/0`.

Steps.

1. Add a struct `cdev` `*cdev_alias` field to the softc.
2. After the successful `make_dev_s()` call in `myfirst_attach()`, call:

```
sc->cdev_alias = make_dev_alias(sc->cdev, "myfirst");
if (sc->cdev_alias == NULL)
    device_printf(dev, "failed to create alias\n");
```

3. In `myfirst_detach()`, destroy the alias before destroying the primary `cdev`:

```
if (sc->cdev_alias != NULL) {
    destroy_dev(sc->cdev_alias);
    sc->cdev_alias = NULL;
}
if (sc->cdev != NULL) {
    destroy_dev(sc->cdev);
    sc->cdev = NULL;
}
```

4. Rebuild, reload, and verify:

```
% ls -l /dev/myfirst /dev/myfirst/0
```

Both paths should respond. `sudo cat </dev/myfirst` and `sudo cat </dev/myfirst/0` should behave identically.

Success criteria.

- Both paths exist while the driver is loaded.
- Both paths disappear on unload.
- The driver does not panic or leak if the alias creation fails; comment out the `make_dev_alias` line temporarily to confirm this.

8.15.3 Lab 8.3: Per-Open State

Goal. Give each `open(2)` its own small structure, and verify from userland that two descriptors see independent data.

Steps.

1. Add the struct `myfirst_fh` type and the `myfirst_fh_dtor()` destructor as shown earlier in this chapter.
2. Rewrite `myfirst_open()` to allocate a `myfirst_fh`, call `devfs_set_cdevpriv()`, and free on registration failure. Remove the exclusive-open check.

3. Rewrite `myfirst_read()` and `myfirst_write()` so each starts with a call to `devfs_get_cdevpriv(&fh)`. Leave the body unchanged for now; Chapter 9 fills it in.
4. Rebuild, reload, then run two `probe_myfirst` processes side by side:

```
% (sudo ./probe_myfirst &) ; sudo ./probe_myfirst
```

5. In `dmesg`, confirm that the two `open (per-open fh=...)` messages show different pointers.

Success criteria.

- Two simultaneous opens succeed. No `EBUSY`.
- Two distinct `fh=` pointers appear in the kernel log.
- `kldunload myfirst` is only possible once both probes have exited.

8.15.4 Lab 8.4: devfs.conf Persistence

Goal. Make the ownership change from Lab 8.1 survive reboots, without editing the driver again.

Steps.

1. In Lab 8.1, revert `args.mda_gid` and `args.mda_mode` to the Chapter 7 defaults (`GID_WHEEL, 0600`).
2. Create or edit `/etc/devfs.conf` and add:

```
own    myfirst/0      root:operator
perm   myfirst/0      0660
```

3. Apply the change without rebooting:

```
% sudo service devfs restart
```

4. Reload the driver and confirm that `ls -l /dev/myfirst/0` again shows `root operator 0660`, even though the driver itself asked for `root wheel 0600`.

Success criteria.

- With the driver loaded and `devfs.conf` in place, the node shows the `devfs.conf` values.
- With the driver loaded and the `devfs.conf` lines commented out and `devfs` restarted, the node returns to the driver's baseline.

Notes. Lab 8.4 is an operator-side lab. The driver does not change between steps. The point is to see the two-layer policy model at work: driver sets the baseline, `devfs.conf` shapes the view.

8.15.5 Lab 8.5: Two-Node Driver (Data and Control)

Goal. Extend `myfirst` to expose two distinct nodes: a data node at `/dev/myfirst/0` and a control node at `/dev/myfirst/0ctl`, each with its own `cdevsw` and its own permission mode.

Prerequisites. Completed Lab 8.3 (stage 2 with `per-open` state).

Steps.

1. Define a second struct `cdevsw` in the driver, named `myfirst_ctl_cdevsw`, with `d_name = "my-first_ctl"` and only `d_ioctl` stubbed (you will not implement `ioctl` commands; just make the function exist and return `ENOTTY`).
2. Add a struct `cdev *cdev_ctl` field to the `softc`.
3. In `myfirst_attach`, after the data node is created, create the control node with a second `make_dev_s` call. Use `"myfirst/%d.ctl"` as the format. Set the mode to `0640` and the group to `GID_WHEEL` so the control node is narrower than the data node.
4. Pass `sc` through `mda_si_drv1` for the control `cdev` as well, so `d_ioctl` can find it.
5. In `myfirst_detach`, destroy the control `cdev` **before** the data `cdev`. Log each destruction.
6. Rebuild, reload, and verify:

```
% ls -l /dev/myfirst
total 0
crw-rw----  1 root  operator  0x5a Apr 17 10:02 0
crw-r-----  1 root  wheel    0x5b Apr 17 10:02 0.ctl
```

Success criteria.

- Both nodes appear on load.
- Both nodes disappear on unload.
- The data node is group-readable by operator; the control node is not.
- Attempting `cat </dev/myfirst/0.ctl` from a non-root non-wheel user fails with `Permission denied`.

Notes. In real drivers the control node is where `ioctl` commands for configuration live. This chapter does not implement any `ioctl` commands; that work is Chapter 25's. The point of Lab 8.5 is to show that you can have two nodes with different policies wired into one driver.

8.15.6 Lab 8.6: Parallel Probe Verification

Goal. Use the `parallel_probe` tool from the companion tree to prove that per-open state really is per-descriptor.

Prerequisites. Completed Lab 8.3. The stage 2 driver is loaded.

Steps.

1. Build the userland tools:

```
% cd examples/part-02/ch08-working-with-device-files/userland
% make
```

2. Run `parallel_probe` with four descriptors:

```
% sudo ./parallel_probe /dev/myfirst/0 4
opened /dev/myfirst/0 as fd 3
opened /dev/myfirst/0 as fd 4
opened /dev/myfirst/0 as fd 5
```

```
opened /dev/myfirst/0 as fd 6
holding 4 descriptors; press enter to close
```

3. Open a second terminal and inspect dmesg:

```
% dmesg | tail -20
```

You should see four open via myfirst/0 fh=<pointer> (active=N) lines, each with a different pointer value.

4. In the second terminal, check the active_opens sysctl:

```
% sysctl dev.myfirst.0.stats.active_fhs
dev.myfirst.0.stats.active_fhs: 4
```

5. Return to the first terminal and press Enter. The probe closes all four descriptors. Check dmesg again:

```
% dmesg | tail -10
```

You should see four per-open dtor fh=<pointer> lines, one per descriptor, with the same pointer values that appeared in the open log.

6. Verify active_fhs is back to zero:

```
% sysctl dev.myfirst.0.stats.active_fhs
dev.myfirst.0.stats.active_fhs: 0
```

Success criteria.

- Four distinct fh= pointers in the open log.
- Four matching pointers in the destructor log.
- active_fhs increments to four and decrements back to zero.
- No kernel log messages about leaked memory or unexpected state.

Notes. Lab 8.6 is the strongest evidence you can easily produce that per-open state is isolated. If any step fails, the most common culprit is a missed call to `devfs_set_cdevpriv` or a destructor that does not decrement `active_fhs`.

8.15.7 Lab 8.7: devfs.rules for a Jail

Goal. Make `/dev/myfirst/0` visible inside a jail through a devfs ruleset.

Prerequisites. A working FreeBSD jail on your lab system. If you do not have one yet, skip this lab and return after Part 7.

Steps.

1. Add a ruleset to `/etc/devfs.rules`:

```
[myfirst_jail=100]
add include $devfsrules_jail
```

```
add path 'myfirst'    unhide
add path 'myfirst/*' unhide
add path 'myfirst/*' mode 0660 group operator
```

2. Add a devfs entry to the jail's jail.conf:

```
myfirstjail {
    path = "/jails/myfirstjail";
    host.hostname = "myfirstjail.example.com";
    mount.devfs;
    devfs_ruleset = 100;
    exec.start = "/bin/sh";
    persist;
}
```

3. Reload devfs and start the jail:

```
% sudo service devfs restart
% sudo service jail start myfirstjail
```

4. Inside the jail, confirm the node:

```
% sudo jexec myfirstjail ls -l /dev/myfirst
```

5. Verify the ruleset is working by commenting out the `add path 'myfirst/*' unhide` line, restarting devfs and the jail, and observing the node disappear.

Success criteria.

- `/dev/myfirst/0` appears inside the jail with mode `0660` and group operator.
- Removing the `unhide` rule removes the node from inside the jail.
- The host continues to see the node regardless of the jail's ruleset.

Notes. Jail configuration is ordinarily covered in later chapters; this lab is a preview to demonstrate the driver-side outcome. If the lab is difficult on your system, come back to it after you have jails configured for other purposes.

8.15.8 Lab 8.8: Destroy-Dev Drain

Goal. Demonstrate the difference between `destroy_dev` and `destroy_dev_drain` when a `cdevsw` is being freed along with many `cdevs`.

Prerequisites. Completed Lab 8.3. Your driver is loaded and quiet.

Steps.

1. Review the Stage 2 detach code. The one-cdev driver does not need `destroy_dev_drain`. The lab models what goes wrong in a multi-cdev driver that does.
2. Build the `stage4-destroy-drain` variant of the driver (in the companion tree). This variant creates five `cdevs` in `attach` and uses `destroy_dev_sched` to schedule their destruction in `detach`, without draining.

3. Load the variant, then immediately unload it while a userland process is holding one of the cdevs open:

```
% sudo kldload ./stage4.ko
% sudo ./hold_myfirst 60 /dev/myfirstN/3 &
% sudo kldunload stage4
```

4. Observe the kernel log. You should see complaints or, depending on timing, a panic. The variant is deliberately unsafe.
5. Switch to the “fixed” version of the stage 4 source, which calls `destroy_dev_drain(&mycdevsw)` after the per-cdev destroy-sched calls. Repeat the load/hold/unload sequence.
6. Confirm that the fixed version unloads cleanly, waiting for the held descriptor to close before the module goes away.

Success criteria.

- The broken variant produces an observable problem (message, hang, or panic) when unloaded with a descriptor held.
- The fixed variant completes the unload cleanly.
- Reading the source makes it clear which call made the difference.

Notes. This lab deliberately triggers a bad state. Run it in a throwaway VM, not on a system you care about. The point is to build intuition for why `destroy_dev_drain` exists; once you have watched the broken path fail, you will remember to call it in multi-cdev drivers.

8.16 Challenge Exercises

These build on the labs. Take your time; none of them introduce new mechanics, they only stretch the ones you just practised.

8.16.1 Challenge 1: Use the Alias

Change `probe_myfirst.c` to open `/dev/myfirst` instead of `/dev/myfirst/0` by default. Confirm from the kernel log that your `d_open` runs, and that `devfs_set_cdevpriv` succeeds exactly once per `open(2)`. Then change the path back. You should not have to edit the driver.

8.16.2 Challenge 2: Observe Per-Open Cleanup

Add a `device_printf` inside `myfirst_fh_dtor()` that logs the `fh` pointer being freed. Run `probe_myfirst` once and confirm that exactly one destructor line appears in `dmesg` per run. Then write a tiny program that opens the device, sleeps for 30 seconds, and exits without calling `close(2)`. Confirm that the destructor still fires when the process exits. Cleanup is not a politeness; it is guaranteed.

8.16.3 Challenge 3: Experiment with devfs.rules

If you have a FreeBSD jail configured, add a `myfirst_lab` ruleset to `/etc/devfs.rules` that makes `/dev/myfirst/*` visible inside the jail. Start the jail, open the device from inside it, and confirm that the driver sees a new open. If you do not have a jail yet, skip this challenge for now and return to it after Part 7.

8.16.4 Challenge 4: Read Two More Drivers

Pick two drivers from `/usr/src/sys/dev/` that you have not read yet. Good candidates are `/usr/src/sys/dev/random/randomdev.c`, `/usr/src/sys/dev/hwpmc/hwpmc_mod.c`, `/usr/src/sys/dev/kbd/kbd.c`, or anything else short enough to skim. For each driver, find:

- The `cdevsw` definition and its `d_name`.
- The `make_dev*` call and the permission mode it sets.
- The `destroy_dev` calls, or absence of them.
- Whether the driver uses `devfs_set_cdevpriv`.
- Whether the driver creates a subdirectory under `/dev`.

Write a short paragraph for each driver classifying its device-file surface. The point is to sharpen your eye; there is no single correct taxonomy.

8.16.5 Challenge 5: devd Configuration

Write a minimal `/etc/devd.conf` rule that logs a message every time `/dev/myfirst/0` appears or disappears. The `devd` configuration format is documented in `devd.conf(5)`. A starting template:

```
notify 100 {
    match "system"      "DEVFS";
    match "subsystem"   "CDEV";
    match "cdev"        "myfirst/0";
    action              "/usr/bin/logger -t myfirst event=$type";
};
```

Install the rule, restart `devd` (`service devd restart`), load and unload the driver, then verify that `grep myfirst /var/log/messages` shows both events.

8.16.6 Challenge 6: Add a Status Node

Modify `myfirst` to expose a read-only status node alongside the data node. The status node lives at `/dev/myfirst/0.status`, mode `0444`, owner `root:wheel`. Its `d_read` returns a short plaintext string summarising the current state of the driver:

```
attached_at=12345
active_fhs=2
open_count=17
```


Hint: allocate a small fixed-size buffer in the softc, format the string under the mutex, and return it to the user with `uiomove(9)` if you have read Chapter 9, or with a manual implementation for now.

If you are not yet comfortable with `uiomove`, defer this challenge until after Chapter 9. It is a natural first use of what Chapter 9 teaches.

8.17 Error Codes for Device File Operations

Every `d_open` and `d_close` that returns a non-zero value tells devfs something specific. The `errno` values you choose are the contract between your driver and every user program that ever touches your node. Getting them right costs nothing; getting them wrong produces bug reports you will not understand at first read.

This section surveys the `errno` values that come up in practice on the device-file surface. Chapter 9 will treat the `errno` choices for `d_read` and `d_write` separately, because the data-path choices are different in character. Here we stay focused on `open`, `close`, and `ioctl`-adjacent returns.

8.17.1 The Short List

In rough order of how often you will reach for them:

- **ENXIO (No such device or address):** “The device is not in a state where it can be opened.” Use when the driver is attached but not ready, when the hardware is known to be missing, when the softc is in a transient state. The user sees `Device not configured`.
- **EBUSY (Device busy):** “The device is already open and this driver does not allow concurrent access.” Use for exclusive-open policies. The user sees `Device busy`.
- **EACCES (Permission denied):** “The credential presenting this open is not allowed.” The kernel normally catches permission failures before your handler runs, but a driver may check a secondary policy (for instance, an `ioctl`-only node that refuses opens for read) and return `EACCES` itself.
- **EPERM (Operation not permitted):** “The operation requires privilege the caller does not have.” Similar to `EACCES` in spirit but aimed at privilege distinctions (`priv_check(9)` failures) rather than UNIX file permissions.
- **EINVAL (Invalid argument):** “The call was structurally valid but the driver does not accept these arguments.” Use when `oflags` specifies a combination the driver refuses.
- **EAGAIN (Resource temporarily unavailable):** “The device could be opened in principle, but not right now.” Use this when you have a temporary shortage (a slot is full, a resource is being reconfigured) and the user should retry later. The user sees `Resource temporarily unavailable`.
- **EINTR (Interrupted system call):** Returned when a sleep inside your handler is interrupted by a signal. You will not normally return this from `d_open` because opens do not usually sleep interruptibly. It shows up more in data-path handlers.
- **ENOENT (No such file or directory):** Almost always synthesised by devfs itself when the path does not resolve. A driver rarely returns this from its own handlers.
- **ENODEV (Operation not supported by device):** “The operation itself is valid but this device does not support it.” Use when a secondary interface of the driver refuses an operation the other interface supports.
- **EOPNOTSUPP (Operation not supported):** A cousin of `ENODEV`. Used in some subsystems for similar situations.

8.17.2 Which Value for Which Situation?

Real drivers fall into patterns. Here are the patterns you will write most often.

Pattern A: Driver attached but softc not yet ready. You might hit this during a two-stage attach where the cdev is created before some initialisation completes, or during detach while the cdev still exists.

```
if (sc == NULL || !sc->is_attached)
    return (ENXIO);
```

Pattern B: Exclusive-open policy.

```
mtx_lock(&sc->mtx);
if (sc->is_open) {
    mtx_unlock(&sc->mtx);
    return (EBUSY);
}
sc->is_open = 1;
mtx_unlock(&sc->mtx);
```

This is what Chapter 7 did. Chapter 8's stage 2 removes the exclusive check because per-open state is available; the EBUSY is simply no longer needed.

Pattern C: Read-only node refusing writes.

```
if ((oflags & FWRITE) != 0)
    return (EACCES);
```

Use this when the node is conceptually read-only and opening for write is a caller mistake.

Pattern D: Privileged-only interface.

```
if (priv_check(td, PRIV_DRIVER) != 0)
    return (EPERM);
```

Returns EPERM when a non-privileged caller tries to open a node that enforces additional privilege checks beyond the filesystem mode.

Pattern E: Temporarily unavailable.

```
if (sc->resource_in_flight) {
    return (EAGAIN);
}
```

Use this when the driver can accept the open later but not now, and the user should retry.

Pattern F: Driver-specific invalid combination.

```
if ((oflags & O_NONBLOCK) != 0 && !sc->supports_nonblock) {
    return (EINVAL);
}
```

Use this when the caller's oflags specify a mode your driver does not implement.

8.17.3 Returning Errors From `d_close`

`d_close` has its own considerations. The kernel does not usually care about errors from `close`, because by the time `close(2)` returns to userland the descriptor is already gone. But `close` is still your last chance to notice a failure and log it, and some callers may check. The safest pattern is:

- Return zero from ordinary close paths.
- Return a non-zero `errno` only when something genuinely unusual happened and userland should know about it.
- When in doubt, log with `device_printf(9)` and return zero.

A driver that returns random errors from `d_close` is a driver whose tests will mysteriously fail, because most userland code ignores close errors. Save `errno` for open and for `ioctl`, where it matters.

8.17.4 Mapping Your `errno` to User Messages

The values defined in `/usr/include/errno.h` have stable textual representations through `strerror(3)` and `perror(3)`. Every `err(3)` and `warn(3)` message in a userland program will use these. A short table of the mappings:

errno	strerror text	Typical user program behavior
ENXIO	Device not configured	Wait or give up; report clearly
EBUSY	Device busy	Retry later or abort
EACCES	Permission denied	Prompt for sudo or exit
EPERM	Operation not permitted	Similar to EACCES
EINVAL	Invalid argument	Report bug in calling code
EAGAIN	Resource temporarily unavailable	Retry after a short delay
EINTR	Interrupted system call	Retry, usually in a loop
ENOENT	No such file or directory	Verify driver is loaded
ENODEV	Operation not supported by device	Report design mismatch
EOPNOTSUPP	Operation not supported	Report design mismatch

Appendix E of this book collects the full list of kernel `errno` values and their meanings. For Chapter 8 the list above covers everything you will reach for on the device-file surface.

8.17.5 A Quick Checklist Before You Pick an `errno`

When you are unsure which `errno` fits, ask three questions:

1. **Is the problem about identity?** “This device cannot be opened now” is `ENXIO`. “This device does not exist” is `ENOENT`. Rarely the driver’s call; `devfs` usually handles it.
2. **Is the problem about permission?** “You do not have permission” is `EACCES`. “You lack a specific privilege” is `EPERM`.
3. **Is the problem about arguments?** “The call was structurally fine but the driver will not accept these arguments” is `EINVAL`.

When two `errno` values could plausibly fit, pick the one whose textual representation matches what you would want a frustrated user to read. Remember that `errno` values become error messages in tools you do not control, and the clearer the mapping between kernel intent and user-facing text, the more kindly your driver will be reviewed.

8.17.6 A Short Narrative: `errno` Chosen Three Times

To make the abstract concrete, here are three small scenes drawn from real driver review conversations. Each is about the choice of a single `errno` value.

Scene 1. The too-early open.

A driver attaches an onboard sensor. The sensor takes a hundred milliseconds after power-on to produce valid data. During those hundred milliseconds, user programs that try to read will get garbage.

The first draft of the driver returns `EAGAIN` from `d_open` during the warm-up window. The reviewer flags it. `EAGAIN` means “retry later”, which is fine, but the user-facing text is “Resource temporarily unavailable”, and that does not match what the user is seeing: the device exists and can in principle be opened, but it is not producing data yet.

The revised draft returns `ENXIO` during the warm-up. The user sees “Device not configured”, which is closer to the truth. A well-written userland program can special-case that `errno` if it wants to wait for the device. A typical tool will print a clear message and exit.

Lesson: think about what the user sees, not just what you mean internally.

Scene 2. The wrong permission error.

A driver has a configurable mode: a `sysctl` can set it to “read-only”. When the `sysctl` is set, `d_write` returns an error. The first draft returns `EPERM`. The reviewer flags it. `EPERM` is about privilege; the kernel uses it when a specific `priv_check(9)` call fails. But in this driver, no privilege check is being performed; the device is simply in a read-only state.

The revised draft returns `EROFS`, “Read-only file system”. The textual mapping is almost perfect for this scenario.

Lesson: the nearer `errno` value is usually the better `errno` value. Do not default to `EPERM` for every refusal.

Scene 3. The busy file.

A driver that enforces exclusive access returns `EBUSY` from `d_open` when a second opener arrives. That is correct. In code review, one reviewer points out that the driver also returns `EBUSY` from a control-node `ioctl` that

refuses during an in-progress reconfiguration. The review argument is that those are different situations and the two uses of `EBUSY` will confuse operators who are reading logs.

The discussion lands on a compromise: `EBUSY` for the open-path exclusive check, `EAGAIN` for the reconfiguration-in-progress case. The distinction is that the open-path refusal is “will be busy until the other user closes”, while the reconfiguration refusal is “retry in a moment, it will clear on its own”.

Lesson: two situations that feel similar may map to different `errno` values if the reasoning about the user’s next action differs.

These scenes are small, but the principle is not. Every `errno` value is a hint to the user about what to do next. Choose it with the user’s perspective in view, not only yours.

8.17.7 Using `err(3)` and `warn(3)` to Exercise `errno` Values

The `err(3)` family in FreeBSD’s `libc` prints a clean “program: message: `errno`-string” when an operation fails. Your userland probes use `err(3)` because it is the shortest path to a readable error. You can verify your driver’s `errno` choices by running a probe that deliberately triggers each one:

```
fd = open("/dev/myfirst/0", O_RDWR);
if (fd < 0)
    err(1, "open /dev/myfirst/0");
```

When the driver returns `EBUSY`, the program prints:

```
probe_myfirst: open /dev/myfirst/0: Device busy
```

When the driver returns `ENXIO`, the program prints:

```
probe_myfirst: open /dev/myfirst/0: Device not configured
```

Run the probe against each of the error cases you can construct. Read the messages out loud. If any of them would confuse a user who had not read your driver’s source code, reconsider the `errno`.

8.17.8 `errno` Values Your Driver Should Almost Never Return

For balance, a list of values that rarely fit a device-file open or close:

- **ENOMEM**: let the `malloc` call report this by returning it through your function, but do not invent it.
- **EIO**: reserved for hardware I/O errors. If your device has no hardware, this value is out of place.
- **EFAULT**: used when userland hands the kernel a bad pointer. On the open path you rarely touch user pointers, so `EFAULT` does not fit.
- **ESRCH**: “No such process”. Unlikely to be right for a device-file operation.
- **ECHILD**: process-relationship `errno`. Not applicable.
- **EDOM** and **ERANGE**: math errors. Not applicable.

When in doubt, if the value does not appear in the Chapter 8 “Short List” earlier in this section, it is almost certainly wrong for an open or close. Save the unusual values for the unusual operations that genuinely produce them.

8.18 Tools for Inspecting /dev

Several small utilities are worth knowing about, because once you reach Chapter 9 you will be leaning on them to confirm behavior quickly. This section introduces each in enough depth to use it, and ends with two short troubleshooting walkthroughs.

8.18.1 ls -l for Permissions and Existence

The first stop. `ls -l /dev/yourpath` confirms existence, type, ownership, and mode. If the node is missing after a load, your `make_dev_s` likely failed; check `dmesg` for the error code.

`ls -l` on a devfs directory works the way you expect: `ls -l /dev/myfirst` lists the entries in the subdirectory. Combined with `-d`, it reports on the directory itself:

```
% ls -ld /dev/myfirst
dr-xr-xr-x  2 root  wheel  512 Apr 17 10:02 /dev/myfirst
```

The mode on a devfs subdirectory is 0555 by default, and it is not directly configurable through `devfs.conf`. The subdirectory exists only because at least one node is inside it; when the last node inside disappears, the directory disappears too.

8.18.2 stat and stat(1)

`stat(1)` prints a structured view of any node. The default output is verbose and includes timestamps. A more useful form is a custom format:

```
% stat -f '%Sp %Su %Sg %T %N' /dev/myfirst/0
crw-rw---- root operator Character Device /dev/myfirst/0
```

The placeholders are documented in `stat(1)`. The five above are permissions, user name, group name, file type description, and path. This form is useful inside scripts that need a stable text representation.

For comparing two paths to check that they resolve to the same cdev, `stat -f '%d %i %Hr,%Lr'` prints the device of the filesystem, the inode, and the major and minor components of `rdev`. On two devfs nodes that refer to the same cdev, the `rdev` component will match.

8.18.3 fstat(1): Who Has It Open?

`fstat(1)` lists every open file on the system. Filtered to a device path, it tells you which processes have the node open:

```
% fstat /dev/myfirst/0
USER      CMD          PID   FD MOUNT      INUM  MODE      SZ|DV R/W NAME
root      probe_myfir 1234   3 /dev          4 crw-rw---- 0,90 rw  /dev/myfirst/0
```

This is the tool that solves the “kldunload returns EBUSY and I do not know why” puzzle. Run it against your node, identify the offending process, and either wait for it to finish or terminate it.

`fstat -u username` filters by user, useful when you suspect a particular user’s daemons are holding the node. `fstat -p pid` inspects one process.

8.18.4 `procstat -f`: A Process-First View

`fstat(1)` lists files and tells you who holds them. `procstat -f pid` does the inverse: it lists the files held by a given process. When you have the PID of a running program and want to confirm which device nodes it currently has open, this is the tool:

```
% procstat -f 1234
  PID COMM          FD T V FLAGS      REF  OFFSET PRO NAME
1234 probe_myfirst    3 v c rw-----   1      0    /dev/myfirst/0
```

Column `T` shows the file type (`v` for vnode, which includes device files), and column `V` shows the vnode type (`c` for character-device vnode). This is the quickest way to confirm what a debugger shows you.

8.18.5 `devinfo(8)`: The Newbus Side

`devinfo(8)` does not look at devfs at all. It walks the Newbus device tree and prints the device hierarchy. Your `myfirst0` child of `nexus0` shows up there regardless of whether a `cdev` exists:

```
% devinfo -v
nexus0
  myfirst0
  pcib0
    pci0
      <...lots of PCI children...>
```

This is the tool you reach for when something is missing from `/dev` and you need to check whether the device itself attached. If `devinfo` shows `myfirst0` but `ls /dev` does not, your `make_dev_s` failed. If neither shows the device, your `device_identify` or `device_probe` did not create the child. Two different bugs, two different fixes.

The `-r` flag filters to the Newbus hierarchy rooted at a specific device, which becomes useful in complex systems with lots of PCI devices.

8.18.6 devfs(8): Rulesets and Rules

`devfs(8)` is the low-level administrative interface to `devfs` rulesets. You met it in Section 10. Three subcommands come up often:

- `devfs rule showsets` lists ruleset numbers currently loaded.
- `devfs rule -s N show` prints the rules inside ruleset `N`.
- `devfs rule -s N add path 'pattern' action args` adds a rule at runtime.

Rules added at runtime do not persist; to make them permanent, add them to `/etc/devfs.rules` and run `service devfs restart`.

8.18.7 sysctl dev.* and Other Hierarchies

`sysctl dev.myfirst` prints every `sysctl` variable under your driver's namespace. From Chapter 7 you already have a `dev.myfirst.0.stats` tree. Reading it confirms the softc is present, the attach ran, and the counters are advancing.

`Sysctls` are a complementary surface to `/dev`. They are primarily for observability; they are cheaper to read than opening a device; they have no file-descriptor cost. When a piece of information is simple enough to be a number or a short string, consider exposing it as a `sysctl` rather than as a read on the device node.

8.18.8 kldstat: Is the Module Loaded?

When a node is missing, the question “is my driver even loaded?” is worth asking first.

```
% kldstat | grep myfirst
8      1 0xfffffffff82a00000      3a50 myfirst.ko
```

If you see the module in `kldstat`, the module is in the kernel. If `devinfo` shows the device but `ls /dev` does not show the node, the issue is inside your driver. If `kldstat` does not show the module, the issue is outside: you forgot to `kldload`, or the load failed. Check `dmesg`.

8.18.9 dmesg: The Log of What Happened

Every `device_printf` and `printf` call from a driver ends up in the kernel message buffer, and `dmesg` (or `dmesg -a`) prints it. When something goes wrong on this surface, `dmesg` is the first place to look:

```
% dmesg | tail -20
```

Your attach and detach messages, any `make_dev_s` failures, and any panic messages from destruction paths land here. Get in the habit of watching `dmesg` with a second terminal open to `tail -f /var/log/messages` during development.

8.18.10 Troubleshooting Walkthrough 1: The Node Is Missing

A checklist for “I expected `/dev/myfirst/0` to exist and it does not”.

1. Is the module loaded? `kldstat | grep myfirst`.
2. Did attach run? `devinfo -v | grep myfirst`.
3. Did `make_dev_s` succeed? `dmesg | tail` should show your attach-success message.
4. Is `devfs` mounted on `/dev`? `mount | grep devfs`.
5. Are you looking at the right path? If your format string was `"myfirst%d"`, the node is `/dev/myfirst0`, not `/dev/myfirst/0`. Typos happen.
6. Is a `devfs.rules` entry hiding the node? `devfs rule showsets` and inspect.

Nine times out of ten, one of the first three questions yields the answer.

8.18.11 Troubleshooting Walkthrough 2: `kldunload` Returns `EBUSY`

A checklist for “I can load my module but cannot unload it”.

1. Is the node still open? `fstat /dev/myfirst/0` shows the holder.
2. Is your detach returning `EBUSY` itself? Check `dmesg` for a message from your driver. Stage 2’s detach returns `EBUSY` when `active_fhs > 0`.
3. Is a `devfs.conf` link pointing at your node? The link can keep a reference if the target is held open.
4. Is a kernel thread stuck inside one of your handlers? Look for a `Still N threads in foo` message in `dmesg`. If present, you need a `d_purge`.

Most `EBUSY`s are open descriptors. The other cases are rare.

8.18.12 A Note on Habits

None of these tools are unusual. They are the everyday instruments of FreeBSD administration. What matters is the habit of reaching for them in a known order when something looks wrong. The first three times you debug a missing node, you will grope for the right tool; the fourth time, the order will feel automatic. Build that reflex now, while the problems are small.

8.19 Pitfalls and Things Worth Watching

A field guide to the mistakes that catch beginners most often. Each one names the symptom, the cause, and the cure.

- **Creating the device node before the softc is ready.** *Symptom:* `open` causes a NULL dereference as soon as the driver loads. *Cause:* `si_drv1` still unset, or a `softc` field that `open()` consults has not been initialized. *Cure:* set `mda_si_drv1` in `make_dev_args` and finalize `softc` fields before the `make_dev_s` call. Think of `make_dev_s` as publishing, not preparing.
- **Destroying the softc before the device node.** *Symptom:* occasional panics during or shortly after `kldunload`. *Cause:* reversing the order of teardown in `detach()`. *Cure:* always destroy the `cdev` first, then the alias, then the lock, then the `softc`. The `cdev` is the door; close it before dismantling the rooms behind it.

- **Storing per-open state on the cdev.** *Symptom:* works fine with one user, garbled state with two. *Cause:* read positions or similar per-descriptor data stored in `si_drv1` or in the `softc`. *Cure:* move them into a struct `myfirst_fh` and register it with `devfs_set_cdevpriv`.
- **Forgetting that /dev changes are not persistent.** *Symptom:* a `chmod` you ran by hand is gone after a reboot or module reload. *Cause:* `devfs` is live, not on disk. *Cure:* put the change in `/etc/devfs.conf` and service `devfs` restart.
- **Leaking the alias on detach.** *Symptom:* `kldunload` returns `EBUSY` and the driver is wedged. *Cause:* the alias `cdev` is still live. *Cure:* call `destroy_dev(9)` on the alias before the primary in `detach()`.
- **Calling devfs_set_cdevpriv twice.** *Symptom:* the second call returns `EBUSY` and your handler returns the error to the user. *Cause:* two independent paths in `open` both tried to register private data, or the handler ran twice for the same `open`. *Cure:* audit the code path so exactly one successful `devfs_set_cdevpriv` happens per `d_open` invocation.
- **Allocating fh without freeing it on the error path.** *Symptom:* steady memory leak correlated with failed opens. *Cause:* `devfs_set_cdevpriv` returned an error and the allocation was abandoned. *Cure:* on any error after `malloc` and before a successful `devfs_set_cdevpriv`, free the allocation explicitly.
- **Confusing aliases with symlinks.** *Symptom:* permissions set through `devfs.conf` on a link do not match what the driver advertises on the primary. *Cause:* mixing both mechanisms on the same name. *Cure:* pick one tool per name; use aliases when the driver owns the name, symlinks when operator convenience is the goal.
- **Using wide-open modes for “just testing”.** *Symptom:* a driver that shipped to staging with `0666` suddenly needs to have that narrowed without breaking consumers. *Cause:* a temporary lab mode turned into a default. *Cure:* default to `0600`, widen only when a concrete consumer asks, and note the reason in a comment next to the `mda_mode` line.
- **Using make_dev in new code.** *Symptom:* the driver compiles and works, but a reviewer flags the call. *Cause:* `make_dev` is the oldest form of the family and panics on failure. *Cure:* use `make_dev_s` with a populated struct `make_dev_args`. The newer form is easier to read, easier to error-check, and friendlier to future API additions. *How you catch it earlier:* run `mandoc -Tlint` on your driver and read the `SEE ALSO` in `make_dev(9)`.
- **Forgetting D_VERSION.** *Symptom:* the driver loads but the first open returns a cryptic failure, or the kernel prints a `cdevsw` version mismatch. *Cause:* the `d_version` field of `cdevsw` was left zero. *Cure:* set `.d_version = D_VERSION` as the first field in every `cdevsw` literal. *How you catch it earlier:* a code template that includes the field keeps you from ever typing a `cdevsw` without it.
- **Shipping with D_NEEDGIANT “because it compiled”.** *Symptom:* the driver works but every operation serialises behind the Giant lock, making SMP-heavy workloads slow. *Cause:* the flag was copied from an older driver, or added to silence a warning, and never removed. *Cure:* delete the flag. If your driver actually needs Giant to hold together, it has a real locking bug that needs real fixing, not a flag.
- **Hard-coding the hex identifier in test scripts.** *Symptom:* a test fails on a slightly different machine because the `0x5a` in `ls -l` output is different there. *Cause:* `devfs`’s `rdev` identifier is not stable across reboots, kernels, or systems. *Cure:* compare `stat -f '%d %i'` across two paths to check alias equivalence rather than scraping `ls -l` for the hex identifier.
- **Assuming devfs.conf runs before your driver loads.** *Symptom:* a `devfs.conf` line for your driver’s node does not take effect after `kldload`. *Cause:* service `devfs` start runs early in boot, before modules loaded at runtime. *Cure:* service `devfs` restart after loading the driver, or statically compile the driver so its nodes exist before `devfs` starts.
- **Relying on node names with non-POSIX characters.** *Symptom:* shell scripts break with quoting errors;

`devfs.rules` patterns fail to match. *Cause:* the node name uses spaces, colons, or non-ASCII characters. *Cure:* stick to lowercase ASCII letters, digits, and the three separators `/`, `-`, `..`. Other characters will sometimes work and sometimes not, and the “sometimes not” always surfaces at the worst moment.

- **Leaking per-open state on the error path of `d_open`.** *Symptom:* subtle memory leak, detected much later by running a stress test for hours. *Cause:* `malloc` succeeded, `devfs_set_cdevpriv` failed, and the allocation was abandoned without being freed. *Cure:* every error path in `d_open` between `malloc` and the successful `devfs_set_cdevpriv` must free the allocation. Writing the error path first, before the success path, is a useful habit.
- **Registering `devfs_set_cdevpriv` twice in the same open.** *Symptom:* the second call returns `EBUSY` and the user sees `Device busy` on open, for no reason they can discern. *Cause:* two independent code paths in `d_open` both try to attach private data, or the open handler runs twice for the same file. *Cure:* audit the code path so exactly one successful `devfs_set_cdevpriv` happens per `d_open` invocation. If the driver genuinely wants to replace the data, use `devfs_clear_cdevpriv(9)` first, but this is almost always a sign the design needs a rethink.

8.19.1 Pitfalls That Are Really About Lifecycle

A separate cluster of pitfalls comes from confusion about lifecycle. They are worth calling out explicitly.

- **Freeing the `softc` before the `cdev` is destroyed.** *Symptom:* a panic shortly after `kldunload`, usually a `NULL` dereference or a use-after-free in a handler. *Cause:* the driver tore down `softc` state in `detach` before `destroy_dev` finished draining the `cdev`, and an in-flight handler then dereferenced the freed state. *Cure:* destroy the `cdev` first and rely on its draining behavior; only tear down `softc` after. *How you catch it earlier:* run any of the stress tests while watching `dmesg` for kernel panics; the race is easy to hit on an SMP system with moderate load.
- **Assuming `destroy_dev` returns immediately.** *Symptom:* a deadlock, usually in a handler that holds a lock and then calls a function that ends up needing the same lock. *Cause:* `destroy_dev` blocks until in-flight handlers return; if the caller holds a lock one of those handlers needs, the system deadlocks. *Cure:* never call `destroy_dev` while holding a lock an in-flight handler might need. For the common case in `detach`, hold nothing.
- **Forgetting to set `is_attached = 0` on an error unwind.** *Symptom:* subtle misbehavior after a failed load-unload-reload cycle; handlers believe the device is still attached and try to use freed state. *Cause:* a `goto fail_*` path that did not clear the flag. *Cure:* the single-label unwind pattern from Chapter 7; the last fail label always clears `is_attached` before returning.

8.19.2 Pitfalls in Permission and Policy

Two categories of mistakes around permission tend to surface long after the driver ships.

- **Assuming a node is “only visible to root” because you created it with `0600`.** *Symptom:* a security review flags the node as reachable from a jail that should not see it. *Cause:* mode alone does not filter jail visibility; `devfs.rules` is the filter, and the default may be inclusive enough to pass the node to the jail. *Cure:* if the node must not be visible inside jails, ensure the default jail ruleset hides it. `devfs_rules_hide_all` is the conservative starting point.

- **Relying on `devfs.conf` to keep a node secret on a shared lab machine.** *Symptom:* a collaborator changes `devfs.conf` and the node becomes readable by everyone. *Cause:* `devfs.conf` is operator policy; any operator with write access to `/etc` can change it. *Cure:* the driver's own baseline should be safe in the absence of any `devfs.conf` entry. Treat `devfs.conf` as a permission widener, never as a permission tightener relative to a fundamentally safe baseline.

8.19.3 Pitfalls in Observability

A handful of pitfalls have nothing to do with code but a lot to do with how easy your driver is to debug.

- **Logging every open and close at full volume.** *Symptom:* the kernel message buffer fills with routine driver noise; real errors are harder to find. *Cause:* the driver uses `device_printf` for every `d_open` and `d_close`. *Cure:* gate routine messages with `if (bootverbose)` or remove them entirely once the driver is stable. Leave `device_printf` for lifecycle events and for genuine errors.
- **Not exposing enough sysctls to diagnose unusual states.** *Symptom:* a user reports a bug, you cannot tell what the driver thinks is happening, and adding diagnostics to the driver requires a rebuild and a reload. *Cause:* the sysctl tree is sparse. *Cure:* expose counters generously. `active_fhs`, `open_count`, `read_count`, `write_count`, `error_count` are cheap. Add an `attach_ticks` and a `last_event_ticks` to let operators tell how long the driver has been up and how recently it was active.

8.20 A Final Study Plan

If you want to deepen your grasp of the material beyond the labs and challenges, here is a suggested plan across the week after you finish the chapter.

Day 1: Re-read a section. Pick any single section that felt shakiest on first reading and read it again with the companion tree open next to the text. Just read. Do not try to code yet.

Day 2: Rebuild stage 2 from scratch. Starting from Chapter 7's stage 2 source, make every change the Chapter 8 stages describe, one commit at a time. Diff your work against the companion tree at each stage.

Day 3: Break the driver on purpose. Introduce three different bugs, one at a time: skip the destructor, forget to destroy the alias, return the wrong `errno`. Predict what each bug does. Run the probes. See whether the failure matches your prediction.

Day 4: Read `null.c` and `led.c` end-to-end. Two small drivers, focused on the device-file surface. Write one paragraph on each summarising what you noticed.

Day 5: Add the status node from Challenge 6. Implement the read-only status node with a hand-rolled `uiomove`-equivalent for now; Chapter 9 will show the real idiom.

Day 6: Try the jail lab. If you have not done Lab 8.7 yet, do it now. Jails are worth the effort to set up because later chapters will assume familiarity.

Day 7: Move on. Do not wait to feel like you have “mastered” Chapter 8. You will return to its material naturally as later chapters build on it. The way to become fluent is to keep building; the way to become stuck is to wait for perfection.

8.21 Companion Tree Quick Reference

Because the companion source tree is part of how this chapter teaches, a quick index of what lives where may help you find things during the labs and challenges.

8.21.1 Driver Stages

- `examples/part-02/ch08-working-with-device-files/stage0-structured-name/` is the Lab 8.1 output: Chapter 7's stage 2 driver with the node moved to `/dev/myfirst/0` and ownership tightened to `root:operator 0660`.
- `examples/part-02/ch08-working-with-device-files/stage1-alias/` is the Lab 8.2 output: stage 0 plus `make_dev_alias("myfirst")`.
- `examples/part-02/ch08-working-with-device-files/stage2-perhandle/` is the Lab 8.3 output: stage 1 plus `devfs_set_cdevpriv` per-open state and removal of the exclusive-open check. This is the driver most of the chapter's other exercises use.
- `examples/part-02/ch08-working-with-device-files/stage3-two-nodes/` is the Lab 8.5 output: adds a control node at `/dev/myfirst/%d.ctl` with its own `cdevsw` and a narrower permission mode.
- `examples/part-02/ch08-working-with-device-files/stage4-destroy-drain/` is the Lab 8.8 exercise: a multi-cdev driver demonstrating the difference between `destroy_dev` alone and `destroy_dev_drain`. Build with `make CFLAGS+=-DUSE_DRAIN=1` for the correct variant.

8.21.2 Userland Probes

- `userland/probe_myfirst.c`: one-shot open, read, close.
- `userland/hold_myfirst.c`: open and sleep without closing, to exercise the `cdevpriv` destructor on process exit.
- `userland/stat_myfirst.c`: report `stat(2)` metadata for one or more paths; useful for comparing alias and primary.
- `userland/parallel_probe.c`: open `N` descriptors from one process, hold, close all.
- `userland/stress_probe.c`: loop open/close to shake leaks out.
- `userland/devd_watch.sh`: subscribe to `devd(8)` events and filter for `myfirst`.

8.21.3 Configuration Samples

- `devfs/devfs.conf.example`: Lab 8.4 persistence entries.
- `devfs/devfs.rules.example`: Lab 8.7 jail ruleset.
- `devfs/devd.conf.example`: Challenge 5 `devd` rule.
- `jail/jail.conf.example`: Lab 8.7 jail definition that references ruleset 100.

8.21.4 How the Stages Differ

Every stage is a diff against Chapter 7's stage 2. A useful first exercise after reading the chapter is to run `diff` between each pair of stages and read the result. The changes are small enough to understand line by line, and

the diff tells the progressive story of the chapter's code changes more compactly than re-reading each source file.

```
% diff -u examples/part-02/ch07-writing-your-first-driver/stage2-final/myfirst.c \  
↔ examples/part-02/ch08-working-with-device-files/stage0-structured-name/myfirst.c  
  
% diff -u  
↔ examples/part-02/ch08-working-with-device-files/stage0-structured-name/myfirst.c \  
    examples/part-02/ch08-working-with-device-files/stage1-alias/myfirst.c  
  
% diff -u examples/part-02/ch08-working-with-device-files/stage1-alias/myfirst.c \  
    examples/part-02/ch08-working-with-device-files/stage2-perhandle/myfirst.c
```

Each diff should be a handful of additions and no unexpected subtractions. If you see surprising changes, the chapter text is where the reasoning is.

8.21.5 On Re-using This Tree Later

The stages here are not meant to be the “final” driver. They are snapshots that correspond to checkpoints in the chapter. When you continue into Chapter 9, you will edit stage 2 in place, and it will keep growing. By the time you reach the end of Part 2, the driver has evolved into something much richer than any one stage captures. That is the point: each chapter adds a layer, and the companion tree is there to show each layer individually so you can see the progression.

8.22 A Closing Reflection on Interfaces

Every chapter in this book teaches something different, but a few chapters teach something that runs across the whole practice of driver writing. Chapter 8 is one of them. The specific subject is device files, but the broader subject is **interface design**: how do you shape the boundary between a piece of code you control and a world you do not?

The UNIX philosophy has an answer that has survived half a century. Make the boundary look as much like an ordinary file as you can. Let the existing vocabulary of `open`, `read`, `write`, and `close` do the heavy lifting. Choose names and permissions so operators can reason about them without reading your source. Expose only what the user needs, and no more. Clean up after yourself so aggressively that the kernel can tell when you have lost track of something. Document every choice you make with a comment, a `device_printf`, or a `sysctl`.

None of these principles are unique to device files. They show up again in network interface design, in storage layering, in the kernel's internal APIs, in the userland tools that talk to the kernel. The reason we spent a whole chapter on the small surface under `/dev` is that the same habits, practised here on something tangible and bounded, will serve you in every layer of the kernel you go on to touch.

When you read a driver in `/usr/src/sys` that feels elegant, one of the reasons is almost always that its device-file surface is narrow and honest. When you read a driver that feels tangled, one of the reasons is almost

always that its device-file surface was designed in haste, or widened in response to a short-term pressure, and never narrowed again. The goal of this chapter has been to help you notice that difference, and to give you the vocabulary and the discipline to write the first kind of driver rather than the second.

8.23 Wrapping Up

You now understand the layer between your driver and user space well enough to shape it deliberately. Specifically:

- `/dev` is not a directory on disk. It is a `devfs` view of live kernel objects.
- A `struct cdev` is the kernel-side identity of your node. A `vnode` is how the VFS reaches it. A `struct file` is how an individual `open(2)` sits in the kernel.
- `mda_uid`, `mda_gid`, and `mda_mode` set the baseline of what `ls -l` shows. `devfs.conf` and `devfs.rules` layer operator policy on top.
- The node's path is whatever your format string says, slashes included. Subdirectories under `/dev` are a normal and welcome way to group related nodes.
- `make_dev_alias(9)` lets one `cdev` answer to more than one name. Remember to destroy the alias when you tear down the primary.
- `devfs_set_cdevpriv(9)` gives each `open(2)` its own state, with automatic cleanup. This is the tool you will lean on hardest in the next chapter.

The driver you carry into Chapter 9 is the same `myfirst` you started with, but with a cleaner name, a saner set of permissions, and per-open state ready to hold the read positions, byte counters, and small book keeping that real I/O will need. Keep the file open. You will be editing it again soon.

8.23.1 A Short Self-Check

Before moving on, make sure you can answer each of the following without looking back at the chapter. If any answer is fuzzy, revisit the relevant section before starting Chapter 9.

1. What is the difference between a `struct cdev`, a `devfs vnode`, and a `struct file`?
2. Where does `make_dev_s(9)` get the ownership and mode for the node it creates?
3. Why does a `chmod` on `/dev/yournode` not survive a reboot?
4. What does `make_dev_alias(9)` do, and how does it differ from `link` in `devfs.conf`?
5. When does the destructor registered with `devfs_set_cdevpriv(9)` run, and when does it *not* run?
6. How would you confirm from userland that two paths resolve to the same `cdev`?
7. Why is `D_VERSION` required on every `cdevsw`, and what happens when it is missing?
8. When would you choose `make_dev_s` over `make_dev_p`, and why?
9. What guarantees does `destroy_dev(9)` give you about threads currently inside your handlers?
10. If a jail does not see `/dev/myfirst/0` but the host does, where is the policy that hid it, and how would you inspect it?

If you can answer all ten in your own words, the next chapter will feel like a natural continuation rather than a jump.

8.23.2 Recap Organised by Topic

The chapter covered a lot. Here is a short re-organisation of the material by topic rather than by section, so you can anchor what you learned.

On the relationship between the kernel and the filesystem:

- `devfs` is a virtual filesystem that presents the kernel's live collection of `struct cdev` objects as file-like nodes under `/dev`.
- It has no on-disk storage. Every node reflects something the kernel currently holds.
- It supports only a small, well-defined set of operations on its nodes.
- Changes made interactively (with `chmod`, for instance) do not persist. Persistent policy lives in `/etc/devfs.conf` and `/etc/devfs.rules`.

On the objects your driver interacts with:

- A `struct cdev` is the kernel-side identity of a device node. One per node, regardless of how many file descriptors point at it.
- The `struct cdevsw` is the dispatch table your driver provides. It maps each kind of operation to a handler in your code.
- `struct file` and the `devfs vnode` sit between the user's file descriptor and your `cdev`. They carry per-open state and route operations.

On creating and destroying nodes:

- `make_dev_s(9)` is the modern, recommended way to create a `cdev`. Fill in a `struct make_dev_args`, pass it in, get a `cdev` back.
- `make_dev_alias(9)` creates a second name for an existing `cdev`. Aliases are first-class `cdevs`; the kernel keeps them in step with the primary.
- `destroy_dev(9)` destroys a `cdev` synchronously, draining in-flight handlers. Its cousins `destroy_dev_sched` and `destroy_dev_drain` cover deferred and sweep cases respectively.

On per-open state:

- `devfs_set_cdevpriv(9)` attaches a driver-provided pointer to the current file descriptor, along with a destructor.
- `devfs_get_cdevpriv(9)` retrieves that pointer inside later handlers.
- The destructor fires exactly once per successful set call, when the last reference to the file descriptor drops.
- This is the primary mechanism for per-open bookkeeping in modern FreeBSD drivers.

On policy:

- The driver sets a baseline mode, uid, and gid in the call to `make_dev_s`.
- `/etc/devfs.conf` can adjust those per-node on host `devfs` mounts.
- `/etc/devfs.rules` can define named rulesets that filter and adjust per-mount, typically for jails.
- Three layers can act on the same `cdev`, and the order matters.

On userland:

- `ls -l`, `stat(1)`, `fstat(1)`, `procstat(1)`, `devinfo(8)`, `devfs(8)`, `sysctl(8)`, and `kldstat(8)` are the everyday tools for inspecting and manipulating the surface your driver exposes.
- Small userland C programs that open, read, close, and stat the device are worth writing. They give you control over timing and let you test edge cases cleanly.

On discipline:

- Default to narrow permissions and widen only when a concrete consumer asks.
- Use named constants (`UID_ROOT`, `GID_WHEEL`) rather than raw numbers.
- Destroy in the reverse order of creation.
- Free allocations on every error path before returning.
- Log lifecycle events with `device_printf(9)` so that `dmesg` tells the story of what your driver is doing.

That is a lot. You do not need to hold it all at once. The labs and challenges are where the material becomes muscle memory; the text is only the reading guide.

8.23.3 Looking Ahead to Chapter 9

In Chapter 9 we will fill in `d_read` and `d_write` properly. You will learn how the kernel moves bytes between user memory and kernel memory with `uiomove(9)`, why `struct uio` looks the way it does, and how to design a driver that is safe against short reads, short writes, misaligned buffers, and misbehaving user programs. The per-open state you just wired in will carry read offsets and write state. The alias will keep the old user interfaces working while the driver grows. And the permissions model you set up here will keep your lab scripts honest as you start sending real data through.

Specifically, Chapter 9 will need the fields you added to `struct myfirst_fh` for two things. The reads counter will grow a matching `read_offset` field so each descriptor remembers where it was in a synthesised data stream. The writes counter will be joined by a small ring buffer that `d_write` appends into and `d_read` drains from. The `fh` pointer you retrieve with `devfs_get_cdevpriv` in every handler will be the entry point for all of that state.

The alias you created in Lab 8.2 will keep working without any changes: both `/dev/myfirst` and `/dev/myfirst/0` will produce data, and per-descriptor state will be independent between them.

The permissions you set in Lab 8.1 and Lab 8.4 will remain the correct defaults for development: tight enough to force a conscious `sudo` when a raw user reaches for the device, open enough that a test harness in the operator group can run the data-path tests without escalating.

You have built a well-shaped door. In the next chapter, the rooms behind it come alive.

8.24 Reference: `make_dev_s` and `cdevsw` at a Glance

This reference collects the most useful declarations and flag values in one place, cross-referenced to the sections of the chapter that explained each one. Keep it open while you write your own drivers; most of the mistakes that cost a day are mistakes about one of these values.

8.24.1 Canonical make_dev_s Skeleton

A disciplined template for a single-node driver:

```
struct make_dev_args args;
int error;

make_dev_args_init(&args);
args.mda_devsw    = &myfirst_cdevsw;
args.mda_uid      = UID_ROOT;
args.mda_gid      = GID_OPERATOR;
args.mda_mode     = 0660;
args.mda_si_drv1  = sc;

error = make_dev_s(&args, &sc->cdev, "myfirst/%d", sc->unit);
if (error != 0) {
    device_printf(dev, "make_dev_s: %d\n", error);
    /* unwind and return */
    goto fail;
}
```

8.24.2 Canonical cdevsw Skeleton

```
static struct cdevsw myfirst_cdevsw = {
    .d_version = D_VERSION,
    .d_name    = "myfirst",
    .d_open    = myfirst_open,
    .d_close   = myfirst_close,
    .d_read    = myfirst_read,
    .d_write   = myfirst_write,
    .d_ioctl   = myfirst_ioctl,    /* add in Chapter 25 */
    .d_poll    = myfirst_poll,    /* add in Chapter 10 */
    .d_kqfilter = myfirst_kqfilter, /* add in Chapter 10 */
};
```

Fields omitted are equivalent to NULL, which the kernel interprets either as “not supported” or “use the default behavior” depending on the field.

8.24.3 The make_dev_args Structure

From `/usr/src/sys/sys/conf.h`:

```

struct make_dev_args {
    size_t      mda_size;           /* set by make_dev_args_init */
    int         mda_flags;         /* MAKEDEV_* flags */
    struct cdevsw *mda_devsw;      /* required */
    struct ucred *mda_cr;         /* usually NULL */
    uid_t       mda_uid;          /* see UID_* in conf.h */
    gid_t       mda_gid;          /* see GID_* in conf.h */
    int         mda_mode;         /* octal mode */
    int         mda_unit;         /* unit number (0..INT_MAX) */
    void        *mda_si_drv1;     /* usually the softc */
    void        *mda_si_drv2;     /* second driver pointer */
};

```

8.24.4 The MAKEDEV Flag Word

Flag	Meaning
MAKEDEV_REF	Add an extra reference at creation.
MAKEDEV_NOWAIT	Do not sleep for memory; return ENOMEM if tight.
MAKEDEV_WAITOK	Sleep for memory (default for make_dev_s).
MAKEDEV_ETERNAL	Mark the cdev as never-to-be-destroyed.
MAKEDEV_CHECKNAME	Validate the name; return error on bad names.
MAKEDEV_WHTOUT	Create a whiteout entry (stacked filesystems).
MAKEDEV_ETERNAL_KLD	MAKEDEV_ETERNAL when static, zero when built as KLD.

8.24.5 The cdevsw d_flags Field

Flag	Meaning
D_TAPE	Category hint: tape device.
D_DISK	Category hint: disk device (legacy; modern disks use GEOM).
D_TTY	Category hint: TTY device.
D_MEM	Category hint: memory device such as /dev/mem.
D_TRACKCLOSE	Call d_close for every close(2) on every descriptor.
D_MMAP_ANON	Anonymous-mmap semantics for this cdev.
D_NEEDGIANT	Force Giant-lock dispatch. Avoid in new code.
D_NEEDMINOR	Driver uses clone_create(9) for minor number allocation.

8.24.6 Common UID and GID Constants

Constant	Numeric	Purpose
UID_ROOT	0	Superuser. Default owner for most nodes.
UID_BIN	3	Daemon executables.
UID_UUCP	66	UUCP subsystem.
UID_NOBODY	65534	Unprivileged placeholder.
GID_WHEEL	0	Trusted administrators.
GID_KMEM	2	Read access to kernel memory.
GID_TTY	4	Terminal devices.
GID_OPERATOR	5	Operational tools.
GID_BIN	7	Daemon-owned files.
GID_VIDEO	44	Video framebuffer access.
GID_DIALER	68	Serial-port dial-out programs.
GID_NOGROUP	65533	No group.
GID_NOBODY	65534	Unprivileged placeholder.

8.24.7 Destruction Functions

Function	When to use
<code>destroy_dev(cdev)</code>	Ordinary, synchronous destruction with drain.
<code>destroy_dev_sched(cdev)</code>	Deferred destruction when you cannot sleep.
<code>destroy_dev_sched_cb(cdev, cb, arg)</code>	Deferred destruction with a follow-up callback.
<code>destroy_dev_drain(cdevsw)</code>	Wait for all cdevs of a cdevsw to finish, before freeing it.
<code>delist_dev(cdev)</code>	Remove a cdev from devfs without fully destroying yet.

8.24.8 Per-Open State Functions

Function	Purpose
<code>devfs_set_cdevpriv(priv, dtor)</code>	Attach private data to the current descriptor.
<code>devfs_get_cdevpriv(&priv)</code>	Retrieve the private data of the current descriptor.
<code>devfs_clear_cdevpriv()</code>	Detach and run the destructor early.
<code>devfs_foreach_cdevpriv(dev, cb, arg)</code>	Iterate all per-open records on a cdev.

8.24.9 Alias Functions

Function	Purpose
<code>make_dev_alias(pdev, fmt, ...)</code>	Create an alias for a primary cdev.
<code>make_dev_alias_p(flags, &cdev, pdev, fmt, ...)</code>	Create an alias with flags and error return.
<code>make_dev_physpath_alias(...)</code>	Create a topology-path alias.

8.24.10 Reference-Count Helpers

Usually not called directly by drivers. Listed here for recognition.

Function	Purpose
<code>dev_ref(cdev)</code>	Acquire a long-lived reference.
<code>dev_rel(cdev)</code>	Release a long-lived reference.
<code>dev_refthread(cdev, &ref)</code>	Acquire a reference for a handler call.
<code>dev_relthread(cdev, ref)</code>	Release the handler call's reference.

8.24.11 Where to Read More

- `make_dev(9)`, `destroy_dev(9)`, `cdev(9)` manual pages for the API surface.
- `devfs(5)`, `devfs.conf(5)`, `devfs.rules(5)`, `devfs(8)` for the filesystem-layer documentation.
- `/usr/src/sys/sys/conf.h` for the canonical struct and flag definitions.
- `/usr/src/sys/kern/kern_conf.c` for the implementation of the `make_dev*` family.
- `/usr/src/sys/fs/devfs/devfs_vnops.c` for the implementation of `devfs_set_cdevpriv` and friends.
- `/usr/src/sys/fs/devfs/devfs_rule.c` for the rules subsystem.

This reference is kept short on purpose. The chapter is where the reasoning lives; this section is just the lookup table.

8.24.12 A Condensed Pattern Catalogue

The table below summarises the main patterns the chapter has shown, each paired with the section that explains it in detail. When you are partway through building a driver and need a quick orientation, scan this list first.

Pattern	Section in the chapter
Create one data node in <code>attach</code> , destroy in <code>detach</code>	Chapter 7, referenced in Chapter 8 Lab 8.1
Move the node into a subdirectory under <code>/dev</code>	Naming, Unit Numbers, and Subdirectories

Pattern	Section in the chapter
Expose both a data node and a control node	Multiple Nodes Per Device; Lab 8.5
Add an alias so the driver answers on two paths	Aliases: One cdev, More Than One Name; Lab 8.2
Widen or narrow permissions at the operator level	Persistent Policy; Lab 8.4
Hide or expose a node inside a jail	Persistent Policy; Lab 8.7
Give each open its own state and counters	Per-Open State with <code>devfs_set_cdevpriv</code> ; Lab 8.3
Run pre-open allocation that is safe against crashes	Per-Open State; Challenge 2
Enforce exclusive open with <code>EBUSY</code>	Error Codes; Recipe 1
Tear down many cdevs in one detach	Destroying cdevs Safely; Lab 8.8
React to node creation in userland via <code>devd</code>	Exercising Your Device From Userland; Challenge 5
Compare two paths to verify they share a cdev	Exercising Your Device From Userland
Expose driver state through <code>sysctl</code>	Practical Workflows; Chapter 7 reference

Each row names a pattern. Each pattern has a short recipe somewhere in the chapter. When you face a design problem, find the row that fits and follow the link back.

8.24.13 Common errno Values by Operation

A compact cross-reference of which `errno` values are conventional for which operations. Pair with Section 13.

Operation	Common <code>errno</code> returns
<code>d_open</code>	<code>0</code> , <code>ENXIO</code> , <code>EBUSY</code> , <code>EACCES</code> , <code>EPERM</code> , <code>EINVAL</code> , <code>EAGAIN</code>
<code>d_close</code>	<code>0</code> almost always; log unusual conditions, do not return them
<code>d_read</code>	<code>0</code> on success, <code>ENXIO</code> if device gone, <code>EFAULT</code> for bad buffers, <code>EINTR</code> on signal, <code>EAGAIN</code> for non-blocking retry
<code>d_write</code>	Same family as <code>d_read</code> , plus <code>ENOSPC</code> for out-of-space
<code>d_ioctl</code> (Chapter 25)	<code>0</code> on success, <code>ENOTTY</code> for unknown commands, <code>EINVAL</code> for bad arguments
<code>d_poll</code> (Chapter 10)	Returns a revents mask, not an <code>errno</code>

Your Chapter 8 driver is concerned mostly with the first two rows. Chapter 9 will extend into the third and fourth.

8.24.14 A Short Glossary of Terms Used in the Chapter

For readers who have not seen every term before, or who want a quick reminder.

- **cdev**: the kernel-side identity of a device file, one per node.

- **cdevsw**: the dispatch table that maps operations to driver handlers.
- **cdevpriv**: per-open state attached to a file descriptor via `devfs_set_cdevpriv(9)`.
- **devfs**: the virtual filesystem that presents cdevs as nodes under `/dev`.
- **mda_***: members of the `make_dev_args` structure passed to `make_dev_s(9)`.
- **softc**: per-device private data allocated by Newbus and reachable through `device_get_softc(9)`.
- **SI_***: flags stored on a `struct cdev` in its `si_flags` field.
- **D_***: flags stored on a `struct cdevsw` in its `d_flags` field.
- **MAKEDEV_***: flags passed to `make_dev_s(9)` and its relatives through `mda_flags`.
- **UID_*** and **GID_***: symbolic constants for standard user and group identities.
- **destroy_dev_drain**: the cdevsw-level drain function used when unloading a module that has created many cdevs.
- **devfs.conf**: the host-level policy file for persistent node ownership and mode.
- **devfs.rules**: the ruleset file that shapes per-mount views of devfs, primarily for jails.

The glossary will grow as the book progresses. Chapter 8 introduced most of the terms it will need; later chapters will add their own and reference back to this list.

8.25 Consolidation and Review

Before you put the chapter down, one more pass through the material is worthwhile. This section ties the pieces together in a way that the section-by-section structure could not quite do.

8.25.1 The Three Ideas That Matter Most

If you could only remember three things from Chapter 8, let them be these:

First, `/dev` is a live filesystem maintained by the kernel. Every node is backed by a `struct cdev` your driver owns. Nothing you see in `/dev` is persistent; it is a window into the current state of the kernel. When you write a driver, you are adding to and removing from that window, and the kernel is honest about reflecting your changes.

Second, the device-file surface is part of your driver's public interface. The name, the permissions, the ownership, the existence of aliases, the set of operations you implement, the `errno` values you return, the order of destruction, all of these are decisions a user depends on. Treat them as contractual from day one. Widening or tightening after the fact is always more disruptive than picking the right baseline the first time.

Third, per-open state is the right home for per-descriptor information. `devfs_set_cdevpriv(9)` exists because UNIX's descriptor model is more expressive than a single `softc` can represent. When two processes open the same node, they each deserve their own view of it. Giving them per-open state costs a small allocation and a destructor; the alternative is a maze of shared-state races that you will not want to debug.

Everything else in Chapter 8 elaborates one of these three ideas.

8.25.2 The Shape of the Driver You End the Chapter With

By the end of Lab 8.8, your `myfirst` driver has grown into something that looks much more like a real FreeBSD driver than it did at the end of Chapter 7. Specifically:

- It has a softc, a mutex, and a sysctl tree.
- It creates its node in a subdirectory under /dev, with ownership and mode chosen on purpose.
- It offers an alias for the legacy name so existing users keep working.
- It allocates per-open state on every open(2) and cleans it up through a destructor that fires reliably in all cases.
- It counts active opens and refuses to detach while any are still alive.
- It destroys its cdevs in a sensible order during detach.

That is the shape of almost every small driver in /usr/src/sys/dev. You do not need to build every driver you ever write from scratch; most of the time, you will start from a template that looks exactly like this, and add the subsystem-specific logic on top.

8.25.3 What to Practice Before Starting Chapter 9

A short list of exercises that cement the chapter's material, in rough order of increasing stretch:

1. **Rebuild myfirst stage by stage without looking at the companion tree.** Open Chapter 7's stage 2 source. Make the changes for Lab 8.1 from scratch. Then the changes for Lab 8.2. Then Lab 8.3. Compare your result to the companion tree's stage 2 source. Differences are things worth understanding.
2. **Break a stage on purpose.** Introduce a deliberate bug into Lab 8.3 (for instance, skip the `devfs_set_cdevpriv` call). Predict what will happen when you load and run the parallel probe. Run it. See whether the failure matches your prediction.
3. **Add a third cdev.** Extend Lab 8.5's stage-3 driver with a second control node serving a different namespace. Watch the node appear and disappear in lockstep with the driver.
4. **Write a userland service.** Write a small daemon that opens /dev/myfirst/0 at startup, holds the descriptor, and responds to SIGUSR1 by reading and logging. Install it. Test it across driver load and unload. Notice what happens when the driver unloads while the daemon still has its descriptor open.
5. **Read a new driver.** Pick a driver from /usr/src/sys/dev that you have not touched yet, read it through the device-file lens, and classify it using the decision tree from Section 15. Write a paragraph describing what you found.

Each exercise takes between thirty minutes and an hour. Doing two or three of them is enough to move the chapter's material from "I read this once" to "I am comfortable with it". Doing all five gives you intuition that will serve you for the rest of the book.

Chapter 9 is next. The rooms behind the door come alive.

9 Reading and Writing to Devices

9.1 Reader Guidance & Outcomes

Chapter 7 taught you to stand a driver up. Chapter 8 taught you how that driver meets userland through `/dev`. The driver you finished the previous chapter with attaches as a Newbus device, creates `/dev/myfirst/0`, carries an alias at `/dev/myfirst`, allocates per-open state, logs cleanly, and detaches without leaking. Every one of those pieces was important, but none of them actually moved a byte.

This chapter is where the bytes start to move.

When a user program calls `read(2)` or `write(2)` on one of your device nodes, the kernel has to deliver real data between the user's address space and the driver's memory. That transfer is not a simple `memcpy`. It crosses a trust boundary. The buffer pointer the user passed might be invalid. The buffer might not all be resident. The length might be zero, or enormous, or part of a scatter-gather list. The user might be in a jail, might have a signal pending, might be reading with `O_NONBLOCK`, might have redirected the result through a pipe. Your driver does not need to understand every one of those cases in isolation, but it does need to cooperate with the single kernel abstraction that solves them all. That abstraction is `struct uio`, and the primary tool for using it is `uiomove(9)`.

This chapter is where we finally implement the `d_read` and `d_write` entry points that Chapter 7 left as stubs. Along the way, we will look carefully at how the kernel describes an I/O request, why the book has been saying “do not touch user pointers directly” ever since Chapter 5, and how to shape a driver so that partial transfers, misaligned buffers, signalled reads, and short writes all behave the way a classical UNIX file would.

9.1.1 Why This Chapter Earns Its Own Place

It would be tempting to write a short chapter that just says “call `uiomove`” and moves on. That would leave a reader with a driver that passes the simplest test and then fails in twenty subtle ways. The reason this chapter has the length it does is that I/O is where beginner drivers most often go wrong, and the places they go wrong are not where the code looks risky. The mistakes are usually in the return values, in the handling of `uio_resid`, in the treatment of a zero-length transfer, in what happens when the driver wakes up from `msleep(9)` because the process was killed, in the direction in which a partial read should drain.

A driver that gets these details wrong compiles cleanly, passes a single `cat /dev/myfirst`, and then produces corrupted data when a real program starts pushing bytes through it. That is the kind of bug that eats days. The goal of this chapter is to stop that class of bug at the source.

9.1.2 Where Chapter 8 Left the Driver

At the end of Chapter 8 your `myfirst` driver had the following shape. It is worth a checkpoint because Chapter 9 builds directly on top of it:

- A single Newbus child, created in `device_identify`, registered under `nexus0`.
- A struct `myfirst_softc` allocated by Newbus and initialised in `attach`.
- A mutex named after the device, used to guard `softc` counters.
- A sysctl tree under `dev.myfirst.0.stats` exposing `attach_ticks`, `open_count`, `active_fhs`, and `bytes_read`.
- A primary cdev at `/dev/myfirst/0` with ownership `root:operator` and mode `0660`.
- An alias cdev at `/dev/myfirst` pointing at the primary.
- A struct `myfirst_fh` allocated per `open(2)`, registered through `devfs_set_cdevpriv(9)`, and freed by a destructor that fires exactly once per descriptor.
- Stub `d_read` and `d_write` handlers that retrieve the per-open state, optionally look at it, and return immediately: `d_read` returns zero bytes (EOF), `d_write` claims to have consumed every byte by setting `uio_resid = 0`.

Chapter 9 takes those stubs and makes them real. The shape of the driver does not change much at the outside. A new reader should still see `/dev/myfirst/0`, still see the alias, still see the sysctls. What changes is that a `cat /dev/myfirst/0` will now produce output, a `echo hello > /dev/myfirst/0` will now store the text into driver memory, and a second `cat` will read back exactly what the first write deposited. By the end of the chapter your driver will be a small, disciplined, in-memory buffer that you can push bytes through and pull bytes out of. It will not yet be a circular buffer with blocking reads; that is Chapter 10's job. It will be a driver that moves bytes correctly.

9.1.3 What You Will Learn

After finishing this chapter you will be able to:

- Explain how `read(2)` and `write(2)` flow from user space through `devfs` into your `cdevsw` handlers.
- Read and write the fields of `struct uio` without memorising them.
- Use `uiomove(9)` to transfer bytes between a kernel buffer and the caller's buffer in either direction.
- Use `uiomove_frombuf(9)` when the kernel buffer has a fixed size and you want automatic offset accounting.
- Decide when to reach for `copyin(9)` or `copyout(9)` instead of `uiomove(9)`.
- Return correct byte counts for short transfers, empty transfers, end-of-file, and interrupted reads.
- Choose an appropriate `errno` value for every error path a driver read or write can take.
- Design an internal buffer that the driver fills from `d_write` and drains from `d_read`.
- Identify and fix the most common `d_read` and `d_write` bugs.
- Exercise the driver from the base-system tools (`cat`, `echo`, `dd`, `od`, `hexdump`) and from a small C program.

9.1.4 What You Will Build

You will take the `myfirst` driver from the end of Chapter 8 through three incremental stages.

1. **Stage 1, a static-message reader.** `d_read` returns the contents of a fixed kernel-space string. Every open starts at offset zero and reads its way through the message. This is the “hello world” of device reads, but done with correct offset handling.

2. **Stage 2, a write-once / read-many buffer.** The driver owns a fixed-size kernel buffer. `d_write` appends into it. `d_read` returns whatever has been written so far, from a per-descriptor offset that remembers how far each reader has consumed. Two concurrent readers still see their own progress independently.
3. **Stage 3, a small echo driver.** The same buffer, now used as a first-in / first-out store. Every `write(2)` appends bytes to the tail. Every `read(2)` removes bytes from the head. A two-process test script writes in one terminal and reads the echoed data in another. This is the handoff point to Chapter 10, where we will rebuild the same driver around a true circular buffer, add partial-I/O and non-blocking support, and wire up `poll(2)` and `kqueue(9)`.

All three stages compile, load, and behave predictably. You will exercise each one from `cat`, `echo`, and a small userland program called `rw_myfirst.c` that exercises the edge cases `cat` does not reach on its own.

9.1.5 What This Chapter Does Not Cover

Several topics touch `read` and `write` but are deliberately postponed:

- **Circular buffers and wrap-around:** Chapter 10 implements a real ring buffer. Stage 3 here uses a simple linear buffer so we can keep focus on the I/O path itself.
- **Blocking reads and `poll(2)`:** Chapter 10 introduces `msleep(9)`-based blocking and the `d_poll` handler. This chapter keeps all reads non-blocking at the driver level; an empty buffer produces an immediate zero-byte read.
- **`ioctl(2)`:** Chapter 25 builds out `d_ioctl`. We touch it only where the reader needs to understand why certain control paths belong there rather than in `write`.
- **Hardware registers and DMA:** Part 4 handles bus resources, `bus_space(9)`, and DMA. The memory we read and write in this chapter is ordinary kernel heap, allocated with `malloc(9)` from `M_DEVBUF`.
- **Concurrency correctness under load:** Part 3 is dedicated to race conditions, locking, and verification. We take mutex-protected care where a race would corrupt the stage-3 buffer, but the deeper discussion is deferred.

Staying inside these lines is how we keep the chapter honest. A beginner chapter that drifts into `ioctl`, DMA, and `kqueue` is a beginner chapter that teaches none of them well.

9.1.6 Estimated Time Investment

- **Reading only:** roughly one hour.
- **Reading plus typing the three stages:** around three hours, including a couple of load / unload cycles per stage.
- **Reading plus all labs and challenges:** five to seven hours over two or three sessions.

Give yourself a fresh lab boot at the start. Do not rush. The stages are small on purpose, and the real value comes from watching `dmesg`, watching `sysctl`, and probing the device from userland after every change.

9.1.7 Prerequisites

Before starting this chapter, confirm:

- You have a working `myfirst` driver equivalent to the Chapter 8 stage 2 source under `examples/part-02/ch08-working-with-device-files/stage2-perhandle/`. If you have not yet reached the end of Chapter 8, pause here and come back.
- Your lab machine is running FreeBSD 14.3 with a matching `/usr/src`.
- You have been through Chapter 4's discussion of pointers, structures, and memory layout, and Chapter 5's discussion of kernel-space idioms and safety.
- You understand what a `struct cdev` is and how it is related to a `cdevsw`. Chapter 8 covered this in detail.

If you are uncertain about any of these, the rest of the chapter will be harder than it needs to be. Revisit the relevant sections first.

9.1.8 How to Get the Most Out of This Chapter

Three habits pay off immediately.

First, keep `/usr/src/sys/dev/null/null.c` open in a second terminal. It is the shortest, cleanest, most readable example of `d_read` and `d_write` in the tree. Every idea this chapter introduces appears somewhere in `null.c` in fifty lines or fewer. Real FreeBSD drivers are the textbook; this book is the reading guide.

Second, keep `/usr/src/sys/sys/uio.h` and `/usr/src/sys/_uio.h` open. The declarations there are short and stable. Read them once now so that when the chapter refers to `uio_iov`, `uio_iovcnt`, `uio_offset`, and `uio_resid`, you do not have to trust the prose alone.

Third, rebuild between changes and confirm behaviour from userland before the next change. This is the habit that separates writing drivers from writing prose about drivers. You will run `cat`, `echo`, `dd`, `stat`, `sysctl`, `dmesg`, and a short C program at every checkpoint. Do not skip them. The failure modes this chapter is teaching you to recognise only become visible when you run the code.

9.1.9 Roadmap Through the Chapter

The sections in order are:

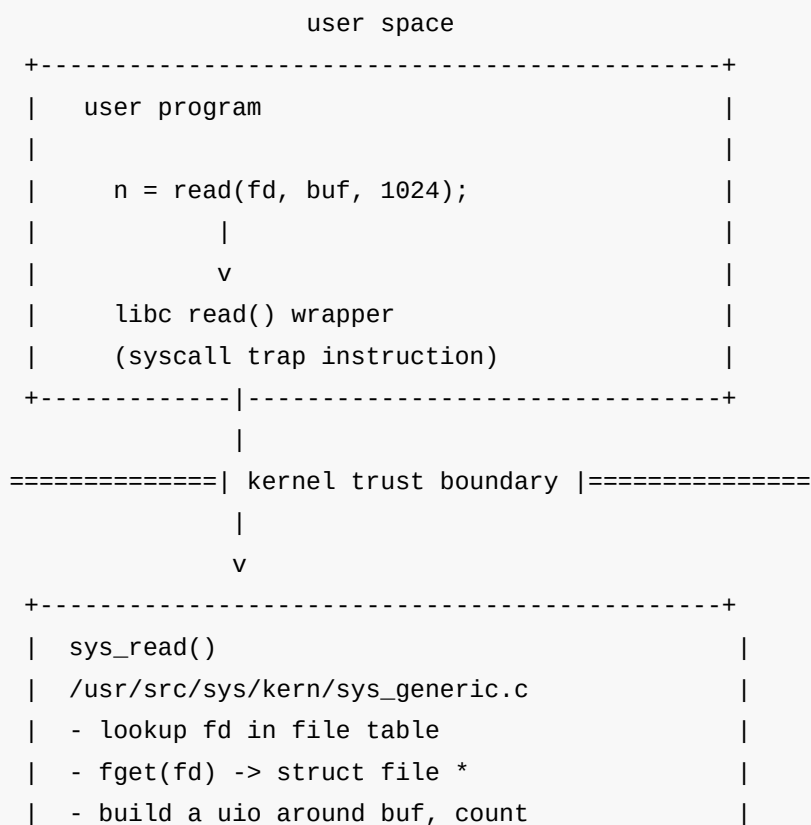
1. A visual map of the full I/O path, from `read(2)` in user space to `uiomove(9)` inside your handler.
2. A short refresher on what `read` and `write` mean in UNIX, and what they mean specifically to a driver writer.
3. The anatomy of `d_read`: its signature, what it is asked to do, and what it is asked to return.
4. The anatomy of `d_write`: the mirror of `d_read`, plus a few details that only apply in the write direction.
5. A reading protocol for unfamiliar handlers, followed by a second real-driver walkthrough (`mem(4)`) to show a different shape.
6. The `ioflag` argument: where it comes from, what bits matter, and why Chapter 9 mostly ignores it.
7. A close look at `struct uio`, the kernel's I/O description object, field by field, including three snapshots of the same `uio` through a call.
8. `uiomove(9)` and its companions, the functions that actually move the bytes.
9. `copyin(9)` and `copyout(9)`: when to reach for them, and when to leave them alone in favour of `uiomove`. Plus a cautionary case study about structured data.
10. Internal buffers: static, dynamic, and fixed-size. How to choose one, how to own it safely, and the kernel helpers you should recognise.

11. Error handling: the `errno` values that matter for I/O, how to signal end of file, and how to think about partial transfers.
12. The three-stage `myfirst` implementation, driver source included.
13. A step-by-step trace of `read(2)` from user space through the kernel down to your handler, plus a mirrored write trace.
14. A practical workflow for testing: `cat`, `echo`, `dd`, `truss`, `ktrace`, and the discipline that turns them into a development rhythm.
15. Observability: `sysctl`, `dmesg`, and `vmstat -m`, with a concrete snapshot of the driver under light load.
16. Signed, unsigned, and the perils of off-by-one, a short but high-value section.
17. Troubleshooting notes for the mistakes this chapter's material is most likely to produce, and a contrast table of correct versus buggy handler patterns.
18. Hands-on labs (seven) that walk you through each stage and consolidate the observability workflow.
19. Challenge exercises (eight) that stretch the patterns.
20. Wrapping up and a bridge to Chapter 10.

If this is your first time through the chapter, read linearly and do the labs as you hit them. If you are revisiting the material to consolidate, the reference-style sections at the end stand alone.

9.2 A Visual Map of the I/O Path

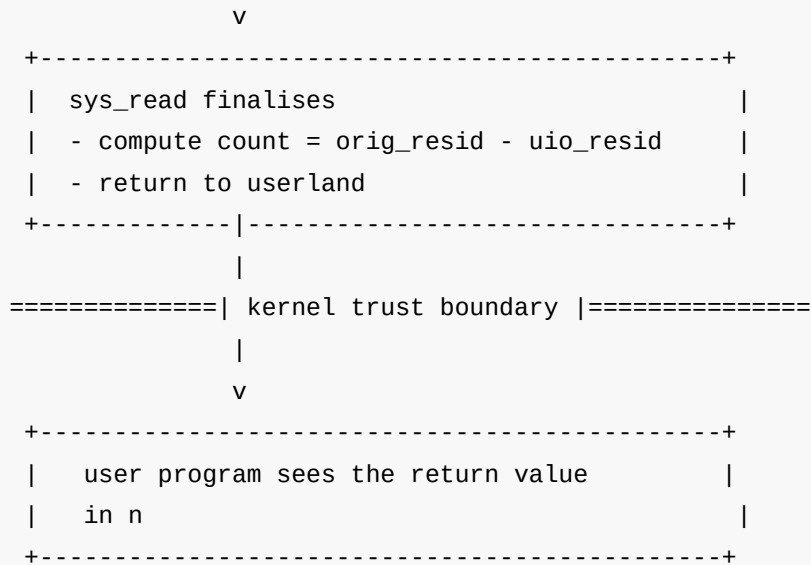
Before the prose gets deep, a single picture is worth keeping in mind. The diagram below is the path a `read(2)` call takes from a user program down to your driver, and back up to the caller. Every box is a real piece of kernel code you can find under `/usr/src/sys/`. Every arrow is a real function call. None of it is metaphorical.



```

+-----+|-----+
|
|      v
+-----+|-----+
| struct file ops -> vn_read |
| /usr/src/sys/kern/vfs_vnops.c |
+-----+|-----+
|
|      v
+-----+|-----+
| devfs_read_f() |
| /usr/src/sys/fs/devfs/devfs_vnops.c |
| - devfs_fp_check -> cdev + cdevsw |
| - acquire thread-count ref |
| - compose ioflag from f_flag |
| - call cdevsw->d_read(dev, uio, ioflag) |
+-----+|-----+
|
|      v
+-----+|-----+
| YOUR HANDLER (myfirst_read) |
| - devfs_get_cdevpriv(&fh) |
| - verify is_attached |
| - call uiomove(9) to transfer bytes |
| | | |
| |      v |
| | +-----+ |
| | | uiomove_faultflag() | |
| | | /usr/src/sys/kern/subr_uio.c | |
| | | - for each iovec entry | |
| | | copyout(kaddr, uaddr, n) ===> |====|====> user's buf
| | | decrement uio_resid | |
| | | advance uio_offset | |
| | +-----+ |
| | - return 0 or an errno |
+-----+|-----+
|
|      v
+-----+|-----+
| devfs_read_f continues |
| - release thread-count ref |
| - update atime if bytes moved |
+-----+|-----+
|

```



A handful of features of the picture are worth pinning down because they recur throughout the chapter.

The trust boundary is crossed exactly twice. Once on the way down (the user enters the kernel via a syscall trap), and once on the way up (the kernel returns control to user space). Everything in between is kernel-only execution. Your handler runs entirely inside the kernel, on a kernel stack, with the user’s registers saved out of the way.

Your handler is the only place the driver’s knowledge enters the path. Everything above it is kernel machinery that works identically for every character device in the tree. Everything below it is uiomove and copyout, also kernel machinery. Your handler is the single function where the answer to “what bytes should this read produce?” is computed.

The user’s buffer is never touched by your driver directly. It is touched by copyout from inside uiomove. Your driver hands uiomove a kernel pointer, and uiomove is the only code that dereferences the user pointer on your behalf. This is the shape of the trust boundary, drawn as code: user memory is accessed only through the one API that knows how to do it safely.

Every step has a matching step on the way back up. The thread-count reference acquired by devfs is released after your handler returns; the uio’s state is inspected to compute the byte count; control unwinds through each layer and returns to user space. Understanding this symmetry is what makes reference-counting feel natural rather than arbitrary.

Print this diagram or sketch it on paper. When you read an unfamiliar driver later in the book, refer back to it. Every `d_read` or `d_write` you will ever study sits at exactly this point in the call chain. The differences between drivers are in the handler; the path around the handler is constant.

For `d_write`, the picture is the mirror image. `devfs_write_f` dispatches to `cdevsw->d_write`, your handler calls `uiomove(9)` in the other direction, `uiomove` calls `copyin` instead of `copyout`, and the kernel unwinds back to `write(2)`. Every arrow in the diagram has a twin; every property listed above applies to writes as well.

9.3 Devices in UNIX: A Quick Refresher

It is worth ten minutes of refresher before we start writing code. Chapter 6 introduced the UNIX I/O model at a conceptual level; Chapter 7 put it into practice; Chapter 8 made the device-file surface tidy. All three of those treatments had reasons not to dwell on the behaviour of `read(2)` and `write(2)` themselves, because the drivers in those chapters did not carry real data. Now we do, and a tight refresher sets the stage for everything that follows.

9.3.1 What Makes a Device Different from a File?

From the outside, they look identical. Both are opened with `open(2)`. Both are read with `read(2)` and written with `write(2)`. Both are closed with `close(2)`. A user program that works on a regular file almost always works on a device file without changes to its source, because the user-space API makes no distinction between them.

From the inside, there are real differences, and a driver author needs to internalise them.

A regular file has a backing store, usually bytes on disk managed by a filesystem. The kernel decides when to read ahead, when to cache, when to flush. The data has a persistent identity; two programs reading byte zero of a file see the same byte. Seeking is cheap and unlimited within the file's size.

A device file has no backing store in the filesystem sense. When a user program reads from it, the driver decides what bytes to produce. When a user program writes to it, the driver decides what to do with them. The data's identity is whatever your driver defines. Two programs reading from the same device do not necessarily see the same bytes; depending on the driver, they may see the same bytes, may see disjoint halves of a single stream, or may see completely independent streams. Seeking may be meaningful, or meaningless, or actively forbidden.

The practical consequence for your `d_read` and `d_write` handlers is that **the driver is the authoritative definition of what read and write mean** on this device. The kernel will deliver you an I/O request; it will not tell you what to do with it. The conventions UNIX programs expect, a byte stream, consistent return values, honest error codes, end of file as zero bytes returned, are conventions your driver has to honour on purpose. The kernel does not enforce them.

9.3.2 How UNIX Treats Devices as Streams of Data

The word “stream” is worth pinning down, because it appears in every discussion of UNIX I/O and carries at least three different meanings depending on context.

For our purposes a stream is a **sequence of bytes delivered in order**. Neither the caller nor the driver knows the total length in advance. Either side can stop at any time. The sequence may have a natural end (a file that has been fully read) or may go on indefinitely (a terminal, a network socket, a sensor). The rules are the same either way: the reader requests some number of bytes, the writer requests that some number of bytes be accepted, and the kernel reports how many bytes actually moved.

A stream has no side-effects beyond the data transfer itself. If your driver needs to expose a control surface, a way to change configuration, reset state, or negotiate parameters, that surface does not belong in read and write. The interface for control is `ioctl(2)`, covered in Chapter 25. Do not smuggle control commands through the data stream. It makes your driver harder to use, harder to test, and harder to evolve.

A stream is unidirectional per call. `read(2)` moves bytes from the driver to the user. `write(2)` moves bytes from the user to the driver. A single system call never does both. If you need duplex behaviour, for example a request-response pattern, you implement it as a write followed by a read, with whatever coordination your driver requires inside it.

9.3.3 Sequential vs. Random Access

Most drivers produce sequential streams: the bytes come out in the order they arrive, and `lseek(2)` either does nothing interesting or is refused. A terminal, a serial port, a packet-capture device, a log stream, all of these are sequential.

A few drivers are random-access: the caller can address any byte through `lseek(2)`, and the same offset always reads the same data. A memory-disk driver, `/dev/mem`, and a handful of others fit this model. They look more like regular files than like devices in most respects.

A driver author chooses where on this spectrum the driver sits. Your `myfirst` driver will sit on the sequential end for most of this chapter, with one nuance: each open descriptor carries its own read offset, so two processes reading concurrently start from different points in the stream. This is the compromise most small character devices use. It gives each reader a consistent view of what they have consumed, without imposing a true random-access contract on the driver.

The choice shows up in two places in code:

- **Your `d_read` updates `uio->uio_offset`** (which `uiomove(9)` does for you) if and only if the offset is meaningful to you. For a truly sequential device where offset has no meaning, the value is ignored.
- **Your driver either honours or ignores the incoming `uio->uio_offset`** at the start of each read. Sequential drivers ignore it and serve from wherever they are. Random-access drivers treat it as an address into a linear space.

For the three-stage `myfirst` we will treat `uio->uio_offset` as a per-call snapshot of where this descriptor is in the stream and update our internal counters to match.

9.3.4 The Role of `read()` and `write()` in Device Drivers

Inside the kernel, `read(2)` and `write(2)` on a device file eventually call into your `cdevsw->d_read` and `cdevsw->d_write` function pointers. Everything between the system call and your function is `devfs` and `VFS` machinery; everything after your function returns is the kernel delivering the result back to userland. Your handler is the single place where the driver-specific answer to “what happens on this call?” is computed.

The handler’s job is not complicated in the abstract:

1. Look at the request. How many bytes are being asked for or handed to you?
2. Move the bytes. Use `uiomove(9)` to transfer data between your kernel buffer and the user’s.
3. Return a result. A zero for success (with `uio_resid` updated accordingly), or an `errno` value for failure.

What makes the handler non-trivial is that step 2 is the trust boundary between user memory and kernel memory, and every interaction with user memory has to be safe against misbehaving or malicious user programs. That is why `uiomove(9)` exists. You do not write the safety logic; the kernel does, as long as you ask through the right API.

9.3.5 Character Devices vs. Block Devices Revisited

Chapter 8 noted that FreeBSD has not shipped block-special device nodes to userland for many years. Storage drivers live in GEOM and publish themselves up as character devices. For the purposes of this chapter, character device is the only shape we care about.

The practical consequence is that everything in this chapter applies to every driver you are likely to write in Parts 2 through 4. `d_read` and `d_write` are the entry points. `struct uio` is the carrier. `uiomove(9)` is the mover. When we get to Part 6 and look at GEOM-backed storage drivers, their data path will look different, but it will still be built out of the same primitives we are studying now.

9.3.6 Exercise: Classifying Real Devices on Your FreeBSD System

Before the rest of the chapter digs into code, take five minutes on your lab machine. Open a terminal and walk through `/dev`:

```
% ls /dev
% ls -l /dev/null /dev/zero /dev/random /dev/urandom /dev/console
```

For each node you see, ask yourself three questions:

1. Is it sequential or random access?
2. If I cat it, should it produce any bytes? What bytes?
3. If I echo something > it, should anything be visible? Where?

Try a few:

```
% head -c 16 /dev/zero | od -An -tx1
% head -c 16 /dev/random | od -An -tx1
% echo "hello" > /dev/null
% echo $?
```

Notice that `/dev/zero` is inexhaustible, `/dev/random` delivers unpredictable bytes, `/dev/null` swallows writes silently and returns success, and none of these three are seekable in a useful sense. Those behaviours are not accidents. They are the `d_read` and `d_write` handlers of those drivers, doing exactly what we are about to study.

If you open `/usr/src/sys/dev/null/null.c` and look at `null_write`, you will see the one-line implementation: `uio->uio_resid = 0; return 0;`. That is a fully functional write handler. The driver has announced “I consumed every byte; no error”. That is the smallest meaningful write implementation in FreeBSD, and by the end of this chapter you will be able to write it, and many larger ones, without hesitation.

9.4 The Anatomy of `d_read()`

Your driver’s read path begins the moment `devfs` dispatches a call to `cdevsw->d_read`. The signature is fixed, declared in `/usr/src/sys/sys/conf.h`:

```
typedef int d_read_t(struct cdev *dev, struct uio *uio, int ioflag);
```

Every `d_read` function in the FreeBSD tree has exactly this shape. The three arguments are the complete description of the call:

- `dev` is the `struct cdev *` representing the device node that was opened. In a driver that handles more than one `cdev` per instance, it tells you which one the call is on. In `myfirst`, where the primary and its alias dispatch through the same handler, both resolve to the same underlying softc via `dev->si_drv1`.
- `uio` is the `struct uio *` that describes the I/O request: what buffers the user provided, how big they are, where in the stream the read is supposed to start, and how many bytes still need to move. We will dissect it in the next section.
- `ioflag` is a bitmask of flags defined in `/usr/src/sys/sys/vnode.h`. The one that matters for non-blocking I/O is `IO_NDELAY`, which is set when the user opened the descriptor with `O_NONBLOCK` (or passed `O_NONBLOCK` later via `fcntl(F_SETFL, ...)`). There are a handful of other flags related to vnode-based filesystem I/O, but for character-device drivers you will usually inspect only `IO_NDELAY`.

The return value is an `errno`-style integer: zero on success, a positive `errno` code on failure. It is **not** a byte count. The kernel computes the byte count by looking at how much `uio_resid` decreased during the call and reports that value up to user space as the return value of `read(2)`. This inversion is one of the two or three most important things to internalise from the chapter. `d_read` returns an error code; the number of bytes transferred is implicit in the `uio`.

9.4.1 What `d_read` Is Asked to Do

Reduced to a single sentence, the job is: **produce up to `uio->uio_resid` bytes from the device, deliver them through `uiomove(9)` into whatever buffer the `uio` describes, and return zero.**

A few corollaries follow from that sentence and are worth making explicit.

The function may produce fewer bytes than were asked for. Short reads are legitimate and expected. A user program that asked for 4096 bytes and received 17 does not treat that as an error; it treats it as “the driver had 17 bytes to give right now”. The number is visible to the caller because `uiomove(9)` decremented `uio_resid` by 17 as it moved the bytes.

The function may produce zero bytes. A zero-byte read is how UNIX reports end of file. If your driver has no more data to give and is not going to have any more, return zero and leave `uio_resid` untouched. The caller sees a zero-byte `read(2)` and knows the stream is done.

The function must not produce more bytes than were asked for. `uiomove(9)` enforces this for you; it will not move more than `MIN(uio_resid, n)` bytes in a single call. If you call `uiomove` repeatedly inside a single `d_read`, make sure your loop also respects `uio_resid`.

The function must return an `errno` on failure. On success the return value is zero. Non-zero return values are interpreted by the kernel as errors; the kernel propagates them to user space through `errno`. Common values are `ENXIO`, `EFAULT`, `EIO`, `EINTR`, and `EAGAIN`. We will walk through each in the error-handling section.

The function can sleep. `d_read` runs in a process context (the context of the caller), so `msleep(9)` and friends are legal. This is how drivers implement blocking reads that wait for data. We will not use `msleep(9)` in this chapter (Chapter 10 introduces it formally), but it is worth knowing that you have the right to block.

9.4.2 What `d_read` Is Not Asked to Do

A short list of things the handler is explicitly not responsible for, because the kernel or devfs takes care of them:

- **Locating user memory.** The `uio` already describes the target buffer. Your handler does not need to look up page tables or validate addresses.
- **Checking permissions.** The user's credentials were verified by `open(2)`; by the time `d_read` runs, the caller is allowed to read from this descriptor.
- **Counting bytes for the caller.** The kernel computes the byte count from `uio_resid`. You never return a byte count.
- **Enforcing the global size limit.** The kernel has already clamped `uio_resid` to something the system can handle.

Every one of these is a temptation at some point. Resist them all. Each one is a place where a handler can introduce a subtle bug that a correct use of `uiomove` avoids by construction.

9.4.3 A First Real `d_read`

Here is the smallest useful `d_read` in the FreeBSD tree. It is the `zero_read` function from `/usr/src/sys/dev/null/null.c`, and it is how `/dev/zero` produces an infinite stream of zero bytes:

```
static int
zero_read(struct cdev *dev __unused, struct uio *uio, int flags __unused)
{
    void *zbuf;
    ssize_t len;
    int error = 0;

    zbuf = __DECONST(void *, zero_region);
    while (uio->uio_resid > 0 && error == 0) {
        len = uio->uio_resid;
        if (len > ZERO_REGION_SIZE)
            len = ZERO_REGION_SIZE;
        error = uiomove(zbuf, len, uio);
    }
    return (error);
}
```

Pause on that for a minute. The loop body is three lines. The termination condition is two: either `uio_resid` reaches zero (we transferred everything the caller asked for) or `uiomove` returns an error. Each iteration moves as much of the zero-filled region as the request has room for. The function returns the last error code, which is zero if the transfer completed cleanly.

The loop is necessary because the zero region is finite: a single `uiomove` call cannot move arbitrarily many bytes from it, so the loop chunks the transfer. For a driver whose source data fits in a single kernel buffer of modest size, the loop collapses to a single call. Stage 1 of `myfirst` will be exactly that shape.

Notice also what the function does **not** do. It does not look at `uio_offset`. It does not care where in some imaginary stream the read is starting; every read of `/dev/zero` produces zero bytes. It does not check the `cdev`. It does not check the flags. It does exactly one job, and it does that job using one API.

That is the model. Your `d_read` will usually look like some variation of that loop.

9.4.4 A Variant: `uiomove_frombuf`

When your source data is a fixed-size kernel buffer and you want the driver to behave like a file backed by that buffer, the helper function `uiomove_frombuf(9)` does the offset arithmetic for you.

Its declaration, from `/usr/src/sys/sys/uio.h`:

```
int uiomove_frombuf(void *buf, int buflen, struct uio *uio);
```

Its implementation, from `/usr/src/sys/kern/subr_uio.c`, is short enough to reproduce:

```
int
uiomove_frombuf(void *buf, int buflen, struct uio *uio)
{
    size_t offset, n;

    if (uio->uio_offset < 0 || uio->uio_resid < 0 ||
        (offset = uio->uio_offset) != uio->uio_offset)
        return (EINVAL);
    if (buflen <= 0 || offset >= buflen)
        return (0);
    if ((n = buflen - offset) > IOSIZE_MAX)
        return (EINVAL);
    return (uiomove((char *)buf + offset, n, uio));
}
```

Read that carefully, because the behaviour is precise. The function takes a pointer `buf` to a kernel buffer of size `buflen`, consults the `uio->uio_offset`, and:

- If the offset is negative or otherwise nonsensical, returns `EINVAL`.
- If the offset is past the end of the buffer, returns zero without moving any bytes. This is end of file: the caller will see a zero-byte read.
- Otherwise, calls `uiomove(9)` with a pointer into `buf` at the current offset and a length equal to the remaining tail of the buffer.

The function does not loop; `uiomove` will move as many bytes as `uio_resid` has room for, and will decrement `uio_resid` accordingly. The driver never has to touch `uio_offset` after the fact, because `uiomove` does.

If your driver exposes a fixed buffer as a readable file, a one-line `d_read` suffices:

```
static int
myfirst_read(struct cdev *dev, struct uio *uio, int ioflag)
{
    struct myfirst_softc *sc = dev->si_drv1;
    return (uiomove_frombuf(sc->buf, sc->buflen, uio));
}
```

Stage 1 of this chapter uses exactly that pattern, with a small adjustment to track the per-descriptor read offset so two concurrent readers see their own progress.

9.4.5 The Signature in Use: myfirst_read Stage 1

Here is what our Stage 1 `d_read` will look like. Do not type it in yet; we will walk through the full source in the implementation section. Seeing it here and now is mostly to anchor the discussion.

Before you read the code, pause on one detail that will recur in almost every handler for the rest of this chapter. The first four lines of any per-open-aware handler follow a fixed **boilerplate pattern**:

```
struct myfirst_fh *fh;
int error;

error = devfs_get_cdevpriv((void **)&fh);
if (error != 0)
    return (error);
```

This pattern retrieves the per-descriptor `fh` that `d_open` registered through `devfs_set_cdevpriv(9)`, and it propagates any failure back to the kernel unchanged. You will see it at the top of `myfirst_read`, `myfirst_write`, `myfirst_ioctl`, `myfirst_poll`, and the `kqfilter` helpers. When a later lab says “retrieve the per-open state with the usual `devfs_get_cdevpriv` boilerplate”, this is the block it refers to, and the rest of the chapter will not re-explain it. If a handler ever re-orders these lines, treat that as a red flag: running any logic before this call means the handler does not yet know which open it is serving. The one subtlety worth remembering is that the `sc == NULL` liveness check comes *after* this boilerplate, not before, because you need the per-open state retrieved safely even on a device that is being torn down.

```
static int
myfirst_read(struct cdev *dev, struct uio *uio, int ioflag)
{
    struct myfirst_softc *sc = dev->si_drv1;
    struct myfirst_fh *fh;
    off_t before;
    int error;

    error = devfs_get_cdevpriv((void **)&fh);
    if (error != 0)
```

```
        return (error);

    if (sc == NULL || !sc->is_attached)
        return (ENXIO);

    before = uio->uio_offset;
    error = uiomove_frombuf(__DECONST(void *, sc->message),
        sc->message_len, uio);
    if (error == 0)
        fh->reads += (uio->uio_offset - before);
    fh->read_off = uio->uio_offset;
    return (error);
}
```

A few points worth noticing before we continue. The function retrieves the per-open structure through `devfs_get_cdevpriv(9)`, checks that the softc is live, and then hands the real work to `uiomove_frombuf`. We snapshot `uio->uio_offset` into a local `before` at entry so that after the call we can compute the number of bytes the kernel just moved as `uio->uio_offset - before`. That increment is recorded into the per-descriptor counter. The closing assignment to `fh->read_off` remembers the stream position so the rest of the driver can report it later.

If the driver has no data to deliver, `uiomove_frombuf` returns zero and `uio_resid` is unchanged, which is how end of file is reported. If an error occurs inside `uiomove`, we propagate it back up by returning the error code. Nothing about this handler needs copyin or copyout directly. The safety of the transfer is handled by `uiomove` on our behalf.

9.4.6 Reading `d_read` in the Tree

A good reading exercise, after finishing this section, is to `grep` for `d_read` across `/usr/src/sys/dev` and look at what other drivers do inside it. You will find three recurring shapes:

- **Drivers that read from a fixed buffer.** They use `uiomove_frombuf(9)` or a hand-rolled equivalent, one call, done. `/usr/src/sys/fs/pseudofs/pseudofs_vnops.c` uses the helper extensively; the pattern is identical for character devices.
- **Drivers that read from a dynamic buffer.** They take an internal lock, snapshot how much data is available, call `uiomove(9)` with that length, release the lock, and return. We will build one of these in Stage 2.
- **Drivers that read from a blocking source.** They check whether data is available, and if not, either sleep on a condition variable (blocking mode) or return `EAGAIN` (non-blocking mode). This is Chapter 10 territory.

All three shapes share the same four-line spine: retrieve per-open state if you use it, verify liveness, call `uiomove` (or a relative), return the error code. The differences are in how they prepare the buffer, not in how they transfer it.

9.5 The Anatomy of `d_write()`

The write handler is the mirror image of the read handler, with a few small differences at the edges. The signature, from `/usr/src/sys/sys/conf.h`:

```
typedef int d_write_t(struct cdev *dev, struct uio *uio, int ioflag);
```

The shape is identical. The three arguments carry the same meaning. The return value is an `errno`, zero on success. The byte count is still computed from `uio_resid`: the kernel looks at how much `uio_resid` decreased during the call and reports that as the return value of `write(2)`.

9.5.1 What `d_write` Is Asked to Do

One sentence again: **consume up to `uio->uio_resid` bytes from the user, deliver them through `uiomove(9)` into wherever your driver keeps its data, and return zero.**

The corollaries are almost exactly the same as for reads, with two notable differences:

- A short write is legitimate but unusual. A driver that accepts fewer bytes than were offered must update `uio_resid` to reflect the truth, and the kernel will report the partial count to user space. Most well-behaved user programs will loop and retry the remainder; many will not. The rule of thumb is: accept everything you can, and if you cannot accept more, return `EAGAIN` for non-blocking callers and (eventually) sleep for blocking callers.
- A zero-byte write is not end of file. It is simply a write that moved zero bytes. `d_write` does not have an EOF concept; only reads do. A driver that wants to refuse a write returns a non-zero `errno`.

The most common return value on the error side is `ENOSPC` (no space left on device) when the driver's buffer is full, `EFAULT` when a pointer-related failure occurs inside `uiomove`, and `EIO` for catch-all hardware errors. A driver that enforces a per-write length limit can return `EINVAL` or `EMSGSIZE` for writes that exceed the limit; we will look at which to pick later in the chapter.

9.5.2 What `d_write` Is Not Asked to Do

The same list as for `d_read`: it does not locate user memory, does not check permissions, does not count bytes for the caller, and does not enforce system-wide limits. The kernel takes care of all four.

One addition specific to writes: **do not assume the incoming data is null-terminated or otherwise structured.** Users may write arbitrary bytes. If your driver expects structured input, it must parse it defensively. If your driver expects binary data, it must handle writes that are not aligned with any natural boundary. `write(2)` is a byte stream, not a message queue. Chapter 25's `ioctl` path is where structured, framed commands belong.

9.5.3 A First Real `d_write`

The simplest non-trivial `d_write` in the tree is `null_write` from `/usr/src/sys/dev/null/null.c`:


```
static int
null_write(struct cdev *dev __unused, struct uio *uio, int flags __unused)
{
    uio->uio_resid = 0;

    return (0);
}
```

Two lines. The handler tells the kernel “I consumed all the bytes” by setting `uio_resid` to zero, and returns success. The kernel reports the original request length to user space as the number of bytes written. `/dev/null` does not actually do anything with the bytes; that is the whole point of `/dev/null`. But the pattern is instructive: **setting `uio_resid = 0` is the shortest way to mark a write as fully consumed**, and it is exactly what `uiomove(9)` would have done if we had given it a destination.

A marginally more interesting case is `full_write`, also in `null.c`:

```
static int
full_write(struct cdev *dev __unused, struct uio *uio __unused, int flags __unused)
{
    return (ENOSPC);
}
```

This is the backing for `/dev/full`, a device that is full forever. Every write fails with `ENOSPC`, and the caller sees the corresponding `errno` value. The handler does not touch `uio_resid`; the kernel sees that no bytes moved and reports a return value of -1 with `errno = ENOSPC`.

Together, these two handlers illustrate the two extremes of the write side: accept everything, or reject everything. Real drivers sit somewhere in between, deciding how many of the offered bytes they can accept and storing those bytes somewhere.

9.5.4 A Write That Actually Stores Data

Here is the shape of the write handler we will implement at the end of this chapter. Again, do not type it in yet; this is a preview for orientation.

```
static int
myfirst_write(struct cdev *dev, struct uio *uio, int ioflag)
{
    struct myfirst_softc *sc = dev->si_drv1;
    struct myfirst_fh *fh;
    size_t avail, towrite;
    int error;

    error = devfs_get_cdevpriv((void **)&fh);
```

```

    if (error != 0)
        return (error);

    if (sc == NULL || !sc->is_attached)
        return (ENXIO);

    mtx_lock(&sc->mtx);
    avail = sc->buflen - sc->bufused;
    if (avail == 0) {
        mtx_unlock(&sc->mtx);
        return (ENOSPC);
    }
    towrite = MIN((size_t)uio->uio_resid, avail);
    error = uiomove(sc->buf + sc->bufused, towrite, uio);
    if (error == 0) {
        sc->bufused += towrite;
        fh->writes += towrite;
    }
    mtx_unlock(&sc->mtx);
    return (error);
}

```

The handler locks the softc mutex, checks how much buffer space is left, clamps the transfer to whatever fits, calls `uiomove(9)` with that length, and accounts for the successful transfer by advancing `bufused`. If the buffer is full, it returns `ENOSPC` to signal the caller. Everything the handler does to cope with concurrency or partial writes is captured in the combination of the lock and the clamp.

Note that `uiomove` itself is called **with the mutex held**. This is fine as long as the mutex is an ordinary `MTX_DEF` mutex (as `myfirst`'s is) and the calling context is a regular kernel thread that can sleep. `uiomove` may page-fault when copying to or from user memory, and page faults may require the kernel to sleep waiting for a disk read. A sleep while holding an `MTX_DEF` mutex is legal; a sleep while holding a spin lock (`MTX_SPIN`) would be a bug. Part 3 covers the locking rules formally; for now, trust the lock type you picked in Chapter 7.

9.5.5 Symmetry With `d_read`

Reads and writes are almost identical from the driver's point of view. The data flows in opposite directions, and the `uio->uio_rw` field tells `uiomove` which direction to move the bytes. On the driver side you pass the same arguments: a pointer to kernel memory, a length, and the `uio`. On the user side, `uiomove` either copies out of the kernel buffer (for a read) or into it (for a write). You rarely have to think about the direction; `uio_rw` is already set.

What changes between the two handlers is the **intent**. A read is the driver's opportunity to produce data. A write is the driver's opportunity to consume data. Your code in each handler knows what role it is playing and does the appropriate bookkeeping: a reader tracks how much it has delivered, a writer tracks how much it has stored.

9.5.6 Reading `d_write` in the Tree

After reading this section, take a few minutes with `grep d_write /usr/src/sys/dev | head -20` and look at what other drivers do. Three shapes show up:

- **Drivers that discard writes.** Usually one line: `set uio_resid = 0` and return zero. The `null` driver's `null_write` is the prototype.
- **Drivers that store writes.** They lock, check capacity, call `uiomove(9)`, account, unlock, and return. Our Stage 3 handler is this shape.
- **Drivers that forward writes to hardware.** They pull data out of the `uio`, stage it into a DMA buffer or a hardware-owned ring, and trigger the hardware. This shape is out of scope until Part 4; the mechanics of `uiomove` are the same, but the destination is a DMA-mapped region rather than a `malloc'd` buffer.

Every real driver fits one of these three. The ones that accumulate or reshape data first, and then forward it somewhere, tend to combine shape 2 and shape 3, but the primitives are identical.

9.5.7 Reading an Unfamiliar `d_read` or `d_write` in the Wild

A chapter like this one is most useful when it helps you read other people's code, not only your own. As you explore the FreeBSD tree you will encounter handlers that look nothing like `null_write` or `zero_read`. The shape will still be there; the decoration will differ. Here is a small reading protocol that takes the guesswork out.

Step one: find the return type and argument names. Every `d_read_t` and `d_write_t` takes the same three arguments. If the handler has renamed them from `dev`, `uio`, and `ioflag`, note what the author chose (`cdev`, `u`, `flags` are all common). Keep those names in mind as you read.

Step two: find the `uiomove` call (or relative). Track backward from there to understand what kernel pointer is being handed to it and what length. That pair is the heart of the handler. Everything before the `uiomove` call is preparing the pointer and length; everything after is accounting.

Step three: find the lock acquisition and release. A handler that takes a lock before `uiomove` and releases it after is serializing with other handlers. A handler with no lock is either operating on read-only data or using some other synchronisation primitive (a condition variable, a `refcount`, a reader lock). Name which it is.

Step four: find the `errno` returns. List the `errno` values the handler can produce. If the list is short and each value has an obvious trigger, the handler is well-written. If the list is long or opaque, the author probably left loose ends.

Step five: find the state transitions. What counters does the handler increment? What per-handle fields does it touch? Those transitions are the driver's behavioural signature, and they are usually the part that differs most between one driver and the next.

Apply this protocol to `zero_read` in `/usr/src/sys/dev/null/null.c`. The argument names are the standard ones. The `uiomove` call hands the kernel pointer `zbuf` (pointing at `zero_region`) and a length clamped by `ZERO_REGION_SIZE`. There is no lock; the data is constant. The only `errno` the handler can return is whatever `uiomove` returned. There are no state transitions; `/dev/zero` is stateless.

Now apply the same protocol to `myfirst_write` at Stage 3. Argument names: standard. `uiomove` call: kernel pointer `sc->buf + bufhead + bufused`, length `MIN((size_t)uio->uio_resid, avail)`. Lock: `sc->mtx`

taken before and released after. Errno returns: ENXIO (device gone), ENOSPC (buffer full), EFAULT via uiomove, or zero. State transitions: `sc->bufused += towrite`, `sc->bytes_written += towrite`, `fh->writes += towrite`.

Two drivers, same protocol, two coherent descriptions of what the handler does. Once you apply this reading habit half a dozen times, unfamiliar handlers stop looking unfamiliar.

9.5.8 What If You Leave `d_read` or `d_write` Unset?

A detail that beginners sometimes wonder about: what happens if your `cdevsw` does not set `.d_read` or `.d_write`? The short answer is that the kernel substitutes a default that returns either `ENODEV` or acts like a no-op, depending on which specific slot is empty and which other `d_flags` are set. The long answer is worth knowing because real drivers do use the defaults, on purpose, when they want to express “this device does not do reads” or “writes are silently discarded”.

Look at how `/usr/src/sys/dev/null/null.c` wires up its three drivers:

```
static struct cdevsw null_cdevsw = {
    .d_version =    D_VERSION,
    .d_read =      (d_read_t *)nullop,
    .d_write =     null_write,
    .d_ioctl =     null_ioctl,
    .d_name =      "null",
};
```

`.d_read` is set to the kernel helper `nullop`, cast to a `d_read_t *`. `nullop` is a universal “do nothing, return zero” function declared in `/usr/src/sys/sys/system.h` and defined in `/usr/src/sys/kern/kern_conf.c`; it takes no arguments and returns zero. It is used across the kernel wherever a method slot needs a harmless default. The cast works because `d_read_t` expects a function returning `int`, and `nullop`’s `int (*)(void)` shape is close enough for the `cdevsw` dispatch to call it without surprise.

For `/dev/null`, `(d_read_t *)nullop` means “every read returns zero bytes, forever”. A user who `cat /dev/null` sees an immediate EOF. This is different from `/dev/zero`, which installs `zero_read` to produce an infinite stream of zero bytes. The contrast between the two drivers is a contrast between two default read behaviours, and both are exactly one line in the `cdevsw`.

If you omit both `.d_read` and `.d_write` entirely, the kernel fills them with defaults that return `ENODEV`. That is the right choice when the device honestly does not support data transfer; callers see a clear error rather than silent success. But for devices that should quietly accept writes or produce zero-byte reads, setting the slot to `(d_read_t *)nullop` is the idiomatic FreeBSD gesture.

Practical rule: decide deliberately. Either implement the handler (for real behaviour), or set it to `(d_read_t *)nullop` / `(d_write_t *)nullop` (for harmless defaults), or leave it entirely unset (for `ENODEV`). Every real driver in the tree picks one of these three on purpose, and the choice is visible to users.

9.5.9 A Second Real Driver: How mem(4) Uses One Handler for Both Directions

null.c is the canonical minimal example. A slightly richer example is worth a look before we move on, because it demonstrates a pattern you will meet often in the tree: **a single handler that serves both d_read and d_write**, relying on uio->uio_rw to tell the two directions apart.

The driver is mem(4), which exposes /dev/mem and /dev/kmem. The common pieces live in /usr/src/sys/dev/mem/memdev.c, and the architecture-specific read and write logic lives under /usr/src/sys/<arch>/<arch>/mem.c. On amd64, the file is /usr/src/sys/amd64/amd64/mem.c, and the function is memrw.

Look first at the cdevsw:

```
static struct cdevsw mem_cdevsw = {
    .d_version =    D_VERSION,
    .d_flags =      D_MEM,
    .d_open =       memopen,
    .d_read =       memrw,
    .d_write =      memrw,
    .d_ioctl =      memioctl,
    .d_mmap =       memmmap,
    .d_name =       "mem",
};
```

Both .d_read and .d_write point at the same function. This is legal because the d_read_t and d_write_t typedefs are identical (both are int (*)(struct cdev *, struct uio *, int)), so a single function can satisfy both. The trick is reading uio->uio_rw inside the handler to decide which direction to move.

A condensed sketch of memrw looks like this:

```
int
memrw(struct cdev *dev, struct uio *uio, int flags)
{
    struct iovec *iov;
    /* ... locals ... */
    ssize_t orig_resid;
    int error;

    error = 0;
    orig_resid = uio->uio_resid;
    while (uio->uio_resid > 0 && error == 0) {
        iov = uio->uio_iov;
        if (iov->iov_len == 0) {
            uio->uio_iov++;
            uio->uio_iovcnt--;
            continue;
        }
    }
```

```

    }
    /* compute a page-bounded chunk size into c */
    /* ... direction-independent mapping logic ... */
    error = uiomove(kernel_pointer, c, uio);
}
/*
 * Don't return error if any byte was written. Read and write
 * can return error only if no i/o was performed.
 */
if (uio->uio_resid != orig_resid)
    error = 0;
return (error);
}

```

There are three ideas in this sketch that generalise to your own drivers.

First, one handler for both directions saves code when the per-byte work is identical. The mapping logic in `memrw` resolves a user-space offset to a piece of kernel-accessible memory; whether you are reading from or writing to that memory is decided later, by `uiomove` looking at `uio->uio_rw`. You save yourself the duplication of a near-identical read-vs-write pair at the cost of a single function that has to be clear about which direction it is in. If the two directions share almost nothing, write two functions; if they share nearly everything, combine them.

Second, `memrw` walks the `iovec` itself. Unlike `myfirst`, which hands the whole transfer to `uiomove` in one or two calls, `memrw` walks `iovec` entries explicitly so that it can map each requested offset into kernel memory and then call `uiomove` on the mapped region. This is the pattern you use when the *kernel pointer* your driver hands to `uiomove` depends on the offset being serviced. It is less common than the `myfirst` style, but it is the right shape when each chunk of the transfer corresponds to a different piece of the driver's backing storage.

Third, note the `orig_resid` trick at the end. The handler saves `uio_resid` at entry, then after the loop checks whether anything at all moved. If something did, it returns zero (success) even if an error happened later, because UNIX conventions require that a read or write with a non-zero byte count return that count to the caller rather than failing the whole call. This is the “partial success” idiom: if any byte moved, report the byte count; only fail when no bytes moved at all.

Your `myfirst` handlers do not need the idiom, because they call `uiomove` exactly once. If `uiomove` succeeds, everything moved; if it fails, nothing moved (from the driver's accounting point of view). The `orig_resid` idiom matters when your handler loops and the loop can be interrupted mid-way by an error from `uiomove`. Remember the pattern; you will use it in later chapters when your driver serves data from multiple sources.

9.5.10 Why This Walkthrough Was Worth the Detour

Two drivers. Two very different backing stores. One primitive. In `null.c`, `zero_read` serves a pre-allocated zero region; in `memrw`, the handler serves physical memory mapped on demand. The code looks different in the middle, because the middle is where the driver's unique knowledge lives. The ends look the same: both functions take a `uio`, both loop on `uio_resid`, both call `uiomove(9)` to do the actual transfer, both return zero on success or an `errno`.

That uniformity is the point. Every character-device read and write in the tree obeys this shape. Once you recognise it, you can open any unfamiliar driver under `/usr/src/sys/dev` and read the handler with confidence: the part you do not yet understand is always the middle, never the ends.

9.6 Understanding the `ioflag` Argument

Both `d_read` and `d_write` receive a third argument that the rest of the chapter has barely used so far. This section is the short but useful explanation of what `ioflag` is, where it comes from, and when a character-device driver should actually look at it.

9.6.1 Where `ioflag` Comes From

Every time a process performs a `read(2)` or `write(2)` on a `devfs` node, the kernel composes an `ioflag` value from the current file-descriptor flags before calling your handler. The composition lives in `devfs` itself, in `/usr/src/sys/fs/devfs/devfs_vnops.c`. The relevant lines from `devfs_read_f` are:

```
ioflag = fp->f_flag & (O_NONBLOCK | O_DIRECT);
if (ioflag & O_DIRECT)
    ioflag |= IO_DIRECT;
```

The pattern in `devfs_write_f` is the mirror image. The kernel takes whichever bits of the file table's `f_flag` word are interesting for I/O, masks them out, and passes that subset on as `ioflag`.

This is important for two reasons. First, it means the `ioflag` your driver receives is a *snapshot*. If the user program changes its non-blocking setting (via `fcntl(F_SETFL, O_NONBLOCK)`) between two `read(2)` calls, each call will carry its own up-to-date `ioflag`. You do not need to cache the state or watch for changes; the kernel re-derives the value at every dispatch.

Second, it means most of the constants you might expect to see never reach your handler. Things like `O_APPEND`, `O_TRUNC`, `O_CLOEXEC`, and the various `O_EXLOCK`-style flags belong to the filesystem and file-table layers. They do not influence character-device I/O and are not forwarded.

9.6.2 The Flag Bits That Matter

The `IO_*` flags are declared in `/usr/src/sys/sys/vnode.h`. For character-device drivers, only a small subset is worth remembering:

```
#define IO_UNIT      0x0001    /* do I/O as atomic unit */
#define IO_APPEND    0x0002    /* append write to end */
#define IO_NDELAY    0x0004    /* FNDELAY flag set in file table */
#define IO_DIRECT    0x0010    /* attempt to bypass buffer cache */
```

Of these, **only `IO_NDELAY` and `IO_DIRECT` are composed into the `ioflag` your handler receives**. The first three bits exist for filesystem I/O. A character-device driver that inspects `IO_UNIT` or `IO_APPEND` is looking at values that will always be zero.

`IO_NDELAY` is the common case. It is set when the descriptor is in non-blocking mode. A driver that implements blocking reads (Chapter 10) uses this bit to decide between sleeping and returning `EAGAIN`. A Chapter 9 driver does not sleep on anything, so the bit is informational only, but future chapters rely on it.

`IO_DIRECT` is a hint that the user program opened the descriptor with `O_DIRECT`, asking the kernel to bypass buffer caches where possible. For a simple character driver it is almost always irrelevant. Storage-adjacent drivers may choose to honour it; most do not.

Note the numerical identity: `O_NONBLOCK` in `/usr/src/sys/sys/fcntl.h` has the value `0x0004`, and `IO_NDELAY` in `/usr/src/sys/sys/vnode.h` has the same value. That is not an accident. The header comment above the `IO_*` definitions explicitly states that `IO_NDELAY` and `IO_DIRECT` are aligned with the corresponding `fcntl(2)` bits so that devfs does not need to translate. Your driver can inspect the bit either way and get the same answer.

9.6.3 A Handler That Checks `ioflag`

Here is what a non-blocking-aware read handler looks like at the skeleton level. We will not use this shape in Chapter 9 because we never sleep, but studying it now makes Chapter 10's introduction quicker.

```
static int
myfirst_read_nb(struct cdev *dev, struct uio *uio, int ioflag)
{
    struct myfirst_softc *sc = dev->si_drv1;
    int error;

    mtx_lock(&sc->mtx);
    while (sc->bufused == 0) {
        if (ioflag & IO_NDELAY) {
            mtx_unlock(&sc->mtx);
            return (EAGAIN);
        }
        /* ... would msleep(9) here in Chapter 10 ... */
    }
    /* ... drain buffer, uiomove, unlock, return ... */
    error = 0;
    mtx_unlock(&sc->mtx);
    return (error);
}
```

The branch on `IO_NDELAY` is the only decision the handler makes about blocking. Everything else in the function is ordinary I/O code. That narrowness is part of why `ioflag` is a single integer: a driver's response to flag bits is usually a single `if` statement near the top of the handler, not a sprawling state machine.

9.6.4 What the Chapter 9 Stages Do With `ioflag`

Stage 1, Stage 2, and Stage 3 do **not** inspect `ioflag`. They cannot block, so the non-blocking bit is moot; they do not care about `IO_DIRECT`. The argument is present in their handler signatures because the typedef demands it, and it is silently ignored.

Silently ignoring an argument is not a bug when the ignored behaviour is obviously correct. A reader who opens one of our descriptors with `O_NONBLOCK` will see identical behaviour to a reader who did not: neither call ever sleeps, so the flag has no observable effect. Chapter 10 is where we will wire the flag through.

9.6.5 A Small Debugging Aid

If you are curious what `ioflag` contains during a test, a single `device_printf` at entry will tell you:

```
device_printf(sc->dev, "d_read: ioflag=0x%x resid=%zd offset=%jd\n",
    ioflag, (ssize_t)uio->uio_resid, (intmax_t)uio->uio_offset);
```

Load the driver, run `cat /dev/myfirst/0`, and observe the hex value. Then run a small program that uses `fcntl(fd, F_SETFL, O_NONBLOCK)` before reading, and observe the difference. This is an instructive two-minute detour when you are first making the mechanism real in your head.

9.6.6 `ioflag` in the Tree

Search `/usr/src/sys/dev` for `IO_NDELAY` and you will find dozens of hits. Almost every one of them is the same pattern: check the bit, return `EAGAIN` if set and the driver has nothing to serve, otherwise sleep. The uniformity is deliberate. FreeBSD drivers treat non-blocking I/O the same way whether they are pseudo-devices, TTY lines, USB endpoints, or GEOM-backed storage, and that consistency is part of why user programs written for one kind of device port cleanly to another.

9.7 Understanding `struct uio` Deeply

`struct uio` is the kernel's representation of an I/O request. It is passed to every `d_read` and `d_write` invocation. Every successful `uiomove(9)` call mutates it. Every driver author you will ever meet has at some point stared at its fields and wondered which ones to trust. This section is where we make the structure less mysterious.

9.7.1 The Declaration

From `/usr/src/sys/sys/uio.h`:

```
struct uio {
    struct iovec *uio_iov;          /* scatter/gather list */
    int uio_iovcnt;                /* length of scatter/gather list */
```

```

    off_t    uio_offset;           /* offset in target object */
    ssize_t  uio_resid;           /* remaining bytes to process */
    enum     uio_seg uio_segflg;   /* address space */
    enum     uio_rw uio_rw;       /* operation */
    struct    thread *uio_td;      /* owner */
};

```

Seven fields. Each one has a specific purpose, and there is exactly one function, `uiomove(9)`, that uses all of them in concert. Your driver will read some of the fields directly; a few fields you will never touch.

9.7.2 `uio_iov` and `uio_iovcnt`: The Scatter-Gather List

A single `read(2)` or `write(2)` operates on one contiguous user buffer. The related `readv(2)` and `writev(2)` operate on a list of buffers (an “iovec”). The kernel represents both cases uniformly as a list of `iovec` entries, using a list of length one for the simple case.

`uio_iov` points at the first entry of that list. `uio_iovcnt` is the number of entries. Each entry is a struct `iovec`, declared in `/usr/src/sys/sys/_iovec.h`:

```

struct iovec {
    void      *iov_base;
    size_t    iov_len;
};

```

`iov_base` is a pointer into user memory (for a `UIO_USERSPACE` `uio`) or kernel memory (for a `UIO_SYSSPACE` `uio`). `iov_len` is the number of bytes remaining in that entry.

You will almost never touch these fields directly. `uiomove(9)` walks the `iovec` list for you, consuming entries as it moves bytes, and leaving the list consistent with the remaining transfer. If your driver reaches for `uio_iov` or `uio_iovcnt` by hand, you are either writing a very unusual driver or doing something wrong. The conventional pattern is: let `uiomove` manage the `iovec`, and read the other fields to understand the state of the request.

9.7.3 `uio_offset`: The Offset Into the Target

For a read or write on a regular file, `uio_offset` is the position in the file where the I/O is taking place. The kernel increments it as bytes move, so a sequential `read(2)` naturally advances through the file.

For a device file, the meaning of `uio_offset` is defined by the driver. A device that is truly sequential and has no notion of position will ignore the incoming value and leave the outgoing value to reflect whatever `uiomove` did. A device that backs itself with a fixed buffer will treat the offset as an address into that buffer and honour it.

`uiomove(9)` updates `uio_offset` in lockstep with `uio_resid`: for every byte it moves, it decrements `uio_resid` by one and increments `uio_offset` by one. If your driver calls `uiomove` once per handler, you rarely need to read `uio_offset` by hand. If your driver calls `uiomove` more than once, or if it uses the offset to index into its own buffer, `uiomove_frombuf(9)` is the helper you want.

9.7.4 `uio_resid`: The Remaining Bytes

`uio_resid` is the number of bytes that still need to move. At the start of `d_read`, it is the total length the user asked for. At the end of a successful transfer, it is whatever did not move; the kernel subtracts this from the original length to produce the return value of `read(2)`.

Two signed-arithmetic traps are worth calling out. First, `uio_resid` is an `ssize_t`, which is signed. A negative value is illegal (and `uiomove` will `KASSERT` on it in debug kernels), but beware of accidentally constructing one through sloppy arithmetic. Second, `uio_resid` can be zero at the start of the call. This happens when a user program calls `read(fd, buf, 0)` or `write(fd, buf, 0)`. Your handler must not treat zero as “no user intent” and then proceed to do I/O against what might be an uninitialised buffer. The safe pattern is to check for zero early and return zero (or accept zero and return zero, for writes). `uiomove` handles this cleanly: it returns zero immediately without touching anything. So in practice the “check early” is often redundant; what matters is that you do not *assume* it is non-zero.

9.7.5 `uio_segflg`: Where the Buffer Lives

This field says where the `iovec` pointers refer to: user space (`UIO_USERSPACE`), kernel space (`UIO_SYSSPACE`), or a direct object map (`UIO_NOCOPY`). The enumeration is in `/usr/src/sys/sys/_uio.h`:

```
enum uio_seg {
    UIO_USERSPACE,      /* from user data space */
    UIO_SYSSPACE,       /* from system space */
    UIO_NOCOPY          /* don't copy, already in object */
};
```

For a `d_read` or `d_write` called on behalf of a user syscall, `uio_segflg` is `UIO_USERSPACE`. `uiomove(9)` reads the field and picks the right transfer primitive: `copyin` / `copyout` for user-space segments, `bcopy` for kernel-space segments. Your driver does not need to branch on this; `uiomove` does it for you.

You will occasionally see code that builds a kernel-mode `uio` by hand, typically to re-use a function that takes a `uio` but serve it from a kernel buffer. That code sets `uio_segflg` to `UIO_SYSSPACE`. It is legitimate and useful, and we will meet it briefly in the labs. Do not confuse it with a user-space `uio`: the safety properties are very different.

9.7.6 `uio_rw`: The Direction

The direction of transfer. The enumeration is in the same header:

```
enum uio_rw {
    UIO_READ,
    UIO_WRITE
};
```

For a `d_read` handler, `uio_rw` is `UIO_READ`. For a `d_write` handler, `uio_rw` is `UIO_WRITE`. The field tells `uiomove` whether to copy kernel->user (read) or user->kernel (write). Some handlers assert on this as a sanity check:

```
KASSERT(uio->uio_rw == UIO_READ,
        ("Can't be in %s for write", __func__));
```

That assertion is from `zero_read` in `/usr/src/sys/dev/null/null.c`. It is a cheap way to document the invariant. Your driver does not need assertions like this to be correct, but they can be a useful safety net during development.

9.7.7 `uio_td`: The Owning Thread

The `struct thread *` of the caller. For a `uio` built on behalf of a syscall, this is the thread that made the syscall. Some kernel APIs want a thread pointer; using `uio->uio_td` rather than `curthread` keeps the association explicit when the `uio` is handed around.

In a straightforward `d_read` or `d_write`, you rarely need `uio_td`. It becomes useful if your driver wants to inspect the credentials of the caller mid-call, beyond what `open(2)` already validated. That is uncommon.

9.7.8 An Illustration: What Happens to `uio` During a `read(fd, buf, 1024)` Call

Walking through a single `read(2)` helps cement how the fields move. Suppose a user program calls:

```
ssize_t n = read(fd, buf, 1024);
```

The kernel constructs a `uio` that looks roughly like this when it reaches your `d_read`:

- `uio_iov` points to a one-entry list.
- The one entry has `iov_base = buf` (the user's buffer) and `iov_len = 1024`.
- `uio_iovcnt = 1`.
- `uio_offset = <wherever the current file pointer was>`. For a newly opened seekable device, zero.
- `uio_resid = 1024`.
- `uio_segflg = UIO_USERSPACE`.
- `uio_rw = UIO_READ`.
- `uio_td = <calling thread>`.

Your handler calls, say, `uiomove(sc->buf, 300, uio)`. Inside `uiomove`, the kernel:

- Takes the first `iovec` entry.
- Determines that 300 is less than 1024, so it will move 300 bytes.
- Calls `copyout(sc->buf, buf, 300)`.
- Decrements `iov_len` by 300, to 724.
- Advances `iov_base` by 300, to `buf + 300`.

- Decrements `uio_resid` by 300, to 724.
- Increments `uio_offset` by 300.

Your handler returns zero. The kernel computes the byte count as $1024 - 724 = 300$ and returns 300 from `read(2)`. The user sees 300 bytes in `buf[0..299]` and knows to either call `read(2)` again to get the rest, or proceed with what they have.

That is everything `uiomove` is doing, in order. There is no magic.

9.7.9 How `readv(2)` Differs

If the user called `readv(fd, iov, 3)` with three `iovec` entries, the `uio` at the start of `d_read` would have `uio_iovcnt = 3`, `uio_iov` pointing at the list of three entries, and `uio_resid` equal to the sum of their lengths. Your handler makes one `uiomove` call (or several, in a loop) and `uiomove` walks the list for you. The driver code is identical.

This is one of the quiet benefits of the `uio` abstraction: scatter-gather reads and writes are free. Your driver was written for a single buffer; it already handles multi-buffer requests.

9.7.10 Re-entering a Handler With a Partly Consumed `uio`

A convention that occasionally trips beginners: **a single `d_read` or `d_write` invocation can make multiple `uiomove` calls**. Each call shrinks `uio_resid` and advances `uio_iov`. The `uio` remains consistent between calls. If your handler's first `uiomove` moves 128 bytes and the next moves 256, the kernel just sees a single handler call that transferred 384 bytes.

What you should **not** do is save a `uio` pointer across a handler call and try to resume it later. A `uio` is valid for the duration of the dispatch that produced it. Between dispatches, the memory it points at (including the `iovec` array) may not be valid. If you need to queue a request for later processing, you copy the necessary data out of the `uio` (into your own kernel buffer) and use your own queue.

9.7.11 What Your Driver Needs to Read and What It Should Leave Alone

A short crib sheet, in decreasing order of frequency:

Field	Read it?	Write it?
<code>uio_resid</code>	Yes, often	Only to mark a transfer consumed (e.g., <code>uio_resid = 0</code>)
<code>uio_offset</code>	Yes, if you honour it	No, let <code>uiomove</code> update it
<code>uio_rw</code>	Occasionally, for KASSERTs	No
<code>uio_segflg</code>	Rarely	No, unless building a kernel-mode <code>uio</code>
<code>uio_td</code>	Rarely	No
<code>uio_iov</code>	Almost never	Never
<code>uio_iovcnt</code>	Almost never	Never

Field	Read it?	Write it?

If a beginner-level driver writes to `uio_iov` or `uio_iovcnt`, something has gone seriously off the rails. If it writes to `uio_resid` other than the `uio_resid = 0` “I consumed everything” trick, something is slightly off. If it reads from the first three rows, it is on the normal path.

9.7.12 The uio Fields in Practice

All of this is less intimidating once you have seen a handler actually use it. The myfirst stages in this chapter inspect `uio_resid` (to clamp transfers), occasionally read `uio_offset` (to know where a reader is), and hand everything else to `uiomove`. The helpers do the real work, and the driver code stays small.

9.7.13 The Life of a Single uio: Three Snapshots

To cement the field-by-field discussion, it is worth walking through a `uio`’s state at three points in its life: the moment your handler is called, the moment after a partial `uiomove`, and the moment just before your handler returns. Each snapshot captures the same `uio`, so you can see exactly how the fields evolve.

The example is a `read(fd, buf, 1024)` call on a driver whose read handler will serve 300 bytes per call.

Snapshot 1: At the entry of `d_read`.

```
uio_iov      -> [ { iov_base = buf,          iov_len = 1024 } ]
uio_iovcnt   = 1
uio_offset   = 0          (this is the first read on the descriptor)
uio_resid    = 1024
uio_segflg   = UIO_USERSPACE
uio_rw       = UIO_READ
uio_td       = <calling thread>
```

The `uio` describes a complete request. The user asked for 1024 bytes, the buffer is in user space, the direction is read, the offset is zero. This is what the kernel hands your handler.

Snapshot 2: After `uiomove(sc->buf, 300, uio)` returns success.

```
uio_iov      -> [ { iov_base = buf + 300, iov_len = 724 } ]
uio_iovcnt   = 1
uio_offset   = 300
uio_resid    = 724
uio_segflg   = UIO_USERSPACE      (unchanged)
uio_rw       = UIO_READ           (unchanged)
uio_td       = <calling thread>   (unchanged)
```

Four fields changed in lockstep. `iov_base` advanced by 300 so the next transfer will place bytes after the ones just written. `iov_len` shrunk by 300, because the `iovec` entry now describes only the remaining 724 bytes. `uio_offset` grew by 300, because 300 bytes of stream position moved. `uio_resid` shrunk by 300, because 300 bytes of work are done.

Three fields stayed fixed: `uio_segflg`, `uio_rw`, and `uio_td` describe the *shape* of the request, which does not change mid-transfer. If your handler ever needs to check one of them, it can do so before or after `uiomove` and get the same answer.

Snapshot 3: Just before `d_read` returns.

Imagine the handler, having served 300 bytes, decides it has no more data and returns zero without calling `uiomove` again.

```
uio_iov      -> [ { iov_base = buf + 300, iov_len = 724 } ]
uio_iovcnt   = 1
uio_offset   = 300
uio_resid    = 724
uio_segflg   = UIO_USERSPACE
uio_rw       = UIO_READ
uio_td       = <calling thread>
```

Identical to Snapshot 2. The handler did not touch anything; it just returned. The kernel will see `uio_resid` = 724 versus the starting `uio_resid` = 1024 and compute $1024 - 724 = 300$, which it returns to user space as the result of `read(2)`. The caller sees a return value of 300 and knows the driver produced 300 bytes.

If the handler had looped on `uiomove` until `uio_resid` reached zero, the snapshot at return would instead have `uio_resid` = 0 and the kernel would return 1024 to user space (a full transfer). If the handler had called `uiomove` and got an error, `uio_resid` would reflect whatever partial progress happened before the fault, and the handler would return the `errno`.

9.7.14 What This Mental Model Buys You

Three observations fall out of the snapshots that are worth naming explicitly.

First, `uio_resid` is the contract. Whatever value is in `uio_resid` when your handler returns, the kernel will trust. If it is smaller than at entry, some bytes moved; the difference is the byte count. If it is unchanged, nothing moved; the return value will be zero (EOF) or an `errno` (depending on what your handler returned).

Second, `uiomove` is the only thing you should rely on to decrement `uio_resid`. A driver that manually subtracts from `uio_resid` is almost certainly doing something wrong; the kernel's fault handling, `iovec` walking, and offset updates are all baked into the `uiomove` code path. Setting `uio_resid` = 0 is the one exception, used by drivers like `null.c`'s `null_write` to say "pretend all the bytes were consumed".

Third, the `uio` is scratch space. A `uio` is not a long-lived object. It is created per `syscall`, decays as `uiomove` consumes it, and is discarded when your handler returns. Saving a `uio` pointer for later use is a lifetime bug waiting to fire. If your driver needs data from the `uio` beyond the current call, it copies the bytes out into its own storage (which is what `d_write` does: it copies bytes through `uiomove` into `sc->buf`, leaving nothing in the `uio` for later).

These three facts are the foundation everything else in the chapter is built on. If you internalise them, the rest of the uio machinery stops looking mysterious.

9.8 Safe Data Transfer: `uiomove`, `copyin`, `copyout`

The previous sections described `struct uio` and named `uiomove(9)` as the function that moves bytes. This section explains why that function exists, what it does under the hood, and when a driver should reach for `copyin(9)` or `copyout(9)` directly instead.

9.8.1 Why Direct Memory Access Is Unsafe

A user process has its own virtual address space. When a process calls `read(2)` with a buffer pointer, that pointer is a virtual address in the process's address space. It may refer to a page of memory that is present in physical RAM, or to a page that has been swapped out, or to a page that is not mapped at all. It may even be a pointer the user program fabricated deliberately to try to make the kernel crash.

From the kernel's point of view, the user's address space is not directly addressable. The kernel has its own address space; a user pointer handed to the kernel is not meaningful as a kernel pointer. Even if the kernel can resolve the user pointer through page-table machinery, using it directly is dangerous: the page might fault, the memory protection might be wrong, the address might fall outside the process's mapped regions, or the pointer might have been constructed to point at kernel memory in an attempt to leak or corrupt it.

Direct memory access, in other words, is not a feature the kernel gets for free. It is a privilege that must be exercised carefully, with every access routed through functions that know how to handle faults, check protection, and keep the user and kernel address spaces distinct.

On FreeBSD those functions are `copyin(9)` (user to kernel), `copyout(9)` (kernel to user), and `uiomove(9)` (either direction, driven by the `uio`).

9.8.2 What `copyin(9)` and `copyout(9)` Do

From `/usr/src/sys/sys/systm.h`:

```
int copyin(const void * __restrict udaddr,
           void * __restrict kaddr, size_t len);

int copyout(const void * __restrict kaddr,
            void * __restrict udaddr, size_t len);
```

`copyin` takes a user-space pointer, a kernel-space pointer, and a length. It copies `len` bytes from user to kernel. `copyout` is the reverse: kernel pointer, user pointer, length. Copies kernel to user.

Both functions validate the user address, fault in the user page if necessary, perform the copy, and catch any fault that occurs. They return zero on success and `EFAULT` if the user address was invalid or the copy otherwise failed. They never silently corrupt memory; they always either complete the copy or report the error.

These two primitives are the foundation on which all user / kernel memory transfers are built. They are what `uiomove(9)` calls underneath when the `uio` is in user space. They are what `fubyte(9)`, `subyte(9)`, and a handful of other convenience functions use. They are the functions the kernel trusts to be the trust boundary.

9.8.3 What `uiomove(9)` Does

`uiomove(9)` is a wrapper around `copyin / copyout` that understands the `uio` structure. Its implementation is short and worth reading; it lives in `/usr/src/sys/kern/subr_uio.c`.

Roughly, the algorithm is:

1. Sanity-check the `uio`: the direction is valid, the `resid` is non-negative, the owning thread is the current thread if the segment is user-space.
2. Loop: while the caller has asked for more bytes (`n > 0`) and the `uio` still has room (`uio->uio_resid > 0`), consume the next `iovec` entry.
3. For each `iovec` entry, compute how many bytes to move (the minimum of the entry length, the caller's remaining count, and the `uio`'s `resid`), and call `copyin` or `copyout` (for user-space segments) or `bcopy` (for kernel-space segments), depending on `uio_rw` and `uio_segflg`.
4. Advance the `iovec`'s `iov_base` and `iov_len` as bytes move; decrement `uio_resid`, increment `uio_offset`.
5. If any copy fails, break out and return the error.

The function returns zero on success or an `errno` code on failure. The most common failure is `EFAULT` from a bad user pointer.

The critical property of `uiomove` is that it is **the only function your driver should use to move bytes through a `uio`**. Not `bcopy`, not `memcpy`, not `copyout`. The `uio` carries the information `uiomove` needs to pick the right primitive, and the driver does not need to second-guess.

9.8.4 When to Use Which

The division of labour is straightforward in practice.

Use `uiomove(9)` in `d_read` and `d_write` handlers whenever the `uio` describes the transfer. This is the overwhelmingly common case.

Use `copyin(9)` and `copyout(9)` directly when you have a user pointer from somewhere other than a `uio`. Examples:

- Inside a `d_ioctl` handler, for control commands that carry user-space pointers as arguments (Chapter 25).
- Inside a kernel thread that accepts user-supplied data through a mechanism you built yourself, not through a `uio`.
- When reading or writing a small, fixed-size piece of user memory that is not the subject of the syscall.

Do **not** use `copyin` or `copyout` inside `d_read` or `d_write` to fetch from the `uio`'s `iovec`. Always go through `uiomove`. The `iovec` is not guaranteed to be a single contiguous buffer, and even if it is, your driver has no business reaching past the `uio` abstraction to touch it directly.

9.8.5 A Table for Quick Reference

Situation	Preferred tool
Transferring bytes through a uio (read or write)	<code>uiomove(9)</code>
Transferring bytes through a uio, with a fixed kernel buffer and automatic offset	<code>uiomove_frombuf(9)</code>
Reading a known user pointer not carried by a uio	<code>copyin(9)</code>
Writing to a known user pointer not carried by a uio	<code>copyout(9)</code>
Reading a null-terminated string from user space	<code>copyinstr(9)</code>
Reading a single byte from user space	<code>fubyte(9)</code>
Writing a single byte to user space	<code>subyte(9)</code>

`fubyte` and `subyte` are niche; most drivers never use them. They are listed here for recognition. `copyinstr` is occasionally useful in control paths that take a user string; we will not reach for it in this chapter.

9.8.6 Why Not a Straight `memcpy`?

A beginner sometimes asks “can I just cast the user pointer and `memcpy` the bytes?” The answer is an unqualified no, and it is worth understanding why.

`memcpy` assumes both pointers refer to accessible memory in the current address space. A user pointer is not guaranteed to refer to accessible memory. On architectures that separate user and kernel pointers at the hardware level (SMAP on amd64, for example), the CPU will refuse the access. On architectures that share the address space, the pointer may still be invalid, or may refer to a page that has been swapped out, or may refer to a page the kernel is forbidden to touch. None of these cases are safe to handle inside a plain `memcpy`; the resulting fault would either panic the system or leak information across the trust boundary.

The kernel primitives `copyin` and `copyout` exist precisely to handle these cases correctly. They install a fault handler before the access, so a bad user pointer returns `EFAULT` instead of panicking. They respect SMAP and similar protections. They can wait for the page to be paged in. None of that is optional, and none of it is something your driver should replicate.

The practical rule: if a pointer came from user space, route it through `copyin` / `copyout` / `uiomove`. Do not dereference it directly. Do not `memcpy` through it. Do not pass it to any function that will `memcpy` through it. If you stop at the abstraction boundary, the kernel gives you a stable, safe, well-documented interface. If you cross it, you own every bug forever.

9.8.7 What Happens Under a Fault

Something concrete: what does `uiomove` actually do when the user pointer is bad?

The kernel installs a fault handler before the copy, typically through its trap-handler table. When the CPU takes a fault on the user access, the fault handler notices that the faulting instruction is inside a `copyin` or `copyout`

code path, skips ahead to the failure-return path, and returns `EFAULT`. No panic. No data corruption. The caller of `uiomove` sees a non-zero return value, propagates it to the caller of `d_read` or `d_write`, and the syscall returns to userland with `errno = EFAULT`.

The driver does not need to do anything special to cooperate with this machinery. It just needs to check the return value of `uiomove` and propagate errors. We will do that in every handler in the chapter.

9.8.8 Alignment and Type Safety

One more subtlety worth naming. The user's buffer is a stream of bytes. It carries no type information. If your driver puts a struct into the buffer and the user pulls it out, the user gets bytes; those bytes may or may not be correctly aligned for a struct access on the caller's architecture.

For `myfirst` the problem does not arise, because the bytes are arbitrary user text. For drivers that want to export structured data, the convention is either to require the user to memcpy the bytes into an aligned local structure before interpreting them, or to include an explicit alignment and version negotiation in the data format. `ioctl(2)` sidesteps the problem because its data layout is part of the `IOCTL` command number; `read` and `write` do not have that luxury.

This is one of the places where bolting structured data onto `read/write` is tempting and wrong. If your driver wants to hand typed data to userland, the `ioctl` interface or an external RPC mechanism are the right tools. `read` and `write` carry bytes. That is the promise, and the promise is what keeps them portable.

9.8.9 A Small Worked Example

Suppose a user program writes four integers to your device:

```
int buf[4] = { 1, 2, 3, 4 };
write(fd, buf, sizeof(buf));
```

In the driver's `d_write`, the `uio` looks like:

- One `iovec` entry with `iov_base = <user address of buf[0]>`, `iov_len = 16`.
- `uio_resid = 16`, `uio_offset = 0`, `uio_segflg = UIO_USERSPACE`, `uio_rw = UIO_WRITE`.

A naive handler might call `uiomove(sc->intbuf, 16, uio)`, where `sc->intbuf` is `int sc->intbuf[4]`; `uiomove` would issue a copyin that copies the 16 bytes. On success, `sc->intbuf` would hold the four integers in the byte order of the calling program.

But note: the user may have written those integers in the byte order of a completely different CPU if the driver is ever used across architectures. The user may have used `int32_t` where the driver used `int`. The user may have padded the structure differently. For `myfirst` none of this matters because we treat the data as opaque bytes. For a driver that exposes structured data over `read/write`, these issues multiply quickly, and they are the reason most real drivers either use `ioctl` for structured payloads or declare an explicit wire format (byte order, field widths, alignment) in their documentation.

The lesson: `uiomove` moves bytes. It does not know or care about types. Your driver has to decide what those bytes mean.

9.8.10 A Mini Case Study: When a Struct Round-Trip Goes Wrong

To make the “bytes are not types” point concrete, walk through a plausible but mistaken attempt to expose a kernel counter via `read(2)` as a typed structure.

Suppose your driver maintains a set of counters:

```
struct myfirst_stats {
    uint64_t reads;
    uint64_t writes;
    uint64_t errors;
    uint32_t flags;
};
```

And suppose, optimistically, you expose them via `d_read`:

```
static int
stats_read(struct cdev *dev, struct uio *uio, int ioflag)
{
    struct myfirst_softc *sc = dev->si_drv1;
    struct myfirst_stats snap;

    mtx_lock(&sc->mtx);
    snap.reads  = sc->stat_reads;
    snap.writes = sc->stat_writes;
    snap.errors = sc->stat_errors;
    snap.flags  = sc->stat_flags;
    mtx_unlock(&sc->mtx);

    return (uiomove(&snap, sizeof(snap), uio));
}
```

At first glance this is fine. The bytes reach user space. A reader can cast the buffer to a `struct myfirst_stats` and look at the fields. The author tests it on amd64, sees the correct values, ships the driver.

Three problems are sitting there waiting.

Problem 1: Struct padding. The layout of `struct myfirst_stats` depends on the compiler and architecture. On amd64 with the default ABI, `uint64_t` is 8-byte-aligned, so the struct is 8 bytes for `reads`, 8 for `writes`, 8 for `errors`, 4 for `flags`, plus 4 bytes of trailing padding to round the size to 32. The user program must declare a struct with the *same* padding to read the fields correctly. A user program that redeclares the struct with `#pragma pack(1)` or uses a different compiler version will parse the bytes wrong and see garbage in errors.

Problem 2: Byte order. An amd64 machine stores `uint64_t` little-endian. A user program running on the same architecture decodes correctly. A user program running remotely on a big-endian machine, reading the bytes through a network pipe, sees the integers byte-swapped. The driver did not choose an on-the-wire byte order, so the format is CPU-dependent by accident.

Problem 3: Atomicity of the snapshot. The reader may pull the bytes out through `uiomove` after `mtx_unlock` releases the mutex and before the kernel returns control to the caller. Between those two moments, the fields `snap.reads`, `snap.writes`, etc. are already captured in the stack-local `snap`, so *that* bit is fine. But the example is small enough that the bug does not appear; a larger snapshot might be captured across several mutex acquisitions and exhibit torn reads.

The fix is not to “try harder” with struct layout. The fix is to stop using `read(2)` for structured data. Two better options exist:

- **`sysctl`:** the chapter has been using this throughout. Individual counters are exposed as named nodes of known type. `sysctl(3)` on the user side returns integers directly; no struct layout, no padding, no byte order.
- **`d_ioctl`:** Chapter 25 builds out `ioctl` properly. For this use case, an `ioctl` with a well-defined request structure would be appropriate, and the `_IOR / _IOW` macros document the size and direction.

The `read(2)` interface promises “a byte stream the driver defines”; nothing more. If you respect the promise, your driver is portable, testable, and resistant to silent layout drift. If you break the promise by exposing typed structures, you inherit every ABI trap that network protocols spent decades learning to work around.

For `myfirst` in this chapter we never hit the problem, because we only ever push and pull opaque byte streams. The point of the case study is to help you recognise the shape of the mistake before someone hands you a driver that is already making it.

9.8.11 Summary of This Section

- Use `uiomove(9)` inside `d_read` and `d_write`. It reads the `uio`, picks the right primitive, and handles user / kernel faults for you.
- Use `uiomove_frombuf(9)` when you want automatic offset-into-buffer arithmetic.
- Use `copyin(9)` and `copyout(9)` only when you have a user pointer outside the `uio` context, typically in `d_ioctl`.
- Do not dereference user pointers directly. Ever.
- Check return values. Any copy can fail with `EFAULT`, and your handler must propagate the error.

These rules are short, but they cover nearly every I/O safety mistake that beginner drivers make.

9.9 Managing Internal Buffers in Your Driver

The read and write handlers are the visible surface of your driver’s I/O path. Behind them, the driver has to store data somewhere. This section is about how that storage is designed, allocated, protected, and cleaned up, at the beginner-friendly level that this chapter needs. Chapter 10 will extend the buffer into a true ring and make it concurrent-safe under load; we are staying short of that here on purpose.

9.9.1 Why You Need Buffers

A buffer is temporary storage between I/O calls. A driver uses one for at least three reasons:

1. **Rate matching.** Producers and consumers do not arrive at the same time. A write can deposit bytes that a later read will pick up.
2. **Request reshaping.** A user may read in units that do not line up with how the driver produces data. A buffer absorbs the mismatch.
3. **Isolation.** The bytes inside the driver's buffer are kernel data. They are not user pointers, not DMA addresses, not in a scatter-gather list. Everything in a kernel buffer is addressable by the driver safely and uniformly.

For `myfirst`, the buffer is a small piece of in-RAM storage. `d_write` writes into it; `d_read` reads from it. The buffer is the driver's state. The uio machinery is the plumbing that moves bytes in and out.

9.9.2 Static vs Dynamic Allocation

Two reasonable designs exist for where the buffer lives.

Static allocation puts the buffer inside the `softc` structure or as a module-level array:

```
struct myfirst_softc {  
    ...  
    char buf[4096];  
    size_t bufused;  
    ...  
};
```

Pros: allocation never fails, sizing is explicit, lifetime is trivially tied to the `softc`. Cons: the size is baked in at compile time; if you later want to make it tunable, you will refactor.

Dynamic allocation uses `malloc(9)` from an `M_*` bucket:

```
sc->buf = malloc(sc->buflen, M_DEVBUF, M_WAITOK | M_ZERO);
```

Pros: size can be chosen at attach time from a `sysctl` or tunable; can be resized if you are careful. Cons: allocation can fail (less relevant with `M_WAITOK`, more so with `M_NOWAIT`); the driver owns another free path.

For small buffers, static allocation inside the `softc` is the simplest choice, and the one Chapter 7 used implicitly by relying on Newbus to allocate the whole `softc`. Chapter 9 will use dynamic allocation because the buffer is large enough that putting it in the `softc` is slightly wasteful, and because the dynamic path is the pattern you will use repeatedly later in the book.

9.9.3 The `malloc(9)` Call

The kernel's `malloc(9)` takes three arguments: the size, a `malloc` type (which is a tag the kernel uses for accounting and debugging), and a flag word. A common shape:

```
sc->buf = malloc(sc->buflen, M_DEVBUF, M_WAITOK | M_ZERO);
```

`M_DEVBUF` is the generic “device buffer” malloc type, defined across the tree and appropriate for driver-private data that does not deserve a dedicated type. If your driver grows large enough to warrant its own tag, you can declare one with `MALLOC_DECLARE(M_MYFIRST)` and `MALLOC_DEFINE(M_MYFIRST, "myfirst", "myfirst driver data")`, and use `M_MYFIRST` instead. For now, `M_DEVBUF` is fine.

The flag bits most relevant at this stage:

- `M_WAITOK`: it is OK for the allocation to sleep waiting for memory. In attach context this is almost always the right choice.
- `M_NOWAIT`: do not sleep; return NULL if memory is tight. Needed when you are in a context that cannot sleep (interrupt handler, inside a non-sleepable lock).
- `M_ZERO`: zero the memory before returning. Pair with `M_WAITOK` or `M_NOWAIT` as appropriate.

A call with `M_WAITOK` without `M_NOWAIT` is guaranteed to return a valid pointer on FreeBSD 14.3. The kernel will sleep and possibly trigger reclamation if needed, but it will not return NULL in practice. Still, checking for NULL is defensive and costs nothing; we will do it.

9.9.4 The Matching `free(9)` Call

Every `malloc(9)` has a matching `free(9)`. The signature is:

```
free(sc->buf, M_DEVBUF);
```

The malloc type passed to `free` must match the one passed to `malloc` for the same pointer. Passing a different type would corrupt the kernel’s accounting and is one of the bugs `INVARIANTS`-enabled kernels catch at runtime.

Where to put the `free` depends on where the `malloc` was: attach allocates, detach frees. If attach fails partway, the error-unwind path frees whatever was allocated before the failure. We saw this pattern in Chapter 7; we will reuse it here.

9.9.5 Buffer Sizing

Picking a buffer size is a design choice. For a classroom driver, any small size works. A few guidelines:

- **Small** (a few hundred bytes to a few kilobytes): fine for demonstration. Easy to reason about. User workloads larger than the buffer will observe `ENOSPC` or short reads quickly; this is a feature for teaching, not a bug.
- **Page-sized** (4096 bytes): a common, sensible default. Memory allocation is page-aligned for free, and many tools treat 4 KiB as a natural unit.
- **Larger** (many kilobytes to a megabyte): appropriate for drivers that expect to buffer a lot of data. Remember that kernel memory is not infinite; a runaway driver that allocates a megabyte per open can destabilise the system.

For `myfirst` Stage 2 we will use a 4096-byte buffer. It is large enough that a reasonable test (a paragraph of text, a few integers) fits, and small enough that `ENOSPC` behaviour is easy to trigger from a shell.

9.9.6 Buffer Overflows

The single most common bug in a driver that manages its own buffer is writing past the end of the buffer. The bug is absolutely fatal in kernel space. A user-space program that overruns a buffer might corrupt its own heap; a kernel module that does the same may corrupt another subsystem's memory, and the crash (or worse, the silent misbehaviour) can show up far from the bug.

The defence is arithmetic discipline. Every time your code is about to write N bytes starting at offset 0 into a buffer of size S , verify that $0 + N \leq S$ before the write. In the Stage 3 handler above, the expression `towrite = MIN((size_t)uio->uio_resid, avail)` is exactly that check: `towrite` is clamped to `avail`, where `avail` is `sc->buflen - sc->bufused`. No way to exceed `sc->buflen`.

A related bug is signed-vs-unsigned confusion. `uio_resid` is `ssize_t`; `sc->bufused` is `size_t`. Mixing them carelessly can produce a negative value that wraps around when cast to `size_t`, with catastrophic results. The `MIN` macro and an explicit `(size_t)` cast are worth the tiny amount of noise they add to the code.

9.9.7 Locking Considerations

If your driver can be entered from more than one user context simultaneously, the buffer needs a lock. Two simultaneous writers could race on `bufused`; two simultaneous readers could race on read offsets; a writer and a reader could interleave their state updates in a way that corrupts both.

In `myfirst`, the `struct mtx mtx` field we have carried since Chapter 7 is the lock we will use. It is an ordinary `MTX_DEF` mutex, which means it can be held across a `uiomove` call (which may sleep on a page fault). We will hold it during every update to `bufused` and during the `uiomove` that is transferring bytes into or out of the shared buffer.

Part 3 goes much deeper into locking strategy. For now, the rule is: **protect any field that more than one handler might touch at the same time**. In Stage 3, that is `sc->buf` and `sc->bufused`. Your per-open `fh` is per-descriptor; it does not need the same lock, because two handlers cannot run for the same descriptor concurrently in the cases we will exercise.

9.9.8 A Preview of Circular Buffers

Chapter 10 builds a proper ring buffer: a fixed-size buffer where head and tail pointers chase each other around. It differs from the linear buffer we are using in Chapter 9 in two ways:

1. It does not need to be reset between uses. The pointers wrap; the buffer is reused in place.
2. It can support streaming at steady state. A linear buffer fills up and then refuses writes; a ring buffer maintains a moving window of recent data.

Stage 3 of this chapter does *not* implement a ring. It implements a linear buffer that `d_write` appends into and `d_read` drains from. When the buffer is full, `d_write` returns `ENOSPC`; when it is empty, `d_read` returns zero bytes. This is enough to get the I/O path right without the extra bookkeeping of a ring. Chapter 10 adds that bookkeeping on top of the same handler shapes.

9.9.9 A Note on Thread Safety of the Per-Descriptor fh

The `struct myfirst_fh` your driver allocates in `d_open` is per-descriptor. Two handlers for the same descriptor cannot execute concurrently in the scenarios this chapter exercises (the kernel serialises per-file operations through the file-descriptor machinery for the common cases), so the fields inside `fh` do not need their own lock. Two handlers for *different* descriptors do run concurrently, but they touch different `fh` structures.

This is a comforting invariant but not an absolute one. A driver that arranges to pass the `fh` pointer to a kernel thread that runs in parallel with the syscall must add its own synchronisation. We will not do that in this chapter; for now the `fh` is safe as long as you only touch it from inside the handler that was given the descriptor.

9.9.10 Kernel Helpers You Should Recognise

Before we move on, it is worth naming the small handful of helper macros and functions the chapter has been using. They are defined in standard FreeBSD headers, and beginners sometimes copy-paste code that uses them without knowing where they come from or what constraints apply.

`MIN(a, b)` and `MAX(a, b)` are available in kernel code through `<sys/libkern.h>`, which is pulled in transitively by `<sys/system.h>`. They evaluate each argument at most twice, so a `MIN(count++, limit)` is a bug: `count` increments twice. A well-written driver avoids side effects inside `MIN/MAX` arguments.

```
towrite = MIN((size_t)uio->uio_resid, avail);
```

The explicit `(size_t)` cast is part of the pattern, not a stylistic flourish. `uio_resid` is `ssize_t`, which is signed; `avail` is `size_t`, which is unsigned. Without the cast, the compiler picks one type for the comparison, and modern compilers warn when signed and unsigned meet in the same `MIN / MAX`. The cast makes the intent explicit: we have already checked that `uio_resid` is non-negative (the kernel guarantees it), so the cast is safe.

`howmany(x, d)`, defined in `<sys/param.h>`, computes $(x + d - 1) / d$. Use it when you need ceiling division. A driver that allocates pages to hold a byte-count often writes:

```
npages = howmany(buflen, PAGE_SIZE);
```

`rounddown(x, y)` and `roundup(x, y)` align `x` down or up to the nearest multiple of `y`. `roundup2` and `rounddown2` are faster variants that only work when `y` is a power of two. These are how drivers page-align buffers or block-align offsets.

`__DECONST(type, ptr)` casts away `const` without a compiler warning. It is the polite way to tell the compiler “I know this pointer is declared `const`, but I have verified that the function I am calling will not modify the data, so please stop complaining”. Used it around `zero_region` in `null.c`’s `zero_read`; we used it in Stage 1’s `myfirst_read`. Prefer it to a plain `(void *)` cast, because it signals intent.

`curthread` is an architecture-specific macro (resolved through a per-CPU register) that points at the currently executing thread. `uio->uio_td` usually equals `curthread` when the `uio` came from a syscall; the two are interchangeable in that context but the `uio`-carried value is more self-documenting.

`bootverbose` is an integer that is set to non-zero if the kernel was booted with `-v` or if the operator toggled it via `sysctl`. Guarding chatty log lines with `if (bootverbose)` is the FreeBSD idiom for debug logging that is visible on demand but silent by default.

Recognising these helpers when you encounter them in other drivers shortens the time it takes to read unfamiliar code. None of them are exotic; all of them are part of the standard vocabulary a kernel contributor is expected to read without looking up.

9.10 Error Handling and Edge Cases

A beginner driver that “works on the happy path” is a driver that will eventually crash the kernel. The interesting part of I/O handling is the parts that are not the happy path: zero-length reads, partial writes, bad user pointers, signals delivered mid-call, exhausted buffers, and several dozen variations of those. This section walks through the common cases and the `errno` values that go with them.

9.10.1 The Errno Values That Matter for I/O

FreeBSD has a large `errno` space. Only a handful are frequent in driver I/O paths; learning them well is more useful than skimming the whole list.

`0`: success. Return this when the transfer completed cleanly. The byte count is implicit in `uio_resid`.

`ENXIO` (“Device not configured”): the operation cannot proceed because the device is not in a usable state. Return this from `d_open`, `d_read`, or `d_write` if the `softc` is missing, `is_attached` is false, or the driver has been told to shut down. It is the idiomatic “the `cdev` exists but the backing device does not” error.

`EFAULT` (“Bad address”): a user pointer was invalid. You rarely return this directly; `uiomove(9)` returns it on your behalf when a `copyin/copyout` fails. Propagate it by returning whatever error `uiomove` produced.

`EINVAL` (“Invalid argument”): some parameter is nonsensical. For a read or write, this is usually an out-of-range offset (if your driver honours offsets) or a malformed request. Avoid using it as a catch-all.

`EAGAIN` (“Resource temporarily unavailable”): the operation would block, but `O_NONBLOCK` was set. For a `d_read` that has no data, this is the right answer in non-blocking mode. For a `d_write` that has no space, same story. We will handle this in Stage 3.

`EINTR` (“Interrupted system call”): a signal was delivered while the thread was blocked inside the driver. Your `d_read` may return `EINTR` if a sleep was interrupted by a signal. The kernel will then either retry the syscall transparently (depending on the `SA_RESTART` flag) or return to userland with `errno = EINTR`. We will see `EINTR` handling in Chapter 10; Chapter 9 does not block and so does not produce `EINTR`.

`EIO` (“Input/output error”): a catch-all for hardware errors. Use when your driver talks to real hardware and the hardware reports a failure. Rare in `myfirst`, which has no hardware.

`ENOSPC` (“No space left on device”): the driver’s buffer is full and cannot accept more data. The correct response for a write when there is no room. Stage 3 returns this.

`EPIPE` (“Broken pipe”): used by pipe-like drivers when the peer has closed. Not relevant for `myfirst`.

`ERANGE`, `Eoverflow`, `EMSGSIZE`: less common in character drivers; they show up when the kernel or the driver wants to say “the number you asked for is out of range”. We will not use them in this chapter.

9.10.2 End of File in a Read

By convention, a read that returns zero bytes (because `uiomove` did not move anything and your handler returned zero) is interpreted by the caller as end of file. Shells, `cat`, `head`, `tail`, `dd`, and most other base-system tools rely on this convention.

The implication for your `d_read`: when your driver has nothing more to deliver, return zero. Do not return an `errno`. `uio_resid` should still have its original value, because no bytes were moved.

In Stage 1 and Stage 2 of `myfirst`, EOF happens when the per-descriptor read offset has reached the buffer's length. `uiomove_frombuf` returns zero in that case naturally, so we do not need a special code path.

In Stage 3, where `d_read` drains a buffer that `d_write` has appended to, EOF behaviour is subtler: “no data right now” is not the same as “no more data ever”. We will report “no data right now” as a zero-byte read. A thoughtful user program might interpret that as EOF and stop; a less thoughtful one will loop and call `read(2)` again. Chapter 10 introduces the proper blocking-read or `poll(2)` strategies that allow a user program to wait for more data without spinning.

9.10.3 Zero-Length Reads and Writes

A zero-length request (`read(fd, buf, 0)` or `write(fd, buf, 0)`) is legal. It means “do nothing, but tell me whether you could have done something”. The kernel handles most of the dispatch for you: if `uio_resid` is zero at entry, any `uiomove` call is a no-op, and your handler returns zero. The caller sees a zero-byte transfer and no error.

Two subtle points. First, do not treat `uio_resid == 0` as an error condition. It is not. It is a legitimate request. Second, do not assume `uio_resid == 0` means end of file; it just means the caller asked for zero bytes. EOF is about the driver running out of data, not about the caller asking for none.

9.10.4 Short Transfers

A short read is a read that returns fewer bytes than were requested. A short write is a write that consumed fewer bytes than were offered. Both are legal and expected in UNIX I/O; well-written user programs handle them by looping.

Your driver is the authoritative decider of how much to transfer. The `uiomove` family of functions transfers at most `MIN(user_request, driver_offer)` bytes in a call. If your code calls `uiomove(buf, 128, uio)` and the user asked for 1024, the kernel transfers 128 and leaves 896 in `uio_resid`. The caller sees a 128-byte return from `read(2)`.

A badly-behaved user program that does not loop on short I/O will miss bytes. That is not your driver's problem; UNIX has been like this since 1971. A well-behaved driver is one that returns honest byte counts (through `uio_resid`) and predictable `errno` values, even when partial transfers happen.

9.10.5 Handling EFAULT From uiomove

When `uiomove(9)` returns a non-zero error, the most common value is `EFAULT`. By the time you see it, the kernel has already:

- Installed a fault handler around the copy.
- Observed the fault.
- Unwound the partial copy.
- Returned EFAULT to uiomove's caller.

Your handler has two options for how to respond:

1. **Propagate the error.** Return EFAULT (or whatever errno was returned) from `d_read / d_write`. This is the simplest and almost always correct.
2. **Adjust driver state and return success.** If some bytes moved before the fault, `uio_resid` may have already decreased. The kernel will report that partial success to user space. You may want to update any driver-side counter that reflects how far the transfer got.

In practice, option 1 is the universal answer unless you have a specific reason to do more. Option 2 adds complexity that is rarely worth it.

9.10.6 Defensive Programming for User Input

Every byte a user writes into your device is untrusted. That sounds dramatic; it is also literally true. A kernel module that parses a user write as a structure and dereferences a pointer from that structure is a kernel module with a trivial arbitrary-memory-write vulnerability.

The rule of thumb: **treat the bytes in your buffer as arbitrary data, not as a typed structure, unless you have deliberately chosen a wire format that you validate at every boundary.** For `myfirst` this is easy, because we never interpret the bytes; they are payload. For drivers that expose a structured write interface (for example, a driver that lets users configure behaviour through writes), the defensive path is to:

- Validate the length of the write against the expected message size.
- Copy the bytes into a kernel-space structure (not a user pointer).
- Validate every field of that structure before acting on it.
- Never store a user pointer inside your driver for later use.

These rules are easier to follow than they sound, but they are easy to break without noticing. Chapter 25 revisits them when we look at `ioctl` design. For now, the bar is low: your `myfirst` driver should copy bytes through `uiomove`, not interpret them.

9.10.7 Logging Errors vs Silent Failures

When a handler returns an `errno`, the error propagates to user space as the `errno` value of the failed syscall. Most user programs will see it there and report it. Some will swallow it.

For driver development, it helps to also log significant errors to `dmesg`. `device_printf(9)` is the right tool, because it tags each line with the Newbus device name so you can tell which instance produced the message. An example from Stage 3:

```
if (avail == 0) {
    mtx_unlock(&sc->mtx);
    if (bootverbose)
```

```

        device_printf(sc->dev, "write rejected: buffer full\n");
    return (ENOSPC);
}

```

The `if (bootverbose)` guard is a common FreeBSD idiom for chatty logging: it only prints if the kernel was booted with the `-v` flag or if the `bootverbose` sysctl was set, which keeps production logs quiet while still giving developers a way to see detail.

Do not log every error on every call; that produces log spam, which makes real problems harder to find. Log the first occurrence of a condition, or log periodically, or log only under `bootverbose`. The choice depends on the driver. For `myfirst`, a single log per transition (buffer-empty, buffer-full) is plenty.

9.10.8 Predictability and User-Friendliness

A beginner writing a driver often focuses on making the good path fast. A more experienced driver author focuses on making the error paths predictable. The difference is this: when an operator runs your driver and something breaks, the `errno` value, the log message, and the user-space reaction need to add up to a clear story. If `read(2)` returns `-1` with `errno = EIO` and the logs are silent, the operator has nowhere to start. If the logs say “`myfirst0: read failed, device detached`” and the user gets `ENXIO`, the story tells itself.

Aim for that. Return the right `errno`. Log the underlying reason once. Make partial transfers honest. Never silently discard data.

9.10.9 A Short Table of Conventions

Situation in <code>d_read</code>	Return
No data to deliver, more might arrive later	0 with <code>uio_resid</code> unchanged
No data, never will be any more (EOF)	0 with <code>uio_resid</code> unchanged
Some data delivered, some not	0 with <code>uio_resid</code> reflecting remainder
Full delivery	0 with <code>uio_resid = 0</code>
User pointer invalid	<code>EFAULT</code> (from <code>uiomove</code>)
Device not ready / detaching	<code>ENXIO</code>
Non-blocking, would block	<code>EAGAIN</code>
Hardware error	<code>EIO</code>

Situation in <code>d_write</code>	Return
Full acceptance	0 with <code>uio_resid = 0</code>
Partial acceptance	0 with <code>uio_resid</code> reflecting remainder
No room, would block	EAGAIN (non-blocking) or sleep (blocking)
No room, permanent	ENOSPC
Invalid pointer	EFAULT (from <code>uiomove</code>)
Device not ready	ENXIO
Hardware error	EIO

Both tables are short on purpose. Most drivers use only four or five `errno` values in total. The cleaner your error story, the better your driver is to work with.

9.11 Evolving Your Driver: The Three Stages

With the theory in place, we turn to code. This section walks through three stages of `myfirst`, each one small, each one a complete driver that loads, runs, and exercises a specific I/O pattern.

The stages are designed to build on one another:

- **Stage 1** adds a read path that serves a fixed kernel-space message. This is `myfirst_read` at its simplest.
- **Stage 2** adds a write path that deposits user data into a kernel buffer, and a read path that reads from the same buffer. The buffer is sized at attach time and does not wrap.
- **Stage 3** turns Stage 2 into a first-in / first-out buffer, so that writes append and reads drain, and the driver can serve a continuous (if finite) stream.

All three stages start from the Chapter 8 stage 2 source. The build system (`Makefile`) does not change. The attach and detach handlers grow slightly at each stage. The shape of `cdevsw`, the Newbus methods, the per-open `fh` plumbing, and the `sysctl` tree stay the same. You will spend most of your time looking at `d_read` and `d_write`.

9.11.1 Stage 1: A Static-Message Reader

The Stage 1 driver holds a fixed message in kernel memory and serves it to readers. `d_read` uses `uiomove_frombuf(9)` to deliver the message. `d_write` stays a stub: it returns success without consuming any bytes. This stage is the bridge from the Chapter 8 stub to a real reader; it introduces `uiomove_frombuf` in the smallest possible context.

Add a pair of fields to the `softc` to hold the message and its length, and a per-descriptor offset to the `fh`:

```
struct myfirst_softc {
    /* ...existing Chapter 8 fields... */

    const char *message;
    size_t      message_len;
};

struct myfirst_fh {
    struct myfirst_softc *sc;
    uint64_t      reads;
    uint64_t      writes;
    off_t         read_off;
};
```

In `myfirst_attach`, initialise the message:

```
static const char myfirst_message[] =
    "Hello from myfirst.\n"
    "This is your first real read path.\n"
    "Chapter 9, Stage 1.\n";

sc->message = myfirst_message;
sc->message_len = sizeof(myfirst_message) - 1;
```

Note the `- 1`: we do not want to serve the terminating NUL byte to user space. Text files do not carry a NUL at the end, and neither should a device that behaves like one.

The new `myfirst_read`:

```
static int
myfirst_read(struct cdev *dev, struct uio *uio, int ioflag)
{
    struct myfirst_softc *sc = dev->si_drv1;
    struct myfirst_fh *fh;
    off_t before;
    int error;

    error = devfs_get_cdevpriv((void **)&fh);
    if (error != 0)
        return (error);

    if (sc == NULL || !sc->is_attached)
        return (ENXIO);
```

```

    before = uio->uio_offset;
    error = uiomove_frombuf(__DECONST(void *, sc->message),
        sc->message_len, uio);
    if (error == 0)
        fh->reads += (uio->uio_offset - before);
    fh->read_off = uio->uio_offset;
    return (error);
}

```

Two details are worth stopping on.

First, `uio->uio_offset` is the per-descriptor position in the stream. The kernel maintains it across calls, advancing it as `uiomove_frombuf` moves bytes. The first `read(2)` on a freshly opened descriptor starts at offset zero; each subsequent `read(2)` starts where the previous one ended. When the offset reaches `sc->message_len`, `uiomove_frombuf` returns zero without moving any bytes, and the caller sees EOF.

Second, `before` captures `uio->uio_offset` at entry so we can compute how many bytes moved. After `uiomove_frombuf` returns, the difference is the transfer size, and we add it to the per-descriptor `reads` counter. This is where the `fh->reads` field from Chapter 8 finally earns its keep.

The `__DECONST` cast is a FreeBSD idiom for casting away `const`. `uiomove_frombuf` takes a non-`const` `void *` because it is prepared to move in either direction, but in this context we know the direction is kernel-to-user (a read), so we know the kernel buffer will not be modified. Stripping the `const` here is safe; using a plain `(void *)` cast would work as well but is less self-documenting.

`myfirst_write` stays as Chapter 8 left it for Stage 1:

```

static int
myfirst_write(struct cdev *dev, struct uio *uio, int ioflag)
{
    struct myfirst_fh *fh;
    int error;

    error = devfs_get_cdevpriv((void **)&fh);
    if (error != 0)
        return (error);

    (void)fh;
    uio->uio_resid = 0;
    return (0);
}

```

Writes are accepted and discarded, the `/dev/null` shape. Stage 2 will change this.

Build and load. A quick smoke test from userland:


```
% cat /dev/myfirst/0
Hello from myfirst.
This is your first real read path.
Chapter 9, Stage 1.
%
```

A second read from the same descriptor returns EOF, because the offset is already past the end of the message:

```
% cat /dev/myfirst/0 /dev/myfirst/0
Hello from myfirst.
This is your first real read path.
Chapter 9, Stage 1.
Hello from myfirst.
This is your first real read path.
Chapter 9, Stage 1.
```

Wait: `cat` reads the message twice. That is because `cat` opens the file twice (once per argument) and each open gets a fresh descriptor with its own `uio_offset`. If you want to verify that two opens really do see independent offsets, open the device from a small C program and read more than once from the same descriptor:

```
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>

int
main(void)
{
    int fd = open("/dev/myfirst/0", O_RDONLY);
    if (fd < 0) { perror("open"); return 1; }
    char buf[64];
    ssize_t n;
    while ((n = read(fd, buf, sizeof(buf))) > 0) {
        fwrite(buf, 1, n, stdout);
    }
    close(fd);
    return 0;
}
```

The first `read(2)` returns the message; the second returns zero (EOF); the program exits. This confirms that `uio_offset` is being maintained per descriptor.

Stage 1 is intentionally short. It introduces three ideas (the `uiomove_frombuf` helper, per-descriptor offsets, the `__DECONST` idiom) without overloading the reader. The rest of the chapter builds on this.

9.11.2 Stage 2: A Write-Once / Read-Many Buffer

Stage 2 extends the driver to accept writes. The driver allocates a kernel buffer at attach, writes deposit into it, reads deliver from it. There is no wraparound: once the buffer fills, further writes return ENOSPC. Reads see whatever has been written so far, starting from their own per-descriptor offset.

The shape of `myfirst_softc` grows by a few fields:

```
struct myfirst_softc {
    /* ...existing Chapter 8 fields... */

    char    *buf;
    size_t   buflen;
    size_t   bufused;

    uint64_t bytes_read;
    uint64_t bytes_written;
};
```

`buf` is the pointer returned by `malloc(9)`. `buflen` is its size, a compile-time constant for simplicity; you can make it tunable later. `bufused` is the high-water mark: the number of bytes that have been written so far.

Two new sysctl nodes for observability:

```
SYSCALL_ADD_U64(&sc->sysctl_ctx, SYSCALL_CHILDREN(sc->sysctl_tree),
    OID_AUTO, "bytes_written", CTLFLAG_RD,
    &sc->bytes_written, 0, "Total bytes written into the buffer");

SYSCALL_ADD_UINT(&sc->sysctl_ctx, SYSCALL_CHILDREN(sc->sysctl_tree),
    OID_AUTO, "bufused", CTLFLAG_RD,
    &sc->bufused, 0, "Current byte count in the buffer");
```

`bufused` is a `size_t`, and the sysctl macro for unsigned integer is `SYSCALL_ADD_UINT` on 32-bit platforms or `SYSCALL_ADD_U64` on 64-bit platforms. Since this driver targets FreeBSD 14.3 on amd64 in the typical lab, `SYSCALL_ADD_UINT` is fine; the field will be presented as an unsigned `int` even though the internal type is `size_t`. If you target arm64 or another 64-bit platform, use `SYSCALL_ADD_U64` and cast accordingly.

Allocate the buffer in `attach`:

```
#define MYFIRST_BUFSIZE 4096

sc->buflen = MYFIRST_BUFSIZE;
sc->buf = malloc(sc->buflen, M_DEVBUF, M_WAITOK | M_ZERO);
if (sc->buf == NULL) {
    error = ENOMEM;
    goto fail_mtx;
```

```
}  
sc->bufused = 0;
```

Free it in detach:

```
if (sc->buf != NULL) {  
    free(sc->buf, M_DEVBUF);  
    sc->buf = NULL;  
}
```

Adjust the error-unwind in attach to include the buffer free:

```
fail_dev:  
    if (sc->cdev_alias != NULL) {  
        destroy_dev(sc->cdev_alias);  
        sc->cdev_alias = NULL;  
    }  
    destroy_dev(sc->cdev);  
    sysctl_ctx_free(&sc->sysctl_ctx);  
    free(sc->buf, M_DEVBUF);  
    sc->buf = NULL;  
fail_mtx:  
    mtx_destroy(&sc->mtx);  
    sc->is_attached = 0;  
    return (error);
```

Now the read handler:

```
static int  
myfirst_read(struct cdev *dev, struct uio *uio, int ioflag)  
{  
    struct myfirst_softc *sc = dev->si_drv1;  
    struct myfirst_fh *fh;  
    off_t before;  
    size_t have;  
    int error;  
  
    error = devfs_get_cdevpriv((void **)&fh);  
    if (error != 0)  
        return (error);  
    if (sc == NULL || !sc->is_attached)  
        return (ENXIO);
```

```

    mtx_lock(&sc->mtx);
    have = sc->bufused;
    before = uio->uio_offset;
    error = uiomove_frombuf(sc->buf, have, uio);
    if (error == 0) {
        sc->bytes_read += (uio->uio_offset - before);
        fh->reads += (uio->uio_offset - before);
    }
    fh->read_off = uio->uio_offset;
    mtx_unlock(&sc->mtx);
    return (error);
}

```

The read handler takes the mutex to read `bufused` consistently, then calls `uiomove_frombuf` with the current high-water mark as the effective buffer size. A reader that runs before any writes happen will see `have = 0` and `uiomove_frombuf` will return zero, which the caller interprets as EOF. A reader that runs after some writes will see the current `bufused` and receive up to that many bytes.

The write handler:

```

static int
myfirst_write(struct cdev *dev, struct uio *uio, int ioflag)
{
    struct myfirst_softc *sc = dev->si_drv1;
    struct myfirst_fh *fh;
    size_t avail, towrite;
    int error;

    error = devfs_get_cdevpriv((void **)&fh);
    if (error != 0)
        return (error);
    if (sc == NULL || !sc->is_attached)
        return (ENXIO);

    mtx_lock(&sc->mtx);
    avail = sc->buflen - sc->bufused;
    if (avail == 0) {
        mtx_unlock(&sc->mtx);
        return (ENOSPC);
    }
    towrite = MIN((size_t)uio->uio_resid, avail);
    error = uiomove(sc->buf + sc->bufused, towrite, uio);
    if (error == 0) {
        sc->bufused += towrite;
        sc->bytes_written += towrite;
    }
}

```

```

        fh->writes += towrite;
    }
    mtx_unlock(&sc->mtx);
    return (error);
}

```

Notice the clamp: `towrite = MIN(uio->uio_resid, avail)`. If the user asked to write 8 KiB and we have 512 bytes of room, we accept 512 bytes and let the kernel report a short write of 512 back to user space. A well-behaved caller will loop with the remaining bytes; a less well-behaved caller will lose the surplus. That is the caller's responsibility; the driver has done its part honestly.

Smoke-test from userland:

```

% sudo kldload ./myfirst.ko
% echo "hello" | sudo tee /dev/myfirst/0 > /dev/null
% cat /dev/myfirst/0
hello
% echo "more" | sudo tee -a /dev/myfirst/0 > /dev/null
% cat /dev/myfirst/0
hello
more
% sysctl dev.myfirst.0.stats.bufused
dev.myfirst.0.stats.bufused: 11
%

```

The buffer grew by 6 bytes for "hello\n", then by 5 more for "more\n", yielding 11 bytes. `cat` reads all 11 bytes back. A second `cat` from a fresh open starts at offset zero and reads them again.

What happens if we write more than the buffer can hold?

```

% dd if=/dev/zero bs=1024 count=8 | sudo tee /dev/myfirst/0 > /dev/null
dd: stdout: No space left on device
tee: /dev/myfirst/0: No space left on device
8+0 records in
7+0 records out

```

`dd` wrote 7 blocks of 1024 bytes before the 8th one failed. `tee` reports the error. The driver accepted up to its limit and then returned `ENOSPC` cleanly. The kernel carried the `errno` value back to user space.

9.11.3 Stage 3: A First-In / First-Out Echo Driver

Stage 3 turns the buffer into a FIFO. Writes append to the tail. Reads drain from the head. When the buffer is empty, reads return zero bytes (EOF-on-empty). When the buffer is full, writes return `ENOSPC`.

The buffer remains linear: no wrap-around. After a read that drains all the data, bufused is zero, and the next write starts again at offset zero in `sc->buf`. This keeps the bookkeeping minimal and focuses the stage on the I/O direction change rather than ring-buffer mechanics.

The `softc` gains one more field:

```
struct myfirst_softc {
    /* ...existing fields... */

    size_t  bufhead;  /* index of next byte to read */
    size_t  bufused;  /* bytes in the buffer, from bufhead onward */

    /* ...remaining fields... */
};
```

`bufhead` is the offset of the first byte still to be read. `bufused` is the number of valid bytes starting at `bufhead`. The invariant `bufhead + bufused <= buflen` always holds.

Reset both in `attach`:

```
sc->bufhead = 0;
sc->bufused = 0;
```

New read handler:

```
static int
myfirst_read(struct cdev *dev, struct uio *uio, int ioflag)
{
    struct myfirst_softc *sc = dev->si_drv1;
    struct myfirst_fh *fh;
    size_t toread;
    int error;

    error = devfs_get_cdevpriv((void **)&fh);
    if (error != 0)
        return (error);
    if (sc == NULL || !sc->is_attached)
        return (ENXIO);

    mtx_lock(&sc->mtx);
    if (sc->bufused == 0) {
        mtx_unlock(&sc->mtx);
        return (0); /* EOF-on-empty */
    }
    toread = MIN((size_t)uio->uio_resid, sc->bufused);
```

```

    error = uiomove(sc->buf + sc->bufhead, toread, uio);
    if (error == 0) {
        sc->bufhead += toread;
        sc->bufused -= toread;
        sc->bytes_read += toread;
        fh->reads += toread;
        if (sc->bufused == 0)
            sc->bufhead = 0;
    }
    mtx_unlock(&sc->mtx);
    return (error);
}

```

A few details differ from Stage 2. The read no longer honours `uio->uio_offset`; the per-descriptor offset is meaningless for a FIFO where every descriptor sees the same stream and the stream disappears as it is consumed. When `bufused` reaches zero, we reset `bufhead` to zero, which keeps the next write aligned at the beginning of the buffer and avoids pushing data toward the end.

This “collapse on empty” trick is not a ring buffer, but it is close enough for a pedagogical FIFO. The extra re-align step is $O(1)$; it costs almost nothing.

New write handler (mostly unchanged from Stage 2, but note where it appends):

```

static int
myfirst_write(struct cdev *dev, struct uio *uio, int ioflag)
{
    struct myfirst_softc *sc = dev->si_drv1;
    struct myfirst_fh *fh;
    size_t avail, tail, towrite;
    int error;

    error = devfs_get_cdevpriv((void **)&fh);
    if (error != 0)
        return (error);
    if (sc == NULL || !sc->is_attached)
        return (ENXIO);

    mtx_lock(&sc->mtx);
    tail = sc->bufhead + sc->bufused;
    avail = sc->buflen - tail;
    if (avail == 0) {
        mtx_unlock(&sc->mtx);
        return (ENOSPC);
    }
    towrite = MIN((size_t)uio->uio_resid, avail);
}

```

```
error = uiomove(sc->buf + tail, towrite, uio);
if (error == 0) {
    sc->bufused += towrite;
    sc->bytes_written += towrite;
    fh->writes += towrite;
}
mtx_unlock(&sc->mtx);
return (error);
}
```

The write appends at `sc->bufhead + sc->bufused`, not at `sc->bufused` alone, because the valid data slice has moved as reads drained it.

Smoke-test:

```
% echo "one" | sudo tee /dev/myfirst/0 > /dev/null
% echo "two" | sudo tee -a /dev/myfirst/0 > /dev/null
% cat /dev/myfirst/0
one
two
% cat /dev/myfirst/0
%
```

After the first cat, the buffer is empty. The second cat sees no data and exits immediately.

This is the Stage 3 shape. The driver is a small, honest, in-memory FIFO. Users can push bytes in, pull them out, and observe the counters from `sysctl`. That is real I/O, and it is the waypoint Chapter 10 builds from.

9.12 Tracing a read(2) From User Space to Your Handler

Before you start working through the labs, take a step-by-step look at exactly what happens when a user program calls `read(2)` on one of your nodes. Understanding this path is one of those things that changes how you read driver code. Every handler you see in the tree is sitting at the bottom of the call chain described below; once you recognise the chain, every handler starts to look familiar.

9.12.1 Step 1: The User Program Calls read(2)

The C library's `read` wrapper is a thin translation of the call into a system-call trap: it places the file descriptor, the buffer pointer, and the count into the appropriate registers and executes the trap instruction for the current architecture. Control transfers to the kernel.

This part has nothing to do with drivers. It is the same for every syscall. What matters is that the kernel is now executing on behalf of the user process, in the kernel's address space, with the user's registers saved and the process's credentials visible through `curthread->td_ucred`.

9.12.2 Step 2: The Kernel Looks Up the File Descriptor

The kernel calls `sys_read(2)` (in `/usr/src/sys/kern/sys_generic.c`), which validates the arguments, looks up the file descriptor in the calling process's file table, and acquires a reference on the resulting `struct file`.

If the descriptor is not open, the call fails here with `EBADF`. If the descriptor is open but is not readable (for instance, the user opened the device with `O_WRONLY`), the call fails with `EBADF` as well. The driver is not involved; `sys_read` enforces the access mode.

9.12.3 Step 3: The Generic File-Operation Vector Dispatches

The `struct file` has a file-type tag (`f_type`) and a file-operations vector (`f_ops`). For a regular file the vector dispatches to the VFS layer; for a socket it dispatches to sockets; for a device opened through `devfs`, it dispatches to `vn_read`, which in turn calls the vnode operation `VOP_READ` on the vnode behind the file.

This may sound like indirection for its own sake. It is actually how the kernel keeps the rest of the syscall path identical for every kind of file. Drivers do not need to know about this layer; `devfs` and VFS hand the call to your handler eventually.

9.12.4 Step 4: VFS Calls Into `devfs`

The vnode's filesystem ops point to `devfs`'s implementation of the vnode interface (`devfs_vnops`). `VOP_READ` on a `devfs` vnode calls `devfs_read_f`, which looks at the `cdev` behind the vnode, acquires a thread-count reference on it (incrementing `si_threadcount`), and calls `cdevsw->d_read`. That is your function.

Two details from this step carry implications for your driver.

First, **the `si_threadcount` increment is what `destroy_dev(9)` uses to know your handler is active**. When a module unloads and `destroy_dev` runs, it waits until every current invocation of every handler returns. The reference is incremented before your `d_read` is called and released after it returns. The mechanism is why your driver can be safely unloaded while a user is in the middle of `read(2)`.

Second, **the call is synchronous from the VFS layer's point of view**. VFS calls your handler, waits for it to return, and then propagates the result. You do not need to do anything special to participate in this synchronisation; just return from your handler when you are done.

9.12.5 Step 5: Your `d_read` Handler Runs

This is where we have been all chapter. The handler:

- Receives a `struct cdev *dev` (the node being read), a `struct uio *uio` (the I/O description), and an `int ioflag` (flags from the file-table entry).
- Retrieves per-open state via `devfs_get_cdevpriv(9)`.
- Verifies liveness.
- Transfers bytes through `uiomove(9)`.
- Returns zero or an `errno`.

Nothing about this step should be mysterious by now.

9.12.6 Step 6: The Kernel Unwinds and Reports

`devfs_read_f` sees your return value. If zero, it computes the byte count from the decrease in `uio->uio_resid` and returns that count. If non-zero, it converts the `errno` into the syscall's error return. VFS's `vn_read` passes the result upward to `sys_read`. `sys_read` writes the result into the return-value register.

Control transfers back to user space. The C library's `read` wrapper examines the result: a positive value is returned as the return value of `read(2)`; a negative value sets `errno` and returns `-1`.

The user program sees the integer it expected, and its control flow continues.

9.12.7 Step 7: The Reference Counts Unwind

On the way out, `devfs_read_f` releases the thread-count reference on the `cdev`. If `destroy_dev(9)` had been waiting for `si_threadcount` to reach zero, it may now proceed with the tear-down.

This is why the whole chain is structured as carefully as it is. Every reference is paired; every increment has a matching decrement; every piece of state the handler touches is either owned by the handler, owned by the `softc`, or owned by the per-open `fh`. If any of those invariants breaks, unload becomes unsafe.

9.12.8 Why This Trace Matters to You

Three takeaways.

The first: the mechanism above is why your handler does not need to do anything exotic to coexist with module unload. Provided you return from `d_read` in finite time, the kernel will let your driver unload cleanly. This is part of why Chapter 9 keeps all reads non-blocking at the driver level.

The second: every layer between `read(2)` and your handler is set up by the kernel before your code runs. The user's buffer is valid (or `uiomove` will report `EFAULT`), the `cdev` is alive (or `devfs` would have refused the call), the access mode is compatible with the descriptor (or `sys_read` would have refused), and the process's credentials are the current thread's. You can focus on your driver's job and trust the layers.

The third: when you read an unfamiliar driver in the tree and its `d_read` looks weird, you can walk the chain in reverse. Who called this handler? What state did they prepare? What invariants does my handler promise on return? The chain tells you. The answers are usually the same as they are for `myfirst`.

9.12.9 The Mirror: Tracing a `write(2)`

A write follows the same kind of chain, mirrored. A full seven-step breakdown would be mostly a restatement of the read trace with words substituted, so the paragraph below is deliberately compressed.

The user calls `write(fd, buf, 1024)`. The C library traps into the kernel. `sys_write(2)` in `/usr/src/sys/kern/sys_generic.c` validates arguments, looks up the descriptor, and acquires a reference on its `struct file`. The file-ops vector dispatches to `vn_write`, which calls `VOP_WRITE` on the `devfs` vnode. `devfs_write_f` in `/usr/src/sys/fs/devfs/devfs_vnops.c` acquires the thread-count reference on the `cdev`, composes the `ioflag` from `fp->f_flag`, and calls `cdevsw->d_write` with the `uio` describing the caller's buffer.

Your `d_write` handler runs. It retrieves per-open state via `devfs_get_cdevpriv(9)`, checks liveness, takes whatever lock the driver needs around the buffer, clamps the transfer length to whatever space is available, and calls `uiomove(9)` to copy bytes from user space into the kernel buffer. On success, the handler updates its bookkeeping and returns zero. `devfs_write_f` releases the thread-count reference. `vn_write` unwinds through `sys_write`, which computes the byte count from the decrease in `uio_resid` and returns it. The user sees the return value of `write(2)`.

Three things differ from the read chain in substantive ways.

First, the kernel runs copyin inside uiomove instead of copyout. Same mechanism, opposite direction. The fault handling is identical: a bad user pointer returns `EFAULT`, a short copy leaves `uio_resid` consistent with whatever did transfer, and the handler just propagates the error code.

Second, `ioflag` carries `IO_NDELAY` in the same way, but the driver's interpretation is different. On a read, non-blocking means “return `EAGAIN` if there is no data”. On a write, non-blocking means “return `EAGAIN` if there is no space”. Symmetric conditions, symmetric `errno` values.

Third, the `atime` / `mtime` updates are direction-specific. `devfs_read_f` stamps `si_atime` if bytes moved; `devfs_write_f` stamps `si_mtime` (and `si_ctime` in some paths) if bytes moved. These are what `stat(2)` on the node reports, and why `ls -lu /dev/myfirst/0` shows different timestamps for reads versus writes. Your driver does not manage these fields; `devfs` does.

Once you recognise the read and write traces as mirror images, you have internalised most of the character-device dispatch path. Every chapter from here on will add hooks (a `d_poll`, a `d_kqfilter`, a `d_ioctl`, an `mmap` path) that sit on the same chain at slightly different slots. The chain itself stays constant.

9.13 Practical Workflow: Testing Your Driver From the Shell

The base-system tools are your first and best test harness. This section is a short field guide to using them well on a driver you are developing. None of the commands below are new to you, but using them for driver work has a rhythm worth learning explicitly.

9.13.1 `cat(1)`: the first check

`cat` reads from its arguments and writes to standard output. For a driver that serves a static message or a drained buffer, `cat` is the fastest way to see what the read path produces:

```
% cat /dev/myfirst/0
```

If the output is what you expect, the read path is alive. If it is empty, either your driver has nothing to deliver (check `sysctl dev.myfirst.0.stats.bufused`) or your handler is returning EOF on the first call. If the output is garbled, either your buffer is uninitialised or you are handing out bytes past `bufused`.

`cat` opens its argument once and reads from it until EOF. Every `read(2)` is a separate call into your `d_read`. Use `truss(1)` to see how many calls `cat` makes:

```
% truss cat /dev/myfirst/0 2>&1 | grep read
```

The output shows each `read(2)` with its arguments and return value. If you expected one read and see three, that tells you about your buffer sizing; if you expected three reads and see one, your handler delivered all the data in a single call.

9.13.2 `echo(1)` and `printf(1)`: simple writes

`echo` is the quickest way to get a known string into your driver's write path:

```
% echo "hello" | sudo tee /dev/myfirst/0 > /dev/null
```

Two things to notice. First, `echo` appends a newline by default; the string you sent is six bytes, not five. Use `echo -n` to suppress the newline when that matters. Second, the `tee` invocation is there to solve a permission problem: shell redirection (`>`) runs with the user's privileges, so a `sudo echo > /dev/myfirst/0` fails to open the node. Piping through `tee`, which runs under `sudo`, sidesteps that.

`printf` gives you more control:

```
% printf 'abc' | sudo tee /dev/myfirst/0 > /dev/null
```

Three bytes, no newline. Use `printf '\x41\x42\x43'` for binary patterns.

9.13.3 `dd(1)`: the precision tool

For any test that needs a specific byte count or a specific block size, `dd` is the right tool. `dd` is also one of the only base-system tools that reports short reads and short writes in its summary, which makes it uniquely useful for testing driver behaviour:

```
% sudo dd if=/dev/urandom of=/dev/myfirst/0 bs=128 count=4
4+0 records in
4+0 records out
512 bytes transferred in 0.001234 secs (415000 bytes/sec)
```

The `X+Y records in / X+Y records out` counters have a precise meaning: `X` is the number of full-block transfers, `Y` is the number of short transfers. A line reading `0+4 records out` means every block was accepted only partially. That is a driver telling you something.

`dd` also lets you read with a known block size:

```
% sudo dd if=/dev/myfirst/0 of=/tmp/dump bs=64 count=1
```

This issues exactly one `read(2)` for 64 bytes. Your handler sees `uio_resid = 64`; you respond with whatever you have; the result is what `dd` writes to `/tmp/dump`.

The `iflag=fullblock` flag tells `dd` to loop on short reads until it has filled the requested block. Useful when you want to soak all of the driver's output without losing bytes to the short-read default.

9.13.4 `od(1)` and `hexdump(1)`: byte-level inspection

For driver testing, `od` and `hexdump` let you see the exact bytes your driver emitted:

```
% sudo dd if=/dev/myfirst/0 bs=32 count=1 | od -An -tx1z
    68 65 6c 6c 6f 0a                >hello.<
```

The `-An` flag suppresses address printing. `-tx1z` shows bytes in hex and ASCII. If the expected output is text, you see it on the right; if it is binary, you see the hex on the left.

These tools become essential when a read produces unexpected bytes. “It looks weird” and “I can see every byte in hex” are very different debugging states.

9.13.5 `sysctl(8)` and `dmesg(8)`: the kernel’s voice

Your driver publishes counters through `sysctl` and lifecycle events through `dmesg`. Both are worth checking during every test:

```
% sysctl dev.myfirst.0
% dmesg | tail -20
```

The `sysctl` output is your view into the driver’s state right now. `dmesg` is your view into the driver’s history since boot (or since the ring buffer wrapped).

A useful habit: after every test, run both. If the numbers do not match your expectation, you have narrowed down the bug quickly.

9.13.6 `fstat(1)`: who has the descriptor open?

When your driver refuses to unload (“module busy”), the question is “who has `/dev/myfirst/0` open right now?”. `fstat(1)` answers it:

```
% fstat -p $(pgrep cat) /dev/myfirst/0
```

USER	CMD	PID	FD	MOUNT	INUM	MODE	SZ DV	R/W	NAME
ebrandi	cat	1234	3	/dev	0	crw-rw----	myfirst/0	r	/dev/myfirst/0

Alternatively, `fuser(8)`:

```
% sudo fuser /dev/myfirst/0
/dev/myfirst/0:      1234
```

Either tool names the processes holding the descriptor. Kill the culprit (carefully; do not kill anything you did not start) and the module will unload.

9.13.7 truss(1) and ktrace(1): watching the syscalls

For a user program whose interaction with your driver you want to inspect, `truss` shows every syscall and its return value:

```
% truss ./rw_myfirst
open("/dev/myfirst/0",O_WRONLY,0666)      = 3 (0x3)
write(3,"round-trip test payload\n",24)    = 24 (0x18)
close(3)                                   = 0 (0x0)
...
```

`ktrace` records to a file that `kdump` prints later; it is the right tool when you want to capture a trace of a long-running program.

These two tools are not driver-specific, but they are how you confirm from the outside that your driver is producing the results a user program will see.

9.13.8 A Suggested Test Rhythm

For each stage of the chapter, try this loop:

1. Build and load.
2. `cat` to produce initial output, confirm by eye.
3. `sysctl dev.myfirst.0` to see counters match.
4. `dmesg | tail` to see lifecycle events.
5. Write something with `echo` or `dd`.
6. Read it back.
7. Repeat with a larger size, a boundary size, and a pathological size.
8. Unload.

After a couple of iterations this becomes automatic and fast. It is the kind of rhythm that turns driver development from a slog into a routine.

9.13.9 A Concrete truss Walkthrough

Running a userland program under `truss(1)` is one of the fastest ways to see exactly what syscalls it makes to your driver and what return values the kernel produces. Here is a typical session with the Stage 3 driver loaded and empty:

```
% truss ./rw_myfirst rt 2>&1
open("/dev/myfirst/0",O_WRONLY,00)        = 3 (0x3)
write(3,"round-trip test payload, 24b\n",29) = 29 (0x1d)
close(3)                                   = 0 (0x0)
open("/dev/myfirst/0",O_RDONLY,00)         = 3 (0x3)
read(3,"round-trip test payload, 24b\n",255) = 29 (0x1d)
```

```
close(3)                = 0 (0x0)
exit(0x0)
```

A few things are worth pausing on. Each line shows a single syscall, its arguments, and its return value in both decimal and hex. The `write` call received 29 bytes and the driver accepted all 29 (the return value matches the request length). The `read` call received a buffer of 255 bytes of room and the driver produced 29 bytes of content; a short read, which the user program explicitly accepts. Both open calls returned 3, because file descriptors 0, 1, and 2 are standard streams and the first free descriptor is 3.

If you force a short write by limiting the driver, `truss` will show it plainly:

```
% truss ./write_big 2>&1 | head
open("/dev/myfirst/0", O_WRONLY, 00)      = 3 (0x3)
write(3, "<8192 bytes of data>", 8192)     = 4096 (0x1000)
write(3, "<4096 bytes of data>", 4096)     ERR#28 'No space left on device'
close(3)                                  = 0 (0x0)
```

The first write requested 8192 bytes and was accepted for 4096. The second write had nothing to say because the buffer is full; the driver returned `ENOSPC`, which `truss` rendered as `ERR#28 'No space left on device'`. This is the view from the user side; your driver side was returning zero (with `uio_resid` decremented to 4096) for the first call and `ENOSPC` for the second. Comparing what `truss` sees against what your `device_printf` says is an excellent way to catch mismatches between the driver's intent and the kernel's reporting.

`truss -f` follows forks, which is useful when your test harness spawns worker processes. `truss -d` prefixes each line with a relative timestamp; useful for reasoning about latency between calls. Both flags are small investments; the rewards add up quickly when you start running multi-process stress tests.

9.13.10 A Quick Note on `ktrace`

`ktrace(1)` is `truss`'s bigger sibling. It records a binary trace to a file (`ktrace.out` by default) which you then format with `kdump(1)`. It is the right tool when:

- The test run is long and you do not want to watch output live.
- You want to capture detail that is too fine-grained for `truss` (syscall timing, signal delivery, namei lookups).
- You want to replay a trace later, perhaps on a different machine.

A typical session:

```
% sudo ktrace -i ./stress_rw -s 5
% sudo kdump | head -40
2345 stress_rw CALL  open(0x800123456, 0x1<O_WRONLY>)
2345 stress_rw NAMI  "/dev/myfirst/0"
2345 stress_rw RET   open 3
2345 stress_rw CALL  write(0x3, 0x800123500, 0x40)
```

```

2345 stress_rw RET    write 64
2345 stress_rw CALL  write(0x3,0x800123500,0x40)
2345 stress_rw RET    write 64
...

```

For Chapter 9 the difference between `truss` and `ktrace` is small. Use `truss` as the default; reach for `ktrace` when you need more detail or a recorded trace.

9.13.11 Watching Kernel Memory With `vmstat -m`

Your driver allocates kernel memory through `malloc(9)` with the `M_DEVBUF` type. FreeBSD's `vmstat -m` reveals how many allocations are active in each type bucket. Run it while your driver is loaded and idle, then again while it has a buffer allocated, and the increase will be visible in the `devbuf` row:

```

% vmstat -m | head -1
      Type InUse MemUse HighUse Requests  Size(s)
% vmstat -m | grep devbuf
      devbuf   415   4120K          -    39852  16, 32, 64, 128, 256, 512, 1024, 2048, ...

```

The `InUse` column is the current count of live allocations of this type. `MemUse` is the total size currently in use. `HighUse` is the all-time high-water mark since boot. `Requests` is the lifetime count of `malloc` calls that selected this type.

Load the Stage 2 driver. `InUse` goes up by one (the 4096-byte buffer), `MemUse` goes up by approximately 4 KiB, and `Requests` increments. Unload. `InUse` goes down by one; `MemUse` goes down by the 4 KiB. If it does not, you have a memory leak, and `vmstat -m` just told you so.

This is the second observability channel worth adding to your test rhythm. `sysctl` shows driver-owned counters. `dmesg` shows driver-owned log lines. `vmstat -m` shows kernel-owned allocation counts, and it catches a class of bug (forgot to free) that the first two cannot see.

For a driver that declares its own `malloc` type via `MALLOC_DEFINE(M_MYFIRST, "myfirst", ...)`, `vmstat -m | grep myfirst` is even better: it isolates your driver's allocations from the generic `devbuf` pool. `myfirst` stays with `M_DEVBUF` throughout this chapter for simplicity, but upgrading to a dedicated type is a small change you may want to make before shipping a driver outside the book's lab environment.

9.14 Observability: Making Your Driver Legible

A driver that does the right thing is worth more if you can confirm, from outside the kernel, that it is doing the right thing. This section is a short meditation on the observability choices this chapter has been making, and why.

9.14.1 Three Surfaces: `sysctl`, `dmesg`, `userland`

Your driver presents three surfaces to the operator:

- **`sysctl`** for live counters: point-in-time values the operator can poll.
- **`dmesg` (`device_printf`)** for lifecycle events: open, close, errors, transitions.
- **`/dev`** nodes for the data path: the actual bytes.

Each has a distinct role. `sysctl` tells the operator *what is true right now*. `dmesg` tells the operator *what changed recently*. `/dev` is the thing the operator is actually using.

A well-observed driver uses all three, deliberately. A minimally-observed driver uses only the third, and debugging it requires either a debugger or a lot of guessing.

9.14.2 `Sysctl`: Counters vs State

`myfirst` exposes counters through the `sysctl` tree under `dev.myfirst.0.stats`:

- `attach_ticks`: a point-in-time value (when the driver attached).
- `open_count`: a monotonically-increasing counter (lifetime opens).
- `active_fhs`: a live count (current descriptors).
- `bytes_read`, `bytes_written`: monotonically-increasing counters.
- `bufused`: a live value (current buffer occupancy).

Monotonically-increasing counters are easier to reason about than live values, because their rate of change is informative even when the absolute value is not. An operator who sees `bytes_read` increasing at 1 MB/s has learned something even if 1 MB/s is meaningless out of context.

Live values are essential when the state matters for decisions (`active_fhs > 0` means unload will fail). Choose monotonically-increasing counters first, live values when you need them.

9.14.3 `dmesg`: Events Worth Seeing

`device_printf(9)` writes to the kernel message buffer, which `dmesg` shows. Every line is worth seeing exactly once: use `dmesg` for events, not for continuous status.

The events `myfirst` logs:

- Attach (once per instance).
- Open (once per open).
- Destructor (once per descriptor close).
- Detach (once per instance).

That is four lines per instance per load/unload cycle, plus two lines per open/close pair. Comfortable.

What we do not log:

- Every read or write call. That would flood `dmesg` in any real workload.
- Every `sysctl` read. Those are passive.
- Every successful transfer. The `sysctl` counters carry that information, and they carry it more compactly.

If a driver needs to log something that happens many times a second, the usual answer is to guard the logging with `if (bootverbose)`, so it is silent on production systems but available to developers who boot with `boot -v`. For `myfirst` we do not need even that.

9.14.4 The Trap of Over-Logging

A driver that logs every operation is a driver that hides its important events in a sea of noise. If your `dmesg` shows ten thousand lines of `read returned 0 bytes`, the line that says `buffer full, returning ENOSPC` is invisible.

Keep logs sparse. Log transitions, not states. Log once per instance, not once per call. When in doubt, silence.

9.14.5 Counters You Will Add Later

Chapters 10 and beyond will extend the counter tree with:

- `reads_blocked, writes_blocked`: count of calls that had to sleep (Chapter 10).
- `poll_waiters`: count of active `poll(2)` subscribers (Chapter 10).
- `drain_waits, overrun_events`: ring-buffer diagnostics (Chapter 10).

Each one is one more thing an operator can look at to understand what the driver is doing. The pattern is the same: expose the counters, keep the mechanism silent, let the operator decide when to inspect.

9.14.6 What Your Driver Looks Like Under Light Load

A concrete example is more useful than abstract advice. Load Stage 3, run the companion `stress_rw` program for a few seconds with `sysctl dev.myfirst.0.stats` watching from another terminal, and you see something like this:

Before `stress_rw` starts:

```
dev.myfirst.0.stats.attach_ticks: 12345678
dev.myfirst.0.stats.open_count: 0
dev.myfirst.0.stats.active_fhs: 0
dev.myfirst.0.stats.bytes_read: 0
dev.myfirst.0.stats.bytes_written: 0
dev.myfirst.0.stats.bufused: 0
```

Zero activity, one attach, buffer empty.

During `stress_rw`, with `watch -n 0.5 sysctl dev.myfirst.0.stats`:

```
dev.myfirst.0.stats.attach_ticks: 12345678
dev.myfirst.0.stats.open_count: 2
dev.myfirst.0.stats.active_fhs: 2
dev.myfirst.0.stats.bytes_read: 1358976
```

```
dev.myfirst.0.stats.bytes_written: 1359040
dev.myfirst.0.stats.bufused: 64
```

Two active descriptors (writer + reader), counters climbing, buffer holding 64 bytes of in-flight data. `bytes_written` is slightly ahead of `bytes_read`, which is exactly what you would expect: the writer produced a chunk the reader has not quite consumed yet. The difference equals `bufused`.

After `stress_rw` exits:

```
dev.myfirst.0.stats.attach_ticks: 12345678
dev.myfirst.0.stats.open_count: 2
dev.myfirst.0.stats.active_fhs: 0
dev.myfirst.0.stats.bytes_read: 4800000
dev.myfirst.0.stats.bytes_written: 4800000
dev.myfirst.0.stats.bufused: 0
```

Both descriptors closed. Lifetime opens is 2 (cumulative). Active is 0. `bytes_read` equals `bytes_written`; the reader caught up fully. Buffer is empty.

Three signatures to notice. First, `active_fhs` always tracks live descriptors; it is a live value, not a cumulative counter. Second, `bytes_read == bytes_written` at steady state when the reader is keeping up, plus whatever is sitting in `bufused`. Third, the `open_count` is a lifetime value that never decreases; a quick way to spot churn is to watch it grow while `active_fhs` stays stable.

A driver that behaves predictably under load is a driver you can operate with confidence. Once the counters line up the way this paragraph describes, you have your first real driver, not a toy.

9.15 Signed, Unsigned, and the Perils of Off-by-One

A short section on a class of bug that has caused more kernel panics than almost any other. It shows up especially often in I/O handlers.

9.15.1 `ssize_t` vs `size_t`

Two types dominate I/O code:

- `size_t`: unsigned, used for sizes and counts. `sizeof(x)` returns `size_t`. `malloc(9)` takes `size_t`. `memcpy` takes `size_t`.
- `ssize_t`: signed, used when a value could be negative (usually -1 for error). `read(2)` and `write(2)` return `ssize_t`. `uio_resid` is `ssize_t`.

The two types have the same width on every platform FreeBSD supports, but they do not silently convert between each other without warnings, and they behave very differently when arithmetic underflows.

A subtraction of `size_t` values that would produce a negative result instead wraps around to a huge positive value, because `size_t` is unsigned. For example:

```
size_t avail = sc->buflen - sc->bufused;
```

If `sc->bufused` is larger than `sc->buflen`, `avail` is an enormous number, and the next `uiomove` attempts a transfer that blows past the end of the buffer.

The defence is the invariant. In every buffer-management section of the chapter, we maintain `sc->bufhead + sc->bufused <= sc->buflen`. As long as that invariant holds, `sc->buflen - (sc->bufhead + sc->bufused)` cannot underflow.

The risk is in code paths that violate the invariant accidentally. A double-free that restores an already-consumed value; a write that updates `bufused` twice; a race between writers. Those are the bugs to hunt for when `avail` ever looks wrong.

9.15.2 `uio_resid` Can Be Compared Against Unsigned

`uio_resid` is `ssize_t`. Your buffer sizes are `size_t`. Code like this:

```
if (uio->uio_resid > sc->buflen) ...
```

Will be compiled with a signed-vs-unsigned comparison. Modern compilers warn about this; the warning should be taken seriously.

The safer pattern is to cast explicitly:

```
if ((size_t)uio->uio_resid > sc->buflen) ...
```

Or to use `MIN`, which we have been using:

```
towrite = MIN((size_t)uio->uio_resid, avail);
```

The cast is defensible because `uio_resid` is documented to be non-negative in valid `uios` (and `uiomove` KASSERTs on it). The cast makes the compiler happy and makes the intent explicit.

9.15.3 Off-by-One in Counters

A counter updated on the wrong side of an error check is a classic bug:

```
sc->bytes_read += towrite;          /* BAD: happens even on error */
error = uiomove(sc->buf, towrite, uio);
```

The correct shape is to increment after success:

```
error = uiomove(sc->buf, towrite, uio);  
if (error == 0)  
    sc->bytes_read += towrite;
```

This is why we have `if (error == 0)` guarding every counter update in the chapter. The cost is one line of code. The benefit is that your counters match reality.

9.15.4 The `uio_offset` - before Idiom

When you want to know “how many bytes did `uiomove` actually move?”, the cleanest way is to compare `uio_offset` before and after:

```
off_t before = uio->uio_offset;  
error = uiomove_frombuf(sc->buf, sc->buflen, uio);  
size_t moved = uio->uio_offset - before;
```

This works for both full and short transfers. `moved` is the actual byte count, regardless of what the caller asked for or how much was available.

The idiom is free at runtime (two subtractions) and unambiguous in code. Use it when your driver wants to count bytes; the alternative, inferring the count from `uio_resid`, requires knowing the original request size, which is more bookkeeping.

9.16 Additional Troubleshooting: The Edge Cases

Expanding on the earlier troubleshooting section, here are a few more scenarios you are likely to hit the first time you write a real driver.

9.16.1 “The second read on the same descriptor returns zero”

Expected for a static-message driver (Stage 1): once `uio_offset` reaches the end of the message, `uiomove_frombuf` returns zero.

Unexpected for a FIFO driver (Stage 3): the first read drained the buffer, and no writer has refilled it. The caller should not be issuing a second read back-to-back without a write happening in between.

To distinguish the two cases, check `sysctl dev.myfirst.0.stats.bufused`. If it is zero, the buffer is empty. If it is non-zero and you still see zero bytes, you have a bug.

9.16.2 “The driver returns zero bytes immediately when the buffer has data”

The read handler is taking the wrong branch. Common causes:

- A `bufused == 0` check placed in the wrong spot. If the check runs before the per-open state retrieval, it might short-circuit the read before the real work.

- An accidental `return 0`; earlier in the handler (for example, in a debug branch left from a previous experiment).
- A missing `mtx_unlock` on an error path, making every subsequent call block on the mutex forever. Symptom: the second call hangs, not a zero-byte return; but it is worth checking.

9.16.3 “My `uiomove_frombuf` always returns zero regardless of the buffer”

Two common causes:

- The `buflen` argument is zero. `uiomove_frombuf` returns zero immediately if `buflen <= 0`.
- `uio_offset` is already at or past `buflen`. `uiomove_frombuf` returns zero to signal EOF in that case.

Add a `device_printf` logging the arguments at entry to confirm which case you are in.

9.16.4 “The buffer overflows into adjacent memory”

Your arithmetic is wrong. Somewhere you are calling `uiomove(sc->buf + X, N, uio)` where $X + N > sc->buflen$. The write proceeds silently and corrupts kernel memory.

Your kernel will usually panic shortly thereafter, possibly in a completely unrelated subsystem. The panic message will not mention your driver; it will mention whichever heap neighbour got clobbered.

If you suspect this, rebuild with `INVARIANTS` and `WITNESS` (and on many targets, `KASAN` on amd64). These kernel features catch buffer overruns much earlier than the default kernel does.

9.16.5 “A process reading from the device hangs forever”

Since Chapter 9 does not implement blocking I/O, this should not happen with `myfirst` Stage 3. If it does, the most likely cause is the process holding a file descriptor while you tried to unload the driver; `destroy_dev(9)` is waiting for `si_threadcount` to reach zero, and the process is sitting inside your handler for some reason.

To diagnose: `ps auxH | grep <your-test>; gdb -p <pid> and bt`. The stack should reveal where the thread is parked.

If your Stage 3 handler accidentally sleeps (for instance, because you added a `tsleep` while experimenting with Chapter 10 material early), the fix is to remove the sleep. Chapter 9’s driver does not block.

9.16.6 “`kldunload` says `kldunload: can't unload file: Device busy`”

Classic symptom of a descriptor still open. Use `fuser /dev/myfirst/0` to find the offending process, close the descriptor or kill the process, and retry.

9.16.7 “I modified the driver and make compiles but `kldload` fails with version mismatch”

Your build environment does not match your running kernel. Check:

```
% freebsd-version -k
14.3-RELEASE
% ls /usr/obj/usr/src/amd64.amd64/sys/GENERIC
```

If `/usr/src` is for a different release, your headers produce a module that the kernel refuses. Rebuild against the matching sources. In a lab VM this usually means syncing `/usr/src` with the running release via `fetch` or `freebsd-update src-install`.

9.16.8 “I see every byte written through the device printed twice in dmesg”

You have a `device_printf` inside the hot path that prints every transfer. Remove it or guard it with `if (bootverbose)`.

A milder version of the same bug: a single-line log that prints the length of every transfer. For small test workloads that looks fine; for a real user workload it will bury `dmesg` and cause timestamp compression in the kernel buffer.

9.16.9 “My `d_read` is called but my `d_write` is not”

Either the user program never calls `write(2)` on the device, or it calls `write(2)` with the descriptor not opened for writing (`O_RDONLY`). Check both.

Also: confirm that `cdevsw.d_write` is assigned to `myfirst_write`. A copy-paste bug that assigns it to `myfirst_read` results in both directions hitting the read handler, with predictably confusing results.

9.17 Design Notes: Why Each Stage Stops Where It Does

A short meta-section on why Chapter 9’s three stages have the boundaries they have. This is the kind of chapter-design reasoning that is worth making explicit, because it is the reasoning you will apply when you design your own drivers.

9.17.1 Why Stage 1 Exists

Stage 1 is the smallest possible `d_read` that is not `/dev/null`. It introduces:

- The `uiomove_frombuf(9)` helper, the easiest way to get a fixed buffer out to user space.
- Per-descriptor offset handling.
- The pattern of using `uio_offset` as the state carrier.

Stage 1 does not do anything with writes; the Chapter 8 stub is fine.

Without Stage 1, the jump from stubs to a buffered read/write driver is too large. Stage 1 lets you confirm, with minimal code, that the read handler is wired up correctly. Everything else builds on that confirmation.

9.17.2 Why Stage 2 Exists

Stage 2 introduces:

- A dynamically-allocated kernel buffer.
- A write path that accepts user data.
- A read path that honours the caller's offset across the accumulating buffer.
- The first realistic use of the `softc` mutex in an I/O handler.

Stage 2 deliberately does not drain reads. The buffer grows until full; subsequent writes return `ENOSPC`. This lets two concurrent readers confirm that they each have their own `uio_offset`, which is the property Stage 1 couldn't demonstrate (because Stage 1 had nothing to write).

9.17.3 Why Stage 3 Exists

Stage 3 introduces:

- Reads that drain the buffer.
- The coordination between a head pointer and a used count.
- The FIFO semantics that most real drivers approximate.

Stage 3 does not wrap around. The head and used pointers walk forward through the buffer and the buffer collapses to the beginning when empty. A proper ring buffer (with head and tail wrapping around a fixed-size array) belongs in Chapter 10 because it pairs naturally with blocking reads and `poll(2)`: a ring makes steady-state operation efficient, and efficient steady-state operation is exactly what a blocking reader needs.

9.17.4 Why No Ring Buffer Here

A ring buffer is five to fifteen lines of additional bookkeeping beyond what Stage 3 does. Adding it now would not be a large amount of code. The reason it is deferred is pedagogical: the two concepts ("I/O path semantics" and "ring buffer mechanics") are independently confusing to a beginner, and splitting them into two chapters lets each chapter address one pile of confusion at a time.

By the time Chapter 10 introduces the ring, the reader is fluent in the I/O path. The new material is only the ring bookkeeping.

9.17.5 Why No Blocking

Blocking is useful, but it introduces `msleep(9)`, condition variables, the `d_purge` teardown hook, and a thicket of correctness issues around what to wake and when. Each of those is a substantial topic. Mixing them into Chapter 9 would double its length and halve its clarity.

Chapter 10's first section is "when your driver has to wait". It is a natural follow-on.

9.17.6 What the Stages Are Not Trying to Be

The stages are not a simulation of a hardware driver. They do not mimic DMA. They do not simulate interrupts. They do not pretend to be anything other than what they are: in-memory drivers that exercise the UNIX I/O path.

This matters because later in the book, when we write actual hardware drivers, the I/O path will look identical. The hardware specifics (where the bytes come from, where the bytes go to) will change, but the handler shape, the `uiomove` usage, the `errno` conventions, the counter patterns, all of these will be recognisable from Chapter 9.

A driver that moves bytes correctly across the user/kernel trust boundary is 80% of any real driver. Chapter 9 teaches that 80%.

9.18 Hands-On Labs

The labs below track the three stages above. Each lab is a checkpoint that proves your driver is doing the thing the text just described. Read the lab fully before starting, and do them in order.

9.18.1 Lab 9.1: Build and Load Stage 1

Goal: Build the Stage 1 driver, load it, read the static message, and confirm per-descriptor offset handling.

Steps:

1. Start from the companion tree: `cp -r examples/part-02/ch09-reading-and-writing/stage1-static-message ~/drivers/ch09-stage1`. Alternatively, modify your Chapter 8 stage 2 driver according to the Stage 1 walkthrough above.

2. Change into the directory and build:

```
% cd ~/drivers/ch09-stage1
% make
```

3. Load the module:

```
% sudo kldload ./myfirst.ko
```

4. Confirm the device is present:

```
% ls -l /dev/myfirst/0
crw-rw----  1 root  operator ... /dev/myfirst/0
```

5. Read the message:

```
% cat /dev/myfirst/0
Hello from myfirst.
This is your first real read path.
Chapter 9, Stage 1.
```

6. Build the `rw_myfirst.c` userland tool from the companion tree and run it in “read twice” mode:

```
% cc -o rw_myfirst rw_myfirst.c
% ./rw_myfirst read
[read 1] 75 bytes:
Hello from myfirst.
This is your first real read path.
Chapter 9, Stage 1.
[read 2] 0 bytes (EOF)
```

7. Confirm the per-descriptor counter:

```
% dmesg | tail -5
```

You should see the `open via /dev/myfirst/0 fh=...` and `per-open dtor fh=...` lines from Chapter 8, plus the message body was read.

8. Unload:

```
% sudo kldunload myfirst
```

Success criteria:

- `cat` prints the message.
- The userland tool shows 75 bytes on the first read and 0 bytes on the second.
- `dmesg` shows one open and one destructor per `./rw_myfirst read` invocation.

Common mistakes:

- Forgetting the `-1` on `sizeof(myfirst_message) - 1`. The message will include a trailing NUL byte that appears as a stray character in user output.
- Not calling `devfs_get_cdevpriv` before the `sc == NULL` check. The rest of the chapter depends on this order; run it to see why it is the right one.
- Using `(void *)sc->message` instead of `__DECONST(void *, sc->message)`. Both work on most compilers; the `__DECONST` form is the convention and suppresses a warning on some compiler configurations.

9.18.2 Lab 9.2: Exercise Stage 2 with Writes and Reads

Goal: Build Stage 2, push data in from userland, pull it back out, and observe the `sysctl` counters.

Steps:

1. From the companion tree: `cp -r examples/part-02/ch09-reading-and-writing/stage2-readwrite ~/drivers/ch09-stage2`.
2. Build and load:

```
% cd ~/drivers/ch09-stage2
% make
% sudo kldload ./myfirst.ko
```

3. Check the initial state:

```
% sysctl dev.myfirst.0.stats
dev.myfirst.0.stats.attach_ticks: ...
dev.myfirst.0.stats.open_count: 0
dev.myfirst.0.stats.active_fhs: 0
dev.myfirst.0.stats.bytes_read: 0
dev.myfirst.0.stats.bytes_written: 0
dev.myfirst.0.stats.bufused: 0
```

4. Write a line of text:

```
% echo "the quick brown fox" | sudo tee /dev/myfirst/0 > /dev/null
```

5. Read it back:

```
% cat /dev/myfirst/0
the quick brown fox
```

6. Observe the counters:

```
% sysctl dev.myfirst.0.stats.bufused
dev.myfirst.0.stats.bufused: 20
% sysctl dev.myfirst.0.stats.bytes_written
dev.myfirst.0.stats.bytes_written: 20
% sysctl dev.myfirst.0.stats.bytes_read
dev.myfirst.0.stats.bytes_read: 20
```

7. Trigger ENOSPC:

```
% dd if=/dev/zero bs=1024 count=8 | sudo tee /dev/myfirst/0 > /dev/null
```

Expect a short-write error. Inspect `sysctl dev.myfirst.0.stats.bufused`; it should be 4096 (the buffer size).

8. Confirm reads still deliver the content:

```
% sudo cat /dev/myfirst/0 | od -An -c | head -3
```

9. Unload:

```
% sudo kldunload myfirst
```

Success criteria:

- Writes deposit bytes; reads deliver them back.
- `bufused` matches the number of bytes written since the last reset.
- `dd` exhibits a short write when the buffer fills; the driver returns `ENOSPC`.
- `dmesg` shows open and destructor lines for every process that opened the device.

Common mistakes:

- Forgetting to free `sc->buf` in `detach`. The driver will unload without complaint, but a subsequent kernel memory leak check (`vmstat -m | grep devbuf`) will show drift.
- Holding the `softc` mutex while calling `uiomove`, without being sure the mutex is an `MTX_DEF` and not a spin lock. Chapter 7's `mtx_init(..., MTX_DEF)` is the right choice; do not change it.
- Omitting the `sc->bufused = 0` reset in `attach`. Newbus initialises `softc` to zero for you, but making the initialisation explicit is the convention; it also makes a later refactor less error-prone.

9.18.3 Lab 9.3: Stage 3 FIFO Behaviour

Goal: Build Stage 3, exercise FIFO behaviour from two terminals, and confirm that reads drain the buffer.

Steps:

1. From the companion tree: `cp -r examples/part-02/ch09-reading-and-writing/stage3-echo ~/drivers/ch09-stage3`.

2. Build and load:

```
% cd ~/drivers/ch09-stage3
% make
% sudo kldload ./myfirst.ko
```

3. In terminal A, write some bytes:

```
% echo "message A" | sudo tee /dev/myfirst/0 > /dev/null
```

4. In terminal B, read them:

```
% cat /dev/myfirst/0
message A
```

5. Read again in terminal B:

```
% cat /dev/myfirst/0
```

Expect no output. The buffer is empty.

6. In terminal A, write two lines in rapid succession:

```
% echo "first" | sudo tee /dev/myfirst/0 > /dev/null
% echo "second" | sudo tee /dev/myfirst/0 > /dev/null
```

7. In terminal B, read:

```
% cat /dev/myfirst/0
first
second
```

Expect the two lines concatenated. Both writes appended to the same buffer before either read happened.

8. Inspect the counters:

```
% sysctl dev.myfirst.0.stats
```

bufused should be back to zero. bytes_read and bytes_written should match.

9. Unload:

```
% sudo kldunload myfirst
```

Success criteria:

- Writes append to the buffer; reads drain it.
- A read after the buffer is drained returns immediately (EOF-on-empty).
- bytes_read always equals bytes_written once the reader has caught up.

Common mistakes:

- Not resetting bufhead = 0 when bufused reaches zero. The buffer will “drift” toward the end of sc->buf and refuse writes long before it is full.
- Forgetting to update bufhead as reads drain. The driver will read the same bytes repeatedly.
- Using uio->uio_offset as a per-descriptor offset. In a FIFO, offsets are shared; a per-descriptor offset does not make sense and will confuse testers.

9.18.4 Lab 9.4: Using dd to Measure Transfer Behaviour

Goal: Use dd(1) to generate known-size transfers, read the results back, and check that the counters agree.

dd is the tool of choice here because it lets you control the block size, the number of blocks, and the behaviour on short transfers.

1. Reload the Stage 3 driver fresh:

```
% sudo kldunload myfirst; sudo kldload ./myfirst.ko
```

2. Write 512 bytes in a single block:

```
% sudo dd if=/dev/urandom of=/dev/myfirst/0 bs=512 count=1
1+0 records in
1+0 records out
512 bytes transferred
```

3. Observe bufused = 512:

```
% sysctl dev.myfirst.0.stats.bufused
dev.myfirst.0.stats.bufused: 512
```

4. Read them back with matching block size:

```
% sudo dd if=/dev/myfirst/0 of=/tmp/out bs=512 count=1
1+0 records in
```

```
1+0 records out
512 bytes transferred
```

5. Check that the FIFO is now empty:

```
% sysctl dev.myfirst.0.stats.bufused
dev.myfirst.0.stats.bufused: 0
```

6. Write 8192 bytes in one big block:

```
% sudo dd if=/dev/urandom of=/dev/myfirst/0 bs=8192 count=1
dd: /dev/myfirst/0: No space left on device
0+0 records in
0+0 records out
0 bytes transferred
```

The driver accepted 4096 bytes (the buffer size) of the 8192 requested and returned a short write for the rest.

7. Alternatively, use bs=4096 with count=2:

```
% sudo dd if=/dev/urandom of=/dev/myfirst/0 bs=4096 count=2
dd: /dev/myfirst/0: No space left on device
1+0 records in
0+0 records out
4096 bytes transferred
```

The first block of 4096 bytes succeeded in full; the second block failed with ENOSPC.

8. Drain:

```
% sudo dd if=/dev/myfirst/0 of=/tmp/out bs=4096 count=1
% sudo kldunload myfirst
```

Success criteria:

- dd reports the expected byte counts at each step.
- The driver accepts up to 4096 bytes and refuses the rest with ENOSPC.
- bufused tracks the buffer state after every operation.

9.18.5 Lab 9.5: A Small Round-Trip C Program

Goal: Write a short userland C program that opens the device, writes known bytes, closes the descriptor, opens it again, reads the bytes back, and verifies they match.

1. Save the following as `rw_myfirst.c` in `~/drivers/ch09-stage3`:

```
#include <fcntl.h>
#include <unistd.h>
#include <string.h>
#include <stdio.h>

static const char payload[] = "round-trip test payload\n";

int
main(void)
{
    int fd;
    ssize_t n;

    fd = open("/dev/myfirst/0", O_WRONLY);
    if (fd < 0) { perror("open W"); return 1; }
    n = write(fd, payload, sizeof(payload) - 1);
    if (n != (ssize_t)(sizeof(payload) - 1)) {
        fprintf(stderr, "short write: %zd\n", n);
        return 2;
    }
    close(fd);

    char buf[128] = {0};
    fd = open("/dev/myfirst/0", O_RDONLY);
    if (fd < 0) { perror("open R"); return 3; }
    n = read(fd, buf, sizeof(buf) - 1);
    if (n < 0) { perror("read"); return 4; }
    close(fd);

    if ((size_t)n != sizeof(payload) - 1 ||
        memcmp(buf, payload, n) != 0) {
        fprintf(stderr, "mismatch: wrote %zu, read %zd\n",
            sizeof(payload) - 1, n);
        return 5;
    }

    printf("round-trip OK: %zd bytes\n", n);
    return 0;
}
```

2. Build and run:

```
% cc -o rw_myfirst rw_myfirst.c
% sudo ./rw_myfirst
round-trip OK: 24 bytes
```

3. Inspect dmesg to see the two opens and two destructors.

Success criteria:

- The program prints round-trip OK: 24 bytes.
- dmesg shows one open/destructor pair for the write and one for the read.

Common mistakes:

- Writing fewer bytes than the payload and not checking the return value. `write(2)` can return a short count; your test must handle it.
- Forgetting `O_WRONLY` vs `O_RDONLY`. `open(2)` enforces the mode against the access bits of the node; opening with the wrong mode returns `EACCES` (or similar).
- Assuming `read(2)` returns the requested count. It can return less; again, the caller loops.

9.18.6 Lab 9.6: Inspecting Binary Round-Trips

Goal: Confirm that the driver handles arbitrary binary data, not only text, by pushing random bytes through and checking that the same bytes come back.

1. With Stage 3 loaded and empty, write 256 random bytes:

```
% sudo dd if=/dev/urandom of=/tmp/sent bs=256 count=1
% sudo dd if=/tmp/sent of=/dev/myfirst/0 bs=256 count=1
```

2. Read the same number of bytes back:

```
% sudo dd if=/dev/myfirst/0 of=/tmp/received bs=256 count=1
```

3. Compare:

```
% cmp /tmp/sent /tmp/received && echo MATCH
MATCH
```

4. Inspect both files byte-for-byte:

```
% od -An -tx1 /tmp/sent | head -2
% od -An -tx1 /tmp/received | head -2
```

5. Try a pathological pattern: all zeros, all `0xff`, then a file full of a single byte. Confirm every pattern round-trips exactly.

Success criteria:

- `cmp` reports no differences.
- The driver preserves every bit of the input.
- No byte-ordering, no “helpful” interpretation, no surprise transformations.

This lab is short but important: it verifies that your driver is a transparent byte store, not a text filter that accidentally interprets some bytes specially. If you ever see differences between the sent and received files, you have a bug in the transfer path, probably a length miscount or an off-by-one in the buffer arithmetic.

9.18.7 Lab 9.7: Observing a Running Driver End-to-End

Goal: Combine `sysctl`, `dmesg`, `truss`, and `vmstat` into a single end-to-end observation of the Stage 3 driver under real load. This lab has no new code; it is the bridge from “I wrote the driver” to “I can see what it is doing”.

Steps:

1. With Stage 3 loaded fresh, open four terminals. Terminal A will run the driver load / unload cycles. Terminal B will monitor `sysctl`. Terminal C will tail `dmesg`. Terminal D will run a user workload.

2. **Terminal A:**

```
% sudo kldload ./myfirst.ko
% vmstat -m | grep devbuf
```

Note the `devbuf` row's `InUse` and `MemUse` values.

3. **Terminal B:**

```
% watch -n 1 sysctl dev.myfirst.0.stats
```

4. **Terminal C:**

```
% sudo dmesg -c > /dev/null
% sudo dmesg -w
```

The `-c` clears accumulated messages; the `-w` watches for new ones.

5. **Terminal D:**

```
% cd examples/part-02/ch09-reading-and-writing/userland
% make
% sudo truss ./rw_myfirst rt 2>&1 | tail -10
```

6. Check terminal B: you should see `open_count` increment by 2 (one for the write, one for the read), `active_fhs` return to 0, and `bytes_read == bytes_written`.
7. Check terminal C: you should see two open lines and two destructor lines from `device_printf`.
8. In terminal A, run `vmstat -m | grep devbuf` again. `InUse` and `MemUse` should have decreased back to their pre-load values plus whatever the driver itself allocated (typically just the 4 KiB buffer and the softc).
9. **Stress run:** in terminal D,

```
% sudo ./stress_rw -s 5
```

Watch terminal B. You should see `bufused` oscillate, counters climb, and `active_fhs` hit 2 while the test runs.

10. When the stress run finishes, in terminal B verify `active_fhs` is 0. In terminal A,

```
% sudo kldunload myfirst
% vmstat -m | grep devbuf
```

`InUse` should have returned to its pre-load baseline. If it has not, your driver leaked an allocation and `vmstat -m` just told you.

Success criteria:

- `Sysctl` counters match the workload you ran.
- `Dmesg` shows one open/destructor pair per descriptor open/close.
- Truss output matches your mental model of what the program did.
- `vmstat -m | grep devbuf` returns to its baseline after unload.
- No panics, no warnings, no unexplained counter drift.

Why this lab matters: this is the first lab that exercises the full observability toolchain at once. In production, the signal that something is wrong almost never comes from a crash; it comes from a counter that has drifted out of bounds, a `dmesg` line nobody expected, or a `vmstat -m` reading that does not match reality. Building the habit of looking at all four surfaces together is what separates “I wrote a driver” from “I am responsible for a driver”.

9.19 Challenge Exercises

These challenges stretch the material without introducing topics that belong to later chapters. Each one uses only the primitives we have introduced. Try them before looking at the companion tree; the learning is in the attempt, not the answer.

9.19.1 Challenge 9.1: Per-Descriptor Read Counters

Extend Stage 2 so that the per-descriptor reads counter is exposed via a `sysctl`. The counter should be available per active descriptor, which means a per-`fh` `sysctl` rather than a per-`softc` one.

This challenge is harder than it looks: `sysctls` are allocated and freed at known points in the `softc` lifecycle, and the per-descriptor structure lives only as long as its descriptor. A clean solution registers a `sysctl` node per `fh` in `d_open` and unregisters it in the destructor. Be careful about lifetimes; the `sysctl` context must be freed before the `fh` memory.

Hint: `sysctl_ctx_init` and `sysctl_ctx_free` are per-context. You can give each `fh` its own context, and free it in the destructor.

Alternative: keep a linked list of `fh` pointers in the `softc` (under the mutex) and expose it through a custom `sysctl` handler that walks the list on demand. This is the pattern `/usr/src/sys/kern/tty_info.c` uses for per-process stats.

9.19.2 Challenge 9.2: A `readv(2)`-Aware Test

Write a user program that uses `readv(2)` to read from the driver into three separate buffers of sizes 8, 16, and 32 bytes. Confirm that the driver delivers bytes into all three buffers in sequence.

The kernel and `uiomove(9)` already handle `readv(2)`; the driver does not need changes. The purpose of this challenge is to convince yourself of that fact.

Hint: `struct iovec iov[3] = {{buf1, 8}, {buf2, 16}, {buf3, 32}};`, then `readv(fd, iov, 3)`. The return value is the total bytes delivered across all three buffers; the individual `iov_len` values are not modified on the user side.

9.19.3 Challenge 9.3: Short-Write Demonstration

Modify Stage 2's `myfirst_write` to accept at most 128 bytes per call, regardless of `uio_resid`. A user program that writes 1024 bytes should see a short write of 128 every time.

Then write a short test program that writes 1024 bytes in a single `write(2)` call, observes the short-write return value, and loops until all 1024 bytes have been accepted.

Questions to think through:

- Does `cat` handle short writes correctly? (Yes.)
- Does `echo > /dev/myfirst/0 "..."` handle them correctly? (Usually, via `printf` in the shell, but sometimes not; worth testing.)
- What happens if you remove the short-write behaviour and try to exceed the buffer size? (You get `ENOSPC` after the first 4096-byte write.)

This challenge teaches you to separate “the driver does the right thing” from “user programs assume what drivers do”.

9.19.4 Challenge 9.4: An `ls -l` Sensor

Make the driver's response to a read depend on the `ls -l` output of the device itself. That is: every read produces the current timestamp of the device node.

Hint: `sc->cdev->si_ctime` and `sc->cdev->si_mtime` are `struct timespec` fields on the `cdev`. You can convert them to a string with `printf` formatting, place the string in a kernel buffer, and `uiomove_frombuf(9)` it out.

Warning: `si_ctime` / `si_mtime` may be updated by `devfs` as nodes are touched. Observe what happens when you touch `/dev/myfirst/0` and read again.

9.19.5 Challenge 9.5: A Reverse-Echo Driver

Modify Stage 3 so that every read returns the bytes in reverse order from how they were written. A write of “hello” followed by a read should produce “olleh”.

This challenge is entirely about buffer bookkeeping. The `uiomove` calls stay the same; you change the addresses you hand to them.

Hint: You can either reverse the buffer on every read (expensive) or store bytes in reverse order on the write side (cheaper). Neither is the “right” answer; each has different correctness and concurrency properties. Pick one and argue for it in a comment.

9.19.6 Challenge 9.6: Binary Round Trip

Write a user program that writes a `struct timespec` to the driver, then reads one back. Compare the two structures. Are they equal? They should be, because `myfirst` is a transparent byte store.

Extend the program to write two `struct timespec` values, then `lseek(fd, sizeof(struct timespec), SEEK_SET)` and read the second one. What happens? (Clue: the FIFO does not support seeks meaningfully.)

This challenge illustrates the “read and write carry bytes, not types” point from the safe-data-transfer section. The bytes round-trip perfectly; the type information does not.

9.19.7 Challenge 9.7: A Hex-View Test Harness

Write a short shell script that, given a byte count `N`, generates `N` random bytes with `dd if=/dev/urandom bs=$N count=1`, pipes them into your Stage 3 driver, then reads them back with `dd if=/dev/myfirst/0 bs=$N count=1`, and compares the two streams with `cmp`. The script should report success for matching streams and diff-like output for mismatching streams. Run it with `N = 1, 2, 4, ..., 4096` to sweep small, boundary, and capacity-filling sizes.

Questions to answer as you run the sweep:

- Does every size round-trip cleanly up to and including 4096?
- At 4097, what does the driver do? Does the test harness report the error meaningfully?
- Is there any size at which `cmp` reports a difference? If so, what was the underlying cause?

This challenge rewards combining the tools in the Practical Workflow section: `dd` for precise transfers, `cmp` for byte-level verification, `sysctl` for counters, and the shell for orchestration. A robust test harness like this is the kind of habit that pays for itself every time you refactor a driver and want to know quickly whether the behaviour is still right.

9.19.8 Challenge 9.8: Who Has the Descriptor Open?

Write a small C program that opens `/dev/myfirst/0`, blocks on `pause()` (so it holds the descriptor indefinitely), and runs until `SIGTERM`. In a second terminal, run `fstat | grep myfirst` and then `fuser /dev/myfirst/0`. Note the output. Now try to `kldunload myfirst`. What error do you get? Why?

Now kill the holder with `SIGTERM` or plain `kill`. Observe the destructor fire in `dmesg`. Try `kldunload` again. It should succeed.

This challenge is short, but it grounds one of the chapter’s subtler invariants: a driver cannot unload while any descriptor is open on one of its `cdevs`, and FreeBSD gives operators a standard set of tools to find the holder. The next time a real-world `kldunload` fails with `EBUSY`, you will have seen the shape of the problem before.

9.20 Troubleshooting Common Mistakes

Every `d_read` / `d_write` mistake you are likely to make falls into one of a small number of categories. This section is a short field guide.

9.20.1 “My driver returns zero bytes even though I wrote data”

This is usually one of two bugs.

Bug 1: You forgot to update `bufused` (or equivalent) after the successful `uiomove`. The write arrived, the bytes moved, but the driver’s state never reflected the arrival. The next read sees `bufused == 0` and reports EOF.

Fix: always update your tracking fields inside `if (error == 0) { ... }` after `uiomove` returns.

Bug 2: You reset `bufused` (or `bufhead`) somewhere inappropriate. A common pattern is adding a reset line inside `d_open` or `d_close` “for cleanliness”. That wipes out the data the previous caller wrote.

Fix: reset driver-wide state only in `attach` (at load) or `detach` (at unload). Per-descriptor state belongs in `fh`, reset by `malloc(M_ZERO)` and cleaned up by the destructor.

9.20.2 “My reads return garbage”

The buffer is uninitialised. `malloc(9)` without `M_ZERO` returns a block of memory whose contents are undefined. If your `d_read` reaches past `bufused`, or reads from offsets that have not been written, the bytes you see are leftovers from whatever memory the kernel recycled.

Fix: always pass `M_ZERO` to `malloc` in `attach`. Always clamp reads to the current high-water mark (`bufused`), not to the buffer’s total size (`buflen`).

There is a more serious variant of this bug. A driver that returns uninitialised kernel memory to user space has just leaked kernel state into user space. In production that is a security hole. In development it is a bug; in production it is a CVE.

9.20.3 “The kernel panics with a pagefault on a user address”

You called `memcpy` or `bcopy` directly on a user pointer instead of going through `uiomove` / `copyin` / `copyout`. The access faulted, the kernel had no fault handler installed, and the result was a panic.

Fix: never dereference a user pointer directly. Route through `uiomove(9)` (in handlers) or `copyin(9)` / `copyout(9)` (in other contexts).

9.20.4 “The driver refuses to unload”

You have at least one file descriptor still open. `detach` returns `EBUSY` when `active_fhs > 0`; the module will not unload until every `fh` has been destroyed.

Fix: close the descriptor in userland. If a background process is holding it, kill the process (after confirming it is yours; do not kill system daemons). `fstat -p <pid>` shows which files a process has open; `fuser /dev/myfirst/0` shows which processes have the node open.

Chapter 10 will introduce `destroy_dev_drain` patterns for drivers that need to coerce a blocked reader to exit. Chapter 9 does not block, so this issue does not arise in normal operation; when it arises, it is because userland is holding the descriptor somewhere unexpected.

9.20.5 “My write handler returns EFAULT”

Your `uiomove` call hit an invalid user address. The common causes:

- A user program called `write(fd, NULL, n)` or `write(fd, (void*)0xdeadbeef, n)`.
- A user program wrote a pointer it had freed.
- You accidentally passed a kernel pointer as the destination to `uiomove`. This can happen if you build a `uio` by hand for kernel-space data and then pass it to a handler expecting a user-space `uio`. The resulting copyout sees a “user” address that is actually a kernel address; depending on the architecture, you either get `EFAULT` or a subtle corruption.

Fix: check `uio->uio_segflg`. For user-driven handlers, it should be `UIO_USERSPACE`. If you are passing around a kernel-space `uio`, make sure `uio_segflg == UIO_SYSSPACE` and that your code paths know the difference.

9.20.6 “My counters are wrong under concurrent writes”

Two writers raced on `bufused`. Each read the current value, added to it, and wrote back, and the second writer overwrote the first writer’s update with a stale value.

Fix: take `sc->mtx` around every read-modify-write of shared state. Part 3 makes this a first-class topic; for Chapter 9, a single mutex around the whole critical section is enough.

9.20.7 “sysctl counters do not reflect the real state”

Two variants.

Variant A: the counter is a `size_t`, but the `sysctl` macro is `SYSCTL_ADD_U64`. On 32-bit architectures, the macro reads 8 bytes where the field is only 4 bytes wide; half the value is junk.

Fix: match the `sysctl` macro to the field type. `size_t` pairs with `SYSCTL_ADD_UINT` on 32-bit platforms and `SYSCTL_ADD_U64` on 64-bit platforms. To be portable, use `uint64_t` for counters and cast when updating.

Variant B: the counter is never updated because the update is inside the `if (error == 0)` block and `uiomove` returned a non-zero error. That is actually correct behaviour: you should not count bytes you did not move. The symptom only looks like a bug if you are trying to use the counter to debug the error.

Fix: add an `error_count` counter that ticks on every non-zero return, independently of `bytes_read` and `bytes_written`. Useful for debugging.

9.20.8 “The first read after a fresh load returns zero bytes”

Usually intentional. In Stage 3, an empty buffer returns zero bytes. If you expected the static message from Stage 1, check that you are running the Stage 1 driver, not a later one.

If it is unintentional, double-check that `attach` is setting `sc->buf`, `sc->buflen`, and `sc->message_len` as expected. A common bug is copy-pasting the `attach` code from Stage 1 into Stage 2 and leaving the `sc->message = ...` assignment in place, which then takes precedence over the `malloc` line.

9.20.9 “The build fails with unknown reference to `uiomove_frombuf`”

You forgot to include `<sys/uio.h>`. Add it to the top of `myfirst.c`.

9.20.10 “My handler is called twice for one `read(2)`”

It almost certainly is not. What is more likely: your handler is being called once with `uio_iovcnt > 1` (a `readv(2)` call), and inside `uiomove` each `iovec` entry is being drained in turn. The internal loop in `uiomove` may make multiple copyout calls in what is a single invocation of your handler.

Verify by adding a `device_printf` at entry and exit of your `d_read`. You should see one entry and one exit per user-space `read(2)` call, regardless of `iovec` count.

9.21 Contrast Patterns: Correct vs. Buggy Handlers

The troubleshooting guide above is reactive: it helps when something has already gone wrong. This section is the prescriptive companion. Each entry shows a plausible but wrong way to write part of a handler, pairs it with the correct rewrite, and explains the distinction. Studying the contrasts in advance is the fastest way to avoid the bugs in the first place.

Read each pair carefully. The correct version is the pattern you should reach for; the buggy version is the shape your own hands may produce when you are moving fast. Recognising the mistake in the wild, months from now, is worth the five minutes it takes to internalise the difference today.

9.21.1 Contrast 1: Returning a Byte Count

Buggy:

```
static int
myfirst_read(struct cdev *dev, struct uio *uio, int ioflag)
{
    /* ... */
    error = uiomove_frombuf(sc->message, sc->message_len, uio);
    if (error)
        return (error);
    return (sc->message_len); /* BAD: returning a count */
}
```

Correct:

```
static int
myfirst_read(struct cdev *dev, struct uio *uio, int ioflag)
{
    /* ... */
    return (uiomove_frombuf(sc->message, sc->message_len, uio));
}
```

Why it matters. The handler's return value is an `errno`, not a count. The kernel computes the byte count from the change in `uio->uio_resid` and reports it to user space. A non-zero positive return is interpreted as an `errno`; if you returned `sc->message_len`, the caller would receive a very strange `errno` value. For example, returning 75 would manifest as `errno = 75`, which on FreeBSD happens to be `EPROGMISMATCH`. The bug is both wrong and deeply confusing to anybody looking at it from the user side.

The rule is simple and absolute: handlers return `errno` values, never counts. If you want to know the byte count, compute it from the `uio`.

9.21.2 Contrast 2: Handling a Zero-Length Request

Buggy:

```
static int
myfirst_read(struct cdev *dev, struct uio *uio, int ioflag)
{
    if (uio->uio_resid == 0)
        return (EINVAL); /* BAD: zero-length is legal */
    /* ... */
}
```

Correct:

```
static int
myfirst_read(struct cdev *dev, struct uio *uio, int ioflag)
{
    /* No special case. uiomove handles zero-resid cleanly. */
    return (uiomove_frombuf(sc->message, sc->message_len, uio));
}
```

Why it matters. A `read(fd, buf, 0)` call is legal UNIX. A driver that rejects it with `EINVAL` breaks programs that use zero-byte reads to check descriptor state. `uiomove` returns zero immediately if the `uio` has nothing to move; your handler does not need to special-case it. Special-casing it wrong is worse than not special-casing it at all.

9.21.3 Contrast 3: Buffer Capacity Arithmetic

Buggy:

```
mtx_lock(&sc->mtx);
avail = sc->buflen - sc->bufused;
towrite = uio->uio_resid;          /* BAD: no clamp */
error = uiomove(sc->buf + sc->bufused, towrite, uio);
if (error == 0)
    sc->bufused += towrite;
mtx_unlock(&sc->mtx);
return (error);
```

Correct:

```
mtx_lock(&sc->mtx);
avail = sc->buflen - sc->bufused;
if (avail == 0) {
    mtx_unlock(&sc->mtx);
    return (ENOSPC);
}
towrite = MIN((size_t)uio->uio_resid, avail);
error = uiomove(sc->buf + sc->bufused, towrite, uio);
if (error == 0)
    sc->bufused += towrite;
mtx_unlock(&sc->mtx);
return (error);
```

Why it matters. The buggy version hands `uiomove` a length of `uio_resid`, which may exceed the buffer's remaining capacity. `uiomove` will not move more than `uio_resid` bytes, but the *destination* is `sc->buf + sc->bufused`, and the math does not know about `sc->buflen`. If the user writes 8 KiB into a 4 KiB buffer with `bufused = 0`, the handler will write 4 KiB past the end of `sc->buf`. That is a classic kernel heap overflow: the crash will not be immediate, will not implicate your driver, and may reveal itself as a panic inside a completely unrelated subsystem half a second later.

The correct version clamps the transfer to `avail`, guaranteeing that the pointer arithmetic stays inside the buffer. The clamp is one `MIN` call, and it is not optional.

9.21.4 Contrast 4: Holding a Spin Lock Across `uiomove`

Buggy:

```
mtx_lock_spin(&sc->spin);          /* BAD: spin lock, not a regular mutex */
error = uiomove(sc->buf + off, n, uio);
```

```
mtx_unlock_spin(&sc->spin);
return (error);
```

Correct:

```
mtx_lock(&sc->mtx);                /* MTX_DEF mutex */
error = uiomove(sc->buf + off, n, uio);
mtx_unlock(&sc->mtx);
return (error);
```

Why it matters. uiomove(9) may sleep. When it calls copyin or copyout, the user page may be paged out, and the kernel may need to page it in from disk, which requires waiting on I/O. A sleep while holding a spin lock (MTX_SPIN) deadlocks the system. FreeBSD's WITNESS framework panics on this the first time it happens, if WITNESS is enabled. On a non-WITNESS kernel the result is silent livelock.

The rule is straightforward: spin locks cannot be held across functions that may sleep, and uiomove may sleep. Use an MTX_DEF mutex (the default, and the one myfirst uses) for softc state that is touched by I/O handlers.

9.21.5 Contrast 5: Resetting Shared State in d_open

Buggy:

```
static int
myfirst_open(struct cdev *dev, int oflags, int devtype, struct thread *td)
{
    struct myfirst_softc *sc = dev->si_drv1;
    /* ... */
    mtx_lock(&sc->mtx);
    sc->bufused = 0;                /* BAD: wipes other readers' data */
    sc->bufhead = 0;
    mtx_unlock(&sc->mtx);
    /* ... */
}
```

Correct:

```
static int
myfirst_open(struct cdev *dev, int oflags, int devtype, struct thread *td)
{
    struct myfirst_softc *sc = dev->si_drv1;
    struct myfirst_fh *fh;
    /* ... no shared-state reset ... */
    fh = malloc(sizeof(*fh), M_DEVBUF, M_WAITOK | M_ZERO);
```

```
/* fh starts zeroed, which is correct per-descriptor state */
/* ... register fh with devfs_set_cdevpriv, bump counters ... */
}
```

Why it matters. `d_open` runs once per descriptor. If two readers open the device, the second open will wipe whatever the first open left behind. Driver-wide state (`sc->bufused`, `sc->buf`, counters) belongs to the whole driver, and is reset only at attach and detach. Per-descriptor state belongs in `fh`, which `malloc(M_ZERO)` initialises to zeros automatically.

A driver that resets shared state in `d_open` looks like it works under a single opener and silently corrupts state when two openers appear. The bug is invisible until the day two users read the device at once.

9.21.6 Contrast 6: Accounting Before Knowing the Outcome

Buggy:

```
sc->bytes_written += towrite;      /* BAD: count before success */
error = uiomove(sc->buf + tail, towrite, uio);
if (error == 0)
    sc->bufused += towrite;
```

Correct:

```
error = uiomove(sc->buf + tail, towrite, uio);
if (error == 0) {
    sc->bufused += towrite;
    sc->bytes_written += towrite;
}
```

Why it matters. If `uiomove` fails part-way through, some bytes may have moved and some may not. The `sc->bytes_written` counter should reflect what actually reached the buffer, not what the driver attempted. Updating counters before the outcome is known makes the counters lie. If a user reads the `sysctl` to diagnose a problem, they see numbers that do not correspond to reality.

The rule: update counters inside the `if (error == 0)` branch, so success is the only path that increments them. This is a small cost for a large correctness benefit.

9.21.7 Contrast 7: Dereferencing a User Pointer Directly

Buggy:

```
/* Imagine the driver somehow gets a user pointer, maybe through ioctl. */
static int
handle_user_string(void *user_ptr)
```

```
{
    char buf[128];
    memcpy(buf, user_ptr, 128);    /* BAD: user pointer in memcpy */
    /* ... */
}
```

Correct:

```
static int
handle_user_string(void *user_ptr)
{
    char buf[128];
    int error;

    error = copyin(user_ptr, buf, sizeof(buf));
    if (error != 0)
        return (error);
    /* ... */
}
```

Why it matters. `memcpy` assumes both pointers refer to memory accessible in the current address space. A user pointer does not. Depending on the platform, the result of passing a user pointer to `memcpy` in kernel context ranges from an EFAULT-equivalent fault (on amd64 with SMAP enabled) to silent data corruption (on platforms without user/kernel separation) to an outright kernel panic.

`copyin` and `copyout` are the one-and-only correct way to access user memory from kernel context. They install a fault handler, validate the address, walk page tables safely, and return EFAULT on any failure. The performance cost is a few extra instructions; the correctness benefit is “the kernel does not panic when a buggy user program is running”.

9.21.8 Contrast 8: Leaking a Per-Open Structure on `d_open` Failure

Buggy:

```
static int
myfirst_open(struct cdev *dev, int oflags, int devtype, struct thread *td)
{
    struct myfirst_fh *fh;
    int error;

    fh = malloc(sizeof(*fh), M_DEVBUF, M_WAITOK | M_ZERO);
    /* ... set fields ... */
    error = devfs_set_cdevpriv(fh, myfirst_fh_dtor);
    if (error != 0)
```

```

        return (error);          /* BAD: fh is leaked */
    return (0);
}

```

Correct:

```

static int
myfirst_open(struct cdev *dev, int oflags, int devtype, struct thread *td)
{
    struct myfirst_fh *fh;
    int error;

    fh = malloc(sizeof(*fh), M_DEVBUF, M_WAITOK | M_ZERO);
    /* ... set fields ... */
    error = devfs_set_cdevpriv(fh, myfirst_fh_dtor);
    if (error != 0) {
        free(fh, M_DEVBUF);      /* free before returning */
        return (error);
    }
    return (0);
}

```

Why it matters. When `devfs_set_cdevpriv` fails, the kernel does not register the destructor, so the destructor will never run on this `fh`. If the handler returns without freeing `fh`, the memory is leaked. Under steady load, repeated `d_open` failures can leak enough memory to destabilise the kernel.

The rule: in error-unwind paths, every allocation made so far must be freed. The Chapter 8 reader has seen this pattern for `attach`; it applies equally to `d_open`.

9.21.9 How to Use This Contrast Table

These eight pairs are not an exhaustive list. They are the bugs we have seen most often in early student drivers, and the bugs the chapter's text has been trying to help you avoid. Read through them once now. Before you write your first real driver outside this book, read them again.

A useful habit while developing: whenever you finish a handler, walk it against the contrast table mentally. Does the handler return a count? Does it special-case zero-resid? Does it have a capacity clamp? Is the mutex type right? Does it reset shared state in `d_open`? Does it account for bytes on failure? Does it dereference any user pointer directly? Does it leak on `d_open` failure? Eight questions, five minutes. The price of the check is small; the cost of shipping one of these bugs into production is large.

9.22 Self-Assessment Before Chapter 10

Chapter 9 has covered a lot of ground. Before you put it down, run through the following checklist. If any item makes you hesitate, the relevant section is worth re-reading before moving on. This is not a test; it is a quick

way to identify the spots where your mental model may still be thin.

Concepts:

- ☐ I can explain in one sentence what `struct uio` is for.
- ☐ I can name the three fields of `struct uio` my driver reads most often.
- ☐ I can explain why `uiomove(9)` is preferred over `copyin / copyout` inside `d_read` and `d_write`.
- ☐ I can explain why `memcpy` across the user / kernel boundary is unsafe.
- ☐ I can explain the difference between `ENXIO`, `EAGAIN`, `ENOSPC`, and `EFAULT` in driver terms.

Mechanics:

- ☐ I can write a minimal `d_read` handler that serves a fixed buffer using `uiomove_frombuf(9)`.
- ☐ I can write a minimal `d_write` handler that appends to a kernel buffer with a correct capacity clamp.
- ☐ I know where to put the mutex acquire and release around the transfer.
- ☐ I know how to propagate an `errno` from `uiomove` back to user space.
- ☐ I know how to mark a write as fully consumed with `uio_resid = 0`.

Observability:

- ☐ I can read `sysctl dev.myfirst.0.stats` and interpret each counter.
- ☐ I can spot a memory leak with `vmstat -m | grep devbuf`.
- ☐ I can use `truss(1)` to see what syscalls my test program makes.
- ☐ I can use `fstat(1)` or `fuser(8)` to find who is holding a descriptor.

Traps:

- ☐ I would not return a byte count from `d_read / d_write`.
- ☐ I would not reject a zero-length request with `EINVAL`.
- ☐ I would not reset `sc->bufused` inside `d_open`.
- ☐ I would not hold a spin lock across a `uiomove` call.

Any “no” here is a signal, not a verdict. Re-read the relevant section; run a small experiment in your lab; come back to the checklist. By the time every box is ticked, you are solidly ready for Chapter 10.

9.23 Wrapping Up

You have just implemented the entry points that make a driver alive. At the end of Chapter 7 your driver had a skeleton. At the end of Chapter 8 it had a well-shaped door. Now, at the end of Chapter 9, data flows through the door in both directions.

The central lesson of the chapter is shorter than it looks. Every `d_read` you will ever write has the same three-line spine: get per-open state, verify liveness, call `uiomove`. Every `d_write` you will ever write has a similar spine with one extra decision (how much room do I have?) and a clamp (`MIN(uio_resid, avail)`) that prevents buffer overruns. Everything else in the chapter is context: why `struct uio` looks the way it does, why `uiomove` is the only safe mover, why `errno` values matter, why counters matter, why the buffer has to be freed on every error path.

9.23.1 The Three Ideas That Matter Most

First, `struct uio` is the contract between your driver and the kernel's I/O machinery. It carries everything your handler needs to know about a call: what the user asked for, where the user's memory is, what direction the transfer should move, and how much progress has been made. You do not need to memorise all seven fields. You need to recognise `uio_resid` (the remaining work), `uio_offset` (the position, if you care), and `uio_rw` (the direction), and you need to trust `uiomove(9)` with the rest.

Second, `uiomove(9)` is the boundary between user memory and kernel memory. Everything your driver ever moves between the two passes through it (or through one of its close relatives: `uiomove_frombuf`, `copyin`, `copyout`). This is not a suggestion. Direct pointer access across the trust boundary either corrupts memory or leaks information, and the kernel has no cheap way to catch the mistake before it becomes a CVE. If a pointer came from user space, route it through the kernel's trust-boundary functions. Always.

Third, a correct handler is usually a short one. If your `d_read` or `d_write` is longer than fifteen lines, something is probably wrong. Longer handlers either duplicate logic that belongs elsewhere (in the buffer management, in the per-open state setup, in the `sysctl`s), or they are trying to do something the driver should not be doing in a data-path handler (typically, something that belongs in `d_ioctl`). Keep the handlers short. Put the machinery they call into well-named helper functions. Your future self will thank you.

9.23.2 The Shape of the Driver You End the Chapter With

Your Stage 3 `myfirst` is a small, honest, in-memory FIFO. The salient features:

- A 4 KiB kernel buffer, allocated in `attach` and freed in `detach`.
- A per-instance mutex guarding `bufhead`, `bufused`, and the associated counters.
- A `d_read` that drains the buffer and advances `bufhead`, collapsing to zero when the buffer empties.
- A `d_write` that appends to the buffer and returns `ENOSPC` when it fills.
- Per-descriptor counters stored in `struct myfirst_fh`, allocated in `d_open`, freed in the destructor.
- A `sysctl` tree exposing the live driver state.
- Clean `attach` error unwind and clean `detach` ordering.

That shape will come back, recognisable, in half the drivers you will read in Part 4 and Part 6. It is a general pattern, not a one-off demo.

9.23.3 What You Should Practice Before Starting Chapter 10

Five exercises, in rough order of increasing challenge:

1. Rebuild all three stages from scratch, without looking at the companion tree. Compare your result to the tree afterward; the differences are what you have left to internalise.
2. Introduce an intentional bug in Stage 3: forget to reset `bufhead` when `bufused` reaches zero. Observe what happens on the second big write. Explain the symptom in terms of the code.
3. Add a `sysctl` that exposes `sc->buflen`. Make it read-only. Then convert it into a tunable that can be set at load time via `kenv` or `loader.conf` and picked up in `attach`. (Chapter 10 revisits tunables formally; this is a preview.)

4. Write a shell script that writes random data of a known length to `/dev/myfirst/0` and then reads it back through `sha256`. Compare the hashes. Do the hashes match even when the write size exceeds the buffer? (They should not; think about why.)
5. Find a driver under `/usr/src/sys/dev` that implements both `d_read` and `d_write`. Read its handlers. Map them against the patterns in this chapter. Good candidates: `/usr/src/sys/dev/null/null.c` (you already know it), `/usr/src/sys/dev/random/randomdev.c`, `/usr/src/sys/dev/speaker/spkr.c`.

9.23.4 Looking Ahead to Chapter 10

Chapter 10 takes the Stage 3 driver and makes it scale. Four new capabilities show up:

- **A circular buffer** replaces the linear buffer. Writes and reads can both happen continuously without the explicit collapse that Stage 3 uses.
- **Blocking reads** arrive. A reader that calls `read(2)` on an empty buffer can sleep until data is available, rather than returning zero bytes immediately. The kernel's `msleep(9)` is the primitive; the `d_purge` handler is the teardown safety net.
- **Non-blocking I/O** becomes a first-class feature. `O_NONBLOCK` users get `EAGAIN` where a blocking caller would sleep.
- **`poll(2)` and `kqueue(9)` integration.** A user program can wait for the device to become readable or writable without actively attempting the operation. This is the standard way to integrate a device into an event loop.

All four of these build on the same `d_read` / `d_write` shapes you just implemented. You will extend the handlers rather than rewriting them, and the per-descriptor state you have in place will carry the necessary bookkeeping. The chapter before that (this one) is the one where the I/O path itself is correct. Chapter 10 is where the I/O path becomes efficient.

Before you close the file, a last reassurance. The material in this chapter is not as difficult as it may feel on a first read. The pattern is small. The ideas are real, but they are finite, and you have just exercised every one of them against working code. When you read a real driver's `d_read` or `d_write` in the tree, you will now recognise what the function is doing and why. You are not a beginner at this any more. You are an apprentice with a real tool in your hands.

9.24 Reference: The Signatures and Helpers Used in This Chapter

A consolidated reference for the declarations, helpers, and constants the chapter leans on. Keep this page bookmarked while you write drivers; most beginner questions are a lookup in one of these tables.

9.24.1 `d_read` and `d_write` Signatures

From `/usr/src/sys/sys/conf.h`:

```
typedef int d_read_t(struct cdev *dev, struct uio *uio, int ioflag);
typedef int d_write_t(struct cdev *dev, struct uio *uio, int ioflag);
```


The return value is zero on success, a positive errno on failure. The byte count is computed from the change in `uio->uio_resid` and reported to user space as the return value of `read(2)` / `write(2)`.

9.24.2 The Canonical struct uio

From `/usr/src/sys/sys/uio.h`:

```
struct uio {
    struct iovec *uio_iov;           /* scatter/gather list */
    int uio_iovcnt;                  /* length of scatter/gather list */
    off_t uio_offset;                 /* offset in target object */
    ssize_t uio_resid;                /* remaining bytes to process */
    enum uio_seg uio_segflg;          /* address space */
    enum uio_rw uio_rw;               /* operation */
    struct thread *uio_td;            /* owner */
};
```

9.24.3 The uio_seg and uio_rw Enumerations

From `/usr/src/sys/sys/_uio.h`:

```
enum uio_rw { UIO_READ, UIO_WRITE };
enum uio_seg { UIO_USERSPACE, UIO_SYSSPACE, UIO_NOCOPY };
```

9.24.4 uiomove Family

From `/usr/src/sys/sys/uio.h`:

```
int uiomove(void *cp, int n, struct uio *uio);
int uiomove_frombuf(void *buf, int buflen, struct uio *uio);
int uiomove_fromphys(struct vm_page *ma[], vm_offset_t offset, int n,
                     struct uio *uio);
int uiomove_nofault(void *cp, int n, struct uio *uio);
int uiomove_object(struct vm_object *obj, off_t obj_size, struct uio *uio);
```

In beginner driver code, only `uiomove` and `uiomove_frombuf` are common. The others support specific kernel subsystems (physical-page I/O, page-fault-free copies, VM-backed objects) and are out of scope for this chapter.

9.24.5 copyin and copyout

From `/usr/src/sys/sys/systm.h`:

```
int copyin(const void * __restrict uaddr,
           void * __restrict kaddr, size_t len);
int copyout(const void * __restrict kaddr,
            void * __restrict uaddr, size_t len);
int copyinstr(const void * __restrict uaddr,
              void * __restrict kaddr, size_t len,
              size_t * __restrict lencopied);
```

Use these in control paths (`d_ioctl`) where a user pointer arrives outside the `uio` abstraction. Inside `d_read` and `d_write`, prefer `uiomove`.

9.24.6 `ioflag` Bits That Matter for Character Devices

From `/usr/src/sys/sys/vnode.h`:

```
#define IO_NDELAY      0x0004 /* FNDELAY flag set in file table */
```

Set when the descriptor is in non-blocking mode. Your `d_read` or `d_write` can use this to decide whether to block (missing flag) or return `EAGAIN` (flag set). Most of the other `IO_*` flags are filesystem-level and irrelevant to character devices.

9.24.7 Memory Allocation

From `/usr/src/sys/sys/malloc.h`:

```
void *malloc(size_t size, struct malloc_type *type, int flags);
void free(void *addr, struct malloc_type *type);
```

Common flags: `M_WAITOK`, `M_NOWAIT`, `M_ZERO`. Common types for drivers: `M_DEVBUF` (generic) or a driver-specific type declared via `MALLOC_DECLARE` / `MALLOC_DEFINE`.

9.24.8 Per-Open State (Chapter 8 carryover, used here)

From `/usr/src/sys/sys/conf.h`:

```
int devfs_set_cdevpriv(void *priv, d_priv_dtor_t *dtr);
int devfs_get_cdevpriv(void **datap);
void devfs_clear_cdevpriv(void);
```

The pattern is: allocate in `d_open`, register with `devfs_set_cdevpriv`, retrieve in every later handler with `devfs_get_cdevpriv`, clean up in the destructor that `devfs_set_cdevpriv` registered.

9.24.9 Errno Values Used in This Chapter

Errno	Meaning in a driver context
0	Success.
ENXIO	Device not configured (softc missing, not attached).
EFAULT	Bad user address. Usually propagated from uiomove.
EIO	Input / output error. Hardware issue.
ENOSPC	No space left on device. Buffer full.
EAGAIN	Would block; relevant in non-blocking mode (Chapter 10).
EINVAL	Invalid argument.
EACCES	Permission denied at open(2).
EPIPE	Broken pipe. Not used by myfirst.

9.24.10 Helpful device_printf(9) Patterns

```
device_printf(sc->dev, "open via %s fh=%p\n", devtoname(sc->cdev), fh);
device_printf(sc->dev, "write rejected: buffer full (used=%zu)\n",
    sc->bufused);
device_printf(sc->dev, "read delivered %zd bytes\n",
    (ssize_t)(before - uio->uio_offset));
```

These are written for readability. A line in dmesg you have to decode is a line that probably will not be read when it matters.

9.24.11 The Three Stages at a Glance

Stage	d_read	d_write
1	Serve fixed message via uiomove_frombuf	Discard writes (like /dev/null)
2	Serve buffer up to bufused	Append to buffer, ENOSPC if full
3	Drain buffer from bufhead, reset on empty	Append at bufhead + bufused, ENOSPC if full

Stage 3 is the foundation Chapter 10 builds on.

9.24.12 Consolidated File List for the Chapter

Companion files under `examples/part-02/ch09-reading-and-writing/`:

- `stage1-static-message/`: Stage 1 driver source and Makefile.
- `stage2-readwrite/`: Stage 2 driver source and Makefile.
- `stage3-echo/`: Stage 3 driver source and Makefile.
- `userland/rw_myfirst.c`: small C program to exercise read and write round-trips.
- `userland/stress_rw.c`: multi-process stress test for Lab 9.3 and beyond.
- `README.md`: a short map of the companion tree.

Each stage is independent; you can build, load, and exercise any of them without building the others. The Makefiles are identical except for the driver name (always `myfirst`) and optional tuning flags.

9.25 Appendix A: A Closer Look at `uiomove`'s Internal Loop

For readers who want to see exactly what `uiomove(9)` does, this appendix walks through the core loop of `uiomove_faultflag` as it appears in `/usr/src/sys/kern/subr_uio.c`. You do not need to read this to write a driver. It is here because one reading of the loop will clarify every later question you have about `uio` semantics.

9.25.1 The Setup

At entry, the function has:

- A kernel pointer `cp` provided by the caller (your driver).
- An integer `n` provided by the caller (the max bytes to move).
- The `uio` provided by the kernel dispatch.
- A boolean `nofault` indicating whether page faults during the copy should be handled or fatal.

It sanity-checks a few invariants: the direction is `UIO_READ` or `UIO_WRITE`, the owning thread is the current thread when the segment is user-space, and `uio_resid` is non-negative. Any violation is a `KASSERT` and will panic a kernel with `INVARIANTS` enabled.

9.25.2 The Main Loop

```
while (n > 0 && uio->uio_resid) {
    iov = uio->uio_iov;
    cnt = iov->iov_len;
    if (cnt == 0) {
        uio->uio_iov++;
        uio->uio_iovcnt--;
        continue;
    }
    if (cnt > n)
```

```
        cnt = n;

    switch (uio->uio_segflg) {
    case UIO_USERSPACE:
        switch (uio->uio_rw) {
        case UIO_READ:
            error = copyout(cp, iov->iov_base, cnt);
            break;
        case UIO_WRITE:
            error = copyin(iov->iov_base, cp, cnt);
            break;
        }
        if (error)
            goto out;
        break;

    case UIO_SYSSPACE:
        switch (uio->uio_rw) {
        case UIO_READ:
            bcopy(cp, iov->iov_base, cnt);
            break;
        case UIO_WRITE:
            bcopy(iov->iov_base, cp, cnt);
            break;
        }
        break;
    case UIO_NOCOPY:
        break;
    }
    iov->iov_base = (char *)iov->iov_base + cnt;
    iov->iov_len -= cnt;
    uio->uio_resid -= cnt;
    uio->uio_offset += cnt;
    cp = (char *)cp + cnt;
    n -= cnt;
}
```

Each iteration does one unit of work: copy up to `cnt` bytes (where `cnt` is `MIN(iov->iov_len, n)`) between the current `iovec` entry and the kernel buffer. The direction is chosen by the two nested `switch` statements. After a successful copy, all the accounting fields advance in lockstep: the `iovec` entry shrinks by `cnt`, the `uio`'s `resid` shrinks by `cnt`, the `uio`'s `offset` grows by `cnt`, the kernel pointer `cp` advances by `cnt`, and the caller's `n` shrinks by `cnt`.

When an `iovec` entry is fully drained (`cnt == 0` at loop entry), the function advances to the next entry. When the caller's `n` reaches zero or the `uio`'s `resid` reaches zero, the loop terminates.

If `copyin` or `copyout` returns non-zero, the function jumps to out without updating the fields for that iteration, so the partial-copy accounting is consistent: whatever bytes did copy are reflected in `uio_resid`, whatever did not copy is still pending.

9.25.3 What You Should Take Away

Three invariants fall out of the loop that matter for your driver code.

- **Your call to `uiomove(cp, n, uio)` moves at most `MIN(n, uio->uio_resid)` bytes.** There is no way to ask for more than the `uio` has room for; the function caps at whichever side is smaller.
- **On a partial transfer, the state is consistent.** `uio_resid` reflects exactly the bytes that did not move. You can make another call and it will pick up correctly.
- **The fault handling is inside the loop, not around it.** A fault during a `copyin` / `copyout` returns `EFAULT` for the remainder; the fields are still consistent.

These three facts are why the three-line spine we keep returning to (`uiomove`, check error, update state) is sufficient. The kernel is doing the complicated work inside the loop; your driver just has to cooperate.

9.26 Appendix B: Why `read(fd, buf, 0)` Is Allowed

A short note on a question that comes up frequently: why does UNIX allow a `read(fd, buf, 0)` or `write(fd, buf, 0)` call at all?

There are two answers, and both are worth knowing.

The practical answer: zero-length I/O is a free test. A user program that wants to check whether a descriptor is in a reasonable state can call `read(fd, NULL, 0)` without committing to a real transfer. If the descriptor is broken, the call returns an error. If it is fine, the call returns zero and costs almost nothing.

The semantic answer: the UNIX I/O interface uses byte counts consistently, and special-casing zero is more work than allowing it. A call with `count == 0` is a well-defined no-op: the kernel has to do nothing, and can return zero immediately. The alternative, returning `EINVAL` for zero-count calls, would force every user program that computed a count dynamically to guard against the case. That is the kind of change that breaks decades of code for no benefit.

The driver-side consequence, which we noted earlier: your handler must not panic or error on a zero `uio_resid`. The kernel effectively handles the case for you when you go through `uiomove`, which returns zero immediately if there is nothing to move.

If you ever find yourself writing `if (uio->uio_resid == 0) return (EINVAL);` in a driver, stop. That is the wrong answer. Zero-count I/O is valid; return zero.

9.27 Appendix C: A Short Tour of `/dev/zero`'s Read Path

As a closing piece of analysis, it is worth walking through exactly what happens when a user program calls `read(2)` on `/dev/zero`. The driver is `/usr/src/sys/dev/null/null.c` and the handler is `zero_read`. Once you understand this path, you understand everything in Chapter 9.

9.27.1 From User Space to Kernel Dispatch

The user calls:

```
ssize_t n = read(fd, buf, 1024);
```

The C library makes the `read` syscall. The kernel looks up `fd` in the calling process's file table, retrieves the `struct file`, identifies its `vnode`, dispatches the call into `devfs`.

`devfs` identifies the `cdev` associated with the `vnode`, acquires a reference on it, and calls its `d_read` function pointer (`zero_read`) with the `uio` the kernel prepared.

9.27.2 Inside `zero_read`

```
static int
zero_read(struct cdev *dev __unused, struct uio *uio, int flags __unused)
{
    void *zbuf;
    ssize_t len;
    int error = 0;

    KASSERT(uio->uio_rw == UIO_READ,
        ("Can't be in %s for write", __func__));
    zbuf = __DECONST(void *, zero_region);
    while (uio->uio_resid > 0 && error == 0) {
        len = uio->uio_resid;
        if (len > ZERO_REGION_SIZE)
            len = ZERO_REGION_SIZE;
        error = uiomove(zbuf, len, uio);
    }
    return (error);
}
```

- Assert that the direction is correct. Good practice; a `KASSERT` costs nothing in production kernels.
- Set `zbuf` to point at `zero_region`, a large pre-allocated zero-filled area.
- Loop: while the caller wants more bytes, determine the transfer size (min of `uio_resid` and the zero region's size), call `uiomove`, accumulate any error.
- Return.

9.27.3 Inside `uiomove`

For the first iteration, `uiomove` sees `uio_resid = 1024`, `len = 1024` (since `ZERO_REGION_SIZE` is much larger), `uio_segflg = UIO_USERSPACE`, `uio_rw = UIO_READ`. It selects `copyout(zbuf, buf, 1024)`. The kernel performs the copy, handling any page fault on the user buffer. On success, `uio_resid` drops to zero, `uio_offset` grows by 1024, and the `iovec` is fully consumed.

9.27.4 Back Up the Stack

uiomove returns zero. The loop in zero_read sees `uio_resid == 0` and exits. zero_read returns zero.

devfs releases its reference on the cdev. The kernel computes the byte count as $1024 - 0 = 1024$. `read(2)` returns 1024 to the user.

The user's buffer now holds 1024 zero bytes.

9.27.5 What This Tells You About Your Own Driver

Two observations.

First, every data-path decision in zero_read is one you are now making too. How large of a chunk to move per iteration; which buffer to read from; how to handle the error from uiomove. Your driver's decisions will differ in the specifics (your buffer is not a pre-allocated zero region, your chunk size is not `ZERO_REGION_SIZE`), but the shape is identical.

Second, everything above zero_read is kernel machinery you do not have to write. You implement the handler, and the kernel takes care of the syscall, the file-descriptor lookup, the VFS dispatch, the devfs routing, the reference counting, and the fault handling. That is the power of the abstraction: you add your driver's knowledge, and everything else comes for free.

The flip side is that when you write a driver, you are committing to *cooperating* with that machinery. Every invariant that uiomove and devfs rely on is now your responsibility to uphold. The chapter has been walking you through those invariants one at a time, by building three small drivers that each exercise a different subset.

By now, the pattern should be familiar.

9.28 Appendix D: Common read(2) / write(2) Return Values at the User Side

A short cheat sheet for what a user program sees when it talks to your driver. This is not driver code; it is the view from the other side of the trust boundary. Reading it occasionally is the best inoculation against the subtle bugs that arise when the driver does something other than what a well-behaved UNIX program expects.

9.28.1 read(2)

- A positive integer: that many bytes were placed into the caller's buffer. Less than the requested count means a short read; the caller loops.
- Zero: end of file. No more bytes will ever be produced on this descriptor. The caller stops.
- -1 with `errno = EAGAIN`: non-blocking mode, no data available right now. The caller waits (via `select(2)` / `poll(2)` / `kqueue(2)`) and tries again.
- -1 with `errno = EINTR`: a signal interrupted the read. The caller usually retries unless the signal handler tells it not to.
- -1 with `errno = EFAULT`: the buffer pointer was invalid. The caller has a bug.
- -1 with `errno = ENXIO`: the device is gone. The caller should close the descriptor and give up.
- -1 with `errno = EIO`: the device reported a hardware error. The caller may retry or report.

9.28.2 write(2)

- A positive integer: that many bytes were accepted. Less than the offered count means a short write; the caller loops with the remainder.
- Zero: theoretically possible, rarely seen in practice. Usually treated the same as a short write of zero bytes.
- -1 with `errno = EAGAIN`: non-blocking mode, no space right now. The caller waits and retries.
- -1 with `errno = ENOSPC`: permanently no space. The caller either stops writing or reopens the descriptor.
- -1 with `errno = EPIPE`: the reader closed. Relevant for pipe-like devices, not for `myfirst`.
- -1 with `errno = EFAULT`: the buffer pointer was invalid.
- -1 with `errno = EINTR`: interrupted by a signal. Usually retried.

9.28.3 What This Means for Your Driver

Two takeaways.

First, `EAGAIN` is how non-blocking callers expect a driver to say “no data / no room right now, come back later”. A non-blocking caller that sees `EAGAIN` does not treat it as an error; it waits for a wake-up (usually via `poll(2)`) and retries. Chapter 10 makes this mechanism work for `myfirst`.

Second, `ENOSPC` is how a driver signals a permanent out-of-room condition on a write. It differs from `EAGAIN` in that the caller does not expect retries to succeed soon. For `myfirst` Stage 3 we use `ENOSPC` when the buffer fills and there is no reader actively draining; Chapter 10 will layer `EAGAIN` on top of the same condition for non-blocking readers and writers.

A driver that returns the wrong `errno` here is almost indistinguishable from a driver that is misbehaving. The cost of getting it right is tiny. The cost of getting it wrong shows up in confused user programs months later.

9.29 Appendix E: The One-Page Cheat Sheet

If you only have five minutes before starting Chapter 10, here is the one-page version of everything above.

The signatures:

```
static int myfirst_read(struct cdev *dev, struct uio *uio, int ioflag);
static int myfirst_write(struct cdev *dev, struct uio *uio, int ioflag);
```

Return zero on success, a positive `errno` on failure. Never return a byte count.

The three-line spine for reads:

```
error = devfs_get_cdevpriv((void **)&fh);
if (error) return error;
return uiomove_frombuf(sc->buf, sc->buflen, uio);
```

Or, for a dynamic buffer:

```
mtx_lock(&sc->mtx);
toread = MIN((size_t)uio->uio_resid, sc->bufused);
error = uiomove(sc->buf + offset, toread, uio);
if (error == 0) { /* update state */ }
mtx_unlock(&sc->mtx);
return error;
```

The three-line spine for writes:

```
mtx_lock(&sc->mtx);
avail = sc->buflen - (sc->bufhead + sc->bufused);
if (avail == 0) { mtx_unlock(&sc->mtx); return ENOSPC; }
towrite = MIN((size_t)uio->uio_resid, avail);
error = uiomove(sc->buf + sc->bufhead + sc->bufused, towrite, uio);
if (error == 0) { sc->bufused += towrite; }
mtx_unlock(&sc->mtx);
return error;
```

What to remember about uio:

- `uio_resid`: bytes still pending. `uiomove` decrements this.
- `uio_offset`: position, if meaningful. `uiomove` increments this.
- `uio_rw`: direction. Trust `uiomove` to use it.
- Everything else: do not touch.

What not to do:

- Do not dereference user pointers directly.
- Do not use `memcpy` / `bcopy` between user and kernel.
- Do not return byte counts.
- Do not reset driver-wide state in `d_open` / `d_close`.
- Do not forget `M_ZERO` on `malloc(9)`.
- Do not hold a spin lock across `uiomove`.

Errno values:

- 0: success.
- `ENXIO`: device not ready.
- `ENOSPC`: buffer full (permanent).
- `EAGAIN`: would block (non-blocking).
- `EFAULT`: from `uiomove`, propagate.
- `EIO`: hardware error.

That is the chapter.

9.30 Chapter Summary

This chapter built the data path. Starting from the Chapter 8 stubs, we implemented `d_read` and `d_write` in three stages, each a complete and loadable driver.

- **Stage 1** exercised `uiomove_frombuf(9)` against a static kernel string, with per-descriptor offset handling that made two concurrent readers' progress independent.
- **Stage 2** introduced a dynamic kernel buffer, a write path that appended into it, and a read path that served out of it. The buffer was sized at attach time, and a full buffer rejected further writes with `ENOSPC`.
- **Stage 3** turned the buffer into a first-in / first-out queue. Reads drained from the head, writes appended to the tail, and the driver collapsed `bufhead` to zero when the buffer emptied.

Along the way we dissected `struct uio` field by field, explained why `uiomove(9)` is the only legitimate way to cross the user / kernel trust boundary in a read or write handler, and built a small vocabulary of `errno` values that a well-behaved driver uses: `ENXIO`, `EFAULT`, `ENOSPC`, `EAGAIN`, `EIO`. We walked through the internal loop of `uiomove` so that its guarantees feel earned rather than mysterious. And we finished with five labs, six challenges, a troubleshooting guide, and a one-page cheat sheet.

The Stage 3 driver is the waypoint into Chapter 10. It moves bytes correctly. It does not yet move them efficiently: an empty buffer returns zero bytes immediately, a full buffer returns `ENOSPC` immediately, there is no blocking, no `poll(2)` integration, no ring buffer. Chapter 10 fixes all of that, building on exactly the shapes we just drew.

The pattern you just learned repeats. Every character-device I/O handler in `/usr/src/sys/dev` builds on the same three-argument signature, the same `struct uio`, and the same `uiomove(9)` primitive. The differences between drivers are in how they prepare the data, not in how they move it. Now that you recognise the moving machinery, every handler you open becomes legible almost immediately.

You now know enough to read any `d_read` or `d_write` in the FreeBSD tree and understand what it is doing. That is a significant milestone. Take a minute to appreciate it before turning the page.

10 Handling Input and Output Efficiently

10.1 Reader Guidance & Outcomes

Chapter 9 ended with a small but honest driver. Your `myfirst` module attaches as a Newbus device, creates `/dev/myfirst/0` with an alias at `/dev/myfirst`, allocates per-open state, transfers bytes through `uiomove(9)`, and maintains a simple first-in / first-out kernel buffer. You can write data into it, read the same data back, see the byte counters rise in `sysctl dev.myfirst.0.stats`, and watch your per-open destructor run when each descriptor closes. That is a complete, loadable, source-grounded driver, and the three stages you built in Chapter 9 are the foundation Chapter 10 is about to build on.

There is still something unsatisfying about that Stage 3 FIFO, however. It moves bytes correctly, but it does not use the buffer well. Once `bufhead` walks forward, the space it leaves behind is wasted until the buffer empties and `bufhead` collapses back to zero. A steady producer and a matching consumer can exhaust the capacity long before either side has truly run out of work. A full buffer returns `ENOSPC` immediately, even if a reader is a millisecond away from draining half of it. An empty buffer returns zero bytes as a pseudo end-of-file, even if the caller is prepared to wait. None of those behaviours are wrong for a teaching checkpoint, but none of them scale.

This chapter is where we make the I/O path efficient.

Real drivers move bytes in the background of other work. A reader that reaches the end of the current data may want to block until more arrives. A writer that finds the buffer full may want to block until a reader has made room. A non-blocking caller wants a clear `EAGAIN` instead of a polite fiction. A tool like `cat` or `dd` wants to read and write in chunks that match its own buffer size, not the driver's internal constraints. And the kernel wants the driver to do all of this without losing bytes, without corrupting shared state, and without the sprawling tangle of indices and edge cases that beginner drivers so often accumulate.

The way real drivers achieve this is through a handful of disciplined patterns. A **circular buffer** replaces the linear Stage 3 buffer and keeps the whole capacity in play. **Partial I/O** lets `read(2)` and `write(2)` return whatever portion of the request is actually available, instead of all-or-nothing. **Non-blocking mode** lets a well-written caller ask “is there any data yet?” without committing to sleep. And a careful **refactor** converts the buffer from an ad hoc set of fields inside the `softc` into a small, named abstraction that Chapter 11 can then protect with real synchronization primitives.

All of this sits firmly inside character-device territory. We are still writing a pseudo-device, still reading and writing through `struct uio`, still loading and unloading our module with `kldload(8)` and `kldunload(8)`. What changes is the *shape* of the data plane between the kernel buffer and the user program, and the quality of the promises the driver makes.

10.1.1 Why This Chapter Earns Its Own Place

It would be possible to shortcut this material. Many tutorials show a circular buffer in ten lines, sprinkle a `mtx_sleep` somewhere, and declare victory. That approach produces code that passes one test and then develops mysterious bugs under load. The mistakes are usually not in the read and write handlers themselves. They are in how the circular buffer wraps, how `uiomove(9)` is called when the live bytes span the physical end of the buffer, how `IO_NDELAY` is interpreted, how `selrecord(9)` and `selwakeup(9)` compose with a non-blocking caller that never calls `poll(2)`, and how partial writes should be reported back to a user-space program that is itself looping.

This chapter works through those details one at a time. The result is a driver you can stress with `dd`, bombard with a producer-consumer pair, inspect through `sysctl`, and hand to Chapter 11 as a stable base for concurrency work.

10.1.2 Where Chapter 9 Left the Driver

Checkpoint the state you should be working from. If your source tree and lab box match this outline, everything in Chapter 10 will land cleanly. If they do not, go back and bring Stage 3 to the shape below before continuing.

- One Newbus child under `nexus0`, created by `device_identify`.
- A `struct myfirst_softc` sized for the FIFO buffer plus statistics.
- A mutex `sc->mtx` named after the device, protecting `_softc` counters and buffer indices.
- A `sysctl` tree at `dev.myfirst.0.stats` exposing `attach_ticks`, `open_count`, `active_fhs`, `bytes_read`, `bytes_written`, and the current `bufhead`, `bufused`, and `buflen`.
- A primary cdev at `/dev/myfirst/0` with ownership `root:operator` and mode `0660`.
- An alias cdev at `/dev/myfirst` pointing at the primary.
- Per-open state via `devfs_set_cdevpriv(9)` and a `myfirst_fh_dtor` destructor.
- A linear FIFO buffer of `MYFIRST_BUFSIZE` bytes, with `bufhead` collapsing to zero on empty.
- `d_read` returning zero bytes when `bufused == 0` (our approximation of EOF at this stage).
- `d_write` returning `ENOSPC` when the tail reaches `buflen`.

Chapter 10 takes that driver and replaces the linear FIFO with a true circular buffer. Then it grows the `d_read` and `d_write` handlers to honour partial I/O properly, adds an `O_NONBLOCK`-aware path with `EAGAIN`, wires in a `d_poll` handler so that `select(2)` and `poll(2)` begin to work, and finishes by pulling the buffer logic into a named abstraction ready for locking.

10.1.3 What You Will Learn

Once you have finished this chapter, you will be able to:

- Explain, in plain language, what buffering buys a driver and where it starts to hurt.
- Design and implement a fixed-size byte-oriented circular buffer in kernel space.
- Reason about wrap-around correctly: detect it, split transfers across it, and keep the `uio` accounting honest while you do so.
- Integrate that circular buffer into the evolving `myfirst` driver without regressing any earlier behaviour.
- Handle partial reads and partial writes the way classical UNIX programs expect.

- Interpret and honour `IO_NDELAY` in your read and write handlers.
- Implement `d_poll` so `select(2)` and `poll(2)` work against `myfirst`.
- Stress-test buffered I/O from user space using `dd(1)`, `cat(1)`, `hexdump(1)`, and a small producer-consumer pair.
- Recognise the read-modify-write hazards the current driver contains, and make the refactors that will let Chapter 11 introduce real locking without restructuring code.
- Read `d_mmap(9)` and understand when a character-device driver wants to let user space map buffers directly, and when it genuinely does not.
- Talk about zero-copy in a way that distinguishes real savings from slogans.
- Recognise readahead and write-coalescing patterns when you see them in a driver, and describe why they matter to throughput.

10.1.4 What You Will Build

You will take the Stage 3 driver from Chapter 9 through four main stages, plus a short optional fifth stage that adds memory-mapping support.

1. **Stage 1, a standalone circular buffer.** Before touching the kernel, you will build `cbuf.c` and `cbuf.h` in userland, write a handful of small tests against them, and confirm that wrap-around, empty, and full behave the way your mental model says they should. This is the only part of the chapter you can develop entirely in userland, and it will pay for itself when the driver starts failing in ways that would have been caught by a three-line unit test.
2. **Stage 2, a circular-buffer driver.** You will splice the circular buffer into `myfirst` so that `d_read` and `d_write` now drive the new abstraction. `bufhead` becomes `cb_head`, `bufused` becomes `cb_used`, the fields live inside a small `struct cbuf`, and the wrap-around arithmetic is visible in one place. No behaviour visible to user space changes yet, but the driver is immediately better behaved under steady load.
3. **Stage 3, partial I/O and non-blocking support.** You will grow the handlers to honour partial reads and writes correctly, interpret `IO_NDELAY` as `EAGAIN`, and introduce the blocking read path with `mtx_sleep(9)` and `wakeup(9)`. The driver now rewards polite callers with low latency and rewards patient callers with blocking semantics.
4. **Stage 4, poll-aware and refactor-ready.** You will add a `d_poll` handler, wire up a `struct selinfo`, and refactor all buffer access through a tight set of helper functions so that Chapter 11 can drop in a real locking strategy.
5. **Stage 5, memory mapping (optional).** You will add a small `d_mmap` handler so user space can read the buffer through `mmap(2)`. This stage is explored alongside the supplementary topics in Section 8 and the matching lab at the end of the chapter. You can skip it on a first pass without losing the thread; it revisits the same buffer from a different angle.

You will exercise each stage with the base-system tools, with a small `cb_test` userland program you compile in Stage 1, and with two new helpers: `rw_myfirst_nb` (a non-blocking tester) and `producer_consumer` (a fork-based load harness). Each stage lives on disk under a dedicated directory in `examples/part-02/ch10-handling-io-efficiently/`, and the README there mirrors the chapter's checkpoints.

10.1.5 What This Chapter Does Not Cover

It is worth naming explicitly what we will *not* try to do here. The deeper discussions of these topics belong in later chapters, and dragging them in now would blur the lessons of this one.

- **Real concurrency correctness.** Chapter 10 uses the mutex that already exists in the softc, and it uses `mtx_sleep(9)` with that mutex as the sleep-interlock argument. That is safe by construction. But this chapter does not explore the full space of race conditions, nor does it categorize lock classes, nor does it teach the reader how to prove a piece of shared state is correctly protected. That is Chapter 11's job, and it is the reason the last section here is called "Refactoring and Preparing for Concurrency" rather than "Concurrency in Drivers."
- **`ioctl(2)`.** The driver still does not implement `d_ioctl`. A few clean-up primitives (flush the buffer, query its fill level) would fit nicely under `ioctl`, but Chapter 25 is the right place for that.
- **`kqueue(2)`.** This chapter implements `d_poll` for `select(2)` and `poll(2)`. The companion `d_kqfilter` handler, along with `knlist` machinery and `EVFILT_READ` filters, is introduced later alongside `taskqueue`-driven drivers.
- **Hardware mmap.** We will build a minimal `d_mmap` handler that lets user space map a preallocated kernel buffer as read-only pages, and we will discuss the design decisions and the things this pattern does and does not achieve. We will not venture into `bus_space(9)`, `bus_dmamap_create(9)`, or the `dev_pager` machinery; those are Part 4 and Part 5 material.
- **Real devices with real backpressure.** The backpressure model here is "the buffer has a fixed capacity and blocks or returns `EAGAIN` when full." Storage and network drivers have richer models (watermarks, credit, BIO queues, mbuf chains). Those details belong in their own chapters.

Keeping the chapter inside those lines is how we keep it honest. The material you will learn is enough to write a respectable pseudo-device and to read most character-device drivers in the tree with confidence.

10.1.6 Estimated Time Investment

- **Reading only:** roughly ninety minutes, maybe two hours if you are pausing at the diagrams.
- **Reading plus typing the four stages:** four to six hours, split into at least two sessions with a reboot or two.
- **Reading plus all labs and challenges:** eight to twelve hours over three sessions. The challenges are genuinely richer than the main labs, and they reward patient work.

As with Chapter 9, give yourself a fresh lab boot at the start. Do not rush the four stages. The value of the sequence is in watching the driver's behaviour change, one pattern at a time, as you add each capability.

10.1.7 Prerequisites

Before starting this chapter, confirm:

- Your driver source matches the Chapter 9 Stage 3 example under `examples/part-02/ch09-reading-and-writing/stage3-echo/`. If it does not, stop here and bring it up to that shape first. Chapter 10 assumes it.
- Your lab machine is running FreeBSD 14.3 with a matching `/usr/src`. The APIs and file layouts you will see in this chapter align with that release.

- You read Chapter 9 carefully, including Appendix E (the one-page cheat sheet). The “three-line spine” there for reads and writes is exactly what we are about to grow.
- You are comfortable loading and unloading your own module, watching `dmesg`, reading `sysctl dev.myfirst.0.stats`, and reading the output of `truss(1)` or `ktrace(1)` when a test surprises you.

If any of those are shaky, fixing them now is a better use of time than pushing through this chapter.

10.1.8 How to Get the Most Out of This Chapter

Three habits stay useful.

First, keep `/usr/src/sys/dev/evdev/cdev.c` open in a second terminal. It is one of the cleanest examples in the tree of a character device that implements a ring buffer, blocks callers when the buffer is empty, honours `O_NONBLOCK`, and wakes sleepers through both `wakeup(9)` and the `selinfo` machinery. We will point at it several times.

Second, keep `/usr/src/sys/kern/subr_uio.c` bookmarked. Chapter 9 walked through this file’s `uiomove` internals; here we will revisit it when the buffer wrap forces us to split a transfer. Reading the real code reinforces the right mental model.

Third, run your tests under `truss(1)` occasionally, not only under normal shells. Tracing the syscall return values is the fastest way to distinguish a driver that is honouring partial I/O from one that is silently throwing bytes away.

10.1.9 Roadmap Through the Chapter

The sections in order are:

1. What buffered I/O is and why it matters. Buckets and pipelines, unbuffered vs. buffered patterns, and where each fits in a driver.
2. Creating a circular buffer. The data structure, invariants, wrap-around arithmetic, and a standalone user-land implementation you will test before you put it in the kernel.
3. Integrating that circular buffer into `myfirst`. How `d_read` and `d_write` change, how to handle the split transfer around the wrap, and how to log buffer state so debugging does not require guesswork.
4. Partial reads and partial writes. What they are, why they are correct UNIX behaviour, how to report them through `uio_resid`, and the edge cases you must not trip over.
5. Non-blocking I/O. The `IO_NDELAY` flag, its relationship to `O_NONBLOCK`, the `EAGAIN` convention, and the design of a simple blocking read path using `mtx_sleep(9)` and `wakeup(9)`.
6. Testing buffered I/O from user space. A consolidated test kit: `dd`, `cat`, `hexdump`, `truss`, a small non-blocking tester, and a producer-consumer harness that fork a reader and a writer against `/dev/myfirst`.
7. Refactoring and preparing for concurrency. Where the current code is at risk, how to factor the buffer into helpers, and the shape you want to hand to Chapter 11.
8. Three supplementary topics: `d_mmap(9)` as a minimal kernel-memory mapping (the optional Stage 5), zero-copy considerations for pseudo-devices, and the patterns real high-throughput drivers use (readahead on the read side, write coalescing on the write side).
9. Hands-on labs, a set of concrete exercises you should be able to complete directly against the driver.
10. Challenge exercises that stretch the same skills without introducing brand-new foundations.

11. Troubleshooting notes for the classes of bug this chapter's patterns tend to produce.
12. Wrapping Up and a bridge to Chapter 11.

If this is your first pass, read linearly and do the labs in order. If you are revisiting the material to consolidate, each numbered supplementary topic and the troubleshooting section stand alone.

10.2 Section 1: What Buffered I/O Is and Why It Matters

Every driver that moves bytes between user space and a hardware-side or kernel-side data source has to decide *where* those bytes live in between. The Stage 3 driver from Chapter 9 already makes that decision implicitly. The buffered bytes that have been written but not yet read are sitting in a kernel buffer. The reader pulls them out from the head; the writer appends new ones at the tail. The `myfirst` driver is, in that sense, already buffered.

What changes in this chapter is not whether you have a buffer. It is *how* the buffer is shaped, *how much* of its capacity you can keep in play at once, and *what promises* the driver makes to its callers about that buffer's behaviour. Before we look at the data structure, it is worth pausing on the difference between unbuffered and buffered I/O at a conceptual level. That contrast will guide every decision the rest of the chapter asks you to make.

10.2.1 A Plain-Language Definition

In the simplest framing, **unbuffered I/O** means that every `read(2)` or `write(2)` call touches the underlying source or sink directly. There is no intermediate storage that absorbs bursts, no place where the producer can leave bytes for the consumer to pick up later, no way to decouple the rate at which bytes are produced from the rate at which they are consumed. Each call goes all the way down.

Buffered I/O, by contrast, places a small region of memory between the producer and the consumer. A writer drops bytes into the buffer; a reader picks them up. As long as the buffer has free space, writers do not need to wait. As long as the buffer has live bytes, readers do not need to wait. The buffer absorbs short-term mismatches between the two sides.

That sounds like a small distinction, but in driver code it is often the difference between something that works under load and something that does not.

It is worth pausing on a small but important wrinkle. The kernel itself buffers at several layers above and below your driver. The C library's `stdio` buffers writes before they reach the `write(2)` syscall. The VFS path buffers I/O on regular files in the buffer cache. The disk drivers down in Part 7 will buffer at the BIO and queue level. When this chapter says "buffered I/O", it means a buffer *inside the driver*, between the user-facing read and write handlers and whatever data source or sink the driver represents. We are not arguing about whether buffering exists at all; we are deciding where to put one more buffer, and what it should do.

10.2.2 Two Concrete Pictures

Picture an unbuffered pseudo-device first. Imagine a driver whose `d_write` immediately hands every byte to whatever upstream code consumes them. If the consumer is busy, the writer waits. If the consumer is fast, the writer flies through. There is no give in the system. A burst on either side translates directly into pressure on the other.

Now picture a buffered pseudo-device. The same `d_write` deposits bytes into a small buffer. The consumer pulls bytes out at its own pace. A short burst of writes can complete instantly because the buffer absorbs them. A short pause from the consumer does not stall the writer because the buffer holds the backlog. Both sides feel like they are running smoothly even though their rates do not match exactly at every instant.

The buffered case is what most useful drivers look like in practice. It is not magic; the buffer is finite, and once it fills the producer has to wait or back off. But it gives the system a place to tolerate normal variability, and that tolerance is what makes throughput predictable.

10.2.3 Buckets vs. Pipelines

A useful analogy here is the difference between carrying water in buckets and carrying water through a pipeline.

When you carry water in buckets, every transfer is a discrete event. You walk to the well, fill the bucket, walk back, empty the bucket, walk again. The producer (the well) and the consumer (the cistern) are coupled tightly through your two arms and your walking pace. If you trip, the system stops. If the cistern is busy, you wait at it. If the well is busy, you wait at it. Each handoff requires that both sides be ready at the same instant.

A pipeline replaces that coupling with a length of pipe. Water enters at the well end and exits at the cistern end. The pipe holds some quantity of water in flight at any moment. The producer can pump as long as there is room in the pipe. The consumer can drain as long as there is water in the pipe. Their schedules no longer have to match. They only have to match *on average*.

A driver buffer is exactly that pipe. It is a finite reservoir that decouples writer rate from reader rate, as long as both rates average out to something the buffer's capacity can absorb. The bucket model corresponds to unbuffered I/O. The pipeline model corresponds to buffered I/O. Both are valid in different situations, and a driver writer's job is to know which one to build.

10.2.4 Performance: System Calls and Context Switches

The performance advantages of buffering inside a driver are real but indirect. They come from three places.

The first place is **system call overhead**. Every `read(2)` or `write(2)` is a transition from user space to kernel space and back. That transition is cheap on a modern processor, but it is not free. A writer that calls `write(2)` once with a thousand bytes pays one transition. A writer that calls `write(2)` a thousand times with one byte each pays a thousand transitions. If the driver buffers internally, callers can comfortably issue larger reads and writes, and the per-syscall overhead becomes a smaller fraction of the total cost.

The second place is **context switch reduction**. A call that has to wait, because nothing is available right now, often results in the calling thread being suspended and another thread being scheduled. Each suspend and resume is more expensive than a system call. A buffer absorbs the brief mismatches that would otherwise force a sleep, and the threads on both sides keep running.

The third place is **batching opportunities**. A driver that knows it has thousands of bytes ready to ship can sometimes hand the whole batch downstream in one operation, where a driver that processes byte by byte would have to do the same setup and teardown work for each transfer. We will not see this directly with the pseudo-device in this chapter, but it is the underlying argument for the read coalescing and write coalescing patterns we will look at later in the chapter.

None of these advantages should be applied blindly. Buffering also adds latency, since a byte may sit in the buffer for some time before the consumer notices. It adds memory cost. It introduces a pair of indices that must be kept consistent under concurrent access. And it forces a set of design decisions about what to do when the buffer fills (block? drop? overwrite?) and what to do when it empties (block? short-read? signal end-of-file?). There is a reason this chapter spends real time on those decisions.

10.2.5 Buffered Data Transfers in Device Drivers

Where exactly does buffering pay off in a device driver?

The clearest case is a driver whose data source produces bursts. A serial port driver receives characters whenever the UART chip raises an interrupt; if the consumer is not currently reading, those characters need somewhere to live until it does. A keyboard driver collects key events in the interrupt handler and hands them to user space at whatever rate the application is willing to read. A network driver assembles packets in DMA buffers and feeds them to the protocol stack as soon as it can. In each case, the driver needs a place to hold incoming data between the moment it arrives and the moment it can be delivered.

The mirror case is a driver whose data sink absorbs bursts. A graphics driver may queue commands until the GPU is ready to process them. A printer driver may accept a document and dribble it out at the printer's pace. A storage driver may collect write requests and let an elevator algorithm reorder them for the disk. Again, the driver needs a place to hold outgoing data between the moment the user wrote it and the moment the device is ready.

The pseudo-device we are building in this book sits in the middle of those two patterns. There is no real hardware on either end, but the *shape* of the data path mirrors what real drivers do. When you write to `/dev/myfirst`, the bytes land in a buffer the driver owns. When you read, the bytes come out of that same buffer. Once the buffer is circular and the I/O handlers know how to honour partial transfers, you can stress the driver with `dd if=/dev/zero of=/dev/myfirst bs=1m count=10` from one terminal and `dd if=/dev/myfirst of=/dev/null bs=4k` from another, and the driver will behave the same way a real character device behaves under analogous load.

10.2.6 When to Use Buffered I/O in a Driver

Almost every driver wants some form of buffering. The interesting question is not whether to buffer, but at what *granularity* and with what *backpressure model*.

Granularity is about the size of the buffer relative to the rates on either side. A buffer that is too small fills constantly and forces the writer to wait, defeating the point. A buffer that is too large hides problems for too long, lets memory grow without bound, and increases worst-case latency. The right size depends on what the buffer is for: the buffer of an interactive keyboard driver only needs to hold a handful of recent events; the buffer of a network driver may hold thousands of packets at peak load.

Backpressure is about what to do when the buffer fills (or empties) and the calling pattern does not match what the driver wants. There are three common strategies, each appropriate in a different setting.

The first is **block**. When the buffer is full, the writer waits. When the buffer is empty, the reader waits. This is the classical UNIX semantics, and it is the right default for terminal devices, pipes, and most general-purpose pseudo-devices. We will implement blocking reads (and optional blocking writes) in Section 5.

The second is **drop**. When the buffer is full, the writer drops the byte (or marks an overflow event) and proceeds. When the buffer is empty, the reader sees zero bytes and proceeds. This is the right default for some real-time and high-rate scenarios, where waiting would do more harm than missing data. Loss must be observable, however, or the driver will silently corrupt the stream from the user's perspective.

The third is **overwrite**. When the buffer is full, the writer overwrites the oldest data with new data. When the buffer is empty, the reader sees zero bytes. This is the right default for a circular log of recent events: a `dmesg(8)`-like history where the most recent bytes are always preserved at the cost of the oldest.

The driver in this chapter uses **block** for blocking-mode callers and **EAGAIN** for non-blocking-mode callers, with no overwrite path. That is the most common pattern in the FreeBSD source tree and the easiest to reason about. The other two strategies appear in later chapters when their use cases come up naturally.

10.2.7 A First Look at the Cost of an Unbuffered Driver

It is worth being concrete about why the Stage 3 driver from Chapter 9 already starts to hurt at scale.

Suppose you are running a `dd` that writes 64-byte blocks at high rate into the driver, and a parallel `dd` that reads 64-byte blocks out of it. With the Stage 3 buffer of 4096 bytes, you have at most 64 in-flight blocks before the writer hits `ENOSPC` and stops. If the reader pauses for any reason (page faulting in its destination buffer, getting preempted by the scheduler, being moved to a different CPU), the writer parks immediately. As soon as the reader resumes and drains a single block, the writer can fit one more. The overall throughput is the *minimum* of what the two halves achieve in lockstep, plus a steady stream of `ENOSPC` errors that user-space programs are not expected to see.

A circular buffer of the same size holds the same number of in-flight blocks but never wastes the trailing capacity. A non-blocking writer that hits a full buffer cleanly receives `EAGAIN` (the conventional “try again later” signal) instead of `ENOSPC` (the conventional “this device is out of room”), so a tool like `dd` can decide whether to retry or back off. A blocking writer that hits a full buffer goes to sleep on a clear condition variable and wakes up the moment a reader frees space. Each of those changes is small. Together they make a driver feel responsive instead of brittle.

10.2.8 Where We Are Going

You now have the conceptual ground. The rest of the chapter will translate it into code. Section 2 walks through the data structure, with diagrams and a userland implementation you can test before you trust it inside the kernel. Section 3 moves the implementation into the driver, replacing the linear FIFO. Sections 4 and 5 grow the I/O path so that partial transfers and non-blocking semantics work the way users expect. Section 6 builds the test harness you will use through the rest of Part 2 and most of Part 3. Section 7 prepares the code for the locking work that defines Chapter 11.

The order matters. Each section assumes the changes from the previous one are in place. If you skip ahead, you will land in code that does not compile or behaves in surprising ways. As always, the slow path is the fast path.

10.2.9 Wrapping Up Section 1

We named the difference between unbuffered and buffered I/O, and we named the costs and benefits of each. We picked an analogy (buckets vs. pipeline) that we can keep returning to. We discussed where buffering pays off in driver code, what backpressure strategies are common, and which one we are committing to for the rest of the chapter. And we set the stage for the data structure that powers the whole thing: the circular buffer.

If you are not yet sure which backpressure strategy your driver should use, that is fine. The default we will build, “block in the kernel and EAGAIN outside it,” is the safe and conventional choice for general-purpose pseudo-devices, and you will have a clean code shape to revisit if you need a different strategy later. We are about to make that buffer real.

10.3 Section 2: Creating a Circular Buffer

A circular buffer is one of those data structures whose idea is older than the operating systems we now use. It shows up in serial chips, in audio sample queues, in network receive paths, in keyboard event queues, in trace buffers, in `dmesg(8)`, in `printf(3)` libraries, and in nearly every other place where one piece of code wants to leave bytes for another piece of code to pick up later. The structure is simple. The implementation is short. The bugs that beginners make in it are predictable. We will build it once, in userland, with care, and then carry the verified version into the driver in Section 3.

10.3.1 What a Circular Buffer Is

A linear buffer is the simplest thing that could possibly work: a region of memory plus a “next free byte” index. You write into it from the start and you stop when you reach the end. Once it fills, you either grow it, copy it, or stop accepting new data.

A circular buffer (also called a ring buffer) is the same region of memory, but with a twist in how indices behave. There are two indices: the *head*, which points at the next byte to be read, and the *tail*, which points at the next byte to be written. When either index reaches the end of the underlying memory, it wraps back to the start. The buffer is treated as if its first byte were adjacent to its last byte, forming a closed loop.

Two derived counts matter for using the structure correctly. The number of *live* bytes (how many are currently stored) is what readers care about. The number of *free* bytes (how much capacity is unused) is what writers care about. Both counts can be derived from the head and tail, plus the total capacity, with a small piece of arithmetic.

Visually, the structure looks like this when it is partly full and the live region does not wrap:

```
+---+---+---+---+---+---+---+
| _ | _ | A | B | C | D | _ | _ |
+---+---+---+---+---+---+---+
          ^               ^
        head            tail

capacity = 8, used = 4, free = 4
```

After enough writes, the tail catches up to the end of the underlying memory and wraps to the start. Now the live region itself wraps:

```

+---+---+---+---+---+---+---+---+
| F | G | _ | _ | _ | _ | D | E |
+---+---+---+---+---+---+---+
          ^           ^
        tail        head

```

```
capacity = 8, used = 4, free = 4
```

```
live region: head -> end of buffer, then start of buffer -> tail
```

The “live region wraps” case is the one that catches beginners. A naive bcopy of the live data treats the buffer as if it were linear; the bytes you copy are not the bytes you wanted. The correct way to handle this case is to perform the transfer in *two pieces*: from head to the end of the buffer, and then from the start of the buffer to tail. We will codify exactly this pattern in the helpers below.

10.3.2 Managing Read and Write Pointers

The head and tail pointers (we will call them indices, because they are integer offsets into a fixed-size array) follow simple rules.

When you read n bytes, the head advances by n , modulo the capacity. When you write n bytes, the tail advances by n , modulo the capacity. Reads remove bytes; the live count drops by n . Writes add bytes; the live count rises by n .

The interesting question is how to detect the two boundary conditions: empty and full. With only head and tail, the structure is *almost* enough on its own. If $\text{head} == \text{tail}$, the buffer could be empty (no bytes stored) or full (all capacity stored). The two states look identical from the indices alone. Implementations resolve the ambiguity in one of three ways.

The first way is to keep a separate **count** of live bytes. With used available, $\text{used} == 0$ is unambiguously empty and $\text{used} == \text{capacity}$ is unambiguously full. The structure is slightly larger but the code is short and obvious. This is the design we will use in this chapter.

The second way is to **always leave one byte unused**. With this rule, $\text{head} == \text{tail}$ always means empty, and the buffer is full when $(\text{tail} + 1) \% \text{capacity} == \text{head}$. The structure is one byte smaller and the code does not need a used field, but every transfer involves an off-by-one that is easy to get wrong. This is the design used in some classic embedded code; it is fine, but it offers no real advantage in our setting.

The third way is to use **monotonically increasing indices** that are never reduced mod capacity, and to compute the live count as $\text{tail} - \text{head}$. Wrap-around is then a function of how you index into the array ($\text{tail} \% \text{capacity}$), not how you advance the pointer. This is the design used by `buf_ring(9)` in the FreeBSD kernel, which uses a 32-bit counter and trusts the wrap to behave. It is elegant, but it complicates atomic operations and debugging. We will not use it; the explicit used field is the right tradeoff for a teaching driver.

10.3.3 Detecting Buffer Full and Buffer Empty

With the explicit used count, the boundary checks become trivial:

- **Empty:** `cb->cb_used == 0`. There are no bytes available to read.
- **Full:** `cb->cb_used == cb->cb_size`. There is no space available to write.

The arithmetic for the head and tail indices is also straightforward:

- After reading `n` bytes: `cb->cb_head = (cb->cb_head + n) % cb->cb_size; cb->cb_used -= n;`
- After writing `n` bytes: `cb->cb_used += n;` (the tail is computed when needed as `(cb->cb_head + cb->cb_used) % cb->cb_size`)

We will keep the tail implicit, derived from head and used. Some implementations track the tail explicitly. Either choice works as long as you commit to it consistently. With `tail` implicit, we never have to update two indices on a single operation, which removes one whole class of bug.

Two helper-derived quantities will appear repeatedly:

- `cb_free(cb) = cb->cb_size - cb->cb_used`: how many bytes can still be written before the buffer is full.
- `cb_tail(cb) = (cb->cb_head + cb->cb_used) % cb->cb_size`: where the next write should land.

Both are pure functions of the head and used count and the capacity. They have no side effects and are safe to call at any time.

10.3.4 Allocating a Fixed-Size Circular Buffer

The buffer needs three pieces of state and one chunk of backing memory. Here is the structure we will use:

```
struct cbuf {
    char    *cb_data;        /* backing storage, cb_size bytes */
    size_t  cb_size;         /* total capacity, in bytes */
    size_t  cb_head;         /* index of next byte to read */
    size_t  cb_used;         /* count of live bytes */
};
```

Three lifetime functions cover the basics:

```
int  cbuf_init(struct cbuf *cb, size_t size);
void cbuf_destroy(struct cbuf *cb);
void cbuf_reset(struct cbuf *cb);
```

`cbuf_init` allocates the backing storage, initialises the indices, and returns zero on success or a positive `errno` on failure. `cbuf_destroy` releases the backing storage and zeros the structure. `cbuf_reset` empties the buffer without freeing memory; both indices return to zero.

Three accessor functions give the rest of the code the boundary information it needs:

```

size_t cbuf_used(const struct cbuf *cb);
size_t cbuf_free(const struct cbuf *cb);
size_t cbuf_size(const struct cbuf *cb);

```

These are tiny inline-friendly functions. They do not lock anything; the caller is expected to hold whatever synchronization the larger system requires. (In Stage 4 of this chapter, the driver's mutex will provide that synchronization.)

The two interesting functions are the byte-moving primitives:

```

size_t cbuf_write(struct cbuf *cb, const void *src, size_t n);
size_t cbuf_read(struct cbuf *cb, void *dst, size_t n);

```

`cbuf_write` copies up to `n` bytes from `src` into the buffer and returns the number actually copied. `cbuf_read` copies up to `n` bytes from the buffer into `dst` and returns the number actually copied. Both functions handle the wrap-around case internally. The caller provides a contiguous source or destination; the buffer takes care of splitting the transfer when the live region or free region crosses the end of the underlying storage.

That signature deserves a moment of attention. Notice that the functions return `size_t`, not `int`. They report progress, not error. Returning fewer bytes than requested is *not* an error condition; it is the correct way to express that the buffer was full (for writes) or empty (for reads). This mirrors the way `read(2)` and `write(2)` themselves work: a positive return value smaller than the request is a “partial transfer”, not a failure. We will lean on this in Section 4 when we make the driver honour partial I/O properly.

10.3.5 A Walkthrough of the Wrap-Around

The wrap-around logic is short, but it is worth tracing one example carefully. Suppose the buffer has capacity 8, head is 6, and used is 4. The live bytes are stored at positions 6, 7, 0, 1.

```

+---+---+---+---+---+---+---+---+
| C | D | _ | _ | _ | _ | A | B |
+---+---+---+---+---+---+---+
      ^           ^
    tail       head

```

capacity = 8, used = 4, head = 6, tail = (6+4)%8 = 2

Now the caller asks for 3 bytes via `cbuf_read`. The function does the following:

1. Compute `n = MIN(3, used) = MIN(3, 4) = 3`. The caller will get up to 3 bytes.
2. Compute `first = MIN(n, capacity - head) = MIN(3, 8 - 6) = 2`. That is the contiguous chunk starting at the head.
3. Copy `first = 2` bytes from `cb_data + 6` into `dst`. Those are A and B.
4. Compute `second = n - first = 1`. That is the part of the transfer that has to come from the start of the buffer.
5. Copy `second = 1` byte from `cb_data + 0` into `dst + 2`. That is C.

6. Advance `cb_head = (6 + 3) % 8 = 1`. Decrement `cb_used` by 3, leaving 1.

The caller's destination now holds A, B, C. The buffer state has D as its only live byte, with `head = 1` and `used = 1`. The next read will return D from position 1.

The same logic applies to `cbuf_write`, with `tail` taking the role of `head`. The function computes `tail = (head + used) % capacity`, then `first = MIN(n, capacity - tail)`, copies first bytes from `src` to `cb_data + tail`, then copies the remainder from `src + first` to `cb_data + 0`, and advances `cb_used` by the total written.

There is exactly one step in either function that wraps. Either the destination wraps (in a write) or the source wraps (in a read), but never both. This is the key property that makes the implementation manageable: the buffer's wrap is a property of the internal data, not of the caller's data, so the caller's source and destination are always treated as ordinary contiguous memory.

10.3.6 Avoiding Overwrites and Data Loss

A common beginner mistake is to make `cbuf_write` overwrite older data when the buffer fills, on the theory that “newer data is more important.” That is sometimes the right policy, as we noted in Section 1, but it must be a *deliberate* design choice and it must be visible to the caller, not a silent mutation of state. The conventional default is that `cbuf_write` returns the number of bytes it actually wrote, and the caller is expected to look at the return value.

The same goes for `cbuf_read`: when the buffer is empty, `cbuf_read` returns zero. The caller is expected to interpret zero as “no bytes available right now”, not as an error. Coupling that signal with the driver's `EAGAIN` or with a blocking sleep is the job of the I/O handler, not the buffer itself.

If you ever want a circular buffer with overwrite semantics (a `dmesg`-style log, for instance), the cleanest approach is to write a separate `cbuf_overwrite` function and leave `cbuf_write` strict. Two separate names mean two separate intentions, and a future reader of the code does not have to guess which behaviour is in effect.

10.3.7 Implementing It in Userland

The right way to learn this structure is to type it once in userland and run it through a few small tests, before you ask the kernel to trust it. The same source can then move into the kernel module almost unchanged, except for the allocation and free calls.

Below is the userland source. It lives in `examples/part-02/ch10-handling-io-efficiently/cbuf-userland/`.

`cbuf.h`:

```
/* cbuf.h: a fixed-size byte-oriented circular buffer. */
#ifndef CBUF_H
#define CBUF_H

#include <stddef.h>
```

```
struct cbuf {
    char    *cb_data;
    size_t   cb_size;
    size_t   cb_head;
    size_t   cb_used;
};

int         cbuf_init(struct cbuf *cb, size_t size);
void        cbuf_destroy(struct cbuf *cb);
void        cbuf_reset(struct cbuf *cb);

size_t      cbuf_size(const struct cbuf *cb);
size_t      cbuf_used(const struct cbuf *cb);
size_t      cbuf_free(const struct cbuf *cb);

size_t      cbuf_write(struct cbuf *cb, const void *src, size_t n);
size_t      cbuf_read(struct cbuf *cb, void *dst, size_t n);

#endif /* CBUF_H */
```

cbuf.c:

```
/* cbuf.c: userland implementation of the byte-oriented ring buffer. */
#include "cbuf.h"

#include <errno.h>
#include <stdlib.h>
#include <string.h>

#ifndef MIN
#define MIN(a, b) (((a) < (b)) ? (a) : (b))
#endif

int
cbuf_init(struct cbuf *cb, size_t size)
{
    if (cb == NULL || size == 0)
        return (EINVAL);
    cb->cb_data = malloc(size);
    if (cb->cb_data == NULL)
        return (ENOMEM);
    cb->cb_size = size;
    cb->cb_head = 0;
    cb->cb_used = 0;
}
```

```
        return (0);
    }

void
cbuf_destroy(struct cbuf *cb)
{
    if (cb == NULL)
        return;
    free(cb->cb_data);
    cb->cb_data = NULL;
    cb->cb_size = 0;
    cb->cb_head = 0;
    cb->cb_used = 0;
}

void
cbuf_reset(struct cbuf *cb)
{
    if (cb == NULL)
        return;
    cb->cb_head = 0;
    cb->cb_used = 0;
}

size_t
cbuf_size(const struct cbuf *cb)
{
    return (cb->cb_size);
}

size_t
cbuf_used(const struct cbuf *cb)
{
    return (cb->cb_used);
}

size_t
cbuf_free(const struct cbuf *cb)
{
    return (cb->cb_size - cb->cb_used);
}

size_t
cbuf_write(struct cbuf *cb, const void *src, size_t n)
```

```
{
    size_t avail, tail, first, second;

    avail = cbuf_free(cb);
    if (n > avail)
        n = avail;
    if (n == 0)
        return (0);

    tail = (cb->cb_head + cb->cb_used) % cb->cb_size;
    first = MIN(n, cb->cb_size - tail);
    memcpy(cb->cb_data + tail, src, first);
    second = n - first;
    if (second > 0)
        memcpy(cb->cb_data, (const char *)src + first, second);

    cb->cb_used += n;
    return (n);
}

size_t
cbuf_read(struct cbuf *cb, void *dst, size_t n)
{
    size_t first, second;

    if (n > cb->cb_used)
        n = cb->cb_used;
    if (n == 0)
        return (0);

    first = MIN(n, cb->cb_size - cb->cb_head);
    memcpy(dst, cb->cb_data + cb->cb_head, first);
    second = n - first;
    if (second > 0)
        memcpy((char *)dst + first, cb->cb_data, second);

    cb->cb_head = (cb->cb_head + n) % cb->cb_size;
    cb->cb_used -= n;
    return (n);
}
```

Two things in this code are worth noticing.

First, both `cbuf_write` and `cbuf_read` clamp `n` to the available space or live data *before* doing any copying. This is the key to the partial-transfer semantics: the function is happy to do less work than asked, and it tells the

caller exactly how much it did. There is no error path for “buffer is full”, because that is not an error.

Second, the second `> 0` guard around the second `memcpy` is not strictly necessary (`memcpy(dst, src, 0)` is well-defined and does nothing), but it makes the wrap-around reasoning visible at a glance. A future reader can tell that the second copy is conditional and that the wrap case is handled.

10.3.8 A Tiny Test Program

The companion `cb_test.c` exercises the structure with a small but meaningful set of cases. It is short enough to read in full:

```
/* cb_test.c: simple sanity tests for the cbuf userland implementation. */
#include "cbuf.h"

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define CHECK(cond, msg) \
    do { if (!(cond)) { fprintf(stderr, "FAIL: %s\n", msg); exit(1); } } while (0)

static void
test_basic(void)
{
    struct cbuf cb;
    char in[8] = "ABCDEFGH";
    char out[8] = {0};
    size_t n;

    CHECK(cbuf_init(&cb, 8) == 0, "init");
    CHECK(cbuf_used(&cb) == 0, "init used");
    CHECK(cbuf_free(&cb) == 8, "init free");

    n = cbuf_write(&cb, in, 4);
    CHECK(n == 4, "write 4");
    CHECK(cbuf_used(&cb) == 4, "used after write 4");

    n = cbuf_read(&cb, out, 2);
    CHECK(n == 2, "read 2");
    CHECK(memcmp(out, "AB", 2) == 0, "AB content");
    CHECK(cbuf_used(&cb) == 2, "used after read 2");

    cbuf_destroy(&cb);
    printf("test_basic OK\n");
}
```

```
static void
test_wrap(void)
{
    struct cbuf cb;
    char in[8] = "ABCDEFGH";
    char out[8] = {0};
    size_t n;

    CHECK(cbuf_init(&cb, 8) == 0, "init");

    /* Push head forward by writing and reading 6 bytes. */
    n = cbuf_write(&cb, in, 6);
    CHECK(n == 6, "write 6");
    n = cbuf_read(&cb, out, 6);
    CHECK(n == 6, "read 6");

    /* Now write 6 more, which should wrap. */
    n = cbuf_write(&cb, in, 6);
    CHECK(n == 6, "write 6 after wrap");
    CHECK(cbuf_used(&cb) == 6, "used after wrap write");

    /* Read all of it back; should return ABCDEF. */
    memset(out, 0, sizeof(out));
    n = cbuf_read(&cb, out, 6);
    CHECK(n == 6, "read 6 after wrap");
    CHECK(memcmp(out, "ABCDEF", 6) == 0, "content after wrap");
    CHECK(cbuf_used(&cb) == 0, "empty after drain");

    cbuf_destroy(&cb);
    printf("test_wrap OK\n");
}

static void
test_partial(void)
{
    struct cbuf cb;
    char in[8] = "12345678";
    char out[8] = {0};
    size_t n;

    CHECK(cbuf_init(&cb, 4) == 0, "init small");

    n = cbuf_write(&cb, in, 8);
```

```
    CHECK(n == 4, "write clamps to free space");
    CHECK(cbuf_used(&cb) == 4, "buffer full");

    n = cbuf_read(&cb, out, 8);
    CHECK(n == 4, "read clamps to live data");
    CHECK(memcmp(out, "1234", 4) == 0, "content of partial");
    CHECK(cbuf_used(&cb) == 0, "buffer empty after partial drain");

    cbuf_destroy(&cb);
    printf("test_partial OK\n");
}

int
main(void)
{
    test_basic();
    test_wrap();
    test_partial();
    printf("all tests OK\n");
    return (0);
}
```

Compile and run with:

```
$ cc -Wall -Wextra -o cb_test cbuf.c cb_test.c
$ ./cb_test
test_basic OK
test_wrap OK
test_partial OK
all tests OK
```

The three tests cover the cases that matter: a basic write/read cycle, a write that wraps around the end of the buffer, and a partial transfer that hits the capacity boundary. They are not exhaustive, but they are enough to catch the most common implementation mistakes. The challenge exercises at the end of the chapter ask you to extend them.

10.3.9 Why Build It in Userland First

It might feel like a detour to write the buffer in userland when you know the driver is what matters. Three reasons make the detour worth it.

First, the kernel is a hostile place to debug. A bug in the kernel buffer can lock the machine, panic the kernel, or quietly corrupt unrelated state. A bug in the same code in userland is just a failed test that prints a friendly message.

Second, the kernel-side and userland-side implementations of this buffer are virtually identical. The only differences are the allocation primitive (`malloc(9)` with `M_DEVBUF` and `M_WAITOK | M_ZERO` versus `libc malloc(3)`) and the free primitive (`free(9)` versus `libc free(3)`). Once the userland version is correct, the kernel version is almost a copy-paste with a small adjustment.

Third, building the buffer once in isolation forces you to think about its API in calm conditions. By the time you are ready to splice it into the driver, you already know what `cbuf_write` returns, what `cbuf_read` returns, what `cbuf_used` means, and how wrap-around is supposed to work. None of that has to be relearned in the middle of a kernel session.

10.3.10 Things That Can Still Go Wrong

Even with the helpers above, there are a few mistakes worth flagging now so you do not make them in Section 3.

The first is **forgetting to clamp the request to the available space**. If `cbuf_write` is called with `n = 100` on a buffer with `free = 30`, the function returns 30, not 100. Callers must check the return value and act accordingly. The driver's `d_write` will translate this into a partial write by leaving `uio_resid` at the unconsumed amount. We will be very explicit about this in Section 4.

The second is **forgetting that `cbuf_used` and `cbuf_free` can change between two checks**. In single-threaded userland tests this is impossible. In the kernel, a different thread can modify the buffer between any two function calls if no lock is held. Section 3 holds the softc mutex around all buffer access; Section 7 explains why.

The third is **mixing up indices**. Some implementations track the tail explicitly and the count implicitly. Others do the reverse. Both work. Mixing the two on a single buffer does not. Pick one and stick with it. We pick “head and used”; the tail is always derived.

The fourth is **integer wrap of the indices themselves**. With `size_t` and a buffer that is a few thousand bytes, the indices can never exceed `cb_size`, and `(cb_head + n) % cb_size` is always well-defined. If you ever extend this code to a buffer larger than `SIZE_MAX / 2`, that is no longer true; you would need 64-bit indices and explicit modular arithmetic. For our pseudo-device with a 4 KB or 64 KB buffer, the basic structure is more than sufficient.

10.3.11 Wrapping Up Section 2

You now have a clean, tested, byte-oriented circular buffer. It clamps requests to the available space, reports the actual transfer size, and handles wrap-around in the only place where wrap-around is meaningful: inside the buffer itself. The userland tests give you a small piece of evidence that the implementation behaves the way the diagrams said it should.

Section 3 takes this code into the kernel. The shape stays almost identical; the allocation and synchronization change. By the end of the next section, your driver's `d_read` and `d_write` will be calling `cbuf_read` and `cbuf_write` instead of doing their own arithmetic, and the logic that used to live inline in `myfirst.c` will have a name.

10.4 Section 3: Integrating a Circular Buffer into Your Driver

The userland implementation of `cbuf` is the same code you are about to drop into the kernel. Almost. There are three small changes: the allocator, the deallocator, and a paranoia level that the kernel demands which userland does not. After the splice, the driver's read and write handlers shrink considerably, and the wrap-around arithmetic disappears from `myfirst.c` into the helpers where it belongs.

This section walks through the splice carefully. We will start with the kernel-side variant of the buffer, then move to the integration changes inside `myfirst.c`, and finally look at how to add a few `sysctl` knobs that make the driver's internal state visible while you are debugging.

10.4.1 Moving `cbuf` Into the Kernel

The kernel-side header `cbuf.h` is identical to the userland one:

```
#ifndef CBUF_H
#define CBUF_H

#include <sys/types.h>

struct cbuf {
    char    *cb_data;
    size_t   cb_size;
    size_t   cb_head;
    size_t   cb_used;
};

int    cbuf_init(struct cbuf *cb, size_t size);
void    cbuf_destroy(struct cbuf *cb);
void    cbuf_reset(struct cbuf *cb);

size_t   cbuf_size(const struct cbuf *cb);
size_t   cbuf_used(const struct cbuf *cb);
size_t   cbuf_free(const struct cbuf *cb);

size_t   cbuf_write(struct cbuf *cb, const void *src, size_t n);
size_t   cbuf_read(struct cbuf *cb, void *dst, size_t n);

#endif /* CBUF_H */
```

The kernel-side `cbuf.c` is almost a copy of the userland file with two replacements. `malloc(3)` becomes `malloc(9)` from `M_DEVBUF` with the `M_WAITOK | M_ZERO` flags. `free(3)` becomes `free(9)` from `M_DEVBUF`. The `memcpy(3)` calls remain valid in kernel context: the kernel has its own `memcpy` and `bcopy` symbols. Here is the full kernel version:

```
#include <sys/param.h>
#include <sys/kernel.h>
#include <sys/system.h>
#include <sys/malloc.h>

#include "cbuf.h"

MALLOC_DEFINE(M_CBUF, "cbuf", "Chapter 10 circular buffer");

int
cbuf_init(struct cbuf *cb, size_t size)
{
    if (cb == NULL || size == 0)
        return (EINVAL);
    cb->cb_data = malloc(size, M_CBUF, M_WAITOK | M_ZERO);
    cb->cb_size = size;
    cb->cb_head = 0;
    cb->cb_used = 0;
    return (0);
}

void
cbuf_destroy(struct cbuf *cb)
{
    if (cb == NULL || cb->cb_data == NULL)
        return;
    free(cb->cb_data, M_CBUF);
    cb->cb_data = NULL;
    cb->cb_size = 0;
    cb->cb_head = 0;
    cb->cb_used = 0;
}

void
cbuf_reset(struct cbuf *cb)
{
    if (cb == NULL)
        return;
    cb->cb_head = 0;
    cb->cb_used = 0;
}

size_t
cbuf_size(const struct cbuf *cb)
```

```
{
    return (cb->cb_size);
}

size_t
cbuf_used(const struct cbuf *cb)
{
    return (cb->cb_used);
}

size_t
cbuf_free(const struct cbuf *cb)
{
    return (cb->cb_size - cb->cb_used);
}

size_t
cbuf_write(struct cbuf *cb, const void *src, size_t n)
{
    size_t avail, tail, first, second;

    avail = cbuf_free(cb);
    if (n > avail)
        n = avail;
    if (n == 0)
        return (0);

    tail = (cb->cb_head + cb->cb_used) % cb->cb_size;
    first = MIN(n, cb->cb_size - tail);
    memcpy(cb->cb_data + tail, src, first);
    second = n - first;
    if (second > 0)
        memcpy(cb->cb_data, (const char *)src + first, second);

    cb->cb_used += n;
    return (n);
}

size_t
cbuf_read(struct cbuf *cb, void *dst, size_t n)
{
    size_t first, second;

    if (n > cb->cb_used)
```

```

        n = cb->cb_used;
    if (n == 0)
        return (0);

    first = MIN(n, cb->cb_size - cb->cb_head);
    memcpy(dst, cb->cb_data + cb->cb_head, first);
    second = n - first;
    if (second > 0)
        memcpy((char *)dst + first, cb->cb_data, second);

    cb->cb_head = (cb->cb_head + n) % cb->cb_size;
    cb->cb_used -= n;
    return (n);
}

```

Three points are worth a short comment.

The first is `MALLOC_DEFINE(M_CBUF, "cbuf", ...)`. This declares a private memory tag for the buffer's allocations so that `vmstat -m` can show how much memory the `cbuf` code is using, separately from the rest of the driver. We declare it once, in `cbuf.c`, with internal linkage to the rest of the module. The driver's `softc` still uses `M_DEVBUF`. The two tags can coexist; they are bookkeeping labels, not pools.

The second is the `M_WAITOK` flag. Because we never call `cbuf_init` from interrupt context (we call it from `myfirst_attach`, which runs in normal kernel thread context during module load), it is safe to wait for memory if the system is briefly low. With `M_WAITOK`, `malloc(9)` will not return `NULL`; if the allocation cannot proceed it will sleep until it can. We therefore do not need to test the result for `NULL`. If we ever want to call `cbuf_init` from a context where sleep is forbidden, we would need to switch to `M_NOWAIT` and handle a possible `NULL`. For Chapter 10's purposes, `M_WAITOK` is the right choice.

The third is that **the kernel `cbuf` does not lock**. It is a pure data structure. The locking strategy is the *caller's* responsibility. Inside `myfirst.c`, we will hold `sc->mtx` across every call into `cbuf`. That keeps the abstraction small and gives Chapter 11 a clean refactor target.

10.4.2 What Changes in `myfirst.c`

Bring up the Stage 3 file from Chapter 9 in your editor. The integration involves the following changes:

1. Replace the four buffer-related `softc` fields (`buf`, `buflen`, `bufhead`, `bufused`) with a single struct `cbuf` cb member.
2. Remove the `MYFIRST_BUFSIZE` macro from `myfirst.c` (we keep it but in a single header to avoid duplication).
3. Initialise the buffer in `myfirst_attach` with `cbuf_init`.
4. Tear it down in `myfirst_detach` and on the attach failure path with `cbuf_destroy`.
5. Rewrite `myfirst_read` to call `cbuf_read` against a stack-resident bounce buffer, then `uiomove` the bounce buffer out.
6. Rewrite `myfirst_write` to `uiomove` into a stack-resident bounce buffer and then `cbuf_write` into the ring.

The last two changes deserve a short discussion before we look at the code. Why a bounce buffer? Why not call `uiomove` directly against the `cbuf` storage?

The answer is that `uiomove` does not understand wrap-around. It expects a contiguous destination (for reads) or a contiguous source (for writes). If the live region of our circular buffer wraps, calling `uiomove(cb->cb_data + cb->cb_head, n, uio)` would copy past the end of the underlying memory and into whatever is allocated next. That is a heap corruption bug waiting to happen. Two safe shapes exist; you can pick either.

The first safe shape is to call `uiomove` *twice*, once for each side of the wrap. The driver computes the contiguous chunk available at `cb->cb_data + cb->cb_head`, calls `uiomove` for that chunk, and then calls `uiomove` again for the wrapped portion at `cb->cb_data + 0`. This is efficient because there is no extra copy. It is also more complex and harder to get right; the driver has to do partial accounting of `uio_resid` between the two `uiomove` calls, and any cancellation in the middle (signal, page fault) leaves the buffer in a partially drained state.

The second safe shape is to use a kernel-side **bounce buffer**: a small temporary on the stack that lives only for the duration of the I/O call. The driver reads bytes out of the `cbuf` into the bounce buffer with `cbuf_read`, then `uiomoves` the bounce buffer into user space. On the write side, it `uiomoves` from user space into the bounce buffer, then `cbuf_writes` the bounce buffer into the `cbuf`. The cost is one extra in-kernel copy per chunk; the benefit is simplicity, locality of error handling, and the ability to do all wrap-aware logic inside the `cbuf` where it belongs.

The bounce buffer approach is what we will use in this chapter. It is the same approach that drivers like `evdev/cdev.c` use (with `bcopy` between the per-client ring and a stack-resident event structure, before `uiomove`-ing the structure to user space). The stack-resident bounce is small (256 or 512 bytes is plenty), the loop runs as many times as the user's transfer size demands, and each iteration is independently restartable if `uiomove` fails. The performance cost is negligible for everything except very-high-throughput hardware drivers, and even there the trade-off is usually worth it for the readability gain.

10.4.3 The Stage 2 Driver: Handlers Refactored

Here is what the relevant pieces of the driver look like after the splice. The full source is in `examples/part-02/ch10-handling-io-efficiently/stage2-circular/myfirst.c`. We will show the I/O handlers inline, then walk through what changed.

```
#define MYFIRST_BUFSIZE      4096
#define MYFIRST_BOUNCE      256

struct myfirst_softc {
    device_t      dev;
    int           unit;

    struct mtx     mtx;

    uint64_t      attach_ticks;
    uint64_t      open_count;
    uint64_t      bytes_read;
    uint64_t      bytes_written;
```

```
int                active_fhs;
int                is_attached;

struct cbuf        cb;

struct cdev        *cdev;
struct cdev        *cdev_alias;

struct sysctl_ctx_list sysctl_ctx;
struct sysctl_oid   *sysctl_tree;
};

static int
myfirst_read(struct cdev *dev, struct uio *uio, int ioflag)
{
    struct myfirst_softc *sc = dev->si_drv1;
    struct myfirst_fh *fh;
    char bounce[MYFIRST_BOUNCE];
    size_t take, got;
    int error;

    error = devfs_get_cdevpriv((void **)&fh);
    if (error != 0)
        return (error);
    if (sc == NULL || !sc->is_attached)
        return (ENXIO);

    while (uio->uio_resid > 0) {
        mtx_lock(&sc->mtx);
        take = MIN((size_t)uio->uio_resid, sizeof(bounce));
        got = cbuf_read(&sc->cb, bounce, take);
        if (got == 0) {
            mtx_unlock(&sc->mtx);
            break;          /* empty: short read or EOF */
        }
        sc->bytes_read += got;
        fh->reads += got;
        mtx_unlock(&sc->mtx);

        error = uiomove(bounce, got, uio);
        if (error != 0)
            return (error);
    }
}
```

```
        return (0);
    }

static int
myfirst_write(struct cdev *dev, struct uio *uio, int ioflag)
{
    struct myfirst_softc *sc = dev->si_drv1;
    struct myfirst_fh *fh;
    char bounce[MYFIRST_BOUNCE];
    size_t want, put, room;
    int error;

    error = devfs_get_cdevpriv((void **)&fh);
    if (error != 0)
        return (error);
    if (sc == NULL || !sc->is_attached)
        return (ENXIO);

    while (uio->uio_resid > 0) {
        mtx_lock(&sc->mtx);
        room = cbuf_free(&sc->cb);
        mtx_unlock(&sc->mtx);
        if (room == 0)
            break;          /* full: short write */

        want = MIN((size_t)uio->uio_resid, sizeof(bounce));
        want = MIN(want, room);
        error = uiomove(bounce, want, uio);
        if (error != 0)
            return (error);

        mtx_lock(&sc->mtx);
        put = cbuf_write(&sc->cb, bounce, want);
        sc->bytes_written += put;
        fh->writes += put;
        mtx_unlock(&sc->mtx);

        /*
         * cbuf_write may store less than 'want' if another
         * writer slipped in between our snapshot of 'room'
         * and our cbuf_write call and consumed some of the
         * space we had sized ourselves against. With a single
         * writer that cannot happen and put == want always.
         * We still handle it defensively: a serious driver

```

```

        * would reserve space up front to avoid losing bytes,
        * and Chapter 11 will revisit this with proper
        * multi-writer synchronization.
        */
    if (put < want) {
        /*
         * The 'want - put' bytes we copied into 'bounce'
         * with uiomove have already left the caller's
         * uio and cannot be pushed back. Record the
         * loss by breaking out of the loop; the kernel
         * will report the bytes actually stored via
         * uio_resid. This path is only reachable under
         * concurrent writers, which the design here
         * does not yet handle.
         */
        break;
    }
    return (0);
}

```

A few things changed from Stage 3 of Chapter 9.

The first change is the **loop**. Both handlers now loop until `uio_resid` reaches zero or until the buffer cannot satisfy the next iteration. Each iteration moves at most `sizeof(bounce)` bytes, which is the size of the stack bounce. For a small request, the loop runs once. For a large request, it runs as many times as needed. This is what makes partial I/O work cleanly: the handlers naturally produce a short read or write when the buffer reaches a boundary.

The second change is that **all buffer access is bracketed by `mtx_lock/mtx_unlock`**. The `cbuf` data structure is unaware of locking; the driver provides it. We hold the lock around every `cbuf_*` call and around every update of the byte counters. We do *not* hold the lock across `uiomove(9)`. Holding a mutex across `uiomove` is a real bug in FreeBSD: `uiomove` may sleep on a page fault, and sleeping with a mutex held is a sleep-with-mutex panic. The Chapter 9 walkthrough discussed this; we are now operationalising the rule by separating the `cbuf` access (under lock) from the `uiomove` (without lock).

The third change is that the **read handler returns 0** when the buffer is empty, after possibly having transferred some bytes. The old Stage 3 behaviour was identical at this layer. What changes is that the *next* section makes it possible for the read to block instead, and the section after that adds an `EAGAIN` path for non-blocking callers. The structure here is the foundation for both extensions.

The fourth change is that the **write handler honours partial writes**. When `cbuf_free(&sc->cb)` returns zero, the loop exits and the handler returns 0 with `uio_resid` reflecting the bytes that were not consumed. The user-space `write(2)` call will see a short write count, which is the conventional UNIX way to say “I accepted this many of your bytes; please call me again with the rest later.” Section 4 talks at length about why this matters and how to write user code that handles it.

10.4.4 Updating attach and detach

The lifecycle changes are small but real:

```
static int
myfirst_attach(device_t dev)
{
    struct myfirst_softc *sc;
    struct make_dev_args args;
    int error;

    sc = device_get_softc(dev);
    sc->dev = dev;
    sc->unit = device_get_unit(dev);

    mtx_init(&sc->mtx, device_get_nameunit(dev), "myfirst", MTX_DEF);

    sc->attach_ticks = ticks;
    sc->is_attached = 1;
    sc->active_fhs = 0;
    sc->open_count = 0;
    sc->bytes_read = 0;
    sc->bytes_written = 0;

    error = cbuf_init(&sc->cb, MYFIRST_BUFSIZE);
    if (error != 0)
        goto fail_mtx;

    make_dev_args_init(&args);
    args.mda_devsw = &myfirst_cdevsw;
    args.mda_uid = UID_ROOT;
    args.mda_gid = GID_OPERATOR;
    args.mda_mode = 0660;
    args.mda_si_drv1 = sc;

    error = make_dev_s(&args, &sc->cdev, "myfirst/%d", sc->unit);
    if (error != 0)
        goto fail_cb;

    sc->cdev_alias = make_dev_alias(sc->cdev, "myfirst");
    if (sc->cdev_alias == NULL)
        device_printf(dev, "failed to create /dev/myfirst alias\n");

    sysctl_ctx_init(&sc->sysctl_ctx);
    sc->sysctl_tree = SYSCTL_ADD_NODE(&sc->sysctl_ctx,
```

```

        SYSCTL_CHILDREN(device_get_sysctl_tree(dev)),
        OID_AUTO, "stats", CTLFLAG_RD | CTLFLAG_MPSAFE, 0,
        "Driver statistics");

SYSCTL_ADD_U64(&sc->sysctl_ctx, SYSCTL_CHILDREN(sc->sysctl_tree),
        OID_AUTO, "attach_ticks", CTLFLAG_RD,
        &sc->attach_ticks, 0, "Tick count when driver attached");
SYSCTL_ADD_U64(&sc->sysctl_ctx, SYSCTL_CHILDREN(sc->sysctl_tree),
        OID_AUTO, "open_count", CTLFLAG_RD,
        &sc->open_count, 0, "Lifetime number of opens");
SYSCTL_ADD_INT(&sc->sysctl_ctx, SYSCTL_CHILDREN(sc->sysctl_tree),
        OID_AUTO, "active_fhs", CTLFLAG_RD,
        &sc->active_fhs, 0, "Currently open descriptors");
SYSCTL_ADD_U64(&sc->sysctl_ctx, SYSCTL_CHILDREN(sc->sysctl_tree),
        OID_AUTO, "bytes_read", CTLFLAG_RD,
        &sc->bytes_read, 0, "Total bytes drained from the FIFO");
SYSCTL_ADD_U64(&sc->sysctl_ctx, SYSCTL_CHILDREN(sc->sysctl_tree),
        OID_AUTO, "bytes_written", CTLFLAG_RD,
        &sc->bytes_written, 0, "Total bytes appended to the FIFO");
SYSCTL_ADD_PROC(&sc->sysctl_ctx, SYSCTL_CHILDREN(sc->sysctl_tree),
        OID_AUTO, "cb_used",
        CTLTYPE_UINT | CTLFLAG_RD | CTLFLAG_MPSAFE,
        sc, 0, myfirst_sysctl_cb_used, "IU",
        "Live bytes currently held in the circular buffer");
SYSCTL_ADD_PROC(&sc->sysctl_ctx, SYSCTL_CHILDREN(sc->sysctl_tree),
        OID_AUTO, "cb_free",
        CTLTYPE_UINT | CTLFLAG_RD | CTLFLAG_MPSAFE,
        sc, 0, myfirst_sysctl_cb_free, "IU",
        "Free bytes available in the circular buffer");
SYSCTL_ADD_UINT(&sc->sysctl_ctx, SYSCTL_CHILDREN(sc->sysctl_tree),
        OID_AUTO, "cb_size", CTLFLAG_RD,
        (unsigned int *)&sc->cb.cb_size, 0,
        "Capacity of the circular buffer");

device_printf(dev,
        "Attached; node /dev/%s (alias /dev/myfirst), cbuf=%zu bytes\n",
        devtoname(sc->cdev), cbuf_size(&sc->cb));
return (0);

fail_cb:
    cbuf_destroy(&sc->cb);
fail_mtx:
    mtx_destroy(&sc->mtx);
    sc->is_attached = 0;

```

```
        return (error);
    }

static int
myfirst_detach(device_t dev)
{
    struct myfirst_softc *sc;

    sc = device_get_softc(dev);

    mtx_lock(&sc->mtx);
    if (sc->active_fhs > 0) {
        mtx_unlock(&sc->mtx);
        device_printf(dev,
            "Cannot detach: %d open descriptor(s)\n",
            sc->active_fhs);
        return (EBUSY);
    }
    mtx_unlock(&sc->mtx);

    if (sc->cdev_alias != NULL) {
        destroy_dev(sc->cdev_alias);
        sc->cdev_alias = NULL;
    }
    if (sc->cdev != NULL) {
        destroy_dev(sc->cdev);
        sc->cdev = NULL;
    }
    sysctl_ctx_free(&sc->sysctl_ctx);
    cbuf_destroy(&sc->cb);
    mtx_destroy(&sc->mtx);
    sc->is_attached = 0;
    return (0);
}
```

The two new sysctl handlers are short:

```
static int
myfirst_sysctl_cb_used(SYSCTL_HANDLER_ARGS)
{
    struct myfirst_softc *sc = arg1;
    unsigned int val;

    mtx_lock(&sc->mtx);
```

```

        val = (unsigned int)cbuf_used(&sc->cb);
        mtx_unlock(&sc->mtx);
        return (sysctl_handle_int(oidp, &val, 0, req));
    }

static int
myfirst_sysctl_cb_free(SYSCTL_HANDLER_ARGS)
{
    struct myfirst_softc *sc = arg1;
    unsigned int val;

    mtx_lock(&sc->mtx);
    val = (unsigned int)cbuf_free(&sc->cb);
    mtx_unlock(&sc->mtx);
    return (sysctl_handle_int(oidp, &val, 0, req));
}

```

The handlers exist because we want a *consistent* snapshot of the buffer state when the user reads `sysctl dev.myfirst.0.stats.cb_used`. Reading the field directly (the way Stage 3 did with `bufused`) is racy: a concurrent write could be modifying it as `sysctl(8)` reads it, producing a torn value. The handler holds the mutex around the read, so the value the user sees is at least *self-consistent* (it represents the buffer state at one moment in time, not a half-update). The buffer can change immediately after the handler releases the lock, of course; that is fine, because by the time `sysctl(8)` formats and prints the number, the buffer has often changed anyway. What we are preventing is a read of a partially modified field, not a stale read.

10.4.5 Logging Buffer State for Debugging

When the driver misbehaves, the first question is almost always “what was the buffer doing?” Adding a small amount of `device_printf` traffic to the I/O handlers, behind a `sysctl`-controlled debug flag, makes this question easy to answer. Here is the pattern:

```

static int myfirst_debug = 0;
SYSCTL_INT(_dev_myfirst, OID_AUTO, debug, CTLFLAG_RW,
    &myfirst_debug, 0, "Verbose I/O tracing for the myfirst driver");

#define MYFIRST_DBG(sc, fmt, ...) do { \
    if (myfirst_debug) \
        device_printf((sc)->dev, fmt, ##__VA_ARGS__); \
} while (0)

```

Then in the I/O handlers, call `MYFIRST_DBG(sc, "read got=%zu used=%zu free=%zu\n", got, cbuf_used(&sc->cb), cbuf_free(&sc->cb))`; after a successful `cbuf_read`. With `myfirst_debug` set to 0, the macro reduces to a no-op and the production path is untouched. With `sysctl dev.myfirst.debug=1`,

every transfer prints a one-line trace into `dmesg`, which is invaluable when the driver is doing something you do not understand.

Be polite about how much trace you emit. A single log line per transfer is fine. A log line per byte transferred would melt `dmesg`'s ring buffer in seconds and would change the timing of the driver enough to hide some bugs. The pattern above logs once per `cbuf_read` or `cbuf_write` call, which is once per loop iteration, which is once per chunk of up to 256 bytes. That is roughly the right granularity.

Finally, remember to set `myfirst_debug = 0` before you load the driver in production. The line is there as a development aid, not as a permanent feature.

10.4.6 Why the `cbuf` Has Its Own Memory Tag

When you run `vmstat -m` on a FreeBSD system, you see a long list of memory tags and how much memory each one currently holds. Tags are an essential observability tool: if memory is leaking somewhere in the kernel, the tag whose count keeps growing tells you where to look. We gave `cbuf` its own tag (`M_CBUF`) so that its allocations are visible separately from the rest of the driver's allocations.

To see the effect, load the Stage 2 driver and run:

```
$ vmstat -m | head -1
      Type InUse MemUse Requests  Size(s)
$ vmstat -m | grep -E '(\s+Type|cbuf|myfirst)'
      Type InUse MemUse Requests  Size(s)
      cbuf      1      4K          1  4096
```

The four kilobytes correspond to the single 4 KB allocation `cbuf_init` made for `sc->cb.cb_data`. Unload the driver and the count drops back to zero. If at any point the count ever rises *without* a corresponding driver attach, you have a leak in `cbuf_init` or `cbuf_destroy`. This is the kind of regression that would otherwise be invisible until the system ran out of memory hours later.

10.4.7 A Quick Trace of an Aligned and a Wrapped Transfer

To make the wrap-around behaviour real, let us trace two writes through the Stage 2 driver. Assume the buffer is empty at the start, capacity is 4096, and a user calls `write(fd, buf, 100)` followed later by `write(fd, buf2, 100)`.

The first write goes through `myfirst_write`:

1. `uio_resid = 100, cbuf_free = 4096, room = 4096`.
2. Loop iteration 1: `want = MIN(100, 256, 4096) = 100`. `uiomove` copies 100 bytes from user space into bounce. `cbuf_write(&sc->cb, bounce, 100)` returns 100, advances `cb_used` to 100, leaves `cb_head = 0`. The implicit tail is now 100.
3. `uio_resid = 0`. The loop exits. The handler returns 0. The user sees a write count of 100.

The buffer state is: `cb_data[0..99]` holds the data, `cb_head = 0, cb_used = 100, cb_size = 4096`.

Now the second write arrives. Before it does, suppose a reader has consumed 80 bytes, leaving `cb_head = 80, cb_used = 20`. The implicit tail is at position 100. `myfirst_write` runs:

1. `uio_resid = 100, cbuf_free = 4076, room = 4076.`
2. Loop iteration 1: `want = MIN(100, 256, 4076) = 100.` `uiomove` copies 100 bytes from user space into bounce. `cbuf_write(&sc->cb, bounce, 100)` advances the implicit tail from 100 to 200, sets `cb_used = 120`, returns 100.
3. `uio_resid = 0.` The handler returns 0. The user sees a write count of 100.

Both transfers were “aligned” in the sense that neither crossed the end of the underlying buffer. Now imagine a much later state where `cb_head = 4000` and `cb_used = 80`. The live bytes occupy positions 4000..4079, and the implicit tail is 4080. Capacity is 4096. The free space is 4016 bytes, but it splits across the wrap: 16 bytes contiguous after position 4080, then 4000 contiguous from position 0.

A user calls `write(fd, buf, 64)`:

1. `uio_resid = 64, cbuf_free = 4016, room = 4016.`
2. Loop iteration 1: `want = MIN(64, 256, 4016) = 64.` `uiomove` copies 64 bytes into bounce. `cbuf_write(&sc->cb, bounce, 64)` runs:
 - `tail = (4000 + 80) % 4096 = 4080.`
 - `first = MIN(64, 4096 - 4080) = 16.` Copies 16 bytes from bounce + 0 to `cb_data + 4080`.
 - `second = 64 - 16 = 48.` Copies 48 bytes from bounce + 16 to `cb_data + 0`.
 - `cb_used += 64`, becoming 144.
3. `uio_resid = 0.` Handler returns 0.

The wrap was handled inside `cbuf_write` and was invisible to the driver. That is the whole point of putting the abstraction in its own file. The `myfirst.c` source has no wrap-around arithmetic; the wrap-around lives in `cbuf.c` where it can be tested in isolation.

10.4.8 What the User Sees

After the splice, a user-space program cannot tell what shape the buffer is. `cat /dev/myfirst` still prints whatever has been written, in order. `echo hello > /dev/myfirst` still stores `hello` for later reading. The byte counters in `sysctl dev.myfirst.0.stats` still tick up by one for each byte. The new `cb_used` and `cb_free` `sysctls` expose the buffer’s state, but the data path is byte-for-byte identical to Stage 3 of Chapter 9.

What differs is what happens *under load*. With the linear FIFO, a sustained writer eventually saw `ENOSPC` even when a reader was actively consuming bytes, because `bufhead` only collapsed back to zero when `bufused` reached zero. With the circular buffer, the writer keeps going indefinitely as long as the reader keeps up, because the free space and live bytes can occupy any combination of positions inside the underlying memory. The buffer’s full capacity is now actually usable.

You will be able to see this difference clearly in Section 6 when we run `dd` against the new driver and compare the throughput numbers to Stage 3 of Chapter 9. For now, take it on faith and finish the splice.

10.4.9 Handling Detach with Live Data

There is one detach-time subtlety worth addressing. With the circular buffer integrated, the buffer may hold data when the user runs `kldunload myfirst`. The Chapter 9 detach refused to unload while any descriptor was

open; that check still applies. It does not, however, refuse to unload if the buffer is non-empty but no descriptor is open. Should it?

The conventional answer is no. A buffer is a transient resource. If no one is currently reading the device, the bytes in the buffer are not going to be read; the user has implicitly accepted their loss by closing all descriptors. The detach path simply frees the buffer along with everything else. If you wanted to preserve the bytes across an unload (to a file, for instance), that would be a feature, not a bug fix, and it would belong in user space, not in the driver.

We therefore make no change to the detach lifecycle. `cbuf_destroy` is called unconditionally; the bytes are released along with the backing memory.

10.4.10 Wrapping Up Section 3

The driver now uses a real circular buffer. The wrap-around logic lives in a small abstraction that has its own header, its own source file, its own memory tag, and its own userland test program. The I/O handlers in `myfirst.c` are simpler than they were in Stage 3 of Chapter 9, and the tricky arithmetic is no longer scattered across them.

What you have at this point still does not handle partial reads and writes elegantly. If a user calls `read(fd, buf, 4096)` and the buffer holds 100 bytes, the loop will execute exactly once, transfer 100 bytes, and return zero with `uio_resid` reflecting the unconsumed portion. That is the correct behaviour, but the *prose* around what the user should expect, what `read(2)` returns, and how a well-written caller loops is what Section 4 is about. We will also resolve the question of what `d_read` should do when the buffer is empty and the caller is willing to wait, which is the door into non-blocking I/O in Section 5.

10.5 Section 4: Improving the Driver's Behaviour with Partial Reads and Writes

The Stage 2 driver from the previous section already implements partial reads and writes correctly, almost by accident. The loops in `myfirst_read` and `myfirst_write` exit when the circular buffer can no longer satisfy the next iteration, leaving `uio->uio_resid` at whatever portion of the request remains unconsumed. The kernel computes the user-visible byte count as the original request size minus that residual. Both `read(2)` and `write(2)` then return that number to user space.

What we have not done is *think clearly* about what those partial transfers mean from both sides of the trust boundary. This section does that thinking. By the end of it, you will know which user-space programs handle short reads and writes correctly, which ones do not, what your driver should report when nothing at all is available, and what the rare zero-byte transfer means.

10.5.1 What “Partial” Means in UNIX

A `read(2)` returns one of three things:

- A *positive integer* less than or equal to the requested count: that many bytes were placed in the caller's buffer.

- *Zero*: end of file. No more bytes will ever be produced on this descriptor; the caller should close.
- *-1*: an error occurred; the caller examines `errno` to decide what to do.

The first case is where partial transfers live. A “full” read returns exactly the requested count. A “partial” read returns fewer bytes. UNIX has always allowed partial reads, and any program that calls `read(2)` and assumes it got the full requested count is wrong. Robust programs always look at the return value and either loop until they have what they need, or accept the partial result and move on.

`write(2)` follows the same shape:

- A *positive integer* less than or equal to the requested count: that many bytes were accepted by the kernel.
- Sometimes *zero* (rarely seen in practice; usually treated as a short write of zero bytes).
- *-1*: an error occurred.

A short write means “I took this many of your bytes; please call me again with the remaining tail.” Robust producers always loop until they have offered the entire payload.

10.5.2 Why Drivers Should Embrace Partial Transfers

It would be tempting to make a driver always satisfy the entire request, even if it has to internally loop or wait. Some drivers do this in special cases (consider the `null` driver’s read, which loops internally to deliver `ZERO_REGION_SIZE`-byte chunks until the caller’s request is exhausted). For most drivers, however, embracing partial transfers is the right design choice for several reasons.

The first reason is **responsiveness**. A reader that asks for 4096 bytes and gets back 100 bytes has 100 bytes of work it can begin doing right now, instead of waiting for another 3996 bytes that may never arrive. The kernel does not have to guess how long the caller is willing to wait.

The second reason is **fairness**. If `myfirst_read` loops internally until it satisfies the entire request, a single greedy reader can hold the buffer’s mutex for an indefinite time, starving every other thread that wants to access the driver. A handler that returns as soon as it cannot make progress lets the kernel scheduler preserve fairness across competing threads.

The third reason is **correctness in the face of signals**. A reader that has been waiting may receive a signal (e.g. `SIGINT` from the user pressing Ctrl-C). The kernel needs the chance to deliver that signal, which usually means returning from the current syscall. A handler that loops indefinitely never gives the kernel that chance, and the user’s `kill -INT` is delayed or lost.

The fourth reason is **composition with `select(2)` / `poll(2)`**. Programs that use these readiness primitives are explicitly assuming partial-transfer semantics. They expect to be told “data is ready” and then to loop on `read(2)` until the descriptor returns zero or `EAGAIN`. A driver that always returns the full requested count breaks the polling model.

For all of these reasons, the `myfirst` driver’s loops in Section 3 are designed to make a single pass through the buffer’s available data, transfer what they can, and return. The next time the caller wants more, it calls `read(2)` again. This is the conventional UNIX shape.

10.5.3 Reporting Accurate Byte Counts

The mechanism by which the driver reports a partial transfer is `uio->uio_resid`. The kernel sets it to the requested count before calling `d_read` or `d_write`. The handler is responsible for decrementing it as it transfers bytes. `uiomove(9)` decrements it automatically. When the handler returns, the kernel calculates the byte count as `original_resid - uio->uio_resid` and returns that to user space.

This means the handler must do exactly two things consistently:

1. Use `uiomove(9)` (or one of its companions, `uiomove_frombuf(9)`, `uiomove_nofault(9)`) to do every byte movement that crosses the trust boundary. This is what keeps `uio_resid` honest.
2. Return zero when it has done as much as it can, regardless of whether `uio_resid` is now zero or some positive number.

A handler that returns a *positive* byte count is wrong. The kernel ignores positive returns; the byte count is computed from `uio_resid`. Returning a positive integer would be silently wasteful. A handler that returns a *negative* number, or any value that is not in `errno.h`, is undefined behaviour.

A common and dangerous variant of this mistake is returning `EAGAIN` when the buffer is empty *and* having transferred some bytes earlier in the same call. The user-space `read(2)` would see `-1/EAGAIN`, and the bytes that were in the user buffer would silently be considered uninvolved. The right pattern is: if the handler has transferred any bytes at all, it returns 0 and lets the partial count speak for itself; only if it has transferred *zero* bytes can it return `EAGAIN`. Section 5 will codify this rule when we add non-blocking support.

10.5.4 End-of-Data: When Should `d_read` Return Zero?

UNIX's "zero means EOF" rule has an interesting consequence for pseudo-devices. A regular file has a definite end: when `read(2)` reaches it, the kernel returns zero. A character device usually does not have a definite end. A serial line, a keyboard, a network device, a tape rewinding past the end of the medium: each of these *might* return zero in special cases, but in normal operation, "no data available right now" is not the same as "no data ever again."

Yet a naive `myfirst_read` that returns zero whenever the buffer is empty looks indistinguishable, from the caller's point of view, from a regular file at end-of-file. A `cat /dev/myfirst` will see zero bytes, treat it as EOF, and exit. That is not what we want. We want the reader to wait until more bytes arrive, or to be told "there are no bytes right now, but try again later" depending on the file descriptor's mode.

Two strategies are common.

The first strategy is **block by default**. `myfirst_read` waits on a sleep queue when the buffer is empty, and the writer wakes the queue when it adds bytes. Reading returns zero only if some condition signals true end-of-file (the device has been removed, the writer has explicitly closed). This is what most pseudo-devices and most TTY-style devices do. It matches `cat`'s expectation that a terminal will deliver lines as the user types them.

The second strategy is **immediate return with `EAGAIN` for non-blocking callers**. If the descriptor was opened with `O_NONBLOCK` (or the user later set the flag with `fcntl(2)`), `myfirst_read` returns `-1/EAGAIN` instead of blocking. This lets event-loop programs use `select(2)`, `poll(2)`, or `kqueue(2)` to multiplex many descriptors without committing to wait on any single one.

Section 5 will implement both strategies. The blocking path is the default; the non-blocking path activates when `IO_NDELAY` is set in `ioflag`. For now, in Stage 2, the driver still returns zero on empty, the same as Chapter 9 did. That is a temporary state; nothing in user space stays stable when the data path can vanish at any moment.

10.5.5 Backpressure on the Write Side

The mirror of “no data right now” is “no room right now.” When the buffer is full and a writer asks to add more bytes, the driver has to choose what to say.

The Stage 3 driver of Chapter 9 returned `ENOSPC`, which is the conventional signal for “the device has run out of space, permanently.” That was a defensible choice in Chapter 9 because the linear FIFO genuinely could not accept more data until the buffer fully drained. With the circular buffer, however, “full” is a transient state: the writer just needs to wait until a reader has consumed something. The right return is therefore *not* `ENOSPC` in the steady state; it is either a blocking sleep until space appears, or `EAGAIN` for non-blocking callers.

The Stage 2 implementation already handles the partial-write case correctly: when the buffer fills mid-transfer, the loop exits and the user sees a write count smaller than the request. What it does *not* do yet is the right thing when the buffer is full *at the start* of the call: it returns 0 with no bytes transferred, which the kernel turns into a `write(2)` return of zero. A return of zero from `write(2)` is technically legal but is an odd thing to see, and most user programs will treat it as an error or loop forever waiting for it to become non-zero.

The conventional fix, again, is mode-dependent. A blocking writer should sleep until space is available; a non-blocking writer should receive `EAGAIN`. We will implement both in Section 5. The structure of the Stage 2 loop is already correct for both cases; what is missing is the choice about what to do when *no* progress was made on the first iteration.

10.5.6 Zero-Length Reads and Writes

A zero-length read or write is a perfectly legal call. `read(fd, buf, 0)` and `write(fd, buf, 0)` are valid syscalls; they exist explicitly so that programs can validate a file descriptor without committing to a transfer. The kernel passes them down to the driver with `uio->uio_resid == 0`.

Your handler must not panic, error, or loop in this case. The Stage 2 driver naturally does the right thing: the `while (uio->uio_resid > 0)` loop never executes, and the handler returns 0 with `uio_resid` still 0. The user sees `read(2)` or `write(2)` return zero. Programs that call zero-length I/O for descriptor validation get the result they expect.

Be cautious about adding “is the request empty?” early returns at the start of your handler. They look like a small optimization but they introduce branching that is easy to get wrong. The Chapter 9 cheat sheet’s rule applies: `if (uio->uio_resid == 0) return (EINVAL);` is a bug.

10.5.7 A User-Space Walk Through the Loop

Watching what a user program does with a partial transfer is the best way to internalise the contract. Here is a small reader written in idiomatic UNIX style:

```
static int
read_all(int fd, void *buf, size_t want)
{
    char *p = buf;
    size_t left = want;
    ssize_t n;

    while (left > 0) {
        n = read(fd, p, left);
        if (n < 0) {
            if (errno == EINTR)
                continue;
            return (-1);
        }
        if (n == 0)
            break;          /* EOF */
        p += n;
        left -= n;
    }
    return (int)(want - left);
}
```

read_all keeps calling read(2) until either it has all want bytes, or it sees end-of-file, or it sees a real error. Short reads are absorbed transparently. A EINTR from a signal causes a retry. The function returns the actual number of bytes obtained.

A correctly written write_all is the mirror image:

```
static int
write_all(int fd, const void *buf, size_t have)
{
    const char *p = buf;
    size_t left = have;
    ssize_t n;

    while (left > 0) {
        n = write(fd, p, left);
        if (n < 0) {
            if (errno == EINTR)
                continue;
            return (-1);
        }
        if (n == 0)
            break;          /* unexpected; treat as error */
        p += n;
    }
}
```

```
        left -= n;
    }
    return (int)(have - left);
}
```

`write_all` calls `write(2)` repeatedly until the whole payload has been accepted by the kernel. Short writes are absorbed transparently. A `EINTR` causes a retry. The function returns the number of bytes accepted.

Both helpers belong in the same file (or a shared utility header) because they are nearly always used together. They are short, robust, and they make user-space code that talks to your driver behave correctly even when the driver is doing partial transfers. We will use both in the test programs you build in Section 6.

10.5.8 What `cat`, `dd`, and Friends Actually Do

The base-system tools you have been using to test the driver each handle short reads and writes differently. It is worth knowing what each does, so that you can interpret what you see.

`cat(1)` reads with a buffer of `MAXBSIZE` (16 KB on FreeBSD 14.3) and writes whatever it gets in a loop. Short reads from the source descriptor are absorbed; `cat` simply makes another `read(2)` call. Short writes to the destination descriptor are also absorbed; `cat` writes the unconsumed tail in a follow-up call. As far as `cat` is concerned, the size of the transfers does not matter; it just keeps moving bytes until it sees end-of-file on the source.

`dd(1)` is more rigid. It reads in blocks of `bs=` bytes (default 512) and writes whatever it received in the same block size. Crucially, `dd` *does not* loop on a short read by default. If `read(2)` returns 100 bytes when `bs=4096`, `dd` writes a 100-byte block and increments its short-read counter. The output you see at the end (`X+Y records in / X+Y records out`) is split between full records (`X`) and short records (`Y`). The total byte count is what matters; the split tells you whether the source was producing short reads.

There is a `dd` flag, `iflag=fullblock`, that makes it loop on the source the way `cat` does. Use it when you want to test throughput without short-read noise: `dd if=/dev/myfirst of=/dev/null bs=4k iflag=fullblock`. Without that flag, you will see records split by every short read.

`hexdump(1)` reads one byte at a time by default but can be told to read larger blocks. It does not care about short reads from the source.

`truss(1)` traces every syscall, including the byte counts returned by each. Running a producer or consumer under `truss` is the most direct way to see what byte counts your driver is returning. If you run `truss -f -t read,write cat /dev/myfirst`, the output will tell you exactly how many bytes each `read(2)` returned, and you can correlate that with `cb_used` in `sysctl`.

10.5.9 Common Mistakes in Partial-Transfer Code

The following mistakes are the ones that show up most often in beginner driver code. Each one has the same shape: the handler does something that looks reasonable in a single test case and silently misbehaves under load.

Mistake 1: returning the byte count from `d_read` or `d_write`. A handler that does return `((int)nbytes);` instead of `return (0);` is wrong. The kernel ignores the positive value (because positive returns are not valid `errno` values) and computes the byte count from `uio_resid`. The handler that returns `nbytes` and *also* does the right thing with `uiomove` accidentally works; the handler that returns `nbytes` and skips the `uiomove` step silently corrupts data. Do not invent your own return convention.

Mistake 2: returning `EAGAIN` after a partial transfer. A handler that has already consumed some bytes from `uio` and then returns `EAGAIN` because no more are available silently throws away the bytes the user already got. The correct rule is: if you transferred any bytes, return 0; only if you transferred zero bytes can you return an `errno` like `EAGAIN`.

Mistake 3: refusing zero-length transfers. As noted above, `read(fd, buf, 0)` and `write(fd, buf, 0)` are legal. A handler that returns `EINVAL` on zero `uio_resid` breaks programs that use zero-length I/O for descriptor validation.

Mistake 4: looping inside the handler when the buffer is empty. A handler that spins inside the kernel waiting for data to appear blocks the calling thread *and* every thread that wants to acquire the same lock. The right mechanism for waiting is `mtx_sleep(9)` or `cv_wait(9)`, not a busy loop. Section 5 covers this.

Mistake 5: holding the buffer mutex across `uiomove`. This is the single most common bug in beginner driver code. `uiomove` may sleep on a page fault. Sleeping while holding a non-sleepable mutex is a `KASSERT` panic on `INVAARIANTS`-enabled kernels and a `WITNESS` warning on kernels with `WITNESS` enabled; on a production kernel built without either, the same pattern can still deadlock the machine or silently corrupt state when the page fault tries to page in a user page. Either way, the behaviour is wrong, and the testing kernel should catch it before production ever does. The Stage 2 handlers carefully release the mutex before calling `uiomove`. Repeat the pattern in every new handler you write.

Mistake 6: not honouring the user's signal. A blocking handler that does not pass `PCATCH` to `mtx_sleep(9)` or `tsleep(9)` cannot be interrupted by a signal. The user's `Ctrl-C` is silently ignored, and only `kill -9` will free the thread. Always allow signals to interrupt a wait, and always handle the resulting `EINTR` cleanly.

Mistake 7: trusting `uio->uio_resid` after a failure. When `uiomove` returns a non-zero error (for instance, `EFAULT` because the user-space buffer is invalid), `uio_resid` may be partially decremented or fully decremented depending on where in the transfer the fault occurred. The convention is: propagate the error, do not retry, and accept that the byte count seen by the user may include some bytes that arrived before the fault. This is rare in practice, and the user gets `EFAULT` plus a byte count that lets them recover.

10.5.10 A Concrete Example: Watching Partial Reads

To make this real, load the Stage 2 driver, write a few hundred bytes into it, and watch a small reader collect them in chunks. With the driver loaded:

```
$ printf 'aaaaaaaaaaaaaaaaaaaaa' > /dev/myfirst          # 20 bytes
$ printf 'bbbbbbbbbbbbbbbbbbbbbb' > /dev/myfirst        # 20 more
$ sysctl dev.myfirst.0.stats.cb_used
dev.myfirst.0.stats.cb_used: 40
```

The buffer holds 40 bytes. Now run a small reader, traced by `truss`:

```

/* shortreader.c */
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>

int
main(void)
{
    int fd = open("/dev/myfirst", O_RDONLY);
    char buf[1024];
    ssize_t n;

    n = read(fd, buf, sizeof(buf));
    printf("read 1: %zd\n", n);
    n = read(fd, buf, sizeof(buf));
    printf("read 2: %zd\n", n);
    close(fd);
    return (0);
}

```

```

$ cc -o shortreader shortreader.c
$ truss -t read,write ./shortreader
... read(3, ...) = 40 (0x28)
read 1: 40
... read(3, ...) = 0 (0x0)
read 2: 0

```

The first `read(2)` returned 40 even though the user asked for 1024. That is a partial read, and it is correct. The second `read(2)` returned 0 because the buffer was empty. With Stage 2 the zero is a stand-in for “no data right now”; with Stage 3 (after we add blocking) the second read will sleep until more data arrives.

Now do the same with a tighter buffer to see partial reads on a larger transfer:

```

$ dd if=/dev/zero bs=1m count=8 | dd of=/dev/myfirst bs=4096 2>/tmp/dd-w &
$ dd if=/dev/myfirst of=/dev/null bs=512 2>/tmp/dd-r

```

When the writer is producing 4096-byte blocks faster than the reader is consuming 512-byte blocks, the buffer fills up. The writer’s `write(2)` calls start returning short counts, and `dd` records each short call as a partial record. The reader keeps reading 512 at a time. When you stop both processes, look at the `records out` line in `/tmp/dd-w` and the `records in` line in `/tmp/dd-r`; the second number on each line is the count of short records.

This is healthy behaviour. The driver is doing exactly what a UNIX device should do: let each side proceed at its own pace, report partial transfers honestly, and never block when there is nothing to wait for. Without partial-transfer semantics, the writer would have hit `ENOSPC` (the Chapter 9 behaviour) and `dd` would have stopped.

10.5.11 Wrapping Up Section 4

The driver's read and write handlers are now correctly partial-transfer-aware. We have not changed the code from Section 3; we have only made the behaviour explicit and built up the vocabulary needed to talk about it. You know what `read(2)` and `write(2)` return when only some of the bytes are available, you know how to write user-space loops that handle those returns, and you know which base-system tools handle partial transfers gracefully and which need a flag.

What is still missing is the right behaviour when the buffer is *completely* empty (for reads) or *completely* full (for writes). The Stage 2 driver still returns zero or stops with no progress; that is a stand-in for the more correct behaviour we are about to add. Section 5 introduces non-blocking I/O, the blocking sleep path, and `EAGAIN`. After that, the driver will behave correctly under all combinations of fill state and caller mode.

10.6 Section 5: Implementing Non-Blocking I/O

Up to this point in the chapter, the driver has been doing one of two things when a caller asks for a transfer that cannot be satisfied right now. It returns zero (on read, mimicking end-of-file) or it stops mid-loop with no bytes transferred (on write, telling the user “zero bytes accepted”). Neither behaviour is what a real character device should do. This section replaces both with the two correct behaviours: a blocking wait for the default case, and a clean `EAGAIN` for callers that opened the descriptor non-blocking.

Before we touch the driver, let us make sure we understand what “non-blocking” means from each side of the trust boundary. That vocabulary is what ties the implementation together.

10.6.1 What Non-Blocking I/O Is

A **blocking** descriptor is one for which `read(2)` and `write(2)` are allowed to sleep. If the driver has no data available, `read(2)` waits; if the driver has no room available, `write(2)` waits. The calling thread is suspended, possibly for a long time, until progress can be made. This is the default behaviour of every file descriptor in UNIX.

A **non-blocking** descriptor is one for which `read(2)` and `write(2)` must *never* sleep. If the driver has no data right now, `read(2)` returns `-1` with `errno = EAGAIN`. If the driver has no room right now, `write(2)` returns `-1` with `errno = EAGAIN`. The caller is expected to do something else (typically call `select(2)`, `poll(2)`, or `kqueue(2)` to find out when the descriptor becomes ready) and then try again.

The per-descriptor flag that turns non-blocking mode on or off is `O_NONBLOCK`. A program sets it either at `open(2)` time (`open(path, O_RDONLY | O_NONBLOCK)`) or later with `fcntl(2)` (`fcntl(fd, F_SETFL, O_NONBLOCK)`). The flag lives in the descriptor's `f_flag` field, which is private to the file structure; the driver does not see the flag directly.

What the driver *does* see is the `ioflag` argument to `d_read` and `d_write`. The devfs layer translates the descriptor's flags into bits of the `ioflag` that the handler can check. Specifically:

- `IO_NDELAY` is set when the descriptor has `O_NONBLOCK`.
- `IO_DIRECT` is set when the descriptor has `O_DIRECT`.
- `IO_SYNC` is set on `d_write` when the descriptor has `O_FSYNC`.

The translation is even simpler than it looks. A `CTASSERT` in `/usr/src/sys/fs/devfs/devfs_vnops.c` declares that `O_NONBLOCK == IO_NDELAY`. The bit values are chosen so the two names are interchangeable, and you can write `(ioflag & IO_NDELAY)` or `(ioflag & O_NONBLOCK)` depending on which convention feels clearer. Both work. The FreeBSD source tree uses `IO_NDELAY` more often, so we will follow that.

10.6.2 When Non-Blocking Behaviour Is Useful

Non-blocking mode is the underlying mechanism that makes event-driven programs possible. Without it, a single thread that wants to read from several descriptors has to pick one, block on it, and ignore the others until it wakes up. With it, a single thread can test several descriptors for readiness, process whichever is ready, and loop back without ever committing to sleep on any single one.

Three common programs rely on this mode heavily. A classical event loop (`libevent`, `libev`, or the now-standard `kqueue`-based pattern in FreeBSD) does nothing but wait in `kevent(2)` for an event, dispatch it, and loop. A networking daemon (`nginx`, `haproxy`) uses the same shape to juggle thousands of connections per thread. A real-time application (audio processing, industrial control) needs bounded worst-case latency and cannot afford a long block.

A driver that wants to play nicely with these programs must implement non-blocking mode correctly. Returning the wrong `errno`, sleeping when `IO_NDELAY` is set, or forgetting to notify `poll(2)` when state changes each produce bugs that are hard to diagnose.

10.6.3 The `IO_NDELAY` Flag: How It Flows to the Driver

Trace the flow once so you know where the flag comes from. The user calls `read(fd, buf, n)` on a descriptor that has `O_NONBLOCK` set. Inside the kernel:

1. `sys_read` looks up the file descriptor and finds a `struct file` with `fp->f_flag` containing `O_NONBLOCK`.
2. `vn_read` or (for character devices) `devfs_read_f` assembles an `ioflag` by masking `fp->f_flag` for the bits that drivers care about. In particular, it computes `ioflag = fp->f_flag & (O_NONBLOCK | O_DIRECT);`.
3. The computed `ioflag` is passed to the driver's `d_read`.

From the driver's point of view, the translation is complete: `ioflag & IO_NDELAY` is true if and only if the caller wants non-blocking semantics. A missing bit means block-if-needed. An extra bit means non-block-and-return-EAGAIN-if-needed.

On the write side the same pattern applies. `devfs_write_f` computes `ioflag = fp->f_flag & (O_NONBLOCK | O_DIRECT | O_FSYNC);` and passes it in. The write handler's check is symmetric: `ioflag & IO_NDELAY` is "do not block."

10.6.4 The `EAGAIN` Convention

When the driver's handler decides that it cannot make progress and the caller is non-blocking, it returns `EAGAIN`. The kernel's generic layer passes this through as `-1 / errno = EAGAIN` at the user level. The user is expected to treat `EAGAIN` as "this descriptor is not ready; wait or try later," not as an error in the traditional sense.

Two details about EAGAIN are worth committing to memory.

First, EAGAIN and EWOULDBLOCK are the same value in FreeBSD. They are two names for a single errno. Some older man pages use EWOULDBLOCK for socket-related contexts and EAGAIN for file-related contexts; the compatibility is tight, and either name is acceptable in driver code. The FreeBSD source tree uses EAGAIN almost exclusively for drivers.

Second, EAGAIN must only be returned when the handler has transferred *zero* bytes. If the handler has already moved some bytes via uiomove and then wants to stop because no more can move right now, it must return 0 (not EAGAIN). The kernel will compute the partial byte count from uio_resid and deliver that to the user. A subsequent call from the user will then see EAGAIN because the buffer is still empty. The rule is: EAGAIN means “no progress at all on this call”; a partial transfer means “progress, but less than requested, and now you need to retry for the rest.”

This is exactly the rule Section 4 introduced. Here we operationalise it in code.

10.6.5 The Blocking Path: `mtx_sleep(9)` and `wakeup(9)`

The blocking path is the default behaviour for a descriptor without `O_NONBLOCK`. When the buffer is empty, the reader sleeps; when a writer adds bytes, it wakes the reader. FreeBSD provides this with a pair of primitives that compose with mutexes.

`mtx_sleep(void *chan, struct mtx *mtx, int priority, const char *wmesg, sbintime_t tmo)` puts the calling thread to sleep on the “channel” `chan` (an arbitrary address used as a key), atomically releasing `mtx`. When the thread wakes, it reacquires `mtx` before returning. The `priority` argument may include `PCATCH` to allow signal delivery to interrupt the sleep, and `wmesg` is a short human-readable name that shows up in `ps -AxH` and similar tools. The `tmo` argument specifies a maximum sleep time; zero means no timeout.

`wakeup(void *chan)` wakes *every* thread sleeping on `chan`. `wakeup_one(void *chan)` wakes only one. For a single-reader driver, `wakeup` is fine; for a multi-reader driver where we want to hand off one chunk of work to one reader, `wakeup_one` is often right. For `myfirst` we will use `wakeup` because we may have both a producer and a consumer waiting, and we want to make sure neither is starved.

The contract between the two is that the sleeper must hold the mutex, check the condition, and call `mtx_sleep` *without* releasing the mutex in between. `mtx_sleep` atomically drops the lock and sleeps; when it returns, the lock is reacquired and the sleeper must re-check the condition (spurious wake-ups are possible; a concurrent thread may have taken the byte we were waiting for). The pattern is the classic `while (condition) mtx_sleep(...)` loop.

A minimal blocking read in our driver looks like this:

```
mtx_lock(&sc->mtx);
while (cbuf_used(&sc->cb) == 0) {
    if (ioflag & IO_NDELAY) {
        mtx_unlock(&sc->mtx);
        return (EAGAIN);
    }
    error = mtx_sleep(&sc->cb, &sc->mtx, PCATCH,
        "myfrd", 0);
}
```

```

    if (error != 0) {
        mtx_unlock(&sc->mtx);
        return (error);
    }
    if (!sc->is_attached) {
        mtx_unlock(&sc->mtx);
        return (ENXIO);
    }
}
/* ... now proceed to read from the cbuf ... */

```

Four points deserve comment.

The first is the **condition in the while loop**. We check `cbuf_used(&sc->cb) == 0`. As long as that is true, we sleep. The while-check is essential: `mtx_sleep` can return for reasons other than “data appeared” (signals, timeouts, spurious wake-ups, or a different thread having consumed the data before we could). After every return from `mtx_sleep`, we must re-check.

The second is the **EAGAIN path**. If the caller is non-blocking and the buffer is empty, we release the lock and return `EAGAIN` without sleeping. The check has to happen *before* `mtx_sleep`, not after; otherwise we would sleep, wake up, then discover the caller was non-blocking all along.

The third is the **PCATCH**. With `PCATCH`, `mtx_sleep` can return `EINTR` or `ERESTART` if a signal is delivered. Propagating that return to the user is the whole purpose of `PCATCH`: we want the user’s Ctrl-C to actually interrupt the read. Without `PCATCH`, `SIGINT` is held until the sleep completes for some other reason, and the user gets a long, unexplained hang.

The fourth is the **detach check**. After `mtx_sleep` returns, it is possible that `myfirst_detach` has begun and that `sc->is_attached` is now zero. We check and return `ENXIO` if so. This prevents a read from proceeding against a partially torn-down driver. The detach code path has to call `wakeup(&sc->cb)` to release any sleepers before tearing the mutex down; we will add that call below.

10.6.6 The Writer Side

The write path is the mirror image:

```

mtx_lock(&sc->mtx);
while (cbuf_free(&sc->cb) == 0) {
    if (ioflag & IO_NDELAY) {
        mtx_unlock(&sc->mtx);
        return (EAGAIN);
    }
    error = mtx_sleep(&sc->cb, &sc->mtx, PCATCH,
        "myfwr", 0);
    if (error != 0) {
        mtx_unlock(&sc->mtx);
    }
}

```

```

        return (error);
    }
    if (!sc->is_attached) {
        mtx_unlock(&sc->mtx);
        return (ENXIO);
    }
}
/* ... now proceed to write into the cbuf ... */

```

The same four points apply: check the condition in a while loop, handle `IO_NDELAY` before sleeping, pass `PCATCH`, re-check `is_attached` after the sleep. Notice that both sleepers use the same “channel” (`&sc->cb`). That is deliberate. When a reader transfers bytes out of the buffer, it calls `wakeup(&sc->cb)` to unblock any writer waiting for space. When a writer transfers bytes into the buffer, it calls `wakeup(&sc->cb)` to unblock any reader waiting for data. A single channel that wakes “everything on this buffer” is simple and correct.

Some drivers use two separate channels (one for readers, one for writers) so that a reader’s wakeup only disturbs writers and vice versa. That is a valid optimisation when you have many readers or many writers. For a pseudo-device whose expected use is one producer and one consumer, a single channel is both simpler and sufficient.

10.6.7 The Full Stage 3 Handlers

Putting the non-blocking checks into the Stage 2 handlers gives us Stage 3. Here is the full shape:

```

static int
myfirst_read(struct cdev *dev, struct uio *uio, int ioflag)
{
    struct myfirst_softc *sc = dev->si_drv1;
    struct myfirst_fh *fh;
    char bounce[MYFIRST_BOUNCE];
    size_t take, got;
    ssize_t nbefore;
    int error;

    error = devfs_get_cdevpriv((void **)&fh);
    if (error != 0)
        return (error);
    if (sc == NULL || !sc->is_attached)
        return (ENXIO);

    nbefore = uio->uio_resid;

    while (uio->uio_resid > 0) {
        mtx_lock(&sc->mtx);
        while (cbuf_used(&sc->cb) == 0) {

```

```

        if (uio->uio_resid != nbefore) {
            /*
             * We already transferred some bytes
             * in an earlier iteration; report
             * success now rather than block further.
             */
            mtx_unlock(&sc->mtx);
            return (0);
        }
        if (ioflag & IO_NDELAY) {
            mtx_unlock(&sc->mtx);
            return (EAGAIN);
        }
        error = mtx_sleep(&sc->cb, &sc->mtx, PCATCH,
            "myfrd", 0);
        if (error != 0) {
            mtx_unlock(&sc->mtx);
            return (error);
        }
        if (!sc->is_attached) {
            mtx_unlock(&sc->mtx);
            return (ENXIO);
        }
    }
    take = MIN((size_t)uio->uio_resid, sizeof(bounce));
    got = cbuf_read(&sc->cb, bounce, take);
    sc->bytes_read += got;
    fh->reads += got;
    mtx_unlock(&sc->mtx);

    wakeup(&sc->cb);          /* space may have freed for writers */

    error = uiomove(bounce, got, uio);
    if (error != 0)
        return (error);
}
return (0);
}

static int
myfirst_write(struct cdev *dev, struct uio *uio, int ioflag)
{
    struct myfirst_softc *sc = dev->si_drv1;
    struct myfirst_fh *fh;

```

```
char bounce[MYFIRST_BOUNCE];
size_t want, put, room;
ssize_t nbefore;
int error;

error = devfs_get_cdevpriv((void **)&fh);
if (error != 0)
    return (error);
if (sc == NULL || !sc->is_attached)
    return (ENXIO);

nbefore = uio->uio_resid;

while (uio->uio_resid > 0) {
    mtx_lock(&sc->mtx);
    while ((room = cbuf_free(&sc->cb)) == 0) {
        if (uio->uio_resid != nbefore) {
            mtx_unlock(&sc->mtx);
            return (0);
        }
        if (ioflag & IO_NDELAY) {
            mtx_unlock(&sc->mtx);
            return (EAGAIN);
        }
        error = mtx_sleep(&sc->cb, &sc->mtx, PCATCH,
            "myfwr", 0);
        if (error != 0) {
            mtx_unlock(&sc->mtx);
            return (error);
        }
        if (!sc->is_attached) {
            mtx_unlock(&sc->mtx);
            return (ENXIO);
        }
    }
    mtx_unlock(&sc->mtx);

    want = MIN((size_t)uio->uio_resid, sizeof(bounce));
    want = MIN(want, room);
    error = uiomove(bounce, want, uio);
    if (error != 0)
        return (error);

    mtx_lock(&sc->mtx);
```

```

        put = cbuf_write(&sc->cb, bounce, want);
        sc->bytes_written += put;
        fh->writes += put;
        mtx_unlock(&sc->mtx);

        wakeup(&sc->cb);          /* data may have appeared for readers */
    }
    return (0);
}

```

Three patterns in this code are worth studying carefully.

The first is the “**transferred any bytes?**” test at the top of the inner loop. `uio->uio_resid != nbefore` is true if any previous iteration transferred data. When that condition holds and the buffer is now empty (read) or full (write), we return 0 immediately instead of blocking. The kernel will report the partial transfer to user space, and the next call will decide whether to block or to return `EAGAIN`. This is the rule from Section 4 in code form: a handler that has already made progress must return 0, not `EAGAIN`, and not a deeper block.

The second is the **wakeup after progress**. When the reader drains bytes, space has freed; the writer may be waiting for space, and we wake it. When the writer adds bytes, data has appeared; the reader may be waiting for data, and we wake it. Every state change is paired with a wakeup. Missing a wakeup causes threads to sleep forever (or until a timer fires, if one exists); spurious wakeup calls are harmless because the while-loops re-check the condition.

The third is the **order of `mtx_unlock` and `uiomove`**. The handler holds the lock while it manipulates the `cbuf`, then releases the lock *before* calling `uiomove`. `uiomove` may sleep; sleeping under a mutex is a bug. Notice also that on the write side, the handler takes a snapshot of room while holding the lock, uses that snapshot to size the bounce, and releases the lock before `uiomove`. If a concurrent thread has modified the buffer while the handler was copying out of user space, the subsequent `cbuf_write` may store less than `want` bytes (the clamp in `cbuf_write` ensures it is safe). In our current single-writer design this race is never triggered, but the code handles it for free.

10.6.8 Waking Sleepers on Detach

We also need to teach `myfirst_detach` to release any sleepers. The pattern is:

```

static int
myfirst_detach(device_t dev)
{
    struct myfirst_softc *sc = device_get_softc(dev);

    mtx_lock(&sc->mtx);
    if (sc->active_fhs > 0) {
        mtx_unlock(&sc->mtx);
        device_printf(dev,

```

```

        "Cannot detach: %d open descriptor(s)\n",
        sc->active_fhs);
    return (EBUSY);
}
sc->is_attached = 0;
wakeup(&sc->cb);           /* release any sleepers */
mtx_unlock(&sc->mtx);

/* ... destroy_dev, cbuf_destroy, mtx_destroy, sysctl_ctx_free ... */
return (0);
}

```

Two details about this code are specific to Chapter 10.

The first is that we set `is_attached = 0` *before* calling `wakeup`. A sleeper that wakes now will see the flag and return `ENXIO` in the blocking loop; a sleeper that has not yet slept will see the flag and return `ENXIO` without ever sleeping. Setting the flag after `wakeup` would allow a race where a sleeper re-acquires the lock, finds the condition still true (buffer empty), and goes back to sleep, *while* the detach is waiting to tear down the mutex.

The second is that the detach checks `active_fhs > 0` and refuses to proceed if any descriptor is open. This is the same check from Chapter 9. It means a sleeper is always holding a descriptor open, which means the detach will not be running concurrently with a sleeper. The `wakeup` call is there as a belt-and-braces check: if a future refactor ever allows detach while a descriptor is still open, the sleepers will not be stuck.

10.6.9 Adding `d_poll` for `select(2)` and `poll(2)`

A non-blocking caller that receives `EAGAIN` needs some way to be notified when the descriptor becomes ready. `select(2)` and `poll(2)` are the classic mechanisms for that; `kqueue(2)` is the modern one. We will implement the classic two here and leave `kqueue` for Chapter 11 (where `d_kqfilter` and `knlist` infrastructure belongs).

The `d_poll` handler is simple in shape:

```

static int
myfirst_poll(struct cdev *dev, int events, struct thread *td)
{
    struct myfirst_softc *sc = dev->si_drv1;
    int revents = 0;

    mtx_lock(&sc->mtx);
    if (events & (POLLIN | POLLRDNORM)) {
        if (cbuf_used(&sc->cb) > 0)
            revents |= events & (POLLIN | POLLRDNORM);
    }
    else
        selrecord(td, &sc->rsel);
}

```

```

        if (events & (POLLOUT | POLLWRNORM)) {
            if (cbuf_free(&sc->cb) > 0)
                revents |= events & (POLLOUT | POLLWRNORM);
            else
                selrecord(td, &sc->wsel);
        }
        mtx_unlock(&sc->mtx);
        return (revents);
    }
}

```

`d_poll` receives the events the user is interested in and must return the subset that are currently ready. For `POLLIN/POLLRDNORM` (readable), we return ready if the buffer has any bytes. For `POLLOUT/POLLWRNORM` (writable), we return ready if the buffer has any free space. If neither is ready, we call `selrecord(td, &sc->rsl)` or `selrecord(td, &sc->wsl)` to register the calling thread so that we can wake it later.

Two new fields are needed in the `softc`: `struct selinfo rsl;` and `struct selinfo wsl;`. `selinfo` is the kernel's per-condition record of pending `select(2)/poll(2)` waiters. It is declared in `/usr/src/sys/sys/selinfo.h`.

The read and write handlers need a matching `selwakeup(9)` call whenever the buffer transitions from empty to non-empty or from full to non-full. `selwakeup(9)` is the plain form; FreeBSD 14.3 also exposes `selwake-uppri(9)`, which wakes the registered threads at a specified priority and is commonly used by network and storage code that wants latency-sensitive wakeups. For a general-purpose pseudo-device, plain `selwakeup` is the right default. We add the calls next to the `wakeup(&sc->cb)` calls:

```

/* In myfirst_read, after a successful cbuf_read: */
mtx_unlock(&sc->mtx);
wakeup(&sc->cb);
selwakeup(&sc->wsl); /* space is now available for writers */

/* In myfirst_write, after a successful cbuf_write: */
mtx_unlock(&sc->mtx);
wakeup(&sc->cb);
selwakeup(&sc->rsl); /* data is now available for readers */

```

`Attach` initialises the `selinfo` fields with `knlist_init_mtx(&sc->rsl.si_note, &sc->mtx);` and `knlist_init_mtx(&sc->wsl.si_note, &sc->mtx);` if you plan to support `kqueue(2)` later. For pure `select(2)/poll(2)` support, the `selinfo` structure is zero-initialised by the `softc` allocation and needs no further setup.

`Detach` must call `seldrain(&sc->rsl);` and `seldrain(&sc->wsl);` before freeing the `softc`, to tear down any lingering selection records.

Add `.d_poll = myfirst_poll,` to the `myfirst_cdevsw` initialiser, and the driver's `select(2)/poll(2)` story is complete.

10.6.10 How a Non-Blocking Caller Uses All of This

Putting the pieces together, here is what a well-written non-blocking reader looks like against `myfirst`:

```
int fd = open("/dev/myfirst", O_RDONLY | O_NONBLOCK);
char buf[1024];
ssize_t n;
struct pollfd pfd = { .fd = fd, .events = POLLIN };

for (;;) {
    n = read(fd, buf, sizeof(buf));
    if (n > 0) {
        /* got some bytes; process them */
    } else if (n == 0) {
        /* EOF; our driver never reaches this case yet */
        break;
    } else if (errno == EAGAIN) {
        /* no data; wait for readiness */
        poll(&pfd, 1, -1);
    } else if (errno == EINTR) {
        /* signal; retry */
    } else {
        perror("read");
        break;
    }
}
close(fd);
```

The loop reads until it gets data or `EAGAIN`. On `EAGAIN`, it calls `poll(2)` to wait until the kernel reports the descriptor readable, then loops back. The `POLLIN` event will be reported when `myfirst_write` runs the `selwakeup(&sc->rsl)` call that follows a successful `cbuf_write`. The driver's `d_poll` is the bridge between the kernel's `select/poll` machinery and the buffer's state.

This is the canonical shape of event-driven UNIX I/O, and your driver now participates in it correctly.

10.6.11 A Note About `O_NONBLOCK` and `select/poll` Composition

It is worth understanding how `select(2)` / `poll(2)` and `O_NONBLOCK` interact. The conventional rule is that a program uses both together: it registers the descriptor with `poll` and then reads from it. Using either alone is valid but less common.

If a program uses `O_NONBLOCK` without `poll`, it will busy-spin. On every `EAGAIN` it will have to `sleep` or `usleep` before retrying, wasting cycles for no good reason. This is almost always wrong, but it works.

If a program uses `poll` without `O_NONBLOCK`, the `poll` reports readiness and then `read(2)` does a blocking call. The blocking call will complete almost immediately in the normal case, because the condition was just reported

ready. However, in the rare case where the kernel's state changes between the `poll` return and the `read` call (another thread drained the buffer, for instance), the `read` will block indefinitely. This is a subtle bug, and most event-driven libraries defend against it by always combining `poll` with `O_NONBLOCK`.

The `myfirst` driver supports both patterns correctly. A well-written program combines the two; a less-careful program will work in simple cases and have the corner case described above.

10.6.12 Observing the Blocking Path in Action

Load the Stage 3 driver and run a quick experiment:

```
$ kldload ./myfirst.ko
$ cat /dev/myfirst &
[1] 12345
```

The `cat` is now blocked inside `myfirst_read`, sleeping on `&sc->cb`. You can confirm with `ps`:

```
$ ps -AxH | grep cat
12345  -  S+      0:00.00 myfrd
```

The `S+` state indicates that the process is sleeping, and the `wmesg` column shows `myfrd`, which is exactly the string we passed to `mtx_sleep`. Now write into the driver from another terminal:

```
$ echo hello > /dev/myfirst
```

The `cat` wakes up, reads `hello`, and either prints it and blocks again, or (if the device is closed by the writer) reaches end-of-file and exits. In our current Stage 3 there is no “writer has closed” mechanism, so the `cat` blocks again after printing. Use `Ctrl-C` in its terminal to interrupt it:

```
$ kill -INT %1
```

Because we passed `PCATCH` to `mtx_sleep`, the signal wakes the sleeper, which returns `EINTR`, which propagates out to `cat` as a failed `read(2)`. `cat` sees it, notices the signal, and exits cleanly.

This is the whole blocking path in action. Nothing mysterious happened; each piece is visible in the source and in `ps`.

10.6.13 Common Mistakes in the Blocking Path

Two mistakes are especially common in this material.

Mistake 1: forgetting to release the mutex before returning `EAGAIN`. The code above explicitly unlocks before every return in the sleep loop. If you forget one of those unlocks, subsequent attempts to take the mutex will panic or deadlock. A `WITNESS` kernel will catch this immediately in a lab environment.

Mistake 2: using `tsleep(9)` when you should use `mtx_sleep(9)`. `tsleep` does not take a mutex argument; it assumes the caller is not holding any interlock. In a driver that uses `mtx_sleep`, the mutex is dropped atomically with the sleep; with `tsleep`, you would have to drop the mutex yourself and then re-acquire it after waking, introducing a race window where a producer can add data and call wakeup before you are back on the sleep queue. `mtx_sleep` is the correct primitive for every case where you hold a mutex and want to sleep while releasing it.

Mistake 3: not handling the `PCATCH` return values. `mtx_sleep` with `PCATCH` can return 0, `EINTR`, `ERESTART`, or `EWOULDBLOCK` (for timeouts). In driver code the conventional thing is to return error without inspecting it further; the kernel knows how to translate `ERESTART` into a syscall restart when the process's signal disposition allows it. Inspecting the value and returning 0 only for `error == 0` is the pattern in the Stage 3 code above.

Mistake 4: using different “channels” for `mtx_sleep` and wakeup. The sleeper uses `&sc->cb` as the channel; the waker must use exactly the same address. A common bug is for one site to use `sc` (the softc pointer) and another to use `&sc->cb`. The sleepers will never wake until a timeout fires or a different wakeup happens to match. Double-check that every `mtx_sleep` / wakeup pair uses the same channel.

10.6.14 Wrapping Up Section 5

The driver now handles both blocking and non-blocking callers correctly. A blocking reader sleeps on an empty buffer and wakes when a writer deposits data. A non-blocking reader receives `EAGAIN` immediately on an empty buffer. The symmetric pair applies to writers. `select(2)` and `poll(2)` are supported through `d_poll` and the `selinfo` machinery, and a well-behaved event-loop program can now multiplex `/dev/myfirst` with other descriptors. `Detach` releases any sleepers before tearing the driver down.

What you have built is a competently behaved character device. It moves bytes efficiently, cooperates with the kernel's readiness and sleep primitives, and honours the user-facing conventions of UNIX I/O. What remains in the rest of the chapter is to test it rigorously (Section 6), refactor it for concurrency work (Section 7), and explore three supplementary topics that often appear alongside this material in real drivers (`d_mmap`, zero-copy thinking, and the throughput patterns of readahead and write coalescing).

10.7 Section 6: Testing Buffered I/O with User Programs

A driver is only as reliable as the tests you run against it. Chapters 7 through 9 established a small test kit (a short `rw_myfirst` exerciser, plus `cat`, `echo`, `dd`, and `hexdump`). Chapter 10 pushes that kit further because the new behaviour the driver now exhibits (blocking, non-blocking, partial I/O, wrap-around) only shows up under realistic load. This section builds out three new user-space tools and walks through a consolidated test plan you can work through after each stage.

The tools in this section live under `examples/part-02/ch10-handling-io-efficiently/userland/`. They are deliberately small. The longest is under 150 lines. Each one exists to exercise a specific pattern the driver is now supposed to handle, and each one produces output that you can read and verify.

10.7.1 Three Tools We Will Build

`rw_myfirst_nb.c` is a non-blocking tester. It opens the device with `O_NONBLOCK`, issues a read, expects `EAGAIN`, writes some bytes, issues another read, expects to receive them, and reports a one-line summary of each step.

This is the smallest tool that exercises the non-blocking path end to end.

`producer_consumer.c` is a fork-based load harness. It spawns a child process that writes random bytes into the driver at a configurable rate, while the parent reads them out and verifies integrity. The purpose is to exercise the circular buffer's wrap-around and the blocking path under real concurrent load.

`stress_rw.c` (evolved from Chapter 9's version) is a single-process stress tester that runs through a table of (block size, transfer count) combinations and prints aggregate timing and byte-counter statistics. The purpose is to catch performance cliffs that a single interactive test would not reveal.

All three compile with a short Makefile that we will show at the end.

10.7.2 Updating `rw_myfirst` for Larger Inputs

The existing `rw_myfirst` from Chapter 9 handles text-sized transfers well but does not stress the buffer at volume. A simple extension lets it take a size argument on the command line:

```
/* rw_myfirst_v2.c: an incremental improvement on Chapter 9's tester. */
#include <sys/types.h>
#include <err.h>
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#define DEVPATH "/dev/myfirst"

static int
do_fill(size_t bytes)
{
    int fd = open(DEVPATH, O_WRONLY);
    if (fd < 0)
        err(1, "open %s", DEVPATH);

    char *buf = malloc(bytes);
    if (buf == NULL)
        err(1, "malloc %zu", bytes);
    for (size_t i = 0; i < bytes; i++)
        buf[i] = (char)('A' + (i % 26));

    size_t left = bytes;
    ssize_t n;
    const char *p = buf;
    while (left > 0) {
```

```
        n = write(fd, p, left);
        if (n < 0) {
            if (errno == EINTR)
                continue;
            warn("write at %zu left", left);
            break;
        }
        p += n;
        left -= n;
    }
    size_t wrote = bytes - left;
    printf("fill: wrote %zu of %zu\n", wrote, bytes);
    free(buf);
    close(fd);
    return (0);
}

static int
do_drain(size_t bytes)
{
    int fd = open(DEVPATH, O_RDONLY);
    if (fd < 0)
        err(1, "open %s", DEVPATH);

    char *buf = malloc(bytes);
    if (buf == NULL)
        err(1, "malloc %zu", bytes);

    size_t left = bytes;
    ssize_t n;
    char *p = buf;
    while (left > 0) {
        n = read(fd, p, left);
        if (n < 0) {
            if (errno == EINTR)
                continue;
            warn("read at %zu left", left);
            break;
        }
        if (n == 0) {
            printf("drain: EOF at %zu left\n", left);
            break;
        }
        p += n;
    }
```

```
        left -= n;
    }
    size_t got = bytes - left;
    printf("drain: read %zu of %zu\n", got, bytes);
    free(buf);
    close(fd);
    return (0);
}

int
main(int argc, char *argv[])
{
    if (argc != 3) {
        fprintf(stderr, "usage: %s fill|drain BYTES\n", argv[0]);
        return (1);
    }
    size_t bytes = strtoul(argv[2], NULL, 0);
    if (strcmp(argv[1], "fill") == 0)
        return (do_fill(bytes));
    if (strcmp(argv[1], "drain") == 0)
        return (do_drain(bytes));
    fprintf(stderr, "unknown mode: %s\n", argv[1]);
    return (1);
}
```

With this tool, you can drive the driver with realistic sizes. For example:

```
$ ./rw_myfirst_v2 fill 4096
fill: wrote 4096 of 4096
$ sysctl dev.myfirst.0.stats.cb_used
dev.myfirst.0.stats.cb_used: 4096
$ ./rw_myfirst_v2 drain 4096
drain: read 4096 of 4096
$ sysctl dev.myfirst.0.stats.cb_used
dev.myfirst.0.stats.cb_used: 0
```

Now try to fill the buffer beyond its capacity and watch what happens at each stage.

10.7.3 Why a Round-Trip Test Matters

Every serious test you write should have a *round-trip* component: write a known pattern into the driver, read it back, and compare. The pattern matters because if you write “Hello, world!” ten times you cannot tell whether the buffer got 140 bytes of “Hello, world!” or 130 or 150 or some weird interleaving. A unique per-position

pattern (like 'A' + (i % 26) above) lets you spot misalignment, missing bytes, and duplicated bytes at a glance.

Round-trip testing is especially important for circular buffers, because the wrap-around arithmetic is the thing that beginner code gets wrong. A write that pushes past the end of the underlying storage and a read that picks up from before the start are the two failure modes you most want to catch. Both appear as “the bytes I read are not the bytes I wrote”, and a round-trip test makes them visible immediately.

10.7.4 Building `rw_myfirst_nb`

This is the non-blocking tester. It is slightly longer than the previous file but still short enough to read in one sitting.

```
/* rw_myfirst_nb.c: non-blocking behaviour tester for /dev/myfirst. */
#include <sys/types.h>
#include <err.h>
#include <errno.h>
#include <fcntl.h>
#include <poll.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

#define DEVPATH "/dev/myfirst"

int
main(void)
{
    int fd, error;
    ssize_t n;
    char rbuf[128];
    struct pollfd pfd;

    fd = open(DEVPATH, O_RDWR | O_NONBLOCK);
    if (fd < 0)
        err(1, "open %s", DEVPATH);

    /* Expect EAGAIN when the buffer is empty. */
    n = read(fd, rbuf, sizeof(rbuf));
    if (n < 0 && errno == EAGAIN)
        printf("step 1: empty-read returned EAGAIN (expected)\n");
    else
        printf("step 1: UNEXPECTED read returned %zd errno=%d\n", n, errno);

    /* poll(POLLIN) with timeout 0 should show not-readable. */
```

```

    pfd.fd = fd;
    pfd.events = POLLIN;
    pfd.revents = 0;
    error = poll(&pfd, 1, 0);
    printf("step 2: poll(POLLIN, 0) = %d revents=0x%x\n",
        error, pfd.revents);

    /* Write some bytes. */
    n = write(fd, "hello world\n", 12);
    printf("step 3: wrote %zd bytes\n", n);

    /* poll(POLLIN) should now show readable. */
    pfd.events = POLLIN;
    pfd.revents = 0;
    error = poll(&pfd, 1, 0);
    printf("step 4: poll(POLLIN, 0) = %d revents=0x%x\n",
        error, pfd.revents);

    /* Non-blocking read should now succeed. */
    memset(rbuf, 0, sizeof(rbuf));
    n = read(fd, rbuf, sizeof(rbuf));
    if (n > 0) {
        rbuf[n] = '\0';
        printf("step 5: read %zd bytes: %s", n, rbuf);
    } else
        printf("step 5: UNEXPECTED read returned %zd errno=%d\n",
            n, errno);

    close(fd);
    return (0);
}

```

Expected output against Stage 3 (non-blocking support) is:

```

step 1: empty-read returned EAGAIN (expected)
step 2: poll(POLLIN, 0) = 0 revents=0x0
step 3: wrote 12 bytes
step 4: poll(POLLIN, 0) = 1 revents=0x41
step 5: read 12 bytes: hello world

```

The 0x41 in step 4 is POLLIN | POLLRDNORM, which is exactly what our `d_poll` handler sets when the buffer has live bytes.

If step 1 fails (i.e. `read(2)` returns 0 instead of `-1/EAGAIN`), your driver is still running the Stage 2 semantics. Go back and add the `IO_NDELAY` check in the handlers.

If step 2 succeeds with `revents != 0`, your `d_poll` is wrongly reporting readable on an empty buffer. Check the condition in `myfirst_poll`.

If step 4 returns zero (i.e. `poll(2)` did not find the descriptor readable), your `d_poll` is not reflecting the buffer state correctly, or the `selwakeup` call is missing from the write path.

These are the three most common non-blocking bugs. The tester catches all of them in under fifty lines of output.

10.7.5 Building `producer_consumer.c`

This is the fork-based load harness. The shape is straightforward: fork a child that writes, have the parent read, and compare what comes out against what went in.

```
/* producer_consumer.c: a two-process load test for /dev/myfirst. */
#include <sys/types.h>
#include <sys/wait.h>
#include <err.h>
#include <errno.h>
#include <fcntl.h>
#include <signal.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#define DEVPATH      "/dev/myfirst"
#define TOTAL_BYTES (1024 * 1024)
#define BLOCK       4096

static uint32_t
checksum(const char *p, size_t n)
{
    uint32_t s = 0;
    for (size_t i = 0; i < n; i++)
        s = s * 31u + (uint8_t)p[i];
    return (s);
}

static int
do_writer(void)
{
    int fd = open(DEVPATH, O_WRONLY);
    if (fd < 0)
        err(1, "writer: open");
}
```

```
char *buf = malloc(BLOCK);
if (buf == NULL)
    err(1, "writer: malloc");

size_t written = 0;
uint32_t sum = 0;
while (written < TOTAL_BYTES) {
    size_t left = TOTAL_BYTES - written;
    size_t block = left < BLOCK ? left : BLOCK;
    for (size_t i = 0; i < block; i++)
        buf[i] = (char)((written + i) & 0xff);
    sum += checksum(buf, block);

    const char *p = buf;
    size_t remain = block;
    while (remain > 0) {
        ssize_t n = write(fd, p, remain);
        if (n < 0) {
            if (errno == EINTR)
                continue;
            warn("writer: write");
            close(fd);
            return (1);
        }
        p += n;
        remain -= n;
    }
    written += block;
}

printf("writer: %zu bytes, checksum 0x%08x\n", written, sum);
close(fd);
free(buf);
return (0);
}

static int
do_reader(void)
{
    int fd = open(DEVPATH, O_RDONLY);
    if (fd < 0)
        err(1, "reader: open");
}
```

```
char *buf = malloc(BLOCK);
if (buf == NULL)
    err(1, "reader: malloc");

size_t got = 0;
uint32_t sum = 0;
int mismatches = 0;
while (got < TOTAL_BYTES) {
    ssize_t n = read(fd, buf, BLOCK);
    if (n < 0) {
        if (errno == EINTR)
            continue;
        warn("reader: read");
        break;
    }
    if (n == 0) {
        /* Only reached if driver signals EOF. */
        printf("reader: EOF at %zu\n", got);
        break;
    }
    for (ssize_t i = 0; i < n; i++) {
        if ((uint8_t)buf[i] != (uint8_t)((got + i) & 0xff))
            mismatches++;
    }
    sum += checksum(buf, n);
    got += n;
}

printf("reader: %zu bytes, checksum 0x%08x, mismatches %d\n",
       got, sum, mismatches);
close(fd);
free(buf);
return (mismatches == 0 ? 0 : 2);
}

int
main(void)
{
    pid_t pid = fork();
    if (pid < 0)
        err(1, "fork");
    if (pid == 0) {
        /* child: writer */
        _exit(do_writer());
    }
}
```

```
    }  
    /* parent: reader */  
    int rc = do_reader();  
    int status;  
    waitpid(pid, &status, 0);  
    int wexit = WIFEXITED(status) ? WEXITSTATUS(status) : -1;  
    printf("exit: reader=%d writer=%d\n", rc, wexit);  
    return (rc || wexit);  
}
```

The test works best against Stage 3 or Stage 4 of the chapter. Against Stage 2 (no blocking), the writer will receive short writes and the reader will occasionally see zero-byte reads, and the total bytes transferred may be less than TOTAL_BYTES. Against Stage 3, both sides block and unblock at the right times, the test runs to completion, and the two checksums match.

A successful run looks like:

```
$ ./producer_consumer  
writer: 1048576 bytes, checksum 0x12345678  
reader: 1048576 bytes, checksum 0x12345678, mismatches 0  
exit: reader=0 writer=0
```

Mismatches are the killer. If the writer's checksum matches the reader's but mismatches are non-zero, it means a byte drifted in position during the round trip (likely a wrap-around bug). If the checksums differ, a byte was lost or duplicated (likely a locking bug). If the test hangs forever, the blocking path's condition never becomes true (likely a missing wakeup).

10.7.6 Using dd(1) for Volume Tests

The base-system `dd(1)` is the fastest way to push volume through the driver without writing any new code. A few patterns are especially useful.

Pattern 1: writer-only. Push a large amount of data into the driver while a reader keeps up.

```
$ dd if=/dev/myfirst of=/dev/null bs=4k &  
$ dd if=/dev/zero of=/dev/myfirst bs=4k count=100000
```

This produces 400 MB of traffic through the driver. Watch `sysctl dev.myfirst.0.stats.bytes_written` grow, and compare it against `bytes_read`; the difference is roughly the buffer fill level.

Pattern 2: rate-limited. Some tests want to stress the driver at a steady rate rather than at maximum throughput. Use `rate` or the GNU `pv(1)` utility (available as `ports/sysutils/pv`) to cap:

```
$ pv -L 10m < /dev/zero | dd of=/dev/myfirst bs=4k
```

This caps the write rate at 10 MB/s. A slower rate lets you observe the buffer’s fill level in `sysctl` and see the blocking path engage when the rate approaches the consumer’s rate.

Pattern 3: full-block. As mentioned in Section 4, default `dd` does not loop on short reads. Use `iflag=fullblock` to make it do so:

```
$ dd if=/dev/myfirst of=/tmp/out bs=4k count=100 iflag=fullblock
```

Without `iflag=fullblock`, the output file might be shorter than the requested 400 KB because of short reads.

10.7.7 Using `hexdump(1)` to Verify Content

`hexdump(1)` is the right tool for verifying the *content* of what the driver delivers. If you write a known byte sequence and want to confirm it comes back intact, `hexdump` shows you.

```
$ printf 'ABCDEFGH' > /dev/myfirst
$ hexdump -C /dev/myfirst
00000000  41 42 43 44 45 46 47 48          |ABCDEFGH|
$
```

The `hexdump -C` output is the canonical “here are the bytes and their ASCII interpretation” format. It is especially useful when the driver is emitting binary data that text-based tools cannot display.

10.7.8 Using `truss(1)` to See Syscall Traffic

`truss(1)` traces the system calls made by a process. Running a test under `truss` shows you exactly what each `read(2)` and `write(2)` returned, including partial transfers and error codes.

```
$ truss -t read,write -o /tmp/trace ./rw_myfirst_nb
$ head /tmp/trace
read(3,0x7fffffffeca0,128)          ERR#35 'Resource temporarily
↳ unavailable'
write(3,"hello world\n",12)          = 12 (0xc)
read(3,0x7fffffffeca0,128)          = 12 (0xc)
...
```

ERR#35 is EAGAIN. Seeing it confirms that the non-blocking path is engaging. Running `producer_consumer` under `truss` shows the pattern of short writes and short reads very clearly; it is a good diagnostic for debugging buffer sizing issues.

A related tool is `kttrace(1)` / `kdump(1)`, which produces a more detailed and decoded trace, at the cost of being a bit more verbose. Either is fine for this level of work.

10.7.9 Using `sysctl(8)` to Watch State Live

The `sysctl` tree `dev.myfirst.0.stats.*` is the driver's live state. Watching it in real time during a test tells you a great deal about what the driver is doing.

```
$ while true; do
    clear
    sysctl dev.myfirst.0.stats | egrep 'cb_|bytes_'
    sleep 1
done
```

Run this in one terminal while a test runs in another. You will see `cb_used` rise as the writer gets ahead, fall as the reader catches up, and oscillate around some steady-state level. The byte counters only ever increase. A stalled test shows up as frozen counters.

10.7.10 Using `vmstat -m` to Watch Memory

If you suspect a leak (maybe you forgot `cbuf_destroy` in the error path of `attach`), `vmstat -m` shows it:

```
$ vmstat -m | grep cbuf
      cbuf      1      4K      1  4096
```

After `kldunload`:

```
$ vmstat -m | grep cbuf
$
```

The tag should disappear entirely when the driver is unloaded. If a count is non-zero, something still holds the allocation. This is the kind of regression you want to catch immediately; it gets worse silently over time.

10.7.11 Building the Test Kit

Here is a Makefile that builds all the userland test programs at once. Place it in `examples/part-02/ch10-handling-io-efficiently/userland/`:

```
# Makefile for Chapter 10 userland testers.

PROGS= rw_myfirst_v2 rw_myfirst_nb producer_consumer stress_rw cb_test

.PHONY: all
all: ${PROGS}

CFLAGS?= -O2 -Wall -Wextra -Wno-unused-parameter
```

```
rw_myfirst_v2: rw_myfirst_v2.c
    ${CC} ${CFLAGS} -o $@ $<

rw_myfirst_nb: rw_myfirst_nb.c
    ${CC} ${CFLAGS} -o $@ $<

producer_consumer: producer_consumer.c
    ${CC} ${CFLAGS} -o $@ $<

stress_rw: stress_rw.c
    ${CC} ${CFLAGS} -o $@ $<

cb_test: ../cbuf-userland/cbuf.c ../cbuf-userland/cb_test.c
    ${CC} ${CFLAGS} -I../cbuf-userland -o $@ $^

.PHONY: clean
clean:
    rm -f ${PROGS}
```

Running make builds all four tools. make cb_test builds only the standalone cbuf test. Keep the two userland directories (cbuf-userland/ for the buffer, userland/ for the driver testers) separate; the first one is the prerequisite for the later stages, and building it in isolation mirrors the order we introduced them in the chapter.

10.7.12 A Consolidated Test Plan

With the tools in place, here is a test plan you can run against each stage of the driver. Run each pass after loading the corresponding myfirst.ko.

Stage 2 (circular buffer, no blocking):

1. ./rw_myfirst_v2 fill 4096; sysctl dev.myfirst.0.stats.cb_used should report 4096.
2. ./rw_myfirst_v2 fill 4097 should show a short write (wrote 4096 of 4097).
3. ./rw_myfirst_v2 drain 2048; sysctl dev.myfirst.0.stats.cb_used should report 2048.
4. ./rw_myfirst_v2 fill 2048; sysctl dev.myfirst.0.stats.cb_used should report 4096, but cb_head should be non-zero (proving wrap-around worked).
5. dd if=/dev/myfirst of=/dev/null bs=4k: should drain 4096 bytes and then return zero.
6. producer_consumer with TOTAL_BYTES = 8192: should complete successfully.

Stage 3 (blocking and non-blocking support):

1. cat /dev/myfirst & should block.
2. echo hi > /dev/myfirst should produce output in the cat terminal.
3. kill -INT %1 should unblock cat cleanly.
4. ./rw_myfirst_nb should print the six-line output above.

5. `producer_consumer` with `TOTAL_BYTES = 1048576`: should complete with no mismatches and matching checksums.

Stage 4 (poll support, refactored helpers):

All Stage 3 tests, plus:

1. `./rw_myfirst_nb` step 4 should show `revents=0x41 (POLLIN|POLLRDNORM)`.
2. A small program that opens one descriptor read-only non-blocking, registers it with `poll(POLLIN)` with timeout -1, and calls `write` from the same process on a second descriptor: the `poll` should return promptly with `POLLIN` set.
3. `dd if=/dev/zero of=/dev/myfirst bs=1m count=10` & paired with `dd if=/dev/myfirst of=/dev/null bs=4k`: should move 10 MB with no errors, roughly in the time the slower side takes.

This plan is by no means exhaustive. The Labs section later in the chapter gives you a deeper sequence. But these are the smoke tests: run them after every non-trivial change, and if they pass, you have not broken anything fundamental.

10.7.13 Debugging When a Test Fails

When a test fails, the sequence of inspection is usually:

1. **`dmesg | tail -100`**: check for kernel warnings, panics, or your own `device_printf` output. If the kernel is complaining about a locking violation or a witness warning, the problem is visible here before you do anything else.
2. **`sysctl dev.myfirst.0.stats`**: compare the current values to what they should be. If `cb_used` is non-zero but no one is holding a descriptor open, something went wrong with the shutdown path.
3. **`truss -t read,write,poll -f`**: run the failing tester under `truss` and see the syscall returns. A spurious `EAGAIN` (or absence thereof) shows up immediately.
4. **`ktrace`**: if `truss` is not enough, `ktrace -di ./test; kdump -f ktrace.out` gives a deeper view including signals.
5. **Add `device_printf` to the driver**: sprinkle one-line traces at the top and bottom of each handler, then reproduce the test. This is the fallback, and it is sometimes the only way to see what the driver is doing at the moments the user-side tools do not capture.

Be careful with the last step. Every `device_printf` goes through the kernel's log buffer, which is itself a finite circular buffer. Dropping a `device_printf` into the `cbuf_write` function that runs every byte will melt the log. Start with one log line per I/O call and increase only if needed.

10.7.14 Wrapping Up Section 6

You now have a test kit that can exercise every non-trivial behaviour the driver promises. `rw_myfirst_v2` covers sized reads and writes and round-trip correctness. `rw_myfirst_nb` covers the non-blocking path and the `poll(2)` contract. `producer_consumer` covers concurrent two-party load with content verification. `dd`, `cat`, `hexdump`, `truss`, `sysctl`, and `vmstat -m` together provide observability into the driver's internal state.

None of these tools are new or exotic. They are standard FreeBSD base-system utilities, and short pieces of code that you can type in an afternoon. The combination is enough to catch most driver bugs before they reach

anyone else's hands. The next section takes the driver you have just finished testing and prepares its code shape for the concurrency work in Chapter 11.

10.8 Section 7: Refactoring and Preparing for Concurrency

The driver works. It buffers, it blocks, it poll-reports correctly, and the user-space tests in Section 6 confirm that bytes flow correctly under realistic load. What we have not yet done is shape the code for the work Chapter 11 will do. This section is the bridge: it identifies the places in the current source that will need attention from a real concurrency standpoint, refactors the buffer access into a tight set of helpers, and finishes by making the driver as honest as possible about its own state.

We are not introducing new locking primitives here. Chapter 11 will explore that material at length, including the alternatives to a single mutex (sleepable locks, sx locks, rwlocks, lock-free patterns), the verification tools (WITNESS, INVARIANTS), and the rules around interrupt context, sleep, and lock ordering. What we are doing in Section 7 is making the *shape* of the code such that those tools can be applied cleanly when the time comes.

10.8.1 Identifying Potential Race Conditions

A “race condition” in driver code is any place where the correctness of the code depends on the order in which two threads execute, where the order is not enforced by anything in the driver. The Stage 4 driver has the right *machinery* in place (a mutex, a sleep channel, sleep-with-mutex semantics through `mtx_sleep`) and the I/O handlers respect it. But there are still places where a careful audit is worthwhile.

Let us walk through the data structures and ask, of each shared field, “who reads it, who writes it, what protects access?”

sc->cb (the circular buffer). Read by `myfirst_read`, written by `myfirst_write`, read by `myfirst_poll`, read by the two sysctl handlers (`cb_used` and `cb_free`), read by `myfirst_detach` (implicitly through `cbuf_destroy`). Protected by `sc->mtx` everywhere it is touched. *Looks safe.*

sc->bytes_read, sc->bytes_written. Updated by the two I/O handlers under `sc->mtx`. Read by sysctl directly through `SYSCTL_ADD_U64` (no handler interposed). The sysctl read is a single 64-bit load on most architectures, which is a torn-read risk on some 32-bit platforms but is atomic on amd64 and arm64. *Mostly safe; see torn-read note below.*

sc->open_count, sc->active_fhs. Updated under `sc->mtx`. Read by sysctl directly. Same torn-read consideration.

sc->is_attached. Read by every handler at entry, set by `attach` (without lock, before `make_dev`), cleared by `detach` (under lock). The unlocked write at attach time is safe because no one else can see the device yet. The locked clear at detach time is correctly ordered with the wakeup. *Looks safe.*

sc->cdev, sc->cdev_alias. Set by `attach`, cleared by `detach`. Once attach is done, these are stable for the lifetime of the device. The handlers reach the softc through `dev->si_drv1` (set during attach) and never dereference these directly during I/O. *Safe by construction.*

sc->rsel, sc->wsel. The `selinfo` machinery is internally locked (it uses the kernel's `selspinlock` and per-mutex `knlist` if you initialise one). For pure `select(2)/poll(2)` use, the `selrecord` and `selwakeup` calls handle their own concurrency. *Safe.*

sc->open_count and friends, again. The torn-read note above is worth being explicit about. On 32-bit platforms (i386, armv7), a 64-bit field can be split across two memory operations, and a concurrent write can yield a read that contains the high half of one value and the low half of another (a “torn read”). The chapter is targeting amd64 where this is not an issue, but it is the kind of thing a real driver should think about. The fix, if needed, is to add a sysctl handler (like the `cb_used` one) that takes the mutex around the load.

The audit above gives a clean bill of health. The bigger refactoring opportunities are not race conditions but *code shape*: places where the buffer logic is mixed in with the I/O logic, where helper functions would clarify intent, and where Chapter 11 can introduce new lock classes without touching the I/O handlers.

10.8.2 The Refactor: Pulling Buffer Access into Helpers

The Stage 3 / Stage 4 handlers contain a fair amount of locking and book-keeping inline. Let us extract that into a small set of helpers. The goal is two-fold: the I/O handlers become obviously correct, and Chapter 11 can substitute different locking strategies into the helpers without touching `myfirst_read` or `myfirst_write`.

Define the following helpers, all in `myfirst.c` (or in a new file `myfirst_buf.c` if you want clearer separation):

```
/* Read up to "n" bytes from the cbuf into "dst". Returns count moved. */
static size_t
myfirst_buf_read(struct myfirst_softc *sc, void *dst, size_t n)
{
    size_t got;

    mtx_assert(&sc->mtx, MA_OWNED);
    got = cbuf_read(&sc->cb, dst, n);
    sc->bytes_read += got;
    return (got);
}

/* Write up to "n" bytes from "src" into the cbuf. Returns count moved. */
static size_t
myfirst_buf_write(struct myfirst_softc *sc, const void *src, size_t n)
{
    size_t put;

    mtx_assert(&sc->mtx, MA_OWNED);
    put = cbuf_write(&sc->cb, src, n);
    sc->bytes_written += put;
    return (put);
}

/* Wait, with PCATCH, until the cbuf is non-empty or the device tears down. */
static int
myfirst_wait_data(struct myfirst_softc *sc, int ioflag, ssize_t nbefore,
```

```

    struct uio *uio)
{
    int error;

    mtx_assert(&sc->mtx, MA_OWNED);
    while (cbuf_used(&sc->cb) == 0) {
        if (uio->uio_resid != nbfore)
            return (-1);           /* signal caller to break */
        if (ioflag & IO_NDELAY)
            return (EAGAIN);
        error = mtx_sleep(&sc->cb, &sc->mtx, PCATCH, "myfrd", 0);
        if (error != 0)
            return (error);
        if (!sc->is_attached)
            return (ENXIO);
    }
    return (0);
}

/* Wait, with PCATCH, until the cbuf has free space or the device tears down. */
static int
myfirst_wait_room(struct myfirst_softc *sc, int ioflag, ssize_t nbfore,
    struct uio *uio)
{
    int error;

    mtx_assert(&sc->mtx, MA_OWNED);
    while (cbuf_free(&sc->cb) == 0) {
        if (uio->uio_resid != nbfore)
            return (-1);           /* signal caller to break */
        if (ioflag & IO_NDELAY)
            return (EAGAIN);
        error = mtx_sleep(&sc->cb, &sc->mtx, PCATCH, "myfwr", 0);
        if (error != 0)
            return (error);
        if (!sc->is_attached)
            return (ENXIO);
    }
    return (0);
}

```

The `mtx_assert(&sc->mtx, MA_OWNED)` calls are a tiny but valuable safety net. If a future caller forgets to acquire the lock before calling one of these helpers, the assertion fires (in a WITNESS kernel). Once you trust the helpers, you can stop thinking about the lock at the call sites.

The four helpers together cover everything the I/O handlers need from the buffer abstraction: reading bytes, writing bytes, waiting for data, waiting for room. Each helper takes the mutex by reference and asserts that it is held. None of them lock or unlock.

With the helpers defined, the I/O handlers shrink considerably:

```
static int
myfirst_read(struct cdev *dev, struct uio *uio, int ioflag)
{
    struct myfirst_softc *sc = dev->si_drv1;
    struct myfirst_fh *fh;
    char bounce[MYFIRST_BOUNCE];
    size_t take, got;
    ssize_t nbefore;
    int error;

    error = devfs_get_cdevpriv((void **)&fh);
    if (error != 0)
        return (error);
    if (sc == NULL || !sc->is_attached)
        return (ENXIO);

    nbefore = uio->uio_resid;
    while (uio->uio_resid > 0) {
        mtx_lock(&sc->mtx);
        error = myfirst_wait_data(sc, ioflag, nbefore, uio);
        if (error != 0) {
            mtx_unlock(&sc->mtx);
            return (error == -1 ? 0 : error);
        }
        take = MIN((size_t)uio->uio_resid, sizeof(bounce));
        got = myfirst_buf_read(sc, bounce, take);
        fh->reads += got;
        mtx_unlock(&sc->mtx);

        wakeup(&sc->cb);
        selwakeup(&sc->wsel);

        error = uiomove(bounce, got, uio);
        if (error != 0)
            return (error);
    }
    return (0);
}
```

```
static int
myfirst_write(struct cdev *dev, struct uio *uio, int ioflag)
{
    struct myfirst_softc *sc = dev->si_drv1;
    struct myfirst_fh *fh;
    char bounce[MYFIRST_BOUNCE];
    size_t want, put, room;
    ssize_t nbefore;
    int error;

    error = devfs_get_cdevpriv((void **)&fh);
    if (error != 0)
        return (error);
    if (sc == NULL || !sc->is_attached)
        return (ENXIO);

    nbefore = uio->uio_resid;
    while (uio->uio_resid > 0) {
        mtx_lock(&sc->mtx);
        error = myfirst_wait_room(sc, ioflag, nbefore, uio);
        if (error != 0) {
            mtx_unlock(&sc->mtx);
            return (error == -1 ? 0 : error);
        }
        room = cbuf_free(&sc->cb);
        mtx_unlock(&sc->mtx);

        want = MIN((size_t)uio->uio_resid, sizeof(bounce));
        want = MIN(want, room);
        error = uiomove(bounce, want, uio);
        if (error != 0)
            return (error);

        mtx_lock(&sc->mtx);
        put = myfirst_buf_write(sc, bounce, want);
        fh->writes += put;
        mtx_unlock(&sc->mtx);

        wakeup(&sc->cb);
        selwakeup(&sc->rsel);
    }
    return (0);
}
```

Each I/O handler now does the same three things in the same order: take the lock, ask the helper for state, release the lock, do the copy, take the lock again, update state, release, wake. The pattern is clear enough that a future reader can verify the locking discipline at a glance.

The error code -1 returned by the wait helpers is a small convention: “no error to report, but the loop should break and the caller should return 0.” Using -1 (which is not a valid errno) makes the convention obvious without adding a third out-parameter. It is local to the driver and never escapes to user space.

10.8.3 Documenting the Locking Strategy

A driver this size benefits from a one-paragraph comment near the top of the file explaining the locking discipline. The comment is for the next person who reads the code, and it is your future self in three months. Add this near the `struct myfirst_softc` declaration:

```
/*
 * Locking strategy.
 *
 * sc->mtx protects:
 *   - sc->cb (the circular buffer's internal state)
 *   - sc->bytes_read, sc->bytes_written
 *   - sc->open_count, sc->active_fhs
 *   - sc->is_attached
 *
 * Locking discipline:
 *   - The mutex is acquired with mtx_lock and released with mtx_unlock.
 *   - mtx_sleep(&sc->cb, &sc->mtx, PCATCH, ...) is used to block while
 *     waiting on buffer state. wakeup(&sc->cb) is the matching call.
 *   - The mutex is NEVER held across uiomove(9), copyin(9), or copyout(9),
 *     all of which may sleep.
 *   - The mutex is held when calling cbuf_*() helpers; the cbuf module is
 *     intentionally lock-free by itself and relies on the caller for safety.
 *   - selwakeup(9) and wakeup(9) are called with the mutex DROPPED, after
 *     the state change that warrants the wake.
 */
```

That comment is enough for Chapter 11 to either follow the same convention or to deliberately change it. A driver that explains its own rules makes future maintenance easier; a driver that does not explain its rules leaves every future reader to infer them from the source, which is slow and error-prone.

10.8.4 Splitting `cbuf` Out of `myfirst.c`

In Stage 2 and Stage 3, the `cbuf` source lived alongside `myfirst.c` in the same module directory but in its own `.c` file. The Makefile is updated to compile both:

```
KMOD=    myfirst
SRCS=    myfirst.c cbuf.c
SRCS+=   device_if.h bus_if.h

.include <bsd.kmod.mk>
```

Two minor pieces are worth noting.

The first is that `cbuf.c` declares its own `MALLOC_DEFINE`. Each `MALLOC_DEFINE` for the same tag in the same module would be a duplicate definition; we therefore put the declaration in exactly one source file (`cbuf.c`) and an extern declaration in `cbuf.h` if needed. In our setup, the tag is local to `cbuf.c` and no external use is needed.

The second is that `cbuf.c` does not need any of the `myfirst` headers. It is a self-contained library that the driver happens to use. If you ever wanted to share `cbuf` with a second driver, you could pull it out into its own KLD or into `/usr/src/sys/sys/cbuf.h` and `/usr/src/sys/kern/subr_cbuf.c` (a hypothetical placement). The discipline of keeping `cbuf` self-contained makes that possible.

10.8.5 Naming Conventions

A small but useful pattern: name your buffer-related fields and functions consistently. We have used `sc->cb` for the buffer, `cbuf_*` for buffer functions, `myfirst_buf_*` for the driver's wrappers. The pattern lets a reader scan the code and instantly know whether a function is touching the raw buffer (`cbuf_*`) or going through the locked driver wrappers (`myfirst_buf_*`).

Avoid mixing styles. Calling the buffer `sc->ring` in some places and `sc->cb` in others, or `cbuf_get` and `cbuf_read`, makes the code harder to skim. Pick one set of names and use them throughout.

10.8.6 Defending Against Buffer-Size Surprises

The `MYFIRST_BUFSIZE` macro determines the capacity of the ring. Right now it is hard-coded to 4096. There is nothing wrong with that, but a `sysctl` knob (read-only) that exposes the value, plus a `module_param`-style override at module load time, would make the driver more usable in tests without needing to recompile.

Here is the pattern for a load-time override using `TUNABLE_INT`:

```
static int myfirst_bufsize = MYFIRST_BUFSIZE;
TUNABLE_INT("hw.myfirst.bufsize", &myfirst_bufsize);
SYSCTL_INT(_hw_myfirst, OID_AUTO, bufsize, CTLFLAG_RDN,
    &myfirst_bufsize, 0, "Default buffer size for new myfirst attaches");
```

`TUNABLE_INT` reads the value from the kernel environment at boot or `kldload` time. A user can set it from the loader prompt (set `hw.myfirst.bufsize=8192`) or by running `kenv hw.myfirst.bufsize=8192` before `kldload`. The `CTLFLAG_RDN` flag indicates “read-only at runtime, but tunable at load time.” After load, `sysctl hw.myfirst.bufsize` shows the chosen value.

Then in `myfirst_attach`, use `myfirst_bufsize` instead of `MYFIRST_BUFSIZE` in the `cbuf_init` call. The change is small but useful: now you can experiment with different buffer sizes without rebuilding the module.

10.8.7 Goals for the Next Milestone

Where Chapter 11 takes the driver:

- The single mutex you have today protects everything. Chapter 11 will discuss whether a single lock is the right design under heavy contention, whether sleepable locks (`sx_*`) would be more appropriate, and how to reason about lock ordering when multiple subsystems get involved.
- The blocking path uses `mtx_sleep`, which is the right primitive for this kind of work. Chapter 11 will introduce `cv_wait(9)` (condition variables) as a more structured alternative for some patterns, and discuss when each is preferable.
- The wake-up strategy uses `wakeup(9)` (wake everyone). Chapter 11 will discuss `wakeup_one(9)` and the thundering-herd problem, and when each is appropriate.
- The `cbuf` is intentionally not thread-safe by itself. Chapter 11 will revisit this decision and discuss the tradeoffs of building locking *into* the data structure versus leaving it to the caller.
- The detach path's "wait for descriptors to close" rule is conservative. Chapter 11 will discuss alternative strategies (forced revocation, reference counting at the `cdev` level, the `destroy_dev_drain(9)` mechanism) for drivers that need to detach despite open descriptors.

You do not need to know any of this material yet. The point is that the current code's *shape* is what makes those topics approachable in Chapter 11. You can swap the mutex for an `sx` lock without touching the helpers' signatures. You can swap `wakeup` for `wakeup_one` with one-line changes. You can introduce a per-reader sleep channel without restructuring the I/O handlers. The refactor pays off as soon as you start asking the next chapter's questions.

10.8.8 A Reading Order for the Next Chapter

When you start Chapter 11, three files in `/usr/src/sys` will repay careful reading.

`/usr/src/sys/kern/subr_sleepqueue.c` is where `mtx_sleep`, `tsleep`, and `wakeup` are implemented. Read it once for context. The implementation is more elaborate than the man pages suggest, but the core of it (chan-keyed sleep queues, atomic dequeue on wake) is straightforward.

`/usr/src/sys/sys/sx.h` and `/usr/src/sys/kern/kern_sx.c` together explain the sleepable-shared-exclusive lock. We mentioned `sx` above as an alternative to `mtx`; reading the actual implementation is the best way to understand the tradeoffs.

`/usr/src/sys/sys/condvar.h` and `/usr/src/sys/kern/kern_condvar.c` document the `cv_wait` family of condition-variable primitives. Like `mtx_sleep`, they build on the kernel's sleep-queue machinery in `subr_sleepqueue.c`, but they expose a distinct structured API where each wait point has its own named `struct cv` instead of an arbitrary address as the channel. Chapter 11 will explain when to prefer each, and why a dedicated `struct cv` is often the cleaner choice for a well-defined wait condition.

These are not required reading; they are the next step on a long path you are clearly already on.

10.8.9 Wrapping Up Section 7

The driver is now in the shape Chapter 11 wants. The buffer abstraction is in its own file, exercised in userland, and called from the driver through a small set of locked wrappers. The locking strategy is documented in a

comment that names exactly what the mutex protects and what the rules are. The blocking path is correct, the non-blocking path is correct, the poll path is correct, and the detach path correctly waits for and wakes any sleepers.

Most of what you do in Chapter 11 will be additive to this base, not a rewrite of it. The patterns we have built (lock around state changes, sleep with the mutex as interlock, wake on every transition) are the same patterns the rest of the kernel uses. The vocabulary is the same, the primitives are the same, the discipline is the same. You are close to being able to read most character-device drivers in the tree without help.

Before we move on to the chapter's supplementary topics and the labs, take a moment to look at your own source. The Stage 4 driver should be roughly 500 lines of code (`myfirst.c`) plus about 110 lines of `cbuf.c` and 20 lines of `cbuf.h`. The total is small, the layering is clean, and almost every line is doing something specific. That density is what well-shaped driver code looks like.

10.9 Section 8: Three Supplementary Topics

This section covers three topics that often appear alongside buffered I/O in the real world. Each one is large enough to fill an entire chapter on its own; we are not going to do that. Instead we are going to introduce each at the level a reader of this book needs in order to recognise the pattern, talk about it sensibly, and know where to look when the time comes to use it. The deeper treatments come later, in the chapters where each topic is the main subject.

The three topics are: `d_mmap(9)` for letting user space map a kernel buffer; zero-copy considerations and what they really mean; and the readahead and write-coalescing patterns used by high-throughput drivers.

10.9.1 Topic 1: `d_mmap(9)` and Mapping a Kernel Buffer

`d_mmap(9)` is the character-device callback that the kernel invokes when a user-space program calls `mmap(2)` on `/dev/myfirst`. The handler's job is to translate a *file offset* into a *physical address* the VM system can map into the user's process. The signature is:

```
typedef int d_mmap_t(struct cdev *dev, vm_ooffset_t offset, vm_paddr_t *paddr,
                    int nprot, vm_memattr_t *memattr);
```

For each page-sized chunk the user wants to map, the kernel calls `d_mmap` with `offset` set to the byte offset within the device. The handler computes the physical address of the corresponding page and stores it through `*paddr`. It can also adjust the memory attributes through `*memattr` (caching, write-combining, and so on). Returning a non-zero error code tells the kernel “this offset cannot be mapped”; returning 0 indicates success.

The reason we are introducing `d_mmap` here is that it is the lightweight cousin of buffered I/O. With `read(2)` and `write(2)`, every byte is copied across the trust boundary on each call. With `mmap(2)` followed by direct memory access, the bytes are visible to user space without any explicit copy. A user-space program reads from or writes to the mapped region exactly as if it were ordinary memory, and the kernel's buffer is the same bytes the user sees.

This pattern is appealing for a small but important class of devices. A frame buffer, a DMA-mapped device buffer, a shared-memory event queue: each of these benefits from being mapped directly so that user

code can manipulate the bytes without ever entering the kernel. The classical example in `/usr/src/sys/dev/mem/memdev.c` (with the architecture-specific `memmmap` function under each `arch` directory) maps `/dev/mem` so that privileged user processes can read or write physical memory pages.

For a learning driver like ours, the goal is more modest: let `mmap(2)` see the same circular buffer that `read(2)` and `write(2)` use. The user can then read out the buffer without going through the syscall path. We will not extend the driver to support writes through `mmap` (that would require careful handling of cache coherency and concurrent updates with the syscall path), but a read-only mapping is a useful capability to add.

10.9.1.1 A Minimal `d_mmap` Implementation

The implementation is short:

```
static int
myfirst_mmap(struct cdev *dev, vm_ooffset_t offset, vm_paddr_t *paddr,
             int nprot, vm_memattr_t *memattr)
{
    struct myfirst_softc *sc = dev->si_drv1;

    if (sc == NULL || !sc->is_attached)
        return (ENXIO);
    if ((nprot & VM_PROT_WRITE) != 0)
        return (EACCES);
    if (offset >= sc->cb.cb_size)
        return (-1);
    *paddr = vtophys((char *)sc->cb.cb_data + (offset & ~PAGE_MASK));
    return (0);
}
```

Add `.d_mmap = myfirst_mmap`, to the `cdevsw`. The handler does four things in sequence.

First, it checks that the device is still attached. A user that has held an `mmap` on a torn-down driver should see `ENXIO`, not a kernel panic.

Second, it refuses write mappings. Allowing `PROT_WRITE` would let user space modify the buffer concurrently with the read and write handlers, which would race with the `cbuf`'s invariants. A read-only mapping is enough for our learning purposes; a real driver that wants writable mappings has to do considerably more work to keep the `cbuf` consistent.

Third, it bounds the offset. The user could request `offset = 1 << 30`, far past the end of the buffer; the handler returns `-1` to refuse. (Returning `-1` tells the kernel “no valid address for this offset”; the kernel treats this as the end of the mappable region.)

Fourth, it computes the physical address with `vtophys(9)`. `vtophys` translates a kernel virtual address into the corresponding physical address for a single page. The buffer was allocated with `malloc(9)`, which returns *virtually* contiguous memory; for an allocation that fits in one page (our `MYFIRST_BUFSIZE` of 4096 bytes on a 4 KB page machine) this is trivially also physically contiguous, and one `vtophys` is enough. For larger buffers,

each page must be looked up individually, because `malloc(9)` does not promise cross-page physical contiguity. The expression `(offset & ~PAGE_MASK)` rounds the caller's offset down to the page boundary so that `vtophys` is called on the correct page base; the kernel then takes care of applying the intra-page offset from the user's `mmap` call. A production driver whose buffer may span more than one page should walk the allocation page by page, or switch to `contigmalloc(9)` when physical contiguity is actually required.

10.9.1.2 Caveats and Limitations

A few important caveats apply to this minimal implementation.

`vtophys` works for memory allocated by `malloc(9)` only when each page of the allocation is contiguous in physical memory. Small allocations (under one page) are always contiguous. Larger allocations made with `malloc(9)` are *virtually* contiguous but not necessarily physically contiguous; the handler would need to compute the per-page physical address rather than assuming linearity. For Chapter 10's 4 KB buffer (which fits in a single page) the simple form works.

For genuinely large buffers, the right primitive is `contigmalloc(9)` (contiguous physical memory) or `dev_pager_*` functions to provide a custom pager. Both belong in later chapters where we discuss VM details properly.

The mapping is read-only. A `PROT_WRITE` request will fail with `EACCES`. Allowing writes would require either a way to invalidate user mappings when the `cbuf` indices change (impractical for a circular buffer), or a fundamentally different design where the user's writes drive the buffer directly. Neither is appropriate for a learning chapter.

Finally, mapping the `cbuf` does *not* let user space see a coherent stream of bytes the way a read does. The mapping shows the *raw* underlying memory, including bytes outside the live region (which may be stale or zero) and ignoring the head/used indices. A user that reads from the mapping needs to consult `sysctl dev.myfirst.0.stats.cb_used` and `cb_used` to know where the live region starts and ends. This is intentional: `mmap` is a low-level mechanism that exposes raw memory, and any structured interpretation has to be layered on top.

10.9.1.3 A Small `mmap` Tester

A user-space program that maps the buffer and walks it looks like:

```
/* mmap_myfirst.c: map the myfirst buffer read-only and dump it. */
#include <sys/mman.h>
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

#define DEVPATH "/dev/myfirst"
#define BUFSIZE 4096

int
```

```
main(void)
{
    int fd = open(DEVPATH, O_RDONLY);
    if (fd < 0) { perror("open"); return (1); }

    char *map = mmap(NULL, BUFSIZE, PROT_READ, MAP_SHARED, fd, 0);
    if (map == MAP_FAILED) { perror("mmap"); close(fd); return (1); }

    printf("first 64 bytes:\n");
    for (int i = 0; i < 64; i++)
        printf(" %02x", (unsigned char)map[i]);
    putchar('\n');

    munmap(map, BUFSIZE);
    close(fd);
    return (0);
}
```

Run it after writing some bytes into the device:

```
$ printf 'ABCDEFGHIJKL' > /dev/myfirst
$ ./mmap_myfirst
first 64 bytes:
 41 42 43 44 45 46 47 48 49 4a 4b 4c 00 00 00 00 ...
```

The first twelve bytes are A, B, ..., L, exactly what was written. The remaining bytes are zero because `cbuf_init` zero-fills the backing memory and we have not written anything past offset 12. This is the basic mechanism.

10.9.1.4 When You Would Actually Use `d_mmap`

Most pseudo-devices do not need `d_mmap`. The syscall path is fast, simple, and well-understood, and the cost of an extra `read(2)` per page is negligible for low-rate data. Use `d_mmap` when one of the following applies:

- The data is being produced into the buffer at very high rates (gigabytes per second in graphics or high-end I/O) and the syscall overhead per byte starts to dominate.
- The user space wants to peek at or process specific positions in a large buffer without copying the whole thing.
- The driver represents hardware whose registers or DMA areas are addressable as memory (for instance, a GPU's command FIFO).

For our pseudo-device, `d_mmap` is mostly a learning exercise. Building it teaches you the call signature, the relationship to the VM system, and the `vtophys/contigmalloc` distinction. Real production use comes when you write a driver that demands the throughput.

10.9.2 Topic 2: Zero-Copy Considerations

“Zero-copy” is one of the most overused words in systems performance discussions. Strictly read, it means “no data is copied between memory locations during the operation.” That definition is too strict to be useful: even DMA from a device to memory is, technically, a copy. In practice, “zero-copy” is shorthand for “the bytes do not pass through the CPU’s caches as part of an explicit copy instruction in the I/O path.”

For a character device like `myfirst`, the question is whether you can avoid the `uiomove(9)` copy in the read and write handlers. The answer, for the patterns we have built, is “no, and trying to is usually a mistake.” Here is why.

`uiomove(9)` does one copy from kernel to user (or user to kernel) for each transfer. That is one set of byte motions per `read(2)` or `write(2)` call. The CPU pulls the source into cache, writes the destination from cache, and goes on with its work. On modern hardware, this copy is fast: an L1 cache line is 64 bytes, the CPU can stream tens of gigabytes per second of memory copies, and the cost per byte is in the single-digit nanoseconds.

To eliminate that copy, you have to find another way to make the bytes visible to user space. The two main mechanisms are `mmap(2)` (which we just discussed) and shared-memory primitives (`shm_open(3)`, sockets with `MSG_PEEK`, `sendfile`). All of them have their own costs: page-table updates, TLB flushes, IPI traffic on multi-CPU systems, the inability to use the source memory for anything else while it is mapped. For small to medium transfers, `uiomove` is *faster* than the alternatives because the alternatives’ setup costs dominate.

There are real cases where zero-copy pays off. A network driver that DMA’s incoming packets into mbufs and hands the mbufs to the protocol stack avoids a copy that would otherwise cost as much as the DMA itself. A storage driver that uses `bus_dmamap_load(9)` to set up a DMA transfer from a user-space buffer (after `vs` locking it) avoids two copies that would otherwise dominate the I/O cost. A high-throughput graphics driver may map GPU command buffers directly into the rendering process to avoid a per-frame copy. All of these are real wins.

For a pseudo-device whose data does not come from real hardware, however, the gain is illusory. The “saved” copy is just rearranging where the bytes are stored; the cost shows up somewhere else (the page-table update, the cache miss when the user touches a page that has been written-through directly from the kernel, the contention when two CPUs touch the same shared page). The Stage 4 driver does a `uiomove` per bounce-buffer-sized chunk; that is roughly one copy per 256 bytes, which is well within the throughput a single core can sustain.

If you find yourself optimizing the copy out of a pseudo-device, two questions are worth asking first.

The first question is whether the copy is actually the bottleneck. Run the driver under `dtrace` or `pmcstat` and measure where the cycles are going. If `uiomove` is not in the top three, optimizing it will make no measurable difference. The most common bottlenecks in this kind of code are lock contention (one CPU is waiting for another to release the mutex), syscall overhead (many small syscalls instead of fewer large ones), and the cost of waking sleepers (every wakeup is a sleep-queue traversal). All of those offer bigger gains than the copy itself.

The second question is whether the *user* of the driver actually wants zero-copy semantics. A user that calls `read(2)` is asking the kernel to give them a copy of the bytes. They are not asking for a pointer to the kernel’s bytes. Switching to a mapping changes the contract; the user has to know about the mapping, manage it explicitly, and understand the cache-coherency rules. That is a trade-off the user has to opt into, not a transparent improvement.

The right framing is: zero-copy is a technique with specific costs and specific benefits. Use it when the benefits clearly outweigh the costs, and not before. For most drivers, especially pseudo-devices, the syscall path with `uiomove` is the right choice.

10.9.3 Topic 3: Readahead and Write Coalescing

The third topic is about throughput. When a driver supports a steady high-rate stream of bytes, two patterns become important: **readahead** on the read side and **write coalescing** on the write side. Both are about doing more work per syscall, and both reduce the per-byte overhead of the I/O path.

10.9.3.1 Readahead

Readahead is the act of fetching more data than the user has currently asked for, on the assumption that they will ask for it next. A regular-file read often triggers readahead at the VFS level: when the kernel notices that a process has read a few sequential blocks, it starts reading the next blocks in the background so that the next `read(2)` finds them already in memory. The user sees lower latency on subsequent reads.

For a pseudo-device, readahead at the VFS level is not directly applicable (there is no underlying file). However, the *driver* can do its own form of readahead by asking the data source to produce ahead of time. Imagine a driver that wraps a slow data source (a hardware sensor, a remote service). When the user reads, the driver pulls data from the source. The user reads again; the driver pulls more. With readahead, the driver might pull a *block* of data from the source the first time the user reads, store the extra bytes in the `cbuf`, and serve subsequent reads directly from the `cbuf` without going back to the source.

This is exactly what the `myfirst` driver already does in spirit. The `cbuf` *is* the readahead buffer. Writes deposit data, reads consume it, and the reader does not have to wait for the writer to write each individual byte. The wider lesson is that having a buffer in the driver is, structurally, the same pattern as readahead: it lets the consumer find data already prepared.

When you do build a driver against a real source, the readahead logic typically lives in a kernel thread or callout that watches `cbuf_used` and triggers a fetch from the source when the count drops below a threshold. The threshold is the *low-water mark*; the fetch stops when the count reaches the *high-water mark*. The `cbuf` becomes a buffer between the source's burst rate and the consumer's burst rate, and the kernel thread keeps it appropriately full.

10.9.3.2 Write Coalescing

Write coalescing is the mirror pattern. A driver that sinks data into a slow destination (a hardware register, a remote service) might collect several small writes into a single large write, reducing the per-write overhead at the destination. The user's `write(2)` calls deposit bytes into the `cbuf`; a kernel thread or callout reads from the `cbuf` and writes to the destination in larger chunks.

Coalescing is especially useful when the destination has high per-operation overhead. Consider a driver that talks to a chip whose command structure expects a header, payload, and footer per write: a single 1024-byte write to the chip might be twenty times faster than a thousand 1-byte writes, because the per-write overhead dominates at small sizes. The driver coalesces by collecting bytes in the `cbuf` and flushing them in larger chunks.

The decision of *when* to flush is the hard part. Two common policies exist: **flush at threshold** (flush when `cbuf_used` exceeds a high-water mark) and **flush at timeout** (flush after a fixed delay since the first byte arrived). Most real drivers use a combination: flush whenever either condition is met. A `callout(9)` (the kernel's deferred-execution primitive) is the natural way to schedule the timeout. Chapter 13 covers `callout` in detail; for now, the conceptual point is that coalescing is a deliberate trade between per-byte latency (worse, because the byte sits in the buffer) and per-operation throughput (better, because the destination sees fewer larger writes).

10.9.3.3 How These Patterns Apply to `myfirst`

The `myfirst` driver does not need either pattern explicitly because it has no real source or sink. The `cbuf` already provides the coupling between the writer and the reader, and the only “flush” is the natural one that happens when the reader calls `read(2)`. But knowing the patterns is useful for two reasons.

First, when you read driver code in `/usr/src/sys/dev/`, you will see these patterns repeatedly. Network drivers coalesce TX writes through queues. Audio drivers do readahead by fetching DMA blocks ahead of the consumer. Block-device drivers use the BIO layer to coalesce I/O requests by sector adjacency. Recognising the pattern lets you skim a thousand lines of driver code without losing the plot.

Second, when you start writing real-hardware drivers in Part 4 and beyond, you will need to decide whether and how to apply these patterns to your driver. The Chapter 10 work has given you the *substrate* (a circular buffer with proper locking and blocking semantics). Adding readahead means starting a kernel thread to fill it. Adding coalescing means flushing it on a timer or threshold. The substrate is the same; the policies are different.

10.9.4 Wrapping Up Section 8

These three topics (`d_mmap`, zero-copy, readahead/coalescing) are common follow-on conversations in driver development. None of them is a Chapter 10 topic in its own right, but each one builds on the buffer abstraction and the I/O machinery you have just put in place.

`d_mmap` adds a complementary path to the buffer: in addition to `read(2)` and `write(2)`, user space can now look at the bytes directly. Zero-copy is the framing that explains why `d_mmap` matters in some cases and is overkill in others. Readahead and write coalescing are the patterns that turn a buffered driver into a high-throughput driver.

The next sections of the chapter return to your current driver: hands-on labs that consolidate the four stages, challenge exercises that stretch your understanding, and a troubleshooting section for the bugs this material is most likely to produce.

10.10 Hands-On Labs

The labs below walk you through the four stages of the chapter, with concrete checkpoints between them. Each lab corresponds to a milestone you can verify with the test kit from Section 6. They are designed to be done in order; later labs assume earlier ones are complete.

A general note: at the start of every lab session, do a `kldunload myfirst` (if the previous module is still loaded) and a fresh `kldload ./myfirst.ko`. Watch `dmesg | tail` for the attach message. If the attach fails, the rest of the lab will fail in confusing ways; fix the attach first.

10.10.1 Lab 1: The Standalone Circular Buffer

Goal: Build and verify the userland `cbuf` implementation. This lab is entirely in user space; no kernel module is involved.

Steps:

1. Create the directory `examples/part-02/ch10-handling-io-efficiently/cbuf-userland/` if it does not already exist.
2. Type `cbuf.h` and `cbuf.c` exactly as shown in Section 2. Resist the temptation to skim the source from the book; typing it forces you to notice every line.
3. Type `cb_test.c` from Section 2.
4. Build with `cc -Wall -Wextra -o cb_test cbuf.c cb_test.c`.
5. Run `./cb_test`. You should see three “OK” lines and the final “all tests OK”.

Checkpoint questions:

- What does `cbuf_write(&cb, src, n)` return when the buffer is already full?
- What does `cbuf_read(&cb, dst, n)` return when the buffer is already empty?
- After `cbuf_init(&cb, 4)` and `cbuf_write(&cb, "ABCDE", 5)`, what is `cbuf_used(&cb)`? What is the content of `cb.cb_data` (positions 0..3)?

If you cannot answer these from your own code, re-read Section 2 and trace through the source.

Stretch goal: add a fourth test, `test_alteration`, that writes one byte, reads it back, writes another byte, reads it back, and so on for 100 iterations. This catches off-by-one errors in `cbuf_read` that the existing tests do not.

10.10.2 Lab 2: The Stage 2 Driver (Circular Buffer Integration)

Goal: Move the verified `cbuf` into the kernel and replace the Stage 3 linear FIFO from Chapter 9.

Steps:

1. Create `examples/part-02/ch10-handling-io-efficiently/stage2-circular/`.
2. Copy `cbuf.h` from your userland directory into the new directory.
3. Type the kernel-side `cbuf.c` from Section 3 (this is the `MALLOC_DEFINE`-using version).
4. Copy `myfirst.c` from `examples/part-02/ch09-reading-and-writing/stage3-echo/` into the new directory.
5. Modify `myfirst.c` to use the `cbuf` abstraction. The changes are:
 - Add `#include "cbuf.h"` near the top.
 - Replace `char *buf; size_t buflen, bufhead, bufused;` with `struct cbuf cb;` in the `softc`.
 - Update `myfirst_attach` to call `cbuf_init(&sc->cb, MYFIRST_BUFSIZE)`. Update the failure path to call `cbuf_destroy`.

- Update `myfirst_detach` to call `cbuf_destroy(&sc->cb)`.
 - Replace `myfirst_read` and `myfirst_write` with the loop-and-bounce versions from Section 3.
 - Update the `sysctl` handlers as in Section 3 (use the `myfirst_sysctl_cb_used` and `myfirst_sysctl_cb_free` helpers).
6. Update the Makefile to build both source files: `SRCS= myfirst.c cbuf.c device_if.h bus_if.h`.
 7. Build with `make`. Fix any compilation errors.
 8. Load with `kldload ./myfirst.ko` and verify with `dmesg | tail`.

Verification:

```
$ printf 'helloworld' > /dev/myfirst
$ sysctl dev.myfirst.0.stats.cb_used
dev.myfirst.0.stats.cb_used: 10
$ cat /dev/myfirst
helloworld
$ sysctl dev.myfirst.0.stats.cb_used
dev.myfirst.0.stats.cb_used: 0
```

Stretch goal 1: write enough bytes to wrap the buffer (write 3000 bytes, read 2000, write 2000 again). Verify that `cb_head` is non-zero in `sysctl` and that the data still comes back correctly.

Stretch goal 2: add a `sysctl`-controlled debug flag (`myfirst_debug`) and a `MYFIRST_DBG` macro (Section 3 shows the pattern). Use it to log every successful `cbuf_read` and `cbuf_write` in the I/O handlers. Set the flag with `sysctl dev.myfirst.debug=1` and watch `dmesg`.

10.10.3 Lab 3: The Stage 3 Driver (Blocking and Non-Blocking)

Goal: Add blocking-on-empty, blocking-on-full, and `EAGAIN` for non-blocking callers.

Steps:

1. Create `examples/part-02/ch10-handling-io-efficiently/stage3-blocking/` and copy your Stage 2 source into it.
2. Modify `myfirst_read` to add the inner sleep loop (Section 5). The new shape includes the `nbefore = uio->uio_resid` snapshot, the `mtx_sleep` call, and the `wakeup(&sc->cb)` after a successful read.
3. Modify `myfirst_write` to add the symmetric sleep loop and the matching `wakeup(&sc->cb)`.
4. Update `myfirst_detach` to set `sc->is_attached = 0` *before* calling `wakeup(&sc->cb)`, all under the mutex.
5. Build, load, and verify.

Verification:

```
$ cat /dev/myfirst &
[1] 12345
$ ps -AxH -o pid,wchan,command | grep cat
12345 myfrd  cat /dev/myfirst
```

```
$ echo hi > /dev/myfirst
hi
[after the cat consumes "hi", it blocks again]
$ kill -INT %1
[1]   Interrupt: 2
```

Verification of EAGAIN:

```
$ ./rw_myfirst_nb      # from the userland directory
step 1: empty-read returned EAGAIN (expected)
step 2: poll(POLLIN, 0) = 0 revents=0x0
...
```

If step 1 still says read returned 0, your `IO_NDELAY` check in `myfirst_read` is missing or wrong.

Stretch goal 1: open two cat processes against `/dev/myfirst` simultaneously. Write 100 bytes from a third terminal. Both cats should wake up; one will get the bytes (whichever wins the race for the lock), the other will block again. You can verify the assignment by tagging each cat with a different output stream: `cat /dev/myfirst > /tmp/a &` and `cat /dev/myfirst > /tmp/b &`, then `cmp /tmp/a /tmp/b` (one will be empty).

Stretch goal 2: time how long `cat /dev/myfirst` takes to wake up after a write, using `time(1)`. The wake-up latency should be in the low microseconds; if it is in milliseconds, something is buffering between the write and the wake (or your machine is heavily loaded).

10.10.4 Lab 4: The Stage 4 Driver (Poll Support and Refactor)

Goal: Add `d_poll`, refactor buffer access into helpers, and document the locking strategy.

Steps:

1. Create `examples/part-02/ch10-handling-io-efficiently/stage4-poll-refactor/` and copy your Stage 3 source.
2. Add `struct selinfo rsel; struct selinfo wsel;` to the `softc`.
3. Implement `myfirst_poll` as in Section 5.
4. Add `selwakeup(&sc->wsel)` after the read's successful `cbuf_read`, and `selwakeup(&sc->rsel)` after the write's successful `cbuf_write`.
5. Add `seldrain(&sc->rsel); seldrain(&sc->wsel);` to detach.
6. Add `.d_poll = myfirst_poll,` to the `cdevsw`.
7. Refactor the I/O handlers to use the four helpers from Section 7 (`myfirst_buf_read`, `myfirst_buf_write`, `myfirst_wait_data`, `myfirst_wait_room`).
8. Add the locking-strategy comment from Section 7.
9. Build, load, and verify.

Verification:

```
$ ./rw_myfirst_nb
step 1: empty-read returned EAGAIN (expected)
step 2: poll(POLLIN, 0) = 0 revents=0x0
step 3: wrote 12 bytes
step 4: poll(POLLIN, 0) = 1 revents=0x41
step 5: read 12 bytes: hello world
```

The key change from Lab 3 is that step 4 should now return 1 (not 0), with `revents=0x41` (`POLLIN | POLLRDNORM`). If it still returns 0, your `selwakeup` call is missing from the write path or the `myfirst_poll` handler is wrong.

Stretch goal 1: run `producer_consumer` with `TOTAL_BYTES = 8 * 1024 * 1024` (8 MB) and verify that the test completes with no mismatches. The producer is generating bytes faster than the consumer is reading them, so the buffer should fill up and trigger the blocking path repeatedly. Watch `sysctl dev.myfirst.0.stats.cb_used` in another terminal; it should oscillate.

Stretch goal 2: run two `producer_consumers` in parallel against the same device. The two writers will compete for buffer space; the two readers will compete for bytes. Each pair should still see consistent checksums, but the *interleaving* of bytes will be unpredictable. This shows that the driver is single-stream per device, not per descriptor; if you need per-descriptor streams, that is a different driver design.

10.10.5 Lab 5: Memory Mapping

Goal: Add `d_mmap` so user space can map the `cbuf` read-only.

Steps:

1. Create `examples/part-02/ch10-handling-io-efficiently/stage5-mmap/` and copy your Stage 4 source.
2. Add `myfirst_mmap` from Section 8 to the source.
3. Add `.d_mmap = myfirst_mmap,` to the `cdevsw`.
4. Build, load, and verify.

Verification:

```
$ printf 'ABCDEFGHijkl' > /dev/myfirst
$ ./mmap_myfirst          # from the userland directory
first 64 bytes:
 41 42 43 44 45 46 47 48 49 4a 4b 4c 00 00 00 ...
```

The first twelve bytes are the bytes you wrote.

Stretch goal 1: write a small program that maps the buffer and reads bytes from `offset = sc->cb_size - 32` (i.e. the last 32 bytes). Verify that the program does not crash. Then write enough bytes to push the buffer head into the wrap region and read from the same offset. The contents will be different, because the *raw* bytes in memory are not the same as the *live* bytes from the `cbuf`'s perspective.

Stretch goal 2: try to map the buffer with `PROT_WRITE`. Your program should see `mmap` fail with `EACCES`, because the driver refuses writable mappings.

10.10.6 Lab 6: Stress and Long-Running Tests

Goal: Run the driver under sustained load for at least an hour without errors.

Steps:

1. Set up four parallel test processes:
 - `dd if=/dev/zero of=/dev/myfirst bs=4k 2>/dev/null &`
 - `dd if=/dev/myfirst of=/dev/null bs=4k 2>/dev/null &`
 - `./producer_consumer`
 - A loop that polls `sysctl dev.myfirst.0.stats` every 5 seconds.
2. Let the test run for at least an hour.
3. Check `dmesg` for any kernel warnings, panics, or WITNESS complaints. Check `vmstat -m | grep cbuf` to confirm no leak. Verify that `producer_consumer` reports zero mismatches.

Verification: No kernel warnings. No memory growth in `vmstat`. `producer_consumer` returns 0.

Stretch goal: run the same test under a WITNESS-enabled debug kernel. The kernel will be slower but will catch any locking-discipline violations. If your driver is correct, no warnings should appear.

10.10.7 Lab 7: Deliberate Failures

Goal: Break the driver in three specific ways and observe what happens. This lab teaches you to recognise the failure modes you most want to avoid.

Steps for failure 1: hold the lock across uiomove.

1. Edit your Stage 4 driver. In `myfirst_read`, comment out the `mtx_unlock(&sc->mtx)` that comes before `uiomove(bounce, got, uio)`.
2. Add a matching `mtx_unlock` after the `uiomove` so the code still compiles.
3. Build and load on a WITNESS-enabled kernel.
4. Run a single `cat /dev/myfirst` and write some bytes from another terminal.

What you should observe: A WITNESS warning in `dmesg` complaining about “sleeping with mutex held”. The system may continue running but the warning is the bug.

Cleanup: restore the original code.

Steps for failure 2: forget the wakeup after a write.

1. In `myfirst_write`, comment out `wakeup(&sc->cb)`.
2. Build and load.
3. Run `cat /dev/myfirst &` and `echo hi > /dev/myfirst`.

What you should observe: The cat does not wake up. It will sit in `myfrd` state forever (or until you interrupt it with Ctrl-C).

Cleanup: restore the wakeup. Verify that cat now wakes up immediately.

Steps for failure 3: missing PCATCH.

1. In `myfirst_wait_data`, change `PCATCH` to 0 in the `mtx_sleep` call.

2. Build and load.
3. Run `cat /dev/myfirst & and try kill -INT %1`.

What you should observe: The cat does not respond to Ctrl-C until you write some bytes to wake it. With PCATCH, the signal would interrupt the sleep immediately.

Cleanup: restore PCATCH. Verify that `kill -INT` works as expected.

These three failures are the most common driver bugs in this chapter's territory. Doing them deliberately, once, is the best way to recognise them when they happen accidentally.

10.10.8 Lab 8: Reading Real FreeBSD Drivers

Goal: Read three character-device drivers in `/usr/src/sys/dev/` and identify how each implements its buffer, sleep, and poll patterns.

Steps:

1. Read `/usr/src/sys/dev/evdev/cdev.c`. Identify:
 - Where the per-client ring buffer is allocated.
 - Where the read handler blocks (look for `mtx_sleep`).
 - How `EVDEV_CLIENT_EMPTYQ` is implemented.
 - How `kqueue` is set up alongside `select/poll` (we have not done `kqueue` yet; just notice the calls to `knlist_*`).
2. Read `/usr/src/sys/dev/random/randomdev.c`. Identify:
 - Where `randomdev_poll` is defined.
 - How it handles a not-yet-seeded random device.
3. Read `/usr/src/sys/dev/null/null.c`. Identify:
 - How `zero_read` loops over `uio_resid`.
 - Why there is no buffer, no sleep, and no poll handler.

Checkpoint questions:

- Why does `evdev`'s read handler use `mtx_sleep` while `null`'s does not?
- What would `randomdev`'s poll handler return if called while the device is unseeded?
- How does `evdev` detect that a client has been disconnected (revoked)?

The point of this lab is not to memorise these drivers. It is to confirm that the patterns you have built in `myfirst` are the same patterns the kernel uses elsewhere. By the end of the lab you should feel that the rest of `dev/` is largely *legible* now, where it might have looked impenetrable two chapters ago.

10.11 Challenge Exercises

The labs above ensure you have a working driver and a working test kit. The challenges below are stretch exercises. Each one extends the chapter's material in a useful direction, and each one rewards careful work. Take your time; some of them are more involved than they look.

10.11.1 Challenge 1: Add a Tunable Buffer Size

The `MYFIRST_BUFSIZE` macro hard-codes the buffer at 4 KB. Make it configurable.

- Add a `TUNABLE_INT("hw.myfirst.bufsize", &myfirst_bufsize)` and a matching `SYCTL_INT(_hw_myfirst, OID_AUTO, bufsize, ...)` so the user can set the buffer size at module load time.
- Use the value in `myfirst_attach` to size the `cbuf`.
- Validate the value (reject zero, reject sizes larger than 1 MB, fall back to a sensible default if the input is bad).
- Verify with `kenv hw.myfirst.bufsize=8192; kldload ./myfirst.ko; sysctl dev.myfirst.0.stats.cb_size`

Stretch: make the buffer size *runtime-tunable* via `sysctl`. This is harder than the load-time tunable because it requires safely reallocating the `cbuf` while the device may be in use; you will need to drain or copy existing bytes, take and release the lock at the right moments, and decide what to do with sleeping callers. (Hint: it may be easier to require that all descriptors be closed before allowing a runtime resize.)

10.11.2 Challenge 2: Implement Overwrite Semantics as an Optional Mode

Add an `ioctl(2)` (or, simpler for now, a `sysctl`) that switches the buffer between “block on full” mode (the default) and “overwrite oldest on full” mode. In overwrite mode, `myfirst_write` always succeeds: when `cbuf_free` is zero, the driver advances `cb_head` to make room and then writes the new bytes.

- Add a `cbuf_overwrite` function alongside `cbuf_write` that implements the overwrite semantics. Do not modify `cbuf_write`; the two should be siblings.
- Add a `sysctl dev.myfirst.0.overwrite_mode` (read-write integer, 0 or 1).
- In `myfirst_write`, dispatch to `cbuf_overwrite` if the flag is set.
- Test with a small writer that produces bytes faster than the reader consumes; in overwrite mode, the reader should see the most recent bytes only, while in normal mode the writer blocks.

Stretch: add a counter for the number of bytes overwritten (lost). Expose it as a `sysctl` so the user can see how much data has been dropped.

10.11.3 Challenge 3: Per-Reader Position

The current driver has one shared read position (`cb_head`). When two readers consume bytes, each `read(2)` call drains some bytes from the buffer; the two readers split the stream between them. Some drivers want the opposite: each reader should see *every* byte, so two readers each get the full stream independently.

This is a substantial refactor:

- Maintain a per-descriptor read position in `myfirst_fh`.
- Track the global “earliest live byte” across all descriptors. The `cbuf`’s effective head becomes `min(per_fh_head)`.
- `myfirst_read` advances only the per-descriptor position; `cbuf_read` is replaced by a per-fh equivalent.
- A new descriptor opened mid-stream sees only bytes written after its open.
- When the buffer is “full” depends on the slowest descriptor; you need backpressure logic that accounts for laggards.

This challenge is harder than it sounds; it is essentially building a multicast pipe. Try it only if you have time to think through the locking carefully.

10.11.4 Challenge 4: Implement `d_kqfilter`

Add `kqueue(2)` support alongside the `d_poll` you already have.

- Implement a `myfirst_kqfilter` function dispatched from `cdevsw->d_kqfilter`.
- For `EVFILT_READ`, register a filter that becomes ready when `cbuf_used > 0`.
- For `EVFILT_WRITE`, register a filter that becomes ready when `cbuf_free > 0`.
- Use `knlist_add(9)` and `knlist_remove(9)` to manage the per-filter list.
- Trigger `KNOTE_LOCKED(...)` from the I/O handlers when the buffer transitions.
- Test with a small `kqueue(2)` user program that opens the device, registers `EVFILT_READ`, calls `kevent(2)`, and reports when the descriptor becomes readable.

This challenge is the natural extension of Stage 4. It also previews the `kqueue` material that Chapter 11 will discuss in more depth alongside concurrency.

10.11.5 Challenge 5: Per-CPU Counters

The `bytes_read` and `bytes_written` counters are updated under the mutex. Under heavy multi-CPU load, this can become a contention point. FreeBSD's `counter(9)` API provides per-CPU counters that can be incremented without a lock and summed for read access.

- Replace `sc->bytes_read` and `sc->bytes_written` with `counter_u64_t` instances.
- Allocate them with `counter_u64_alloc(M_WAITOK)` in `attach`; free them with `counter_u64_free` in `detach`.
- Use `counter_u64_add(counter, n)` to increment.
- Use `counter_u64_fetch(counter)` (with a `sysctl` handler) to read.

Stretch: measure the difference. Run `producer_consumer` against the old and new versions and compare wall-clock time. With a small test the difference will be invisible; with a heavily threaded test (multiple producers and consumers) the per-CPU version should be measurably faster.

10.11.6 Challenge 6: A Hardware-Style Interrupt Simulator

Real driver buffers are usually filled by an interrupt handler, not by a `write(2)` syscall. Simulate this:

- Use `callout(9)` (Chapter 13 covers it; you can read ahead) to run a callback every 100 ms.
- The callback writes a small piece of data into the `cbuf` (for example, the current time as a string).
- The user reads from `/dev/myfirst` and sees a stream of timestamped lines.

This challenge previews Chapter 13's deferred-execution material and shows how the same buffer abstraction supports either a syscall-driven producer or a kernel-thread-driven producer.

10.11.7 Challenge 7: A Logging Buffer with dmesg-Style Behaviour

Build a second character device, `/dev/myfirst_log`, that uses an overwrite-mode cbuf to keep a circular log of recent driver events. Every `MYFIRST_DBG` macro call would write into this log instead of (or in addition to) calling `device_printf`.

- Use a separate struct `cbuf` in the softc.
- Provide a way for the kernel side to push lines into the log (`myfirst_log_printf(sc, fmt, ...)`).
- The user can `cat /dev/myfirst_log` to see the recent N lines.
- A new line that overflows the buffer evicts the oldest line, not just the oldest byte (this requires line-aware eviction logic).

This challenge introduces a fairly common driver pattern (a private debug log) and gives you practice with a second, independently designed buffer use case in the same module.

10.11.8 Challenge 8: Performance Measurement

Build a measurement harness that times the throughput of the driver across the four stages.

- Write a small C program that opens the device, writes 100 MB of data, and times the operation.
- Mirror it with a reader that drains 100 MB and times itself.
- Run the pair against Stage 2, Stage 3, and Stage 4 of the chapter, and produce a small table of throughput numbers.
- Identify which stage is slowest and explain why.

The expected answer is “Stage 3 is slower than Stage 2 because of the extra wakeup and `selwakeup` calls per iteration; Stage 4 is similar to Stage 3 within measurement noise”. But the actual numbers are interesting and may surprise you, depending on your CPU, memory bandwidth, and system load.

Stretch: profile the driver under load with `pmcstat(8)` and identify the top three functions by CPU time. If `uiomove` is in the top three, you have validated the discussion in Section 8 about zero-copy. If `mtx_lock` is in the top three, you have a contention problem that Chapter 11’s locking material will address.

10.11.9 Challenge 9: Cross-Reading Real Drivers

Pick three drivers in `/usr/src/sys/dev/` that you have not read before. For each one, identify:

- Where the buffer is allocated and freed.
- Whether it is a circular buffer, a queue, or a different shape.
- What protects it (mutex, sx, lock-free, none).
- How read and write handlers consume from or produce into it.
- How `select/poll/kqueue` integrate with the buffer state changes.

Suggested starting points: `/usr/src/sys/dev/iicbus/iiconf.c` (different category but uses some of the same primitives) and `/usr/src/sys/fs/cuse/cuse.c` (a driver that exposes its buffer to user space). You will see variations on the same themes you have just built.

10.11.10 Challenge 10: Document Your Driver

Write a one-page README in your `examples/part-02/ch10-handling-io-efficiently/stage4-poll-refactor/` directory. The README should cover:

- What the driver does.
- How to build it (make).
- How to load and unload (`kldload`, `kldunload`).
- The user-space interface: device path, mode, reader/writer expectations, blocking behaviour.
- What the sysctls expose.
- How to enable debug logging.
- A reference to the chapter that produced it.

Documentation is the part of driver work most often skipped. A driver that only its author understands is a maintenance liability. Even a one-page README that explains the basics makes the difference between code that survives a hand-off and code that does not.

10.12 Troubleshooting and Common Mistakes

Most of the bugs that appear in this chapter's territory cluster into a small number of categories. The list below catalogues the categories, the symptoms each one produces, and the fix. Read it once before you work through the labs; come back to it when something goes wrong.

10.12.1 Symptom: `cat /dev/myfirst` blocks forever, even after `echo` writes data

Cause. The write handler is not calling `wakeup(&sc->cb)` after a successful `cbuf_write`. The reader is asleep on the channel `&sc->cb`; without a matching `wakeup`, it will never return.

Fix. Add `wakeup(&sc->cb)` after every state-changing operation that might unblock a waiter. In `myfirst_write`, that means after the `cbuf_write` call. In `myfirst_read`, that means after the `cbuf_read` call (which may unblock a waiting writer).

How to verify. Run `ps -AxH -o pid,wchan,command | grep cat`. If the `wchan` column shows `myfrd` (or whatever `wmesg` you used), the reader is sleeping. The channel address you slept on must match the channel address you wake.

10.12.2 Symptom: Data corruption under heavy load

Cause. Almost always a wrap-around bug or a missing lock around the `cbuf` access. Either the `cbuf`'s internal arithmetic is wrong, or two threads are touching it concurrently without synchronization.

Fix. Re-read the `cbuf` source carefully. Run the userland `cb_test` against your current `cbuf.c` (compile it directly with `cc`). If the userland tests pass, the problem is in the driver's locking, not in the `cbuf`. Check that every `cbuf_*` call is bracketed by `mtx_lock` and `mtx_unlock`. Use `INVARIANTS` and `WITNESS` in your kernel config to catch violations.

How to verify. Run `producer_consumer` with a known checksum. If the checksums match but mismatches are reported, the data is being reordered (a wrap-around bug). If the checksums differ, bytes are being lost or duplicated (a locking bug).

10.12.3 Symptom: Kernel panic with “sleeping with mutex held”

Cause. You called `uiomove(9)`, `copyin(9)`, `copyout(9)`, or another sleeping function while holding `sc->mtx`. The sleeping function tried to fault on user memory, and the page-fault handler tried to sleep, but holding a non-sleepable mutex during a sleep is forbidden.

Fix. Release the mutex before any call that might sleep. The Stage 4 handlers do this carefully: lock to access the `cbuf`, unlock to call `uiomove`, lock again to update state.

How to verify. A WITNESS-enabled kernel will print a warning before panicking. The warning identifies the mutex and the sleeping function. The first time this happens, copy the message into a debug log so you can find the call site.

10.12.4 Symptom: EAGAIN is returned even when data is available

Cause. The handler is checking the wrong flag, or it is checking the flag in the wrong place in the loop. Two common variants: checking `ioflag & O_RDONLY` instead of `ioflag & IO_NDELAY`, or returning `EAGAIN` after some bytes have already been transferred (which is the rule from Section 4 you must not break).

Fix. Re-read Section 5’s handler code carefully. The `EAGAIN` path is inside the inner `while (cbuf_used(&sc->cb) == 0)` loop, after the `nbefore` check, only when `ioflag & IO_NDELAY` is non-zero.

How to verify. Run `rw_myfirst_nb`. Step 5 should successfully read the bytes. If it shows `EAGAIN`, the bug is at one of the two locations above.

10.12.5 Symptom: A write succeeds but a subsequent read gets fewer bytes

Cause. The byte counter is being updated incorrectly, or the `cbuf` is being modified outside the handlers. A specific failure mode: counting want bytes as written when `cbuf_write` only stored put bytes (a race in Stage 2 between the `cbuf_free` check and the `cbuf_write` call, even though it is not exercised in single-writer use).

Fix. Look at the `bytes_written += put` line in `myfirst_write`; it must use the actual return value of `cbuf_write`, not the requested size. Compare `sc->bytes_written` and `sc->bytes_read` over time; they should differ by at most `cbuf_size`.

How to verify. Add a log line: `device_printf(dev, "wrote %zu of %zu\n", put, want);`. If `put != want` ever appears in `dmesg`, you have found the discrepancy.

10.12.6 Symptom: `kldunload` returns `EBUSY`

Cause. Some descriptor is still open against the device. The detach refuses to proceed when `active_fhs > 0`.

Fix. Find the process that holds the descriptor open and close it. `fstat | grep myfirst` lists the offending processes. kill them if necessary.

How to verify. After closing all descriptors (or killing the offending processes), `sysctl dev.myfirst.0.stats.active_fh` should drop to zero. `kldunload myfirst` should then succeed.

10.12.7 Symptom: Memory growth in `vmstat -m | grep cbuf`

Cause. The driver is allocating without freeing. Either the attach failure path forgot to call `cbuf_destroy`, or the detach path forgot, or there is more than one `cbuf` being allocated per attach.

Fix. Audit every code path that calls `cbuf_init`. Each call must be matched by exactly one `cbuf_destroy` call before the surrounding context goes away. The standard idiom is to put `cbuf_init` near the top of `attach` and `cbuf_destroy` near the bottom of `detach`, with the failure-path `goto fail_*` chain calling `cbuf_destroy` if `attach` fails after `cbuf_init`.

How to verify. `kldload` and `kldunload` the module several times. `vmstat -m | grep cbuf` should show 0 after each `kldunload`.

10.12.8 Symptom: `select(2)` or `poll(2)` does not wake up

Cause. The driver is missing a `selwakeup` call when state changes. Either the read path forgot to call `selwakeup(&sc->wsel)` after draining bytes, or the write path forgot to call `selwakeup(&sc->rsl)` after adding bytes.

Fix. The pattern: every state change that might make a previously-not-ready condition into a ready condition must be paired with a `selwakeup` call. Drain bytes -> `selwakeup(&sc->wsel)`. Add bytes -> `selwakeup(&sc->rsl)`.

How to verify. Run `rw_myfirst_nb`. Step 4 should show `revents=0x41`. If it shows `revents=0x0`, your `selwakeup` is missing or the `myfirst_poll` handler is not setting `revents` correctly.

10.12.9 Symptom: `truss` shows `EINVAL` for zero-byte reads

Cause. Your handler is rejecting zero-byte reads with `EINVAL`. As discussed in Section 4, zero-byte reads and writes are legal and the handler must not error on them.

Fix. Remove any `if (uio->uio_resid == 0) return (EINVAL);` early return at the top of `myfirst_read` or `myfirst_write`.

How to verify. A program that calls `read(fd, NULL, 0)` should see the call return 0, not -1 with `EINVAL`.

10.12.10 Symptom: `ps` shows the reader stuck in a sleep state called something different

Cause. Your `mtx_sleep` is being called with a different `wmesg` than you expected. Two common variants: `typo (mfyrd instead of myfrd)`, or the same handler is being called from a code path where the wait reason is actually different.

Fix. Standardise the `wmesg` strings. `myfrd` for “myfirst read”, `myfwr` for “myfirst write”. A unique short string per wait point makes `ps -AxH` immediately informative.

How to verify. `ps -AxH` should show `myfrd` for sleeping readers and `myfwr` for sleeping writers.

10.12.11 Symptom: A signal does not interrupt a blocked read

Cause. The `mtx_sleep` is being called without `PCATCH`. Without `PCATCH`, signals are deferred until the sleep ends for some other reason.

Fix. Always pass `PCATCH` for user-driven sleeps. The exception is sleeps that should be uninterruptible (kernel internal logic that must not be cancelled by a signal). For `myfirst_read` and `myfirst_write`, both are user-driven and both should pass `PCATCH`.

How to verify. `cat /dev/myfirst &` followed by `kill -INT %1` should cause `cat` to exit. If `cat` does not exit until you also write to the device (or send `kill -9`), `PCATCH` is missing.

10.12.12 Symptom: Compiler warns about a missing prototype for `cbuf_read`

Cause. The driver source uses `cbuf_read` but does not include `cbuf.h`.

Fix. Add `#include "cbuf.h"` near the top of `myfirst.c`. The file path is relative to the source directory, so as long as both files are in the same directory the include will resolve.

How to verify. A clean build with no warnings.

10.12.13 Symptom: make complains about missing `bus_if.h` or `device_if.h`

Cause. The Makefile is missing the standard `SRCS+= device_if.h bus_if.h` line that pulls in the auto-generated `kobj` headers for Newbus.

Fix. Use the Makefile from Section 3.

How to verify. `make clean && make` should succeed without missing-header errors.

10.12.14 Symptom: `kldload` fails with `Exec format error`

Cause. The `.ko` was built against a different kernel than the one currently running. This typically happens when you reboot into a different kernel without rebuilding, or when you copy a `.ko` from one machine to another with different kernel sources.

Fix. `make clean && make` against the running kernel's `/usr/src`.

How to verify. `uname -a` should match the kernel version that built the `.ko`. Check `dmesg` after the failed `kldload` for more details.

10.12.15 Symptom: The driver reports correct data but loses it after several runs

Cause. The `cbuf` is not being reset between attaches, or the `softc` is not being zero-initialised. With `M_ZERO` on the `malloc(9)` call (and the `cbuf_init` call zero-initializing its own state), this should not happen, but a partial fix that misses one of these can leave stale state.

Fix. Audit `myfirst_attach` to make sure every field of the `softc` is initialised explicitly. Use `M_ZERO` on the `malloc(9)` call that allocates the `softc` (Newbus does this automatically with `device_get_softc`, but verify). Use `cbuf_init` to set the `cbuf`'s indices to zero.

How to verify. `kldload`, write some data, `kldunload`, `kldload` again. The new attach should report `cb_used = 0`.

10.12.16 Symptom: `producer_consumer` reports a small number of mismatches under heavy load

Cause. A subtle locking bug, often related to the order of wakeup calls and the inner sleep loop's re-check. The classic symptom: under contention, occasionally a thread wakes up and consumes bytes that another thread had thought were still available.

Fix. Verify that every `mtx_sleep` is in a `while` (not `if`) loop, and that the loop re-checks the condition after waking. The wakeup is a *hint*, not a guarantee; a woken thread might find the condition false again because another thread got there first.

How to verify. `producer_consumer` should report zero mismatches across multiple runs. A mismatch count that varies between runs suggests a race; a mismatch count that is always exactly N suggests an off-by-one bug.

10.12.17 General Advice for Debugging

Three habits make driver debugging much faster.

The first is to have a WITNESS-enabled kernel ready. WITNESS catches lock-ordering violations and “sleeping with mutex held” bugs that a production kernel would silently allow. The performance overhead is significant, so run WITNESS in your lab environment, not in production.

The second is to add `device_printf` log lines liberally during development, then remove them or guard them behind `myfirst_debug` before committing. The log buffer is finite, so do not log per-byte; one line per I/O call is the right granularity.

The third is to compile with `-Wall -Wextra` and treat warnings as bugs. The kernel build system passes a lot of warning flags by default; pay attention to them. Almost every warning is the kernel telling you about a real or potential bug.

When all else fails, sit back and trace through the code path on paper. A driver this size is small enough to fit on a single sheet. Ninety percent of the time, drawing the call graph and the lock acquisitions in order shows you the bug.

10.13 Quick Reference: Patterns and Primitives

This reference is the chapter's material reduced to fast-lookup form. Use it after you have read the chapter; it is a reminder, not a tutorial.

10.13.1 The Circular Buffer API

```

struct cbuf {
    char      *cb_data;      /* backing storage */
    size_t    cb_size;      /* total capacity */
    size_t    cb_head;      /* next byte to read */
    size_t    cb_used;      /* live byte count */
};

int      cbuf_init(struct cbuf *cb, size_t size);
void     cbuf_destroy(struct cbuf *cb);
void     cbuf_reset(struct cbuf *cb);
size_t   cbuf_size(const struct cbuf *cb);
size_t   cbuf_used(const struct cbuf *cb);
size_t   cbuf_free(const struct cbuf *cb);
size_t   cbuf_write(struct cbuf *cb, const void *src, size_t n);
size_t   cbuf_read(struct cbuf *cb, void *dst, size_t n);

```

Rules: - The cbuf does not lock; the caller is responsible. - cbuf_write and cbuf_read clamp n to available space or live data and return the actual count. - cbuf_used and cbuf_free return current state under the assumption that the caller holds whatever lock protects the cbuf.

10.13.2 The Driver-Level Helpers

```

size_t   myfirst_buf_read(struct myfirst_softc *sc, void *dst, size_t n);
size_t   myfirst_buf_write(struct myfirst_softc *sc, const void *src, size_t n);
int      myfirst_wait_data(struct myfirst_softc *sc, int ioflag, ssize_t nbefore,
                          struct uio *uio);
int      myfirst_wait_room(struct myfirst_softc *sc, int ioflag, ssize_t nbefore,
                          struct uio *uio);

```

Rules: - All four helpers assert that sc->mtx is held with mtx_assert(MA_OWNED). - The wait helpers return -1 to mean “break the outer loop, return 0 to user space”. - The wait helpers return EAGAIN, EINTR, ERESTART, or ENXIO for the corresponding conditions.

10.13.3 The Read Handler Spine

```

nbefore = uio->uio_resid;
while (uio->uio_resid > 0) {
    mtx_lock(&sc->mtx);
    error = myfirst_wait_data(sc, ioflag, nbefore, uio);
    if (error != 0) {

```

```

        mtx_unlock(&sc->mtx);
        return (error == -1 ? 0 : error);
    }
    take = MIN((size_t)uio->uio_resid, sizeof(bounce));
    got = myfirst_buf_read(sc, bounce, take);
    fh->reads += got;
    mtx_unlock(&sc->mtx);

    wakeup(&sc->cb);
    selwakeup(&sc->wsel);

    error = uiomove(bounce, got, uio);
    if (error != 0)
        return (error);
}
return (0);

```

10.13.4 The Write Handler Spine

```

nbefore = uio->uio_resid;
while (uio->uio_resid > 0) {
    mtx_lock(&sc->mtx);
    error = myfirst_wait_room(sc, ioflag, nbefore, uio);
    if (error != 0) {
        mtx_unlock(&sc->mtx);
        return (error == -1 ? 0 : error);
    }
    room = cbuf_free(&sc->cb);
    mtx_unlock(&sc->mtx);

    want = MIN((size_t)uio->uio_resid, sizeof(bounce));
    want = MIN(want, room);
    error = uiomove(bounce, want, uio);
    if (error != 0)
        return (error);

    mtx_lock(&sc->mtx);
    put = myfirst_buf_write(sc, bounce, want);
    fh->writes += put;
    mtx_unlock(&sc->mtx);

    wakeup(&sc->cb);
    selwakeup(&sc->rsel);
}

```

```

}
return (0);

```

10.13.5 The Sleep Pattern

```

mtx_lock(&sc->mtx);
while (CONDITION) {
    if (uio->uio_resid != nbefore)
        break_with_zero;
    if (ioflag & IO_NDELAY)
        return (EAGAIN);
    error = mtx_sleep(CHANNEL, &sc->mtx, PCATCH, "wmesg", 0);
    if (error != 0)
        return (error);
    if (!sc->is_attached)
        return (ENXIO);
}
/* condition is false now; act on the buffer */

```

Rules: - Use while, not if, around the condition. - Always pass PCATCH for user-driven sleeps. - Always re-check the condition after `mtx_sleep` returns. - Always check `is_attached` after waking, in case detach is pending.

10.13.6 The Wake Pattern

```

/* After a state change that might unblock a sleeper: */
wakeup(CHANNEL);
selwakeup(SELINFO);

```

Rules: - The channel must match the channel passed to `mtx_sleep`. - The `selinfo` must be the one that `sel-`record registered against. - Use `wakeup` (wake all) for shared waiters; `wakeup_one` for single-handoff patterns. - Spurious wakeups are safe; missing wakeups are bugs.

10.13.7 The `d_poll` Handler

```

static int
myfirst_poll(struct cdev *dev, int events, struct thread *td)
{
    struct myfirst_softc *sc = dev->si_drv1;

```



```

    int revents = 0;

    mtx_lock(&sc->mtx);
    if (events & (POLLIN | POLLRDNORM)) {
        if (cbuf_used(&sc->cb) > 0)
            revents |= events & (POLLIN | POLLRDNORM);
        else
            selrecord(td, &sc->rsel);
    }
    if (events & (POLLOUT | POLLWRNORM)) {
        if (cbuf_free(&sc->cb) > 0)
            revents |= events & (POLLOUT | POLLWRNORM);
        else
            selrecord(td, &sc->wsel);
    }
    mtx_unlock(&sc->mtx);
    return (revents);
}

```

10.13.8 The d_mmap Handler

```

static int
myfirst_mmap(struct cdev *dev, vm_ooffset_t offset, vm_paddr_t *paddr,
             int nprot, vm_memattr_t *memattr)
{
    struct myfirst_softc *sc = dev->si_drv1;

    if (sc == NULL || !sc->is_attached)
        return (ENXIO);
    if ((nprot & VM_PROT_WRITE) != 0)
        return (EACCES);
    if (offset >= sc->cb.cb_size)
        return (-1);
    *paddr = vtophys((char *)sc->cb.cb_data + (offset & ~PAGE_MASK));
    return (0);
}

```

10.13.9 The cdevsw

```

static struct cdevsw myfirst_cdevsw = {
    .d_version =    D_VERSION,

```

```

        .d_open =      myfirst_open,
        .d_close =     myfirst_close,
        .d_read =      myfirst_read,
        .d_write =     myfirst_write,
        .d_poll =      myfirst_poll,
        .d_mmap =      myfirst_mmap,
        .d_name =      "myfirst",
};

```

10.13.10 Errno Values for I/O

Errno	Meaning
0	Success
EAGAIN	Would block; retry later
EFAULT	Bad user pointer (from uiomove)
EINTR	Interrupted by a signal
ENXIO	Device not present or torn down
EIO	Hardware error
ENOSPC	Permanent out-of-space (block-on-full preferred)
EACCES	Forbidden access mode
EBUSY	Device is open or otherwise locked

10.13.11 `ioflag` Bits

Bit	Source flag	Meaning
IO_NDELAY	O_NONBLOCK	Caller is non-blocking
IO_DIRECT	O_DIRECT	Bypass caching where possible
IO_SYNC	O_FSYNC	(write only) Synchronous semantics

`O_NONBLOCK == IO_NDELAY` per the kernel's CTASSERT.

10.13.12 `poll(2)` Events

Event	Meaning
POLLIN	Readable: bytes are available
POLLRDNORM	Same as POLLIN for character devices
POLLOUT	Writable: space is available
POLLWRNORM	Same as POLLOUT for character devices
POLLERR	Error condition
POLLHUP	Hangup (peer closed)
POLLNVAL	Invalid file descriptor

A driver typically handles POLLIN | POLLRDNORM for read readiness and POLLOUT | POLLWRNORM for write readiness. The other events are usually set by the kernel, not the driver.

10.13.13 Memory Allocator Reference

Call	When to use
malloc(n, M_DEVBUF, M_WAITOK \ M_ZERO)	Normal allocation, can sleep
malloc(n, M_DEVBUF, M_NOWAIT \ M_ZERO)	Cannot sleep (interrupt context)
free(p, M_DEVBUF)	Free memory allocated above
MALLOC_DEFINE(M_TAG, "name", "desc")	Declare a private memory tag
contigmalloc(n, M_TAG, M_WAITOK, ...)	Physically contiguous allocation

10.13.14 Sleep / Wake Reference

Call	When to use
mtx_sleep(chan, mtx, PCATCH, "msg", 0)	Sleep with mutex interlock
tsleep(chan, PCATCH \ pri, "msg", timo)	Sleep without mutex (rare in drivers)
cv_wait(&cv, &mtx)	Sleep on a condition variable
wakeup(chan)	Wake all sleepers on channel
wakeup_one(chan)	Wake one sleeper (for single-handoff)

10.13.15 Lock Reference

Call	When to use
<code>mtx_init(&mtx, "name", "type", MTX_DEF)</code>	Initialise a sleepable spin/sleep mutex
<code>mtx_destroy(&mtx)</code>	Destroy at detach
<code>mtx_lock(&mtx), mtx_unlock(&mtx)</code>	Acquire / release
<code>mtx_assert(&mtx, MA_OWNED)</code>	Assert lock is held (debug)

10.13.16 Test Tools Reference

Tool	Use
<code>cat, echo</code>	Quick smoke tests
<code>dd</code>	Volume tests, partial-transfer observation
<code>hexdump -C</code>	Verify byte content
<code>truss -t read,write</code>	Trace syscall returns
<code>ktrace</code>	Detailed trace including signals
<code>sysctl dev.myfirst.0.stats</code>	Live driver state
<code>vmstat -m</code>	Memory tag accounting
<code>ps -AxH</code>	Find sleeping threads and their wmesg
<code>dmesg tail</code>	Driver-emitted log lines and kernel warnings

10.13.17 Driver Lifecycle Summary

kldload

- > `myfirst_identify` (optional in this driver: creates the child)
- > `myfirst_probe` (returns `BUS_PROBE_DEFAULT`)
- > `myfirst_attach` (allocates `softc`, `cbuf`, `cdev`, `sysctl`, `mutex`)

steady state

- > `myfirst_open` (allocates per-fh state)
- > `myfirst_read` (drains `cbuf` via `bounce` + `uiomove`)
- > `myfirst_write` (fills `cbuf` via `uiomove` + `bounce`)
- > `myfirst_poll` (reports `POLLIN/POLLOUT` readiness)
- > `myfirst_close` (per-fh dtor releases per-fh state)

kldunload

- > `myfirst_detach` (refuses if any descriptor open)
- > `wakeup` releases sleepers

```
-> destroy_dev
-> cbuf_destroy
-> sysctl_ctx_free
-> mtx_destroy
```

10.13.18 File Layout Summary

```
examples/part-02/ch10-handling-io-efficiently/
```

```
    README.md
```

```
    cbuf-userland/
```

```
        cbuf.h
```

```
        cbuf.c
```

```
        cb_test.c
```

```
        Makefile
```

```
    stage2-circular/
```

```
        cbuf.h
```

```
        cbuf.c
```

```
        myfirst.c
```

```
        Makefile
```

```
    stage3-blocking/
```

```
        cbuf.h
```

```
        cbuf.c
```

```
        myfirst.c
```

```
        Makefile
```

```
    stage4-poll-refactor/
```

```
        cbuf.h
```

```
        cbuf.c
```

```
        myfirst.c
```

```
        Makefile
```

```
    stage5-mmap/
```

```
        cbuf.h
```

```
        cbuf.c
```

```
        myfirst.c
```

```
        Makefile
```

```
    userland/
```

```
        rw_myfirst_v2.c
```

```
        rw_myfirst_nb.c
```

```
        producer_consumer.c
```

```
        stress_rw.c
```

```
        mmap_myfirst.c
```

```
        Makefile
```

Each stage directory is independent; you can make and `kldload` any of them without touching the others. The

userland tools are shared across all stages.

10.13.19 A Mental Model in One Paragraph

The driver owns a circular buffer, protected by a single mutex. A reader holds the mutex while transferring bytes out of the buffer into a stack-resident bounce, releases the mutex, copies the bounce into user space with `uiomove`, wakes any waiting writers, and loops until the user's request is satisfied or the buffer is empty. A writer mirrors this: holds the mutex, copies user bytes into a bounce, releases the mutex, copies the bounce into the buffer, wakes any waiting readers, and loops. When either the buffer is empty (for reads) or full (for writes), the handler either sleeps with the mutex as interlock (default mode) or returns `EAGAIN` (non-blocking mode). `select(2)` and `poll(2)` integration is provided through `selrecord` (in `d_poll`) and `selwakeup` (in the I/O handlers). The detach path waits for all descriptors to close and then frees everything.

That paragraph fits in your head. Everything else in this chapter is the careful elaboration of how to make each piece of it work.

10.14 Appendix: A Source-Reading Walkthrough of `evdev/cdev.c`

The chapter has pointed at `/usr/src/sys/dev/evdev/cdev.c` several times as the cleanest example in the tree of a character device that does what `myfirst` now does: a per-client ring buffer, blocking reads, non-blocking support, `select/poll/kqueue` integration. Reading that file once, with the patterns from this chapter in hand, is the fastest way to confirm that the kernel really does work the way the chapter has been describing. This appendix walks through the relevant pieces.

The goal is *not* to teach `evdev`. It is to use `evdev` as an exhibit. By the end of this walkthrough you should feel that what you built in `myfirst` is the same shape as what the kernel uses for real input devices. The differences are in the details (the protocol, the structures, the layered driver stack), not in the underlying patterns.

10.14.1 What `evdev` Is

`evdev` is FreeBSD's port of the Linux event-device interface. It exposes input devices (keyboards, mice, touchscreens) through `/dev/input/eventN` nodes that user-space programs (X servers, Wayland compositors, console handlers) read from to get a stream of input events. Each event is a fixed-size structure with a timestamp, a type, a code, and a value.

The driver layer that interests us is the per-client `cdev`. When a process opens `/dev/input/event0`, the kernel creates a `struct evdev_client` for that descriptor, attaches it to the underlying device, and uses it as the per-open buffer. Reads pull events out of the buffer; writes push events into it (for some devices); `select/poll/kqueue` report when events are available.

That description should sound very familiar by now. It is the same architecture as `myfirst` Stage 4, with three differences: the buffer is per-descriptor rather than per-device; the unit of transfer is a fixed-size structure rather than a byte; and the driver participates in a larger framework of input handling.

10.14.2 The Per-Client State

Open `/usr/src/sys/dev/evdev/evdev_private.h` (the file is short; you can read the relevant parts in a couple of minutes). The key structure is `struct evdev_client`:

```
struct evdev_client {
    struct evdev_dev *    ec_evdev;
    struct mtx            ec_buffer_mtx;
    size_t                ec_buffer_size;
    size_t                ec_buffer_head;
    size_t                ec_buffer_tail;
    size_t                ec_buffer_ready;
    ...
    bool                  ec_blocked;
    bool                  ec_revoked;
    ...
    struct selinfo        ec_selp;
    struct sigio *        ec_sigio;
    ...
    struct input_event    ec_buffer[];
};
```

Compare this against your `myfirst` softc:

- `ec_evdev` is `evdev`'s analogue of `dev->si_drv1` (a back-pointer from the per-client state to the device-wide state).
- `ec_buffer_mtx` is the per-client mutex; `myfirst`'s `sc->mtx` is per-device.
- `ec_buffer_size`, `ec_buffer_head`, `ec_buffer_tail`, `ec_buffer_ready` are the circular-buffer indices. Notice that `evdev` uses an explicit tail instead of a derived one; the code is slightly different but the structure is the same.
- `ec_blocked` is a hint flag for the wakeup logic.
- `ec_revoked` flags a forcibly-disconnected client; this is the equivalent of `is_attached` in `myfirst`.
- `ec_selp` is the `selinfo` for `select/poll/kqueue` support, exactly like `sc->rsel` and `sc->wsel` in your driver (combined here because `evdev` only does read-readiness; there is no concept of “write would block”).
- `ec_buffer[]` is the flexible-array-member that holds the actual events.

The patterns are the same. The naming is different.

10.14.3 The Read Handler

Open `/usr/src/sys/dev/evdev/cdev.c` and find `evdev_read`:

```
static int
evdev_read(struct cdev *dev, struct uio *uio, int ioflag)
```

```

{
    struct evdev_dev *evdev = dev->si_drv1;
    struct evdev_client *client;
    ...
    ret = devfs_get_cdevpriv((void **)&client);
    if (ret != 0)
        return (ret);

    debugf(client, "read %zd bytes by thread %d", uio->uio_resid,
        uio->uio_td->td_tid);

    if (client->ec_revoked)
        return (ENODEV);

    ...
    if (uio->uio_resid != 0 && uio->uio_resid < evsize)
        return (EINVAL);

    remaining = uio->uio_resid / evsize;

    EVDEV_CLIENT_LOCKQ(client);

    if (EVDEV_CLIENT_EMPTYQ(client)) {
        if (ioflag & O_NONBLOCK)
            ret = EWOULDBLOCK;
        else {
            if (remaining != 0) {
                client->ec_blocked = true;
                ret = mtx_sleep(client, &client->ec_buffer_mtx,
                    PCATCH, "evread", 0);
                if (ret == 0 && client->ec_revoked)
                    ret = ENODEV;
            }
        }
    }

    while (ret == 0 && !EVDEV_CLIENT_EMPTYQ(client) && remaining > 0) {
        head = client->ec_buffer + client->ec_buffer_head;
        ...
        bcopy(head, &event.t, evsize);

        client->ec_buffer_head =
            (client->ec_buffer_head + 1) % client->ec_buffer_size;
        remaining--;
    }
}

```



```

        EVDEV_CLIENT_UNLOCKQ(client);
        ret = uiomove(&event, evsize, uio);
        EVDEV_CLIENT_LOCKQ(client);
    }

    EVDEV_CLIENT_UNLOCKQ(client);

    return (ret);
}

```

Walk through it slowly.

The handler retrieves the per-client state with `devfs_get_cdevpriv`, exactly as `myfirst_read` does. The `ec_revoked` check is `evdev`'s equivalent of `myfirst`'s `is_attached` check, except that `evdev` returns `ENODEV` rather than `ENXIO` (both are valid choices for “the device is gone”).

The handler then validates that the requested transfer size is a multiple of an event-record size (because partial events make no sense to deliver). This is a layer above what `myfirst` does, and it is specific to event-stream devices.

Then, exactly as Section 5 described, the handler enters a *check-wait-recheck* loop. If the buffer is empty (`EVDEV_CLIENT_EMPTYQ(client)`), the handler either returns `EWOULDBLOCK` (the same value as `EAGAIN`) for non-blocking callers, or sleeps with `mtx_sleep` and `PCATCH`. The sleep channel is `client` itself; the mutex interlock is `&client->ec_buffer_mtx`. When the sleep returns, the handler re-checks `ec_revoked`, returning `ENODEV` if the client has been disconnected during the sleep.

After the wait, the handler enters the transfer loop. It picks an event off the buffer (with `bcopy` into a stack-resident event variable), advances `ec_buffer_head` modulo the buffer size, releases the mutex, and calls `uiomove` to push the event to user space. Then it reacquires the mutex and continues until either the buffer empties or the user's request is satisfied.

This is the bounce-buffer pattern from Section 3, with event playing the role of bounce. The `cbuf` operation is `bcopy(head, &event.t, evsize)` (single-event copy out of the ring) followed by `uiomove(&event, evsize, uio)` (transfer to user). The mutex is held only across the `cbuf` operation, never across `uiomove`. This is exactly the rule we made operational in `myfirst`.

10.14.4 The Wakeup

Find `evdev_notify_event` (the function that gets called when a new event is delivered into a client's buffer):

```

void
evdev_notify_event(struct evdev_client *client)
{

    EVDEV_CLIENT_LOCKQ_ASSERT(client);

```

```

    if (client->ec_blocked) {
        client->ec_blocked = false;
        wakeup(client);
    }
    if (client->ec_selected) {
        client->ec_selected = false;
        selwakeup(&client->ec_selp);
    }

    KNOTE_LOCKED(&client->ec_selp.si_note, 0);
}

```

There is the wake-up. `wakeup(client)` matches the `mtx_sleep(client, ...)` from `evdev_read`. `selwakeup(&client->ec_selp)` matches the `selrecord` we will look at in a moment. `KNOTE_LOCKED` is the `kqueue` analogue of `selwakeup`; we have not built that yet (it is Chapter 11 territory) but the pattern is the same.

The `ec_blocked` flag is an optimisation: if no client is currently sleeping, the wakeup is skipped. This is a small but useful optimisation. `myfirst` does not have it because the cost is negligible in our use case, but you could add the same check trivially.

10.14.5 The Poll

Find `evdev_poll`:

```

static int
evdev_poll(struct cdev *dev, int events, struct thread *td)
{
    struct evdev_client *client;
    int revents = 0;
    int ret;

    ret = devfs_get_cdevpriv((void **)&client);
    if (ret != 0)
        return (POLLNVAL);

    if (events & (POLLIN | POLLRDNORM)) {
        EVDEV_CLIENT_LOCKQ(client);
        if (!EVDEV_CLIENT_EMPTYQ(client))
            revents = events & (POLLIN | POLLRDNORM);
        else {
            client->ec_selected = true;
            selrecord(td, &client->ec_selp);
        }
    }
}

```

```

        EVDEV_CLIENT_UNLOCKQ(client);
    }

    return (revents);
}

```

This is essentially identical to the `myfirst_poll` from Section 5, with two differences. `evdev` only handles `POLLIN`; there is no `POLLOUT` because input events are unidirectional. And `evdev` returns `POLLNVAL` if `devfs_get_cdevpriv` fails, which is the conventional “this descriptor is invalid” response (compared to `myfirst`’s simpler return-zero approach).

The pattern is the one Section 5 introduced: check the condition, return ready if it is true, register with `selrecord` if it is not. The `ec_selected` flag is again a wakeup-elision optimisation; you can ignore it for understanding.

10.14.6 The `kqfilter`

Find `evdev_kqfilter`:

```

static int
evdev_kqfilter(struct cdev *dev, struct knote *kn)
{
    struct evdev_client *client;
    int ret;

    ret = devfs_get_cdevpriv((void **)&client);
    if (ret != 0)
        return (ret);

    switch(kn->kn_filter) {
    case EVFILT_READ:
        kn->kn_fop = &evdev_cdev_filterops;
        break;
    default:
        return(EINVAL);
    }
    kn->kn_hook = (caddr_t)client;

    knlist_add(&client->ec_selp.si_note, kn, 0);

    return (0);
}

```

This is the `kqueue` registration handler. It is one of Chapter 11’s topics; we are showing it here only to point out that the `selinfo`’s `si_note` field is what `kqueue` hooks into. The `selrecord`/`selwakeup` machinery for

`select/poll`, and the `knlist_add/KNOTE_LOCKED` machinery for `kqueue`, share the same `selinfo` structure. That sharing is what lets a single set of state-change calls (in `evdev_notify_event`) wake all three readiness-notification paths at once.

When we extend `myfirst` with `kqueue` support in Chapter 11, the changes will fit into roughly the same pattern: a `myfirst_kqfilter` handler that registers against `&sc->rsl.si_note`, a `KNOTE_LOCKED(&sc->rsl.si_note, 0)` call alongside each `selwakeup(&sc->rsl)`. The substrate is here already.

10.14.7 What the Walkthrough Confirms

Three things should be clear now.

The first is that the *patterns* you have been building are not invented for this book. They are the patterns the kernel uses, in a real driver that ships with FreeBSD and is exercised by every keyboard and mouse user every day. You can read this code, recognise what each section is doing, and explain it. That is a real skill that pays off in every later chapter.

The second is that the *details* differ between drivers. `evdev` uses an explicit `tail` index. It uses fixed-size event records instead of bytes. It has per-client buffers instead of per-device. It uses `bcopy` instead of a `cbuf` abstraction. None of these differences invalidate the underlying pattern; they are choices about how to specialise the pattern for a specific use case.

The third is that *you can read more*. With a couple of hours and a coffee, you can work through `/usr/src/sys/dev/uart/uart_core.c` or `/usr/src/sys/dev/snp/snp.c`. Each one will look different at first glance, but the buffer, the locking, the sleep / wake, the poll: those will be familiar. The chapter has handed you a vocabulary; the kernel source is where you exercise it.

10.14.8 A Short Reading Plan

If you want to build the habit of reading kernel source as part of your driver development workflow, here is a short plan. Spend an hour a week, for three weeks, on the following.

Week one: re-read `/usr/src/sys/dev/null/null.c` and `/usr/src/sys/dev/evdev/cdev.c`. Compare them. The first is the simplest possible character device; the second is a competent buffered one. Note exactly which features each file has and why.

Week two: read `/usr/src/sys/dev/random/randomdev.c`. It is bigger than `evdev` but uses the same patterns, with the addition of an entropy-collection layer underneath. Note how `randomdev_read` differs from `evdev_read` and `myfirst_read`, and why.

Week three: pick a driver in `/usr/src/sys/dev/` that interests you (a USB driver, a network driver, a storage driver). Read the part of it that handles user-space I/O. By now the patterns should be familiar enough that the unfamiliar parts (bus binding, hardware register access, DMA setup) stand out as the *new* things to learn, not as obstacles to understanding the I/O path.

After three weeks of this rhythm, you will have read more driver code than most professional kernel developers do in a typical month. The investment compounds.

10.15 Chapter Summary

This chapter took the in-kernel buffer from Chapter 9, which was a linear FIFO that wasted half its capacity, and turned it into a real circular buffer with proper partial-I/O semantics, blocking and non-blocking modes, and `poll(2)` integration. The driver you finish the chapter with is meaningfully better than the one you started with, in four specific ways.

It uses its full capacity. The circular buffer keeps the entire allocation in play. A steady writer and a matching reader can keep the buffer at any fill level indefinitely; the wrap-around is invisible to the I/O handlers because it lives inside the `cbuf` abstraction.

It honours partial transfers. Both `myfirst_read` and `myfirst_write` loop until they cannot make progress, then return zero with `uio_resid` reflecting the unconsumed portion. A user-space caller that loops on `read(2)` or `write(2)` will see correct UNIX semantics. A caller that does not loop will still see correct counts; the driver does not silently throw bytes away.

It blocks correctly. A reader that finds the buffer empty sleeps on a clear channel, with the mutex released atomically; a writer that adds bytes wakes the sleeper. The same pattern works in the other direction. Signals are honoured through `PCATCH`, so the user's Ctrl-C interrupts a blocked reader within microseconds.

It supports non-blocking mode. A descriptor opened with `O_NONBLOCK` (or with the flag set later via `fcntl(2)`) sees `EAGAIN` instead of a sleep. A `d_poll` handler reports `POLLIN` and `POLLOUT` correctly based on the buffer's state, and `selrecord(9)` plus `selwakeup(9)` ensure that `select(2)` and `poll(2)` callers wake when readiness changes.

Each of those capabilities is built up in a numbered section, each with code that compiles, loads, and behaves predictably. The chapter's stages (a userland buffer, a kernel splice, a blocking-aware version, a poll-aware refactor, a memory-mapped variant) form a clear progression that matches the order in which a beginner naturally encounters these concerns.

Along the way we covered three supplementary topics that often appear next to buffered I/O in real drivers: `d_mmap(9)`, the patterns and limitations of zero-copy thinking, and the readahead and write-coalescing patterns used by high-throughput drivers. None of these is a Chapter 10 topic in its own right, but each one builds naturally on the buffer abstraction you have just put in place.

We exercised the driver with five new user-space test programs (`rw_myfirst_v2`, `rw_myfirst_nb`, `producer_consumer`, `stress_rw`, `mmap_myfirst`) plus the standard base-system tools (`dd`, `cat`, `hexdump`, `truss`, `sysctl`, `vmstat`). The combination is enough to catch most of the bugs this material typically produces. The Troubleshooting section catalogues those bugs with their symptoms and fixes; keep it bookmarked.

Finally, we refactored the buffer access into a small set of helpers (`myfirst_buf_read`, `myfirst_buf_write`, `myfirst_wait_data`, `myfirst_wait_room`), wrote a one-paragraph locking-strategy comment near the top of the source, and put the `cbuf` into its own file with its own `MALLOC_DEFINE`. The driver's source is now in the shape that Chapter 11 wants: clear locking discipline, narrow abstractions, no surprises.

10.16 Wrapping Up

The buffered driver you have just finished is the foundation for everything that follows in Part 3. The shape of its I/O path, the discipline of its locking, the way it sleeps and wakes, the way it composes with `poll(2)`: these

are not Chapter 10 patterns. They are *the* patterns the kernel uses, and once you recognise them in your own code, you will recognise them in every character-device driver in `/usr/src/sys/dev/`.

It is worth taking a moment to feel that shift. When you started Chapter 7, the inside of a driver was probably an opaque set of names and signatures. By the end of Chapter 8 you had a sense of the lifecycle. By the end of Chapter 9 you had a sense of the data path. Now, at the end of Chapter 10, you have a sense of how a driver behaves under load: how it shapes itself around concurrent callers, how it manages a finite resource (the buffer), how it cooperates with the kernel's readiness primitives, how it gets out of its own way so that user-space programs can do real work against it.

Most of what you have built will carry through to Chapter 11. The mutex, the buffer, the helpers, the locking strategy, the test kit, the lab discipline: all of it remains. What changes in Chapter 11 is the *depth* of the questions you ask about each piece. Why one mutex and not two? Why wakeup and not wakeup_one? Why `mtx_sleep` and not `cv_wait`? What guarantees does the kernel make about when a sleeper wakes, and what guarantees does it not make? How do you prove a piece of code is correct under concurrency, rather than hope?

Chapter 11 takes those questions seriously. It introduces `WITNESS` and `INVARIANTS` as the kernel's verification tools, walks through the lock classes, and discusses the patterns that turn merely-working concurrency into provably-correct concurrency. It will be a substantial chapter, but the substrate is what you have just built.

Three closing reminders before you move on.

The first is to *commit your code*. Whatever version-control system you use, save the four stage directories as a snapshot. The next chapter's first lab will copy your Stage 4 source and modify it; you do not want to lose the working baseline.

The second is to *try the labs*. Reading driver code teaches you the patterns; writing driver code teaches you the discipline. The labs in this chapter are short on purpose. Even the long ones can be done in a single sitting. The combination of "I built this" and "I broke it on purpose to see what happens" is what the chapter is designed to produce.

The third is to *trust the slow path*. The chapter has been deliberately careful, deliberately patient, deliberately repetitive in places. Driver work rewards that style. The bugs that hurt are the ones that look like they could not possibly happen. The defence against them is to be slow, careful, and methodical, even when the code seems simple. The reader who slows down with each step finishes Chapter 11 ready for Chapter 12; the reader who rushes finishes Chapter 11 with a kernel panic and a lost afternoon.

You are doing well. Keep going.

10.17 Part 2 Checkpoint

Before you cross into Part 3, pause and check that the ground beneath your feet is solid. Part 2 has carried you from "what is a module" to "a multi-stage pseudo-driver that serves real readers and writers under load." The next part will put that driver on a much heavier scale, so the foundation needs to be firm.

By now you should be comfortable doing each of the following without hunting for the answer:

- Writing, compiling, loading, and unloading a kernel module against a running kernel, and reading `dmesg` to confirm the lifecycle.

- Building a Newbus skeleton with `device_probe`, `device_attach`, and `device_detach`, backed by a per-unit softc allocated with `device_get_softc`.
- Exposing a `/dev` node through a `cdevsw` with working `d_open`, `d_close`, `d_read`, `d_write`, and `d_ioctl` handlers, and verifying that `devfs` cleans the node up on unload.
- Managing a circular buffer whose state is protected by a mutex, whose readers can block with `mtx_sleep` and be woken by `wakeup`, and whose readiness is advertised through `selrecord` and `selwakeup`.
- Walking a deliberate failure through the attach path and watching every allocation unwind in reverse order.

If any of those feels unsteady, the labs that anchor them are worth a second pass. A targeted review list:

- Build, load, and unload discipline: Lab 7.2 (Build, Load, and Verify Lifecycle) and Lab 7.4 (Simulate Attach Failure and Verify Unwinding).
- `cdevsw` hygiene and `devfs` nodes: Lab 8.1 (Structured Name and Tighter Permissions) and Lab 8.5 (Two-Node Driver).
- Data path and round-trip behaviour: Lab 9.2 (Exercise Stage 2 with Writes and Reads) and Lab 9.3 (Stage 3 FIFO Behaviour).
- The Chapter 10 core sequence: Lab 2 (Stage 2 Circular Buffer), Lab 3 (Stage 3 Blocking and Non-Blocking), and Lab 4 (Stage 4 Poll Support and Refactor).

Part 3 will assume that all of the above is muscle memory, not a lookup. Specifically, Chapter 11 will expect:

- A working Stage 4 `myfirst` that loads, unloads, and survives concurrent readers and writers without corruption.
- Familiarity with `mtx_sleep/wakeup` and the `selrecord/selwakeup` pair as the kernel's basic blocking and readiness primitives, since Part 3 will compare and contrast them with `cv(9)`, `sx(9)`, and `sema(9)`.
- A kernel built with `INVARIANTS` and `WITNESS`, since every Part 3 chapter leans on both from the first section onward.

If those three items hold, you are ready to turn the page. If one wobbles, fix that first. A quiet hour now saves a bewildering afternoon later.

10.18 Looking Ahead: Bridge to Chapter 11

Chapter 11 is titled “Concurrency in Drivers.” Its job is to take the driver you have just finished and look at it through the lens of concurrency: not the casual “it works under modest load” sense we have used so far, but the rigorous “I can prove this is correct under any interleaving” sense.

The bridge is built on three observations from Chapter 10's work.

First, you already have a single mutex protecting all shared state. That is the simplest non-trivial concurrency design a driver can have, and it is the right starting point for understanding the more elaborate alternatives. Chapter 11 will use your driver as a test case for asking when one mutex is enough, when it is not, and what to do when it is not.

Second, you already have a sleep / wake pattern that uses the mutex as an interlock. `mtx_sleep` and `wakeup` are the building blocks of every blocking primitive in the kernel. Chapter 11 will introduce condition variables (`cv_*`) as a more structured alternative, and will explain when each is appropriate.

Third, you already have a buffer abstraction that is intentionally not thread-safe by itself. The `cbuf` relies on the caller to provide locking. Chapter 11 will discuss the spectrum from “data structure provides no locking” (your `cbuf`) through “data structure provides internal locking” (some kernel primitives) to “data structure is lock-free” (`buf_ring(9)`, `epoch(9)`-based readers). Each end of the spectrum has uses; understanding when to choose which is part of becoming a driver writer.

Specific topics Chapter 11 will cover include:

- The five FreeBSD lock classes (`mtx`, `sx`, `rw`, `rm`, `lockmgr`) and when each is appropriate.
- Lock ordering and how to use `WITNESS` to verify it.
- The interaction between locks and interrupt context.
- Condition variables and when to prefer them over `mtx_sleep`.
- Reader/writer locks and their use cases.
- The `epoch(9)` framework for read-mostly data structures.
- Atomic operations (`atomic_*`) and when they obviate the need for locks.
- A walk-through of common concurrency bugs (lost wakeups, lock-order reversals, ABA, double-free under contention).

You do not need to read ahead to start Chapter 11. Everything in this chapter is sufficient preparation. Bring your Stage 4 driver, your test kit, and your `WITNESS`-enabled kernel; the next chapter starts where this one ended.

A small farewell from this chapter: you have just turned a beginner driver into a respectable one. The bytes that flow through `/dev/myfirst` are now flowing the way bytes flow through every other character device on the system. The patterns are right, the locking is right, the user-space contracts are honoured. The driver is yours to extend, to specialise, and to use as a baseline for whatever real device comes next. Take a moment to enjoy that, and then turn the page.

11 Concurrency in Drivers

11.1 Reader Guidance & Outcomes

Chapter 10 ended on a quietly optimistic note. Your `myfirst` driver now owns a real circular buffer, handles partial reads and writes correctly, sleeps when there is nothing to do, wakes when there is something to do, integrates with `select(2)` and `poll(2)`, and can be stress-tested with `dd`, `cat`, and a small userland harness. If you load it today and exercise it with the Stage 4 test kit, it behaves well. The byte counts match. The checksums agree. `dmesg` stays quiet.

It is tempting, looking at that behaviour, to conclude that the driver is correct. It is *almost* correct. What is missing is not code. What is missing is an explanation of *why* the code we have works.

Pause on that. In every chapter until now, we have justified each choice in the code by showing what it does: `uiomove(9)` moves bytes through the trust boundary, `mtx_sleep(9)` releases the mutex and sleeps, `sel-wakeup(9)` notifies `poll(2)` waiters. The chapters proved that the code is reasonable. They did not prove that the code is *safe under concurrent access*. Chapter 10 gestured at the subject; it acquired the mutex around buffer reads and writes, it released the mutex before calling `uiomove(9)`, it honoured the rule that one must not sleep while holding a non-sleepable mutex. But those moves were presented as patterns to imitate, not as conclusions to a careful argument. The argument is this chapter's work.

This chapter is where concurrency becomes the main subject.

By the end of it, you will understand why the mutex in the `softc` is there; what would go wrong if it were not; what a race condition actually is, down to the instruction level; what atomic operations buy you and what they do not; how mutexes compose with sleep, with interrupts, and with the kernel's correctness-checking tools; and how to tell, by reading a driver, whether its concurrency story is honest. You will also do what Chapter 11 of this book has always been meant to do: take the driver you have been building and bring it into proper, documented, verifiable thread safety. We will not turn your driver upside down; the shape we have is good. What we will do is earn every one of its locks.

This chapter does not try to teach the entire FreeBSD synchronization toolkit. That would be unreasonable at this point in your learning. Chapter 12 is dedicated to the rest of the toolkit: condition variables, sleepable shared-exclusive locks, timeouts, and the deeper aspects of lock ordering. Chapter 11 is the conceptual chapter that makes Chapter 12 possible. We stay close to two primitives, atomics and mutexes, and we spend the chapter learning them well.

11.1.1 Why This Chapter Earns Its Own Place

It would be possible, in principle, to skip directly to Chapter 12 and just teach sleepable locks, condition variables, and the `WITNESS` framework in one burst. Many driver tutorials do this. The result is code that works on

the author's laptop and fails on a reader's four-core desktop, because the reader does not understand what the primitives are for. They imitate the shape and miss the substance.

The cost of that kind of misunderstanding is high. Concurrency bugs are the hardest class of bug there is. They reproduce intermittently. They may appear on one machine and not another. They can sit dormant in a driver for years before a scheduling change, a faster CPU, or a new kernel option exposes them. When they do strike, the symptom is almost always somewhere other than the cause. The reader who has read only the syntax of `mtx_lock` will stare at a panic for days before realising that the bug is two functions away, in a place where nobody thought concurrency was a concern.

The only reliable defence against this is a clear mental model, built from first principles. We build that model in this chapter, one careful step at a time. By the end, `mtx_lock` and `mtx_unlock` will not be lines you copy from another driver. They will be lines whose absence you can spot.

11.1.2 Where Chapter 10 Left the Driver

A quick recap of the state you should be working from. If any of the following is missing from your driver, stop here and return to Chapter 10 before continuing.

- A Newbus child registered under `nexus0`, probed and attached at module load.
- A `struct myfirst_softc` holding a `struct cbuf` (the circular buffer), per-descriptor statistics, and a single `struct mtx` called `sc->mtx`.
- Two `struct selinfo` fields (`sc->rsel` and `sc->wsel`) for `poll(2)` integration.
- `myfirst_read`, `myfirst_write`, `myfirst_poll` handlers that hold `sc->mtx` while touching shared state and drop it before calling `uiomove(9)` or `selwakeup(9)`.
- `mtx_sleep(9)` with `PCATCH` as the blocking primitive, `wakeup(9)` as the matching call.
- A detach path that refuses to unload while descriptors are open and that wakes sleepers before freeing state.

That driver is the substrate Chapter 11 reasons about. We will not change its observable behaviour in this chapter. We will understand *why* it works, we will make its safety claims explicit, and we will add the infrastructure that lets the kernel verify them for us at runtime.

11.1.3 What You Will Learn

Walking away from this chapter, you should be able to:

- Define concurrency in the specific sense that matters inside a driver, and distinguish it from parallelism in a way that survives contact with real code.
- Enumerate the execution contexts a driver encounters on a modern FreeBSD system and reason about which of them can run simultaneously.
- Name the four most common failure modes of unsynchronized shared state and recognise each of them in a short code excerpt.
- Walk through a deliberately broken driver that corrupts its own buffer under load, and explain the corruption in terms of which instruction from which thread wins the race.
- Perform a concurrency audit on a piece of driver code: identify the shared state, mark the critical sections, classify each access, and decide what protects it.

- Explain what atomic operations are at the level the hardware provides them, which atomic primitives FreeBSD exposes through `atomic(9)`, when they are sufficient for the problem at hand, and when they are not.
- Use the `counter(9)` API for per-CPU statistics that scale under contention.
- Distinguish between the two fundamental mutex shapes in FreeBSD (`MTX_DEF` and `MTX_SPIN`), explain when each is appropriate, and describe the rules that apply to each.
- Read and interpret a `WITNESS` warning, understand what lock-order reversal means, and design a lock hierarchy that `WITNESS` will accept.
- State and apply the rule about sleeping with a mutex held, including the subtle cases involving `uiomove(9)`, `copyin(9)`, and `copyout(9)`.
- Describe priority inversion, priority propagation, and how the kernel handles the former with the latter.
- Build a small library of multi-threaded and multi-process test programs that can exercise your driver at levels of concurrency far beyond what `cat`, `dd`, and a single shell can produce.
- Use `INVARIANTS`, `WITNESS`, `mtx_assert(9)`, `KASSERT`, and `dtrace(1)` together to verify that the claims you make in comments are the claims the code actually enforces.
- Refactor the driver into a shape where the locking discipline is documented, the lock holder of every piece of shared state is obvious from the source, and a future reader (including your future self) can audit the concurrency story in minutes.

That is a substantial list. It is also the realistic scope of what a careful reader can absorb in one chapter at this stage of the book. Take it slowly.

11.1.4 What This Chapter Does Not Cover

Several topics touch on concurrency and are deliberately postponed:

- **Condition variables (`cv(9)`).** Chapter 12 introduces `cv_wait`, `cv_signal`, `cv_broadcast`, and `cv_timedwait`, and explains when a named condition variable is a cleaner design than a bare `wakeup(9)` channel. This chapter continues to use `mtx_sleep` and `wakeup`, exactly as Chapter 10 did.
- **Sleepable shared-exclusive locks (`sx(9)`) and reader/writer locks (`rw(9)`, `rm(9)`).** Chapter 12 covers these primitives and the situations they are designed for. For Chapter 11 we stay with the mutex pair we already use.
- **Lock-free data structures.** The FreeBSD kernel uses lock-free queues (`buf_ring(9)`), epoch(9)-based read-mostly readers, and hazard-pointer-like patterns in several places. These are specialised tools that reward careful study and are introduced gradually in Part 4 and the Part 6 network chapter (Chapter 28) when real drivers need them.
- **Concurrency correctness under interrupt context.** Our driver has no real hardware interrupts yet. Chapter 14 introduces interrupt handlers in detail; that chapter also extends the concurrency discussion to cover what happens when a handler can run at any instant and what that means for locking choices. Chapter 11 restricts itself to process context and kernel threads.
- **Advanced `WITNESS` configuration.** We will load `WITNESS` in a debug kernel and read its warnings. We will not construct custom lock classes or explore the full complement of `WITNESS` options.

Staying inside those lines keeps the chapter honest. A chapter on concurrency that tries to teach every synchronization primitive in FreeBSD is a chapter that teaches none of them well. We teach two primitives, we teach them well, and we set up the next chapter to teach the rest from a position of strength.

11.1.5 Estimated Time Investment

- **Reading only:** two and a half to three hours, depending on how often you pause at the diagrams. This chapter has a heavier conceptual load than Chapter 10.
- **Reading plus typing the labs and the broken-and-fixed drivers:** six to eight hours over two sessions with a reboot in between.
- **Reading plus all labs and challenges:** ten to fourteen hours over three or four sessions. The challenges are explicitly designed to stretch your understanding, not merely to consolidate it.

Do not rush. If you feel unsure midway through Section 4, that is normal; atomic operations are a genuinely new mental model for most readers. Stop, stretch, and come back. The value of this chapter compounds when the reader takes it slowly.

11.1.6 Prerequisites

Before starting this chapter, confirm:

- Your driver source is equivalent to the Chapter 10 Stage 4 example in `examples/part-02/ch10-handling-io-efficiently/stage4-poll-refactor/`. If it is not, return to Chapter 10 and bring it up to that shape. The code in this chapter assumes the Stage 4 source.
- Your lab machine is running FreeBSD 14.3 with a matching `/usr/src`. The APIs and file paths cited here align with that release.
- You have a development kernel configuration available that enables `INVARIANTS`, `WITNESS`, and `DDB`. If you do not, Section 7 walks you through building one.
- You are comfortable loading and unloading your own module, watching `dmesg`, and reading `sysctl dev.myfirst.0.stats` to check driver state.
- You read Chapter 10 carefully and understood why `mtx_sleep` takes the mutex as its interlock. If that rule still feels arbitrary, pause and re-read Chapter 10's Section 5 before starting this chapter.

If any of the above is shaky, fixing it now is a better investment than pushing through Chapter 11 with gaps.

11.1.7 How to Get the Most Out of This Chapter

Three habits pay off quickly.

First, keep `/usr/src/sys/kern/kern_mutex.c` and `/usr/src/sys/kern/subr_witness.c` bookmarked. The first is where the mutex primitives live; the second is where the kernel's lock-order checker lives. You do not need to read these files in full, but several times during the chapter we will point at a specific function and ask you to look at the real code for one minute. That minute pays dividends.

Second, read `/usr/src/sys/sys/mutex.h` once, now, before you start Section 5. The header is short, the comments are good, and seeing the full `MTX_*` flag table once makes the later discussion concrete instead of abstract.

Third, build and boot a debug kernel before Lab 1. The labs in this chapter assume `INVARIANTS` and `WITNESS` are enabled, and several of them are designed to produce specific warnings that you can only see with those options. If you have to rebuild your kernel midway through the chapter, the interruption will cost you more than doing it upfront.

11.1.8 Roadmap Through the Chapter

The sections in order are:

1. Why concurrency matters. Where concurrency comes from inside the kernel, why it is not the same thing as parallelism, and what the four fundamental failure modes look like.
2. Race conditions and data corruption. A careful dissection of what a race condition is, at the instruction level, with a worked example on the circular buffer and a deliberately unsafe version of the driver you will actually run.
3. Analyzing unsafe code in your driver. A practical audit procedure. We walk through the Chapter 10 Stage 4 source, identify every piece of shared state, classify its visibility, and document what protects it.
4. Atomic operations. What atomicity is at the hardware level, what the `atomic(9)` API offers, how memory ordering affects correctness, and when atomics are sufficient versus when they are not. We apply atomics to the driver's per-descriptor statistics.
5. Introducing locks (mutexes). The two fundamental mutex shapes, the rules that apply to each, lock hierarchy, priority inversion, deadlocks, and the sleep-with-mutex rule. We re-examine the Chapter 10 mutex choice with the full conceptual toolkit.
6. Testing multi-threaded access. How to build user-space programs that exercise your driver concurrently, using `fork(2)`, `pthread(3)`, and custom harnesses. How to log results from concurrent tests without disturbing them.
7. Debugging and verifying thread safety. Reading a `WITNESS` warning, using `INVARIANTS`, adding `mtx_assert(9)` and `KASSERT(9)` calls, and tracing with `dtrace(1)`.
8. Refactoring and versioning. Organising the code for clarity, versioning the driver, documenting the locking strategy in a `README`, running static analysis, and regression testing.

Hands-on labs and challenge exercises follow, then a troubleshooting reference, a wrapping-up section, and a bridge to Chapter 12.

If this is your first pass, read linearly and do the labs in order. If you are revisiting the material to consolidate, the debugging section and the refactoring section stand alone.

11.2 Section 1: Why Concurrency Matters

Every chapter up to this point has treated the driver as if it lived inside a single timeline. A user program calls `read(2)`; the kernel routes the call into `myfirst_read`; the handler does its work; control returns; the program continues. A second `read(2)` on another descriptor is described as “another call” as if it were somehow later in the same story.

This framing was an approximation. It let us focus on the data path without getting tangled in the question of who was calling our handler and when. The approximation served well for Chapters 7 through 10, because the guard rails we put in (a single mutex, careful lock acquisition around shared state, `mtx_sleep` as the only way to wait) were strong enough to hide the messiness of real kernel execution.

The messiness is not something we invented. It is the actual environment every FreeBSD driver lives in, all the time. This section is where we pull back the curtain.

11.2.1 What Concurrency Means in the Kernel

Concurrency is the property of a system in which multiple independent streams of execution can make progress during overlapping periods of time. The operative word is *independent*: each stream has its own stack, its own instruction pointer, its own register state, and its own reason for doing what it is doing. They may share some data (that is why we care about concurrency), but none of them is waiting for any other to finish a full task before it begins.

Inside the FreeBSD kernel, these independent streams of execution come from several distinct sources. A driver will typically encounter at least four of them, possibly all of them at once:

User-space processes calling into the driver. This is the source you have seen most often in the book so far. A user program opens `/dev/myfirst`, issues a `read(2)` or `write(2)`, and the syscall path enters the driver through `devfs`. Each such call is a separate stream of execution, with its own kernel stack, running on whichever CPU the scheduler assigned it to. Two concurrent callers are two concurrent streams, no less independent of each other than two unrelated processes running in user space.

Kernel threads spawned by the driver or the subsystem. Some drivers create their own kernel threads with `kproc_create(9)` or `kthread_add(9)` to do work that is periodic, background, or long-running. The `myfirst` driver does not yet do this, but many real drivers do; a USB bus driver has a hub-polling thread, a network driver has a TX-complete thread, a storage driver has an I/O completion thread. Each of these runs concurrently with the syscall handlers.

Callout and taskqueue execution. The kernel's `callout(9)` mechanism (Chapter 13) and its `taskqueue(9)` mechanism (Chapter 16) both run functions at times that are not directly triggered by a user syscall. A callout set for one second from now will fire one second from now, regardless of what the driver's handlers are doing at that moment. The handler runs on a different stack, often on a different CPU, and may touch the same state the syscall handler is touching.

Interrupt context. When real hardware is present, an interrupt handler (Chapter 14) runs in response to an external event: a network packet arrived, a timer fired, a USB transfer completed. Interrupts are asynchronous and preemptive: the interrupt handler can start running at any instant the hardware chooses. The code it preempts may itself be in the middle of a critical update to shared state. `myfirst` has no real hardware yet, but we will discuss interrupt context because it shapes some of the decisions we make.

A fully-deployed FreeBSD driver typically interacts with at least three of the four. A driver writer who treats any one of them as “the only thing running” has written a driver that will fail on contact with real load.

The practical consequence is that every line of your driver code is surrounded by other code that may also be running. When your `myfirst_read` is holding `sc->mtx`, it is perfectly ordinary for a different CPU to be trying to acquire the same mutex from `myfirst_write` or from a timer or from a separate caller. The kernel's job is to make that interaction safe; your driver's job is to cooperate. That cooperation is what this chapter teaches.

11.2.2 Where the Streams Come From: a Concrete Picture

It helps to look at a concrete scenario. Imagine you have a FreeBSD 14.3 machine with four CPU cores. You have loaded the Chapter 10 Stage 4 `myfirst` driver. You start the following five commands, each in its own terminal, at roughly the same time:

```
1. cat /dev/myfirst > /tmp/out1.txt
```

```
2. cat /dev/myfirst > /tmp/out2.txt
3. dd if=/dev/zero of=/dev/myfirst bs=512 count=10000
4. sysctl -w dev.myfirst.0.debug=1 (if debug is enabled)
5. while true; do sysctl dev.myfirst.0.stats; sleep 0.1; done
```

At one instant in time, here is what the kernel might be doing:

- CPU 0 is running the first cat, inside `myfirst_read`, holding `sc->mtx`, performing a `cbuf_read` into its stack bounce buffer.
- CPU 1 is running the second cat, inside `myfirst_read`, waiting in `mtx_sleep(&sc->cb, &sc->mtx, ...)` because it tried to acquire the mutex after the first cat and went to sleep.
- CPU 2 is running `dd`, inside `myfirst_write`, having just finished a `uiomove` into its bounce buffer and is now about to reacquire `sc->mtx` to deposit the bytes into the `cbuf`.
- CPU 3 is running the `sysctl` loop, inside the `cb_used` `sysctl` handler, trying to acquire `sc->mtx` to get a consistent snapshot.

Four CPUs. Four streams of execution. Four different parts of the driver, all running concurrently, all contending for the same mutex, all needing to agree on the state of the same circular buffer. This is not a contrived situation. This is what a moderately-loaded system looks like during ordinary testing.

If the mutex and the rules around it did not exist, each of those four threads would be free to observe and modify the `cbuf`'s indices (`cb_head`, `cb_used`) and its backing memory (`cb_data`) independently. The results would be, approximately, whatever the hardware's memory system happened to produce, with no guarantees about correctness. The rest of this chapter is about building an understanding of why that would be a catastrophe and what the primitives that prevent it actually do.

11.2.3 Concurrency vs Parallelism

These two words are often used interchangeably. They are related but not the same, and getting them straight early saves confusion later.

Parallelism is a property of execution: two things happen literally at the same physical time, on different pieces of hardware. On a multi-CPU system, two threads running on two different cores at the same nanosecond are parallel.

Concurrency is a property of design: the system is structured so that multiple independent streams of execution exist and can make progress in any order. Whether they physically run at the same time is a separate question.

A single-core CPU can run a concurrent program (the OS scheduler interleaves two threads on the one core). It cannot run a parallel program (nothing is literally simultaneous). A four-core machine can run both concurrent and parallel programs.

Why does the distinction matter for a driver writer? Because the bugs that matter for us are caused by concurrency, not by parallelism. A two-thread program on a single-core FreeBSD laptop can still exhibit every race condition we will meet in this chapter, as long as the scheduler ever decides to preempt one thread while it is in the middle of an update. Parallelism, on a multi-core machine, merely makes those bugs more common and harder to suppress by running the program slowly. The fundamental problem predates multi-core hardware and will exist on every machine that can ever preempt a thread.

Our focus, therefore, is on designs that tolerate any legal interleaving of independent streams. If the design tolerates that, it tolerates parallelism as well. If the design assumes “well, this will always run before that”, it will fail on some system, probably the first production server it meets.

This is also why you cannot test your way to concurrency correctness. You can test and not find a bug because the interleaving that triggers it never happened during the test. You can deploy the same code and find the bug the first day, because some tiny change in scheduling, or a slightly faster CPU, or a different number of cores, shifts the timing. The only durable way to solve concurrency is to reason about every legal interleaving, not to try to enumerate the likely ones.

11.2.4 What Can Go Wrong: The Four Failure Modes

When multiple streams of execution share data without coordination, four things can go wrong. Every concurrency bug you will ever meet is a specialised instance of one of these four. Naming them early is useful, because later in the chapter, when a specific bug shows up in a specific driver, you will recognise which family it belongs to.

Failure mode 1: Data corruption. The state after the interleaving is not the state any of the threads would have produced on its own. A shared counter becomes a value that no correct sequence of increments could have reached. A linked list has an entry that points to garbage. A buffer’s head index is larger than its capacity. This is the most common kind of concurrency bug, and the hardest to detect, because the corrupted state may lie dormant for a long time before it is used.

Failure mode 2: Lost updates. Two threads each read a shared value, each compute a new value based on what they read, and each store their result. The second write overwrites the first, which means the first thread’s update is discarded as if it never happened. Classic example: two `bytes_read += got;` statements, executed by two threads, can increment the counter by one instead of by two if the interleaving is unlucky. Lost updates are a special case of data corruption with a predictable signature: the counter is correct in form but low in magnitude.

Failure mode 3: Torn values. A value that is larger than the CPU can read or write in a single memory access (on 32-bit systems, typically a 64-bit integer) can be observed in a half-updated state by a concurrent reader. One thread is in the middle of writing bytes 0-3 of a `uint64_t`; another thread reads all 8 bytes and sees the old high half glued to the new low half. The resulting value is nonsense. On FreeBSD 14.3’s primary platforms (amd64, arm64), aligned 64-bit accesses are atomic; on 32-bit platforms and for larger structures, they are not. A torn value is the “read” side of a lost update and can sometimes be observed even when the writers are correct.

Failure mode 4: Inconsistent composite state. A data structure has an invariant that relates several fields. A thread is in the middle of updating one field but has not yet updated the other. A concurrent reader sees the structure in a state that violates the invariant, makes a decision based on what it saw, and produces a wrong result. This is the most subtle of the four modes, because it requires the observer to know what invariant is supposed to hold; a thread that does not know the invariant cannot detect that it is violated. Most of the harder driver bugs involve inconsistent composite state: a ring buffer’s head and tail were updated in two steps, and a reader saw the half-updated state and produced garbage.

All four failure modes share a cause: *multiple streams of execution touched shared state without agreement on an order*. The primitives this chapter introduces, atomics and mutexes, are different ways of imposing that agreement. Atomics make individual operations indivisible. Mutexes enforce serialized access across longer regions of code.

11.2.5 A Thought Experiment

Consider a trivial shared counter:

```
static u_int shared_counter = 0;

/* Called by anything. */
static void
bump(void)
{
    shared_counter = shared_counter + 1;
}
```

If two threads call `bump()` at the same moment, what happens?

The correct behaviour is that `shared_counter` increases by two. Each call should add one, so two calls should add two.

The actual behaviour depends on how the increment compiles. On amd64, `shared_counter = shared_counter + 1;` typically compiles into three machine instructions: a load of the current value into a register, an add of one to the register, and a store of the register back to memory.

Imagine two threads, A and B, running on two different CPUs, both entering `bump()` at essentially the same time. A legal interleaving looks like this:

Time	Thread A	Thread B	Memory
t0			0
t1	load (gets 0)		0
t2		load (gets 0)	0
t3	add 1 (local register=1)		0
t4		add 1 (local register=1)	0
t5	store (writes 1)		1
t6		store (writes 1)	1

Both threads load zero. Both compute one. Both store one. The final value is one. The counter has been incremented exactly once, even though two threads each called `bump()`.

This is a lost update. It is the simplest concurrency bug in the book. It does not require fancy hardware, a complicated data structure, or a rare scheduler decision. It requires only that two threads execute the three-instruction sequence at overlapping moments.

Notice three things about this example.

First, the bug is not in the source code. The C code says `shared_counter = shared_counter + 1;`, which is what we wanted. The bug is in the *interaction between the source code and the hardware's execution model*.

No amount of staring at the C line will reveal the bug, because the bug is at a level of abstraction the C line does not express.

Second, the bug is not reproducible by single-threaded testing. One thread calling `bump()` a thousand times will always produce `shared_counter == 1000`. Add a second thread and the result becomes nondeterministic. You could run the two-thread test a hundred times and see correct results every time; run it again on a different CPU, or with a different workload, and see a missed count.

Third, the fix is not to add more careful code in C. The fix is either to make the increment *atomic* (execute as a single indivisible unit from memory's perspective) or to enforce *mutual exclusion* (no two threads can be executing the three-instruction sequence at the same time). Atomics give us the first option. Mutexes give us the second. They are different shapes of solution to the same problem, and choosing between them is the topic of Sections 4 and 5.

11.2.6 A Preview of the Driver Under Load

To make the rest of the chapter concrete, let us describe what we are going to do to the driver.

In Section 2, we will build a deliberately unsafe version of `myfirst`: one that has had the mutex removed from the I/O handlers. We will load it, run a two-process test, and watch the circular buffer's invariants break in real time. This will be an unpleasant but instructive experience. You will see the data corruption, the torn values, and the inconsistent composite state with your own eyes, in your own terminal, in a driver you have been working on for several chapters.

In Section 3, we will go back to the Stage 4 Chapter 10 driver and audit it: every shared field, every access, every claim. We will annotate the source in place. We will end the section with a complete locking-discipline document.

In Section 4, we will introduce atomics. We will convert `sc->bytes_read` and `sc->bytes_written` from mutex-protected `uint64_t` counters into per-CPU counter(9) counters, which are lock-free and scale to many CPUs. We will discuss where atomics save locks and where they do not.

In Section 5, we will re-examine the mutex from first principles. We will explain why the mutex is specifically an `MTX_DEF` (sleep) mutex, not an `MTX_SPIN` mutex. We will walk through priority inversion and why the kernel's priority-propagation mechanism handles it for us. We will introduce `WITNESS` and run the unsafe driver under a `WITNESS` kernel to see what happens.

In Section 6, we will build a small testing harness: a `pthread`-based multi-threaded tester, a `fork`-based multi-process tester, and a load generator that stresses the driver at levels `dd` and `cat` cannot reach. We will run the safe and the unsafe drivers under this harness.

In Section 7, we will learn to read `WITNESS` warnings, use `INVARIANTS` and `KASSERT`, and deploy `dtrace(1)` probes to observe concurrency behaviour without changing it.

In Section 8, we will refactor the driver for clarity, version it as `v0.4-concurrency`, write a `README` that documents the locking strategy, run `clang - -analyze` against the source, and run a regression test that repeats every test we have built.

By the end, the driver will not be more *functional* than it is now. It will be more *verifiable*. The difference matters: verifiable code is the only code that is safe to extend.

11.2.7 The FreeBSD Scheduler in One Page

To understand why concurrency inside a driver is not just a theoretical concern, it helps to know, at a high level, what the FreeBSD scheduler does. You do not need to read any scheduler source to write a driver, but a mental model of how the scheduler interacts with your driver prevents several classes of surprise.

FreeBSD's default scheduler is **ULE**, implemented in `/usr/src/sys/kern/sched_ule.c`. ULE is a multi-core, SMP-aware, priority-based scheduler. Its job is to decide, at every scheduling decision, which thread should run on each CPU.

The key properties you care about as a driver writer:

Preemption is permitted at almost any point. A thread executing kernel code can be preempted by a higher-priority thread becoming runnable (for example, an interrupt wakes a high-priority thread). The only windows where preemption is disabled are critical sections (entered with `critical_enter`) and spin-locked regions. Inside an `MTX_DEF` mutex's critical section, preemption is still possible; the priority propagation mechanism limits the damage.

Threads migrate between CPUs. The scheduler will move threads to balance load. A driver cannot assume that two successive calls from the same process run on the same CPU.

Wake-up latency is low but not zero. A wakeup on a sleeping thread typically makes the thread runnable within microseconds. The thread actually runs when the scheduler chooses to run it, which depends on its priority and what else is runnable.

The scheduler itself holds locks. When you call `mtx_sleep`, the scheduler's internal locks are acquired at some point. This is the reason `WITNESS` sometimes reports order violations when you hold a driver lock while doing something that reaches the scheduler; the ordering between driver locks and scheduler locks matters.

For our driver, this means:

- Two concurrent callers of `myfirst_read` may be running on two different CPUs, or on the same CPU via preemption, or sequentially. We make no assumption about which.
- The mutex we use works on all of these scenarios; the scheduler guarantees that the lock acquire/release semantics are respected regardless of CPU assignment.
- When `mtx_sleep` is called, the thread is put to sleep quickly; when wakeup is called, the thread becomes runnable quickly. Latency is not a concern for correctness.

Knowing the scheduler is there, doing its work in the background, frees you to focus on the driver's own concurrency decisions. You do not have to "schedule" your own threads or worry about affinity; the kernel handles it.

11.2.8 Kinds of Concurrency by Pedagogy Level

For a beginner, the concurrency landscape can feel enormous. A short hierarchy helps.

Level 0: Single-threaded. No concurrency at all. One thread, one execution timeline. This is the model the reader has used implicitly through the first half of this book.

Level 1: Pre-emptive concurrency on a single CPU. One CPU, but the scheduler can interrupt a thread at any time and run another. Race conditions can still happen, because a thread can be in the middle of a read-modify-write when preempted. All the bugs this chapter discusses are possible at Level 1.

Level 2: True multi-CPU parallelism. Multiple CPUs, multiple threads literally running at the same physical moment. Race conditions are more common and harder to suppress with luck, but they are the same bugs as Level 1.

Level 3: Concurrency with interrupt context. Same as Level 2, but with the addition that interrupt handlers can preempt ordinary threads at any moment. Spin mutexes become necessary for state shared between top-half code and interrupt handlers. Chapter 14 and beyond cover this.

Level 4: Concurrency with RCU-style patterns. Same as Level 3, but with advanced patterns like epoch(9) that allow readers to proceed without any synchronization. Specialised and not needed for most drivers.

For Chapter 11, we are at Level 2. We have multiple CPUs, we have many threads, we have ordinary mutex-based coordination. This is where most drivers live.

11.2.9 Some Concurrency Is Invisible

A subtle point that trips beginners: the concurrency in a driver is not limited to “user processes making syscalls”. Several sources produce concurrent execution in your handlers:

- **Multiple open descriptors on the same device.** Each descriptor has its own struct `myfirst_fh`, but the device-wide struct `myfirst_softc` is shared. Two threads with different descriptors can concurrently enter the same handlers on the same `softc`.
- **Multiple threads sharing a descriptor.** Two pthreads in the same process can both call `read(2)` on the same file descriptor. Both enter the same handlers with the *same* `fh`. This is the subtler case, and the audit in Section 3 addresses it.
- **Sysctl readers.** A user running `sysctl dev.myfirst.0.stats` enters the sysctl handlers from a different thread than the I/O handlers. The handlers observe the same shared state.
- **Kernel threads.** If a future version of the driver spawns its own kernel thread (for timeouts, background work, etc.), that thread is a new source of concurrent access.
- **Callouts.** Similarly, a callout handler runs on its own, independently of the I/O handlers.

Every one of these is a concurrent stream. Every one must be accounted for in the locking strategy. The audit procedure from Section 3 explicitly lists them as possible access sources for each shared field.

11.2.10 Wrapping Up Section 1

Concurrency is the environment every FreeBSD driver lives in, whether the driver writer has thought about it or not. The streams of execution come from user syscalls, kernel threads, callouts, and interrupts. On a multi-core machine they are literally parallel; on any machine they are concurrent, which is what the bugs care about. When multiple streams touch shared state without coordination, four families of failure emerge: data corruption, lost updates, torn values, and inconsistent composite state.

We have not yet said what to do about any of it. The next two sections do the negative work, examining what goes wrong in precise detail. Section 4 starts the positive work, introducing the first primitive that can help. Section 5 introduces the second and more general one.

Take a break here if you need one. The next section is denser.

11.3 Section 2: Race Conditions and Data Corruption

Section 1 described the problem at the level of intuition. This section makes it concrete. We will define a race condition precisely, walk through how one arises in the circular buffer from Chapter 10, build a deliberately unsafe version of the driver that exhibits the bug, and run it on a real system. Watching the bug with your own eyes is the fastest way to develop the instinct you need for the rest of the book.

11.3.1 A Precise Definition

A **race condition** is a situation in which two or more streams of execution can access a piece of shared state in overlapping windows of time, at least one of those accesses is a write, and the correctness of the system depends on the exact order of those accesses.

Three properties make that definition do real work. The first is *overlap*: the accesses do not have to be literally simultaneous, they only have to have windows that touch. A read that starts at time t and ends at time $t+10$, and a write that starts at $t+5$ and ends at $t+15$, are overlapping. The second is *at least one write*: two concurrent reads of the same value are fine, because neither changes the state the other is reading. The third is *order-dependent correctness*: if the code is designed so that any interleaving produces the same outcome, there is no race by this definition, even if the windows overlap.

The first property is why software-level “it’s too fast to collide” reasoning never works. A read and a write are not two points; they are two intervals. Any overlap is enough.

The second property is why code that is read-only is safe without synchronization. If nothing is written, no interleaving can cause inconsistency. This is useful later when we talk about epoch(9) and lock-free reading, but the same observation justifies why certain purely-informational operations do not need protection.

The third property is why locks are not always the answer. If the operation is designed so that every legal interleaving is acceptable, no lock is needed. A counter that is only ever incremented by atomic operations, for example, does not need a lock around each increment. The lock would still work, but it would be wasted cycles.

11.3.2 The Anatomy of a Race

Let us take the counter example from Section 1 and walk through it at the instruction level. The C statement:

```
shared_counter = shared_counter + 1;
```

On amd64, with ordinary compilation, this produces something close to:

```
movl    shared_counter(%rip), %eax    ; load shared_counter into register EAX
addl     $1, %eax                     ; increment EAX by 1
movl     %eax, shared_counter(%rip)   ; store EAX back to shared_counter
```

The memory access happens at two distinct moments: the load and the store. Between them, the value being computed is held in the CPU’s register file, not in memory. Any other CPU observing memory during that interval sees the old value, not the value in flight.

If two threads, A and B, execute this sequence on two CPUs at overlapping times, the memory controller sees four accesses, in some order:

1. A's load
2. B's load
3. A's store
4. B's store

The final value of `shared_counter` depends entirely on the order in which the stores commit. If A's store happens first and B's store happens second, the final value is 1 (because B's store writes the value B computed, which was based on B's load, which saw 0). If the order is reversed, the final value is still 1. There is no interleaving in which both increments take effect, because the reads happened before either write, and each write overwrites the other.

The fix is to make the whole sequence either atomic (so that the four accesses become two inseparable transactions) or mutually exclusive (so that one thread finishes all three instructions before the other starts any of them). Both are legal answers. The difference is what they cost and what they allow us to do elsewhere in the code.

11.3.3 How Races Appear in the Circular Buffer

The shared counter is the simplest possible example. The circular buffer from Chapter 10 is a richer one. It has multiple fields, interrelated invariants, and operations that touch several fields at once. This is where race conditions become really dangerous.

Recall the `cbuf` from Chapter 10:

```
struct cbuf {  
    char    *cb_data;  
    size_t   cb_size;  
    size_t   cb_head;  
    size_t   cb_used;  
};
```

The invariants we depend on are:

- `cb_head < cb_size` (the head index is always valid).
- `cb_used <= cb_size` (the live byte count never exceeds capacity).
- The bytes at positions `[cb_head, cb_head + cb_used) mod cb_size` are the live bytes.

These invariants relate multiple fields. A `cbuf_write` call, for example, updates `cb_used`. A `cbuf_read` call updates both `cb_head` and `cb_used`. If a reader and a writer execute concurrently without coordination, each of those updates can be observed in a half-finished state.

Consider what happens if `myfirst_read` and `myfirst_write` run concurrently on two CPUs, with no mutex:

1. Reader thread starts `cbuf_read` on a buffer with `cb_head = 1000`, `cb_used = 2000`. It computes `first = MIN(n, cb_size - cb_head)` and begins the first `memcpy` from `cb_data + 1000` into its stack bounce.

2. Meanwhile, writer thread calls `cbuf_write`, which updates `cb_used`. The writer reads `cb_used = 2000`, computes the tail as $(1000 + 2000) \% \text{cb\_size} = 3000$, does its `memcpy` into the buffer, and writes `cb_used = 2100`.
3. The reader finishes its first `memcpy`, possibly seeing bytes that the writer had just written (if the writer's `memcpy` crossed over the reader's position).
4. The reader then updates `cb_head = (1000 + n) \% \text{cb\_size}` and decrements `cb_used`.

Several things can go wrong here:

- **Torn `cb_used`:** on a 32-bit machine, `cb_used` is a 32-bit type, so its read and write are atomic. On amd64, `size_t` is 64-bit, and aligned 64-bit accesses are also atomic, so a torn read is unlikely. But if we ever used a structure larger than the word size, we would see bytes from the old value glued to bytes from the new.
- **Observed invariant violation:** the reader's decrement of `cb_used` and the writer's increment of `cb_used` happen at different moments. Between them, `cb_used` may transiently reflect only one of the two operations. If a third reader (for example, a `sysctl` handler) observes `cb_used` in the middle of that interleaving, the observed value is neither the "before" nor the "after" of either individual operation.
- **Concurrent `memcpy` into overlapping regions:** if the reader's `memcpy` and the writer's `memcpy` touch the same byte, the byte's final value is whatever wins the race for memory. The reader may see half of its intended data overwritten with new data from the writer. The writer may overwrite bytes that the reader has already copied into its bounce, meaning the reader returns bytes that do not exist in the buffer any more.

Each of those is a real bug. Each is hard to observe, because the timing window for the corruption is tiny. Each is lethal, because a single instance can corrupt the data a user program depends on.

11.3.4 A Deliberately Unsafe Driver

Theory becomes visceral when you run the code. We are going to build a version of `myfirst` that has the mutex removed from the I/O handlers. You will load it, run a concurrent test against it, and watch the driver produce nonsense.

Warning: load this driver only on a lab machine with no important state. It can corrupt its own state at will and, under some scenarios, wedge the machine. Expect to reboot if the test goes badly. The point is to see the bug, not to push it further.

Create a new directory alongside your Chapter 10 Stage 4 source:

```
$ cd examples/part-03/ch11-concurrency
$ mkdir -p stage1-race-demo
$ cd stage1-race-demo
```

Copy `cbuf.c`, `cbuf.h`, and `Makefile` from the Chapter 10 Stage 4 directory. Then copy `myfirst.c` and edit it to remove the mutex from `myfirst_read` and `myfirst_write`. The easiest way to do this, so you can see exactly what changed, is to use `sed` with a sentinel:

```
$ sed \
-e 's/mtx_lock(&sc->mtx);/\/* RACE: mtx_lock removed *//' \
-e 's/mtx_unlock(&sc->mtx);/\/* RACE: mtx_unlock removed *//' \
.../part-02/ch10-handling-io-efficiently/stage4-poll-refactor/myfirst.c \
> myfirst.c
```

What this does is turn every `mtx_lock` and `mtx_unlock` in the source into a comment. It leaves the rest of the code unchanged, including `mtx_sleep` and `mtx_assert`. This is a surgical deletion of the lock acquisition points; the rest of the driver still thinks the lock is there.

Before we build, open `myfirst.c` and look for the `mtx_assert(&sc->mtx, MA_OWNED)` calls inside `myfirst_buf_read`, `myfirst_buf_write`, and the wait helpers. These are going to fire a KASSERT panic on an INVARIANTS kernel, because the helpers check that the mutex is held and now it never is. This is a feature, not a bug: if you accidentally run this driver on a debug kernel, the assertion will catch the problem before any real damage is done. For the explicit race demo, we want to see the corruption itself, so comment out the `mtx_assert` lines or disable INVARIANTS in the test kernel:

```
$ sed -i '' -e 's|mtx_assert(&sc->mtx, MA_OWNED);|/\/* RACE: mtx_assert removed */|g'
↵ myfirst.c
```

The `-i ''` is FreeBSD's `sed` in-place mode. On Linux, it would be `sed -i` without the quoted empty string.

Also delete the `mtx_init`, `mtx_destroy`, and `mtx_lock/mtx_unlock` calls from `myfirst_attach` and `myfirst_detach` by hand; the `sed` commands above do not catch the attach and detach paths because those have `mtx_lock` calls we want to keep logically (even if they do nothing). A simpler and cleaner approach, if you prefer, is to introduce a macro at the top of `myfirst.c`:

```
/* DANGEROUS: deliberately unsafe driver for Chapter 11 Section 2 demo. */
#define RACE_DEMO

#ifdef RACE_DEMO
#define MYFIRST_LOCK(sc)      do { (void)(sc); } while (0)
#define MYFIRST_UNLOCK(sc)   do { (void)(sc); } while (0)
#define MYFIRST_ASSERT(sc)   do { (void)(sc); } while (0)
#else
#define MYFIRST_LOCK(sc)      mtx_lock(&(sc)->mtx)
#define MYFIRST_UNLOCK(sc)    mtx_unlock(&(sc)->mtx)
#define MYFIRST_ASSERT(sc)    mtx_assert(&(sc)->mtx, MA_OWNED)
#endif
```

Then globally replace `mtx_lock(&sc->mtx)` with `MYFIRST_LOCK(sc)` and similarly for the rest. With `RACE_DEMO` defined, the macros do nothing; without it, they are the original calls. This is the approach the companion source uses. It lets you toggle the race on and off at compile time.

Build the module:


```
$ make
$ kldstat | grep myfirst && kldunload myfirst
$ kldload ./myfirst.ko
$ dmesg | tail -5
```

The attach line should appear as usual. Nothing looks wrong yet.

11.3.5 Running the Race

With the unsafe driver loaded, run the producer/consumer test from Chapter 10:

```
$ cd ../../part-02/ch10-handling-io-efficiently/userland
$ ./producer_consumer
```

Recall that this test produces a fixed pattern into the driver, reads it back in another process, and compares checksums. On the safe driver, the checksums match and the test reports zero mismatches. Run it against the unsafe driver and the result depends on your timing:

- **Likely outcome:** the two checksums differ, and the reader reports some number of mismatches. The exact number varies between runs.
- **Possible outcome:** the test hangs forever, because the buffer state became inconsistent and the reader or writer got stuck waiting for a condition that will never be true.
- **Rare outcome:** the test passes by chance. If this happens, run it again; a single run of a concurrency test tells you almost nothing.
- **Unpleasant outcome:** the kernel panics because of invariants fired inside `cbuf_read` or `cbuf_write` (for example, a negative `cb_used` that was meant to be unsigned, or an out-of-bounds `cb_head`). If this happens, note the backtrace, reboot, and carry on. This is what you came to see.

In most runs you will see output like:

```
writer: 1048576 bytes, checksum 0x8bb7e44c
reader: 1043968 bytes, checksum 0x3f5a9b21, mismatches 2741
exit: reader=2 writer=0
```

The reader got fewer bytes than the writer produced. Its checksum is different. Some specific number of byte positions held the wrong value.

This is not a flaky test. This is a driver working as designed, given that the locking has been removed. The kernel schedules multiple threads onto multiple CPUs. Those threads read and write the same memory without coordination. The result is the state that any specific hardware interleaving happens to produce.

Run the test several times. The numbers will vary. The character of the corruption, however, will not: the writer will claim to have produced some number of bytes, the reader will see a different set. That is what concurrency without synchronization looks like. It is not mysterious or subtle; it is the exact consequence of the definitions in the earlier part of this section.

11.3.6 Observing the Damage with Logs

The producer-consumer test reports aggregate damage. For finer-grained observation, instrument the driver with `device_printf` (guarded by the `debug sysctl`) inside `cbuf_write` and `cbuf_read`. Add something like:

```
device_printf(sc->dev,
    "cbuf_write: tail=%zu first=%zu second=%zu cb_used=%zu\n",
    tail, first, second, cb->cb_used);
```

at the bottom of `cbuf_write`, behind the `myfirst_debug` check you set up in Chapter 10 Stage 2. The same for `cbuf_read`.

Run the unsafe driver again with `sysctl dev.myfirst.debug=1` and watch `dmesg -t` in another terminal:

```
$ sysctl dev.myfirst.debug=1
$ ./producer_consumer
$ dmesg -t | tail -40
```

You will see lines like:

```
myfirst0: cbuf_write: tail=3840 first=256 second=0 cb_used=2048
myfirst0: cbuf_read: head=0 first=256 second=0 cb_used=1792
myfirst0: cbuf_write: tail=1536 first=256 second=0 cb_used=2304
myfirst0: cbuf_read: head=2048 first=256 second=0 cb_used=-536 <-- inconsistent
```

`cb_used` is a `size_t`, which is unsigned, so “negative” here means the field wrapped: a very large number that was formatted with `%zu` and still looks enormous but was not meant to be. That wraparound is one of the failure modes we discussed. A concurrent decrement of `cb_used` read the value from before another thread’s increment finished, subtracted its own amount, and stored a result that is nonsense.

Looking at a log full of such entries drives home the point the definitions did not: concurrency bugs are not rare edge cases. They are the normal case when multiple threads touch the same fields without coordination. The driver does not break occasionally. It breaks constantly. It just sometimes happens to produce a superficially plausible output anyway.

11.3.7 Cleaning Up

Unload the unsafe driver before doing anything else:

```
$ kldunload myfirst
```

If `kldunload` hangs (because a test still holds a descriptor open), find and kill the test process. If the kernel is in a broken state after a panic-like failure, reboot. From here on, work only with the safe driver unless you are deliberately rerunning the demo to compare.

Do not leave the unsafe driver loaded. Do not check in the `RACE_DEMO`-enabled build and forget about it. The companion tree keeps the race demo in a sibling directory precisely so that it cannot be confused with the production source.

11.3.8 The Lesson

Three observations follow from the exercise.

The first is that the mutex in the Chapter 10 driver is not stylistic. Remove it and the driver breaks, instantly, in a way you can demonstrate on any FreeBSD machine. The mutex is load-bearing. It protects exactly the property (the cbuf's internal consistency) that the rest of the driver's correctness depends on.

The second is that testing cannot find this class of bug reliably. If you did not know to run a multi-process stress test, if you tested only with a single cat and a single echo, the driver would look fine. The concurrency bug would be there, invisible, waiting for the first user whose workload happens to involve two concurrent callers. Reviewing code with concurrency in mind is not a substitute for testing; it is the thing that prevents you from shipping code whose tests happen to pass.

The third is that the fix is not a local patch. You cannot sprinkle atomic operations on a field or two and call it done. The cbuf has composite invariants that span multiple fields; protecting it requires a region of code where no other thread can be executing at the same time. That is the job of a mutex, and Section 5 is where we look at mutexes in full. Before we get there, however, we should audit the existing driver (Section 3) and understand the more limited tool that atomics provide (Section 4).

11.3.9 A Second Race: Attach Versus I/O

Not every race is in the data path. Let us look at a race we fixed in Chapter 10 without naming: the race between `myfirst_attach` and the first user to call `open(2)`.

When `myfirst_attach` runs, it does several things in sequence:

1. Sets up `sc->dev` and `sc->unit`.
2. Initializes the mutex.
3. Sets `sc->is_attached = 1`.
4. Initializes the cbuf.
5. Creates the cdev via `make_dev_s`.
6. Creates the cdev alias.
7. Registers the sysctls.
8. Returns.

The cdev is not visible to user space until step 5. A user who calls `open("/dev/myfirst")` before step 5 completes gets `ENOENT`, because the device node does not exist yet. After step 5, the cdev is registered with `devfs`, and `open` can succeed.

If step 5 happened before step 3, a user could open the device and call `read` before `is_attached` was set. The read handler's `if (!sc->is_attached) return (ENXIO)` check would fail the call, returning `ENXIO` even though attach was in progress. That is not catastrophic, but it is confusing and avoidable.

The fix is the order we use: `is_attached = 1` happens before `make_dev_s`. By the time any handler can run, `is_attached` is already set.

The subtle point is that this ordering is *only correct* because single-threaded attach cannot be interrupted between the two writes. If the writes could be reordered by the compiler or the hardware (which they cannot for plain integer stores on amd64 without explicit barriers, but could on some weakly-ordered architectures), we would

need an `atomic_store_rel_int` for the `is_attached` write. The book stays on amd64, so the plain store is fine.

This is a generally useful discipline. Every time you write attach code, list the observable preconditions for each subsequent step and verify they are established before the step is reachable from user space. The order matters, and it is almost always wrong to make the `cdev` before the `softc` is fully ready.

11.3.10 A Third Race: `sysctl` Versus I/O

Another race the Chapter 10 design handles: the `sysctl` handler that reads `cb_used` or `cb_free` must return a self-consistent value, not a torn composite.

The fields `cb_head` and `cb_used` are both `size_t`. Either one, read alone, gives a single-word value that is atomic on amd64. But `cb_used` and `cb_head` together form an invariant (the live bytes are at `[cb_head, cb_head + cb_used) mod cb_size`). A `sysctl` that reads both without the mutex could observe them at inconsistent moments.

For `cb_used` alone, the race is tolerable: the counter may be slightly stale, but the value returned is at least *some* value that was once true. For `cb_head`, the same applies. For anything that combines the two (like “how many bytes until the tail meets the capacity boundary?”), the race produces meaningless numbers.

We protect the `sysctl` reads with the mutex. The critical section is tiny (one load), so contention is minimal; the correctness benefit is that the returned value is guaranteed to be self-consistent.

The rule this illustrates is: **if your observation combines multiple fields that together express an invariant, protect the observation with the same primitive that protects the invariant.**

11.3.11 The Race Taxonomy, Revisited

With the `cbuf` race (Section 2 main text), the attach race, and the `sysctl` race, we now have three concrete examples to classify under the four failure modes from Section 1.

- **Cbuf data corruption:** primarily inconsistent composite state (the `cb_used/cb_head` invariant), with some lost updates on `cb_used` itself.
- **Attach race:** a form of inconsistent composite state, where the “is the device ready?” invariant is transiently false from the user’s perspective.
- **Sysctl torn observation:** a form of inconsistent composite state, where the reader’s computation uses fields from different moments in time.

All three are protected by the same mechanism: serialize access to the composite state with a single mutex. This unity is not a coincidence; it is the reason a single mutex is usually the right answer for a small driver.

11.3.12 Wrapping Up Section 2

A race condition is a very specific thing: overlapping access to shared state, at least one write, correctness that depends on the interleaving. The `cbuf`’s composite invariants make it a rich source of races when the mutex is removed. A deliberately unsafe driver reveals the corruption within seconds of running a concurrent test.

You have now seen the problem in your own terminal, in your own driver. You have seen three distinct races, all in places where composite state is involved, all solved by the same mechanism. Everything that follows is the construction of the tools that make the problem go away, one layer at a time.

11.4 Section 3: Analyzing Unsafe Code in Your Driver

Most driver writers, most of the time, do not create race conditions deliberately. They create them by not noticing that a piece of state is shared across streams of execution. The defence against this is a habit, not a clever tool: every time you touch state that *could* be accessed from outside the current code path, you stop and ask whether it is actually safe to touch without synchronization. Most of the time the answer is “yes, it’s already protected by X”, and you move on. Sometimes the answer is “actually, it isn’t protected, and I need to think about it”. That second case is the one this section teaches you to recognise.

The Chapter 10 Stage 4 driver is reasonably thoughtful. It uses a mutex, it documents what the mutex protects, and it follows consistent conventions. But even a reasonably thoughtful driver benefits from an explicit audit. The goal of this section is to walk through the driver and classify every piece of shared state: who reads it, who writes it, what protects it, and under what conditions. The result is a small document that accompanies the code, makes the concurrency story auditable, and becomes a checkpoint you can use when future changes risk breaking the model.

11.4.1 What “Shared” Means

A variable is **shared** if more than one stream of execution can access it during the same time window. “Same time window” does not require literal simultaneity; it requires that the two accesses could *potentially* overlap.

That definition is broader than beginners usually expect. Three examples clarify it.

A local variable inside a function is not shared. Each invocation of the function gets its own stack frame, and the variable lives in that frame. Another thread can enter the same function, but it will get its own stack and its own copy of the variable. There is no shared memory.

A static variable inside a function is shared. Despite the `static` keyword making it look local, the storage is file-scope: all invocations of the function see the same byte of memory. Two concurrent invocations of the function share that byte.

A field in a dynamically allocated structure is shared if and only if more than one stream of execution has a pointer to the same structure. The `struct myfirst_softc` for a given device is one such structure. Every call into the driver that goes through `dev->si_drv1` observes the same `_softc`. Two concurrent calls see the same `_softc`.

A more subtle variant: a field in a per-descriptor structure (our `struct myfirst_fh`) is shared only if more than one stream has access to the same file descriptor. Two processes with distinct descriptors do not share each other’s `struct myfirst_fh`. Two threads in the same process sharing a descriptor *do*. Most driver writers treat per-descriptor state as “almost” unshared and use lighter protection than device-wide state, but the word “almost” is doing heavy lifting; we will come back to this case in Section 5.

11.4.2 The Shared State of myfirst

Open your Stage 4 source. We are going to annotate, field by field, the contents of `struct myfirst_softc`. You can do this in the source itself (as a comment block near the struct declaration) or in a separate design document. I recommend doing both: a short summary in the source and a detailed version in a file you can evolve.

Here is the structure, as of Chapter 10 Stage 4:

```
struct myfirst_softc {
    device_t          dev;

    int               unit;

    struct mtx        mtx;

    uint64_t          attach_ticks;
    uint64_t          open_count;
    uint64_t          bytes_read;
    uint64_t          bytes_written;

    int               active_fhs;
    int               is_attached;

    struct cbuf        cb;

    struct selinfo     rsel;
    struct selinfo     wsel;

    struct cdev        *cdev;
    struct cdev        *cdev_alias;

    struct sysctl_ctx_list sysctl_ctx;
    struct sysctl_oid  *sysctl_tree;
};
```

For each field we ask five questions:

1. **Who reads it?** Which functions, and from which streams of execution?
2. **Who writes it?** Same question, but for writes.
3. **What protects the accesses from race conditions?**
4. **What invariants does it participate in?**
5. **Is the current protection sufficient?**

Going field by field is tedious. It is also the whole point. A driver whose concurrency story is documented field by field is a driver that almost certainly works. A driver whose concurrency story is left implicit is a driver whose author hoped for the best.

sc->dev: the Newbus device_t handle. - Read by: every handler (indirectly, through sc->dev in device_printf and similar). - Written by: attach (once, during module load). - Protection: write happens before any handler can run (the cdev is not registered until after sc->dev is set). After attach completes, the field is effectively immutable. No lock needed. - Invariants: sc->dev != NULL once attach has returned zero. - Sufficient? Yes.

sc->unit: a convenience copy of device_get_unit(dev). - Same analysis as sc->dev. Immutable after attach. No lock needed.

sc->mtx: the sleep mutex. - “Accessed” by every handler in mtx_lock/mtx_unlock/mtx_sleep calls. - Initialized by mtx_init in attach, destroyed by mtx_destroy in detach. - Protection: the mutex protects itself; the init and destroy are sequenced relative to the cdev’s creation and destruction so that no handler can ever see a half-initialized mutex. - Invariants: the mutex is initialized after mtx_init returns and before mtx_destroy is called. Handlers only run during that window. - Sufficient? Yes.

sc->attach_ticks: the ticks value at attach time, for informational purposes. - Read by: the sysctl handler for attach_ticks. - Written by: attach (once). - Protection: write happens before any sysctl handler can be called. After that, immutable. No lock needed. - Invariants: none beyond “set at attach”. - Sufficient? Yes.

sc->open_count: lifetime count of opens. - Read by: the sysctl handler for open_count. - Written by: myfirst_open. - Protection: the write happens under sc->mtx in myfirst_open. The read, currently, is an unprotected load from the SYSCTL_ADD_U64 handler. - Invariants: monotonically increasing. - Sufficient? On amd64, a 64-bit aligned load is atomic, so a torn read is not possible. The sysctl may observe an old value (a load that happened just before a concurrent increment) but not a corrupted one. This is acceptable for an informational counter. On 32-bit platforms, the same load would not be atomic, and a torn read would be possible. For this book’s purposes (amd64 lab machine), the current protection is sufficient. We will note this in the documentation.

sc->bytes_read, sc->bytes_written: lifetime byte counters. - Same analysis as open_count. Written under the mutex, read unprotected from sysctl, acceptable on amd64 but not 32-bit. These are candidates for migration to counter(9) per-CPU counters, which would remove the torn-read concern and also scale better under contention. Section 4 does this migration.

sc->active_fhs: current count of open descriptors. - Read by: sysctl handler, myfirst_detach. - Written by: myfirst_open, myfirst_fh_dtor. - Protection: all reads and writes happen under sc->mtx. - Invariants: active_fhs >= 0. active_fhs == 0 when detach is called and succeeds. - Sufficient? Yes.

sc->is_attached: flag indicating whether the driver is in the attached state. - Read by: every handler at entry (to fail-fast with ENXIO if the device has been torn down). - Written by: attach (sets to 1), detach (sets to 0, under the mutex). - Protection: the write in detach is under the mutex. The reads in the handlers at entry are *not* under the mutex (they are unprotected plain loads). - Invariants: is_attached == 1 while the device is usable. - Sufficient? There is a subtle question here. A handler that reads is_attached == 1 without the mutex might then proceed to acquire the mutex, and by the time it has the mutex, detach may have run and set the flag to 0. Our handlers handle this case by re-checking is_attached after every sleep (the if (!sc->is_attached) return (ENXIO); pattern after mtx_sleep). The entry check is an optimization that avoids taking the mutex in the common case; correctness does not depend on it. We will document this explicitly.

sc->cb: the circular buffer (all its fields). - Read by: myfirst_read, myfirst_write, myfirst_poll, the sysctl handlers for cb_used and cb_free. - Written by: myfirst_read, myfirst_write, attach (via cbuf_init), detach (via cbuf_destroy). - Protection: every access is under sc->mtx, including the sysctl

handlers (which take the mutex around a brief read). `cbuf_init` runs in `attach` before the `cdev` is registered, so it is effectively single-threaded. `cbuf_destroy` runs in `detach` after all descriptors have been confirmed closed, so it is also effectively single-threaded. - Invariants: `cb_used <= cb_size`, `cb_head < cb_size`, the bytes at `[cb_head, cb_head + cb_used) mod cb_size` are live. - Sufficient? Yes. This is the field the Chapter 10 work spent most of its time getting right.

`sc->rsel, sc->wsel`: `selinfo` structures for `poll(2)` integration. - Read/written by: `selrecord(9)`, `selwakeup(9)`, `seldrain(9)`. These functions handle their own locking internally; the caller's responsibility is merely to call them at the right times. - Protection: internal to the `selinfo` machinery. The caller holds `sc->mtx` around `selrecord` (because it is called inside `myfirst_poll`, which takes the mutex) and drops `sc->mtx` around `selwakeup` (because it may sleep). - Invariants: the `selinfo` is initialised once (it is zero-initialized by the `softc` allocation) and drained at `detach`. - Sufficient? Yes.

`sc->cdev, sc->cdev_alias`: pointers to the `cdev` and its alias. - Read by: nothing critical; they are stored for destruction. - Written by: `attach` (set), `detach` (destroyed and cleared). - Protection: writes happen at known lifecycle points before or after handlers can run. - Invariants: `sc->cdev != NULL` during the attached window. - Sufficient? Yes.

`sc->sysctl_ctx, sc->sysctl_tree`: the `sysctl` context and root node. - Managed by the `sysctl` framework. The framework handles its own locking. - Protection: framework-internal. - Sufficient? Yes.

11.4.3 The Per-Descriptor State

Now the per-descriptor state, in struct `myfirst_fh`:

```
struct myfirst_fh {
    struct myfirst_softc    *sc;
    uint64_t                reads;
    uint64_t                writes;
};
```

`fh->sc`: back-pointer to the `softc`. - Read by: handlers that retrieve it via `devfs_get_cdevpriv`. - Written by: `myfirst_open`, once, before the `fh` is handed to `devfs`. - Protection: the write is sequenced before any handler can see the `fh`. - Invariants: immutable after `myfirst_open`. - Sufficient? Yes.

`fh->reads, fh->writes`: per-descriptor byte counters. - Read by: the `myfirst_fh_dtor` destructor (for a final log message) and potentially future `sysctl` handlers per-descriptor (not currently exposed). - Written by: `myfirst_read`, `myfirst_write`. - Protection: the writes happen under `sc->mtx` (inside the read/write handlers). Two threads holding the same descriptor open could in principle write concurrently; both would hold `sc->mtx`, so the accesses serialize.

This is the subtle case noted earlier. Two threads in the same process can share a file descriptor. Both can call `read(2)` on it. Both will enter `myfirst_read`, both will retrieve the same `fh` via `devfs_get_cdevpriv`, both will want to update `fh->reads`. Because both hold `sc->mtx` during the update, the operations serialize, and no race occurs. If we ever wanted to hold less than the full `sc->mtx` (for instance, if we used a lock-free counter update), the per-descriptor case would force us to be more careful.

11.4.4 Classifying the Critical Sections

With the fields inventoried, we can now identify every **critical section** in the code. A critical section is a contiguous region of code that accesses shared state and must execute without interference from concurrent streams.

Go through `myfirst.c` and find every region bounded by `mtx_lock(&sc->mtx)` and `mtx_unlock(&sc->mtx)`. Each of those regions is a critical section by construction. Then look for regions that should be critical sections but are not. There should be none in the Stage 4 driver, but the audit is still valuable for confirming the gap.

The critical sections in Stage 4 are:

1. **myfirst_open**: updates `open_count` and `active_fhs` under the mutex.
2. **myfirst_fh_dtor**: decrements `active_fhs` under the mutex.
3. **myfirst_read**: multiple critical sections, each bracketing an access to the `cbuf`, separated by the out-of-lock `uiomove` calls. The wait helper runs inside the critical section; the byte counters are updated inside; `wakeup` and `selwakeup` are called with the mutex dropped.
4. **myfirst_write**: the mirror of `myfirst_read`.
5. **myfirst_poll**: one critical section that checks the `cbuf` state and either sets `revents` or calls `selrecord`.
6. **The two `cbuf` sysctl handlers**: each takes the mutex briefly to read `cbuf_used` or `cbuf_free`.
7. **myfirst_detach**: acquires the mutex to check `active_fhs` and to set `is_attached = 0` before waking sleepers.

Each of these should do as little work as possible while holding the mutex. Work that does not need the mutex (the `uiomove`, for example, or any computation on stack-local values) should happen outside. The Stage 4 driver is already careful about this; the audit confirms it.

11.4.5 Annotating the Source

One useful output of the audit is to go back to the source and add one-line comments above each critical section that name the shared state being protected. This is not decorative; it is documentation for the next person to read the code.

For example, the `myfirst_read` handler becomes:

```
static int
myfirst_read(struct cdev *dev, struct uio *uio, int ioflag)
{
    /* ... entry logic ... */
    while (uio->uio_resid > 0) {
        /* critical section: cbuf state + bytes_read + fh->reads */
        mtx_lock(&sc->mtx);
        error = myfirst_wait_data(sc, ioflag, nbfore, uio);
        if (error != 0) {
            mtx_unlock(&sc->mtx);
            return (error == -1 ? 0 : error);
        }
    }
}
```

```

        take = MIN((size_t)uio->uio_resid, sizeof(bounce));
        got = myfirst_buf_read(sc, bounce, take);
        fh->reads += got;
        mtx_unlock(&sc->mtx);

        /* not critical: wake does its own locking */
        wakeup(&sc->cb);
        selwakeup(&sc->wsel);

        /* not critical: uiomove may sleep; mutex must be dropped */
        error = uiomove(bounce, got, uio);
        if (error != 0)
            return (error);
    }
    return (0);
}

```

Three comments. Thirty seconds of work. They transform the handler from a piece of code that a reader has to analyze to a piece of code whose concurrency story is self-evident.

11.4.6 A Locking-Strategy Document

Alongside the source annotations, maintain a short document that describes the overall strategy. The file `LOCKING.md` in your driver source tree is a good location. A minimal version looks like this:

```

# myfirst Locking Strategy

## Overview
The driver uses a single sleep mutex (sc->mtx) to serialize all
accesses to the circular buffer, the sleep channel, the selinfo
structures, and the per-descriptor byte counters.

## What sc->mtx Protects
- sc->cb (the circular buffer's internal state)
- sc->bytes_read, sc->bytes_written
- sc->open_count, sc->active_fhs
- sc->is_attached (writes; reads may be unprotected as an optimization)
- sc->rsel, sc->wsel (through the selinfo API; the mutex is held
  around selrecord, dropped around selwakeup)

## Locking Discipline
- Acquire with mtx_lock, release with mtx_unlock.
- Wait with mtx_sleep(&sc->cb, &sc->mtx, PCATCH, wmesg, 0).
- Wake with wakeup(&sc->cb).

```

- NEVER hold `sc->mtx` across `uiomove`, `copyin`, `copyout`, `selwakeup`, or `wakeup`. Each of these may sleep or take other locks.
- All four `myfirst_buf_*` helpers assert `mtx_assert(MA_OWNED)`.

Known Non-Locked Accesses

- `sc->is_attached` at handler entry: unprotected read. Safe because a stale true is re-checked after every sleep; a stale false is harmless (the handler returns `ENXIO`, which is also what it would do with a fresh false).
- `sc->open_count`, `sc->bytes_read`, `sc->bytes_written` at `sysctl` read time: unprotected load. Safe on amd64 (aligned 64-bit loads are atomic). Would be unsafe on 32-bit platforms.

Lock Order

`sc->mtx` is currently the only lock the driver holds. No ordering concerns arise. Future versions that add additional locks must specify their order here.

That document is the definitive answer to “is this driver safe?”. It is reviewable. It can be checked against the code. It is the output of the audit we just performed.

11.4.7 A Practical Audit Procedure

The procedure we just walked through can be systematized:

1. List every field of every shared structure (softc, per-descriptor, any module-global state).
2. For each field, identify all read sites and all write sites.
3. For each field, identify the protection mechanism (lock held, lifecycle guarantee, framework-internal, atomic type, intentionally unprotected).
4. For each field, identify the invariants it participates in and the other fields those invariants involve.
5. For each invariant, confirm that the protection mechanism covers all relevant accesses.
6. For each known-unprotected access, document why it is acceptable.
7. Annotate the source code to name each critical section and each deliberate unprotected access.
8. Write the summary into a `LOCKING.md` file.

It sounds laborious. For a driver the size of `myfirst`, it takes under an hour. For a driver ten times the size, it takes a day. Either way, it pays for itself the first time it catches a bug that was not yet written.

11.4.8 Common Audit Discoveries

Three patterns recur. If your audit finds any of them, treat them as red flags.

The first is **silent sharing**: a field that you thought was local but is actually accessible from outside the current code path. The usual cause is a pointer that leaks from a structure you thought was private. `devfs_set_cdevpriv` gives a `cdev` a pointer to a per-descriptor structure; if that structure contains a back-pointer,

another thread can reach into it. Per-descriptor state that is accessed only from within the current thread is safer than state reachable from the softc.

The second is **irregular protection**: a field that is protected in most code paths but not in one. Often the unprotected path is the one added most recently. The audit catches it, the unit tests do not.

The third is **invariant drift**: over time, a field acquires new correlations with other fields, and the protection stops covering the composite invariant. The audit reveals it by asking what invariants each field participates in.

A disciplined audit, repeated periodically, catches all three.

11.4.9 A Second Worked Audit: A Hypothetical Multi-Queue Driver

To exercise the audit procedure on something slightly more interesting, imagine a driver that is not `myfirst`. Suppose we are auditing a hypothetical network-adjacent pseudo-device, `netsim`, which simulates a packet source. The softc has more shared state:

```
struct netsim_softc {
    device_t      dev;
    struct mtx    global_mtx;
    struct mtx    queue_mtx;

    struct netsim_queue rx_queue;
    struct netsim_queue tx_queue;

    struct callout tick_callout;
    int            tick_rate_hz;

    uint64_t      packets_rx;
    uint64_t      packets_tx;
    uint64_t      dropped;

    int            is_up;
    int            is_attached;
    int            active_fhs;

    struct cdev    *cdev;
};
```

This is a richer structure. Let us apply the audit procedure.

Two locks. `global_mtx` and `queue_mtx`. We need to decide what each protects and establish an order.

A reasonable split: `global_mtx` protects configuration and lifecycle state (`is_up`, `is_attached`, `tick_rate_hz`, `active_fhs`, `tick_callout`). `queue_mtx` protects the RX and TX queues, which are the high-frequency hot path.

Lock order. Because configuration changes are rare and queue operations are hot, we order: `global_mtx` before `queue_mtx`. A thread that holds `global_mtx` may acquire `queue_mtx`. A thread holding `queue_mtx` may not acquire `global_mtx`.

This is a choice, not an edict. A different driver might reverse it. What matters is consistency. `WITNESS` will catch any violation.

Counter fields. `packets_rx`, `packets_tx`, `dropped` are counters updated on the data path. `counter(9)` is the right tool.

The callout. `tick_callout` fires periodically to generate packets. Its handler runs on a kernel thread, concurrently with user-driven read/write. The callout handler takes `queue_mtx` to enqueue packets. Because the callout is initialised before the `cdev` is registered and is stopped on detach, the callout and the user handlers are properly bounded.

The invariants.

- `is_attached == 1` during the attached window; detach clears it under `global_mtx`.
- The queue structures have their own internal invariants (head/tail indices); `queue_mtx` protects them.
- `tick_rate_hz` is read in the callout; it is written in attach and in `ioctl`s; reads are under `global_mtx` so the rate cannot change mid-tick.

What to write in `LOCKING.md`:

Locks

`global_mtx` (MTX_DEF)

Protects: `is_up`, `is_attached`, `active_fhs`, `tick_rate_hz`, `tick_callout`.

`queue_mtx` (MTX_DEF)

Protects: `rx_queue`, `tx_queue` internal state.

Lock Order

`global_mtx` -> `queue_mtx`

A thread holding `global_mtx` may acquire `queue_mtx`.

A thread holding `queue_mtx` may NOT acquire `global_mtx`.

Lock-Free Fields

`packets_rx`, `packets_tx`, `dropped`: `counter_u64_t`. No lock required.

Callout

`tick_callout` fires every `1/tick_rate_hz` seconds. It acquires `queue_mtx` to enqueue packets. It does not acquire `global_mtx`. Stopping the callout is done in detach, under `global_mtx`, with `callout_drain`.

Notice that the audit produced a document that looks very much like the one for `myfirst`, even though the

driver is larger. The structure is the same: locks, what they protect, the order, rules, lock-free fields, notable subsystems.

That repeatability is the point of having an audit procedure. Every driver's LOCKING.md looks similar in shape, because the questions you ask during the audit are the same. Only the answers differ.

11.4.10 How Audits Catch Bugs

Three concrete examples of bugs that a careful audit would catch:

Example 1: Missing lock on an update. A new sysctl handler is added that allows the user to change `tick_rate_hz` at runtime. The handler writes the field without taking `global_mtx`. The audit catches this: the field is protected by `global_mtx`, but one write path does not take the lock.

Example 2: Lock order inversion. A new function, `netlim_rebalance`, is added. It takes `queue_mtx` to inspect the queue lengths, then takes `global_mtx` to update configuration based on the lengths. This is the wrong order. The audit catches this by asking: “for every function, does its lock acquisition match the global order?”.

Example 3: Torn read of `packets_rx`. A sysctl handler reads `packets_rx` with a plain 64-bit load. On amd64 this is fine; on 32-bit platforms it is not. The audit catches this by documenting which architectures the driver targets and flagging platform-dependent assumptions.

Each of these bugs is the kind that evades single-threaded testing. The audit catches them by being systematic.

11.4.11 Audits as Change Gates

In a mature driver project, the audit becomes a change gate: any commit that modifies shared state, adds a new field, or changes a lock acquisition is required to update LOCKING.md in the same commit. Reviewers check that the update is consistent with the code change.

This discipline sounds bureaucratic. In practice, it is fast (updating the doc takes a minute) and it is the single most effective defense against concurrency regressions. A driver that ships with a correct LOCKING.md is a driver whose concurrency story can be audited by any reviewer without reading every line of the code.

For our `myfirst`, the LOCKING.md is short because the driver is small. For larger drivers, the LOCKING.md scales in proportion. The value scales too.

11.4.12 Wrapping Up Section 3

You now have, for your own driver, a field-by-field account of every piece of shared state, every critical section, and every protection mechanism. You have a LOCKING.md file that documents the locking strategy. You have source annotations that make the concurrency intent visible to any reader. You have seen the same procedure applied to a hypothetical larger driver to show that it scales.

The rest of the chapter introduces the primitives that let you add or relax protection as your design evolves. Section 4 looks at atomics: what they buy you, what they do not, and where they fit in your driver. Section 5 is the full treatment of mutexes, including the subtleties that Chapter 10 gestured at.

11.5 Section 4: Introduction to Atomic Operations

Atomic operations are one of the two tools this chapter introduces. They are useful when the shared state you need to protect is small and the operation you need to perform on it is simple. They do not replace mutexes for anything more complex, but they are often the right tool for the simpler cases, and they are always faster. This section builds the intuition for what atomics are, presents the `atomic(9)` API FreeBSD exposes, discusses memory ordering at a level appropriate for our purposes, and applies the `counter(9)` per-CPU counter API to the driver's statistics.

11.5.1 What “Atomic” Means at the Hardware Level

Recall the counter race from Section 2. Two threads loaded, incremented, and stored the same memory location. Because the load, increment, and store were three separate instructions, another thread could slip in between them. The fix, at the simplest level, is to use a single instruction that performs all three operations indivisibly: no other thread can observe memory midway through.

Modern CPUs provide such instructions. On amd64, the instruction that atomically adds a value to memory is `LOCK XADD`. The `LOCK` prefix tells the CPU to lock the relevant cache line for the duration of the instruction, so that no other CPU can touch the same line until the instruction completes. The `XADD` instruction itself performs an exchange-add: it adds its source to the destination and returns the destination's old value, all in one transaction. After `LOCK XADD`, the memory location has the correct sum, and no other CPU has been able to observe a partial update.

The C primitive that compiles to this instruction is `atomic_fetchadd_int`. FreeBSD's `atomic(9)` API exposes it, along with many siblings, in `/usr/src/sys/sys/atomic_common.h` and the per-architecture `atomic.h` headers.

Atomicity, then, is a property of a *memory operation*: if the operation completes, it does so in a single indivisible transaction from memory's perspective. No other CPU can observe a half-done version. This is a hardware-level guarantee, not a software abstraction. When you use an atomic primitive in C, you are telling the compiler to emit an instruction that the hardware itself guarantees is indivisible.

Three things matter about this guarantee.

First, it covers *one* operation. `atomic_fetchadd_int` does one add. Two successive atomic operations are not jointly atomic: another thread can observe memory between them. If you need two fields to be updated together, you still need a mutex.

Second, it covers *word-sized* operations, for some definition of word. On amd64, atomic primitives cover 8, 16, 32, and 64 bits. Operations on larger structures (arrays, composite structs) cannot be atomic at the hardware level; they require a mutex or a specialised mechanism like `atomic128_cmpset` on platforms that provide it. The `atomic(9)` API exposes the granularities the hardware supports.

Third, it is *cheap*. An atomic increment is typically a few cycles slower than a non-atomic one, because the cache line must be acquired exclusively. Compared to acquiring and releasing a mutex (which itself uses atomics internally and may also take scheduler turns under contention), an atomic operation is usually an order of magnitude faster. When the work you need to do is simple, atomics give you correctness at minimal cost.

11.5.2 The FreeBSD atomic(9) API

The kernel exposes a family of atomic operations through function-like macros. The full set is documented in `atomic(9)` (run `man 9 atomic`). For the purposes of this chapter we care about a small subset.

The four most common operations are:

```
void    atomic_add_int(volatile u_int *p, u_int v);
void    atomic_subtract_int(volatile u_int *p, u_int v);
u_int   atomic_fetchadd_int(volatile u_int *p, u_int v);
int     atomic_cmpset_int(volatile u_int *p, u_int expect, u_int new);
```

- `atomic_add_int(p, v)` atomically computes `*p += v`. The return value is not meaningful; the update is indivisible.
- `atomic_subtract_int(p, v)` atomically computes `*p -= v`. Same shape.
- `atomic_fetchadd_int(p, v)` atomically computes `*p += v` and returns the value that was in `*p` *before* the add. Useful when you want both to update and to observe the previous value in the same transaction.
- `atomic_cmpset_int(p, expect, new)` atomically performs a compare-and-swap: if `*p == expect`, write `new` to `*p` and return 1; otherwise leave `*p` alone and return 0. This is the fundamental primitive for building more complex lock-free data structures.

Each of these has variants for different integer widths (`_long`, `_ptr`, `_64`, `_32`, `_16`, `_8`) and for different memory-ordering requirements (`_acq`, `_rel`, `_acq_rel`). The width variants are obvious: they work on different integer types. The memory-ordering variants deserve their own subsection.

There are also read and store primitives:

```
u_int   atomic_load_acq_int(volatile u_int *p);
void    atomic_store_rel_int(volatile u_int *p, u_int v);
```

These are atomic loads and stores with specific memory-ordering guarantees. For aligned machine-word types on our platforms, ordinary `*p` accesses are atomic in the load/store sense, but the `atomic_load_acq_int` / `atomic_store_rel_int` forms also act as memory barriers: they prevent the compiler and the CPU from reordering surrounding loads and stores past them. We will see why that matters shortly.

Finally, a few specialised primitives:

```
void    atomic_set_int(volatile u_int *p, u_int v);
void    atomic_clear_int(volatile u_int *p, u_int v);
```

These are bitwise OR and AND-NOT respectively: `atomic_set_int(p, FLAG)` sets the `FLAG` bit in `*p`; `atomic_clear_int(p, FLAG)` clears it. They are useful for flag words where multiple threads may be setting and clearing different bits.

11.5.3 A Gentle Introduction to Memory Ordering

Here is a subtlety that bites beginners: even with atomic operations, the *order in which memory accesses become visible to other CPUs* is not always the order in which the code executed them. Modern CPUs and compilers

may reorder instructions for performance, as long as the reordering is invisible to the single thread that issued them. In a multi-threaded program, the reordering is sometimes visible, and it can matter.

The classic example is a producer that prepares a payload and then sets a ready flag, while a consumer spins on the flag and then reads the payload:

```
/* Thread A (producer): */
data = compute_payload();
atomic_store_rel_int(&ready_flag, 1);

/* Thread B (consumer): */
while (atomic_load_acq_int(&ready_flag) == 0)
    ;
use_payload(data);
```

For the pattern to work, the two stores on thread A's side must become visible to thread B in order: thread B must not see `ready_flag == 1` while still seeing the old data. Without memory barriers, the CPU or the compiler is free to reorder those stores, and the consumer would read stale data.

The `_rel` suffix on `atomic_store_rel_int` is a **release** barrier: every write that happened before the store, in program order, is made visible to other CPUs before the store itself is. The `_acq` suffix on `atomic_load_acq_int` is an **acquire** barrier: every read that happens after the load, in program order, sees values at least as fresh as the value the load saw. Paired, release on the publisher and acquire on the consumer ensure the consumer observes everything the publisher did before the release.

We will not build lock-free data structures in this chapter; the subsection “Memory Ordering on a Multi-Core Machine” later in Section 4 walks through the ordering guarantees in a little more depth, and real lock-free patterns belong to Part 6 (Chapter 28) when specific drivers need them. The important point for now is that the same `_rel / _acq` pattern is the reason `mtx_lock` and `mtx_unlock` compose correctly with memory: `mtx_lock` has acquire semantics (nothing inside the critical section leaks out above it), and `mtx_unlock` has release semantics (nothing inside the critical section leaks out below it).

For a beginner, the practical takeaway is this: use the atomic primitive that corresponds to the intent of your operation. A counter increment uses the plain form. A flag that publishes the completion of other work uses the `_rel / _acq` pair. When in doubt, use the stronger form (`_acq_rel`); it costs a little more but is correct in more situations.

11.5.4 When Atomics Suffice

Atomics are the right tool when all four of the following are true:

1. The operation you need is simple (a counter increment, a flag set, a pointer exchange).
2. The shared state is a single word-sized field (or a small number of independent ones).
3. The correctness of the code does not depend on more than one memory location being consistent with another (no composite invariants).
4. The operation is cheap enough that replacing a mutex with an atomic is an observable win.

For `myfirst`, the counters `bytes_read` and `bytes_written` meet all four criteria. They are simple increments, single fields, independent of each other, and frequently updated on the data path. Converting them from mutex-protected fields to atomic (or per-CPU) counters is a clean win.

The `cbuf`, on the other hand, does not meet the criteria. Its correctness depends on a composite invariant (`cb_used <= cb_size`, the bytes at `[cb_head, cb_head + cb_used)` are live) that spans multiple fields. No single atomic operation can preserve that invariant. The `cbuf` needs a mutex, and no amount of cleverness with atomics will change that. This is worth stating clearly because beginners sometimes try to “atomically update `cb_used` and `cb_head`” and end up with a driver that compiles, looks clever, and is still broken.

11.5.5 Per-CPU Counters: the `counter(9)` API

FreeBSD provides, for the specific case of “a counter that is incremented frequently and read infrequently”, a specialised API: `counter(9)`. A `counter_u64_t` is a per-CPU counter, where each CPU has its own private memory for the counter, and a read combines all of them.

The API is:

```
counter_u64_t  counter_u64_alloc(int flags);
void          counter_u64_free(counter_u64_t c);
void          counter_u64_add(counter_u64_t c, int64_t v);
uint64_t      counter_u64_fetch(counter_u64_t c);
void          counter_u64_zero(counter_u64_t c);
```

`counter_u64_alloc(M_WAITOK)` returns a handle to a new per-CPU counter. `counter_u64_add(c, 1)` atomically adds 1 to the calling CPU’s private copy (no cross-CPU synchronization is required). `counter_u64_fetch(c)` sums across all CPUs and returns the total. `counter_u64_free(c)` releases the counter.

The per-CPU design has two consequences. First, adds are very fast: they touch only the calling CPU’s cache line, so there is no cross-CPU contention. Even on a 32-core system, a `counter_u64_add` call does not pay the cost of synchronizing with other cores. Second, reads are expensive: `counter_u64_fetch` sums across all CPUs, which costs roughly one cache-miss per CPU. Reads are therefore infrequent; updates are frequent.

This shape is exactly right for the `bytes_read` and `bytes_written` counters. They are updated on every I/O call (high rate). They are read only when a user runs `sysctl` or when `detach` emits a final log line (low rate). Migrating them to `counter_u64_t` gives us both correctness across 32-bit and 64-bit architectures, and scalability across many CPUs.

11.5.6 Migrating the Driver’s Counters

Here is what the migration looks like. First, change the fields in `struct myfirst_softc`:

```
struct myfirst_softc {
    /* ... */
    counter_u64_t  bytes_read;
```

```
        counter_u64_t    bytes_written;
        /* ... */
};
```

Update `myfirst_attach` to allocate them:

```
static int
myfirst_attach(device_t dev)
{
    /* ... existing setup ... */
    sc->bytes_read = counter_u64_alloc(M_WAITOK);
    sc->bytes_written = counter_u64_alloc(M_WAITOK);
    /* ... rest of attach ... */
}
```

Update `myfirst_detach` to free them:

```
static int
myfirst_detach(device_t dev)
{
    /* ... existing teardown ... */
    counter_u64_free(sc->bytes_read);
    counter_u64_free(sc->bytes_written);
    /* ... rest of detach ... */
}
```

Update the read and write handlers to use `counter_u64_add` instead of the plain increment:

```
counter_u64_add(sc->bytes_read, got);
```

Update the sysctl handlers to use `counter_u64_fetch`:

```
static int
myfirst_sysctl_bytes_read(SYSCTL_HANDLER_ARGS)
{
    struct myfirst_softc *sc = arg1;
    uint64_t v = counter_u64_fetch(sc->bytes_read);
    return (sysctl_handle_64(oidp, &v, 0, req));
}
```

And register the sysctl with a handler instead of a direct pointer:

```

SYSCTL_ADD_PROC(&sc->sysctl_ctx, SYSCTL_CHILDREN(sc->sysctl_tree),
    OID_AUTO, "bytes_read",
    CTLTYPE_U64 | CTLFLAG_RD | CTLFLAG_MPSAFE,
    sc, 0, myfirst_sysctl_bytes_read, "QU",
    "Total bytes drained from the FIFO");

```

Once the migration is done, you can drop the `sc->bytes_read` and `sc->bytes_written` entries from the list of fields protected by `sc->mtx` in your `LOCKING.md`. They now protect themselves, in a way that scales better and is correct on any architecture FreeBSD supports.

11.5.7 A Simple Atomic Flag Example

Let us also look at a smaller, more targeted use of atomics. Suppose we want to track whether the driver is “currently busy” in a coarse sense, for debugging. We do not need strict correctness; we just want a flag we can flip and read.

Add a field:

```
volatile u_int busy_flag;
```

Update it in the handlers:

```

atomic_set_int(&sc->busy_flag, 1);
/* ... do the work ... */
atomic_clear_int(&sc->busy_flag, 1);

```

Read it elsewhere:

```
u_int is_busy = atomic_load_acq_int(&sc->busy_flag);
```

This is *not* a correct way to implement mutual exclusion. Two threads could both read `busy_flag == 0`, both set it to 1, both do their work, both clear it. The flag does not prevent concurrent execution; it is purely informational. For actual mutual exclusion, we need a mutex, which is the subject of the next section.

The point of the flag example is narrower: atomic operations let you set and read a single-word value without tearing. That is useful for an informational field that does not participate in an invariant. Once you need the field to participate in an invariant, atomics alone will not save you.

11.5.8 The Compare-and-Swap Pattern

The most interesting atomic primitive is `atomic_cmpset_int`, which implements the **compare-and-swap** pattern. It lets you write optimistic code: “update the field if nobody else has touched it since I last looked.”

The pattern is:

```
u_int old, new;
do {
    old = atomic_load_acq_int(&sc->field);
    new = compute_new(old);
} while (!atomic_cmpset_int(&sc->field, old, new));
```

The loop reads the current value, computes what the new value should be based on it, and tries to atomically swap the new value in, conditional on the field not having changed. If another thread changed the field between the load and the cmpset, the cmpset fails and the loop retries with the new value. When it succeeds, the update is committed atomically, and `*sc->field` now holds the value derived from the state it actually held at the moment of update.

Compare-and-swap is the building block for most lock-free data structures. You can implement a lock-free stack, a lock-free queue, a lock-free counter (which is how `atomic_fetchadd` is often implemented under the hood), and many other structures using nothing but compare-and-swap.

For our purposes, compare-and-swap is worth knowing about even if we do not use it heavily. When you read FreeBSD source later and see `atomic_cmpset_*` in a tight loop, you will recognize the pattern immediately: optimistic retry.

11.5.9 When Atomics Are Not Enough

Three situations call for a mutex even when the state looks simple.

First, when the operation involves more than one memory location. If `cb_head` and `cb_used` must be updated together, no single atomic operation can do it. Either we accept that the update is two atomics (with a transient half-updated state visible to readers), or we hold a mutex across the update.

Second, when the operation is expensive (includes a function call, a loop, or allocates memory). An atomic operation is cheap only if it is a single fast instruction. A long critical section cannot be folded into a single atomic; it needs a mutex.

Third, when the code must block. No atomic operation can put the caller to sleep. If you need to wait for a condition (“buffer has data”), you need `mtx_sleep` or a condition variable, which require a mutex.

The `cbuf` triggers all three. The counters trigger none. That is why the counters can migrate to atomics or `counter(9)` and the `cbuf` cannot.

11.5.10 A Worked Example: a “driver version” Counter

Here is an exercise to ground the theory. Suppose we want to count how many times the driver has been loaded and unloaded during the current uptime. This is module-level state, not per-softc, and it is updated exactly twice per load/unload cycle: once in the module event handler when the load is successful, and once when it is unloaded.

We could protect it with a global mutex. That would be overkill: the update is one integer, happens rarely, and does not compose with anything else. An atomic is the right choice:

```
static volatile u_int myfirst_generation = 0;

static int
myfirst_modevent(module_t m, int event, void *arg)
{
    switch (event) {
        case MOD_LOAD:
            atomic_add_int(&myfirst_generation, 1);
            return (0);
        case MOD_UNLOAD:
            return (0);
        default:
            return (EOPNOTSUPP);
    }
}
```

A reader (for example a sysctl exposing `hw.myfirst.generation`) can use:

```
u_int gen = atomic_load_acq_int(&myfirst_generation);
```

No lock required. No contention. Correct on every architecture FreeBSD supports. This is the atomic-appropriate case: one field, one operation, no composite invariant.

11.5.11 Visibility, Ordering, and the Everyday Driver

Before closing this section, one more observation about memory ordering and mutexes.

You might wonder, given that `atomic_store_rel_*` and `atomic_load_acq_*` exist, why the driver code we wrote in Chapter 10 didn't use them. The reason is that `mtx_lock` has acquire semantics built in, and `mtx_unlock` has release semantics built in. Every write inside a critical section becomes visible to the next thread that acquires the mutex, at the moment it acquires. Every read inside a critical section sees writes that happened before any previous `mtx_unlock` on the same mutex. The mutex is, among other things, a memory barrier.

So when you write:

```
mtx_lock(&sc->mtx);
sc->field = new_value;
mtx_unlock(&sc->mtx);
```

you do not need to say `atomic_store_rel_int(&sc->field, new_value)`. The mutex already does the necessary ordering. This is an important property: it means that code inside a mutex critical section does not need to reason about memory ordering. Correctness is about the mutual exclusion, full stop.

Outside of critical sections (and only outside), you need to reason about ordering yourself. That is where the `_acq` / `_rel` atomic variants earn their keep.

For the `myfirst` driver, the only state we access outside any lock is `sc->is_attached` (the optimization at handler entry) and the per-CPU counters (which handle their own ordering). Everything else is either immutable after attach or protected by `sc->mtx`. That is a small footprint, and it is the reason our concurrency story stays manageable.

11.5.12 Memory Ordering on a Multi-Core Machine

The earlier “Gentle Introduction to Memory Ordering” sketched what `_acq` and `_rel` do; this section makes the underlying mechanism a little more concrete, because once you have done one publishing pattern by hand, the rest of the kernel’s atomic primitives become much easier to read.

The example is the same producer/consumer pattern with two shared variables, `payload` and `ready`. Without barriers, two orderings can fail:

- **On the producer’s side**, the write to `payload` could become visible to other CPUs *after* the write to `ready`. The consumer then sees `ready == 1` while still seeing the old `payload`.
- **On the consumer’s side**, the read of `payload` could be hoisted *before* the read of `ready`. The consumer commits to the old `payload` before it ever inspects the flag.

Why is either reordering legal? The compiler is allowed to move loads and stores past each other as long as the result is consistent for a single-thread observer. The CPU is allowed to reorder accesses through its store buffer, its prefetch unit, and its out-of-order execution engine, again as long as the local thread cannot tell. Multi-thread visibility is a property neither the compiler nor the CPU enforces by default.

The `_rel` and `_acq` suffixes lift exactly the constraints you need:

- `atomic_store_rel_int(&ready, 1)`: every store that appeared before this line in program order is made visible before the store to `ready` becomes visible. The *release* publishes everything prior.
- `atomic_load_acq_int(&ready)`: every load that appears after this line sees memory at least as fresh as the load itself saw. The *acquire* fences subsequent reads.

The pairing of release and acquire creates a *happens-before* relationship: if the consumer observes `ready == 1` through `atomic_load_acq_int`, every store the producer made before its `atomic_store_rel_int` is visible to the consumer.

In your driver, the same property is supplied by the mutex. `mtx_unlock(&sc->mtx)` has release semantics built in. `mtx_lock(&sc->mtx)` has acquire semantics. Every write made while the mutex was held is visible to the next thread that acquires the same mutex. This is why, inside a critical section, plain C assignment is fine: the mutex boundaries handle ordering for you.

Outside any lock, if you are coordinating multiple fields with atomics, you have to think explicitly about which writes need to be visible together and reach for the matching `_rel` and `_acq` variants. In the Chapter 11 driver, the only place this matters is the per-CPU counters, and `counter(9)` handles the ordering internally.

11.5.13 Atomic Swaps and More Flavours

We have mentioned `atomic_add`, `atomic_fetchadd`, `atomic_cmpset`, `atomic_load_acq`, `atomic_store_rel`, `atomic_set`, and `atomic_clear`. Two more are worth naming, because you will see them in FreeBSD source.

`atomic_swap_int(p, v)`: atomically exchanges the value at `*p` with `v` and returns the old value. This is useful when you want to “claim” a resource: the thread that successfully swaps the flag from “free” to “taken” is the new owner.

`atomic_readandclear_int(p)`: atomically reads `*p` and sets it to zero, returning the old value. Useful for “drain” patterns where one thread periodically collects and resets a counter the other threads have been incrementing.

Both are built on top of the same compare-and-swap primitive the hardware provides, and both have the same cost profile as `atomic_cmpset`.

11.5.14 A Practical Example: Atomic Reference Counting

A common driver pattern is reference counting: an object is allocated, several threads take references to it, and the object is freed when the last reference is dropped. Atomics are the right tool.

```
struct myobj {
    volatile u_int refcount;
    /* ... other fields ... */
};

static void
myobj_ref(struct myobj *obj)
{
    atomic_add_int(&obj->refcount, 1);
}

static void
myobj_release(struct myobj *obj)
{
    u_int old = atomic_fetchadd_int(&obj->refcount, -1);
    KASSERT(old > 0, ("refcount went negative"));
    if (old == 1) {
        /* We were the last reference; free it. */
        free(obj, M_DEVBUF);
    }
}
```

Two details are worth noting.

The first is the use of `atomic_fetchadd_int` for the decrement. Why not `atomic_subtract_int`? Because we need to know whether *our* decrement was the one that brought the count to zero, so that we can

free the object. `atomic_fetchadd_int` returns the value before the add, which tells us exactly that. An `atomic_subtract_int` does not return a value.

The second is that the free must happen *only if* our decrement was the final one. Otherwise, if two threads happen to release concurrently, they might both try to free the object. By keying the free on “old == 1”, we ensure that exactly one thread (the one that took the count from 1 to 0) frees the object. All other threads simply decrement and return.

This pattern is used throughout the FreeBSD kernel. It is the reason `refcount(9)` exists as a small wrapper API (`refcount_init`, `refcount_acquire`, `refcount_release`). For our purposes we do not use reference counting yet; the example is for your toolbox.

11.5.15 counter(9) in More Depth

The `counter(9)` API is used throughout the FreeBSD kernel for high-frequency counters. It is worth understanding how it works, because the design choices explain the API shape.

Each `counter_u64_t` is, internally, an array of per-CPU counters. The array is sized to the number of CPUs in the system. An add targets the calling CPU’s slot. A fetch iterates over all slots and sums.

The per-CPU layout means:

- **Adds do not contend.** Two CPUs adding concurrently touch different cache lines. There is no cross-CPU traffic.
- **Fetches are expensive.** They touch every CPU’s cache line, which on a 32-core machine means 32 cache misses.
- **Fetches are not atomic with adds.** A fetch might catch one CPU mid-update. The returned total may be slightly off from any instantaneous true total. This is fine for informational counters; it would be wrong for counters that feed into decisions.

For our driver, the fetches happen in `sysctl` handlers, which are infrequent, so the asymmetry is in our favour. If we ever needed a counter whose value must be exact at every fetch, `counter(9)` would not be the right primitive.

A subtlety: `counter_u64_zero` resets the counter to zero. Doing so while adds are happening is *not* atomic with them. If a reader fetches, sees a large value, and zeros, some adds in flight may be lost. For counters that are simply informational, this is acceptable. For counters that track budget or quota, zero carefully or not at all.

11.5.16 Lock-Free Data Structures in Context

A full treatment of lock-free data structures would fill its own book. FreeBSD’s kernel uses them in specific places where contention would otherwise be a bottleneck:

- **`buf_ring(9)`** is a lock-free multi-producer queue used by network and storage drivers for hot paths. The driver picks single-consumer or multi-consumer mode through the flags passed to `buf_ring_alloc()`, so the same primitive serves either shape depending on the subsystem’s concurrency needs. For the full API surface, see Appendix B’s `buf_ring(9)` entry.

- **epoch(9)** provides a read-mostly pattern where readers proceed without any synchronization and writers coordinate among themselves.
- **mi_switch fast paths** in the scheduler use atomic operations to avoid mutexes entirely in the common case.

Reading any of these is good study material, but they are specialised. For a beginner driver, the combination of atomic for single fields and mutex for composite state covers 99% of real drivers. We stay with those for Chapter 11 and let Chapter 12 and later chapters introduce the more sophisticated patterns when specific drivers need them.

11.5.17 When to Reach for an Atomic (Decision Guide)

As a decision guide, use this flowchart when deciding whether to make a field atomic or lock it:

1. Is the field a single word (8, 16, 32, or 64 bits on a platform where that size is atomic)? If no, it must be locked.
2. Does the correctness of my code depend on this field being consistent with another field? If yes, it must be locked with the other field.
3. Does my operation on the field involve more than one read-modify-write sequence? If yes, either use `atomic_cmpset` in a loop or lock.
4. Does the code need to sleep while holding the state? If yes, use a mutex and `mtx_sleep`; atomics cannot sleep.
5. Otherwise, an atomic operation is probably the right tool.

For the `myfirst` driver, this flowchart points at atomics for the byte counters (now `counter(9)`), the “generation” counter (Section 4’s load/unload counter), and informational flags. It points at the mutex for everything involving the `cbuf`.

11.5.18 Wrapping Up Section 4

Atomic operations are a precise tool for a precise job: updating a single word-sized field without racing. They are fast, they compose with memory ordering primitives when needed, and they scale well. They do not replace mutexes for anything more complicated than a single field.

In the driver, we migrated the byte counters to `counter(9)` per-CPU counters, which removed them from the set of fields protected by `sc->mtx` and gave us better performance under high load. The `cbuf`, with its composite invariants, stays protected by `sc->mtx`. Section 5 returns to that mutex and explains, in full, what it is and how it works.

11.6 Section 5: Introducing Locks (Mutexes)

This is the central section of the chapter. Everything up to here has been preparation; everything after this reinforces and extends. By the end of the section, you will understand what a mutex is, what kinds FreeBSD provides, what rules apply to each, why Chapter 10 chose the specific kind it used, and how to verify that your driver obeys the rules.

11.6.1 What a Mutex Is

A **mutex** (short for “mutual exclusion”) is a synchronization primitive that allows only one thread at a time to hold it. Threads that try to acquire a mutex while it is held wait until the holder releases it, then one of the waiters acquires it. The guarantee is that, between any two acquisitions of the same mutex, exactly one thread executed the code region between its acquire and its release.

That guarantee is what turns sequences of instructions into **critical sections**: regions of code whose execution is serialized across all threads that use the mutex. A reader who holds the mutex can read any number of fields and trust that no concurrent writer is producing torn values. A writer who holds the mutex can update any number of fields and trust that no concurrent reader is observing a half-updated state.

Mutexes do not prevent concurrency; they shape it. Multiple threads can execute outside the critical section at the same time. They can even be queued up to enter the critical section at the same time. What they cannot do is execute the critical section concurrently. The serialization is the property we buy; the price is the cost of acquiring and releasing, plus any wait time.

11.6.2 FreeBSD’s Two Mutex Shapes

FreeBSD distinguishes two fundamental shapes of mutex, because the kernel itself runs in two distinct sets of contexts and each context has different constraints.

MTX_DEF is the default mutex, often called a **sleep mutex**. When a thread tries to acquire one that is held, it is put to sleep (added to a sleep queue, with its state set to `TDS_SLEEPING`) until the holder releases the mutex. A sleep mutex may block for an arbitrarily long time, so the code using it must be in a context where sleeping is legal: ordinary process context, a kernel thread, or a callout marked as `mpsafe`. Most driver code runs in contexts where sleep mutexes are appropriate.

MTX_SPIN is a **spin mutex**. When a thread tries to acquire one that is held, it *spins*: it executes a tight loop of atomic checks until the holder releases. A spin mutex never sleeps. The reason spin mutexes exist is that some contexts (specifically, hardware interrupt handlers on certain systems) cannot sleep at all. A thread that cannot sleep cannot wait for a sleep mutex; it needs a primitive that makes progress without descheduling. Spin mutexes have additional rules: they disable interrupts on the CPU that acquires them, they must be held for very short durations, and code holding them cannot call any function that could sleep.

For the `myfirst` driver, `MTX_DEF` is the correct choice. Our handlers run in process context, on behalf of a user syscall. They do not run in interrupt context. Sleeping is legal. The mutex can be a sleep mutex, which is simpler and places fewer restrictions on what the critical section can do.

The `mtx_init` call from Chapter 10 specifies `MTX_DEF`:

```
mtx_init(&sc->mtx, device_get_nameunit(dev), "myfirst", MTX_DEF);
```

That last argument is the lock type. If we had wanted a spin mutex, it would be `MTX_SPIN`, and the rest of the driver would have been more constrained.

11.6.3 The Life Cycle of a Mutex

Every mutex has a life cycle: create, use, destroy. The functions are `mtx_init` and `mtx_destroy`:

```
void mtx_init(struct mtx *m, const char *name, const char *type, int opts);
void mtx_destroy(struct mtx *m);
```

`mtx_init` initializes the internal fields of the mutex structure and registers the mutex with `WITNESS` (if enabled) so that lock-order checking can apply. `mtx_destroy` tears it down.

The four arguments to `mtx_init` are:

- `m`: the mutex structure (typically a field of a larger struct, not a separate allocation).
- `name`: a human-readable identifier that shows up in `ps`, `dmesg`, and `WITNESS` messages. For our driver this is `device_get_nameunit(dev)` (for instance, `"myfirst0"`).
- `type`: a short string classifying the mutex; `WITNESS` groups mutexes with the same `type` string as related. For our driver this is `"myfirst"`.
- `opts`: the flags, including `MTX_DEF` or `MTX_SPIN`, and optional flags like `MTX_RECURSE` (allow the same thread to acquire the mutex multiple times) or `MTX_NEW` (guarantee that the memory is uninitialized). For our driver, `MTX_DEF` alone is correct.

The name and type matter for observability. When a user runs `procstat -kk`, they see `myfrd` (the wait message from `mtx_sleep`) and `myfirst0` (the mutex name) in the process's wait information. When `WITNESS` flags a lock-order reversal, it names the mutex by its name. Good names make debugging concurrency problems dramatically easier.

11.6.4 Acquiring and Releasing

The two fundamental operations are `mtx_lock` and `mtx_unlock`. Both take a pointer to the mutex:

```
void mtx_lock(struct mtx *m);
void mtx_unlock(struct mtx *m);
```

`mtx_lock` acquires the mutex. If the mutex is not held, it acquires immediately (the cost is one atomic operation). If the mutex is held, the calling thread goes to sleep on the mutex's wait list. When the current holder releases, one waiter is woken and acquires the mutex.

`mtx_unlock` releases the mutex. If there are waiters, one is woken and scheduled. If there are no waiters, the unlock completes with only the cost of a store.

These two operations compose into the critical-section idiom:

```
mtx_lock(&sc->mtx);
/* critical section: mutual exclusion is guaranteed here */
mtx_unlock(&sc->mtx);
```

Inside the critical section, the thread holding the mutex is the only thread that can be in any critical section protected by the same mutex. Other threads trying to enter the critical section are asleep, waiting.

11.6.5 The Other Operations

Several other operations are occasionally useful:

mtx_trylock(&m): attempts to acquire the mutex. Returns nonzero on success (mutex acquired) and zero on failure (mutex held by another thread). The calling thread never sleeps. This is useful when you want to do something only if the mutex is available, or when you want to avoid holding a lock across a potentially long operation.

mtx_assert(&m, what): asserts that the mutex is or is not held, for debugging. The *what* argument is one of `MA_OWNED` (current thread holds the mutex), `MA_NOTOWNED` (current thread does not hold the mutex), `MA_OWNED | MA_RECURSED` (held, recursively), or `MA_OWNED | MA_NOTRECURSED` (held, not recursively). On an `INVARIANTS` kernel, the assertion fires as a panic if violated. On a non-`INVARIANTS` kernel, it does nothing. Use this liberally; it is free in production and catches bugs in development.

mtx_sleep(&chan, &mtx, pri, wmesg, timo): the sleep primitive we used in Chapter 10. Atomically releases `mtx`, sleeps on `chan`, reacquires `mtx` before returning. The atomicity matters: the lock drop and the sleep cannot be interrupted by a concurrent wakeup.

mtx_initialized(&m): returns nonzero if the mutex has been initialized. Useful in rare teardown paths where you want to check whether you need to call `mtx_destroy`.

11.6.6 Re-Examining the Chapter 10 Design

With the vocabulary in place, let us re-read the Chapter 10 design choices and confirm they are correct.

One mutex, covering all shared softc state. The simplest possible design is one mutex protecting everything that is shared. This is what we have. It is the right starting point for any driver. It is almost never the wrong answer, even if a finer-grained design might perform better. Finer-grained locking has costs (lock ordering, understanding, bugs when new fields are added); a single mutex has the benefit of being obviously correct.

MTX_DEF (sleep mutex). The driver runs in process context. All the work is on behalf of user syscalls. There are no interrupt handlers. `MTX_DEF` is the correct choice.

Lock acquired for every access to shared state. Every read and write of the protected fields is inside a `mtx_lock` / `mtx_unlock` pair. The audit in Section 3 confirmed this.

Lock released before calling functions that may sleep. `uiomove(9)` may sleep (on a user-space page fault). `selwakeup(9)` may take its own locks and should not be called under our mutex. Our handlers drop the mutex before these calls. This is the rule from Chapter 10; we can now explain *why* it matters, which Section 5.6 does below.

mtx_sleep uses the mutex as interlock. The atomic drop-and-sleep operation is what prevents a race between the condition check and the sleep. If we did not use `mtx_sleep` and instead unlocked, then slept, a concurrent wakeup could fire in the window between unlock and sleep, and we would miss the wakeup and sleep forever. `mtx_sleep` exists precisely to close that window.

Every choice Chapter 10 made is now one we can defend from first principles. The mutex is not there by habit; it is there because every aspect of the design requires it.

11.6.7 The Sleep-with-Mutex Rule

One rule recurs often enough to deserve its own treatment: **do not hold a non-sleepable lock across a sleeping operation**. For MTX_DEF mutexes, this translates to: do not hold the mutex across any call that may sleep unless you are using `mtx_sleep` itself (which atomically drops the mutex for the duration of the sleep).

Why does this matter?

First, sleeping with a mutex held blocks any other thread that needs the same mutex, for the entire duration of the sleep. If the sleep is long, throughput collapses; if the sleep is indefinite, the system deadlocks.

Second, sleeping in the kernel involves the scheduler, which may need its own locks. If those locks have a defined order with respect to your mutex, and your held mutex is further up the order, you may trigger a lock-order violation.

Third, on WITNESS-enabled kernels, sleeping with a mutex held raises a warning. On INVARIANTS-enabled kernels, certain specific cases of this rule (the well-known sleeping primitives) will panic.

The rule's scope is broader than it first appears. A call that "may sleep" includes:

- `malloc(9)` with `M_WAITOK`.
- `uiomove(9)`, `copyin(9)`, `copyout(9)` (each may fault on user memory and wait for the page to be paged in).
- Most functions in the `vfs(9)` layer.
- Most functions in the `file(9)` layer.
- `taskqueue_enqueue(9)` in some paths.
- Any function whose implementation chain includes any of the above.

The practical technique is to identify every function you call from inside a critical section and ask: "can this sleep?" If the answer is yes or you are not sure, drop the lock before the call. The Chapter 10 handlers follow this rule: they drop `sc->mtx` before every `uiomove`, and they drop it before every `selwakeup`.

11.6.8 Priority Inversion and Priority Propagation

A subtle concurrency issue is **priority inversion**. Suppose thread L (low priority) acquires mutex M. Thread H (high priority) wants to acquire M and has to wait for L. In the meantime, thread M (medium priority, no relation to our mutex variable) is doing work unrelated to the mutex. The scheduler, seeing that H is blocked and M is runnable, will schedule M over L. L therefore makes no progress. M keeps L from running. H keeps waiting for L. A medium-priority thread has effectively blocked a high-priority thread, even though they share no resources.

This is priority inversion. It is a famous bug; the Mars Pathfinder mission briefly experienced a version of it in the 1990s.

The FreeBSD kernel handles priority inversion through **priority propagation** (also called priority inheritance). When a high-priority thread blocks on a mutex held by a lower-priority thread, the kernel temporarily raises the holder's priority to match the waiter's. L now runs at H's priority, so M cannot preempt it, and L finishes the critical section quickly. When L releases the mutex, its priority drops back, H acquires the mutex, and the inversion is resolved.

The practical consequence for a driver writer is that you mostly do not need to think about priority inversion. The kernel handles it for you on any `MTX_DEF` mutex. But the mechanism has costs: the holder of a contended mutex may run at a higher priority than it would otherwise, potentially for the duration of the critical section. That is another reason to keep critical sections short.

11.6.9 Lock Order and Deadlocks

Priority inversion is a problem within a single mutex. **Deadlock** is a problem across multiple mutexes.

Consider two mutexes, A and B. Thread 1 acquires A, then wants to acquire B. Thread 2 acquires B, then wants to acquire A. Each thread holds what the other wants. Neither releases what it has until it acquires what it wants. Neither can make progress. The system has deadlocked.

The classical defence against deadlock is **lock ordering**: every thread acquires its locks in the same global order. If all threads that need both A and B always acquire A before B, deadlock of this kind is impossible. Thread 2 would acquire A before B, not the other way around; if it could not get A, it would wait for it; once it had A, it would then acquire B; it would not hold B and wait for A.

In FreeBSD, `WITNESS` enforces lock ordering. The first time the kernel observes a thread holding lock A and acquiring lock B, it records the order A-before-B as valid. If it later sees another thread (or even the same thread) hold B and try to acquire A, that is a lock-order reversal, and `WITNESS` prints a warning. If `INVARIANTS` is also enabled and the configuration demands it, the warning becomes a panic.

For the `myfirst` driver, we have only one mutex. There is no lock order to worry about (a mutex has a trivial order with itself: you either hold it or you don't, and `mtx_lock` with `MTX_RECURSE` unset will panic if a thread already holding the mutex tries to acquire it again). As the driver grows, if additional mutexes are introduced, a lock order must be defined and documented. Chapter 12 covers this in depth for the case of multiple lock classes (for example, a mutex for the control path and an `sx` lock for the data path).

11.6.10 WITNESS: What It Catches

`WITNESS` is the kernel's lock-order and locking-discipline checker. It is enabled by options `WITNESS` in the kernel configuration; it is often paired with options `INVARIANTS` for maximum coverage.

What `WITNESS` catches:

- **Lock-order reversals**: thread acquires locks in an order not matching a previously-observed order.
- **Sleeping with a non-sleepable lock held**: thread calls `msleep`, `tsleep`, `cv_wait`, or any other sleeping primitive while holding an `MTX_SPIN` mutex.
- **Recursive acquisition of a non-recursive lock**: thread tries to acquire a mutex it already holds, and the mutex was not initialized with `MTX_RECURSE`.
- **Release of a lock not held**: thread calls `mtx_unlock` on a mutex it does not hold.
- **Sleeping with certain named sleep mutexes held**: more specifically, if `INVARIANTS` is also enabled, `mtx_assert(MA_NOTOWNED)` is checked before sleeping.

What `WITNESS` does not catch:

- **Missing locks**: if you forgot to take a lock, `WITNESS` has no way to know you should have.

- **Use-after-free of a mutex:** if you destroy a mutex and then use it, WITNESS may or may not catch it depending on how quickly the memory is reused.
- **Data races that do not involve locks:** two threads touching the same unlocked variable are invisible to WITNESS.

For this reason, WITNESS is a lint tool, not a proof. A driver can pass WITNESS and still be wrong. But a driver that fails WITNESS is almost certainly wrong, and the warnings are usually specific enough to point at the line.

Turn on WITNESS and INVARIANTS in your development kernel. Run the driver's test suite under the debug kernel. If warnings appear, investigate every one. This is the single most effective habit in driver debugging.

11.6.11 Running the Driver Under WITNESS

If you have not already, build a debug kernel. On a FreeBSD 14.3 release system, the installed GENERIC kernel does not have WITNESS or INVARIANTS enabled, because those options carry a runtime cost that is undesirable in production. You have to build a kernel that does. The simplest way is to copy GENERIC and add the relevant options. The lines you need are:

```
options      INVARIANTS
options      INVARIANT_SUPPORT
options      WITNESS
options      WITNESS_SKIPSPIN
```

Section “Building and Booting a Debug Kernel” later in this chapter walks through the full build, install, and reboot procedure step by step. The short version is:

```
# cd /usr/src
# make buildkernel KERNCONF=MYFIRSTDEBUG
# make installkernel KERNCONF=MYFIRSTDEBUG
# shutdown -r now
```

where MYFIRSTDEBUG is the name of the kernel configuration file you created.

After rebooting, load the driver and run a test. If WITNESS fires, dmesg will show something like:

```
lock order reversal:
 1st 0xfffffe00020b8a30 myfirst0 (myfirst, sleep mutex) @ ...:<line>
 2nd 0xfffffe00020b8a38 foo_lock (foo, sleep mutex) @ ...:<line>
lock order foo -> myfirst established at ...
```

The warning names both mutexes, their addresses, their types, and the source locations involved. From here, you trace the code and fix the order.

For the Chapter 10 Stage 4 driver, WITNESS should be silent: we have one mutex, used consistently, and no sleep-with-mutex violations. If you see warnings, that is a bug worth investigating.

11.6.12 When a Mutex Is the Wrong Answer

Three situations call for something other than a plain `MTX_DEF` mutex. Chapter 12 covers each in depth; we mention them here so the reader knows the landscape.

Many readers, few writers. If the critical section is mostly read-only, with occasional writes, a mutex serializes readers unnecessarily. A reader/writer lock (`sx(9)` or `rw(9)`) allows many readers to hold the lock simultaneously and serializes only writers. The cost is higher overhead per acquire/release and more complexity in the rules; the benefit is scalability on read-heavy workloads.

Blocking wait on a condition, not on a lock. When a thread needs to wait until a specific condition becomes true (for example, “data is available in the `cbuf`”), `mtx_sleep` with a channel is one way to express it. A named **condition variable** (`cv(9)`) is another way, often cleaner and more explicit. Chapter 12 covers `cv_wait`, `cv_signal`, and `cv_broadcast`.

Extremely short operations on a single field. Atomics, as we saw in Section 4. No need for a mutex at all.

For `myfirst`, none of these apply yet. The critical sections are short, the operations involve composite invariants, and neither read-heavy nor condition-variable patterns fit better than the mutex we have. Chapter 12 will introduce the alternatives when the driver evolves to a point where they are justified.

11.6.13 A Mini-Walkthrough: Tracing a Lock

To make the mechanics concrete, let us trace what happens when two threads race for the mutex.

Thread A calls `myfirst_read`. It reaches `mtx_lock(&sc->mtx)`. The mutex is not held (the initial state is “owner = NULL, waiters = none”). `mtx_lock` executes an atomic compare-and-swap: “if owner is NULL, set owner to `curthread`.” It succeeds. A is now in the critical section.

Thread B calls `myfirst_read` on a different CPU. It reaches `mtx_lock(&sc->mtx)`. The mutex is held (owner = A). The compare-and-swap fails. `mtx_lock` now has to arrange for B to wait.

B enters the slow path. It adds itself to the mutex’s wait list. It sets its thread state to blocked. It calls the scheduler, which picks some other runnable thread (possibly none, in which case the CPU idles).

Time passes. A finishes its critical section and calls `mtx_unlock(&sc->mtx)`. `mtx_unlock` sees that there are waiters. It picks one (usually the highest-priority waiter, with FIFO among equal priorities) and wakes it. That waiter, probably B, is made runnable.

The scheduler sees B runnable and schedules it. B resumes inside `mtx_lock`’s slow path. `mtx_lock` now records that the mutex is held by B and returns. B is in the critical section.

Between A’s `mtx_unlock` and B’s `mtx_lock` returning, the mutex was unowned for a brief moment. No other thread could have slipped in, because the unlock and the wakeup are arranged so that whoever wakes up next is the next owner. This is one of the things `mtx_lock` does that a hand-rolled “check a flag, sleep, check again” implementation would not get right.

All of this is happening inside `/usr/src/sys/kern/kern_mutex.c`. If you open that file and look for `__mtx_lock_sleep`, you can see the slow path’s code. It is more elaborate than the sketch above; it handles priority propagation, adaptive spinning, and several corner cases. The core idea, though, is what the sketch describes.

11.6.14 Reading a Lock Contention Story

When a driver performs poorly under load, one of the first things to check is whether the mutex is contended: how often threads have to wait for it and for how long. FreeBSD provides this information through the `debug.lock.prof.*` `sysctl` tree, which is enabled by the `LOCK_PROFILING` kernel option, and through the `lockstat(1)` user-space tool, which is part of the DTrace toolkit.

We will not build a full performance-analysis story in this chapter; that is Chapter 12 and Part 4 material. But if you are curious, on a kernel built with options `LOCK_PROFILING`, try:

```
# sysctl debug.lock.prof.enable=1
# ./producer_consumer
# sysctl debug.lock.prof.enable=0
# sysctl debug.lock.prof.stats
```

The output lists every lock the kernel has seen, the maximum and total wait time, the maximum and total hold time, the acquisition count, and the source file and line number where the lock was last touched. For our driver, `myfirst0` should appear with modest numbers, because the critical sections are short. If the numbers were large, we would have a signal that the mutex is contended and a finer-grained design might help. For Chapter 11's purposes, we are not optimizing; we are ensuring correctness.

11.6.15 Applying What We Have Learned

Let us consolidate. In the driver as of Chapter 10 Stage 4, the mutex `sc->mtx` is:

- An `MTX_DEF` sleep mutex, created in `myfirst_attach` and destroyed in `myfirst_detach`.
- Named "`myfirst0`" (and similar for other unit numbers), typed "`myfirst`".
- Held by the I/O handlers around every access to the `cbuf`, the byte counters, and the other protected fields.
- Released before every call that may sleep (`uiomove`, `selwakeup`).
- Used as the interlock argument to `mtx_sleep`.
- Asserted as held inside the helper functions via `mtx_assert(MA_OWNED)`.
- Documented in the locking comment at the top of the file and in `LOCKING.md`.

With the concepts in Section 4 and Section 5, every one of those properties is defensible from first principles. The mutex is not ceremony. It is the infrastructure that turns a driver that happens to work into a driver that is correct.

11.6.16 Spin Mutexes in More Detail

We have said that spin mutexes (`MTX_SPIN`) exist because some contexts cannot sleep. Let us look more closely at why.

The FreeBSD kernel has several execution contexts. Most driver code runs in **thread context**: the kernel is executing on behalf of some thread (a user thread that made a syscall, a kernel thread dedicated to some task, or a callout running in mpsafe mode). In thread context, sleeping is legal: the scheduler can park the thread and schedule another one.

A small but critical set of contexts cannot sleep. The most important is **hardware interrupt context**: the code that runs when a hardware interrupt fires. An interrupt may preempt any thread at any instant, run a short handler (called an *ithread* or *filter*), and return. While the handler runs, the thread it preempted cannot make progress. The handler must finish quickly, and it must not block. Sleeping would mean calling the scheduler, and calling the scheduler from inside an interrupt is not safe on the platforms FreeBSD supports.

Another non-sleep context is **critical sections** entered with `critical_enter(9)`. These disable preemption on the current CPU; the code inside runs to completion without the scheduler being allowed to pick a different thread. Critical sections are rarely used by driver writers directly; they appear more in low-level kernel code.

For code in any non-sleep context, a sleep mutex is the wrong tool. Acquiring a sleep mutex that is held would require sleeping, and you cannot. You need a mutex that spins: tries in a tight loop until it succeeds.

MTX_SPIN mutexes do exactly that. When you call `mtx_lock_spin(&m)`, the code:

1. Disables interrupts on the current CPU (because otherwise an interrupt handler could preempt you while you hold the spin mutex, leading to deadlock if the handler wants the same mutex).
2. Tries an atomic compare-and-swap to acquire the mutex.
3. If it fails, spins in a loop, periodically retrying.
4. Once acquired, proceeds with the critical section. Interrupts remain disabled.
5. `mtx_unlock_spin(&m)` releases the mutex and re-enables interrupts.

The rules for spin mutexes are strict:

- The critical section must be **very short**: holding a spin mutex is holding back every other CPU that wants it plus disabling interrupts on the current CPU. Microseconds matter.
- You **cannot sleep** while holding a spin mutex. `malloc(9)` with `M_WAITOK` is illegal. `mtx_sleep(9)` is illegal. Even a page fault is illegal (you must not touch user memory or paged kernel memory under a spin mutex).
- You **cannot acquire a sleep mutex** while holding a spin mutex. The sleep mutex would try to sleep if contended, and you cannot sleep.

For the `myfirst` driver, spin mutexes are the wrong choice. We never run in interrupt context. Our critical sections may call into the `cbuf` helpers, which contain `memcpy` loops that are short but not microsecond-tiny. `MTX_DEF` is correct.

For a driver that *does* run in interrupt context (Chapter 14 will cover this), spin mutexes often appear on the shortest critical sections: the ones between the interrupt handler and the top-half code. Longer critical sections can be protected by an `MTX_DEF` mutex that the interrupt handler does not take; the handler simply queues work for the top-half to do under its sleep mutex.

11.6.17 The MTX_RECURSE Flag and Lock Recursion

A subtle flag in `mtx_init` is `MTX_RECURSE`. Without it, a thread that already holds a mutex and tries to acquire it again will panic (under `INVARIANTS`) or deadlock (without `INVARIANTS`). With `MTX_RECURSE`, the second acquisition is counted; the mutex is released only when every acquire has been matched with an unlock.

Most drivers do not need `MTX_RECURSE`. The fact that a function tries to acquire a lock it already holds usually means the code is structured badly: a helper is being called from both lock-holding and non-lock-holding contexts, and the helper does not know which it is.

Fix the structure, not the mutex. Split the helper into a “with lock” and a “without lock” version. Name them `_locked` and `_unlocked` respectively, matching FreeBSD convention. Example:

```
static size_t
myfirst_buf_read_locked(struct myfirst_softc *sc, void *dst, size_t n)
{
    mtx_assert(&sc->mtx, MA_OWNED);
    /* ... buffer logic ... */
}

static size_t
myfirst_buf_read(struct myfirst_softc *sc, void *dst, size_t n)
{
    size_t got;

    mtx_lock(&sc->mtx);
    got = myfirst_buf_read_locked(sc, dst, n);
    mtx_unlock(&sc->mtx);
    return (got);
}
```

Now the two call sites are explicit: one takes the lock, the other does not. Neither needs `MTX_RECURSE`. Neither confuses the reader. This is how you should structure code as your driver grows.

There are rare exceptions where `MTX_RECURSE` is legitimately useful. A complex data structure with internal recursion (for example, a tree that uses the same lock at every node) may need it. `buf_ring`’s drain operation uses it. Those are specialised cases; for the ordinary driver, structure your code to avoid recursion and do not add the flag.

11.6.18 Adaptive Spinning

FreeBSD’s sleep mutexes include an optimization called **adaptive spinning**. When `mtx_lock` cannot acquire an `MTX_DEF` mutex because another CPU holds it, the lock does not immediately put the thread to sleep. It first spins for a short time, hoping that the holder will release the mutex quickly. Only if the spin exceeds a threshold does the lock fall through to the sleep path.

The rationale: most mutex-held intervals are short (microseconds), and sleeping plus waking is expensive. Spinning for a few microseconds usually beats going through the scheduler. Adaptive spinning recovers most of the performance benefit of spin mutexes without the constraints.

You can see the implementation in `/usr/src/sys/kern/kern_mutex.c`, in the function `__mtx_lock_sleep`. The code checks whether the holder is currently running on another CPU and spins as long as it is. If the holder descheduled (went to sleep itself), spinning is pointless and the waiter goes to sleep too.

For the driver writer, adaptive spinning means the mutex performance is better than a naive analysis would suggest. A short critical section on a low-contention mutex costs roughly the atomic compare-and-swap, nothing more. Only under real contention do you pay the full sleep/wake cost.

11.6.19 A Closer Look at `mtx_sleep`'s Atomicity

Chapter 10 explained that `mtx_sleep` atomically drops the mutex and puts the caller on the sleep queue. The atomicity matters because the alternative, drop-then-sleep, has a window through which a wakeup can escape unheard.

Consider the alternative sequence:

```
mtx_unlock(&sc->mtx);  
sleep_on(&sc->cb);  
mtx_lock(&sc->mtx);
```

Between the unlock and the sleep, another CPU could acquire the mutex, observe that the condition we were waiting for has become true, call `wakeup(&sc->cb)`, and return. Our wakeup has been delivered to a sleep queue we have not yet joined. We sleep on a condition that, from our thread's perspective, will never again become true: the signal has already been missed.

This is the classical **lost wakeup** race. It is one of the core reasons the atomic drop-and-sleep operation exists. `mtx_sleep` closes the window by enqueueing the caller on the sleep queue *before* it drops the outer mutex. The “Reference: A Closer Look at Sleep Queues” section later in this chapter walks through the sequence of locks and state transitions in more detail for readers who want to see exactly how the kernel arranges this. For the body of the chapter, the rule is enough: use `mtx_sleep` with the outer mutex as its interlock, and the lost-wakeup race disappears.

11.6.20 A Guided Tour Through `kern_mutex.c`

If you have an hour of patience, opening `/usr/src/sys/kern/kern_mutex.c` is worth the investment. You do not need to understand every line. Three functions are particularly illuminating:

`__mtx_lock_flags`: the fast path for acquiring a mutex. The interesting thing is how short it is. In the uncontested case, acquiring a mutex is essentially one atomic compare-and-swap plus some lock-profiling bookkeeping. That is all.

`__mtx_lock_sleep`: the slow path, reached only when the fast path's compare-and-swap fails. This is where adaptive spinning, priority propagation, and the actual sleep queue work happen. The code is elaborate, but the structure is: try a few spins, hand off to the sleep queue, re-enter the scheduler, eventually acquire.

`__mtx_unlock_flags` and **`__mtx_unlock_sleep`**: the release paths. Also mostly fast: atomic release, then, if waiters exist, wake one up.

You are not expected to read every line. You are expected to be able to say: “this is where the atomic I use actually lives, and this is what it does”. Twenty minutes of skimming gets you that.

11.6.21 Comparing Mutex to Semaphore, Monitor, and Binary Flag

For readers who have encountered other synchronization primitives, it helps to situate FreeBSD's mutex.

A **semaphore** is a counter. Threads “P” (decrement) and “V” (increment) it; a P that would go negative blocks. A binary semaphore (counter values 0 or 1) is similar to a mutex, but semaphores typically allow the “release” to be done by a thread other than the one that acquired. Mutexes require the same thread to release.

A **monitor** is a language-level construct combining a mutex and one or more condition variables, with the mutex automatically acquired on entry and released on exit. C does not have monitors as a language feature, but the pattern of “mutex + condition variable” is the same idea.

A **binary flag** (a `volatile int` that threads set and clear) is what naive programmers sometimes use to implement mutual exclusion. It does not work: two threads can both see the flag as zero and both set it to one, both proceed as if they were exclusive. This is the race we saw with the counter in Section 2. Real mutexes use atomic compare-and-swap, not naked flags.

FreeBSD’s mutex is a traditional mutex: only the acquiring thread may release, recursive acquisition requires explicit opt-in, the primitive integrates with the scheduler for blocking, and priority propagation is built in. It is the simplest and most commonly appropriate tool for driver synchronization.

11.6.22 Mutex Lifetime Rules

Every mutex has a lifetime. The rules are:

1. Call `mtx_init` exactly once before any use.
2. Acquire and release the mutex any number of times.
3. Call `mtx_destroy` exactly once after all use. Calling `mtx_destroy` while any thread is blocked on the mutex is undefined.
4. After `mtx_destroy`, the memory may be reused. Do not access the mutex again.

The third rule is why our detach path is careful about the order: destroy the `cdev` (which stops new handlers from starting and waits for running ones to finish), then destroy the mutex. If we destroyed the mutex first, running handlers could still be inside `mtx_lock` or `mtx_unlock` on an already-destroyed mutex.

Lifetime bugs with mutexes tend to be catastrophic (memory corruption, use-after-free panics). They are easier to avoid than to debug, so the standard advice applies: always think carefully about the order of teardown.

11.6.23 The MTX_NEW Option

`mtx_init`’s last argument can include `MTX_NEW`, which tells the function “this memory is fresh; you do not need to check for a prior initialization”. On an `INVAARIANTS` kernel, `mtx_init` verifies that the mutex was not previously initialized without a matching `mtx_destroy`. `MTX_NEW` skips this check.

Use `MTX_NEW` when you are initializing a mutex in memory that you know has not been used for this purpose before. Use the default (no `MTX_NEW`) when the mutex might be re-initialized (for example, in a handler that attaches and detaches repeatedly on the same softc). For our driver, the softc is re-allocated on each attach, so the mutex memory is always fresh; `MTX_NEW` is harmless but not required.

11.6.24 Nested Locks and Deadlock Patterns

A thread that holds lock A and wants to acquire lock B is performing a **nested lock acquisition**. Nested acquisitions are where deadlocks and lock-order violations live. Four patterns cover nearly all of them.

Pattern 1: Simple two-lock deadlock. Thread 1 holds A, wants B. Thread 2 holds B, wants A. Neither makes progress. Fix: globally order the locks so every thread acquires A before B.

Pattern 2: Lock order across subsystems. Subsystem X always acquires its lock first; subsystem Y always acquires its lock first. If they ever need each other's locks, the order depends on which of X and Y happens to call into the other. Fix: document a subsystem ordering that is consistent across every cross-subsystem path.

Pattern 3: Lock inversion via callback. A function is called with lock A held. Inside, it calls a callback, which tries to acquire A again. If A is not recursive, deadlock. Fix: drop A before the callback, or split the function so the callback is invoked outside the lock.

Pattern 4: Sleep-with-lock deadlock. A function holds a mutex and then calls something that sleeps. If the sleeper needs the same mutex (perhaps through a different code path), deadlock. Fix: drop the mutex before sleeping, or use `mtx_sleep` so the drop happens atomically.

For the `myfirst` driver, we have one lock and no nested acquisitions; none of the patterns apply. Chapter 12 introduces `sx` locks; as soon as we have a second lock class, patterns 1 and 3 become relevant, and `WITNESS` becomes critical.

11.6.25 Wrapping Up Section 5

A mutex is a tool for serializing access to shared state. FreeBSD provides two shapes (`MTX_DEF` for sleep mutexes, `MTX_SPIN` for spin mutexes), and `MTX_DEF` is the correct choice for our driver. The API is small: `mtx_init`, `mtx_lock`, `mtx_unlock`, `mtx_destroy`, `mtx_assert`, `mtx_sleep`. The rules are precise: hold short, do not sleep while holding a non-sleepable lock, acquire locks in a consistent order, release what you acquire, and use `WITNESS` to check.

Section 6 takes the theory we have built and exercises it: how to write user-space programs that actually stress the driver, and how to observe the results.

11.7 Section 6: Testing Multi-threaded Access

Building a correct driver is half the work. The other half is developing tests that can catch the mistakes that slip through the design review. Chapter 10 introduced `producer_consumer.c`, a two-process round-trip test that is the single best test in this book's kit for detecting buffer-level correctness problems. Chapter 11 takes that foundation and adds the tools you need for concurrency specifically: multi-threaded tests using `pthread(3)`, multi-process tests using `fork(2)`, and a load harness that can sustain pressure on the driver for long enough to expose rare timing bugs.

The goal is not to exhaustively test every possible interleaving. That is impossible. The goal is to increase the rate at which the scheduler visits interleavings the driver has not seen, so that, if a bug is present, we are likely to observe its effects.

11.7.1 The Testing Ladder

Concurrency testing climbs a ladder. At the bottom is a single-threaded test: one process, one thread, one call at a time. This is what `cat` and `echo` do. At the top is a distributed multi-process, multi-threaded, timing-varying,

long-running test. The higher you climb, the more likely you are to catch concurrency bugs; the more expensive each test becomes to set up and interpret.

The rungs, from bottom to top:

1. **Single-threaded smoke tests.** One cat, one echo. Useful for confirming the driver is alive; useless for concurrency.
2. **Two-process round trip.** `producer_consumer` from Chapter 10. One writer, one reader, content verification. Catches most single-lock issues.
3. **Multi-threaded within a process.** pthread-based tests where multiple threads in the same process talk to the same descriptor or to distinct descriptors. Exposes intra-process concurrency.
4. **Many-process stress.** fork-based tests with N producers and M consumers. Exposes inter-process concurrency and lock contention.
5. **Long-running stress.** Any of the above, run for hours. Exposes timing-sensitive bugs that are rare on the per-operation basis but certain given enough time.

We will build rungs 3, 4, and 5 in this section. Rung 2 already exists from Chapter 10.

11.7.2 Multiple Threads in One Process

A single process with multiple threads is useful because the threads share file descriptors. Two threads can read from the same descriptor; both calls go into `myfirst_read` with the same `dev` and (critically) the same per-descriptor `fh`.

Here is a minimal multi-threaded reader:

```
/* mt_reader.c: multiple threads reading from one descriptor. */
#include <sys/types.h>
#include <err.h>
#include <errno.h>
#include <fcntl.h>
#include <inttypes.h>
#include <pthread.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#define DEVPATH        "/dev/myfirst"
#define NTHREADS       4
#define BYTES_PER_THR  (256 * 1024)
#define BLOCK          4096

static int      g_fd;
static uint64_t total[NTHREADS];
static uint32_t sum[NTHREADS];
```



```
static uint32_t
checksum(const char *p, size_t n)
{
    uint32_t s = 0;
    for (size_t i = 0; i < n; i++)
        s = s * 31u + (uint8_t)p[i];
    return (s);
}

static void *
reader(void *arg)
{
    int tid = *(int *)arg;
    char buf[BLOCK];
    uint64_t got = 0;
    uint32_t sm = 0;

    while (got < BYTES_PER_THR) {
        ssize_t n = read(g_fd, buf, sizeof(buf));
        if (n < 0) {
            if (errno == EINTR)
                continue;
            warn("thread %d: read", tid);
            break;
        }
        if (n == 0)
            break;
        sm += checksum(buf, n);
        got += n;
    }
    total[tid] = got;
    sum[tid] = sm;
    return (NULL);
}

int
main(void)
{
    pthread_t tids[NTHREADS];
    int ids[NTHREADS];

    g_fd = open(DEVPATH, O_RDONLY);
    if (g_fd < 0)
```

```

        err(1, "open %s", DEVPATH);

    for (int i = 0; i < NTHREADS; i++) {
        ids[i] = i;
        if (pthread_create(&tids[i], NULL, reader, &ids[i]) != 0)
            err(1, "pthread_create");
    }
    for (int i = 0; i < NTHREADS; i++)
        pthread_join(tids[i], NULL);

    uint64_t grand = 0;
    for (int i = 0; i < NTHREADS; i++) {
        printf("thread %d: %" PRIu64 " bytes, checksum 0x%08x\n",
            i, total[i], sum[i]);
        grand += total[i];
    }
    printf("grand total: %" PRIu64 "\n", grand);

    close(g_fd);
    return (0);
}

```

The companion file under `examples/part-03/ch11-concurrency/userland/mt_reader.c` is identical to this listing; you can either type it from the book or copy it from the examples tree.

Build with:

```
$ cc -Wall -Wextra -pthread -o mt_reader mt_reader.c
```

Start a writer in another terminal (or fork one before creating threads), then run this tester. Each thread pulls bytes out of the driver. Because the driver has one mutex, the reads serialize; there is no concurrency benefit, but there is also no incorrectness. Each thread sees a subset of the stream, and the concatenation of all threads' bytes is the full stream.

This is an important property. The driver makes no promise about which thread sees which bytes; it only promises that the total is conserved. If you want per-reader streams, you need a different driver design (Challenge exercise 3 in Chapter 10 explored this). For now, the test confirms that multiple readers in one process behave correctly.

11.7.3 Many Processes in Parallel

A fork-based test spawns *N* children, each doing its own thing against the device. This gives us independent processes, independent file descriptors, and independent schedule decisions. The kernel is more likely to interleave them in novel ways.

Here is the skeleton:

```
/* mp_stress.c: N processes hammering the driver concurrently. */
#include <sys/types.h>
#include <sys/wait.h>
#include <err.h>
#include <errno.h>
#include <fcntl.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#define DEVPATH        "/dev/myfirst"
#define NWRITERS       2
#define NREADERS       2
#define SECONDS        30

static volatile sig_atomic_t stop;

static void
sigalrm(int s __unused)
{
    stop = 1;
}

static int
child_writer(int id)
{
    int fd;
    char buf[1024];
    unsigned long long written = 0;

    fd = open(DEVPATH, O_WRONLY);
    if (fd < 0)
        err(1, "writer %d: open", id);
    memset(buf, 'a' + id, sizeof(buf));

    while (!stop) {
        ssize_t n = write(fd, buf, sizeof(buf));
        if (n < 0) {
            if (errno == EINTR)
                continue;
            break;
        }
    }
}
```

```
        written += n;
    }
    close(fd);
    printf("writer %d: %llu bytes\n", id, written);
    return (0);
}

static int
child_reader(int id)
{
    int fd;
    char buf[1024];
    unsigned long long got = 0;

    fd = open(DEVPATH, O_RDONLY);
    if (fd < 0)
        err(1, "reader %d: open", id);

    while (!stop) {
        ssize_t n = read(fd, buf, sizeof(buf));
        if (n < 0) {
            if (errno == EINTR)
                continue;
            break;
        }
        got += n;
    }
    close(fd);
    printf("reader %d: %llu bytes\n", id, got);
    return (0);
}

int
main(void)
{
    pid_t pids[NWRITERS + NREADERS];
    int n = 0;

    signal(SIGALRM, sigalrm);

    for (int i = 0; i < NWRITERS; i++) {
        pid_t pid = fork();
        if (pid < 0)
            err(1, "fork");
    }
}
```

```
        if (pid == 0) {
            signal(SIGALRM, sigalrm);
            alarm(SECONDS);
            _exit(child_writer(i));
        }
        pids[n++] = pid;
    }
    for (int i = 0; i < NREADERS; i++) {
        pid_t pid = fork();
        if (pid < 0)
            err(1, "fork");
        if (pid == 0) {
            signal(SIGALRM, sigalrm);
            alarm(SECONDS);
            _exit(child_reader(i));
        }
        pids[n++] = pid;
    }

    for (int i = 0; i < n; i++) {
        int status;
        waitpid(pids[i], &status, 0);
    }
    return (0);
}
```

The companion source under `examples/part-03/ch11-concurrency/userland/mp_stress.c` matches this listing. The `signal(SIGALRM, sigalrm)` reinstall after `fork(2)` is intentional; `fork(2)` inherits the parent's signal disposition, but reinstalling makes the intent explicit and survives the (rare) case where the parent's handler had been changed in between.

Build and run:

```
$ cc -Wall -Wextra -o mp_stress mp_stress.c
$ ./mp_stress
writer 0: 47382528 bytes
writer 1: 48242688 bytes
reader 0: 47669248 bytes
reader 1: 47956992 bytes
```

Look at the totals. The sum of readers should equal the sum of writers (plus or minus whatever is left in the buffer at the end of the test window). If they do not, we have either a driver bug or a reporting bug; neither is acceptable.

This test, run for thirty seconds, produces roughly one hundred million bytes of traffic through the driver, with

four concurrent processes. On a four-core machine, all four can be making progress at any instant. If there is a concurrency bug that our review missed, it has thirty seconds of real compute to find it.

11.7.4 Extended-Duration Tests

For the most elusive bugs, run the tests for hours. A simple wrapper:

```
$ for i in $(seq 1 100); do
    echo "iteration $i" >> /tmp/mp_stress.log
    ./mp_stress >> /tmp/mp_stress.log 2>&1
    sleep 1
done
```

After fifty iterations (roughly twenty-five minutes of cumulative driver stress), review the log. If every iteration's byte counts are internally consistent, you have a strong signal that the driver is not suffering from frequent concurrency bugs. If any iteration shows inconsistency, you have a bug; save the log, reproduce with lower NWRITERS/NREADERS, and investigate.

A script like the above is a cheap substitute for a proper continuous-integration pipeline. For serious driver work, a CI that runs the stress suite nightly on multiple hardware configurations is the gold standard. For learning, the script above is enough.

11.7.5 Observing Without Disturbing

A subtle issue: adding logging to a concurrent test can itself disturb the timing and mask the bug. If your test prints a line for every operation, the printing becomes a bottleneck, the threads serialize around stdout, and the interesting interleavings no longer happen.

Techniques that reduce observer effect:

- **Buffer logs in memory; flush at the end.** Each thread appends to its own local array; main prints the arrays after all threads finish.
- **Use `ktrace(1)` instead of in-process printing.** `ktrace` captures syscalls from a running process without modifying the process; the dump can be analyzed afterward.
- **Use `dtrace(1)` probes.** `dtrace` is designed to have minimal impact on the observed code path.
- **Keep counters, not line-by-line logs.** A mismatch count is a single integer; it compresses a lot of information into something cheap to update.

`producer_consumer` from Chapter 10 uses the counter approach: it updates a checksum per block and a total byte count, then reports both at the end. The test is effectively invisible to the driver's timing.

11.7.6 A Testing Workflow

Putting it together, here is a reasonable workflow for a new change to the driver:

1. **Smoke test.** Load the driver, `printf 'hello' > /dev/myfirst`, `cat /dev/myfirst`. Confirm basic operation.

2. **Round-trip test.** Run `producer_consumer`. Confirm zero mismatches.
3. **Multi-threaded in-process.** Run `mt_reader` against a continuously-running `cat /dev/zero > /dev/myfirst`. Confirm the total matches.
4. **Multi-process stress.** Run `mp_stress` for thirty seconds. Confirm the byte counts are consistent.
5. **Long-running stress.** Run the loop wrapper for thirty minutes to an hour.
6. **Debug-kernel regression.** Repeat steps 1-4 on a WITNESS-enabled kernel. Confirm no warnings.

If every step passes, the change is likely to be safe. If any step fails, the failure mode tells you where to look.

11.7.7 A Latency-Measuring Tester

Sometimes what you want to know is not “does the driver work” but “how quickly does it respond under load”. The mutex’s cost, the sleep queue’s wake-up latency, and the scheduler’s decisions all combine to produce a distribution of response times that is worth observing directly.

Here is a simple latency tester. It opens the device, measures how long each `read(2)` takes, and prints a histogram.

```
/* lat_tester.c: measure read latency against /dev/myfirst. */
#include <sys/types.h>
#include <sys/time.h>
#include <err.h>
#include <fcntl.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#define DEVPATH "/dev/myfirst"
#define NSAMPLES 10000
#define BLOCK 1024

static uint64_t
nanos(void)
{
    struct timespec ts;
    clock_gettime(CLOCK_MONOTONIC, &ts);
    return ((uint64_t)ts.tv_sec * 1000000000 + ts.tv_nsec);
}

int
main(void)
{
    int fd = open(DEVPATH, O_RDONLY | O_NONBLOCK);
    if (fd < 0)
```

```

        err(1, "open");
char buf[BLOCK];
uint64_t samples[NSAMPLES];
int nvalid = 0;

for (int i = 0; i < NSAMPLES; i++) {
    uint64_t t0 = nanos();
    ssize_t n = read(fd, buf, sizeof(buf));
    uint64_t t1 = nanos();
    if (n > 0)
        samples[nvalid++] = t1 - t0;
    else
        usleep(100);
}
close(fd);

/* Simple bucketed histogram. */
uint64_t buckets[10] = {0};
const char *labels[10] = {
    "<1us   ", "<10us  ", "<100us ", "<1ms   ",
    "<10ms  ", "<100ms ", "<1s    ", ">=1s   ",
    "", ""
};

for (int i = 0; i < nvalid; i++) {
    uint64_t us = samples[i];
    int b = 0;
    if (us < 1000) b = 0;
    else if (us < 10000) b = 1;
    else if (us < 100000) b = 2;
    else if (us < 1000000) b = 3;
    else if (us < 10000000) b = 4;
    else if (us < 100000000) b = 5;
    else if (us < 1000000000) b = 6;
    else b = 7;
    buckets[b]++;
}

printf("Latency histogram (%d samples):\n", nvalid);
for (int i = 0; i < 8; i++)
    printf("  %s %6llu\n",
        labels[i], (unsigned long long)buckets[i]);
return (0);
}

```


Run with a concurrent writer:

```
$ dd if=/dev/zero of=/dev/myfirst bs=1k &
$ ./lat_tester
Latency histogram (10000 samples):
<1us      3421
<10us     6124
<100us    423
<1ms      28
<10ms     4
<100ms    0
<1s       0
>=1s      0
```

Most reads complete in a microsecond or two. A handful take longer, usually because the reader had to wait on the mutex while the writer was inside. Very occasionally, a read takes milliseconds; that is usually because the thread was preempted or had to sleep.

This is the distribution of your driver's responsiveness under load. If the long tail is unacceptably long, the driver needs finer-grained locking or fewer critical-section operations. For our `myfirst`, the tail is short enough that nothing needs changing.

11.7.8 A Tester That Probes for Torn Reads

If your driver has any unprotected multi-field reads (the book's recommendation is never, but real drivers sometimes have them), you can detect torn reads with a targeted tester.

Imagine we had a field that could be torn on a 32-bit platform. The tester would:

1. Fork a writer that continuously updates the field with known patterns.
2. Fork a reader that continuously reads the field and checks the pattern.
3. Report every read that does not match any valid pattern.

For our driver, no such field exists (all `uint64_t` counters are per-CPU, all composite state is under the mutex). But building the tester is a valuable exercise, because it trains you to write probes for specific classes of bug.

11.7.9 Multi-Core Scaling

A test we have not yet written: does the driver scale as you add more cores?

The idea is simple: run `mp_stress` with increasing `NWRITERS/NREADERS` and observe the total throughput. Ideally, throughput should grow linearly with core count up to some saturation point. In practice, single-mutex drivers saturate early, because every operation serializes on the one mutex.

Build a scaling test:

```
$ for n in 1 2 4 8 16; do
    NWRITERS=$n NREADERS=$n ./mp_stress > /tmp/scale_${n}.txt
    total=$(grep ^writer /tmp/scale_${n}.txt | awk '{s+=$3} END {print s}')
    echo "$n writers, $n readers: $total bytes"
done
```

The `mp_stress` program needs to accept `NWRITERS` and `NREADERS` from environment variables for this to work; exercise for the reader.

On a four-core machine, a single-mutex driver typically saturates around 2-4 writers plus 2-4 readers. Beyond that, throughput flattens or drops, because the mutex becomes the bottleneck. This is the signal that finer-grained locking (Chapter 12's `sx` locks, for example) might be justified. For `myfirst` at this stage, saturation is not a concern; the teaching pedagogy matters more than absolute throughput.

11.7.10 Coverage of the Failure Modes

A final check: does the test kit cover every failure mode we enumerated in Section 1?

- **Data corruption:** `producer_consumer` catches this directly by comparing checksums.
- **Lost updates:** if a counter is wrong, the `producer_consumer` total will not match.
- **Torn values:** not directly tested, but any torn read that affects correctness propagates to a wrong checksum.
- **Inconsistent composite state:** if the `cbuf` indices become inconsistent, the driver will either corrupt data (caught by checksum) or deadlock (caught by the test hanging and our timeout).

The test kit is comprehensive against the failure modes we know about. It is not comprehensive against unknown bugs, but no test kit can be. The combination of debug-kernel runs, regression tests, and long-running stress tests is as close as we can get.

11.7.11 Wrapping Up Section 6

Testing is the fastest-feedback part of driver work. The tests we have just built cover three orders of magnitude more interleavings than a single cat could. Running them on both a production and a debug kernel gives you verification from both angles.

Section 7 takes the next step: what do you do when a test fails?

11.8 Section 7: Debugging and Verifying Thread Safety

A concurrency bug's most reliable signature is that it does not reproduce on demand. You run the test; it fails. You run it again; it passes. You run it a hundred times; it fails twice, on different lines. This section is about the tools and techniques that let you corner such bugs despite their unreliability.

11.8.1 Typical Symptoms

Before we get to tools, let us catalogue the symptoms. Each symptom suggests a specific class of bug.

Symptom: the test intermittently reports corrupted data, but the driver does not panic. This is usually a missed lock: a field that is accessed without synchronization from more than one stream. The corruption is the result of the race; the driver survives because the corrupted value does not immediately cause a derivable failure.

Symptom: the driver panics with a backtrace inside `mtx_lock` or `mtx_unlock`. This is often a use-after-free of the mutex itself. The most common cause is that `mtx_destroy` was called while another thread was still using the mutex. The cure is to re-examine the detach path: the mutex must outlive every use of it.

Symptom: `WITNESS` reports a lock-order reversal. A thread acquired locks in an order inconsistent with an earlier-observed order. The fix is to define a global lock order and make every code path obey it.

Symptom: `KASSERT` fires inside `mtx_assert(MA_OWNED)`. A helper expected the mutex to be held and it was not. The fix is to find the call site of the helper and add the missing `mtx_lock`.

Symptom: the test hangs forever. A thread is sleeping on a channel and nobody ever calls wakeup on that channel. The fix is usually a missing `wakeup(&sc->cb)` in the I/O path (or equivalent). Chapter 10 Section 5 enumerated the rules.

Symptom: the driver performs very poorly under load. The mutex is contended more heavily than it needs to be. The fix is usually to shorten critical sections (move more work outside the lock) or to split the lock into finer pieces (Chapter 12 material).

Symptom: the driver crashes weeks after deployment, with a backtrace that does not obviously involve concurrency. This is the worst-case scenario and almost always a concurrency bug whose effects finally compounded. The fix is to audit as in Section 3 and run stress tests as in Section 6.

11.8.2 INVARIANTS and KASSERT

INVARIANTS is a kernel build option that enables assertions throughout the kernel. When INVARIANTS is set, `KASSERT(cond, args)` evaluates `cond` and panics with `args` if the condition is false. When INVARIANTS is not set, `KASSERT` compiles to nothing.

This makes `KASSERT` nearly free in production and extremely valuable in development. Every invariant your code depends on should be a `KASSERT`. For example:

```
KASSERT(sc->cb.cb_used <= sc->cb.cb_size,  
        ("cbuf used exceeds size: %zu > %zu",  
         sc->cb.cb_used, sc->cb.cb_size));
```

This says “I believe `cb_used` never exceeds `cb_size`. If the code that produces this truth is ever broken, I want to know immediately, not hours later when something else fails.”

Add `KASSERT` calls liberally to your driver. Every precondition, every invariant, every “this cannot happen” branch. The cost is zero in production; the benefit is that development bugs get caught at the moment they appear, not downstream.

For the `myfirst` driver, some useful additions:

```
/* In cbuf.c, at the top of cbuf_write: */
KASSERT(cb->cb_used <= cb->cb_size,
    ("cbuf_write: cb_used %zu exceeds cb_size %zu",
     cb->cb_used, cb->cb_size));
KASSERT(cb->cb_head < cb->cb_size,
    ("cbuf_write: cb_head %zu not less than cb_size %zu",
     cb->cb_head, cb->cb_size));

/* In myfirst_buf_read and myfirst_buf_write: */
mtx_assert(&sc->mtx, MA_OWNED);
```

Every such check is a small bet that catches a future mistake.

11.8.3 WITNESS in Action

When WITNESS warns, the output is detailed. A typical warning for a lock-order reversal looks like this:

```
lock order reversal:
 1st 0xfffffe000123a000 foo_lock (foo, sleep mutex) @ /usr/src/sys/dev/foo/foo.c:100
 2nd 0xfffffe000123a080 bar_lock (bar, sleep mutex) @ /usr/src/sys/dev/bar/bar.c:200
lock order reversal detected for lock group "bar" -> "foo"
stack backtrace:
...
```

The key pieces are:

- The two locks involved, with their names, types, and the source locations where they were acquired.
- The conflicting order (`bar -> foo`), which is the reverse of a previously-established order (`foo -> bar`).
- A stack backtrace showing the path that led to the reversal.

The fix is one of two things: either change one of the acquisition sites to match the existing order, or recognize that the two code paths should not both be holding both locks. The stack backtrace tells you which code path is involved; reading the source tells you which change is the right one.

For the `myfirst` driver, which has one lock, WITNESS cannot report a lock-order reversal. The only WITNESS warnings we could see relate to sleeping with a lock held or recursive acquisition. Both are worth testing deliberately, as we do in Lab 7.3.

11.8.4 Reading a Kernel Panic Backtrace

Sometimes a bug produces a panic. The kernel prints a backtrace, enters the debugger (if DDB is configured), or reboots (if not). Your job is to extract as much information as possible from the backtrace before the system goes away.

The first lines of a panic typically identify the failure:

```
panic: mtx_lock() of spin mutex @ ...: recursed on non-recursive mutex myfirst0 @ ...
cpuid = 2
...
```

From here:

- The panic message is the most important line. `mtx_lock of spin mutex` hints at one kind of bug; sleeping with mutex held at another; general protection fault at a different class entirely.
- The `cpuid` tells you which CPU experienced the panic, which can matter if the bug is specific to a particular scheduling environment.
- The stack backtrace shows the functions on the way down. Read it top-to-bottom: the bottom is where the panic was detected; the top is where the execution began. The function just above the panic function is usually the one that made the mistake.

If DDB is configured, you can interact with the debugger at panic time: `bt` (backtrace), `show mutex <addr>` (inspect a mutex), `show alllocks` (every lock held by every thread). This is the ninja mode of driver debugging; Chapter 12 and the debugging references touch it.

11.8.5 dtrace(1) as a Silent Observer

`dtrace(1)` is FreeBSD's dynamic tracing framework. It lets you attach probes to kernel functions (and user-space functions, with the right libraries) and collect data with minimal impact on the observed code.

A simple `dtrace` command to count mutex acquisitions on `myfirst`'s mutex:

```
# dtrace -n 'fbt::__mtx_lock_flags:entry /arg0 != 0/ { @[execname] = count(); }'
```

Run the driver under load, then Ctrl-C the `dtrace`. You get a table of per-process lock acquisition counts. If one process dominates, that's the source of contention.

Another useful one-liner: trace when threads enter and exit our read handler:

```
# dtrace -n 'fbt::myfirst_read:entry { printf("%d reading %d bytes", tid, arg1); }'
```

The exact arg indices depend on the function's ABI; `dtrace -l | grep myfirst` lists the probes available.

`dtrace` is not magic: it uses real kernel mechanisms (typically the function-boundary-tracing provider) that have nonzero cost. But that cost is dramatically lower than `printf`-based logging, and it can be turned on and off without modifying the driver.

Chapter 15 will cover `dtrace` in more detail. For Chapter 11, the key idea is that `dtrace` is your friend for observing a live driver under load.

11.8.6 A Debugging Checklist

When a concurrency bug appears, work through this checklist in order:

1. **Can you reproduce it reliably?** If yes, great; if no, run the test in a loop and find the failure rate.
2. **Does `WITNESS` report anything?** Boot with options `WITNESS INVARIANTS`. Rerun. Collect every warning.
3. **Are there `KASSERT` failures?** These fire as panics; the message identifies the invariant.
4. **Is the bug deterministic given sufficient stress?** If it appears every time with four writers and four readers, you have a workload to bisect; if it appears only under specific conditions, start isolating those conditions.
5. **Which field is corrupted?** Add targeted logging or `dtrace` probes around the suspect accesses.
6. **What is the missing synchronization?** Audit the access paths against the rules in Section 3.
7. **Is the fix consistent with the rest of the driver's locking strategy?** Document the change in `LOCKING.md`.
8. **Does the test now pass reliably?** Run the stress tests for twice as long as you would otherwise to confirm.

Most concurrency bugs yield to this process. Some require deep diving into the specific driver or the kernel; those are the ones Chapter 12 and later material help you prepare for.

11.8.7 A Walkthrough: Diagnosing a Missing Wakeup

To make the debugging process concrete, let us walk through a hypothetical bug. Suppose you have modified `myfirst_write` to update some auxiliary state, and after the change, a two-terminal test (one cat, one echo) hangs. The cat is asleep, the echo has returned, and the bytes are not coming out.

Step 1: confirm the symptom.

```
$ ps -AxH -o pid,wchan,command | grep cat
12345  myfrd  cat /dev/myfirst
```

The cat is sleeping on `myfrd`. That is our sleep channel name. It is waiting for data.

Step 2: inspect the driver state.

```
$ sysctl dev.myfirst.0.stats.cb_used
dev.myfirst.0.stats.cb_used: 5
```

The buffer has 5 bytes in it. The reader should be able to drain those bytes and return them to cat. So why isn't it?

Step 3: look at the code. The cat's thread is in `mtx_sleep(&sc->cb, ..., "myfrd", ...)`. It is waiting for `wakeup(&sc->cb)`. Who calls that?

Grep the source:

```
$ grep -n 'wakeup(&sc->cb)' myfirst.c
180:         wakeup(&sc->cb);
220:         wakeup(&sc->cb);
```

Two call sites. One is in `myfirst_read` (after a successful read, to wake writers waiting for space). The other is in `myfirst_write` (after a successful write, to wake readers waiting for data).

Step 4: inspect the write path. Did the write actually run the wakeup?

```
mtx_lock(&sc->mtx);
put = myfirst_buf_write(sc, bounce, want);
/* ... update aux state ... */
mtx_unlock(&sc->mtx);

/* new code: do some bookkeeping */
update_stats(sc, put);

wakeup(&sc->cb);
selwakeup(&sc->rsel);
```

Looks fine. The wakeup is there. The `selwakeup` is there.

Step 5: trace deeper. Maybe the wakeup is being called before the bytes are actually in the buffer. In the original code, the order was: add bytes under lock, unlock, wakeup. In the modified code, you inserted `update_stats` between the unlock and the wakeup. That should be fine; the bytes are in the buffer, just waiting for a waker to call wakeup.

But wait: `update_stats` is new. What does it do?

```
static void
update_stats(struct myfirst_softc *sc, size_t n)
{
    mtx_lock(&sc->mtx);
    sc->aux_counter++;
    mtx_unlock(&sc->mtx);
}
```

It takes the mutex. And then releases it. That is a full acquire/release cycle. Could it change anything?

Actually, look closer. In the write path, the sequence is:

1. Take mutex.
2. Write bytes to `cbuf`.
3. Release mutex.
4. Call `update_stats`, which re-acquires and re-releases.
5. Call wakeup.

Between steps 3 and 5, there is a window where a reader could acquire the mutex, observe the new bytes, and proceed. In most cases this is fine: the reader does not care about the auxiliary statistics. But if the reader is currently inside `mtx_sleep`, waiting for wakeup, it is stuck until step 5.

That looks normal; step 5 always runs. So the wakeup does fire. But why is the reader not waking?

Step 6: add a `device_printf` at the start of `update_stats` and at the wakeup call:

```
device_printf(sc->dev, "update_stats called\n");
/* ... */
device_printf(sc->dev, "wakeup(&sc->cb) called\n");
wakeup(&sc->cb);
```

Reload, retest. Watch dmesg:

```
myfirst0: update_stats called
```

Only one message. The wakeup is never called.

Step 7: look at update_stats again. Could it have a path that returns early?

```
static void
update_stats(struct myfirst_softc *sc, size_t n)
{
    if (n == 0)
        return;
    mtx_lock(&sc->mtx);
    sc->aux_counter++;
    mtx_unlock(&sc->mtx);
}
```

Ah, a short-circuit for `n == 0`. What if put was zero for some reason? Then update_stats returns. The caller continues, but wait, wakeup is still after update_stats, so it should fire anyway.

Unless the caller's code also has a short-circuit:

```
update_stats(sc, put);
if (put < want)
    break;
wakeup(&sc->cb);
```

That's the bug. The early break skips the wakeup. Under most conditions `put == want`, and the wakeup runs. Under the rare condition where `put < want` (for example, the buffer filled up during this iteration), the wakeup is skipped. A reader waiting for data never sees the bytes that did make it in.

Step 8: fix it. Move the wakeup before the break:

```
update_stats(sc, put);
wakeup(&sc->cb);           /* must happen even on short write */
selwakeup(&sc->rsl);
if (put < want)
    break;
```


Test again. The hang is gone.

This is a realistic debugging story. The bug is not in the concurrency machinery; it is in the business logic that controls when the wakeup fires. The fix is local. The debugging process was: observe, narrow, trace, inspect, fix.

11.8.8 Patterns for Adding Debug Traces

When you add debug traces to a driver, several patterns pay off.

Traces behind a debug flag. We saw this in Chapter 10: a `myfirst_debug` integer controlled by `sysctl`, and a `MYFIRST_DBG` macro that compiles to nothing when the flag is zero. With the flag off, traces have no cost; with it on, they emit to `dmesg`. This lets you ship the driver with traces included and turn them on only when needed.

One line per meaningful event. The temptation is to trace every byte transferred. Resist it. Trace once per handler invocation, not once per byte. Trace once per lock acquisition during debugging of a specific bug, then remove. A log flooded with lines tells you nothing.

Include the thread ID. `curthread->td_tid` is the current thread's ID. Printing it in traces lets you distinguish concurrent activities in the log. Useful format: `device_printf(dev, "tid=%d got=%zu\n", curthread->td_tid, got)`.

Include before/after state. For a state change, log both the old and new values. `device_printf(dev, "cb_used: %zu -> %zu\n", before, after)` is more useful than `cb_used: %zu`, because you can see the transition.

Remove or guard before shipping. Trace lines are development aids. Either remove them or put them behind `MYFIRST_DBG` before the code is deployed. A shipping driver that emits debug output per I/O is wasting the log buffer.

11.8.9 Debugging the Hard Cases

Some concurrency bugs resist all the techniques we have discussed. They reproduce on a specific machine, with a specific workload, on a specific kernel, and nowhere else. When you encounter such a bug, the options are:

Bisect the workload. If the bug occurs with *X* concurrent readers and *Y* concurrent writers, try to reduce *X* and *Y* until the bug no longer occurs. The minimal reproducer is the easiest to reason about.

Bisect the kernel. If the bug occurs on version *N* but not on *N-1*, find the change in the kernel that introduced it. `git bisect` is the tool. This is slow but effective.

Inspect the hardware. Some bugs are caused by specific CPU features (cache coherency quirks, TSO behavior, weakly-ordered memory models). If the bug occurs on ARM64 but not amd64, memory ordering is suspect.

Ask on the mailing lists. The FreeBSD community includes many people who have seen similar bugs. `freebsd-hackers` and `freebsd-current` lists welcome detailed bug reports. The more information you provide (exact kernel version, workload, failure mode, what you have tried), the more likely someone will recognize the pattern.

Concurrency debugging is, ultimately, a skill. The tools help, but the skill is built by doing it, writing tests that fail, writing fixes, and confirming the tests pass. The chapters ahead will give you more opportunities.

11.8.10 Wrapping Up Section 7

Debugging concurrency is a trade of patience for tooling. `INVARIANTS`, `WITNESS`, `KASSERT`, `mtx_assert`, and `dtrace` are the tools FreeBSD gives you. Used in combination, they can corner bugs that would otherwise go undetected until production.

Section 8 closes the chapter with the housekeeping: refactoring the driver, versioning it, documenting it, running static analysis, and regression-testing the whole lot.

11.9 Section 8: Refactoring and Versioning Your Concurrent Driver

The driver now has well-understood concurrency semantics and a locking strategy that can be defended from first principles. This final section of the chapter is the hygiene pass: organizing the code for clarity, versioning the driver so future-you can tell what changed when, writing a README that documents the design, running static analysis, and regression-testing the whole thing.

This is not glamorous work. It is also what separates a driver that is delivered once from a driver that lives usefully for years.

11.9.1 8.1: Organizing the Code for Clarity

The Stage 4 driver from Chapter 10 was already well-organized. The `cbuf` is in its own file. The I/O handlers use helper functions. The locking discipline is documented in a comment at the top. For Chapter 11, the organizational work is marginal: improving commenting, grouping related code, and ensuring naming consistency.

Three improvements are worth making.

Group related code. Reorder the functions in `myfirst.c` so related ones live next to each other. Lifecycle (`attach`, `detach`, `modevent`) at the top; I/O handlers (`read`, `write`, `poll`) in the middle; helpers and `sysctl` callbacks near the end. The compile order does not matter; what matters is that a reader scrolling through the file encounters related functions together.

Isolate locking into inline wrappers. Instead of repeating `mtx_lock(&sc->mtx)` and `mtx_unlock(&sc->mtx)` throughout the code, define:

```
#define MYFIRST_LOCK(sc)      mtx_lock(&(sc)->mtx)
#define MYFIRST_UNLOCK(sc)    mtx_unlock(&(sc)->mtx)
#define MYFIRST_ASSERT(sc)    mtx_assert(&(sc)->mtx, MA_OWNED)
```

Use the macros throughout. If you ever need to change the lock type (for example, to an `sx` lock in a future refactor), you change one place, not twenty.

Name things consistently. All driver-level buffer helpers are `myfirst_buf_*`. All wait helpers are `myfirst_wait_*`. All `sysctl` handlers are `myfirst_sysctl_*`. A reader scanning the function names can tell what category each function falls into without reading the body.

None of these is a correctness improvement. All of them are readability improvements that pay off when you come back to the code in six months.

11.9.2 8.2: Versioning the Driver

The driver should expose its version so that you can tell at load time what you are running. Add a version string:

```
#define MYFIRST_VERSION "0.4-concurrency"
```

Print it at attach:

```
device_printf(dev,
    "Attached; version %s, node /dev/%s (alias /dev/myfirst), "
    "cbuf=%zu bytes\n",
    MYFIRST_VERSION, devtoname(sc->cdev), cbuf_size(&sc->cb));
```

Expose it as a read-only sysctl:

```
SYSCTL_STRING(_hw_myfirst, OID_AUTO, version, CTLFLAG_RD,
    MYFIRST_VERSION, 0, "Driver version");
```

A user can now query `sysctl hw.myfirst.version` or check `dmesg` to confirm which version is loaded. When debugging, this removes any doubt about which code is running.

Pick a versioning scheme and stick with it. Semantic versioning (major.minor.patch) is fine. Date-based (2026.04) is fine. Book-chapter-based (0.1 after Chapter 7, 0.2 after Chapter 8, 0.3 after Chapter 9, 0.4 after Chapter 10, 0.5 after Chapter 11) is fine and is what the companion source uses. The important property is consistency, not specifics.

For Chapter 11 specifically, the appropriate version bump is to 0.5-concurrency. The changes are: counter(9) migration, KASSERTs added, LOCKING.md added, annotations added. All are safety and clarity changes; none is a behavioural change visible from user space. Document them in a `CHANGELOG.md`:

myfirst Changelog

0.5-concurrency (Chapter 11)

- Migrated `bytes_read`, `bytes_written` to `counter_u64_t` (lock-free).
- Added KASSERTs throughout `cbuf_*` helpers.
- Added LOCKING.md documenting the locking strategy.
- Added source annotations naming each critical section.
- Added MYFIRST_LOCK/UNLOCK/ASSERT macros for future lock changes.

0.4-poll-refactor (Chapter 10, Stage 4)

- Added `d_poll` and `selinfo`.
- Refactored I/O handlers to use wait helpers.
- Added locking-strategy comment.

0.3-blocking (Chapter 10, Stage 3)

- Added mtx_sleep-based blocking read/write paths.
- Added IO_NDELAY -> EAGAIN handling.

0.2-circular (Chapter 10, Stage 2)

- Replaced linear FIFO with cbuf circular buffer.

0.1 (Chapter 9)

- Initial read/write via uiomove.

A CHANGELOG.md you can consult beats git history when you want the quick answer. Keep it updated with each change.

11.9.3 8.3: README and Comment Pass

Alongside LOCKING.md, write a README.md for the driver. The audience is a future maintainer (possibly you) who has just checked out the source and needs to know what the project is.

A minimal version:

myfirst

A FreeBSD 14.3 pseudo-device driver that demonstrates buffered I/O, concurrency, and modern driver conventions. Developed as the running example for the book "FreeBSD Device Drivers: From First Steps to Kernel Mastery."

Status

Version 0.5-concurrency (Chapter 11).

Features

- A Newbus pseudo-device under nexus0.
- A primary device node at /dev/myfirst/0 (alias: /dev/myfirst).
- A circular buffer (cbuf) as the I/O buffer.
- Blocking and non-blocking reads and writes.
- poll(2) support via d_poll and selinfo.
- Per-CPU byte counters via counter(9).
- A single sleep mutex protects composite state; see LOCKING.md.

Build and Load

\$ make

```
# kldload ./myfirst.ko
# dmesg | tail
# ls -l /dev/myfirst
# printf 'hello' > /dev/myfirst
# cat /dev/myfirst
# kldunload myfirst
```

Tests

See ../../userland/ for the test programs. The most useful one is `producer_consumer`, which exercises the round-trip correctness of the circular buffer.

License

BSD 2-Clause. See individual source files for SPDX headers.

Every driver benefits from a README like this. Without it, a new maintainer (possibly you, six months from now) has to reverse-engineer what the project is. With it, the onboarding is minutes.

In parallel, do a comment pass on the source itself. Focus on the critical sections: every `mtx_lock` should have a brief comment naming the shared state it is about to protect. Every helper should have a one-line description above the function definition. Every non-obvious piece of arithmetic (cbuf wrap-around, `nbefore` comparison, etc.) should have a sentence of explanation.

The goal is not exhaustive commentary. It is to make the code self-explanatory to a reader who has not seen it before.

11.9.4 8.4: Static Analysis

FreeBSD's base system includes `clang`, which has a `--analyze` mode that performs static analysis without compiling. For a kernel module, invoke it via:

```
$ make WARNS=6 CFLAGS+="-Weverything -Wno-unknown-warning-option" clean all
```

Or, more directly:

```
$ clang --analyze -I/usr/src/sys -I/usr/src/sys/amd64/conf/GENERIC \
-D_KERNEL myfirst.c
```

The output is a list of potential issues with file-line annotations. Triage them: false positives (clang does not understand some kernel idioms) are fine to ignore; genuine issues (uninitialized variables, null-pointer dereferences, memory leaks) deserve fixes.

Add a `lint` target to your `Makefile`:

```
.PHONY: lint
lint:
    cd ${.CURDIR}; clang --analyze -D_KERNEL *.c
```

Run `make lint` periodically. A clean run is the baseline; any new warning deserves attention before being merged.

11.9.5 8.5: Regression Testing

Assemble a regression test that runs every verification you have built. A shell script in the `tests/` subdirectory:

```
#!/bin/sh
# regression.sh: run every test from Chapters 7-11 in sequence.

set -eu

die() { echo "FAIL: $" ">&2; exit 1; }
ok()  { echo "PASS: $*"; }

# Preconditions
[ $(id -u) -eq 0 ] || die "must run as root"
kldstat | grep -q myfirst && kldunload myfirst
[ -f ./myfirst.ko ] || die "myfirst.ko not built; run make first"

kldload ./myfirst.ko
trap 'kldunload myfirst 2>/dev/null || true' EXIT

sleep 1
[ -c /dev/myfirst ] || die "device node not created"
ok "load"

printf 'hello' > /dev/myfirst || die "write failed"
cat /dev/myfirst >/tmp/out.$$
[ "$(cat /tmp/out.$$)" = "hello" ] || die "round-trip content mismatch"
rm -f /tmp/out.$$
ok "round-trip"

cd ../userland && make -s clean && make -s && cd -

../userland/producer_consumer || die "producer_consumer failed"
ok "producer_consumer"

../userland/mp_stress || die "mp_stress failed"
```

```
ok "mp_stress"

sysctl dev.myfirst.0.stats >/dev/null || die "sysctl not accessible"
ok "sysctl"

echo "ALL TESTS PASSED"
```

Run this after every change:

```
# ./tests/regression.sh
PASS: load
PASS: round-trip
PASS: producer_consumer
PASS: mp_stress
PASS: sysctl
ALL TESTS PASSED
```

A green regression is the evidence that the change did not break anything. Pair it with the debug-kernel stress runs and you have a reasonable quality gate for a driver at this stage of the book.

11.9.6 A Complete LOCKING.md Template

For readers who want a more substantive starting template than the minimal one in Section 3, here is a fuller LOCKING.md that can be adapted to any driver:

```
# <driver-name> Locking Strategy

## Overview

One sentence describing the overall approach. For example:
"The driver uses a single sleep mutex to serialize all access to
shared state, with separate per-CPU counters for hot-path
statistics."

## Locks Owned by This Driver

### sc->mtx (mutex(9), MTX_DEF)

**Protects:**
- sc->cb (circular buffer state: cb_head, cb_used, cb_data)
- sc->active_fhs
- sc->is_attached (writes)
- sc->rsel, sc->wsel (indirectly: selrecord inside a critical
```

```
section; selwakeup outside)
```

```
**Wait channels used under this mutex:**
```

```
- &sc->cb: data available / space available
```

Locks Owned by Other Subsystems

```
- selinfo internal locks: handled by the selinfo API; we must  
  call selrecord under our lock and selwakeup outside.
```

Unprotected Accesses

sc->is_attached (reads at handler entry)

```
**Why it's safe:**
```

A stale "true" is re-checked after every sleep; a stale "false" merely causes the handler to return ENXIO early, which is harmless since the device really is gone.

sc->bytes_read, sc->bytes_written (counter(9) fetches)

```
**Why it's safe:**
```

counter(9) handles its own consistency internally. Fetches may be slightly stale but are never torn.

Lock Order

The driver currently holds only one lock. No ordering rules apply.

When new locks are added, document the order here before merging the change. The format is:

```
sc->mtx -> sc->other_mtx
```

meaning: a thread holding sc->mtx may acquire sc->other_mtx, but not the reverse.

Rules

1. Never sleep while holding sc->mtx except via mtx_sleep, which atomically drops and reacquires.
2. Never call uiomove, copyin, copyout, selwakeup, or wakeup while holding sc->mtx.
3. Every cbuf_* call must happen with sc->mtx held (the helpers


```
assert this with mtx_assert).
```

4. The detach path clears `sc->is_attached` under the mutex and then wakes any sleepers before destroying the mutex.

History

- 0.5-concurrency (Chapter 11): migrated byte counters to `counter(9)`; added `mtx_assert` calls; formalized the strategy.
- 0.4-poll-refactor (Chapter 10): added `d_poll`, refactored into helpers, documented the initial strategy inline in the source.
- Earlier versions: strategy was implicit in the code.

A document this shape is auditable. Any future change to the driver's concurrency story touches this file in the same commit. The review diff shows the rule change, not just the code change.

11.9.7 A Pre-Commit Checklist

Before committing a concurrency-affecting change, run through this checklist:

1. **Have I updated `LOCKING.md`?** If the change adds a field to shared state, names a new lock, changes lock ordering, adds a wait channel, or changes a rule, update the doc.
2. **Have I run the full regression suite?** On a debug kernel, with `WITNESS` and `INVARIANTS`. A green regression is the minimum.
3. **Have I run the stress tests?** Not just once; for long enough to matter. Thirty minutes is a reasonable minimum for a significant change.
4. **Have I run `clang --analyze`?** Treat new warnings as bugs.
5. **Have I added a `KASSERT` for any new invariant?** Every invariant your code assumes should be a `KASSERT`.
6. **Have I bumped the version string and updated `CHANGELOG.md`?** Even if the change is small. Future you will thank present you.
7. **Have I verified the test kit compiles?** The test programs are part of the change; they should not bitrot.

Most changes clear the checklist in minutes. The changes that do not are the ones that should be caught by the checklist: those are the ones most likely to regress.

11.9.8 Versioning Strategy: Why 0.x?

We are using a 0.x scheme for the driver because the driver is not yet feature-complete. A 1.0 release implies stability and completeness that we cannot claim. The 0.x versions track our progress through the book:

- 0.1: basic scaffold (Chapter 7).
- 0.2: device file and per-open state (Chapter 8).
- 0.3: basic read/write via `uiomove` (Chapter 9).
- 0.4: buffered I/O, non-blocking, poll (Chapter 10).
- 0.5: concurrency hardening (Chapter 11).

Chapter 12 will introduce 0.6. Eventually, when the driver is feature-complete and stable, we would call it 1.0.

This is not the only sensible scheme. Semantic versioning (MAJOR.MINOR.PATCH) is appropriate for drivers with a stable API. Date-based versioning (2026.04) is appropriate for drivers where the release cadence is temporal. The choice matters less than the commitment: pick one, apply it consistently, document it in the README.

11.9.9 A Note on Git Hygiene

The companion source tree is organized so that each chapter's stage lives in its own directory. You can build any stage independently; you can diff stage-over-stage to see what changed. If you are using git, a clean commit history further enhances this:

- One commit per logical change.
- Commit messages that explain *why*, not just *what*.
- A separate commit for LOCKING.md updates when the code changes.

A git log like:

```
0.5 add KASSERT for cbuf_write overflow invariant
0.5 migrate bytes_read, bytes_written to counter(9)
0.5 document locking strategy in LOCKING.md
0.5 add MYFIRST_LOCK/UNLOCK/ASSERT macros
0.4 refactor I/O handlers to use wait helpers
```

is readable years later. A log like:

```
fix stuff
more fixes
wip
```

is not. Your driver is a living artifact; treat its history as documentation.

11.9.10 Wrapping Up Section 8

The driver is now not only correct but *verifiably* correct and *maintainably* organized. It has:

- A documented locking strategy in LOCKING.md.
- A versioning scheme and a changelog.
- A README for future maintainers.
- Source annotations for every critical section.
- Static analysis via `clang --analyze`.
- A regression test that runs everything.

That is a substantial body of infrastructure. Most beginner drivers have none of it. Yours now does, and the work to maintain it is routine rather than heroic.

11.10 Hands-on Labs

These labs consolidate the concepts of Chapter 11 through direct, hands-on experience. They are ordered from least to most demanding. Each is designed to be completable in a single lab session.

11.10.1 Pre-Lab Setup Checklist

Before starting any lab, confirm the four items below. Two minutes of preparation here will save you significant time when something goes wrong inside a lab and you cannot tell whether the problem is your code, your environment, or your kernel.

1. **Debug kernel running.** Confirm with `sysctl kern.ident` that the booted kernel is the one with `IN-variants` and `WITNESS` enabled. The reference section “Building and Booting a Debug Kernel” later in this chapter walks through the build, install, and reboot if you have not already done it.
2. **WITNESS active.** Run `sysctl debug.witness.watch` and confirm a non-zero value. If the `sysctl` is missing, your kernel does not have `WITNESS` compiled in; rebuild before continuing.
3. **Driver source matches Chapter 10 Stage 4.** From your driver directory, make `clean` && `make` should compile cleanly. The `myfirst.ko` artefact must exist before any lab attempts to load it.
4. **A clean dmesg.** Run `dmesg -c >/dev/null` once before the first lab so any prior warnings do not confuse you. Subsequent `dmesg` checks then show only what your lab produced.

If any of the four is uncertain, fix it now. The labs assume all four.

11.10.2 Lab 11.1: Observe a Race Condition

Goal. Build the race-demo driver from Section 2, run it, and observe the corruption. The purpose is not to produce a correct driver; it is to see with your own eyes what “no synchronization” means.

Steps.

1. Create `examples/part-03/ch11-concurrency/stage1-race-demo/`.
2. Copy `cbuf.c`, `cbuf.h`, and the `Makefile` from the Chapter 10 Stage 4 directory.
3. Copy `myfirst.c` and add the `RACE_DEMO` macros at the top as shown in Section 2. Replace every `mtx_lock`, `mtx_unlock`, and `mtx_assert` in the I/O paths with the `MYFIRST_LOCK`, `MYFIRST_UNLOCK`, `MYFIRST_ASSERT` macros. Compile with `-DRACE_DEMO` (which is the default in this directory’s `Makefile`).
4. Build and load.
5. In one terminal, run `../part-02/ch10-handling-io-efficiently/userland/producer_consumer`.
6. Observe the output. Record the mismatch count.
7. Run it several times. Observe that the mismatch count varies.
8. Unload the driver.

Verification. Every run produces at least one mismatch. The exact number varies. This demonstrates that concurrency bugs are not a theoretical possibility; they are the certainty when locks are absent.

Stretch goal. Add a `device_printf` inside `cbuf_write` that logs when `cb_used` exceeds `cb_size`. Load the driver, run the test, and observe the log messages. Each message is an invariant violation caught in flight.

11.10.3 Lab 11.2: Verify the Locking Discipline with INVARIANTS

Goal. Convert your Chapter 10 Stage 4 driver to use the MYFIRST_LOCK macros and add `mtx_assert` calls throughout. Run it under a WITNESS-enabled kernel and confirm that no warnings appear.

Steps.

1. Copy your Stage 4 driver into `examples/part-03/ch11-concurrency/stage2-concurrent-safe/`.
2. Add the MYFIRST_LOCK, MYFIRST_UNLOCK, MYFIRST_ASSERT macros at the top. Make them expand to the real `mtx_*` calls (no RACE_DEMO).
3. Replace the `mtx_lock`, `mtx_unlock`, `mtx_assert` calls in the I/O paths with the macros.
4. Add MYFIRST_ASSERT(sc) at the top of every helper that should be called with the lock held.
5. Build with `WARNS=6`.
6. Load on a WITNESS-enabled kernel.
7. Run the Chapter 10 test kit (`producer_consumer`, `rw_myfirst_nb`, `mp_stress`).
8. Confirm no WITNESS warnings appear in `dmesg`.

Verification. `dmesg | grep -i witness` shows no output related to `myfirst`. Every test passes.

Stretch goal. Introduce a deliberate bug: remove one `mtx_unlock` call, leaving the lock held past the end of the handler. Observe the WITNESS warning that appears, confirm it identifies the exact lock and the approximate line, then restore the unlock.

11.10.4 Lab 11.3: Migrate to counter(9)

Goal. Replace the `sc->bytes_read` and `sc->bytes_written` fields with `counter(9)` per-CPU counters.

Steps.

1. Copy Lab 11.2's source into `stage3-counter9/`.
2. Change the struct definition so the two fields are `counter_u64_t`.
3. Update `myfirst_attach` to allocate them and `myfirst_detach` to free them.
4. Update `myfirst_buf_read` and `myfirst_buf_write` to use `counter_u64_add`.
5. Remove the field updates from under the mutex (they no longer need the mutex).
6. Update the `sysctl` handlers to use `counter_u64_fetch`.
7. Update `LOCKING.md` to note that `bytes_read` and `bytes_written` are no longer under the mutex.
8. Build, load, and run the test kit.

Verification. `sysctl dev.myfirst.0.stats.bytes_read` and `bytes_written` still return correct values. No test fails.

Stretch goal. Run `mp_stress` with 8 writers and 8 readers for 60 seconds on a multi-core machine. Compare throughput with Lab 11.2 (mutex-protected counters). The per-CPU counters should be measurably faster if the machine has at least 4 cores.

11.10.5 Lab 11.4: Build a Multi-threaded Tester

Goal. Build `mt_reader` from Section 6. Use it to exercise the driver with multiple threads in one process.

Steps.

1. Under `examples/part-03/ch11-concurrency/userland/`, type `mt_reader.c` from Section 6, or open the companion source file already present in that directory and read it carefully.
2. From the same directory, run `make mt_reader` (the Makefile already includes the right `-pthread` linkage). The standalone command is `cc -Wall -Wextra -pthread -o mt_reader mt_reader.c`.
3. Start a writer in another terminal: `dd if=/dev/zero of=/dev/myfirst bs=4k &`.
4. Run `./mt_reader`.
5. Confirm that each thread sees a nonzero byte count and that the sum is consistent with what the writer produced.
6. When you are done, stop the background `dd` with `kill %1` (or whatever job number it has).

Verification. All four threads report a nonzero count. The grand total equals the writer’s byte count minus the buffer fill at the end.

Stretch goal. Increase `NTHREADS` to 8, 16, 32. Observe the scaling behavior. Does throughput increase, decrease, or remain flat? Explain why (hint: Section 5’s discussion of lock contention).

11.10.6 Lab 11.5: Deliberately Cause a WITNESS Warning

Goal. Introduce a deliberate concurrency bug and observe how `WITNESS` catches it.

Steps.

1. Copy your Stage 3 working directory into a scratch directory such as `stage-lab11-5/`. Do *not* edit Stage 3 in place; the purpose of the lab is to revert cleanly.
2. In `myfirst_read`, move the `MYFIRST_UNLOCK(sc)` call from before `uiomove` to after `uiomove`. The code still compiles, but the mutex is now held across the `uiomove`, which may sleep.
3. Build and load on a `WITNESS`-enabled debug kernel (see the “Building and Booting a Debug Kernel” reference later in this chapter).
4. Run `producer_consumer` in one terminal and watch `dmesg` in another.
5. Observe the warning emitted by `WITNESS` when the `uiomove` is about to sleep while `sc->mtx` is held. Record the exact message and the source line it cites.
6. Unload the module, delete the scratch directory, and confirm that your Stage 3 source is still correct.

Verification. `dmesg` shows a warning similar to “acquiring duplicate lock of same type” or “sleeping thread (pid N) owns a non-sleepable lock” pointing at the line where the unlock used to live.

Stretch goal. Repeat with a different deliberate bug: change the mutex type from `MTX_DEF` to `MTX_SPIN` in `mtx_init`. Observe the resulting warnings, which should be about sleeping while holding a spin mutex. Again, revert cleanly when you are done.

11.10.7 Lab 11.6: Add KASSERTs Throughout

Goal. Add `KASSERT` calls to the `cbuf` helpers and the driver code. Confirm they fire when deliberately triggered.

Steps.

1. Copy Lab 11.3’s source into `stage5-kasserts/`.
2. In `cbuf_write`, add `KASSERT(cb->cb_used + n <= cb->cb_size, ...)` at the top.

3. In `cbuf_read`, add `KASSERT(n <= cb->cb_used, ...)`.
4. In `myfirst_buf_*` helpers, add `mtx_assert(&sc->mtx, MA_OWNED)`.
5. Build and load on an INVARIANTS kernel.
6. Run the test kit. No assertions should fire in normal operation.
7. Introduce a bug: in `cbuf_write`, write `cb->cb_used += n + 1` instead of `cb->cb_used += n`. Confirm that the `KASSERT` in the next call catches the overflow.
8. Revert the bug.

Verification. Every assertion in the reverted source compiles and the driver works normally. The deliberate bug produces a panic with a clear backtrace that names the failed `KASSERT`.

11.10.8 Lab 11.7: Long-Running Stress

Goal. Run the full Chapter 11 test kit for one hour and verify no failures, no kernel warnings, and no memory growth.

Steps.

1. Load your Chapter 11 Stage 3 or Stage 5 driver.
2. In one terminal, run:

```
for i in $(seq 1 60); do
    ./producer_consumer
    ./mp_stress
done
```

3. In a second terminal, run a background `vmstat 10` and `sysctl dev.myfirst.0.stats` periodically.
4. Monitor `dmesg` for warnings.
5. After the loop finishes, check `vmstat -m | grep cbuf` and confirm memory has not grown.

Verification. All loops complete. No `WITNESS` warnings. `vmstat -m | grep cbuf` shows the expected constant allocation. The byte counters in `sysctl` have increased; their ratio of `bytes_read` to `bytes_written` is approximately 1.

11.10.9 Lab 11.8: Run `clang -analyze`

Goal. Run static analysis on the driver and triage the results.

Steps.

1. From your Chapter 11 Stage 3 or Stage 5 directory, run:

```
clang --analyze -D_KERNEL -I/usr/src/sys \
    -I/usr/src/sys/amd64/conf/GENERIC myfirst.c
```

2. Record every warning.
3. For each warning, classify it as (a) a true positive (real bug), (b) a false positive that you understand, or (c) a warning you do not understand.

4. For (a), fix the bug. For (b), document why it is a false positive. For (c), research until it moves to (a) or (b).

Verification. After the pass, the driver has zero unclassified warnings. The `LOCKING.md` or `README.md` documents any known false positives.

11.11 Challenge Exercises

The challenges extend Chapter 11's concepts beyond the baseline labs. Each is optional; each is designed to deepen your understanding.

11.11.1 Challenge 1: A Lock-Free Producer/Consumer

The mutex we use protects composite state. A single-producer-single-consumer queue can be implemented lock-free, using only atomics, because the writer touches only the tail and the reader touches only the head, and neither touches the fields the other updates. `buf_ring(9)` builds on this idea but goes further: it uses compare-and-swap so several producers can enqueue concurrently, and offers either single-consumer or multi-consumer dequeue paths chosen at allocation time.

Read `/usr/src/sys/sys/buf_ring.h`, where the lock-free enqueue and dequeue paths live as inline functions, and `/usr/src/sys/kern/subr_bufring.c`, which holds `buf_ring_alloc()` and `buf_ring_free()`. Understand the `buf_ring(9)` primitive. Then, as an exercise, build a variant of the driver where the circular buffer is replaced by a `buf_ring`. The read and write handlers no longer need a mutex for the ring itself; they may still need one for other state.

This challenge is harder than it looks. Attempt it only if you are comfortable with Chapter 11's material and want a stretch.

11.11.2 Challenge 2: Multi-Mutex Design

Split `sc->mtx` into two locks: `sc->state_mtx` (protects `is_attached`, `active_fhs`, `open_count`) and `sc->buf_mtx` (protects the cbuf and byte counters). Define a lock order (for example, state before buf) and document it in `LOCKING.md`. Update all the handlers.

What do you gain? What do you lose? Is there a workload that measurably benefits?

11.11.3 Challenge 3: WITNESS Under Pressure

Construct a test that provokes a `WITNESS` warning even on a correctly-locked driver. (Hint: `sx` locks have different rules; mixing a mutex and an `sx` lock in conflicting orders produces warnings. Chapter 12 covers `sx`.) This challenge is a preview of Chapter 12 material.

11.11.4 Challenge 4: Instrument with `dtrace`

Write a `dtrace` script that:

- Counts the number of `mtx_lock(&sc->mtx)` calls per second.
- Prints a histogram of the time spent holding the lock.
- Correlates lock-holding time with `bytes_read` and `bytes_written` rates.

Run the script while `mp_stress` is running. Produce a report.

11.11.5 Challenge 5: Bounded Blocking

Modify `myfirst_read` to return `EAGAIN` if the read would block for more than `N` milliseconds, where `N` is a `sysctl-tunable` parameter. (Hint: the `timo` argument of `mtx_sleep` is in ticks, where `hz` ticks equal one second. For example, `hz / 10` is roughly 100 milliseconds. For sub-tick precision, the `msleep_sbt(9)` family uses `sbintime_t` and the `SBT_1S`, `SBT_1MS`, and `SBT_1US` constants from `sys/time.h`.) This challenge previews some of Chapter 12's material on timeouts.

11.11.6 Challenge 6: Write a Lock-Contention Benchmark

Build a benchmark that measures the mutex acquisition rate under varying load:

- 1 writer, 1 reader.
- 1 writer, 4 readers.
- 4 writers, 4 readers.
- 8 writers, 8 readers.

Report acquisition-per-second, average hold time, and maximum wait time. Use `dt race` or the mutex profiling `sysctl`.

11.11.7 Challenge 7: Port to a 32-bit Platform

FreeBSD still supports 32-bit architectures (`i386`, `armv7`). Build your driver for one of them and run the test kit. Do any of the torn-read issues from Section 3 manifest? Is the per-CPU counter migration from Lab 11.3 necessary for correctness, or merely for performance?

(This challenge is genuinely hard and requires access to 32-bit hardware or a cross-build environment.)

11.11.8 Challenge 8: Add a Per-Descriptor Mutex

The per-descriptor struct `myfirst_fh` has reads and writes counters that are currently updated under the device-wide `sc->mtx`. An alternative is to give each `fh` its own mutex. Would this help? Hurt? Write the change, run the benchmark, report.

11.12 Troubleshooting

This reference catalogues the bugs you are most likely to encounter while working through Chapter 11 and after. Each entry has a symptom, a cause, and a fix.

11.12.1 Symptom: `mtx_assert` panic on a driver that used to work

Cause. A helper was called without the mutex held. This usually happens when you refactor and move a call outside the critical section.

Fix. Look at the backtrace; the top frame names the helper. Find the caller. Ensure the caller holds `sc->mtx` at the point of the call.

11.12.2 Symptom: sleepable after non-sleepable acquired warning

Cause. A call that might sleep (typically `uiomove`, `copyin`, `copyout`, `malloc(... M_WAITOK)`) was made while holding a non-sleepable lock. On FreeBSD, `MTX_DEF` is sleepable; the warning is usually more specific.

Fix. Drop the mutex before the sleeping call. If state must be preserved, take a snapshot under the mutex, do the sleeping call outside, and reacquire to commit.

11.12.3 Symptom: Lock-order reversal

Cause. Two acquisition orders observed; one is the reverse of the other. Only possible with two or more locks.

Fix. Define a global order. Update the offending path. Document in `LOCKING.md`.

11.12.4 Symptom: Deadlock (hang)

Cause. Either a missing wakeup (`&sc->cb`) or a sleep with a different channel than the wake.

Fix. Check every `mtx_sleep` channel against every wakeup channel. They must match exactly. `&sc->cb` sleeping and `&sc->buf` waking are different channels.

11.12.5 Symptom: Torn value read from a `uint64_t`

Cause. On a 32-bit platform, a 64-bit read is two 32-bit reads, which can straddle a concurrent write. On amd64, aligned 64-bit accesses are atomic; on 32-bit platforms, they are not.

Fix. Either protect with the mutex (simple) or migrate to `counter(9)` (scalable) or use the `atomic*_64` primitives (precise).

11.12.6 Symptom: Double-free panic on the `cbuf`

Cause. `cbuf_destroy` called twice, typically because the error path in `myfirst_attach` runs `cbuf_destroy` and then `myfirst_detach` runs it again.

Fix. After calling `cbuf_destroy`, set `sc->cb.cb_data = NULL` so a second call is a no-op. Or guard the destroy with an explicit flag.

11.12.7 Symptom: Slow performance under many-core load

Cause. Mutex contention. Every CPU is serializing on `sc->mtx`.

Fix. Migrate byte counters to `counter(9)` (Lab 11.3). If further reduction is needed, split the mutex (Challenge 2) or use finer-grained primitives (Chapter 12).

11.12.8 Symptom: `producer_consumer` occasionally hangs

Cause. A race between the writer finishing and the reader deciding the buffer is “permanently empty”. Our driver returns zero bytes on an empty non-blocking read, which `producer_consumer` may interpret as EOF.

Fix. Use blocking reads in `producer_consumer` (remove `O_NONBLOCK` if it was added). Ensure the writer closes the descriptor or the reader detects end-of-test by total byte count, not by a zero-byte read.

11.12.9 Symptom: `WITNESS` reports “unowned lock released”

Cause. `mtx_unlock` called without a matching `mtx_lock` earlier in the same thread.

Fix. Audit every `mtx_unlock` site. Trace back and confirm each has a matching `mtx_lock` before it, along every path.

11.12.10 Symptom: Compiles cleanly, passes single-threaded tests, fails `mp_stress`

Cause. A missed lock somewhere on the data path. Single-threaded tests never exercise the race; concurrent tests do.

Fix. Audit every access to shared state, as in Section 3. The usual culprit is a field updated outside the critical section because “it’s only a counter” (not true under concurrency).

11.12.11 Symptom: `kldunload` hangs with descriptors open

Cause. The detach path refuses to proceed while `active_fhs > 0`.

Fix. `fstat | grep myfirst` to find the offending processes. kill them. Then retry `kldunload`.

11.12.12 Symptom: Kernel panic with a clean backtrace in `mtx_lock`

Cause. The mutex was destroyed and its memory reused. The atomic operation in `mtx_lock` is operating on whatever happens to be in that memory now.

Fix. Ensure `mtx_destroy` is called only after every handler that could use the mutex has returned. Our detach path does this by destroying the `cdev` first, which waits for in-flight handlers.

11.12.13 Symptom: WITNESS never warns, even when you expect it to

Cause. Three possibilities. First, the running kernel does not actually have WITNESS compiled in; check `sysctl debug.witness.watch` and confirm a non-zero value. Second, WITNESS is enabled but quiet because no rule has been violated yet; not every refactor produces a violation. Third, the offending code path was not exercised; WITNESS only checks the orderings the kernel actually observes at runtime.

Fix. Confirm the kernel has WITNESS (the `sysctl` above is the easy check). Then drive the suspect code path with the multi-process and multi-thread testers from Section 6 so the kernel sees the acquisitions in question.

11.12.14 Symptom: dmesg is silent but the test still fails

Cause. Either the failure is in user-space (the test program) or the kernel did detect a problem but printed nothing because the message buffer wrapped. The latter is rare on a freshly-booted system but common on long-running test machines.

Fix. Increase the buffer for next boot by adding `kern.msgbufsize="4194304"` to `/boot/loader.conf`; the `sysctl` is read-only at runtime. For user-space failures, run the test under `truss(1)` to see the syscalls and their return values; concurrency bugs at the user-space level often surface as EAGAIN, EINTR, or short reads that the test mishandles.

11.13 Wrapping Up

Chapter 11 took the driver you built in Chapter 10 and placed it on a foundation of concurrency understanding. We did not change its behaviour; we built the understanding that lets you change its behaviour safely in the future.

We started with the premise that concurrency is the default environment every driver lives in, and that a driver which ignores that fact is a driver waiting to fail on a user's faster machine, busier workload, or larger core count. The four failure modes (data corruption, lost updates, torn values, inconsistent composite state) are not rare edge cases; they are the certainty when multiple streams touch shared state without coordination.

We then built the tools that prevent those failures. Atomics, for the small and simple cases. Mutexes, for everything else. We applied the tools to the driver: per-CPU counters replaced the mutex-protected byte counters, the single mutex covers the `cbuf` and its composite invariants, and `mtx_sleep` provides the blocking path that Chapter 10 used without explanation.

We learned to verify. INVARIANTS and WITNESS catch sleeping-with-mutex and lock-order violations at development time. KASSERT and `mtx_assert` document invariants and catch their violations. `dtrace` observes without disturbing. A multi-threaded and multi-process test suite exercises the driver at levels the base-system tools cannot reach.

We refactored for maintainability. The locking strategy is documented in `LOCKING.md`. The versioning is explicit. The README introduces the project to a new reader. The regression test runs every test we have built.

The driver is now the most robust character-device driver many beginners have ever written. It is also small, comprehensible, and source-referenced. Every locking decision has a first-principles justification. Every test has a specific purpose. Every piece of infrastructure supports the next chapter's work.

Three closing reminders before moving on.

The first is to *run the tests*. Concurrency bugs do not make themselves obvious. If you have not run `mp_stress` for an hour on a debug kernel, you have not confirmed that your driver is correct; you have only confirmed that your tests cannot find a bug in the time you gave them.

The second is to *keep `LOCKING.md` up to date*. Every future change that touches shared state updates the document. A driver whose concurrency story drifts away from its documentation is a driver that accumulates subtle bugs.

The third is to *trust the primitives*. The kernel's mutex, atomic, and sleep primitives are the result of decades of engineering and are extensively tested. The rules feel arbitrary at first; they are not. Learn the rules, apply them consistently, and the kernel does the hard work for you.

11.14 Chapter 11 Reference Material

The sections that follow are reference material for this chapter. They are not part of the main teaching sequence; they are here because you will return to them every time you need to recall a specific primitive, verify an invariant, or revisit a concept the chapter introduced briefly. Read them in order the first time, then come back to individual sections as needed.

The reference sections are named rather than lettered, to avoid confusion with the book's top-level appendices A through E. Each section stands on its own.

11.15 Reference: Building and Booting a Debug Kernel

Several labs in this chapter assume a kernel built with `INVARIANTS` and `WITNESS`. On a FreeBSD 14.3 release system, the stock `GENERIC` kernel does not include these options, because they carry a runtime cost unsuitable for production. This walkthrough shows how to build and install a debug kernel that does.

11.15.1 Step 1: Prepare a Custom Configuration

Make a copy of the generic configuration so you do not modify the original:

```
# cd /usr/src/sys/amd64/conf
# cp GENERIC MYFIRSTDEBUG
```

Edit `MYFIRSTDEBUG` and add the debug options if they are not already present:

ident	MYFIRSTDEBUG
options	INVARIANTS
options	INVARIANT_SUPPORT
options	WITNESS
options	WITNESS_SKIPSPIN
options	DDB

options	KDB
options	KDB_UNATTENDED

DDB enables the in-kernel debugger, which you can enter on a panic to inspect state. KDB is the kernel debugger framework. KDB_UNATTENDED causes the system to reboot after a panic rather than waiting for a human at the console, which is the right setting for a headless lab machine.

11.15.2 Step 2: Build the Kernel

```
# cd /usr/src
# make buildkernel KERNCONF=MYFIRSTDEBUG -j 4
```

The `-j 4` parallelizes across 4 cores. Adjust to your machine.

On a reasonably fast machine, the build takes 15 to 30 minutes. On a slow machine, it can take an hour or more. This is a one-time cost; incremental rebuilds are much faster.

11.15.3 Step 3: Install and Reboot

```
# make installkernel KERNCONF=MYFIRSTDEBUG
# shutdown -r now
```

The install places the new kernel in `/boot/kernel/` and the old one in `/boot/kernel.old/`. If the new kernel panics on boot, you can boot from `/boot/kernel.old` at the loader prompt.

11.15.4 Step 4: Confirm the Boot

After reboot, confirm the debug options are active:

```
$ sysctl kern.ident
kern.ident: MYFIRSTDEBUG
```

You are now running a debug kernel. Your driver's KASSERT calls will fire on failures. WITNESS warnings will appear in `dmesg`.

11.15.5 Step 5: Rebuild the Driver

Modules must be rebuilt against the running kernel's source:

```
$ cd your-driver-directory
$ make clean && make
```

Load as usual:

```
# kldload ./myfirst.ko
```

11.15.6 Reverting to the Production Kernel

If you want to go back to the non-debug kernel:

```
# shutdown -r now
```

At the loader prompt, type `unload` then `boot /boot/kernel.old/kernel` (or rename the directories as the FreeBSD handbook describes).

The debug kernel is slower than the production kernel. Do not run production workloads under it. Do run every driver test under it.

11.16 Reference: Further Reading on FreeBSD Concurrency

For readers who want to go deeper than this chapter covers, here are pointers to the canonical sources:

11.16.1 Manual Pages

- `mutex(9)`: the definitive reference for the mutex API.
- `locking(9)`: an overview of FreeBSD's locking primitives.
- `lock(9)`: the common lock-object infrastructure.
- `atomic(9)`: the atomic operation API.
- `counter(9)`: the per-CPU counter API.
- `msleep(9)` and `mtx_sleep(9)`: the sleep-with-interlock primitives.
- `condvar(9)`: condition variables (Chapter 12 material).
- `sx(9)`: sleepable shared/exclusive locks (Chapter 12 material).
- `rwlock(9)`: reader/writer locks (Chapter 12 material).
- `witness(4)`: the WITNESS lock-order checker.

11.16.2 Source Files

- `/usr/src/sys/kern/kern_mutex.c`: the mutex implementation.
- `/usr/src/sys/kern/subr_sleepqueue.c`: the sleep queue machinery.
- `/usr/src/sys/kern/subr_witness.c`: the WITNESS implementation.
- `/usr/src/sys/sys/mutex.h`: the mutex header with flag definitions.

- `/usr/src/sys/sys/_mutex.h`: the mutex structure.
- `/usr/src/sys/sys/lock.h`: common lock-object declarations.
- `/usr/src/sys/sys/atomic_common.h` and `/usr/src/sys/amd64/include/atomic.h`: atomic primitives.
- `/usr/src/sys/sys/counter.h`: the counter(9) API header.

11.16.3 External Material

The FreeBSD handbook covers kernel programming at an overview level. For more depth, *The Design and Implementation of the FreeBSD Operating System* (McKusick et al.) is the canonical textbook. For concurrency theory applicable to any OS, *The Art of Multiprocessor Programming* (Herlihy and Shavit) is excellent. Neither is required reading for our purposes, but both are useful references.

11.17 Reference: A Chapter 11 Self-Assessment

Before turning to Chapter 12, use this rubric to confirm you have internalised the Chapter 11 material. Every question should be answerable without re-reading the chapter. If any question is hard, return to the relevant section.

11.17.1 Conceptual Questions

1. **Name the four failure modes of unsynchronised shared state.** Data corruption, lost updates, torn values, inconsistent composite state.
2. **What is a race condition?** Overlapping access to shared state, at least one write, correctness that depends on the interleaving.
3. **Why does `mtx_sleep` take the mutex as an interlock argument, rather than the caller unlocking then sleeping?** Because unlocking then sleeping has a lost-wakeup window: a concurrent wakeup could fire after the unlock but before the sleep, be delivered to nobody, and leave the sleeper waiting forever.
4. **Why is it illegal to hold a non-sleepable mutex across a call to `uiomove`?** Because `uiomove` may sleep on a user-space page fault, and sleeping with a non-sleepable lock held is forbidden by the kernel (detected by `WITNESS` on debug kernels, undefined on production kernels).
5. **What is the difference between `MTX_DEF` and `MTX_SPIN`?** `MTX_DEF` sleeps when contended; `MTX_SPIN` busy-waits and disables interrupts. Use `MTX_DEF` for code in thread context; use `MTX_SPIN` only when sleeping is forbidden (for example, inside hardware interrupt handlers).
6. **When is an atomic operation sufficient to replace a mutex?** When the shared state is a single word-sized field and the operation does not compose with other fields or require blocking.
7. **What does `WITNESS` catch that single-threaded testing does not?** Lock-order reversals, sleeping with the wrong lock held, releasing an unowned lock, recursive acquisition of a non-recursive lock.
8. **Why is the per-CPU counter (`counter(9)`) faster than a mutex-protected counter?** Because per-CPU counters do not contend across CPUs. Each CPU updates its own cache line; summing happens only on read.

11.17.2 Code Reading Questions

Open your Chapter 11 driver source and answer:

1. Every call to `mtx_lock(&sc->mtx)` should be followed eventually by exactly one call to `mtx_unlock(&sc->mtx)` on every path. Verify this by inspection.
2. Every call to `mtx_sleep(&chan, &sc->mtx, ...)` should have a matching `wakeup(&chan)` somewhere in the driver. Find them.
3. The `mtx_assert(&sc->mtx, MA_OWNED)` calls are asserting that the mutex is held. Every caller of those helpers must hold the mutex. Verify by inspection.
4. The detach path sets `sc->is_attached = 0` before calling `wakeup(&sc->cb)`. Why is the order important?
5. The I/O handlers release `sc->mtx` before calling `uiomove`. Why?
6. The `selwakeup` call is placed outside the mutex. Why?

11.17.3 Hands-On Questions

1. Load the Chapter 11 driver on a WITNESS-enabled kernel and run the test kit. Are there any warnings in `dmesg`? If yes, investigate; if no, consider the driver verified.
2. Introduce a deliberate bug: comment out one `wakeup(&sc->cb)` call. Run the test kit. What happens?
3. Revert the bug. Introduce a different deliberate bug: comment out one `mtx_unlock` call and replace it with `mtx_unlock` after the `uiomove` call. Run on WITNESS. What does WITNESS say?
4. Run `mp_stress` for 60 seconds. Does the byte count converge?
5. Run `producer_consumer` 100 times in a loop. Every run should pass. Does it?

If all five hands-on questions pass, your Chapter 11 work is solid. You are ready for Chapter 12.

11.18 Reference: A Closer Look at Sleep Queues

This reference section explains, at a level appropriate for a beginner with curiosity, what happens inside the kernel when a thread calls `mtx_sleep` or `wakeup`. You do not need this material to write a correct driver; you have been doing that for several chapters. The section is here because, once you have seen the primitives work, you may want to know *how* they work. The knowledge pays off when you read other drivers and when you debug rare concurrency issues.

11.18.1 The Sleep Queue

A **sleep queue** is a kernel data structure that tracks threads waiting for a condition. Every active wait corresponds to exactly one sleep queue. When a thread calls `mtx_sleep(chan, mtx, pri, wmesg, timo)`, the kernel looks up (or creates) the sleep queue associated with `chan` (the wait channel) and adds the thread to it. When `wakeup(chan)` is called, the kernel looks up the queue and wakes every thread on it.

The data structure itself is defined in `/usr/src/sys/kern/subr_sleepqueue.c`. A hash table keyed on the channel address maps channels to queues. Each queue has a spin lock protecting its list of waiters. The spin lock is internal to the sleep-queue machinery; you never see it directly.

Why a hash table? Because a channel can be any pointer value, and the kernel may be serving thousands of simultaneous waits. A direct map (one queue per channel) would be too sparse; a single global list would contend too much. The hash table splits the load across many buckets, keeping lookups fast.

11.18.2 The Atomicity of `mtx_sleep`

Here is what `mtx_sleep(chan, mtx, pri, wmesg, timo)` actually does, at a high level:

1. Take the sleep-queue spin lock for `chan`.
2. Add the current thread to the sleep queue under that lock.
3. Release `mtx` (the outer mutex the caller is holding).
4. Call the scheduler, which picks another thread to run.
5. When awakened, the scheduler restores this thread. Control returns inside `mtx_sleep`.
6. Release the sleep-queue spin lock.
7. Reacquire `mtx`.
8. Return to the caller.

The key point is that steps 1 and 2 happen before step 3. Once the thread is on the sleep queue, any subsequent `wakeup(chan)` will remove it. If `wakeup(chan)` fires between step 3 and step 4, it removes the thread from the queue and marks it runnable; step 4 (calling the scheduler) will still happen but will find the thread immediately runnable, so the sleep is zero-length.

The window that would have been “released the mutex, not yet on the sleep queue” does not exist. The sleep queue entry happens *first*, while the mutex is still held; then the mutex is released *while the sleep-queue lock is held*. No wakeup can slip through.

This is the race that the atomic drop-and-sleep prevents, visible at the source level.

11.18.3 The wakeup Path

`wakeup(chan)` does the following:

1. Take the sleep-queue spin lock for `chan`.
2. Remove every thread on the queue.
3. Mark each removed thread as runnable.
4. Release the sleep-queue spin lock.
5. If any of the removed threads had higher priority than the current thread, call the scheduler to preempt.

The removed threads do not immediately run; they are merely runnable. The scheduler will pick them up in due course. If none has a higher priority than the caller, the caller continues to run; if one does, the caller yields.

`wakeup_one(chan)` is the same except that it removes only one thread (the highest-priority one, with FIFO among equal priorities). This is preferred when only one thread can usefully consume the signal. For a single-consumer buffer, `wakeup_one` avoids the thundering-herd problem; for a signal that can benefit multiple consumers, `wakeup` is appropriate.

For our driver, either would work. We use wakeup because both a reader and a writer may be waiting on the same channel and both need to be notified if the buffer state changes.

11.18.4 Priority Propagation Inside `mtx_sleep`

Section 5 introduced priority propagation as the kernel's defence against priority inversion. The mechanism lives in the mutex code itself: when `mtx_sleep` adds the calling thread to the sleep queue, it inspects the owner of the mutex (stored as a thread pointer inside the mutex word). If the waiting thread's priority is higher than the owner's, the owner's priority is boosted. The boost persists until the owner releases the mutex, at which point the owner's original priority is restored and the waiter acquires.

You can observe this effect live on a running system with `top -SH` during a contested workload: the priority of the thread holding a contested lock dances around as different waiters arrive and leave. It is one of the kernel services you benefit from silently; the boost is automatic and correct, and you do not have to opt in.

11.18.5 Adaptive Spinning Revisited

Section 5 introduced adaptive spinning as the optimization that lets `mtx_lock` avoid going to sleep when the holder is about to release. The sleep-queue machinery is where the decision is made. At its core, the slow path does roughly:

```
while (!try_acquire()) {
    if (holder is running on another CPU)
        spin briefly;
    else
        enqueue on the sleep queue and yield;
}
```

The “holder is running on another CPU” check reads the mutex word to get the owner pointer, then asks the scheduler whether that thread is currently on-CPU. If yes, the mutex is likely to be released within microseconds and a short spin beats the cost of a context switch. If no (the holder is itself asleep or has been descheduled), spinning is pointless and the waiter joins the sleep queue immediately. The threshold and exact spin count are kernel tunables; their precise values do not matter for the driver writer, as long as you trust that `mtx_lock` is fast in the common case and falls gracefully to sleep when it has to.

11.18.6 The Priority Argument

`mtx_sleep` takes a priority argument. In the Chapter 10 code, we passed `PCATCH` (allow signal interruption) with no explicit priority. The default priority, in FreeBSD, is whatever the calling thread is currently at; `PCATCH` alone does not change the priority.

Sometimes you want to sleep at a specific priority. The argument takes the form `PRIORITY | FLAGS`, where `PRIORITY` is a number (lower is better) and `FLAGS` include `PCATCH`. For our driver, the default priority is fine. Specialised drivers might pass `PUSER | PCATCH` to ensure the thread sleeps at a priority appropriate for user-visible work; real-time drivers might pass a lower (better) priority.

This is a detail you can safely ignore until a specific driver needs it. The default is the right answer most of the time.

11.18.7 Wait Message (wmesg) Conventions

The `wmesg` argument to `mtx_sleep` is a short string that appears in `ps`, `procstat`, and similar tools. Conventions:

- Five to seven characters. Longer strings are truncated.
- Lowercase.
- Indicate both the subsystem and the operation.
- `myfrd` = “myfirst read”. `myfwr` = “myfirst write”. `tunet` = “tun network”.

A good `wmesg` lets a developer scanning a `ps -AxH` output immediately know what a sleeping thread is waiting for. The few seconds it takes to invent a good name pay off the first time somebody needs to understand a hung system.

11.18.8 Sleep with Timeouts

The `timo` argument to `mtx_sleep` is the maximum wait time in ticks. Zero means “indefinite”; a positive value means “wake after this many ticks even if no one called wakeup”.

When the timeout fires, `mtx_sleep` returns `EWOLDBLOCK`. The caller is expected to check the return value and decide what to do.

Timeouts are how you implement bounded blocking. Chapter 12 covers `msleep_sbt(9)`, which takes an `sbintime_t` deadline instead of an integer tick count and is more convenient for sub-millisecond precision. For Chapter 11, the plain tick-based timeout is sufficient.

Example usage, for a driver that wants a 5-second maximum wait:

```
error = mtx_sleep(&sc->cb, &sc->mtx, PCATCH, "myfrd", hz * 5);
if (error == EWOLDBLOCK) {
    /* Timed out; return EAGAIN or retry. */
}
```

The `hz` constant is the kernel’s tick rate (typically 1000 on FreeBSD 14.3, configurable via `kern.hz`). Multiplying by 5 gives 5 seconds. This is a handy technique for a driver that wants progress guarantees.

11.19 Reference: Reading `kern_mutex.c`

The source for FreeBSD’s mutex implementation lives in `/usr/src/sys/kern/kern_mutex.c`. This reference section walks through the most interesting parts at a level appropriate for a driver writer who wants to know what happens when they call `mtx_lock`.

You do not need to understand every line. You do need to be able to open the file and recognize the shape of what is happening.

11.19.1 The mutex Structure

A mutex is a small structure. Its definition is in `/usr/src/sys/sys/_mutex.h`:

```
struct mtx {
    struct lock_object    lock_object;
    volatile uintptr_t    mtx_lock;
};
```

`lock_object` is the common header used by every lock class (mutexes, sx locks, rw locks, lockmgr locks, rmlocks). It holds the name, the lock flags, and WITNESS tracking state.

`mtx_lock` is the actual lock word. Its value encodes the state:

- 0: unlocked.
- A thread pointer: the ID of the thread holding the lock.
- Thread pointer with low bits set: lock is held with modifiers (`MTX_CONTESTED`, `MTX_RECURSED`, etc.).

Acquiring the mutex is, at its core, a compare-and-swap: change `mtx_lock` from 0 (unlocked) to `curthread` (the calling thread's pointer).

11.19.2 The Fast Path of `mtx_lock`

Open `kern_mutex.c` and find `__mtx_lock_flags`. You will see something like:

```
void
__mtx_lock_flags(volatile uintptr_t *c, int opts, const char *file, int line)
{
    struct mtx *m;
    uintptr_t tid, v;

    m = mtxlock2mtx(c);
    /* ... WITNESS setup ... */

    tid = (uintptr_t)curthread;
    v = MTX_UNOWNED;
    if (!_mtx_obtain_lock_fetch(m, &v, tid))
        _mtx_lock_sleep(m, v, opts, file, line);
    /* ... lock acquired; lock-profiling bookkeeping ... */
}
```

The critical operation is `_mtx_obtain_lock_fetch`, which is a compare-and-swap. It tries to change `mtx->mtx_lock` from `MTX_UNOWNED` (zero) to `tid` (the current thread's pointer). If it succeeds, we hold the mutex, and the function returns without further work. If it fails, we go to `_mtx_lock_sleep`.

The fast path is therefore: one compare-and-swap plus some WITNESS bookkeeping. On an uncontested mutex, `mtx_lock` costs essentially one atomic operation.

11.19.3 The Slow Path

`_mtx_lock_sleep` is the slow path. It handles the case where the mutex is already held by another thread. The function is a few hundred lines; the critical parts are:

```
for (;;) {
    /* Try to acquire. */
    if (_mtx_obtain_lock_fetch(m, &v, tid))
        break;

    /* Adaptive spin if holder is running. */
    if (owner_running(v)) {
        for (i = 0; i < spin_count; i++) {
            if (_mtx_obtain_lock_fetch(m, &v, tid))
                goto acquired;
        }
    }

    /* Go to sleep. */
    enqueue_on_sleep_queue(m);
    schedule_out();
    /* When we resume, loop back and try again. */
}
acquired:
    /* ... priority propagation bookkeeping ... */
```

The structure is: try the compare-and-swap, spin if the holder is running, otherwise sleep. Wake up and retry. Eventually acquire.

Several pieces of the real code complicate this skeleton: handling of the `MTX_CONTESTED` flag, priority propagation, lock profiling, witness tracking, and special cases for recursion. Don't worry about them on a first read; the structure above is the essence.

11.19.4 The Unlock Path

`__mtx_unlock_flags` is the release. Its fast path:

```
void
__mtx_unlock_flags(volatile uintptr_t *c, int opts, const char *file, int line)
{
    struct mtx *m;
    uintptr_t tid;

    m = mtxlock2mtx(c);
    tid = (uintptr_t)curthread;
```

```
if (!_mtx_release_lock(m, tid))
    _mtx_unlock_sleep(m, opts, file, line);
}
```

`_mtx_release_lock` tries to atomically change `mtx->mtx_lock` from `tid` (current thread) to `MTX_UNOWNED`. If it succeeds, the mutex is released cleanly. If it fails, there were complications (the `MTX_CONTESTED` flag was set because waiters had arrived), and `_mtx_unlock_sleep` handles them.

The common case: one atomic store. The uncommon case: dequeue waiters and wake one.

11.19.5 The Takeaway

Reading `kern_mutex.c` once, even briefly, gives you a useful intuition. The mutex is not magic. It is a compare-and-swap with a fallback to a sleep queue. Everything else is bookkeeping and optimization.

The primitives you use (`mtx_lock`, `mtx_unlock`) are thin wrappers around these internals. Understanding them means you can reason about their cost, their guarantees, and their limits. That is the payoff.

11.20 Reference: A Mini-Glossary of Concurrency Vocabulary

Terminology matters in concurrency, more than in most subjects, because people who disagree about terminology often end up disagreeing about whether their code is correct. A short glossary of the terms used in this chapter:

Atomic operation: a memory operation that is indivisible from the perspective of other CPUs. It completes as a single transaction; no other CPU can observe a partial result.

Barrier / fence: a point in the code past which the compiler and CPU are not allowed to move specific kinds of memory accesses. Barriers come in acquire, release, and full forms.

Critical section: a contiguous region of code that accesses shared state and must execute without interference from concurrent threads.

Deadlock: a situation in which two or more threads are each waiting for a resource held by another, with none able to make progress.

Invariant: a property of a data structure that is true outside of critical sections. Code inside a critical section may temporarily violate the invariant; code outside must not.

Livelock: a situation in which threads are making progress in the sense of executing instructions, but the useful state of the system does not advance. Typically caused by spinning retries that all fail.

Lock-free: describing an algorithm or data structure that avoids mutual-exclusion primitives, typically relying on atomics. Lock-free algorithms guarantee that at least one thread always makes progress; they do not necessarily prevent starvation of specific threads.

Memory ordering: the rules governing when a write on one CPU becomes visible to reads on other CPUs. Different architectures have different models (strongly ordered, weakly ordered, release-acquire).

Mutual exclusion (mutex): a synchronization primitive ensuring that only one thread at a time can execute a protected region.

Priority inversion: a situation in which a low-priority thread holds a resource that a high-priority thread needs, and a medium-priority thread preempts the low-priority holder, effectively blocking the high-priority waiter.

Priority propagation (inheritance): the kernel's mechanism for solving priority inversion by temporarily raising a low-priority holder's priority to match its highest-priority waiter.

Race condition: a situation in which the correctness of a program depends on the exact interleaving of concurrent accesses to shared state, and at least one access is a write.

Sleep mutex (MTX_DEF): a mutex that puts contending threads to sleep. Legal to use in contexts where sleeping is allowed.

Spin mutex (MTX_SPIN): a mutex that causes contending threads to spin in a busy loop. Required in contexts where sleeping is forbidden, such as hardware interrupt handlers.

Starvation: a situation in which a thread is runnable but never actually runs, because other threads are always scheduled ahead of it. Priority inversion is a specific cause of starvation.

Wait-free: a stricter property than lock-free. Every thread is guaranteed to make progress within a bounded number of steps, regardless of what other threads do.

WITNESS: FreeBSD's lock-order and locking-discipline checker. A kernel option that tracks lock acquisitions and warns on violations.

11.21 Reference: Concurrency Patterns in Real FreeBSD Drivers

Before Chapter 12 introduces additional synchronization primitives, it helps to survey how concurrency is handled in real drivers in `/usr/src/sys/dev/`. The patterns below are the ones you will see most often. Each shows that the tools you have learned (a mutex, a sleep channel, selinfo) are the same tools that ship with FreeBSD.

11.21.1 Pattern: Per-Driver Softc Mutex

Nearly every character-device driver in the tree uses a single mutex per softc. Examples:

- `/usr/src/sys/dev/evdev/evdev_private.h` declares `ec_buffer_mtx` for the per-client state.
- `/usr/src/sys/dev/random/randomdev.c` uses `sysctl_lock` for its private configuration.
- `/usr/src/sys/dev/null/null.c` does not use a mutex, because its `cbuf` is effectively stateless; this is the exception that confirms the rule.

The per-softc mutex pattern is not novel, not sophisticated, and not optional. It is the backbone of driver concurrency in FreeBSD. Our `myfirst` driver follows it because every small driver follows it.

11.21.2 Pattern: Sleep Channel on a softc Field

The convention for the `chan` argument to `mtx_sleep` is to use the address of a softc field related to the wait condition. Examples:

- `evdev_read` in `/usr/src/sys/dev/evdev/cdev.c` sleeps on the per-client structure (via `mtx_sleep(client, ...)`) and is woken when new events arrive.
- Our `myfirst_read` sleeps on `&sc->cb` and wakes correspondingly.

The address can be anything; what matters is that sleepers and wakers use the same pointer. Using a struct field's address has the benefit that the channel is visible in source and self-documenting: a reader of the code knows what the thread is waiting for.

11.21.3 Pattern: selinfo for poll() Support

Every driver that supports `poll(2)` or `select(2)` has at least one struct `selinfo`, sometimes two (for read and write readiness). The same two calls appear:

- `selrecord(td, &si)` inside the `d_poll` handler.
- `selwakeup(&si)` when the readiness state changes.

This is exactly what `myfirst` does. The `evdev` driver does it. The `tty` drivers do it. It is the standard form.

11.21.4 Pattern: Bounce Buffer with uiomove

Many drivers that have wrap-around buffers use the bounce-buffer pattern we discussed in Chapter 10: copy bytes from the ring into a stack-local scratch, drop the lock, `uiomove` the scratch to user space, reacquire the lock. Examples:

- `evdev_read` copies a single event structure into a scratch variable before calling `uiomove`.
- `ucom_put_data` in `/usr/src/sys/dev/usb/serial/usb_serial.c` has a similar shape for its circular buffer.
- Our `myfirst_read` has the same shape for the `cbuf`.

This is the idiomatic answer to “how do I do `uiomove` from a wrap-around buffer while holding a mutex?”. The bounce is small (often on the stack, sometimes a dedicated small buffer), and the lock is dropped around the `uiomove`.

11.21.5 Pattern: Counter(9) for Statistics

Modern drivers use `counter(9)` for statistics counters. Examples:

- `/usr/src/sys/net/if.c` uses `if_ierrors`, `if_oerrors`, and many other counters as `counter_u64_t`.
- Many of the network drivers (`ixgbe`, `mlx5`, etc.) use `counter(9)` for their per-queue statistics.

Our `myfirst` migration to `counter(9)` puts us in line with this modern pattern. Older drivers that still use plain atomic counters typically predate the `counter(9)` API; newer code uses `counter(9)`.

11.21.6 Pattern: Detach with Outstanding Users

A recurring challenge is detach when user processes still have the device open. Two common approaches:

Refuse detach while in use. Our driver does this: detach returns EBUSY if `active_fhs > 0`. Simple, safe, but means the module cannot be unloaded while anyone is using it.

Forced detach with `destroy_dev_drain`. The kernel provides `destroy_dev_drain(9)`, which waits for all handlers to return and prevents new ones from starting. Combined with a “this device is going away” flag in the softc, this allows forced detach with graceful completion of in-flight operations. Chapter 12 and beyond cover this in more depth.

For my first, the simpler “refuse detach” approach is correct. A production driver that must support hot-unplug (USB, for example) uses the more complex approach.

11.21.7 Pattern: WITNESS-Documented Lock Order

Drivers with more than one lock document the order. Example:

```
/*
 * Locks in this driver, in acquisition order:
 *   sc->sc_mtx
 *   sc->sc_listmtx
 *
 * A thread holding sc->sc_mtx may acquire sc->sc_listmtx, but
 * not the reverse. See sc_add_entry() for the canonical example.
 */
```

WITNESS validates this ordering at runtime; the comment makes it readable for humans. Our driver has one lock and no ordering rules, but as soon as we add a second (Chapter 12 may introduce one), we will add this kind of documentation.

11.21.8 Pattern: Assertions Everywhere

Real drivers are littered with `KASSERT` and `mtx_assert`. Examples:

- `if_alloc` in `/usr/src/sys/net/if.c` asserts invariants on input structures.
- GEOM code (in `/usr/src/sys/geom/`) has `g_topology_assert` and similar in most functions.
- Network drivers often have `NET_EPOCH_ASSERT()` to confirm the caller is in the right context.

Assertions are cheap in production and life-saving in development. Our Chapter 11 work added several; future chapters will add more.

11.21.9 What Real Drivers Do That Ours Does Not

Three patterns appear in real drivers that our driver does not yet use:

Finer-grained locking. A network driver might have a lock per queue; a storage driver might have a lock per request. Our driver has one lock for everything. For small drivers this is correct; for drivers with hot paths on many cores, it is not enough. Chapter 12’s `sx` lock and Chapter 15’s discussion of RCU-style patterns cover the finer-grained cases.

Lock-free hot paths. `buf_ring(9)` gives multi-producer queues with either single-consumer or multi-consumer dequeue, with no locks on the hot path. `epoch(9)` lets readers proceed lock-free. These are optimizations for specific workloads. A beginner driver should not use them first; the justification should be a measured bottleneck, not a speculative one.

Hierarchical locking with multiple classes. A real driver might use a spin mutex for interrupt context, a sleep mutex for normal work, and an sx lock for configuration. Each has its place; the interactions are governed by strict ordering rules. We will introduce these in Chapter 12 and revisit them in later chapters when specific drivers need them.

11.21.10 The Takeaway

The patterns in `/usr/src/sys/dev/` are not exotic. They are the ones we have been building. An `evdev_client` with a mutex, a sleep channel, and a `selinfo` is structurally the same as a `myfirst_softc` with the same three pieces. Reading a few of these drivers with the tools Chapter 11 has built should feel almost familiar. When something looks strange, it is usually a specialisation (for example, `kqueue` filter registration) that future chapters cover.

This is the compounding payoff of the book: each chapter's patterns make a larger fraction of the real source readable.

11.22 Reference: Common Mutex Idioms

This is a quick-lookup collection of the idioms you will use most often in FreeBSD driver code.

11.22.1 Basic Critical Section

```
mtx_lock(&sc->mtx);  
/* access shared state */  
mtx_unlock(&sc->mtx);
```

11.22.2 Assert Held

```
mtx_assert(&sc->mtx, MA_OWNED);
```

Put at the top of every helper that assumes the mutex is held. Compiles to nothing on non-INVARIANTS kernels; panics on INVARIANTS kernels if the mutex is not held.

11.22.3 Try-Lock

```
if (mtx_trylock(&sc->mtx)) {
    /* got it */
    mtx_unlock(&sc->mtx);
} else {
    /* someone else has it; back off */
}
```

Use when you want to avoid blocking. Useful in timer callbacks that should not contend with I/O handlers; if the lock is taken, skip this tick and retry next time.

11.22.4 Sleep Waiting for a Condition

```
mtx_lock(&sc->mtx);
while (!condition)
    mtx_sleep(&chan, &sc->mtx, PCATCH, "wmesg", 0);
/* condition is true now; do work */
mtx_unlock(&sc->mtx);
```

The while loop is essential. Spurious wake-ups are permitted and common; the thread must re-check the condition after every return from `mtx_sleep`.

11.22.5 Wake Sleepers

```
wakeup(&chan);           /* wake all */
wakeup_one(&chan);       /* wake highest-priority one */
```

Call without holding the mutex. The kernel handles ordering.

11.22.6 Sleep with Timeout

```
error = mtx_sleep(&chan, &sc->mtx, PCATCH, "wmesg", hz * 5);
if (error == EWOULDBLOCK) {
    /* timed out */
}
```

Bound the wait to 5 seconds. Convenient for heartbeats and deadline-driven operations.

11.22.7 Drop and Reacquire Around a Potentially-Sleeping Call

```
mtx_lock(&sc->mtx);
/* snapshot what we need */
snap = sc->field;
mtx_unlock(&sc->mtx);

/* do the sleeping call */
error = uiomove(buf, snap, uio);

mtx_lock(&sc->mtx);
/* update shared state */
sc->other_field = /* ... */;
mtx_unlock(&sc->mtx);
```

This is the pattern our I/O handlers use. The mutex is never held across a sleeping call.

11.22.8 Atomic-Instead-of-Lock for Simple Counters

```
counter_u64_add(sc->bytes_read, got);
```

Faster and more scalable than `mtx_lock`; `sc->bytes_read += got`; `mtx_unlock`.

11.22.9 Check and Set Atomically

```
if (atomic_cmpset_int(&sc->flag, 0, 1)) {
    /* we set it from 0 to 1; we are the "first" */
}
```

No mutex needed. The compare-and-swap itself is atomic.

11.23 Looking Ahead: Bridge to Chapter 12

Chapter 12 is titled *Synchronization Mechanisms*. Its scope is the rest of the FreeBSD synchronization toolkit: condition variables, sleepable shared-exclusive locks, read/write locks, timed waits, and the deeper techniques for debugging deadlocks. Everything in Chapter 12 builds on the mutex foundation Chapter 11 laid down.

The bridge is three specific observations from Chapter 11's work.

First, you already use a “channel” (`&sc->cb`) with `mtx_sleep` and `wakeup`. Chapter 12 introduces **condition variables** (`cv(9)`), which are a named, structured alternative to the anonymous-channel pattern. You will

see that `cv_wait_sig(&cb_has_data, &sc->mtx)` is often clearer than `mtx_sleep(&sc->cb, &sc->mtx, PCATCH, "myfrd", 0)`, especially when there are multiple conditions the thread might wait on.

Second, your driver's one mutex serializes everything, including reads that could theoretically proceed in parallel. Chapter 12 introduces `sx(9)` sleepable shared-exclusive locks, which allow multiple readers to hold the lock simultaneously while serializing writers. You will reason about when that design is appropriate (read-heavy workloads, configuration state) and when it is overkill (small critical sections where the mutex is already fine).

Third, your blocking path blocks indefinitely. Chapter 12 introduces **timed waits**: `mtx_sleep` with a nonzero `timo`, `cv_timedwait`, and `sx_sleep` with timeouts. You will learn how to bound the waiting time so a user can press Ctrl-C and have the driver respond within a sensible interval.

Specific topics Chapter 12 will cover include:

- The `cv_wait`, `cv_signal`, `cv_broadcast`, and `cv_timedwait` primitives.
- The `sx(9)` API: `sx_init`, `sx_slock`, `sx_xlock`, `sx_sunlock`, `sx_xunlock`, `sx_downgrade`, `sx_try_upgrade`.
- Reader/writer locks (`rw(9)`) as a spin-mutex sibling of `sx`.
- Timeout handling and the `sbintime_t` family of time primitives.
- Deadlock debugging in depth, including the WITNESS lock-order list.
- Lock-order design patterns and the `LORD_*` conventions.
- A concrete application: upgrading parts of the driver to use `sx` and `cv` where appropriate.

You do not need to read ahead. Chapter 11's material is sufficient preparation. Bring your Stage 3 or Stage 5 driver, your test kit, and your WITNESS-enabled kernel. Chapter 12 starts where Chapter 11 ended.

A small closing reflection. You have just done something unusual. Most driver tutorials teach mutex usage as a formula: put a `mtx_lock` here and a `mtx_unlock` there, and the driver will work. You have done the harder thing: you have understood what the formula means, when it is necessary, why it is sufficient, and how to verify it. That understanding is what separates a driver writer from a driver author. Every chapter from here forward builds on it.

Take a moment. Appreciate how far you have come from the opening "Hello, kernel!" of Chapter 7. Then turn the page.

12 Synchronization Mechanisms

12.1 Reader Guidance & Outcomes

Chapter 11 ended with a driver that was, for the first time in this book, *verifiably* concurrent. You had a single mutex protecting the circular buffer, atomic counters that scaled to many cores, `WITNESS` and `INVARIANTS` watching every lock acquisition, a stress-test kit that ran on a debug kernel, and a `LOCKING.md` document that any future maintainer (including future you) could read to understand the concurrency story. That was real progress. It was not, however, the end of the story.

The synchronization toolkit FreeBSD gives you is much larger than the single primitive Chapter 11 introduced. The mutex you used is the right answer for many situations. It is also the wrong answer for several others. A read-mostly configuration table that twenty threads scan ten thousand times per second wants something other than a mutex serializing the reads. A blocking read that should respond to Ctrl-C within milliseconds wants something other than `mtx_sleep(9)` with no timeout. A coordinated wake-up across a dozen waiters, one per condition, wants something more expressive than an anonymous channel pointer like `&sc->cb`. The kernel has primitives for each of these, and Chapter 12 is where we meet them.

This chapter does for the rest of the synchronization toolkit what Chapter 11 did for the mutex. We introduce each primitive with motivation first, build the mental model, ground it in real FreeBSD source, apply it to the running `myfirst` driver, and verify the result against a `WITNESS` kernel. By the end you will be able to read a real driver and recognise, by the choice of primitive alone, what its author was trying to express. You will also be able to make those choices yourself, in your own driver, without reaching for the wrong tool out of habit.

12.1.1 Why This Chapter Earns Its Own Place

It would be possible, in principle, to stop with Chapter 11. A mutex, an atomic counter, and `mtx_sleep` cover the simple cases. Many small drivers in the FreeBSD tree do not use anything else.

The trouble is that “many small drivers” is not where most of the bugs live. The drivers people maintain longest are the ones that grew. A USB device driver started small, then acquired a control channel, then a configuration table that user space could change at runtime, then a separate event queue with its own waiters. Every one of those additions exposed the limits of “one mutex protects everything”. A driver writer who only knows the mutex ends up either misusing it (a held-too-long mutex blocks the whole subsystem) or working around it (a tangle of busy-wait loops, racy retries, and global flags that “should not happen but sometimes do”). The synchronization primitives this chapter teaches exist precisely to keep those workarounds out of the code.

Each primitive is a different shape of agreement between threads. A mutex says *only one of us at a time*. A condition variable says *I will wait for a specific change you will tell me about*. A shared/exclusive lock says *many of us may read; only one of us may write*. A timed sleep says *and please give up if it is taking too long*. A driver that uses each tool for what it is good at reads cleanly, behaves predictably, and remains comprehensible

long after the original author has stopped looking at it. A driver that uses one tool for everything either suffers in performance or hides bugs in places no one is looking.

This chapter is therefore a vocabulary chapter as much as it is a mechanics chapter. We do introduce the APIs and we do walk through code. The deeper goal is to give you the words for what you are trying to say.

12.1.2 Where Chapter 11 Left the Driver

A quick checkpoint, because Chapter 12 builds directly on top of the Chapter 11 deliverables. If any of the following is missing or feels uncertain, return to Chapter 11 before starting this chapter.

- Your `myfirst` driver compiles cleanly with `WARNS=6`.
- It uses `MYFIRST_LOCK(sc)`, `MYFIRST_UNLOCK(sc)`, and `MYFIRST_ASSERT(sc)` macros that expand to `mtx_lock`, `mtx_unlock`, and `mtx_assert(MA_OWNED)` on the device-wide `sc->mtx` (an `MTX_DEF` sleep mutex).
- The `cbuf`, the per-descriptor counters, the open count, and the active-descriptor count are all protected by `sc->mtx`.
- The byte counters `sc->bytes_read` and `sc->bytes_written` are `counter_u64_t` per-CPU counters; they do not need the mutex.
- The blocking read and write paths use `mtx_sleep(&sc->cb, &sc->mtx, PCATCH, "myfrd"|"myfwr", 0)` as their wait primitive and `wakeup(&sc->cb)` as the matching wake.
- `INVARIANTS` and `WITNESS` are enabled in your test kernel; you have built and booted it.
- A `LOCKING.md` document accompanies the driver and lists every shared field, every lock, every wait channel, and every “deliberately not locked” decision with a justification.
- The Chapter 11 stress kit (`producer_consumer`, `mp_stress`, `mt_reader`, `lat_tester`) builds and runs cleanly.

That driver is the substrate this chapter starts from. We will not throw it away. We will replace some primitives with better-fitting ones, add a small new subsystem to give the new primitives something to protect, and finish with a driver whose synchronization design is both more capable and easier to read.

12.1.3 What You Will Learn

When you close this chapter you should be able to:

- Explain what *synchronization* means in a kernel-level sense, and distinguish it from the narrower concept of *mutual exclusion*.
- Map the FreeBSD synchronization toolkit onto a small decision tree: mutex, condition variable, shared/exclusive lock, reader/writer lock, atomic, sleep with timeout.
- Replace anonymous wait channels with named condition variables (`cv(9)`), and explain why that change improves both correctness and readability.
- Use `cv_wait`, `cv_wait_sig`, `cv_wait_unlock`, `cv_signal`, `cv_broadcast`, and `cv_broadcastpri` correctly with respect to their interlock mutex.
- Bound a blocking operation by a timeout using `cv_timedwait_sig` (or `mtx_sleep` with a non-zero `timo` argument), and design the caller’s response when the timeout fires.
- Distinguish `EINTR`, `ERESTART`, and `EWOULDBLOCK`, and decide which the driver should return when a wait fails.

- Choose between `sx(9)` (sleepable) and `rw(9)` (spin-based) reader/writer locks, understand the rules each places on the calling context, and apply `sx_init`, `sx_xlock`, `sx_slock`, `sx_xunlock`, `sx_sunlock`, `sx_try_upgrade`, and `sx_downgrade`.
- Design a multi-reader, single-writer arrangement that uses one lock for the data path and another for the configuration path, with a documented lock order and a WITNESS-verified absence of inversions.
- Read WITNESS warnings precisely enough to identify the exact lock pair and the source line of the violation.
- Use the in-kernel debugger commands `show locks`, `show all locks`, `show witness`, and `show lockchain` to inspect a system that has hung on a synchronization bug.
- Build a stress workload that exercises the driver across descriptors, `sysctls`, and timed waits at the same time, and read the output of `lockstat(1)` to find the contended primitives.
- Refactor the driver into a shape where the synchronization story is documented, the lock order is explicit, and the version string reflects the new architecture.

That is a substantial list. None of it is optional for a driver that aspires to outlast its author. All of it builds on what Chapter 11 left in your hands.

12.1.4 What This Chapter Does Not Cover

Several adjacent topics are deliberately deferred:

- **Callouts (`callout(9)`).** Chapter 13 introduces timed work that fires from the kernel's clock infrastructure. We touch the topic here only as a sleep-with-timeout primitive seen from the driver's blocking call; the full callout API and its rules belong in Chapter 13.
- **Taskqueues (`taskqueue(9)`).** Chapter 16 introduces the kernel's deferred-work framework. Several drivers use a taskqueue to decouple the blocking thread from the wake-up signal, but doing so well requires its own chapter.
- **epoch(9) and read-mostly lock-free patterns.** Network drivers in particular use `epoch(9)` to let readers proceed without acquiring a lock at all. The mechanism is subtle and is best taught alongside the network driver subsystem in Part 6.
- **Interrupt-context synchronization.** Real hardware interrupt handlers add another layer of constraint on what locks you can hold and which sleep primitives are legal. Chapter 14 introduces interrupt handlers and revisits the synchronization rules from inside that context. For Chapter 12 we stay entirely in process and kernel-thread context.
- **Lockless data structures.** `buf_ring(9)` and friends are effective tools for hot paths, but they reward careful study and require a specific workload to pay back their complexity. Part 6 (Chapter 28) introduces them when a driver in the book actually needs one.
- **Distributed and cross-machine synchronization.** Out of scope. We are a single-host operating system in this book.

Staying inside those lines keeps the chapter focused on what it can teach well. The reader of Chapter 12 should finish with confident control of `cv(9)`, `sx(9)`, and timed waits, and a working sense of where `rw(9)` and `epoch(9)` fit; that confidence is what makes the later chapters readable when they show up.

12.1.5 Estimated Time Investment

- **Reading only:** about three hours. The new vocabulary (condition variables, shared/exclusive locks, sleep-ability rules) takes time to absorb even though the API surface is small.
- **Reading plus typing the worked examples:** six to eight hours over two sessions. The driver evolves in four small stages; each stage adds one primitive.
- **Reading plus all labs and challenges:** ten to fourteen hours over three or four sessions, including time for stress runs and `lockstat(1)` analysis.

If you find yourself confused midway through Section 4, that is normal. The shared/exclusive distinction is genuinely new even for readers comfortable with mutexes, and the temptation to use `sx` for the data path is exactly the temptation Section 5 exists to resolve. Stop, re-read Section 4's example, and continue when the model has settled.

12.1.6 Prerequisites

Before starting this chapter, confirm:

- Your driver source matches the Chapter 11 Stage 3 (`counter9`) or Stage 5 (`KASSERTs`) tree. Stage 5 is preferred because the assertions catch new bugs faster.
- Your lab machine is running FreeBSD 14.3 with `/usr/src` on disk and matching the running kernel.
- A debug kernel with `INVARIANTS`, `WITNESS`, `WITNESS_SKIPSPIN`, `DDB`, `KDB`, and `KDB_UNATTENDED` is built, installed, and booting cleanly. The Chapter 11 reference section “Building and Booting a Debug Kernel” has the recipe.
- You read Chapter 11 carefully. The mutex rules, the sleep-with-mutex rule, the lock-order discipline, and the `WITNESS` workflow are all assumed knowledge here.
- You have run the Chapter 11 stress kit at least once and seen it pass.

If any of the above is shaky, fixing it now is a much better investment than pushing through Chapter 12 and trying to debug from a moving foundation.

12.1.7 How to Get the Most Out of This Chapter

Three habits will pay off quickly.

First, keep `/usr/src/sys/kern/kern_condvar.c`, `/usr/src/sys/kern/kern_sx.c`, and `/usr/src/sys/kern/kern_rwlock.c` bookmarked. Each one is short, well-commented, and the source of authority for what the primitive actually does. Several times in the chapter we will ask you to glance at a specific function. A minute spent there will make the surrounding paragraphs easier to absorb.

A note on line numbers. When we point you at a specific function later in the chapter, open the file and search for the symbol rather than jumping to a numeric line. `_cv_wait_sig` lives in `/usr/src/sys/kern/kern_condvar.c` and `sleepq_signal` in `/usr/src/sys/kern/subr_sleepqueue.c` in the 14.3 tree at time of writing; the names will carry across future point releases, but the line each one sits on will not. The durable reference is always the symbol.

Second, run every code change you make under `WITNESS`. The synchronization primitives we introduce in this chapter have stricter rules than the mutex did. `WITNESS` is the cheapest way to discover that you have broken

one. The Chapter 11 reference section “Building and Booting a Debug Kernel” walks through the kernel build if you need it; do not skip it now.

Third, type the driver changes by hand when you can. The companion source under `examples/part-03/ch12-synchronization-mechanisms/` is the canonical version, but the muscle memory of typing `cv_wait_sig(&sc->data_cv, &sc->mtx)` once is worth more than reading it ten times. The chapter shows the changes incrementally; mirror that incremental pace in your own copy of the driver.

12.1.8 Roadmap Through the Chapter

The sections in order are:

1. What synchronization is in the kernel, and where the various primitives sit in a small decision tree.
2. Condition variables, the cleaner alternative to anonymous wakeup channels, and the first refactor of the `myfirst` blocking paths.
3. Timeouts and interruptible sleep, including signal handling and the choice between `EINTR`, `ERESTART`, and `EWOULDBLOCK`.
4. The `sx(9)` lock, which finally lets us express “many readers, occasional writer” without serializing every reader.
5. A multi-reader, single-writer scenario in the driver: a small configuration subsystem, the lock order between the data path and the configuration path, and the `WITNESS` discipline that keeps it correct.
6. Debugging synchronization issues, including a careful tour of `WITNESS`, the in-kernel debugger commands for inspecting locks, and the most common deadlock patterns.
7. Stress testing under realistic I/O patterns, with `lockstat(1)`, `dtrace(1)`, and the existing Chapter 11 testers extended for the new primitives.
8. Refactoring and versioning the driver: a clean `LOCKING.md`, a bumped version string, an updated changelog, and a regression run that validates the whole thing.

Hands-on labs and challenge exercises follow, then a troubleshooting reference, a wrapping-up section, and a bridge to Chapter 13.

If this is your first pass, read linearly and do the labs in order. If you are revisiting, the debugging section and the refactoring section stand alone and make good single-sitting reads.

12.2 Section 1: What Is Synchronization in the Kernel?

Chapter 11 used the words *synchronization*, *locking*, *mutual exclusion*, and *coordination* somewhat interchangeably. That was acceptable when the only primitive on the table was the mutex, because the mutex collapses all those ideas into one mechanism. With the broader toolkit Chapter 12 introduces, the words start to mean different things, and getting them straight early prevents a lot of confusion later.

This section establishes the vocabulary. It also draws the small decision tree we will refer back to throughout the rest of the chapter when we ask “which primitive should I reach for here?”.

12.2.1 What Synchronization Means

Synchronization is the broader idea: any mechanism by which two or more concurrent threads of execution coordinate their access to shared state, their progress through a shared procedure, or their timing relative to each other.

Three flavours of coordination cover almost everything a driver needs:

Mutual exclusion: at most one thread at a time inside a critical region. Mutexes and exclusive locks deliver this. The guarantee is structural: while you are inside, nobody else is.

Shared access with restricted writes: many threads may inspect a value at the same time, but a thread that wants to change it must wait until everyone is out and nobody else is in. Shared/exclusive locks deliver this. The guarantee is asymmetric: readers tolerate each other; a writer tolerates nobody.

Coordinated waiting: a thread suspends until some condition becomes true, and another thread that knows the condition has become true wakes the waiter. Condition variables and the older `mtx_sleep / wakeup` channel mechanism deliver this. The guarantee is temporal: the waiter does not consume CPU while it is waiting; the waker does not have to know who is waiting; the kernel handles the rendezvous.

A driver typically uses all three. The `cbuf` in `myfirst` already uses two: mutual exclusion to protect the `cbuf` state, and coordinated waiting to suspend a reader when the buffer is empty. Chapter 12 adds the third (shared access) and refines the second (named condition variables instead of anonymous channels).

12.2.2 Synchronization Versus Locking

It is tempting to think that *synchronization* and *locking* are the same word. They are not.

Locking is one technique for synchronization. It is the family of mechanisms that operate on a shared object and grant or refuse access to it. Mutexes, `sx` locks, `rw` locks, and `lockmgr` locks are all locks.

Synchronization includes locking but also includes coordinated waiting (condition variables, sleep channels), event signalling (semaphores), and timed coordination (callouts, timed sleeps). A waiting thread may not be holding any lock at the moment it is suspended (in fact, with `mtx_sleep` and `cv_wait`, the lock is dropped during the wait), and yet it is participating in synchronization with the threads that will eventually wake it.

The mental model that follows from this distinction is useful: locking is *about access*; coordinated waiting is *about progress*. Most non-trivial driver code mixes both. The mutex around the `cbuf` is locking. The sleep on `&sc->cb` while the buffer is empty is coordinated waiting. Both are synchronization. Either one alone would not be enough.

12.2.3 Blocking Versus Spinning

Two basic shapes recur across the FreeBSD primitives. Knowing which shape a primitive uses is half the battle of choosing among them.

Blocking primitives put a contending thread to sleep on the kernel's sleep queue. The sleeping thread consumes no CPU; it will be made runnable again when the holding thread releases or the awaited condition is signalled. Blocking primitives are appropriate when the wait could be long, when the thread is in a context where sleeping

is legal, and when keeping the CPU busy in a tight retry loop would harm overall throughput. `MTX_DEF` mutexes, `cv_wait`, `sx_xlock`, `sx_slock`, and `mtx_sleep` are all blocking.

Spinning primitives keep the thread on-CPU and busy-wait on the lock state, retrying atomically until the holder releases. They are appropriate only when the critical section is very short, when the thread cannot legally sleep (for example, inside a hardware interrupt filter), or when the cost of a context switch would dwarf the wait. `MTX_SPIN` mutexes and `rw(9)` locks are spinning. The kernel itself uses spin locks for the very lowest layers of the scheduler and the interrupt machinery.

Chapter 12 stays mostly in the blocking world. Our driver runs in process context; it is permitted to sleep; the gain from spinning would be marginal. The one exception is when we mention `rw(9)` as a sibling of `sx(9)` for completeness; the deeper treatment of `rw(9)` belongs to chapters where a driver uses it for a real reason.

12.2.4 A Small Map of FreeBSD Primitives

The FreeBSD synchronization toolkit is larger than people expect. For driver work, eight primitives carry essentially all of the load:

Primitive	Header	Behaviour	Best for
<code>mtx(9)</code> (<code>MTX_DEF</code>)	<code>sys/mutex.h</code>	Sleep mutex; one owner at a time	Default lock for most softc state
<code>mtx(9)</code> (<code>MTX_SPIN</code>)	<code>sys/mutex.h</code>	Spin mutex; disables interrupts	Short critical sections in interrupt context
<code>cv(9)</code>	<code>sys/condvar.h</code>	Named wait channel; pairs with a mutex	Coordinated waits with multiple distinct conditions
<code>sx(9)</code>	<code>sys/sx.h</code>	Sleep-mode shared/exclusive lock	Read-mostly state in process context
<code>rw(9)</code>	<code>sys/rwlock.h</code>	Spin-mode reader/writer lock	Read-mostly state in interrupt or short critical sections
<code>rmlock(9)</code>	<code>sys/rmlock.h</code>	Read-mostly lock; cheap reads, expensive writes	Hot read paths with rare configuration changes
<code>sema(9)</code>	<code>sys/sema.h</code>	Counting semaphore	Resource accounting; rarely needed in drivers
<code>epoch(9)</code>	<code>sys/epoch.h</code>	Read-mostly synchronization with deferred reclamation	Hot read paths in network/storage drivers

The ones we use in this chapter, in addition to the mutex Chapter 11 introduced, are `cv(9)` and `sx(9)`. `rw(9)` is mentioned for context. `rmlock(9)`, `sema(9)`, and `epoch(9)` are deferred to later chapters where the driver in question actually justifies them.

12.2.5 Atomics in the Same Map

Strictly speaking, the `atomic(9)` primitives Chapter 11 covered are not part of the synchronization toolkit at all. They are *concurrent operations*: indivisible memory accesses that compose with the locks but do not themselves provide blocking, waiting, or signalling. They sit alongside the locks in the way a power tool sits alongside a hand tool: useful for a specific job, not a substitute for the rest of the toolkit.

We will reach for atomics in this chapter only when a single-word read-modify-write is the right shape for what we want to express. For everything else, the locks and the condition variables earn their keep.

12.2.6 A First Decision Tree

When you face a piece of shared state and you are deciding how to protect it, work the questions in this order. The first question that yields a definitive answer ends the search.

1. **Is the state a single word that needs a single read-modify-write?** Use an atomic. (Examples: a generation counter, a flag word.)
2. **Does the state have a composite invariant that spans more than one field, and is the access in process context?** Use an `MTX_DEF` mutex. (Examples: the `cbuf` head/used pair, a queue head/tail pair.)
3. **Is the access in interrupt context, or must the critical section disable preemption?** Use an `MTX_SPIN` mutex.
4. **Is the state read frequently from many threads but written rarely?** Use `sx(9)` for sleepable callers (most driver code) or `rw(9)` for short critical sections that may run in interrupt context.
5. **Do you need to wait until a specific condition becomes true (not just acquire a lock)?** Use a condition variable (`cv(9)`) paired with the mutex that protects the condition. The older `mtx_sleep/wakeup` channel mechanism is the legacy alternative; new code should prefer `cv(9)`.
6. **Do you need to bound a wait by wall time?** Use the timed variant (`cv_timedwait_sig`, `mtx_sleep` with a non-zero `timo` argument, `msleep_sbt(9)`) and design the caller to handle `EWOULDBLOCK`.

The tree compresses into a short slogan: *atomic for a word, mutex for a structure, sx for read-mostly, cv for waiting, timed for bounded waiting, spin only when you must.*

12.2.7 A Worked Decision: Where Each myfirst State Sits

Walk the tree against each piece of state in your current driver. The exercise is short and useful.

- The `cbuf` indices and backing memory: composite invariant, process context. Use an `MTX_DEF` mutex. (This is what Chapter 11 chose.)
- `sc->bytes_read`, `sc->bytes_written`: high-frequency counters, rarely read. Use `counter(9)` per-CPU counters. (This is what Chapter 11 migrated to.)
- `sc->open_count`, `sc->active_fhs`: low-frequency integers, fine under the same mutex as the `cbuf`. No reason to split them out.
- `sc->is_attached`: a flag, read often at handler entry, written once per attach/detach. The Chapter 11 design reads it without the mutex as an optimization, re-checks after every sleep, and writes it under the mutex.
- The “is the buffer empty?” condition that read waiters block on: a coordinated wait. Currently uses `mtx_sleep(&sc->cb, ...)`. Section 2 will replace this with a named condition variable.

- The “is there room in the buffer?” condition that write waiters block on: another coordinated wait, currently sharing the same channel. Section 2 will give it its own condition variable.
- A future configuration subsystem (added in Section 5): read frequently by every I/O call, written occasionally by a sysctl handler. Use `sx(9)`.

Notice how the tree did the work. We did not have to invent a custom design for any of these; we asked the questions and the right primitive fell out.

12.2.8 Real-World Analogy: Doors, Hallways, and Whiteboards

A small analogy for readers who like them. Imagine a research lab.

The `cbuf` is a precision instrument that only one person at a time can operate. The lab installs a door with a single key on it. Anyone who wants to use the instrument must take the key. While they have the key, nobody else can enter. That is a mutex.

The lab has a status whiteboard that lists the current calibration of the instrument. Anyone may read the whiteboard at any time; they do not interfere with each other. Only the lab manager updates the whiteboard, and they only do so after waiting for everyone else to step away. That is a shared/exclusive lock.

The lab has a coffee pot. People who want coffee but find the pot empty leave a note on the bulletin board: “I am in the lounge; wake me when there is coffee.” When somebody brews a fresh pot, they check the bulletin board and tap the shoulders of everyone whose note is for “coffee”, regardless of how long ago they wrote it. That is a condition variable.

The same person who left the coffee note may also leave a second note: “but only wait fifteen minutes; if no coffee by then, I am going to the cafeteria.” That is a timed wait.

Each mechanism in the lab matches a real coordination problem. None of them is a substitute for any of the others. The same is true in the kernel.

12.2.9 Comparing Primitives Side by Side

Sometimes seeing the primitives next to each other in a single table makes the choice immediate. The properties that differ across them are: whether they block or spin, whether they support shared (multi-reader) access, whether they are sleepable from the perspective of the holder, whether they support priority propagation, whether they are interruptible by signals, and whether the calling context can include sleeping operations.

	<code>mtx(9)</code>	<code>mtx(9)</code>			
Property	<code>MTX_DEF</code>	<code>MTX_SPIN</code>	<code>sx(9)</code>	<code>rw(9)</code>	<code>cv(9)</code>
Behaviour when contended	Sleep	Spin	Sleep	Spin	Sleep
Multiple holders	No	No	Yes (shared)	Yes (read)	n/a (waiters)

Property	mtx(9) MTX_DEF	mtx(9) MTX_SPIN	sx(9)	rw(9)	cv(9)
Caller may sleep while holding	Yes	No	Yes	No	n/a
Priority propagation	Yes	No (interrupts disabled)	No	Yes	n/a
Signal-interruptible variant	n/a	n/a	_sig	No	_sig
Has timed wait variant	mtx_sleep w/ timo	n/a	n/a	n/a	cv_timedwait
Suitable in interrupt context	No	Yes	No	Yes (with care)	No

Two things stand out. First, the column for `cv(9)` does not really fit the same questions because `cv` is not a lock; it is a wait primitive. We include it in the comparison because the choice “should I wait or spin?” is essentially the same as “should I block on a `cv` or spin on an `MTX_SPIN`?”. Second, the priority propagation column distinguishes `mtx(9)` and `rw(9)` from `sx(9)`. `sx(9)` does not propagate priority because its sleep queues do not support it. In practice this matters only for real-time workloads; ordinary drivers do not notice.

Use the table as a quick lookup when you face a new piece of state. The decision tree above gives you the *order* to ask the questions; the table gives you the *answer* once you have asked them.

12.2.10 A Note on Semaphores

FreeBSD also has a counting semaphore primitive (`sema(9)`) that is occasionally useful. A semaphore is a counter; threads decrement it (via `sema_wait` or `sema_trywait`) and block if the counter is zero; threads increment it (via `sema_post`) and may wake a waiter. The classic use is bounded-resource accounting: a queue with a maximum length where producers block when the queue is full and consumers block when it is empty.

Most driver problems that look semaphore-shaped can be solved equally well with a mutex plus a condition variable. The `cv` approach has the advantage that you can attach naming to each condition; the semaphore is anonymous. The semaphore has the advantage that the waiting and signalling are part of the primitive itself, with no need for a separate interlock.

This chapter does not use `sema(9)`. We mention it for completeness; if you encounter it in real driver source, you now know what shape it has.

12.2.11 Wrapping Up Section 1

Synchronization is broader than locking, locking is broader than mutual exclusion, and the FreeBSD toolkit gives you a different primitive for each shape of coordination problem you might encounter. Atomics for

single-word updates, mutexes for composite invariants, sx locks for read-mostly state, condition variables for coordinated waiting, timed sleeps for bounded waiting, and spin variants only when the calling context demands them.

That decision tree will guide every choice we make for the rest of the chapter. Section 2 starts with the first refactor: turning the anonymous `&sc->cb` wakeup channel from Chapter 11 into a pair of named condition variables.

12.3 Section 2: Condition Variables and Sleep/Wakeup

The `myfirst` driver as Chapter 11 left it has two distinct conditions that block the I/O paths. A reader sleeps when `cbuf_used(&sc->cb) == 0` and waits for “data has arrived”. A writer sleeps when `cbuf_free(&sc->cb) == 0` and waits for “room has appeared”. Both currently sleep on the same anonymous channel, `&sc->cb`. Both wakeups call `wakeup(&sc->cb)` after every state change, which wakes every sleeper regardless of which condition triggered the change.

That arrangement works. It is also wasteful, opaque, and harder to reason about than it needs to be. This section introduces condition variables (`cv(9)`), the cleaner FreeBSD primitive for the same coordinated-wait pattern, and walks through the refactor that gives each condition its own variable.

12.3.1 Why Mutex Plus Wakeup Is Not Enough

Chapter 11 used `mtx_sleep(chan, mtx, pri, wmesg, timo)` and `wakeup(chan)` to coordinate the read and write paths. The pair has the great virtue of simplicity: any pointer can be a channel, the kernel keeps a hash table of waiters per channel, and a wakeup on the right channel finds them all.

The vices appear as the driver grows.

The channel is anonymous. A reader of the source sees `mtx_sleep(&sc->cb, &sc->mtx, PCATCH, "myfrd", 0)` and has to infer, from the context and the `wmesg` string, what condition the thread is waiting for. There is nothing in `&sc->cb` that says “data available” instead of “room available” or “device detached”. The channel is just a pointer the kernel uses as a hash key; the meaning lives in the convention.

Multiple conditions share one channel. When `myfirst_write` finishes a write, it calls `wakeup(&sc->cb)`. That wakes every waiter on `&sc->cb`, including readers waiting for data (correct) and writers waiting for room (incorrect; a write does not free room, it consumes it). Each unwanted waiter goes back to sleep after rechecking its condition. This is the *thundering herd* problem in miniature: the wakeup is correct but expensive.

Lost wakeups are still possible if you are careless. If you ever drop the mutex between checking the condition and entering `mtx_sleep`, a wakeup can fire in that window and be missed. Chapter 11 explained that `mtx_sleep` itself is atomic with the lock drop, which closes the window; but the rule is implicit in the API and easy to violate when refactoring.

The `wmesg` argument is the only label. A debugging engineer who runs `procstat -kk` against a hung process sees `myfrd` and has to remember what that means. The string is at most seven characters; it is a hint, not a structured description.

Condition variables solve all four of these. Each `cv` has a name (its description string is checkable from `dt race` and `procstat`). Each `cv` represents exactly one logical condition; there is no question of which waiters a

`cv_signal` will affect. The `cv_wait` primitive enforces the atomicity of the lock drop by taking the mutex as an argument, so misuse is much harder. And the relationship between the waiter and the waker is expressed in the type itself: both sides reference the same struct `cv`.

12.3.2 What a Condition Variable Is

A **condition variable** is a kernel object that represents a logical condition some threads are waiting for and others will eventually signal. The condition variable does not store the condition; the condition lives in your driver's state, protected by your mutex. The condition variable is the rendezvous point: the place where waiters queue up and where wakers find them.

The data structure is small and lives in `/usr/src/sys/sys/condvar.h`:

```
struct cv {
    const char    *cv_description;
    int           cv_waiters;
};
```

Two fields: a description string used for debugging, and a count of currently-waiting threads (an optimization to skip the wakeup machinery when no one is waiting).

The API is also small:

```
void cv_init(struct cv *cvp, const char *desc);
void cv_destroy(struct cv *cvp);

void cv_wait(struct cv *cvp, struct mtx *mtx);
int cv_wait_sig(struct cv *cvp, struct mtx *mtx);
void cv_wait_unlock(struct cv *cvp, struct mtx *mtx);
int cv_timedwait(struct cv *cvp, struct mtx *mtx, int timo);
int cv_timedwait_sig(struct cv *cvp, struct mtx *mtx, int timo);

void cv_signal(struct cv *cvp);
void cv_broadcast(struct cv *cvp);
void cv_broadcastpri(struct cv *cvp, int pri);

const char *cv_wmesg(struct cv *cvp);
```

(The actual prototypes use `struct lock_object` rather than `struct mtx`; the macros in `condvar.h` substitute the right `&mtx->lock_object` pointer for you. The form above is the form you will write in driver code.)

A few rules and conventions matter from the start.

`cv_init` is called once, after the `cv` structure exists in memory and before any waiter or waker can reach it. The matching `cv_destroy` is called once, after every waiter has either woken up or been forced off the queue, and before the `cv` structure is freed. Lifetime mistakes here cause the same kind of catastrophic crashes that mutex lifetime mistakes cause.

`cv_wait` and its variants must be called with the interlock mutex *held*. Inside `cv_wait`, the kernel atomically drops the mutex and puts the calling thread on the `cv`'s wait queue. When the thread is awakened, the mutex is reacquired before `cv_wait` returns. From your code's perspective, the mutex is held both before and after the call; a different thread could not have observed the gap, even though the gap really existed. This is exactly the same atomic-drop-and-sleep contract `mtx_sleep` provides.

`cv_signal` wakes one waiter, and `cv_broadcast` wakes all waiters. Which waiter `cv_signal` picks is worth stating carefully. The `condvar(9)` man page only promises that it unblocks “one waiter”; it does *not* promise strict FIFO order, and your code must not rely on a specific ordering. What the current FreeBSD 14.3 implementation actually does, inside `sleepq_signal(9)` in `/usr/src/sys/kern/subr_sleepqueue.c`, is scan the `cv`'s sleep queue and pick the thread with the highest priority, breaking ties in favour of the thread that has been sleeping the longest. That is a useful mental model, but treat it as an implementation detail rather than an API guarantee. If correctness depends on which thread wakes up next, your design is probably wrong and should use a different primitive or an explicit queue. Both `cv_signal` and `cv_broadcast` are typically called with the interlock mutex held, although the rule is more about correctness of the surrounding logic than about the primitive itself: if you call `cv_signal` without the interlock, it is possible for a new waiter to arrive and miss the signal. The standard discipline is therefore “hold the mutex, change the state, signal, drop the mutex”.

`cv_wait_sig` returns nonzero if the thread was awakened by a signal (typically `EINTR` or `ERESTART`); zero if it was awakened by `cv_signal` or `cv_broadcast`. Drivers that want their blocking I/O paths to honour Ctrl-C use `cv_wait_sig`, not `cv_wait`. Section 3 explores the signal-handling rules in depth.

`cv_wait_unlock` is the rare variant for when the caller wants the interlock dropped on the *waiting* side and not reacquired on return. Useful in teardown sequences where the caller has no further business with the interlock once the wait completes. Drivers seldom need it; we mention it because you will see it in a few places in the FreeBSD tree, and the chapter does not use it further.

`cv_timedwait` and `cv_timedwait_sig` add a timeout in ticks. They return `EWOLDBLOCK` if the timeout fires before any wakeup arrives. Section 3 explains how to bound a blocking operation with these.

12.3.3 A Worked Refactor: Adding Two Condition Variables to `myfirst`

The Chapter 11 driver had one anonymous channel for both conditions. Stage 1 of this chapter splits it into two named condition variables: `data_cv` (“data is available to read”) and `room_cv` (“room is available to write”).

Add two fields to the `softc`:

```
struct myfirst_softc {
    /* ... existing fields ... */
    struct cv      data_cv;
    struct cv      room_cv;
    /* ... existing fields ... */
};
```

Initialize and destroy them in `attach` and `detach`:

```

static int
myfirst_attach(device_t dev)
{
    /* ... existing setup ... */
    cv_init(&sc->data_cv, "myfirst data");
    cv_init(&sc->room_cv, "myfirst room");
    /* ... rest of attach ... */
}

static int
myfirst_detach(device_t dev)
{
    /* ... existing teardown that cleared is_attached and woke sleepers ... */
    cv_destroy(&sc->data_cv);
    cv_destroy(&sc->room_cv);
    /* ... rest of detach ... */
}

```

A small but important subtlety: detach must not destroy a cv that still has waiters. The Chapter 11 detach path already wakes sleepers and refuses to proceed while `active_fhs > 0`, which means by the time we reach `cv_destroy`, no descriptor is open and no thread can still be inside `cv_wait`. We add a `cv_broadcast(&sc->data_cv)` and `cv_broadcast(&sc->room_cv)` immediately before destroying, as belt-and-braces, in case any background path ever does sneak in.

Update the wait helpers to use the new variables:

```

static int
myfirst_wait_data(struct myfirst_softc *sc, int ioflag, ssize_t nbefore,
    struct uio *uio)
{
    int error;

    MYFIRST_ASSERT(sc);
    while (cbuf_used(&sc->cb) == 0) {
        if (uio->uio_resid != nbefore)
            return (-1);
        if (ioflag & IO_NDELAY)
            return (EAGAIN);
        error = cv_wait_sig(&sc->data_cv, &sc->mtx);
        if (error != 0)
            return (error);
        if (!sc->is_attached)
            return (ENXIO);
    }
    return (0);
}

```

```

}

static int
myfirst_wait_room(struct myfirst_softc *sc, int ioflag, ssize_t nbefore,
    struct uio *uio)
{
    int error;

    MYFIRST_ASSERT(sc);
    while (cbuf_free(&sc->cb) == 0) {
        if (uio->uio_resid != nbefore)
            return (-1);
        if (ioflag & IO_NDELAY)
            return (EAGAIN);
        error = cv_wait_sig(&sc->room_cv, &sc->mtx);
        if (error != 0)
            return (error);
        if (!sc->is_attached)
            return (ENXIO);
    }
    return (0);
}

```

Three things changed and nothing else. `mtx_sleep(&sc->cb, &sc->mtx, PCATCH, "myfrd", 0)` became `cv_wait_sig(&sc->data_cv, &sc->mtx)`. The corresponding line in the write path became `cv_wait_sig(&sc->room_cv, &sc->mtx)`. The `wmesg` string is gone (the `cv`'s description string takes its place), the channel is now a real object with a name, and the `PCATCH` flag is implicit in the `_sig` suffix.

Update the wakers. After a successful read, instead of waking everyone on `&sc->cb`, wake only the writers waiting for room:

```

got = myfirst_buf_read(sc, bounce, take);
fh->reads += got;
MYFIRST_UNLOCK(sc);

if (got > 0) {
    cv_signal(&sc->room_cv);
    selwakeup(&sc->wsel);
}

```

After a successful write, wake only the readers waiting for data:

```

put = myfirst_buf_write(sc, bounce, want);
fh->writes += put;

```

```
MYFIRST_UNLOCK(sc);

if (put > 0) {
    cv_signal(&sc->data_cv);
    selwakeup(&sc->rsel);
}
```

Two improvements come together. First, a successful read no longer wakes other readers (a wasted wakeup that immediately goes back to sleep); only writers, who actually have a use for the freed room, are woken. Symmetric on the write side. Second, the source is now self-explanatory: `cv_signal(&sc->room_cv)` reads as “there is room now”; the reader does not have to remember what `&sc->cb` means.

Notice we added a `if (got > 0)` and `if (put > 0)` guard before the signal. There is no point waking a sleeper if nothing changed; the empty signal is benign but cheap to skip. This is a small optimization and a clarification: the signal is announcing a state change, and the guard says so.

`cv_signal` instead of `cv_broadcast`: we wake one waiter per state change, not all of them. The state change (one byte freed by a read, one byte added by a write) is enough for one waiter to make progress. If multiple waiters are blocked, the next signal will wake the next one. This is the per-event correspondence the `cv` API encourages.

12.3.4 When to Signal Versus Broadcast

`cv_signal` wakes one waiter. `cv_broadcast` wakes all of them. The choice matters more than people expect.

Use `cv_signal` when:

- The state change is a per-event update (one byte arrived; one descriptor freed; one packet enqueued). One waiter making progress is enough; the next event will wake the next waiter.
- All waiters are equivalent and any one of them can consume the change.
- The cost of waking a waiter that has nothing to do is non-trivial (because the waiter immediately re-checks the condition and sleeps again).

Use `cv_broadcast` when:

- The state change is global and all waiters need to know (the device is being detached; the configuration changed; the buffer was reset).
- The waiters are not equivalent; each may be waiting on a slightly different sub-condition that the broadcast resolves.
- You want to avoid the bookkeeping of figuring out which subset of waiters can proceed, at the cost of waking some that will go back to sleep.

For the `myfirst` data and room conditions, `cv_signal` is the right call. For the detach path, `cv_broadcast` is the right call: the detach must wake every blocked thread so they can return `ENXIO` and exit cleanly.

Add the broadcasts to detach:

```
MYFIRST_LOCK(sc);
sc->is_attached = 0;
cv_broadcast(&sc->data_cv);
cv_broadcast(&sc->room_cv);
MYFIRST_UNLOCK(sc);
```

That replaces the Chapter 11 `wakeup(&sc->cb)`. The two broadcasts wake every reader and every writer that may be sleeping; each one re-checks `is_attached`, sees that it is now zero, and returns `ENXIO`.

12.3.5 A Subtle Trap: `cv_signal` Without the Mutex

The standard discipline says “hold the mutex, change the state, signal, drop the mutex”. You may have noticed our refactor signals *after* dropping the mutex (the `MYFIRST_UNLOCK(sc)` precedes the `cv_signal`). Is that wrong?

It is not wrong, and the reason is worth understanding.

The race the discipline is meant to prevent is this: a waiter checks the condition (false), is about to call `cv_wait`, but has dropped the mutex. The waker now changes the state, sees no one in the `cv`’s wait queue (because the waiter has not yet enqueued), and skips the signal. The waiter then enqueues and sleeps forever.

`cv_wait` itself prevents that race by enqueueing the waiter on the `cv` *before* dropping the mutex. The kernel’s internal `cv`-queue lock is acquired while the caller’s mutex is still held, the thread is added to the wait queue, and only then is the caller’s mutex dropped and the thread descheduled. Any subsequent `cv_signal` on that `cv`, with or without the caller’s mutex held, will find the waiter and wake it.

The discipline of signalling under the mutex is therefore a defensive convention rather than a strict requirement. We follow it for the simple cases (because it is harder to get wrong) and we relax it for the case where signalling outside the mutex is a measurable improvement (it lets the woken thread acquire the mutex without contending with the signaller). For `myfirst`, signalling after `MYFIRST_UNLOCK(sc)` shaves a few cycles off the wake path; for safety we still take care to not allow a window between the state change and the signal where the state could be reverted. In our refactor, the only thread that can revert the state is also operating under the mutex, so the window is closed.

If you are uncertain, signal under the mutex. It is the safer default and the cost is negligible.

12.3.6 Verifying the Refactor

Build the new driver and load it on a `WITNESS` kernel. Run the Chapter 11 stress kit. Three things should happen:

- All tests pass with the same byte-count semantics as before.
- `dmesg` is silent. No new warnings.
- `procstat -kk` against a sleeping reader now shows the `cv`’s description in the wait-channel column. Tools that report `wmesg` truncate to `WMESGLEN` (eight characters, defined in `/usr/src/sys/sys/user.h`); a description of “myfirst data” therefore shows up as “myfirst ” in `procstat` and `ps`. The full description string remains visible to `dtrace` (which reads `cv_description` directly) and in the source.

If you want the truncated form to be more informative, choose shorter descriptions such as "mfdata" and "mfroom"; the chapter keeps the longer, more readable names because dtrace and the source code use the full string and that is where you spend most of your debugging time.

lockstat(1) will show fewer cv events than the old wakeup mechanism produced wakeups, because the per-condition signalling does not wake threads that have nothing to do. This is the throughput improvement we expected.

12.3.7 A Mental Model: How a cv_wait Plays Out

For readers who learn best from a step-by-step picture, here is the sequence of events when a thread calls cv_wait_sig and is later signalled.

Time t=0: thread A is in myfirst_read. The cbuf is empty.

Time t=1: thread A calls MYFIRST_LOCK(sc). The mutex is acquired. Thread A is now the only thread in any critical section protected by sc->mtx.

Time t=2: thread A enters the wait helper. The check cbuf_used(&sc->cb) == 0 is true. Thread A calls cv_wait_sig(&sc->data_cv, &sc->mtx).

Time t=3: inside cv_wait_sig, the kernel takes the cv-queue spin lock for data_cv, increments data_cv.cv_waiters, and atomically does two things: drops sc->mtx and adds thread A to the cv's wait queue. Thread A's state changes to "sleeping on data_cv".

Time t=4: thread A is descheduled. The CPU runs other threads.

Time t=5: thread B enters myfirst_write from another process. Thread B calls MYFIRST_LOCK(sc). The mutex is currently free; thread B acquires it.

Time t=6: thread B reads from user space (uiomove), commits bytes to the cbuf, updates counters. Thread B calls MYFIRST_UNLOCK(sc).

Time t=7: thread B calls cv_signal(&sc->data_cv). The kernel takes the cv-queue spin lock, finds thread A on the wait queue, decrements cv_waiters, removes thread A from the queue, and marks thread A as runnable.

Time t=8: the scheduler decides thread A is the highest-priority runnable thread (or one of several; FIFO among equal). Thread A is scheduled onto a CPU.

Time t=9: thread A resumes inside cv_wait_sig. The function reacquires sc->mtx (this may itself block if another thread now holds the mutex; if so, thread A is added to the mutex's wait list). Thread A returns from cv_wait_sig with return value 0 (normal wakeup).

Time t=10: thread A continues in the wait helper. The while (cbuf_used(&sc->cb) == 0) check is now false (thread B added bytes). The loop exits.

Time t=11: thread A reads from the cbuf and proceeds.

Three things to take away from the picture. First, the lock state is consistent at every step. The mutex is either held by exactly one thread or held by none; thread A's view of the world is the same before the wait as after. Second, the wakeup is decoupled from the actual scheduling; thread B did not hand the CPU directly to thread A. Third, there is a window between t=9 and t=10 where thread A holds the mutex and another writer could (if

it had been waiting) potentially fill the buffer further. That is fine; thread A's check is on the `cbuf` state at $t=10$, not at $t=7$.

This sequence is the canonical “wait, signal, wake, re-check, proceed” pattern. Every `cv` use in the chapter is an instance of it.

12.3.8 A Look at `kern_condvar.c`

If you have ten minutes, open `/usr/src/sys/kern/kern_condvar.c` and skim. Three functions are particularly worth seeing:

`cv_init` (top of file): very short. It just initializes the description and zeroes the waiter count.

`_cv_wait` (mid-file): the core blocking primitive. It takes the `cv`-queue spin lock, increments `cv_waiters`, drops the caller's interlock, calls the sleep queue machinery to enqueue the thread and yield, and on return decrements `cv_waiters` and reacquires the interlock. The atomic drop-and-sleep is performed by the sleep queue layer, exactly the same machinery that backs `mtx_sleep`. There is nothing magical about `cv` on top of sleep queues; it is a thin, named interface.

`cv_signal` and `cv_broadcastpri`: each takes the `cv`-queue spin lock, finds one (or all) waiters, and uses `sleepq_signal` or `sleepq_broadcast` to wake them.

The takeaway: condition variables are a thin, structured layer on top of the same sleep queue primitives `mtx_sleep` uses. They are not slower; they are not faster; they are clearer.

12.3.9 Wrapping Up Section 2

The refactor in this section gives each waiting condition its own object, its own name, its own waiters' queue, and its own signal. The driver behaves the same to user space, but it now reads more honestly: `cv_signal(&sc->room_cv)` says “there is room”, which is what we mean. The `WITNESS` discipline is preserved; the `mtx_assert` calls in the helpers still hold; the test kit continues to pass. We have moved one level up the synchronization vocabulary without losing any of the safety Chapter 11 built.

Section 3 turns to the orthogonal question of *how long should we wait?*. Indefinite blocking is convenient for the implementation but harsh on the user. Timed waits and signal-aware waits are how a well-behaved driver responds to the world.

12.4 Section 3: Handling Timeouts and Interruptible Sleep

A blocking primitive is, by default, indefinite. A reader that calls `cv_wait_sig` when the buffer is empty will sleep until either someone calls `cv_signal` (or `cv_broadcast`) on the same `cv`, or a signal is delivered to the reader's process. From the kernel's point of view, “indefinite” is a perfectly respectable answer. From the user's point of view, “indefinite” is a hang.

This section is about the two ways the FreeBSD synchronization primitives let you bound a wait: by a wall-clock timeout and by an interruption from a signal. Both are simple to use and both have surprisingly subtle rules around return values. We start with the easier one and work up.

12.4.1 What Goes Wrong With Indefinite Sleeps

Three real problems push us to use timed and interruptible sleeps in a driver.

Hung programs. A user runs `cat /dev/myfirst` in a terminal. There is no producer. The `cat` blocks in `read(2)`, which blocks in `myfirst_read`, which blocks in `cv_wait_sig`. The user presses Ctrl-C. If the wait is interruptible (the `_sig` variant), the kernel delivers `EINTR` and the user gets their shell back. If it is not (`cv_wait` without `_sig`), the kernel ignores the signal and the user has to use Ctrl-Z and `kill %1` from another terminal. Most users do not know how to do that. They reach for the reset button.

Stuck progress. A device driver waits for an interrupt that never arrives because the hardware is wedged. The driver's I/O thread sleeps forever. The whole system slowly fills with processes blocked on this driver. Eventually an administrator notices, but by then there is nothing to do but reboot. A bounded wait would have caught this much earlier.

Bad user experience. A network protocol expects a response within a specified time. A storage operation expects a completion within a service-level agreement. Neither is well-served by a primitive that can wait forever. The driver should be able to enforce a deadline and return a clean error when the deadline is missed.

The FreeBSD primitives that solve these are `cv_wait_sig` and `cv_timedwait_sig`, with the older `mtx_sleep` and `tsleep` family providing the same capabilities through a different shape. We have already met `cv_wait_sig` in Section 2. Here we look more carefully at what its return value tells us and how to add an explicit timeout.

12.4.2 The Three Wait Outcomes

Any blocking sleep primitive can return for one of three reasons:

1. **A normal wakeup.** Some other thread called `cv_signal`, `cv_broadcast`, or (in the legacy API) `wakeup`. The condition this thread was waiting for has changed, and the thread should re-check it.
2. **A signal was delivered to the process.** The thread is being asked to abandon the wait so the signal handler can run. The driver typically returns `EINTR` to user space, which is also the sleep primitive's return value.
3. **A timeout fired.** The thread was waiting with a deadline and the deadline expired before any wakeup arrived. The sleep primitive returns `EWOLDBLOCK`.

The driver's job is to figure out which of the three happened and respond appropriately.

The first case is the easy one. The thread re-checks its condition (the `while` loop around `cv_wait_sig` is what does this); if the condition is now true, the loop ends and the I/O proceeds; if not, the thread sleeps again.

The second case is more interesting. The kernel delivers a signal not to a *thread* but to a *process*. A signal can be a serious condition (`SIGTERM`, `SIGKILL`) or a routine one (`SIGINT` from Ctrl-C, `SIGALRM` from a timer). The thread that was sleeping needs to return to user space promptly so the signal handler can run. The convention is that the sleep primitive returns `EINTR` (interrupted system call), the driver returns `EINTR` from its handler, and the kernel either restarts the system call (if the handler returned with `SA_RESTART`) or returns `EINTR` to user space (if not).

The third case is the bounded-wait case. The driver typically maps `EWOLDBLOCK` to either `EAGAIN` (try again later) or to a more specific error (`ETIMEDOUT`, where appropriate).

12.4.3 EINTR, ERESTART, and the Restart Question

A subtlety lurks in case 2 that is worth understanding before the chapter goes any further.

When `cv_wait_sig` is interrupted by a signal, the actual return value is one of two things:

- `EINTR` if the signal's disposition is "do not restart system calls". The kernel returns `EINTR` to user space, and `read(2)` reports -1 with `errno == EINTR`. The user program is responsible for retrying if it wants to.
- `ERESTART` if the signal's disposition is "restart system calls" (the `SA_RESTART` flag). The kernel transparently re-enters the syscall and the wait happens again. The user program does not see the interruption.

A driver should not return `ERESTART` directly to user space; it is an internal sentinel for the syscall layer. If the driver returns `ERESTART` from its handler, the syscall layer knows to restart. If the driver returns `EINTR`, the syscall layer returns `EINTR` to user space.

The convention most drivers follow: pass the return value of `cv_wait_sig` through unchanged. If you got `EINTR`, the driver returns `EINTR`. If you got `ERESTART`, the driver returns `ERESTART`. The kernel takes it from there. The Chapter 11 driver did this implicitly; the Chapter 12 refactor in Section 2 continues to do so:

```
error = cv_wait_sig(&sc->data_cv, &sc->mtx);
if (error != 0)
    return (error);
```

Returning error directly is the right move. Chapter 12 changes nothing about this rule; it just makes the rule visible in the new APIs.

12.4.4 Adding a Timeout to the Read Path

Now the bounded-wait case. Suppose we want `myfirst_read` to optionally wait for at most some configurable duration before returning `EAGAIN` if no data arrives. (We use `EAGAIN` rather than `ETIMEDOUT` because `EAGAIN` is the conventional UNIX answer to "the operation would block; try again later".)

The driver needs three things:

1. A configuration value for the timeout (in milliseconds, say). Zero means "block indefinitely as before".
2. A way to convert the timeout to ticks, since `cv_timedwait_sig` takes its `timo` argument in ticks.
3. A loop that handles the three outcomes correctly: normal wakeup, signal interruption, timeout.

Add the configuration field to the softc:

```
int    read_timeout_ms; /* 0 = no timeout */
```

Initialize it in `attach`:

```
sc->read_timeout_ms = 0;
```

Expose it as a `sysctl`:

```

SYSCTL_ADD_INT(&sc->sysctl_ctx, SYSCTL_CHILDREN(sc->sysctl_tree),
    OID_AUTO, "read_timeout_ms", CTLFLAG_RW,
    &sc->read_timeout_ms, 0,
    "Read timeout in milliseconds (0 = block indefinitely)");

```

We use a plain `SYSCTL_ADD_INT` for now; the value is one integer, the read is atomic at the word level on amd64, and a slightly stale value is acceptable. (Section 5 will give us a more disciplined way to handle configuration changes.)

Update the wait helper:

```

static int
myfirst_wait_data(struct myfirst_softc *sc, int ioflag, ssize_t nbefore,
    struct uio *uio)
{
    int error, timo;

    MYFIRST_ASSERT(sc);
    while (cbuf_used(&sc->cb) == 0) {
        if (uio->uio_resid != nbefore)
            return (-1);
        if (ioflag & IO_NDELAY)
            return (EAGAIN);

        timo = sc->read_timeout_ms;
        if (timo > 0) {
            int ticks_total = (timo * hz + 999) / 1000;
            error = cv_timedwait_sig(&sc->data_cv, &sc->mtx,
                ticks_total);
        } else {
            error = cv_wait_sig(&sc->data_cv, &sc->mtx);
        }
        if (error == EWOULDBLOCK)
            return (EAGAIN);
        if (error != 0)
            return (error);
        if (!sc->is_attached)
            return (ENXIO);
    }
    return (0);
}

```

A few details deserve commentary.

The $(\text{timo} * \text{hz} + 999) / 1000$ arithmetic converts milliseconds to ticks, rounding up. We want at least the requested wait, never less. A timeout of 1 ms on a 1000 Hz kernel becomes 1 tick. A timeout of 1 ms on a 100

Hz kernel becomes 1 tick (rounded up from 0.1). A timeout of 5500 ms becomes 5500 ticks at 1000 Hz, or 550 at 100 Hz.

The branch on `timo > 0` picks `cv_timedwait_sig` when a positive timeout is requested and `cv_wait_sig` (no timeout) when not. We could always call `cv_timedwait_sig` with `timo = 0`, but the `cv` API treats `timo = 0` as “wait indefinitely”, and the behaviour is identical to `cv_wait_sig`. The explicit branch makes the intent clearer for a reader.

The `EWouldBlock -> EAGAIN` translation gives user space the conventional “try again” indication. A user program that gets `EAGAIN` knows what to do; a user program that gets `ETIMEDOUT` would have to learn a new error code.

The `is_attached` re-check after every sleep remains. Even with a bounded wait, the device might have been detached during the sleep; the `cv` broadcast in `detach` (added in Section 2) wakes us; the timeout itself does not skip the post-sleep checks.

Apply the symmetric change to `myfirst_wait_room` if you want bounded writes, with a separate `write_timeout_ms` `sysctl`. The companion source does both.

12.4.5 Verifying the Timeout

A small user-space tester confirms the new behaviour. Set the timeout to 100 ms, open the device with no producer, and read. You should see `read(2)` return -1 with `errno == EAGAIN` after approximately 100 ms, not block forever.

```
/* timeout_tester.c: confirm bounded reads. */
#include <err.h>
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <sys/sysctl.h>
#include <sys/time.h>
#include <unistd.h>

#define DEVPATH "/dev/myfirst"

int
main(void)
{
    int timeout_ms = 100;
    size_t sz = sizeof(timeout_ms);

    if (sysctlbyname("dev.myfirst.0.read_timeout_ms",
        NULL, NULL, &timeout_ms, sz) != 0)
        err(1, "sysctlbyname set");
}
```

```
int fd = open(DEVPATH, O_RDONLY);
if (fd < 0)
    err(1, "open");

char buf[1024];
struct timeval t0, t1;
gettimeofday(&t0, NULL);
ssize_t n = read(fd, buf, sizeof(buf));
gettimeofday(&t1, NULL);
int saved = errno;

long elapsed_ms = (t1.tv_sec - t0.tv_sec) * 1000 +
    (t1.tv_usec - t0.tv_usec) / 1000;
printf("read returned %zd, errno=%d (%s) after %ld ms\n",
    n, saved, strerror(saved), elapsed_ms);

close(fd);
return (0);
}
```

Run with no producer attached. Expect output similar to:

```
read returned -1, errno=35 (Resource temporarily unavailable) after 102 ms
```

Errno 35 on FreeBSD is EAGAIN. The 102 ms is the 100 ms timeout plus a couple of milliseconds of scheduling jitter.

Reset the sysctl to zero (`sysctl -w dev.myfirst.0.read_timeout_ms=0`) and rerun. Now `read(2)` blocks until you press Ctrl-C, at which point it returns -1 with `errno == EINTR`. The interruptibility (the `_sig` suffix) and the timeout (the `_timedwait` variant) are independent capabilities. We can have neither, either, or both, and the API exposes each on its own switch.

12.4.6 Choosing Between EAGAIN and ETIMEDOUT

When the timeout fires, the driver chooses what error to report. The two reasonable choices are EAGAIN and ETIMEDOUT.

EAGAIN (errno value 35 on FreeBSD; symbolic value `EWOULDBLOCK` is #defined to the same number in `/usr/src/sys/sys/errno.h`) is the conventional UNIX answer to “the operation would block”. User programs that handle `O_NONBLOCK` understand it. Many user programs already retry on EAGAIN. Returning EAGAIN for a timeout is a safe default; it does the right thing for the majority of callers.

ETIMEDOUT (errno value 60 on FreeBSD) is more specific: “the operation has a deadline and the deadline expired”. Network protocols use it; it means something different from “would block right now”. A user program that wants to distinguish “no data yet, retry” from “no data after the agreed deadline, give up” needs ETIMEDOUT.

For `myfirst`, we use `EAGAIN`. The driver does not have a deadline contract with the caller; the timeout is a politeness, not a guarantee. Other drivers may make the other choice; both are legal.

12.4.7 A Note on Fairness Under Timeouts

The timed wait does not change the `cv`'s fairness story. `cv_timedwait_sig` is implemented in terms of the same sleep queue used by `cv_wait_sig`. When a wakeup arrives, the sleep queue picks the highest-priority waiter (FIFO among equal priorities) regardless of whether each waiter has a timeout pending. The timeout is a per-waiter watchdog; it does not affect the order in which non-timed-out waiters are woken.

Practical consequence: a thread with a 50 ms timeout and a thread without a timeout, both waiting on the same `cv`, will both be woken by `cv_signal` in the order the sleep queue chooses. The 50 ms thread does not get priority. If you need the timed waiter to be woken first, you have a different design problem (priority queues, separate `cvs` per priority class) that goes beyond this chapter.

For `myfirst`, all readers are equivalent and the lack of timeout-based prioritization is fine.

12.4.8 When to Use a Timeout

Timeouts are not free. Each timed wait sets up a callout that fires the `cv`'s wakeup if no real wakeup arrives first. The callout has a small per-tick cost and contributes to the kernel's overall callout pressure. Drivers that use timeouts for every blocking call create more callout traffic than drivers that block indefinitely.

Three rules of thumb:

- Use a timeout when the caller has a real deadline (a network protocol, a hardware watchdog, a user-visible response).
- Use a timeout when the wait has a “this should not be possible” backstop. A driver that “just in case” sets a 60 second timeout on a wait that should normally complete in microseconds is using the timeout as a sanity check, not as a deadline. That is fine.
- Do not use a timeout when the wait is naturally indefinite (a `cat /dev/myfirst` on an idle device should block until data arrives or the user gives up; either is fine, neither needs a timeout).

The `myfirst` driver in this chapter exposes a per-device `sysctl` that lets the user choose. The default of zero (block indefinitely) is the right default for a pseudo-device. Real drivers may have stronger opinions.

12.4.9 Sub-Tick Precision With `sbintime_t`

The `cv_timedwait` and `cv_timedwait_sig` macros take their timeout in ticks. A tick on FreeBSD 14.3 is typically one millisecond (because `hz=1000` is the default), so tick precision is millisecond precision. For most driver use cases this is enough. Network and storage drivers occasionally want microsecond precision, and the `_sbt` (scaled binary time) variants are how you get it.

The relevant primitives:

```
int cv_timedwait_sbt(struct cv *cvp, struct mtx *mtx,
                    sbintime_t sbt, sbintime_t pr, int flags);
```

```
int cv_timedwait_sig_sbt(struct cv *cvp, struct mtx *mtx,
                        sbintime_t sbt, sbintime_t pr, int flags);
int msleep_sbt(void *chan, struct mtx *mtx, int pri,
               const char *wmesg, sbintime_t sbt, sbintime_t pr, int flags);
```

The `sbt` argument is the timeout, expressed as an `sbintime_t` (a 64-bit integer where the upper 32 bits are seconds and the lower 32 bits are a binary fraction of a second). The `pr` argument is the precision: how much wiggle the kernel is allowed to have when scheduling the timer (used for power-saving coalescing of timer interrupts). The `flags` argument is one of `C_HARDLOCK`, `C_ABSOLUTE`, `C_DIRECT_EXEC`, etc., which control how the timer is registered.

For a 250-microsecond timeout:

```
sbintime_t sbt = 250 * SBT_1US; /* 250 microseconds */
int err = cv_timedwait_sig_sbt(&sc->data_cv, &sc->mtx, sbt,
                               SBT_1US, C_HARDLOCK);
```

The `SBT_1US` constant (defined in `/usr/src/sys/sys/time.h`) is one microsecond as an `sbintime_t`. Multiplying by 250 gives 250 microseconds. The precision argument `SBT_1US` says “I am happy with one-microsecond precision”; the kernel will not coalesce this timer with other timers more than 1 microsecond apart.

For 5 seconds:

```
sbintime_t sbt = 5 * SBT_1S;
int err = cv_timedwait_sig_sbt(&sc->data_cv, &sc->mtx, sbt,
                               SBT_1MS, C_HARDLOCK);
```

Five-second wait with millisecond precision. The kernel may coalesce up to 1 ms.

For most driver code the millisecond-tick API (`cv_timedwait_sig` with a tick count) is the right level of precision. Reach for `_sbt` when you have a real reason: a network protocol with sub-millisecond timing, a hardware controller with a microsecond-scale watchdog, a measurement where the sleep itself contributes to the result.

12.4.10 What Happens Inside `cv_timedwait_sig`

Conceptually, `cv_timedwait_sig` does the same thing as `cv_wait_sig` but also schedules a callout that will fire the `cv`’s signal if no real signal arrives first. The implementation lives in `/usr/src/sys/kern/kern_condvar.c` in `_cv_timedwait_sig_sbt`. Three observations are worth carrying with you.

First, the callout is registered while the interlock mutex is held, then the thread sleeps. If the callout fires while the thread is sleeping, the kernel marks the thread as awake-with-timeout. The thread returns from the sleep with `EWOLDBLOCK`.

Second, if a real `cv_signal` arrives before the timeout, the callout is cancelled when the thread wakes. The cancellation is racy in principle (the callout could fire just after the thread wakes for a real reason), but the

kernel handles this by checking whether the thread is still sleeping when the callout fires; if not, the callout is a no-op.

Third, every timed wait creates and tears down a callout. On a system with thousands of concurrent timed waits, the callout machinery becomes a measurable cost. For a single driver with at most a few dozen waiters, the cost is negligible.

These details are not things you need to memorize. They explain, however, why a driver that uses timed waits everywhere may show more activity in the callout subsystem than a driver that uses indefinite waits with a separate watchdog thread. If you ever wonder why your driver is producing many callout events, the timed waits are a likely cause.

12.4.11 Wrapping Up Section 3

Bounded waits and interruptible waits are the two ways a kernel sleep primitive cooperates with the world outside it. We added both to the `myfirst` blocking paths: `cv_wait_sig` was already there; `cv_timedwait_sig` is the new addition, gated by a `sysctl`. The user-space test confirms that a Ctrl-C and a 100-millisecond deadline both produce the expected return value; the driver reports `EINTR` and `EAGAIN` respectively.

Section 4 turns to a different shape of synchronization entirely: shared/exclusive locks, where many threads may read at once and only writers must wait their turn.

12.5 Section 4: The `sx(9)` Lock: Shared and Exclusive Access

The mutex `myfirst` uses today protects the `cbuf`, which has a composite invariant. That is the right primitive for that job. Not every piece of state has a composite invariant, however. Some state is read often, written rarely, and never spans more than the field being read. For that state, serializing every reader through a mutex is wasted serialization. A reader-writer lock fits the shape better.

This section introduces `sx(9)`, FreeBSD's sleepable shared/exclusive lock. We first explain what shared/exclusive means and why it matters, then walk through the API, then talk briefly about the spin-mode sibling `rw(9)`, and finish with the rules that distinguish the two and place each in the right context.

12.5.1 What Shared and Exclusive Mean

A **shared lock** (also called a *read lock*) allows multiple holders simultaneously. A thread that holds the lock in shared mode is guaranteed that no other thread is currently holding it in *exclusive* mode. Shared holders may execute concurrently; they do not see each other.

An **exclusive lock** (also called a *write lock*) is held by exactly one thread at a time. A thread holding the lock in exclusive mode is guaranteed that no other thread holds it in any mode.

A lock can transition between modes in two directions:

- **Downgrade:** a holder of an exclusive lock can convert it to a shared lock without releasing it. The conversion is non-blocking; immediately after, the original holder still has the lock (now in shared mode), and other readers may proceed.

- **Upgrade:** a holder of a shared lock can attempt to convert it to an exclusive lock. The attempt may fail if other shared holders are still present. The standard primitive is `sx_try_upgrade`, which returns success/failure rather than blocking.

The asymmetry of the upgrade (try, may fail) reflects a fundamental difficulty: if multiple shared holders both try to upgrade at once, they would deadlock waiting for each other. The non-blocking `sx_try_upgrade` lets one succeed while the others fail and have to release-and-reacquire as exclusive.

Shared/exclusive locks are the right primitive when the access pattern is *many readers, occasional writer*. Examples in the FreeBSD kernel include the namespace lock for `sysctl`, the namespace lock for kernel modules, the superblock locks of file systems, and many configuration-state locks in network drivers.

12.5.2 Why Shared/Exclusive Beats a Plain Mutex Here

Imagine a piece of driver state, “current debug verbosity level”, read at the start of every I/O call to decide whether to log certain events, and changed maybe once an hour by a `sysctl`. Under the Chapter 11 mutex design:

- Every I/O call acquires the mutex, reads the verbosity, releases the mutex.
- Every I/O call serializes on the mutex against every other I/O call’s verbosity check.
- The mutex sees enormous contention even though no one is contending for the underlying *state* (everyone is just reading).

Under an `sx` design:

- Every I/O call acquires the lock in shared mode (cheap on a multi-core system; fast-path reduces to a few atomic operations and no scheduler involvement).
- Multiple I/O calls can hold the lock concurrently. They do not block each other.
- The `sysctl` writer occasionally takes the lock in exclusive mode, briefly excluding readers. Readers retry as shared holders once the writer drops.

For a read-heavy workload, the difference is dramatic. The mutex’s serialization cost grows with the number of cores; the `sx`’s shared-mode cost stays constant.

The trade-off: `sx_xlock` is more expensive per acquisition than `mtx_lock`, because the lock is more elaborate internally. For state that is only ever read once and the readers are not contending, `mtx` is still better. The break-even point depends on the workload, but the rule of thumb is *use `sx` when readers are many and writers are rare; use `mtx` when the access pattern is symmetric or write-heavy*.

12.5.3 The `sx(9)` API

The `sx(9)` functions live in `/usr/src/sys/sys/sx.h` and `/usr/src/sys/kern/kern_sx.c`. The public API is small.

```
void sx_init(struct sx *sx, const char *description);
void sx_init_flags(struct sx *sx, const char *description, int opts);
void sx_destroy(struct sx *sx);
```

```

void  sx_xlock(struct sx *sx);
int   sx_xlock_sig(struct sx *sx);
void  sx_xunlock(struct sx *sx);
int   sx_try_xlock(struct sx *sx);

void  sx_slock(struct sx *sx);
int   sx_slock_sig(struct sx *sx);
void  sx_sunlock(struct sx *sx);
int   sx_try_slock(struct sx *sx);

int   sx_try_upgrade(struct sx *sx);
void  sx_downgrade(struct sx *sx);

void  sx_unlock(struct sx *sx); /* polymorphic: shared or exclusive */
void  sx_assert(struct sx *sx, int what);

int   sx_xlocked(struct sx *sx);
struct thread *sx_xholder(struct sx *sx);

```

The `_sig` variants are interruptible by signals; they return `EINTR` or `ERESTART` if signalled while waiting. The non-`_sig` variants block uninterruptibly. Drivers that hold an `sx` lock across a long operation should consider the `_sig` variants for the same reason they prefer `cv_wait_sig` over `cv_wait`: a Ctrl-C should be able to release the wait.

The flags accepted by `sx_init_flags` include:

- `SX_DUPOK`: allow the same thread to acquire the lock multiple times (mostly a `WITNESS` directive).
- `SX_NOWITNESS`: do not register the lock with `WITNESS` (use rarely; prefer to register and document any exceptions).
- `SX_RECURSE`: allow recursive acquisition by the same thread; the lock is released only when every acquire is matched.
- `SX_QUIET`, `SX_NOPROFILE`: turn off various debugging instrumentation.
- `SX_NEW`: declare that the memory is fresh (skip the prior-init check).

For most driver use cases, `sx_init(sx, "name")` with no flags is the right default.

`sx_assert(sx, what)` checks the lock state and panics under `INVARIANTS` if the assertion fails. The `what` argument is one of:

- `SA_LOCKED`: the lock is held in some mode by the calling thread.
- `SA_SLOCKED`: the lock is held in shared mode.
- `SA_XLOCKED`: the lock is held in exclusive mode by the calling thread.
- `SA_UNLOCKED`: the lock is not held by the calling thread.
- `SA_RECURSED`, `SA_NOTRECURSED`: matches the recursion state.

Use `sx_assert` liberally inside helpers that expect a particular lock state, in the same way Chapter 11 used `mtx_assert`.

12.5.4 A Quick Worked Example

Suppose we have a struct holding driver configuration:

```
struct myfirst_config {
    int     debug_level;
    int     soft_byte_limit;
    char     nickname[32];
};
```

Most reads of these fields happen on the data path (every `myfirst_read` and `myfirst_write` checks `debug_level`). Writes happen rarely, from `sysctl` handlers.

Add an `sx` lock to the softc:

```
struct sx          cfg_sx;
struct myfirst_config  cfg;
```

Initialize and destroy:

```
sx_init(&sc->cfg_sx, "myfirst cfg");
/* in detach: */
sx_destroy(&sc->cfg_sx);
```

Read on the data path:

```
static bool
myfirst_debug_enabled(struct myfirst_softc *sc, int level)
{
    bool enabled;

    sx_slock(&sc->cfg_sx);
    enabled = (sc->cfg.debug_level >= level);
    sx_sunlock(&sc->cfg_sx);
    return (enabled);
}
```

Write from a `sysctl` handler:

```
static int
myfirst_sysctl_debug_level(SYSCTL_HANDLER_ARGS)
{
    struct myfirst_softc *sc = arg1;
    int new, error;
```

```

    sx_slock(&sc->cfg_sx);
    new = sc->cfg.debug_level;
    sx_sunlock(&sc->cfg_sx);

    error = sysctl_handle_int(oidp, &new, 0, req);
    if (error || req->newptr == NULL)
        return (error);

    if (new < 0 || new > 3)
        return (EINVAL);

    sx_xlock(&sc->cfg_sx);
    sc->cfg.debug_level = new;
    sx_xunlock(&sc->cfg_sx);
    return (0);
}

```

Three things to notice in the writer.

First, we read the *current* value under the shared lock so the sysctl framework can fill in the value to display when no new value is being set. We could read it without the lock, but doing so would create a torn-read possibility for the (admittedly small) int. The shared lock is cheap and explicit.

Second, we drop the shared lock, validate the new value with `sysctl_handle_int`, validate the range, and then take the exclusive lock to commit. We cannot upgrade from shared to exclusive in this path because `sx_try_upgrade` may fail; doing it as drop-and-reacquire is simpler and correct.

Third, the validation happens before the exclusive lock, which means we hold the exclusive lock for the minimum time. The exclusive holder excludes all readers; we want it to release as quickly as possible.

12.5.5 Try-Upgrade and Downgrade

`sx_try_upgrade` is the optimistic version of “I have a shared lock, please give me an exclusive one without making me drop and reacquire”. It returns nonzero on success (the lock is now exclusive) and zero on failure (the lock is still shared; another thread held it shared simultaneously and the kernel could not safely promote).

The pattern:

```

sx_slock(&sc->cfg_sx);
/* do some reading */
if (need_to_modify) {
    if (sx_try_upgrade(&sc->cfg_sx)) {
        /* now exclusive; modify */
        sx_downgrade(&sc->cfg_sx);
        /* back to shared; continue reading */
    }
}

```

```
    } else {
        /* upgrade failed; drop and reacquire */
        sx_sunlock(&sc->cfg_sx);
        sx_xlock(&sc->cfg_sx);
        /* now exclusive but our prior view may be stale */
        /* re-validate and modify */
        sx_downgrade(&sc->cfg_sx);
    }
}
sx_sunlock(&sc->cfg_sx);
```

`sx_downgrade` always succeeds: an exclusive holder can always step down to shared without blocking, because no other writer can be present (we held the exclusive) and the existing readers all acquired their shared locks while we were holding exclusive (which they could not), so they cannot exist either.

For our `myfirst` configuration, we do not need upgrade/downgrade: the read and the write are separate paths and the `sysctl` handler is willing to drop and reacquire. Upgrade/downgrade is most useful in algorithms where the same thread reads, decides, and then conditionally modifies, all within one lock acquire-release cycle.

12.5.6 Comparing `sx(9)` to `rw(9)`

`rw(9)` is the spin-mode sibling of `sx(9)`. Both implement the shared/exclusive idea. They differ in how they wait for an unavailable lock.

`sx(9)` uses sleep queues. A thread that cannot acquire the lock immediately is put on a sleep queue and yields. Other threads run on the CPU. When the lock becomes available, the kernel wakes the highest-priority waiter, which then re-tries.

`rw(9)` uses turnstiles, the kernel's spin-based primitive that supports priority propagation. A thread that cannot acquire the lock immediately spins (briefly), then handed to the turnstile machinery for blocking with priority inheritance. The blocking is done in a way that does not give up the CPU as easily as `sx`.

The practical differences:

- `sx(9)` is sleepable in the strict sense: holding an `sx` lock allows you to call functions that may sleep (`uiomove`, `malloc(... M_WAITOK)`). Holding an `rw(9)` lock does *not*; the `rw(9)` lock is treated like a spin lock for sleep purposes.
- `sx(9)` supports `_sig` variants for signal-interruptible waits. `rw(9)` does not.
- `sx(9)` is generally appropriate for code in process context; `rw(9)` is more appropriate when the critical section is short and may run in interrupt context (although the strict choice for interrupt context is still `MTX_SPIN`).

For `myfirst`, all configuration access is from process context, the critical sections are short but include potentially-sleeping calls, and signal interruption is a useful feature. `sx(9)` is the right choice.

A driver with a configuration that is read inside an interrupt handler would have to use `rw(9)` instead, because `sx_slock` could sleep and sleeping in an interrupt is illegal. We will not encounter such a driver in this book until later parts.

12.5.7 The Sleepability Rule, Revisited

Chapter 11 introduced the rule “do not hold a non-sleepable lock across a sleeping operation”. With `sx` and `cv` on the table, the rule needs a small refinement.

The full rule is: *the lock you hold determines what operations are legal in the critical section.*

- Holding an `MTX_DEF` mutex: most operations are legal. Sleeping is permitted (with `mtx_sleep`, `cv_wait`). `uiomove`, `copyin`, `copyout`, and `malloc(M_WAITOK)` are legal in principle but should be avoided to keep critical sections short. The driver convention is to drop the mutex around any of these.
- Holding an `MTX_SPIN` mutex: very few operations are legal. No sleeping. No `uiomove`. No `malloc(M_WAITOK)`. The critical section must be tiny.
- Holding an `sx(9)` lock (shared or exclusive): like `MTX_DEF`. Sleeping is permitted. The same convention of “drop before sleeping if you can” applies, but the absolute prohibition on sleeping does not.
- Holding an `rw(9)` lock: like `MTX_SPIN`. No sleeping. No long-blocking calls.
- Holding a `cv(9)` (i.e., currently inside `cv_wait`): the underlying interlock mutex was atomically dropped by `cv_wait`; from the point of view of “what is held”, you hold nothing.

This refinement says: `sx` is sleepable, `rw` is not. That is the operative difference between them. Choose by which side of the line your critical section needs to be on.

12.5.8 Lock Order and `sx`

`WITNESS` tracks lock ordering across all classes: mutexes, `sx` locks, and `rw` locks. If your driver acquires an `sx` lock while holding a mutex, that establishes an order: mutex first, `sx` second. The reverse order from any path is a violation; `WITNESS` will warn.

For my first Stage 3 (this section), we will hold `sc->mtx` and `sc->cfg_sx` together in some paths. We must declare the order explicitly.

The natural order is *mtx before sx*. Reason: the data path holds `sc->mtx` for the `cbuf` operations; if it needs to read a configuration value during that critical section, it would acquire `sc->cfg_sx` while still holding `sc->mtx`. The reverse (`cfg_sx` first, `mtx` second) is also possible (a `sysctl` writer that wanted to update both configuration and trigger an event could acquire `cfg_sx`, then `mtx`), but a driver should pick one order and document it.

Section 5 elaborates this design and codifies the rule.

12.5.9 A Look at `kern_sx.c`

If you have a few minutes, open `/usr/src/sys/kern/kern_sx.c` and skim. The fast path of `sx_xlock` is one compare-and-swap on the lock word, exactly the same shape as the fast path of `mtx_lock`. The slow path (in `_sx_xlock_hard`) hands the thread to the sleep queue with priority propagation. The shared-lock path (`_sx_slock_int`) is similar but updates the shared-holder count rather than setting the owner.

What matters for the driver writer is that the fast path is cheap, the slow path is correct, and the API is the same shape as the mutex API you already know. If you can use `mtx_lock`, you can use `sx_xlock`; the new vocabulary is the shared-mode operations and the rules around them.

12.5.10 A Brief Tour of rw(9)

We have mentioned rw(9) several times as the spin-mode sibling of sx(9). Although our driver does not use it, you will encounter it in real FreeBSD source, so a short tour is worth the few minutes.

The API mirrors sx(9):

```
void  rw_init(struct rwlock *rw, const char *name);
void  rw_destroy(struct rwlock *rw);

void  rw_wlock(struct rwlock *rw);
void  rw_wunlock(struct rwlock *rw);
int   rw_try_wlock(struct rwlock *rw);

void  rw_rlock(struct rwlock *rw);
void  rw_runlock(struct rwlock *rw);
int   rw_try_rlock(struct rwlock *rw);

int   rw_try_upgrade(struct rwlock *rw);
void  rw_downgrade(struct rwlock *rw);

void  rw_assert(struct rwlock *rw, int what);
```

The differences from sx(9):

- The mode names are different: wlock (write/exclusive) and rlock (read/shared) instead of xlock and slock. Same idea, different vocabulary.
- There are no _sig variants. rw(9) cannot be interrupted by signals because it is implemented over turnstiles, not sleep queues.
- A thread holding any rw(9) lock cannot sleep. No cv_wait, no mtx_sleep, no uiomove, no malloc(M_WAITOK).
- rw(9) supports priority propagation. A thread waiting for an exclusive lock that is held by a lower-priority thread will boost the holder's priority. This is the main reason rw(9) exists rather than just being a thin wrapper around sx(9).

The rw_assert flags are RA_LOCKED, RA_RLOCKED, RA_WLOCKED, plus the same recursion variants sx_assert has.

Where you will see rw(9) in the FreeBSD tree:

- The networking stack uses rw(9) for several read-mostly tables (routing tables, the address resolution table). Read access happens in the receive path, which runs in network interrupt context where sleeping is forbidden.
- The VFS layer uses it for some namespace caches.
- Various subsystems with hot read paths and rare configuration updates.

For our myfirst driver, every cfg access happens in process context, every cfg writer is willing to drop the lock around sysctl_handle_* (which sleeps), and we benefit from signal interruptibility. sx(9) is the right

choice. If you ever need to access the same configuration from an interrupt handler (Chapter 14 will discuss this), the answer is to switch to `rw(9)` and accept the constraint that the `cfg` writer must do all its work without sleeping.

12.5.11 A Worked Example with `rw(9)`

To make the alternative concrete, here is what the `cfg` path would look like with `rw(9)`. The code is structurally identical except for the API and the absence of signal interruptibility:

```
/* In the softc: */
struct rwlock      cfg_rw;
struct myfirst_config  cfg;

/* In attach: */
rw_init(&sc->cfg_rw, "myfirst cfg");

/* In detach: */
rw_destroy(&sc->cfg_rw);

/* Read path: */
static int
myfirst_get_debug_level_rw(struct myfirst_softc *sc)
{
    int level;

    rw_rlock(&sc->cfg_rw);
    level = sc->cfg.debug_level;
    rw_runlock(&sc->cfg_rw);
    return (level);
}

/* Write path (sysctl handler): */
static int
myfirst_sysctl_debug_level_rw(SYSCTL_HANDLER_ARGS)
{
    struct myfirst_softc *sc = arg1;
    int new, error;

    rw_rlock(&sc->cfg_rw);
    new = sc->cfg.debug_level;
    rw_runlock(&sc->cfg_rw);

    error = sysctl_handle_int(oidp, &new, 0, req);
    if (error || req->newptr == NULL)
        return (error);
}
```



```
    if (new < 0 || new > 3)
        return (EINVAL);

    rw_wlock(&sc->cfg_rw);
    sc->cfg.debug_level = new;
    rw_wunlock(&sc->cfg_rw);
    return (0);
}
```

Two things to notice. First, `sysctl_handle_int` is *outside* the lock. Calling it inside an `rw(9)` critical section would be illegal because `sysctl_handle_int` may sleep. This is the same discipline we use for the `sx(9)` version, but with `rw(9)` it is mandatory rather than merely advisable. Second, the read paths look identical to the `sx(9)` version; only the function names changed. That is the point of the symmetric API: the mental model carries over.

If our driver someday needs to support an interrupt-context reader of the configuration (perhaps a hardware interrupt handler that wants to know the current debug level), this is the change we would make. For now, `sx(9)` is correct and we stay with it.

12.5.12 Wrapping Up Section 4

`sx(9)` gives us a way to express “many readers, occasional writer” without serializing every reader. It is sleepable, signal-aware, and follows the same lock-order discipline as the mutex. `rw(9)` is its non-sleepable sibling, useful when the critical section may run in contexts where sleeping is illegal; the worked example above shows the small differences. We use `sx(9)` for `myfirst` because process context and signal interruptibility are both desirable.

Section 5 brings the new primitives together. We add a small configuration subsystem to `myfirst`, decide the lock order between the data path and the configuration path, and verify the design against `WITNESS`.

12.6 Section 5: Implementing a Safe Multi-reader, Single-writer Scenario

The previous three sections introduced primitives in isolation. This section combines them into a coherent driver design. We add a small configuration subsystem to `myfirst`, give it its own `sx` lock, work out the lock order with respect to the existing data-path mutex, and verify the resulting design against a `WITNESS` kernel.

The configuration subsystem is small on purpose. The point is not to demonstrate a complex feature; it is to demonstrate the lock-order discipline that any driver with more than one lock class must follow.

12.6.1 The Configuration Subsystem

We add three configurable parameters:

- `debug_level`: an integer 0 to 3. Higher values produce more verbose `dmesg` output from the driver's data path.
- `soft_byte_limit`: an integer. If non-zero, the driver refuses to accept writes that would push the `cbuf` above this many bytes (it returns `EAGAIN` early). This is a poor man's flow-control knob.
- `nickname`: a short string the driver prints in its log lines. Useful for distinguishing multiple driver instances in `dmesg`.

The struct that holds them:

```
struct myfirst_config {
    int     debug_level;
    int     soft_byte_limit;
    char     nickname[32];
};
```

Add it to the `softc`, alongside its `sx` lock:

```
struct myfirst_softc {
    /* ... existing fields ... */
    struct sx          cfg_sx;
    struct myfirst_config  cfg;
    /* ... rest ... */
};
```

Initialize and destroy:

```
/* In attach: */
sx_init(&sc->cfg_sx, "myfirst cfg");
sc->cfg.debug_level = 0;
sc->cfg.soft_byte_limit = 0;
strncpy(sc->cfg.nickname, "myfirst", sizeof(sc->cfg.nickname));

/* In detach: */
sx_destroy(&sc->cfg_sx);
```

The initial values are set before the `cdev` is created, so no other thread can observe a half-initialized configuration.

12.6.2 The Lock Order Decision

The driver now has two lock classes that can be held simultaneously: `sc->mtx` (the `cbuf` and per-`softc` state) and `sc->cfg_sx` (the configuration). We must decide which is acquired first when both are needed.

The natural questions to ask:

1. Which path holds each lock most often? The data path holds `sc->mtx` constantly (every `myfirst_read` and `myfirst_write` enters and leaves it). The data path also wants to read `sc->cfg.debug_level` to decide whether to log; that is a `sx_slock(&sc->cfg_sx)`. So the data path already wants both, in the order *mtx first, sx second*.
2. Which path holds the `cfg` lock and might want the data lock? A `sysctl` handler that updates the configuration takes `sx_xlock(&sc->cfg_sx)`. Does it ever need `sc->mtx`? In principle, yes: a `sysctl` handler that resets the byte counters would take both. The cleanest design is to *not* take the data mutex from inside the `sx` critical section; the `sysctl` writer stages its work, releases the `sx` lock, and then takes the data mutex if needed. That keeps the order monotonic.

The decision: **`sc->mtx` is acquired before `sc->cfg_sx` whenever both are held simultaneously.**

The reverse order is forbidden. WITNESS will catch any violation.

We document the decision in `LOCKING.md`:

Lock Order

`sc->mtx -> sc->cfg_sx`

A thread holding `sc->mtx` may acquire `sc->cfg_sx` (in either shared or exclusive mode). A thread holding `sc->cfg_sx` may NOT acquire `sc->mtx`.

Rationale: the data path always holds `sc->mtx` and may need to read configuration during its critical section. The configuration path (`sysctl` writers) does not need to update data-path state while holding `sc->cfg_sx`; if it needs to, it releases `sc->cfg_sx` first and then acquires `sc->mtx` separately.

12.6.3 Reading the Configuration on the Data Path

The most frequent access to the configuration is the data path's check of `debug_level` to decide whether to emit a log message. We wrap it in a small helper:

```
static int
myfirst_get_debug_level(struct myfirst_softc *sc)
{
    int level;

    sx_slock(&sc->cfg_sx);
    level = sc->cfg.debug_level;
    sx_sunlock(&sc->cfg_sx);
    return (level);
}
```

Note that this helper takes only `sc->cfg_sx`, not `sc->mtx`. That is intentional: the helper does not need the data mutex to read the configuration. If it is called from a context that already holds `sc->mtx`, the lock order is satisfied (mtx first, sx second). If it is called from a context that holds nothing, that is fine too.

A debug-aware log macro:

```
#define MYFIRST_DBG(sc, level, fmt, ...) do {           \
    if (myfirst_get_debug_level(sc) >= (level))       \
        device_printf((sc)->dev, fmt, ##__VA_ARGS__); \
} while (0)
```

Use it on the data path:

```
MYFIRST_DBG(sc, 2, "read got %zu bytes\n", got);
```

The shared-lock acquisition is the cost of every check. On a multi-core machine this is a few atomic operations; the readers do not contend with each other. On a single-core machine the cost is essentially zero (no other thread can be in the middle of a write).

12.6.4 Reading the Soft Byte Limit

The same pattern for the soft byte limit, used by `myfirst_write` to decide whether to refuse:

```
static int
myfirst_get_soft_byte_limit(struct myfirst_softc *sc)
{
    int limit;

    sx_slock(&sc->cfg_sx);
    limit = sc->cfg.soft_byte_limit;
    sx_sunlock(&sc->cfg_sx);
    return (limit);
}
```

Inside `myfirst_write`, before the actual write happens (note that `want` has not been computed yet at this point in the loop), the limit check uses `sizeof(bounce)` as a worst-case proxy: any single iteration writes at most one bounce buffer's worth of bytes, so refusing when `cbuf_used + sizeof(bounce)` would exceed the limit is a conservative early-out:

```
int limit = myfirst_get_soft_byte_limit(sc);

MYFIRST_LOCK(sc);
if (limit > 0 && cbuf_used(&sc->cb) + sizeof(bounce) > (size_t)limit) {
    MYFIRST_UNLOCK(sc);
```

```

        return (uio->uio_resid != nbefore ? 0 : EAGAIN);
    }
    /* fall through to wait_room and the rest of the iteration */

```

Two acquisitions back to back: the `cfg_sx` for the limit, the `mtx` for the `cbuf` check. Notice that we acquire the `sx` *first* and drop it before taking the mutex. We could in principle hold both (`cfg_sx` then `mtx`) but the order would be wrong; the rule says `mtx` first, `sx` second. So we acquire each independently. The slight cost of two acquisitions is the price of correctness.

A subtle point: between the `cfg_sx` release and the `mtx` acquisition, the limit could change. That is acceptable; the limit is a soft hint, not a hard guarantee. If a `sysctl` writer raises the limit between our two acquisitions and we still refuse the write, the user will retry and succeed on the second attempt. If the limit is lowered and we proceed with a write that the new limit would have refused, no harm is done because the `cbuf` has its own hard size limit.

The choice of `sizeof(bounce)` rather than the actual `want` reflects another subtle point: at this stage in the loop, the driver has not yet computed `want` (that requires knowing how much room the `cbuf` currently has, which requires holding the mutex first). Using `sizeof(bounce)` as the worst-case bound lets the check happen before the room calculation. The companion source file follows exactly this pattern.

12.6.5 Updating the Configuration: a Sysctl Writer

The writer side, exposed as a `sysctl` that can read and write `debug_level`:

```

static int
myfirst_sysctl_debug_level(SYSCTL_HANDLER_ARGS)
{
    struct myfirst_softc *sc = arg1;
    int new, error;

    sx_slock(&sc->cfg_sx);
    new = sc->cfg.debug_level;
    sx_unlock(&sc->cfg_sx);

    error = sysctl_handle_int(oidp, &new, 0, req);
    if (error || req->newptr == NULL)
        return (error);

    if (new < 0 || new > 3)
        return (EINVAL);

    sx_xlock(&sc->cfg_sx);
    sc->cfg.debug_level = new;
    sx_xunlock(&sc->cfg_sx);
}

```

```

        return (0);
    }

```

Walk through the lock acquisitions:

1. `sx_slock` to read the current value (so the `sysctl` framework can return it on a read-only query).
2. `sx_sunlock` before calling `sysctl_handle_int`, because that function may copy data to and from user space (which can sleep) and we do not want to hold the `sx` lock across that.
3. After validation, `sx_xlock` to commit the new value.
4. `sx_xunlock` to release.

We never hold `sc->mtx` in this path. The lock order rule is trivially satisfied: this path never has both locks held at once.

Register the `sysctl` in `attach`:

```

SYSCTL_ADD_PROC(&sc->sysctl_ctx, SYSCTL_CHILDREN(sc->sysctl_tree),
    OID_AUTO, "debug_level",
    CTLTYPE_INT | CTLFLAG_RW | CTLFLAG_MPSAFE,
    sc, 0, myfirst_sysctl_debug_level, "I",
    "Debug verbosity level (0-3)");

```

The `CTLFLAG_MPSAFE` flag tells the `sysctl` framework that our handler is safe to call without acquiring the giant lock; we are. This is the modern default for new `sysctl` handlers.

12.6.6 Updating the Soft Byte Limit

The same shape for the byte limit:

```

static int
myfirst_sysctl_soft_byte_limit(SYSCTL_HANDLER_ARGS)
{
    struct myfirst_softc *sc = arg1;
    int new, error;

    sx_slock(&sc->cfg_sx);
    new = sc->cfg.soft_byte_limit;
    sx_sunlock(&sc->cfg_sx);

    error = sysctl_handle_int(oidp, &new, 0, req);
    if (error || req->newptr == NULL)
        return (error);

    if (new < 0)
        return (EINVAL);
}

```

```
    sx_xlock(&sc->cfg_sx);
    sc->cfg.soft_byte_limit = new;
    sx_xunlock(&sc->cfg_sx);
    return (0);
}
```

And for the nickname (a string, so the sysctl handler is slightly different):

```
static int
myfirst_sysctl_nickname(SYSCTL_HANDLER_ARGS)
{
    struct myfirst_softc *sc = arg1;
    char buf[sizeof(sc->cfg.nickname)];
    int error;

    sx_slock(&sc->cfg_sx);
    strncpy(buf, sc->cfg.nickname, sizeof(buf));
    sx_sunlock(&sc->cfg_sx);

    error = sysctl_handle_string(oidp, buf, sizeof(buf), req);
    if (error || req->newptr == NULL)
        return (error);

    sx_xlock(&sc->cfg_sx);
    strncpy(sc->cfg.nickname, buf, sizeof(sc->cfg.nickname));
    sx_xunlock(&sc->cfg_sx);
    return (0);
}
```

The structure is identical: shared-lock to read, drop, validate via the sysctl framework, exclusive-lock to commit. The string version uses `strncpy` for safety.

12.6.7 A Single Operation Holding Both Locks

Sometimes a path legitimately needs both locks. As an example, suppose we add a sysctl that resets the cbuf and clears all the byte counters at once. That sysctl needs:

1. The exclusive cfg lock if it is going to also reset some configuration (say, reset the debug level).
2. The data mutex to manipulate the cbuf.

Following our lock order, we acquire `sc->mtx` first, then `sc->cfg_sx`:

```
static int
myfirst_sysctl_reset(SYSCTL_HANDLER_ARGS)
{
    struct myfirst_softc *sc = arg1;
    int reset = 0;
    int error;

    error = sysctl_handle_int(oidp, &reset, 0, req);
    if (error || req->newptr == NULL || reset != 1)
        return (error);

    MYFIRST_LOCK(sc);
    sx_xlock(&sc->cfg_sx);

    cbuf_reset(&sc->cb);
    sc->cfg.debug_level = 0;
    counter_u64_zero(sc->bytes_read);
    counter_u64_zero(sc->bytes_written);

    sx_xunlock(&sc->cfg_sx);
    MYFIRST_UNLOCK(sc);

    cv_broadcast(&sc->room_cv); /* room is now available */
    return (0);
}
```

The order of acquisition is mtx then sx. The order of release is the reverse: sx first, mtx second. (Releases must reverse the acquisition order to maintain the lock-order invariant for any thread that observes a lock state in the middle.)

The cv broadcast happens after both locks are released. Waking sleepers does not require holding either lock.

cbuf_reset is a small helper we add to the cbuf module:

```
void
cbuf_reset(struct cbuf *cb)
{
    cb->cb_head = 0;
    cb->cb_used = 0;
}
```

It zeroes the indices but does not touch the backing memory; the contents become irrelevant the moment cb_used is zero.

12.6.8 Verifying Against WITNESS

Build the new driver and load it on a WITNESS kernel. Run the Chapter 11 stress kit plus a new tester that hammers the sysctls while I/O is happening:

```
/* config_writer.c: continuously update config sysctls. */
#include <err.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/sysctl.h>
#include <unistd.h>

int
main(int argc, char **argv)
{
    int seconds = (argc > 1) ? atoi(argv[1]) : 30;
    time_t end = time(NULL) + seconds;
    int v = 0;

    while (time(NULL) < end) {
        v = (v + 1) % 4;
        if (sysctlbyname("dev.myfirst.0.debug_level",
            NULL, NULL, &v, sizeof(v)) != 0)
            warn("sysctl debug_level");

        int limit = (v == 0) ? 0 : 4096;
        if (sysctlbyname("dev.myfirst.0.soft_byte_limit",
            NULL, NULL, &limit, sizeof(limit)) != 0)
            warn("sysctl soft_byte_limit");

        usleep(10000); /* 10 ms */
    }
    return (0);
}
```

Run `mp_stress` in one terminal, `mt_reader` in a second, `config_writer` in a third, all simultaneously. Watch `dmesg` for warnings.

Three things should happen:

1. All tests pass with consistent byte counts.
2. The debug level changes visibly in `dmesg` (when the level is high, the data path emits log messages; when low, it is quiet).
3. WITNESS is silent. No lock-order reversals reported.

If WITNESS does report a reversal, it means the lock order was violated somewhere. Re-read the affected code path against the rule (mtx first, sx second) and fix the violation.

12.6.9 What Goes Wrong if You Get the Order Backwards

To make the rule concrete, deliberately violate it. Take an existing path that holds `sc->mtx` and rewrite it to acquire the locks in the wrong order. For example, in `myfirst_read`:

```
/* WRONG: this is the bug we want WITNESS to catch. */
sx_slock(&sc->cfg_sx); /* sx first */
MYFIRST_LOCK(sc);      /* mtx second; reverses the global order */
/* ... */
MYFIRST_UNLOCK(sc);
sx_sunlock(&sc->cfg_sx);
```

Build, load on a WITNESS kernel, run the test kit. WITNESS should fire on the first run that exercises both this path and any other path that does `mtx-then-sx`:

```
lock order reversal:
 1st 0xfffffe000a1b2c30 myfirst cfg (myfirst cfg, sx) @ ...:<line>
 2nd 0xfffffe000a1b2c50 myfirst0 (myfirst, sleep mutex) @ ...:<line>
lock order myfirst cfg -> myfirst0 attempted at ...
where myfirst0 -> myfirst cfg is established at ...
```

The warning names the two locks, their addresses, the source location of each acquisition, and the previously-established order. The fix is to put the locks back in the canonical order; revert the change and the warning disappears.

This is WITNESS doing its job. A driver with more than one lock class without WITNESS is a driver waiting for a deadlock that nobody can reproduce.

12.6.10 A Slightly Larger Pattern: Snapshot-and-Apply

A common pattern when both locks are needed and the operation has work to do under each is *snapshot-and-apply*. In its simplest form, before the actual transfer size is known:

```
/* Phase 1: snapshot the configuration under the sx (in shared mode). */
sx_slock(&sc->cfg_sx);
int dbg = sc->cfg.debug_level;
int limit = sc->cfg.soft_byte_limit;
sx_sunlock(&sc->cfg_sx);

/* Phase 2: do the work under the mtx, using the snapshot. */
MYFIRST_LOCK(sc);
if (limit > 0 && cbuf_used(&sc->cb) + sizeof(bounce) > (size_t)limit) {
    MYFIRST_UNLOCK(sc);
    return (EAGAIN);
}
```

```

}
/* ... cbuf operations ... */
size_t actual = /* determined inside the critical section */;
MYFIRST_UNLOCK(sc);

if (dbg >= 2)
    device_printf(sc->dev, "wrote %zu bytes\n", actual);

```

The snapshot-and-apply pattern keeps each lock held for the minimum time, avoids holding both locks at once, and produces a clear two-phase shape that is easy to reason about. The cost is that the snapshot may be slightly stale when the apply executes; in practice, the staleness is microseconds and is acceptable for almost any configuration value.

If staleness is unacceptable for some specific value (say, a critical security flag), then the holder must take the lock(s) atomically and follow the global order. The snapshot pattern is a default, not a law.

12.6.11 A Walk-Through of `myfirst_sysctl_reset`

The reset sysctl is the only path in the chapter that legitimately holds both locks at once. It is worth tracing through carefully, because the pattern (mtx then sx, both released in reverse order, broadcasts after both are released) is the one to imitate any time both locks must be held.

When a user runs `sysctl -w dev.myfirst.0.reset=1`, the kernel calls `myfirst_sysctl_reset` with `req->newptr` non-NULL. The handler:

1. Reads the new value via `sysctl_handle_int`. If the value is not 1, returns without doing anything (treats only 1 as a confirmed reset request).
2. Acquires `MYFIRST_LOCK(sc)`. The data mutex is now held.
3. Acquires `sx_xlock(&sc->cfg_sx)`. The `cfg sx` is now held in exclusive mode. Both locks held; lock order satisfied (mtx first, sx second).
4. Calls `cbuf_reset(&sc->cb)`. The buffer is now empty (`cb_used = 0`, `cb_head = 0`).
5. Sets `sc->cfg.debug_level = 0`. The config is now in its initial state.
6. Calls `counter_u64_zero` on each per-CPU counter. The byte counters are now zero.
7. Calls `sx_xunlock(&sc->cfg_sx)`. The `cfg sx` is released. Only the `mtx` is held now.
8. Calls `MYFIRST_UNLOCK(sc)`. The `mtx` is released. No locks held.
9. Calls `cv_broadcast(&sc->room_cv)`. Any writers blocked on `full` are woken; they will recheck `cb_free`, find it equal to `cb_size`, and proceed.
10. Returns 0.

Three observations about this sequence.

The lock acquisitions follow the global order; `WITNESS` is happy. The releases follow the reverse order, which preserves the invariant that any thread observing the sequence sees a consistent state.

The broadcasts happen *after* both locks are released. Holding either lock while broadcasting would needlessly block the awakened threads when they try to acquire what they need; the broadcast is fire-and-forget, and the kernel handles the rest.

The reset of the data path (`cbuf_reset`) and the reset of the configuration (`debug_level = 0`) are atomic with respect to each other. A thread that observes any field after the reset sees the post-reset value of every field; no thread can observe a half-reset state.

If we wanted to add more reset operations (clear a state machine, reset a hardware register), each one fits into this template. The lock holds expand to cover the new fields; the broadcasts at the end notify the appropriate cvs.

12.6.12 Wrapping Up Section 5

The driver now has two lock classes: a mutex for the data path and an sx lock for the configuration. The order is documented (mtx first, sx second), the order is enforced by `WITNESS` at runtime, and the patterns we use (snapshot-and-apply for the common case, single-acquisition for paths that need only one lock, and the carefully-ordered acquisition-release for paths that genuinely need both) keep the design auditable. The new sysctls let user space adjust the driver's behaviour at runtime without disrupting the data path.

The bigger lesson of Section 5 is that adding a second lock class is a real design decision, not a transparent optimization. The cost is the discipline required to maintain lock order, the documentation effort, and the audit work to verify both at runtime. The benefit is that the data path is no longer the only place state lives; the configuration has its own protection, its own pace, and its own sysctl interface. As drivers grow, they typically end up with three or four lock classes, each with its own purpose. The pattern from this section scales.

Section 6 turns to the question that always comes next when you start mixing lock classes: how do you debug a synchronization bug when it appears? The kernel gives you several tools for the job; this is the chapter that introduces them.

12.7 Section 6: Debugging Synchronization Issues

Chapter 11 introduced the basic kernel debugging hooks: `INVARIANTS`, `WITNESS`, `KASSERT`, and `mtx_assert`. We used them to verify that the mutex was held at the right times and that lock-order rules were respected. With the broader synchronization toolkit Chapter 12 has put in your hands, the debugging tools also expand. This section walks through the patterns and the tools you will use most often: reading a `WITNESS` warning carefully, inspecting a hung system with the in-kernel debugger, and recognising the most common failure modes of condition variables and shared/exclusive locks.

12.7.1 A Catalogue of Synchronization Bugs

Six failure shapes cover most of what you will encounter in practice. Recognising the shape is half the diagnosis.

Lost wakeup. A thread enters `cv_wait_sig` (or `mtx_sleep`); the condition it is waiting for becomes true but no `cv_signal` (or `wakeup`) is ever called. The thread sleeps forever. Causes: forgot the `cv_signal` after the state change; signalled the wrong cv; signalled before the state was actually changed; the state change happens in a path that does not include a signal at all.

Spurious wakeup mishandled. A thread is awakened by a signal or other transient event but the condition it was waiting for is still false. If the surrounding code does not loop and re-check, the thread proceeds with the

assumption that the condition is true and operates on stale state. Cure: always wrap `cv_wait` calls in `while (!condition) cv_wait(...)`, never `if (!condition) cv_wait(...)`.

Lock-order reversal. Two locks acquired in opposite orders by two paths. Either deadlocks under contention or WITNESS catches it. Cure: define a global order in `LOCKING.md` and follow it everywhere.

Premature destroy. A `cv` or `sx` is destroyed while a thread is still waiting on it. Symptoms are unpredictable: panics, use-after-free, stale-pointer crashes. Cure: ensure every waiter has been woken (and has actually returned from the wait primitive) before calling `cv_destroy` or `sx_destroy`. The detach path must be careful here.

Sleeping with a non-sleepable lock held. Holding an `MTX_SPIN` mutex or an `rw(9)` lock and then calling `cv_wait`, `mtx_sleep`, `uiomove`, or `malloc(M_WAITOK)`. WITNESS catches it; on a non-WITNESS kernel the system either deadlocks or panics. Cure: drop the spin/rw lock before sleeping, or hold an `MTX_DEF` mutex / `sx(9)` lock instead, both of which permit sleeping.

Race between detach and an active operation. A descriptor is open, a thread is in `cv_wait_sig`, and the device is told to detach. The detach path must wake the sleeper, must wait until the sleeper has returned, and must keep the `cv` (and the mutex) alive until that has happened. Cure: the standard detach pattern is set the “going away” flag, broadcast all `cvs`, wait for `active_fhs` to drop to zero, then destroy primitives.

A driver that survives long enough to hit production has handled each of these at least once. The labs in this chapter let you provoke and resolve several deliberately so the patterns become familiar.

12.7.2 Reading a WITNESS Warning Carefully

A WITNESS warning has three useful parts: the warning text, the source locations of each lock acquisition, and the previously-established order. Pull them apart.

A typical warning for a lock-order reversal:

```
lock order reversal:
 1st 0xfffffe000a1b2c30 myfirst cfg (myfirst cfg, sx) @ /var/.../myfirst.c:120
 2nd 0xfffffe000a1b2c50 myfirst0 (myfirst, sleep mutex) @ /var/.../myfirst.c:240
lock order myfirst cfg -> myfirst0 attempted at /var/.../myfirst.c:241
where myfirst0 -> myfirst cfg is established at /var/.../myfirst.c:280
```

Reading from top to bottom:

- **lock order reversal::** the warning class. Other classes include acquiring duplicate lock of same type, sleeping thread (pid N) owns a non-sleepable lock, WITNESS exceeded the recursion limit.
- **1st 0x... myfirst cfg:** the first lock acquired in the offending path. The address (0x...), name (myfirst cfg), type (sx), and source location (myfirst.c:120) tell you exactly which lock and where.
- **2nd 0x... myfirst0:** the second lock acquired. Same set of fields.
- **lock order myfirst cfg -> myfirst0 attempted at ...:** the order this code is trying to use.
- **where myfirst0 -> myfirst cfg is established at ...:** the order WITNESS previously observed and recorded as canonical, with the source location of the canonical example.

The fix is one of two things. Either the new path is wrong and should follow the canonical order (most common), or the canonical order itself is wrong and needs to change. Picking which is a judgement call; usually the answer is to fix the new path to match the canonical, because the canonical was probably right.

Sometimes the warning is a false alarm: two distinct objects of the same lock type, where the order between them does not matter because they are independent. WITNESS does not always know this; the LOR_DUPOK flag at lock initialization tells it to skip the check. We do not need that for `myfirst`, but real drivers with per-instance locks sometimes do.

12.7.3 Using DDB to Inspect a Hung System

If the test hangs and `dmesg` is silent, the culprit is usually a missed wakeup or a deadlock. The in-kernel debugger (DDB) lets you inspect the state of every thread and every lock at the moment of the hang.

To enter DDB on a hung system, press the Break key on the console (or send the `~b` escape if you are on a serial console with `cu`). DDB prompts you with `db>`.

The most useful commands for synchronization debugging:

- `show locks`: list the locks held by the current thread (the one DDB came in on, which is usually the kernel idle thread; rarely useful by itself).
- `show all locks` (alias `show alllocks`): list every lock held by every thread on the system. This is the command you want most of the time.
- `show witness`: dump the entire WITNESS lock-order graph. Verbose but authoritative.
- `show sleepchain <thread>`: trace the chain of locks and waits a specific thread is involved in. Useful when you suspect a deadlock loop.
- `show lockchain <lock>`: trace from a lock to the thread that holds it and any other locks that thread holds.
- `ps`: list all processes and threads with their state.
- `bt <thread>`: backtrace of a specific thread.
- `continue`: leave DDB and resume the system. Use only if you have not made any changes.
- `panic`: force a panic so the system reboots cleanly. Use if `continue` is not safe.

Workflow for a hung test:

1. Enter DDB via the console break.
2. `show all locks`. Note any threads holding locks.
3. `ps` to find the test threads (look for `myfrd`, `myfwr`, or `cv_w` in the wait channel column).
4. For each interesting thread, `bt <pid> <tid>` to get a backtrace.
5. If a deadlock is suspected, `show sleepchain` for each waiting thread.
6. Once you have enough information, `panic` to reboot.

The DDB transcript becomes your debugging log. Save it (DDB can output to the console and the kernel message buffer; on a debug kernel with `EARLY_AP_STARTUP`, you can redirect output to serial).

12.7.4 Recognising Lost Wakeups

A lost wakeup is the most common `cv` bug. The symptom is a hung waiter that has been there forever; the thread is in `cv_wait_sig` and nothing wakes it.

Detection in DDB:

```
db> ps
... (find the hung thread, in state "*myfirst data" for example)
db> bt 1234 1235
... (backtrace shows the thread inside cv_wait_sig)
db> show locks
... (the thread holds no locks; it is sleeping)
```

The cv in question is identified by the wait channel name. If the channel name is `myfirst data`, the cv is `sc->data_cv`. Now ask: who should have called `cv_signal(&sc->data_cv)`? Search the source:

```
$ grep -n 'cv_signal(&sc->data_cv|cv_broadcast(&sc->data_cv' myfirst.c
```

For each call site, work out whether it was supposed to fire and whether it actually fired. Common culprits:

- The signal is inside an `if` that was never true.
- The signal is after a return that bypassed it.
- The signal targets the wrong cv (`cv_signal(&sc->room_cv)` instead of `data_cv`).
- The state change does not actually change the condition (you incremented `cb_used` but the consumer was checking `cb_free`, which is the same logical state but `cb_free == cb_size - cb_used`; that one is fine, but a similar miscalculation can hide).

Fix the source, rebuild, retest. The symptom should disappear.

12.7.5 Recognising Spurious Wakeups

A spurious wakeup is a wakeup that arrives while the condition is still false. Causes include signals (`cv_wait_sig` returns due to a signal even when the condition has not changed) and timeouts (`cv_timedwait_sig` returns due to the timer). Both are normal; the driver must handle them.

Detection: the bug is *not* the wakeup itself but the failure to handle it. The shape:

```
/* WRONG: */
if (cbuf_used(&sc->cb) == 0)
    cv_wait_sig(&sc->data_cv, &sc->mtx);
/* now reading cbuf assuming there is data, but a spurious wakeup
   could have brought us here while the buffer is still empty */
got = cbuf_read(&sc->cb, bounce, take);
```

`cbuf_read` would return zero in that case, which propagates to user space as a zero-byte read, which `cat` and other utilities interpret as EOF. The user sees a silent end-of-file that is not really an end of file.

Cure: always loop:

```

/* CORRECT: */
while (cbuf_used(&sc->cb) == 0) {
    int error = cv_wait_sig(&sc->data_cv, &sc->mtx);
    if (error != 0)
        return (error);
    if (!sc->is_attached)
        return (ENXIO);
}
got = cbuf_read(&sc->cb, bounce, take);

```

The `myfirst_wait_data` helper from Section 2 already follows this pattern. The general rule is: *never use `if` around a `cv_wait`; always use `while`.*

12.7.6 Recognising Lock Order Reversal

We saw a `WITNESS` warning earlier. The alternative, on a non-`WITNESS` kernel, is a deadlock. Two threads each hold one lock and want the other; neither can proceed.

Detection in DDB:

```

db> show all locks
Process 1234 (test1) thread 0xfffffe...
shared sx myfirst cfg (myfirst cfg) ... locked @ ...:120
shared sx myfirst cfg (myfirst cfg) r = 1 ... locked @ ...:120

Process 5678 (test2) thread 0xfffffe...
exclusive sleep mutex myfirst0 (myfirst) r = 0 ... locked @ ...:240

```

Then show `sleepchain` for each waiting thread:

```

db> show sleepchain 1234
Thread 1234 (pid X) blocked on lock myfirst0 owned by thread 5678
db> show sleepchain 5678
Thread 5678 (pid Y) blocked on lock myfirst cfg owned by thread 1234

```

That cycle is the deadlock. Each thread is blocked on a lock held by the other. The fix is to revisit the lock order; one of the two paths is acquiring in the wrong order. The fix is the same as for the `WITNESS` warning: find the offending acquisition and reorder it.

12.7.7 Sleeping With a Non-Sleepable Lock

If your driver uses `MTX_SPIN` mutexes or `rw(9)` locks and somewhere you call `cv_wait`, `mtx_sleep`, `uiomove`, or `malloc(M_WAITOK)` while holding one, `WITNESS` will fire:


```
sleeping thread (pid 1234, tid 5678) owns a non-sleepable lock:  
exclusive rw lock myfirst rw (myfirst rw) r = 0 ... locked @ ...:100
```

The warning names the lock and the location of the acquisition. The fix is to drop the lock before the sleeping operation. `myfirst` does not use `MTX_SPIN` or `rw(9)`, so we will not encounter this directly; if you reuse the patterns in another driver, watch for it.

12.7.8 Premature Destroy

A `cv` or `sx` that is destroyed while a thread is still waiting causes use-after-free style crashes. The symptom is usually a panic with a backtrace inside `cv_wait` or `sx_xlock` after `cv_destroy` or `sx_destroy` has already run.

The Chapter 11 detach pattern (refuse detach while `active_fhs > 0`) prevents this for most situations. The Chapter 12 driver extends the pattern with `cv_broadcast` calls before destroying:

```
MYFIRST_LOCK(sc);  
sc->is_attached = 0;  
cv_broadcast(&sc->data_cv);  
cv_broadcast(&sc->room_cv);  
MYFIRST_UNLOCK(sc);  
  
/* now wait for any thread that was sleeping to return.  
   The cv_broadcast above wakes them; they see is_attached  
   is false and return ENXIO. They release the mutex on  
   their way out. */  
  
/* Once we know no one is in the I/O paths, destroy the  
   primitives. By construction, active_fhs == 0 when we get  
   here, so no thread can re-enter. */  
cv_destroy(&sc->data_cv);  
cv_destroy(&sc->room_cv);  
sx_destroy(&sc->cfg_sx);  
mtx_destroy(&sc->mtx);
```

The order is important. The mutex is the interlock for the cvs (a thread inside `cv_wait_sig` was sleeping with `sc->mtx` released to the kernel; on wake-up, `cv_wait_sig` reacquires `sc->mtx` before it returns). If we destroy `sc->mtx` first and then `cv_destroy`, a not-yet-fully-woken thread can be stuck inside the kernel trying to reacquire a mutex whose memory we have just torn down. Destroying the cvs first guarantees that no thread is still inside `cv_wait_sig` by the time the mutex goes away. The same reasoning applies to the `sx`: a thread blocked inside `sx_xlock` does not hold the mutex, but its post-wake reacquisition path may trip over the order if the `sx` and the mutex are torn down simultaneously. Destroy in the reverse of the order in which a thread might still be holding or waiting on each primitive: cvs first (waiters drained), then the `sx` (no readers or writers left), then the mutex (no interlock partner left).

12.7.9 Helpful Asserts to Add

Sprinkle `sx_assert` and `mtx_assert` calls throughout the helpers to document the expected lock state. Each is free at production time (`INVARIANTS` is compiled out) and catches new bugs on the debug kernel.

Examples:

```
static int
myfirst_get_debug_level(struct myfirst_softc *sc)
{
    int level;

    sx_slock(&sc->cfg_sx);
    sx_assert(&sc->cfg_sx, SA_SLOCKED); /* document the lock state */
    level = sc->cfg.debug_level;
    sx_sunlock(&sc->cfg_sx);
    return (level);
}
```

The `sx_assert(&sc->cfg_sx, SA_SLOCKED)` is technically redundant after `sx_slock` (the lock was just acquired), but it makes the intent obvious to a reader and catches refactoring mistakes (someone moves the function and forgets the lock).

A more useful pattern: assertions in helpers that *expect* the caller to have acquired a lock:

```
static void
myfirst_apply_debug_level(struct myfirst_softc *sc, int level)
{
    sx_assert(&sc->cfg_sx, SA_XLOCKED); /* caller must hold xlock */
    sc->cfg.debug_level = level;
}
```

If a future call site tries to use this helper without the lock, the assertion fires. The function's expectations are now executable, not just documented.

12.7.10 Tracing Lock Activity With `dtrace` and `lockstat`

Two user-space tools let you observe lock behaviour without modifying the driver.

`lockstat(1)` summarises lock contention over a period:

```
# lockstat -P sleep 10
```

This runs for 10 seconds and prints a table of every lock that was contended, with hold times and wait times. For `myfirst` under `mp_stress`, you should see `myfirst0` (the device mutex) and `myfirst cfg` (the `sx`) at the top

of the list. Whether either has contention worth worrying about depends on the workload; for our pseudo-device, neither should.

`dtrace(1)` lets you trace specific events. To see every `cv_signal` on the data cv:

```
# dtrace -n 'fbt::cv_signal:entry /args[0]->cv_description == "myfirst data"/ {
↵  stack(); }'
```

This prints a kernel stack trace every time the signal is sent. Useful for pinpointing exactly which path is signalling and from where.

Both tools have minimal overhead and can be used on a production kernel as well as a debug one.

12.7.11 A Worked Walk-Through: A Lost Wakeup Bug

To make the diagnostic workflow concrete, walk through a hypothetical lost-wakeup bug from symptom to fix.

You make a change to the driver that subtly breaks the signal pairing in `myfirst_write`. After the change, the test suite mostly passes, but `mt_reader` occasionally hangs. You can reproduce it by running `mt_reader` ten or twenty times; once or twice the program hangs at one of its threads.

Step 1: confirm the symptom with `procstat`.

```
$ ps -ax | grep mt_reader
12345 ?? S+   mt_reader

$ procstat -kk 12345
  PID   TID COMM          TDNAME          KSTACK
12345 67890 mt_reader      -               mi_switch+0xc1 _cv_wait_sig+0xff
                                myfirst_wait_data+0x4e
                                myfirst_read+0x91
                                dofileread+0x82
                                sys_read+0xb5
```

The thread is in `_cv_wait_sig`. Looking up its wait channel:

```
$ ps -axHo pid,tid,wchan,command | grep mt_reader
12345 67890 myfirst      mt_reader
12345 67891 -          mt_reader
```

One thread blocked on `myfirst` (the eight-character truncation of "myfirst data"; see Section 2). The others have exited. So the cv data_cv has a waiter, and presumably the driver did not signal it when it should have.

Step 2: check the cbuf state.

```
$ sysctl dev.myfirst.0.stats.cb_used
dev.myfirst.0.stats.cb_used: 17
```

The buffer has 17 bytes in it. The reader should be able to drain those bytes and return them. Why is the reader still asleep?

Step 3: examine the source. The reader is in `cv_wait_sig(&sc->data_cv, &sc->mtx)`. Who calls `cv_signal(&sc->data_cv)`? Search:

```
$ grep -n 'cv_signal(&sc->data_cv|cv_broadcast(&sc->data_cv)' myfirst.c
180:         cv_signal(&sc->data_cv);
220:         cv_broadcast(&sc->data_cv); /* in detach */
```

Two callers. The relevant one is line 180, in `myfirst_write`. Look at it:

```
put = myfirst_buf_write(sc, bounce, want);
fh->writes += put;
MYFIRST_UNLOCK(sc);

if (put > 0) {
    cv_signal(&sc->data_cv);
    selwakeup(&sc->rsel);
}
```

The signal is conditional on `put > 0`. It looks correct. But the bug introduced earlier might have changed something else. Look further up:

```
MYFIRST_LOCK(sc);
error = myfirst_wait_room(sc, ioflag, nbfore, uio);
if (error != 0) {
    MYFIRST_UNLOCK(sc);
    return (error == -1 ? 0 : error);
}
room = cbuf_free(&sc->cb);
MYFIRST_UNLOCK(sc);

want = MIN((size_t)uio->uio_resid, sizeof(bounce));
want = MIN(want, room);
error = uiomove(bounce, want, uio);
if (error != 0)
    return (error);

MYFIRST_LOCK(sc);
put = myfirst_buf_write(sc, bounce, want);
```

Here is the bug. After `uiomove`, the code jumps straight back into the `cbuf` write. But what if `want` was computed against a stale room? Suppose between the `cbuf_free` call and the second `MYFIRST_LOCK`, another writer added bytes. The second `myfirst_buf_write` could be called with a `want` that exceeds the actual current room.

In our case, `myfirst_buf_write` returns the actual number of bytes written, which may be less than `want`. We update `bytes_written` correctly. But we then signal `data_cv` only if `put > 0`. So far so good.

But wait. Look at the buggy line carefully: imagine the change introduced was to wrap the signal in a different condition:

```
if (put == want) { /* WRONG: was put > 0 */
    cv_signal(&sc->data_cv);
    selwakeup(&sc->rsel);
}
```

Now if `put < want` (because another writer beat us to the room), we do not signal. The bytes were added to the `cbuf`, but the readers are not woken. A reader currently in `cv_wait_sig` will sleep until somebody else writes a complete buffer.

That is the lost-wakeup bug. The fix is to signal whenever `put > 0`, not whenever `put == want`. Apply the fix, rebuild, retest. The hang disappears.

Step 4: prevent regression. Add a `KASSERT` at the wakeup site that documents the contract:

```
KASSERT(put <= want, ("myfirst_buf_write returned %zu > want=%zu",
    put, want));
if (put > 0) {
    cv_signal(&sc->data_cv);
    selwakeup(&sc->rsel);
}
```

The `KASSERT` does not catch the bug we just fixed (it was triggered by `put != want`, which is permitted). But it documents that the signal-condition is “any forward progress”, which is the rule the next maintainer should preserve.

This walk-through is contrived; real bugs are messier. The pattern is real. Symptom -> instrumentation -> source examination -> hypothesis -> fix -> regression guard. Practice it on the labs in this chapter.

12.7.12 A Worked Walk-Through: a Lock Order Reversal

Another common scenario: `WITNESS` reports a reversal that you do not immediately understand.

The warning, simplified:

```
lock order reversal:
1st 0xfffffe000a1b2c30 myfirst cfg (myfirst cfg, sx) @ myfirst.c:120
2nd 0xfffffe000a1b2c50 myfirst0 (myfirst, sleep mutex) @ myfirst.c:240
```

```
lock order myfirst cfg -> myfirst0 attempted at myfirst.c:241
where myfirst0 -> myfirst cfg is established at myfirst.c:280
```

Step 1: identify the locks. `myfirst cfg` is the `sx`; `myfirst0` is the device mutex.

Step 2: identify the canonical order. From the warning: `myfirst0 -> myfirst cfg`. So `mtx` first, then `sx`.

Step 3: identify the violating path. The warning says the violating path is at `myfirst.c:241`, where it tries to acquire `myfirst0` while already holding `myfirst cfg`. Open the source at line 241 and trace upward to find when the `cfg sx` was acquired (line 120, by the warning's 1st field).

Step 4: decide the fix. Two options. Either reorder the violating path to match the canonical order (acquire `mtx` first, then `sx`; this is usually the cheaper change), or accept that the violating path has a real reason to acquire in the reverse order, in which case the canonical order needs to change globally and `LOCKING.md` updated.

For our `myfirst`, the violating path almost certainly should match the canonical. The fix is to read the `cfg` value via the snapshot-and-apply pattern: drop the `sx` before acquiring the `mtx`.

Step 5: verify. Apply the fix, rebuild, rerun the test that triggered the warning. `WITNESS` should now be silent. If it is not, the warning has shifted to a different path and you have a second violation to investigate.

12.7.13 Common Mistakes That Hide From WITNESS

`WITNESS` is excellent at what it checks but does not check everything. Three classes of bug it cannot detect:

Locks held across function pointers. If a function holds lock A and calls a callback that the function pointer is supplied by user-controlled configuration, `WITNESS` cannot predict what locks the callback might acquire. The lock order with respect to the callback is undefined. Avoid this pattern; if you must use it, document the acceptable lock states for any callback.

Race conditions on lockless fields. A field that is intentionally accessed without a lock is invisible to `WITNESS`. If two threads race on that field and the racing matters, `WITNESS` will not warn. Use atomics or the appropriate lock; never assume a lockless field is safe just because no warning fires.

Incorrect protection. A field is protected by a mutex on the write path but read without the mutex on the read path. Intermittent torn reads result. `WITNESS` does not flag this; the audit procedure from Chapter 11 Section 3 does.

The cure for all three is the discipline of writing `LOCKING.md` and keeping it accurate. `WITNESS` confirms that the locks you claim to hold, you actually hold; the document confirms that the rules you claim to follow are the rules the design intends.

12.7.14 Wrapping Up Section 6

Synchronization debugging is a vocabulary first and a toolset second. The vocabulary is the six failure shapes: lost wakeup, spurious wakeup, lock-order reversal, premature destroy, sleeping with a non-sleepable lock, race between detach and active operation. Each has a recognisable signature and a standard cure. The toolset is

WITNESS, INVARIANTS, KASSERT, `sx_assert`, the in-kernel debugger commands `show all locks` and `show sleepchain`, and the user-space observability tools `lockstat(1)` and `dtrace(1)`.

Section 7 puts these tools to work on a realistic stress scenario.

12.8 Section 7: Stress Testing with Realistic I/O Patterns

The Chapter 11 stress kit (`producer_consumer`, `mp_stress`, `mt_reader`, `lat_tester`) tested the data path under simple multi-threaded and multi-process workloads. The Chapter 12 driver has new primitives (`cv`, `sx`) and a new code path (the configuration `sysctls`). This section extends the testing to exercise the new surfaces and shows you how to read the resulting data.

The goal is not exhaustive coverage; it is *realistic* coverage. A driver that survives a stress workload that resembles real production traffic is a driver you can trust.

12.8.1 What Realistic Looks Like

A real driver typically has:

- Multiple producers and consumers running concurrently.
- `Sysctl` reads sprinkled throughout (monitoring tools, dashboards).
- Occasional `sysctl` writes (configuration changes from an administrator).
- Bursts of activity interleaved with idle periods.
- Mixed-priority threads competing for the CPU.

A test that exercises only one of these axes can miss bugs that only appear when several axes interact. For example, a `sysctl` write that takes the `cfg sx` in exclusive mode while a data path is mid-operation might expose a subtle ordering issue that pure I/O testing would not.

12.8.2 A Composite Workload

Build a script that runs three things at once for a fixed duration:

```
#!/bin/sh
# Composite stress: I/O + sysctl readers + sysctl writers.

DUR=60

(./mp_stress &) >/tmp/mp.log
(./mt_reader &) >/tmp/mt.log
(./config_writer $DUR &) >/tmp/cw.log

# Burst sysctl readers
for i in 1 2 3 4; do
    (while sleep 0.5; do
```

```

        sysctl -q dev.myfirst.0.stats >/dev/null
        sysctl -q dev.myfirst.0.debug_level >/dev/null
        sysctl -q dev.myfirst.0.soft_byte_limit >/dev/null
    done) &
done
SREAD_PIDS=$!

sleep $DUR

# Stop everything
pkill -f mp_stress
pkill -f mt_reader
pkill -f config_writer
kill $SREAD_PIDS 2>/dev/null

wait

echo "=== mp_stress ==="
cat /tmp/mp.log
echo "=== mt_reader ==="
cat /tmp/mt.log
echo "=== config_writer ==="
cat /tmp/cw.log

```

The script runs for one minute. During that minute, the driver sees:

- Two writer processes hammering writes.
- Two reader processes hammering reads.
- Four pthreads in `mt_reader` hammering reads on a single descriptor.
- The `config_writer` toggling debug level and soft byte limit every 10 ms.
- Four shell loops reading sysctls every 0.5 seconds.

Under a debug kernel with `WITNESS`, this is enough activity to catch most lock-ordering and signal-pairing bugs. Run it. If it completes without panic, without `WITNESS` warnings, and with consistent byte counts in the readers and writers, the driver has passed a meaningful synchronization test.

12.8.3 Long-Duration Variant

For the most subtle bugs, run the composite for an hour:

```

$ for i in $(seq 1 60); do
    ./composite_stress.sh
    echo "iteration $i complete"
    sleep 5
done

```


Sixty iterations of a one-minute test gives an hour of cumulative coverage. Bugs that appear once per million events (which is roughly what an hour of `mp_stress` produces on a modern machine) usually surface during this run.

12.8.4 Latency Under Mixed Load

The Chapter 11 `lat_tester` measured the latency of a single read with no other load. Under realistic load, latency tells a different story: it includes time waiting for the mutex, time waiting for the sx, and time inside `cv_wait_sig`.

Run `lat_tester` while `mp_stress` and `config_writer` are running. The histogram should show a longer tail than the unloaded case. A few microseconds for uncontested operations, a few tens of microseconds when the mutex is briefly held by another thread, and a small spike of milliseconds when the cv had to actually sleep waiting for data. If the tail extends to seconds, something is wrong.

12.8.5 Reading lockstat Output

`lockstat(1)` is the canonical tool for measuring lock contention. Run it during a heavy stress:

```
# lockstat -P sleep 30 > /tmp/lockstat.out
```

The `-P` flag includes spin lock data; without it, only adaptive locks are reported. The 30 means “sample for 30 seconds”.

The output is organized by lock, with hold time and wait time statistics. For our driver, look for lines mentioning `myfirst0` (the `mtx`), `myfirst cfg` (the `sx`), and the `cvs` (`myfirst data`, `myfirst room`).

A healthy result for `myfirst` under typical stress:

- The `mtx` is acquired millions of times. Hold time per acquisition is tens of nanoseconds. Wait time is occasional and small.
- The `sx` is acquired tens of thousands of times. Most acquisitions are shared; the few exclusive acquisitions correspond to `sysctl` writes. Hold time is low.
- The `cvs` are signalled and broadcast in proportion to the I/O rate. Wait counts on each `cv` correspond to the number of times a reader or writer actually had to block.

If any lock shows wait time as a significant fraction of total time, that lock is contended. The fix is one of: shorter critical sections, finer-grained locking, or a different primitive.

For our pseudo-device with a single-mutex design on the data path, the `mtx` will saturate around 4-8 cores depending on the speed of `cbuf` operations. This is expected; we have not optimized for high core counts. The point of the chapter is correctness, not throughput.

12.8.6 Tracing With dtrace

When a specific event needs visibility, `dtrace` is the right tool. Example: count how many times each `cv` was signalled during a 10-second window:

```
# dtrace -n 'fbt::cv_signal:entry { @[args[0]->cv_description] = count(); }' \
-n 'fbt::cv_broadcastpri:entry { @[args[0]->cv_description] = count(); }' \
-n 'tick-10sec { exit(0); }'
```

After 10 seconds, dtrace prints a table:

```
myfirst data      48512
myfirst room      48317
...
```

The numbers should be roughly equal for data_cv and room_cv if the workload is symmetric (equal reads and writes). A large imbalance suggests one side is sleeping more than the other, which usually means a flow-control issue.

Another useful one-liner: histogram of cv_wait latency on the data cv:

```
# dtrace -n '
fbt::_cv_wait_sig:entry /args[0]->cv_description == "myfirst data"/ {
    self->ts = timestamp;
}
fbt::_cv_wait_sig:return /self->ts/ {
    @ = quantize(timestamp - self->ts);
    self->ts = 0;
}
tick-10sec { exit(0); }
'
```

The histogram shows the distribution of times threads spent inside _cv_wait_sig. Most should be short (signalled promptly). A long tail indicates threads sleeping for extended periods, which is normal for an idle device but suspicious for a busy one.

12.8.7 Watching With vmstat and top

For a coarser view, vmstat and top running in the background provide context.

vmstat 1 shows per-second statistics: CPU time spent in user, system, and idle; context switches; interrupts. During a stress run, sy (system time) should rise; cs (context switches) should rise as well, due to the cv signalling.

top -SH (the -S shows system processes; -H shows individual threads) shows per-thread CPU usage. During a stress run, the test threads should be visible. The WCHAN column shows what they are waiting on; expect to see the truncated cv descriptions (myfirst for both data_cv and room_cv, since the trailing word is cut by WMESGLEN) plus, for any thread still using the Chapter 11 anonymous channel, the address of &sc->cb printed as a small numeric string.

Both are useful as background companions to a long stress run. They do not produce structured data, but they confirm at a glance that things are happening.

12.8.8 Observing the Sysctls

A simple sanity check during the stress: read the sysctls periodically and verify they make sense.

```
$ while sleep 1; do
    sysctl dev.myfirst.0.stats.bytes_read \
        dev.myfirst.0.stats.bytes_written \
        dev.myfirst.0.stats.cb_used \
        dev.myfirst.0.debug_level \
        dev.myfirst.0.soft_byte_limit
done
```

The byte counters should monotonically increase. The `cb_used` should hover in some range. The configuration should change as `config_writer` updates it.

If any `sysctl` read hangs (the `sysctl` command does not return), there is a synchronization problem with the `sysctl` handler. Probably a held mutex blocking the `sysctl` from acquiring the `sx`, or vice versa. Use `procstat -kk $$` from another terminal to see what the hanging shell is waiting on.

12.8.9 Stress Test Acceptance Criteria

A driver passes a synchronization stress test if:

1. The composite script completes without panic.
2. WITNESS reports no warnings (`dmesg | grep -i witness | wc -l` returns zero).
3. Byte counts from readers and writers are within 1% of each other (small drift is acceptable due to timing of test stop).
4. `lockstat(1)` shows no lock with wait time exceeding 5% of total time.
5. The latency histogram from `lat_tester` shows the 99th percentile under one millisecond for an idle device, or under the configured timeout for a busy one.
6. Repeated runs (the long-duration loop) all pass.

These are not absolute thresholds; they are the values that have served the chapter's example. Real drivers may have stricter or looser bounds depending on their workload.

12.8.10 Interpreting lockstat Output in Detail

`lockstat(1)` produces tables that look intimidating on first encounter. A short tour of the columns demystifies them.

A typical line for a contended lock:

```
Adaptive mutex spin: 1234 events in 30.000 seconds (41.13 events/sec)
```

```
-----
Count    nsec    ----- Lock -----    Hottest Caller
1234      321    myfirst0                    myfirst_read+0x91
```

What the columns mean:

- Count: number of events of this kind (acquires, in this case).
- nsec: average duration of the event (here, average time spinning before acquiring the lock).
- Lock: the lock's name.
- Hottest Caller: the function that most often experienced this event.

Lower in the output:

```
Adaptive mutex block: 47 events in 30.000 seconds (1.57 events/sec)
```

```
-----
Count    nsec      ----- Lock -----      Hottest Caller
   47    58432    myfirst0                      myfirst_read+0x91
```

The “block” event is when the spin failed and the thread had to actually sleep. Average sleep time was 58 microseconds. That is high; it means a writer was holding the mutex during what should have been a short critical section.

Taken together, the spin events (1234) and block events (47) tell us the lock was contended 1281 times in 30 seconds, and 96% of the time the spin succeeded. That is a healthy pattern: most contention is brief, and only the rare long hold causes a real sleep.

For sleep locks (sx, cv), the columns are similar but the events are categorized differently:

```
SX shared block: 2014 events in 30.000 seconds (67.13 events/sec)
```

```
-----
Count    nsec      ----- Lock -----      Hottest Caller
  2014    2105    myfirst cfg                      myfirst_get_debug_level+0x12
```

This says: shared waiters on the cfg sx blocked 2014 times, average wait 2.1 microseconds, mostly from the debug-level helper. With a config writer running, that is expected. Without the writer, it should be near zero.

The key skill in reading lockstat output is calibration: knowing what numbers are expected for your workload. A driver that has never been measured under load is a driver whose expected numbers are unknown. Run lockstat once with a known workload and save the output as the baseline. Future runs are then compared to the baseline; significant deviations are signal.

12.8.11 Tracing Specific Code Paths With dtrace

Beyond the cv-counting and sleep-latency examples earlier, a few more dtrace recipes are useful for a chapter-12-style driver.

Count cv waits per cv per second:

```
# dtrace -n '
fbt::_cv_wait_sig:entry { @[args[0]->cv_description] = count(); }
tick-1sec { printa(@); trunc(@); }'
```

Prints a per-second tally of cv waits, broken down by cv name. Useful for spotting bursts.

Trace which thread acquires the cfg sx exclusively:

```
# dtrace -n '
fbt::_sx_xlock:entry /args[0]->lock_object.lo_name == "myfirst cfg"/ {
    printf("%s pid %d acquires cfg xlock\n", execname, pid);
    stack();
}'
```

Useful for confirming that the only writers are the sysctl handlers, not some other unexpected path.

Histogram of myfirst_read latency:

```
# dtrace -n '
fbt::myfirst_read:entry { self->ts = timestamp; }
fbt::myfirst_read:return /self->ts/ {
    @ = quantize(timestamp - self->ts);
    self->ts = 0;
}
tick-30sec { exit(0); }'
```

Same pattern as the cv-wait latency histogram, but at the handler level. Includes the time spent inside `cv_wait_sig` plus the time inside the `cbuf` operations and the `uiomove`.

These recipes are starting points. The `dtrace` provider for kernel functions (`fbt`) gives access to every function entry and return; the language is rich enough to express almost any aggregation.

12.8.12 Wrapping Up Section 7

Realistic stress testing exercises the whole driver, not just one path. A composite workload that combines I/O, `sysctl` reads, and `sysctl` writes catches lock-ordering bugs that pure I/O testing would miss. `lockstat(1)` and `dtrace(1)` give you observability into the lock and cv activity without modifying the driver. A driver that passes the composite stress kit on a WITNESS kernel for an hour is a driver you can promote to the next chapter with confidence.

Section 8 closes the chapter with the housekeeping work: the documentation pass, the version bump, the regression test, and the changelog entry that tells your future self what you did and why.

12.9 Section 8: Refactoring and Versioning Your Synchronized Driver

The driver now uses three primitives (mtx, cv, sx), has two lock classes with a documented order, supports interruptible and timed reads, and has a small configuration subsystem. The remaining work is the housekeeping pass: cleaning up the source for clarity, updating the documentation, bumping the version, running static analysis, and validating the regression test passes.

This section covers each. None of it is glamorous. All of it is what separates a working driver from a maintainable one.

12.9.1 Cleaning Up the Source

After a chapter of focused changes, the source has accumulated some inconsistencies that are worth tidying.

Group related code. Move all the cv-related helpers next to each other (the wait helpers, the signal calls, the cv_init/cv_destroy in attach/detach). Move all the sx-related helpers together. The compiler does not care about ordering, but a reader does.

Standardise the macro vocabulary. Chapter 11 introduced MYFIRST_LOCK, MYFIRST_UNLOCK, MYFIRST_ASSERT. Add the symmetric set for the sx:

```
#define MYFIRST_CFG_SLOCK(sc)    sx_slock(&(sc)->cfg_sx)
#define MYFIRST_CFG_SUNLOCK(sc) sx_sunlock(&(sc)->cfg_sx)
#define MYFIRST_CFG_XLOCK(sc)   sx_xlock(&(sc)->cfg_sx)
#define MYFIRST_CFG_XUNLOCK(sc) sx_xunlock(&(sc)->cfg_sx)
#define MYFIRST_CFG_ASSERT_X(sc) sx_assert(&(sc)->cfg_sx, SA_XLOCKED)
#define MYFIRST_CFG_ASSERT_S(sc) sx_assert(&(sc)->cfg_sx, SA_SLOCKED)
```

Now every lock acquisition in the driver goes through a macro. If we later switch from sx to rw, the change is in one header, not scattered across the source.

Eliminate dead code. If a helper from Chapter 11 is no longer called (perhaps the old wakeup channel is gone), remove it. Dead code attracts confusion.

Comment the non-obvious bits. Every lock acquisition that follows the lock-order rule deserves a one-line comment. Every place where snapshot-and-apply is used deserves a comment explaining why. The locking is the most subtle part of the driver; the comments should reflect that.

12.9.2 Updating LOCKING.md

The Chapter 11 LOCKING.md documented one lock and a small set of fields. The Chapter 12 driver has more to say. The new version:

```
# myfirst Locking Strategy
```

```
Version 0.6-sync (Chapter 12).
```

Overview

The driver uses three synchronization primitives: a sleep mutex (sc->mtx) for the data path, an sx lock (sc->cfg_sx) for the configuration subsystem, and two condition variables (sc->data_cv, sc->room_cv) for blocking reads and writes. Byte counters use counter(9) per-CPU counters and protect themselves.

Locks Owned by This Driver

sc->mtx (mutex(9), MTX_DEF)

Protects:

- sc->cb (the circular buffer's internal state)
- sc->open_count, sc->active_fhs
- sc->is_attached (writes; reads at handler entry may be unprotected as an optimization, re-checked after every sleep)

Lock-Free Plain Integers

- sc->read_timeout_ms, sc->write_timeout_ms: plain ints, accessed without locking. Safe because aligned int reads and writes are atomic on every architecture FreeBSD supports, and the values are advisory; a stale read just produces a slightly different timeout for the next wait. The sysctl framework writes them directly via CTLFLAG_RW.

sc->cfg_sx (sx(9))

Protects:

- sc->cfg.debug_level
- sc->cfg.soft_byte_limit
- sc->cfg.nickname

Shared mode: every read of any cfg field.

Exclusive mode: every write of any cfg field.

sc->data_cv (cv(9))

Wait condition: data is available in the cbuf.

Interlock: sc->mtx.

Signalled by: myfirst_write after a successful cbuf write.

Broadcast by: myfirst_detach.

Waiters: myfirst_read in myfirst_wait_data.

`sc->room_cv (cv(9))`

Wait condition: room is available in the cbuf.

Interlock: `sc->mtx`.

Signalled by: `myfirst_read` after a successful cbuf read, and `myfirst_sysctl_reset` after resetting the cbuf.

Broadcast by: `myfirst_detach`.

Waiters: `myfirst_write` in `myfirst_wait_room`.

Lock-Free Fields

- `sc->bytes_read`, `sc->bytes_written`: `counter_u64_t`. Updates via `counter_u64_add`; reads via `counter_u64_fetch`.

Lock Order

`sc->mtx` -> `sc->cfg_sx`

A thread holding `sc->mtx` may acquire `sc->cfg_sx` in either mode.

A thread holding `sc->cfg_sx` may NOT acquire `sc->mtx`.

Rationale: the data path always holds `sc->mtx` and may need to read configuration during its critical section. The configuration path (sysctl writers) does not need the data mutex; if a future feature requires both, it must acquire `sc->mtx` first.

Locking Discipline

1. Acquire mutex with `MYFIRST_LOCK(sc)`, release with `MYFIRST_UNLOCK(sc)`.
2. Acquire `sx` in shared mode with `MYFIRST_CFG_SLOCK`, exclusive with `MYFIRST_CFG_XLOCK`. Release with the matching unlock.
3. Wait on a `cv` with `cv_wait_sig` (interruptible) or `cv_timedwait_sig` (interruptible + bounded).
4. Signal a `cv` with `cv_signal` (one waiter) or `cv_broadcast` (all waiters). Use `cv_broadcast` only for state changes that affect all waiters (detach, configuration reset).
5. NEVER hold `sc->mtx` across `uiomove(9)`, `copyin(9)`, `copyout(9)`, `selwakeup(9)`, or `wakeup(9)`. Each of these may sleep or take other locks. `cv_wait_sig` is the exception (it atomically drops the interlock).
6. NEVER hold `sc->cfg_sx` across `uiomove(9)` etc., for the same reason.
7. All `cbuf_*` calls must happen with `sc->mtx` held (the helpers


```
assert MA_OWNED).
```

8. The detach path clears `sc->is_attached` under `sc->mtx`, broadcasts both cvs, and refuses detach while `active_fhs > 0`.

Snapshot-and-Apply Pattern

When a path needs both `sc->mtx` and `sc->cfg_sx`, it should follow the snapshot-and-apply pattern:

1. `sx_slock(&sc->cfg_sx)`; read `cfg` into local variables; `sx_sunlock(&sc->cfg_sx)`.
2. `MYFIRST_LOCK(sc)`; do `cbuf` operations using the snapshot; `MYFIRST_UNLOCK(sc)`.

The snapshot may be slightly stale by the time it is used. For configuration values that are advisory (debug level, soft byte limit), this is acceptable.

Known Non-Locked Accesses

`sc->is_attached` at handler entry

Unprotected plain read. Safe because:

- A stale "true" is re-checked after every sleep via `if (!sc->is_attached) return (ENXIO)`.
- A stale "false" causes the handler to return `ENXIO` early, which is also what it would do with a fresh false.

`sc->open_count`, `sc->active_fhs` at `sysctl` read time

Unprotected plain loads. Safe on amd64 and arm64 (aligned 64-bit loads are atomic). Acceptable on i386 because the torn read, if it ever happened, would produce a single bad statistic with no correctness impact.

Wait Channels

- `sc->data_cv`: data has become available.
- `sc->room_cv`: room has become available.

(The legacy `&sc->cb` wakeup channel from Chapter 10 has been retired in Chapter 12.)

History

- 0.6-sync (Chapter 12): added cv channels, sx for configuration, bounded reads via cv_timedwait_sig.
- 0.5-kasserts (Chapter 11, Stage 5): KASSERT calls added throughout cbuf helpers and wait helpers.
- 0.5-counter9 (Chapter 11, Stage 3): byte counters migrated to counter(9).
- 0.5-concurrency (Chapter 11, Stage 2): MYFIRST_LOCK/UNLOCK/ASSERT macros, explicit locking strategy.
- Earlier versions: see Chapter 10 / Chapter 11 history.

That document is now the authoritative description of the driver's synchronization story. Any future change updates the document in the same commit as the code change. A reviewer who wants to know whether a change is safe reads the diff against the document, not against the code.

12.9.3 Bumping the Version

Update the version string:

```
#define MYFIRST_VERSION "0.6-sync"
```

Print it at attach (the existing device_printf line in attach already includes the version):

```
device_printf(dev,
    "Attached; version %s, node /dev/%s (alias /dev/myfirst), "
    "cbuf=%zu bytes\n",
    MYFIRST_VERSION, devtoname(sc->cdev), cbuf_size(&sc->cb));
```

Update the changelog:

0.6-sync (Chapter 12)

- Replaced anonymous wakeup channel (&sc->cb) with two named condition variables (sc->data_cv, sc->room_cv).
- Added bounded read support via sc->read_timeout_ms sysctl, using cv_timedwait_sig under the hood.
- Added a small configuration subsystem (sc->cfg) protected by an sx lock (sc->cfg_sx).
- Added sysctl handlers for debug_level, soft_byte_limit, and nickname.
- Added myfirst_sysctl_reset that takes both locks in the canonical order to clear the cbuf and reset counters.
- Updated LOCKING.md with the new primitives, the lock order, and

- the snapshot-and-apply pattern.
- Added MYFIRST_CFG_* macros symmetric with the existing MYFIRST_* mutex macros.
- All Chapter 11 tests continue to pass; new sysctl-based tests added under userland/.

12.9.4 Updating the README

The Chapter 11 README named the driver and described its features. The Chapter 12 README adds the new ones:

myfirst

A FreeBSD 14.3 pseudo-device driver that demonstrates buffered I/O, concurrency, and modern synchronization primitives. Developed as the running example for the book "FreeBSD Device Drivers: From First Steps to Kernel Mastery."

Status

Version 0.6-sync (Chapter 12).

Features

- A Newbus pseudo-device under nexus0.
- A primary device node at /dev/myfirst/0 (alias: /dev/myfirst).
- A circular buffer (cbuf) as the I/O buffer.
- Blocking, non-blocking, and timed reads and writes.
- poll(2) support via d_poll and selinfo.
- Per-CPU byte counters via counter(9).
- A single sleep mutex protects composite cbuf state; see LOCKING.md.
- Two named condition variables (data_cv, room_cv) coordinate read and write blocking.
- An sx lock protects the runtime configuration (debug_level, soft_byte_limit, nickname).

Configuration

Three runtime-tunable parameters via sysctl:

- dev.myfirst.<unit>.debug_level (0-3): controls dmesg verbosity.
- dev.myfirst.<unit>.soft_byte_limit: refuse writes that would push cb_used above this threshold (0 = no limit).
- dev.myfirst.<unit>.nickname: a string used in log messages.

```
- dev.myfirst.<unit>.read_timeout_ms: bound a blocking read.
```

(The last is per-instance; see myfirst.4 for details, when written.)

Build and Load

```
$ make
# kldload ./myfirst.ko
# dmesg | tail
# ls -l /dev/myfirst
# printf 'hello' > /dev/myfirst
# cat /dev/myfirst
# kldunload myfirst
```

Tests

See ../../userland/ for the test programs. The Chapter 12 tests include config_writer (toggles sysctls during stress) and timeout_tester (verifies bounded reads).

License

BSD 2-Clause. See individual source files for SPDX headers.

12.9.5 Running Static Analysis

Run clang --analyze against the Chapter 12 driver:

```
$ make WARNS=6 clean all
$ clang --analyze -D_KERNEL -I/usr/src/sys \
  -I/usr/src/sys/amd64/conf/GENERIC myfirst.c
```

Triage the output. New warnings since Chapter 11 should be either:

1. False positives (clang does not understand the locking discipline). Document each.
2. Real bugs. Fix each.

Common false positives in driver code involve `sx_assert` and `mtx_assert` macros that clang cannot see through; the analyser thinks the lock might not be held even when the assert proves it is. These are acceptable to silence with `__assert_unreachable()` or by restructuring the code to make the lock state more obvious to the analyser.

12.9.6 Running the Regression Suite

The Chapter 11 regression script extends naturally:

```
#!/bin/sh
# regression.sh: full Chapter 12 regression.

set -eu

die() { echo "FAIL: $" ">&2; exit 1; }
ok() { echo "PASS: $"; }

[ $(id -u) -eq 0 ] || die "must run as root"
kldstat | grep -q myfirst && kldunload myfirst
[ -f ./myfirst.ko ] || die "myfirst.ko not built; run make first"

kldload ./myfirst.ko
trap 'kldunload myfirst 2>/dev/null || true' EXIT

sleep 1
[ -c /dev/myfirst ] || die "device node not created"
ok "load"

# Chapter 7-10 tests.
printf 'hello' > /dev/myfirst || die "write failed"
cat /dev/myfirst >/tmp/out.$$
[ "$(cat /tmp/out.$$)" = "hello" ] || die "round-trip content mismatch"
rm -f /tmp/out.$$
ok "round-trip"

cd ../userland && make -s clean && make -s && cd -

../userland/producer_consumer || die "producer_consumer failed"
ok "producer_consumer"

../userland/mp_stress || die "mp_stress failed"
ok "mp_stress"

# Chapter 12-specific tests.
../userland/timeout_tester || die "timeout_tester failed"
ok "timeout_tester"

../userland/config_writer 5 &
CW=$!
../userland/mt_reader || die "mt_reader (under config writer) failed"
wait $CW
ok "mt_reader under config writer"
```

```

sysctl dev.myfirst.0.stats >/dev/null || die "sysctl stats not accessible"
sysctl dev.myfirst.0.debug_level >/dev/null || die "sysctl debug_level not accessible"
sysctl dev.myfirst.0.soft_byte_limit >/dev/null || die "sysctl soft_byte_limit not
↪ accessible"
ok "sysctl"

# WITNESS check.
WITNESS_HITS=$(dmesg | grep -ci "witness\|lor" || true)
if [ "$WITNESS_HITS" -gt 0 ]; then
    die "WITNESS warnings detected ($WITNESS_HITS lines)"
fi
ok "witness clean"

echo "ALL TESTS PASSED"

```

A green run after each commit is the minimum bar. A green run on a WITNESS kernel after a long-duration composite is the higher bar.

12.9.7 Pre-Commit Checklist

The Chapter 11 checklist gets two new items for Chapter 12:

1. Have I updated `LOCKING.md` with any new locks, cvs, or order changes?
2. Have I run the full regression suite on a WITNESS kernel?
3. Have I run the long-duration composite stress for at least 30 minutes?
4. Have I run `clang --analyze` and triaged every new warning?
5. Have I added a `sx_assert` or `mtx_assert` for any new helper that expects a lock state?
6. Have I bumped the version string and updated `CHANGELOG.md`?
7. Have I verified that the test kit builds and runs?
8. **(New)** Have I checked that every `cv` has both a signaller and a documented condition?
9. **(New)** Have I checked that every `sx_xlock` has a paired `sx_xunlock` on every code path, including error paths?

The two new items capture the most common bugs in Chapter 12-style code. A `cv` without a signaller is dead weight (waiters will never wake). An `sx_xlock` without a paired unlock on an error path is a quiet deadlock waiting to happen.

12.9.8 Wrapping Up Section 8

The driver is now not only correct but verifiably correct, well-documented, and versioned. It has:

- An updated `LOCKING.md` describing three primitives, two lock classes, and one canonical lock order.
- A new version string (0.6-sync) reflecting the Chapter 12 work.
- A regression script that exercises every primitive and validates WITNESS cleanliness.
- A pre-commit checklist that catches the two new failure modes Chapter 12 introduced.

That is the close of the chapter's main teaching arc. The labs and challenges follow.

12.10 Hands-on Labs

These labs consolidate the Chapter 12 concepts through direct hands-on experience. They are ordered from least to most demanding. Each is designed to be completable in a single lab session.

12.10.1 Pre-Lab Setup Checklist

Before starting any lab, confirm the four items below. The Chapter 11 checklist applies; we add three Chapter 12-specific ones.

1. **Debug kernel running.** `sysctl kern.ident` reports the kernel with `INVARIANTS` and `WITNESS`.
2. **WITNESS active.** `sysctl debug.witness.watch` returns a non-zero value.
3. **Driver source matches Chapter 11 Stage 5 (kasserts).** From your driver directory, `make clean && make` should compile cleanly.
4. **A clean dmesg.** `dmesg -c >/dev/null` once before the first lab.
5. **(New) Companion userland built.** From `examples/part-03/ch12-synchronization-mechanisms/userland/`, `make` should produce the `config_writer` and `timeout_tester` binaries.
6. **(New) Chapter 11 stress kit available.** The labs reuse `mp_stress`, `mt_reader`, and `producer_consumer` from Chapter 11.
7. **(New) Backup of Stage 5.** Copy the working Stage 5 driver to a safe location before starting any lab that modifies the source. Several labs intentionally introduce bugs that need to be reverted cleanly.

12.10.2 Lab 12.1: Replace Anonymous Wakeup Channels With Condition Variables

Goal. Convert the Chapter 11 driver from `mtx_sleep/wakeup` on the anonymous channel `&sc->cb` to two named condition variables (`data_cv` and `room_cv`).

Steps.

1. Copy your Stage 5 driver into `examples/part-03/ch12-synchronization-mechanisms/stage1-cv-channels/`.
2. Add `struct cv data_cv` and `struct cv room_cv` to `struct myfirst_softc`.
3. In `myfirst_attach`, call `cv_init(&sc->data_cv, "myfirst data")` and `cv_init(&sc->room_cv, "myfirst room")`. Place them after the mutex init.
4. In `myfirst_detach`, before `mtx_destroy`, call `cv_broadcast` on each `cv` to wake any sleepers, then `cv_destroy` on each.
5. Replace the `mtx_sleep(&sc->cb, ...)` calls in `myfirst_wait_data` and `myfirst_wait_room` with `cv_wait_sig(&sc->data_cv, &sc->mtx)` and `cv_wait_sig(&sc->room_cv, &sc->mtx)` respectively.
6. Replace the `wakeup(&sc->cb)` calls in `myfirst_read` and `myfirst_write` with `cv_signal(&sc->room_cv)` and `cv_signal(&sc->data_cv)` respectively. Note the swap: a successful read frees room (so wake writers); a successful write produces data (so wake readers).
7. Build, load, run the Chapter 11 stress kit.

Verification. All Chapter 11 tests pass. `procstat -kk` against a sleeping reader shows wait channel `myfirst` (the truncated form of "myfirst data"; see the Section 2 note about `WMESGLEN`). No `WITNESS` warnings.

Stretch goal. Use `dtrace` to count signals to each `cv` during `mp_stress`. Confirm that the signal counts are roughly equal between `data_cv` and `room_cv` (because reads and writes are roughly equal).

12.10.3 Lab 12.2: Add Bounded Reads

Goal. Add a `read_timeout_ms` sysctl that bounds blocking reads.

Steps.

1. Copy Lab 12.1 into `stage2-bounded-read/`.
2. Add an `int read_timeout_ms` field to the `softc`. Initialize to 0 in `attach`.
3. Register a `SYSCTL_ADD_INT` for it under `dev.myfirst.<unit>.read_timeout_ms`, with `CTLFLAG_RW`.
4. Modify `myfirst_wait_data` to use `cv_timedwait_sig` when `read_timeout_ms > 0`, converting milliseconds to ticks. Translate `EWOULDBLOCK` to `EAGAIN`.
5. Build and load.
6. Build the `timeout_tester` from `examples/part-03/ch12-synchronization-mechanisms/userland/`.
7. Set the sysctl to 100, run `timeout_tester`, observe that `read(2)` returns `EAGAIN` after about 100 ms.
8. Reset the sysctl to 0, run `timeout_tester` again. The read blocks until you Ctrl-C, returning `EINTR`.

Verification. The output of `timeout_tester` matches expectations for both timeout and signal-interruption cases. Stress kit still passes.

Stretch goal. Add a symmetric `write_timeout_ms` sysctl and verify it bounds writes when the buffer is full.

12.10.4 Lab 12.3: Add an sx-Protected Configuration Subsystem

Goal. Add the `cfg` struct and `cfg_sx` lock from Section 5; expose `debug_level` as a sysctl.

Steps.

1. Copy Lab 12.2 into `stage3-sx-config/`.
2. Add struct `sx cfg_sx` and struct `myfirst_config cfg` to the `softc`. Initialize in `attach` (`sx_init(&sc->cfg_sx, "myfirst cfg");` default values for `cfg` fields). Destroy in `detach`.
3. Add a `myfirst_sysctl_debug_level` handler following the snapshot-and-apply pattern. Register it.
4. Add a `MYFIRST_DBG(sc, level, fmt, ...)` macro that consults `sc->cfg.debug_level` via `sx_slock`.
5. Sprinkle a few `MYFIRST_DBG(sc, 1, ...)` calls in the read/write paths to log when the buffer becomes empty or full.
6. Build and load.
7. Run `mp_stress`. Confirm no log spam (`debug_level` defaults to 0).
8. `sysctl -w dev.myfirst.0.debug_level=2` and run `mp_stress` again. Now `dmesg` should show debug messages.
9. Reset the level to 0.

Verification. Debug messages appear and disappear as the sysctl changes. No `WITNESS` warnings during the toggle.

Stretch goal. Add the `soft_byte_limit` sysctl. Set it to 1024 and run a writer that produces bursts of 4096 bytes; confirm that the writer sees EAGAIN early.

12.10.5 Lab 12.4: Inspect Held Locks With DDB

Goal. Use the in-kernel debugger to inspect a hung test.

Steps.

1. Make sure the debug kernel has options `DDB` and a configured way to drop into DDB (typically `Ctrl-Alt-Esc` on a serial console, or the Break key).
2. Load Lab 12.3's driver.
3. Start a `cat /dev/myfirst` in one terminal. It blocks (no producer).
4. From the console (or via `sysctl debug.kdb.enter=1`), drop into DDB.
5. Run `show all locks`. Note any threads holding locks.
6. Run `ps`. Find the `cat` process and the `myfirst` data wait channel.
7. Run `bt <pid> <tid>` for the `cat` thread. Confirm the backtrace ends in `_cv_wait_sig`.
8. `continue` to leave DDB.
9. Send `SIGINT` to the `cat` (Ctrl-C).

Verification. The `cat` returns with `EINTR`. No panics. You have a transcript of the DDB session.

Stretch goal. Repeat with `mp_stress` running concurrently. Compare `show all locks` output: more locks, more activity, but the same shape.

12.10.6 Lab 12.5: Detect a Deliberate Lock Order Reversal

Goal. Introduce a deliberate LOR and observe WITNESS catching it.

Steps.

1. Copy Lab 12.3 into a scratch directory `stage-lab12-5/`. Do not modify Lab 12.3 in place.
2. Add a path that violates the lock order. For example, in a small experimental sysctl handler:

```
/* WRONG: sx first, then mtx, reversing the canonical order. */
sx_xlock(&sc->cfg_sx);
MYFIRST_LOCK(sc);
/* trivial work */
MYFIRST_UNLOCK(sc);
sx_xunlock(&sc->cfg_sx);
```

3. Build and load on the WITNESS kernel.
4. Run `mp_stress` (which exercises the canonical order via the data path) and trigger the new sysctl simultaneously.
5. Watch `dmesg` for the `lock order reversal` warning.
6. Record the warning text. Note the line numbers.

7. Delete the scratch directory; do not commit the bug.

Verification. `dmesg` shows a `lock_order_reversal` warning naming both locks and both source locations.

Stretch goal. Determine, just from the `WITNESS` output, where the canonical order was first established. Open the source at that line and confirm.

12.10.7 Lab 12.6: Long-Running Composite Stress

Goal. Run the composite stress workload from Section 7 for 30 minutes and verify clean.

Steps.

1. Boot the debug kernel.
2. Build and load `examples/part-03/ch12-synchronization-mechanisms/stage4-final/`. This is the final integrated driver (cv channels + bounded reads + sx-protected configuration + the reset sysctl). All the Section 7 sysctls that the composite script touches are present here.
3. Build the userland testers.
4. Save the Section 7 composite stress script as `composite_stress.sh`.
5. Wrap it in a 30-minute loop:

```
for i in $(seq 1 30); do
    ./composite_stress.sh
    echo "iteration $i done"
done
```

6. Monitor `dmesg` periodically.
7. After completion, check:
 - `dmesg | grep -ci witness` returns 0.
 - All loop iterations completed.
 - `vmstat -m | grep cbuf` shows the expected static allocation (no growth).

Verification. All criteria met. The driver survives 30 minutes of composite stress on a debug kernel without warning, panic, or memory growth.

Stretch goal. Run the same loop for 24 hours on a dedicated test machine. Bugs that appear at this scale are the ones that cost the most in production.

12.10.8 Lab 12.7: Verify the Snapshot-and-Apply Pattern Holds Under Contention

Goal. Show that the snapshot-and-apply pattern in `myfirst_write` correctly handles concurrent updates to the soft byte limit.

Steps.

1. Set the soft byte limit to a small value: `sysctl -w dev.myfirst.0.soft_byte_limit=512`.
2. Start `mp_stress` with two writers and two readers.

3. From a third terminal, repeatedly toggle the limit: `while sleep 0.1; do sysctl -w dev.myfirst.0.soft_byte_limit=$RANDOM; done.`
4. Watch the writer output. Some writes will succeed; others will return `EAGAIN` (the limit was below the current `cb_used` at the moment of check).
5. Watch `dmesg` for WITNESS warnings.

Verification. No WITNESS warnings. The byte counts in `mp_stress` are slightly lower than usual (because some writes were refused), but the total written approximately equals the total read.

Stretch goal. Modify `myfirst_write` to violate the lock-order rule by acquiring the `cfg sx` while holding the data mutex. Reload, run the same test. WITNESS should fire on the first run that exercises both paths simultaneously. Revert the change.

12.10.9 Lab 12.8: Profile With lockstat

Goal. Use `lockstat(1)` to characterize the contended locks under stress.

Steps.

1. Load Lab 12.3's driver on the debug kernel.
2. Start `mp_stress` in one terminal.
3. From another, run `lockstat -P sleep 30 > /tmp/lockstat.out.`
4. Open the output file. Find the entries for `myfirst0 (mtx)` and `myfirst cfg (sx)`.
5. Note: the maximum hold time, the average hold time, the maximum wait time, the average wait time, and the acquisition count.
6. Repeat with `config_writer` running. Compare the `myfirst cfg` numbers.

Verification. The numbers match the expected profile. The mutex shows millions of acquisitions with short hold times. The `sx` shows tens of thousands of acquisitions, mostly shared, with very short hold times.

Stretch goal. Modify the driver to artificially extend a critical section (e.g., add a 10 ms pause(9) inside the mutex). Re-run `lockstat`. Observe the contention spike. Revert the modification.

12.11 Challenge Exercises

The challenges extend Chapter 12 beyond the baseline labs. Each is optional; each is designed to deepen your understanding.

12.11.1 Challenge 1: Use `sx_downgrade` for a Config Refresh

The `myfirst_sysctl_debug_level` handler currently drops the shared lock and reacquires the exclusive lock. An alternative is to acquire shared, attempt `sx_try_upgrade`, and `sx_downgrade` after the modification. Implement this variant. Compare the behaviour under contention. When does each pattern win?

12.11.2 Challenge 2: Implement a Drain Operation Using `cv_broadcast`

Add an `ioctl` or `sysctl` that “drains” the `cbuf`: blocks until `cb_used == 0`, then returns. The implementation should use `cv_wait_sig(&sc->room_cv, ...)` in a loop on the condition `cb_used > 0`. Verify that `cv_broadcast(&sc->room_cv)` after the drain wakes every waiter, not just one.

12.11.3 Challenge 3: A `dtrace` Script for `cv_wait` Latency

Write a `dtrace` script that produces a histogram of how long threads spend inside `cv_wait_sig` on each of `data_cv` and `room_cv`. Run it during `mp_stress`. What does the distribution look like? Where is the long tail?

12.11.4 Challenge 4: Replace `cv` With Anonymous Channels

Re-implement the data and room conditions using `mtx_sleep` and wakeup on anonymous channels (a regression to the Chapter 11 design). Run the tests. The driver should still work, but the `procstat -kk` output and the `dtrace` queries become less informative. Describe the readability difference.

12.11.5 Challenge 5: Add Per-Descriptor `read_timeout_ms`

The `read_timeout_ms` `sysctl` is per-device. Add a per-descriptor timeout via an `ioctl(2)`: `MY-FIRST_SET_READ_TIMEOUT(int ms)` on a file descriptor sets that descriptor’s timeout. The driver code becomes more interesting because the timeout now lives in `struct myfirst_fh` rather than `struct myfirst_softc`. Beware: the per-fh state is not shared with other descriptors (no lock needed for the field itself), but the choice of timeout still affects the wait helper.

12.11.6 Challenge 6: Use `rw(9)` Instead of `sx(9)`

Replace `sx_init` with `rw_init`, `sx_xlock` with `rw_wlock`, and so on. Run the tests. What breaks? (Hint: the `cfg` path may include a sleeping operation; `rw` is not sleepable.) What does the failure look like? When would `rw(9)` be the right choice?

12.11.7 Challenge 7: Implement a Multi-CV Drain

The driver has two `cvs`. Suppose detach should be considered complete only when both `cvs` have zero waiters. Implement a check in detach that loops until `data_cv.cv_waiters == 0` and `room_cv.cv_waiters == 0`, sleeping briefly between checks. (Note: accessing `cv_waiters` directly from outside the `cv` API is non-portable; this is an exercise in understanding the internal state. Real production code should use a different mechanism.)

12.11.8 Challenge 8: Lock-Order Visualization

Use `dtrace` or `lockstat` to produce a graph of lock acquisitions during `mp_stress`. The nodes are locks; the edges are “holder of A acquired B while still holding A”. Compare the graph to your `LOCKING.md` lock order. Are there acquisitions you did not anticipate?

12.11.9 Challenge 9: Sleep-Channel Comparison

Build two versions of the driver: one using `cv` (the Chapter 12 default) and one using legacy `mtx_sleep/wakeup` on anonymous channels (the Chapter 11 default). Run identical workloads on both. Measure: maximum throughput, latency at the 99th percentile, WITNESS cleanliness, and the readability of the source. Write a one-page report.

12.11.10 Challenge 10: Bound Configuration Writes

The Chapter 12 driver allows configuration writes at any time. Add a `sysctl cfg_write_cooldown_ms` that limits how often a configuration change can occur (e.g., at most one write per 100 ms). Implement this with a timestamp field in the `cfg` struct and a check in each `cfg` `sysctl` handler. Decide what to do when the cooldown is violated: return `EBUSY`, queue the change, or silently coalesce. Document the choice.

12.12 Troubleshooting

This reference catalogues the bugs you are most likely to encounter while working through Chapter 12.

12.12.1 Symptom: Reader hangs forever despite data being written

Cause. A lost wakeup. The writer added bytes but did not signal `data_cv`, or the signal targeted the wrong `cv`.

Fix. Search the source for every place that adds bytes; ensure `cv_signal(&sc->data_cv)` is called. Confirm the `cv` is the one waiters are blocked on.

12.12.2 Symptom: WITNESS warns “lock order reversal” between `sc->mtx` and `sc->cfg_sx`

Cause. A path took the locks in the wrong order. The canonical order is `mtx` first, `sx` second.

Fix. Find the offending path (the warning names the lines). Either reorder the acquisitions to match the canonical order, or refactor the path to avoid holding both locks simultaneously (snapshot-and-apply).

12.12.3 Symptom: `cv_timedwait_sig` returns `EWOULDBLOCK` immediately

Cause. The timeout in ticks was zero or negative. Most likely the conversion from milliseconds to ticks rounded down to zero.

Fix. Use the $(\text{timo\_ms} * \text{hz} + 999) / 1000$ formula to round up to at least one tick. Verify `hz` is the expected value (typically 1000 on FreeBSD 14.3).

12.12.4 Symptom: detach hangs

Cause. A thread is sleeping on a cv that has not been broadcast, or the detach is waiting for `active_fhs > 0` to drop and a descriptor is open.

Fix. Confirm that detach broadcasts both cvs before the `active_fhs` check. Use `fstat | grep myfirst` from a separate terminal to find any process holding the device open; kill it.

12.12.5 Symptom: sysctl write hangs

Cause. The sysctl handler is waiting for a lock that is held by a thread doing something blocking. Most commonly, the `cfg sx` is held in exclusive mode by a slow `sysctl_handle_string`.

Fix. Verify the sysctl handler follows the snapshot-and-apply pattern: acquire shared, read, release; then `sysctl_handle_*` outside the lock; then exclusive lock to commit. Holding the lock across `sysctl_handle_*` is the bug.

12.12.6 Symptom: sx_destroy panics with “lock still held”

Cause. `sx_destroy` was called while another thread still held the lock or was waiting for it.

Fix. Confirm that detach refuses to proceed while `active_fhs > 0`. Confirm that no kernel thread or callout is using the `cfg sx` after detach starts.

12.12.7 Symptom: cv_signal or cv_broadcast wakes nothing visible

Cause. No one was waiting on the cv at the moment of the signal. Both `cv_signal` and `cv_broadcast` are no-ops when the wait queue is empty, and a `dtrace` probe on the wake side sees no follow-on activity.

Fix. None needed; the empty wake is correct and harmless. If you expected a waiter and there was none, the bug is upstream: either the waiter never reached `cv_wait_sig`, or the waker is targeting the wrong cv. Confirm via `dtrace` that the signal is firing on the cv you intend, and `procstat -kk` against the waiter to confirm where it is sleeping.

12.12.8 Symptom: read_timeout_ms set to 100 produces 200 ms latency

Cause. The kernel’s `hz` value is lower than expected. The +999 round-up means a 100 ms timeout at `hz=100` becomes 10 ticks (100 ms), but if `hz=10` it becomes 1 tick (100 ms). Different rounding.

Fix. Confirm `hz` with `sysctl kern.clockrate`. For tighter timeouts, use `cv_timedwait_sig_sbt` directly with `SBT_1MS * timo_ms` to avoid the tick rounding.

12.12.9 Symptom: A deliberately-broken lock order does not produce a WITNESS warning

Cause. Either the path with the bug is not exercised by the test, or `WITNESS` is not enabled in the running kernel.

Fix. Confirm `sysctl debug.witness.watch` returns nonzero. Confirm the offending path runs (add a `device_printf` to verify). Run the test under `mp_stress` to maximize the chance of the bug surfacing.

12.12.10 Symptom: lockstat shows enormous wait times on the data mutex

Cause. The mutex is being held across a long operation. Common offenders: `uiomove` accidentally inside the critical section; a debug `device_printf` that prints a large string while holding the lock.

Fix. Audit the critical sections. Move long operations outside. The mutex should be held for tens of nanoseconds, not microseconds.

12.12.11 Symptom: mp_stress reports byte-count mismatch after Chapter 12 changes

Cause. A wakeup was missed during the `cv` refactor. A reader started waiting after a writer's signal had already been delivered (no waiter at signal time, signal lost).

Fix. Verify that the wait helpers use `while`, not `if`, around `cv_wait_sig`. Verify that the signal happens after the state change, not before.

12.12.12 Symptom: timeout_tester shows latency longer than the configured timeout

Cause. Scheduler latency. The kernel scheduled the thread some milliseconds after the timer fired. This is normal; expect a few ms of jitter.

Fix. None for typical workloads. For real-time workloads, raise the thread's priority via `rtprio(2)`.

12.12.13 Symptom: kldunload reports busy when no descriptor is open

Cause. A taskqueue or background thread is still using a primitive in the driver. (Should not happen for our chapter, but worth knowing.)

Fix. Audit any taskqueue-, callout-, or kthread-spawning code. The detach must drain or terminate all of them before declaring it safe to unload.

12.12.14 Symptom: cv_wait_sig wakes immediately and returns 0

Cause. A signal arrived while the wait was being set up, or the `cv` was signalled by a thread that ran just before the wait was issued. Not actually a bug; the `while` loop is supposed to handle it.

Fix. Confirm that the surrounding `while (!condition)` re-checks. The loop turns the spurious-looking wakeup into a no-op: re-check, find condition false, sleep again.

12.12.15 Symptom: Two threads waiting on the same cv get woken in unexpected order

Cause. `cv_signal` wakes one waiter chosen by sleep-queue policy (highest priority, FIFO among equal). It does not wake them in arrival order if their priorities differ.

Fix. None usually needed; the kernel's choice is correct. If you require strict arrival-order wake-up, use a different design (per-waiter `cv`, or an explicit queue).

12.12.16 Symptom: `sx_xlock` under heavy reader load takes seconds to acquire

Cause. Many shared holders, each releasing slowly because the `cfg sx` is being acquired and released on every I/O. The writer is starved by the constant trickle of readers.

Fix. The kernel uses the `SX_LOCK_WRITE_SPINNER` flag to give writers priority once they begin waiting; the starvation is bounded but can still produce visible latency. If the latency is unacceptable, redesign so writers happen during quiescent windows or under a different protocol.

12.12.17 Symptom: Tests pass on a non-WITNESS kernel but fail on WITNESS

Cause. Almost always a real bug that WITNESS has detected. The most common: a lock acquired in a path that violates the global order, but the deadlock has not yet manifested because the contending workload has not been hit.

Fix. Read the WITNESS warning carefully. The warning text includes the source location of every violation. Fix the violation; the test should then pass on both kernels.

12.12.18 Symptom: Locking macros expand to nothing on the non-debug kernel

Cause. This is by design. `mtx_assert`, `sx_assert`, `KASSERT`, and `MYFIRST_ASSERT(sc)` (which expands to `mtx_assert`) compile to nothing without `INVARIANTS`. The asserts are free at production time and informative at development time.

Fix. None needed. Confirm that your test kernel has `INVARIANTS` enabled and the asserts will fire when violated.

12.12.19 Symptom: Sysctl handler blocks the entire system

Cause. A sysctl handler that holds a lock across a slow operation can effectively serialize every other operation that needs the same lock. If the lock is the device's main mutex, every I/O is blocked until the sysctl returns.

Fix. Sysctl handlers should follow the same discipline as I/O handlers: hold the lock for the minimum time, release before any potentially-slow operation. The snapshot-and-apply pattern works equally well here.

12.12.20 Symptom: A reader gets EAGAIN even with `read_timeout_ms=0`

Cause. The read returned `EAGAIN` because of `O_NONBLOCK` (the file descriptor was opened non-blocking, or `fcntl(2)` set `O_NONBLOCK` on it). The driver's `IO_NDELAY` check returns `EAGAIN` regardless of the timeout sysctl.

Fix. Confirm the descriptor is blocking: `fcntl(fd, F_GETFL)` should return a value without the `O_NONBLOCK` bit set. If non-blocking is intended, `EAGAIN` is the correct response.

12.12.21 Symptom: kldunload after a successful test still hangs briefly

Cause. The detach path is waiting for in-flight handlers to return. Each waiter that was sleeping on a cv must wake (because of the broadcast), reacquire the mutex, see `!is_attached`, return, and exit the kernel. This takes a few milliseconds for several waiters.

Fix. None usually needed; a few-millisecond delay is normal. If the delay is longer, check that every waiter does have a post-sleep `is_attached` check.

12.12.22 Symptom: Two separate driver instances both report WITNESS warnings about the same lock name

Cause. Both instances initialize their locks with the same name (`myfirst0` for both, for example). WITNESS treats locks of the same name as the same logical lock and may warn about duplicate acquisition or invented order issues across instances.

Fix. Initialize each instance's lock with a unique name that includes the unit number, for example via `device_get_nameunit(dev)` which yields `myfirst0`, `myfirst1`, etc. Our chapter already does this for the device mutex; do the same for cvs and sx.

12.12.23 Symptom: A cv with many waiters takes a long time to broadcast

Cause. `cv_broadcast` walks the wait queue, marking each waiter as runnable. The walk is $O(n)$ in the number of waiters. With hundreds of waiters this becomes a measurable cost.

Fix. The broadcast itself is rarely a bottleneck for normal workloads; it is the subsequent thundering-herd contention as every awakened thread tries to acquire the interlock that causes visible pause. If your driver routinely has hundreds of waiters on the same cv, reconsider the design; per-waiter cvs or a queue-based approach may scale better.

12.13 Wrapping Up

Chapter 12 took the driver you built in Chapter 11 and gave it a richer synchronization vocabulary. The single mutex from Chapter 11 is still there, doing the same job, with the same rules. Around it now sit two named condition variables that replace the anonymous wakeup channel, an sx lock that protects a small but real configuration subsystem, and a bounded-wait capability that lets the read path return promptly when the user expects it to. The lock order between the two lock classes is documented, enforced by WITNESS, and verified by a stress kit that runs the data path and the configuration path concurrently.

We learned to think about synchronization as a vocabulary, not just a mechanism. A condition variable says *I am waiting for a specific change; tell me when it happens*. A shared lock says *I am reading; do not let a writer in*. A timed wait says *and please give up if it takes too long*. Each primitive is a different shape of agreement between threads, and using the right shape for each agreement produces code that reads like the design rather than fights against it.

We also learned to debug synchronization carefully. The six failure shapes (lost wakeup, spurious wakeup, lock-order reversal, premature destroy, sleeping with a non-sleepable lock, race between detach and active operation)

cover almost every bug you will hit in practice. `WITNESS` catches the ones the kernel can detect at runtime; the in-kernel debugger lets you inspect a hung system; `lockstat(1)` and `dtrace(1)` give you observability without modifying the source.

We finished with a refactor pass. The driver now has a documented lock order, a clean `LOCKING.md`, a bumped version string, an updated changelog, and a regression test that verifies every primitive on every supported workload. That infrastructure scales: when Chapter 13 adds timers and Chapter 14 adds interrupts, the documentation pattern absorbs the new primitives without growing brittle.

12.13.1 What You Should Now Be Able to Do

A short self-checklist of capabilities you should now have, before turning to Chapter 13:

- Look at a piece of shared state in any driver and choose the right primitive (atomic, mutex, sx, rw, cv) by walking the decision tree.
- Replace any anonymous wakeup channel in any driver with a named cv, and explain why the change is an improvement.
- Add a bounded blocking primitive to any wait path, and explain when to use `EAGAIN`, `EINTR`, `ERESTART`, or `EWOULDBLOCK`.
- Design a multi-reader, single-writer subsystem with documented lock order.
- Read a `WITNESS` warning and identify the offending lock pair from the source location alone.
- Diagnose a hung system in DDB using `show all locks` and `show sleepchain`.
- Run a composite stress workload and measure lock contention with `lockstat(1)`.
- Write a `LOCKING.md` document that another developer can use as authoritative reference.

If any of those feels uncertain, the labs in Chapter 12 are the place to build the muscle memory. None requires more than a couple of hours; together they cover every primitive and every pattern the chapter introduced.

12.13.2 Three Closing Reminders

The first is to *run the composite stress before you commit*. The composite kit catches the cross-primitive bugs that single-axis tests miss. Thirty minutes on a debug kernel is a small investment for the confidence it produces.

The second is to *keep the lock order honest*. Every new lock you introduce starts a new question: where in the order does it sit? Answer the question explicitly in `LOCKING.md` before you write the code. The cost of getting the answer wrong scales with the size of the driver; the cost of writing it down at the start is one minute.

The third is to *trust the primitives and use the right one*. The kernel's mutex, cv, sx, and rw locks are the result of decades of engineering. The temptation to roll your own coordination using flags and atomic flags is real and almost always misguided. Pick the primitive that names what you are trying to say. The code will be shorter, clearer, and provably more correct.

12.14 Reference: The Driver's Stage Progression

Chapter 12 evolves the driver in four discrete stages, each of which is its own directory under `examples/part-03/ch12-synchronization-mechanisms/`. The progression mirrors the chapter's narrative; it lets a reader build the driver one primitive at a time and see what each addition contributes.

12.14.1 Stage 1: cv-channels

Replaces the anonymous `&sc->cb` wakeup channel with two named condition variables (`data_cv`, `room_cv`). The wait helpers use `cv_wait_sig` instead of `mtx_sleep`. The signallers use `cv_signal` (or `cv_broadcast` in detach) on the cv that matches the state change.

What changes: the sleep/wake mechanism. The driver behaves identically from user space.

What you can verify: `procstat -kk` shows the cv name (`myfirst_data` or `myfirst_room`) instead of the `wmesg` (`myfrd`). `dtrace` can attach to specific cvs. Throughput is slightly higher because per-event signalling avoids waking unrelated waiters.

12.14.2 Stage 2: bounded-read

Adds a `read_timeout_ms` sysctl that bounds blocking reads via `cv_timedwait_sig`. A symmetric `write_timeout_ms` is also possible.

What changes: the read path can now return `EAGAIN` after a configurable timeout. The default of zero preserves the indefinite-wait behaviour from Stage 1.

What you can verify: `timeout_tester` reports `EAGAIN` after approximately the configured timeout. Setting the timeout to zero restores indefinite waits. Ctrl-C still works either way.

12.14.3 Stage 3: sx-config

Adds a `cfg` struct to the `softc`, protected by an `sx_lock` (`cfg_sx`). Three configuration fields (`debug_level`, `soft_byte_limit`, `nickname`) are exposed as sysctls. The data path consults `debug_level` for log emission and `soft_byte_limit` for write rejection.

What changes: the driver gains a configuration interface. The `MYFIRST_DBG` macro consults the current debug level. Writes that would exceed the soft limit return `EAGAIN`.

What you can verify: `sysctl -w dev.myfirst.0.debug_level=2` produces visible debug messages. Setting `soft_byte_limit` causes writes to start failing once the buffer reaches the limit. `WITNESS` reports the lock order (mtx first, sx second) and is silent under stress.

12.14.4 Stage 4: final

The combined version with all three primitives, plus a `LOCKING.md` update, the version bump to `0.6-sync`, and the new `myfirst_sysctl_reset` that exercises both locks together.

What changes: integration. No new primitives.

What you can verify: the regression suite passes; the composite stress workload runs cleanly for at least 30 minutes; `clang --analyze` is silent.

This four-stage progression is the canonical Chapter 12 driver. The companion examples mirror the stages exactly so a reader can compile and load any one of them.

12.15 Reference: Migrating From `mtx_sleep` to `cv`

If you are working on an existing driver that uses the legacy `mtx_sleep/wakeup` channel mechanism, the migration to `cv(9)` is mechanical. A short recipe.

A note before starting: the legacy mechanism is not deprecated and is still widely used in the FreeBSD tree. Many drivers will keep `mtx_sleep` indefinitely and that is perfectly correct. The migration is worth doing when you have multiple distinct conditions sharing a single channel (the thundering-herd case), or when you want the `procstat` and `dtrace` visibility that named `cvs` provide. For a driver with a single condition and a single channel, the migration is purely cosmetic; do it for readability if you want, skip it if you do not.

12.15.1 Step 1: Identify Each Logical Wait Channel

Read the source. Find every `mtx_sleep` call. For each, ask: what is the condition this thread is waiting for?

In the Chapter 11 driver, there were two logical conditions both using `&sc->cb`:

- `myfirst_wait_data`: waiting for `cbuf_used > 0`.
- `myfirst_wait_room`: waiting for `cbuf_free > 0`.

Two conditions; one channel. The migration assigns each its own `cv`.

12.15.2 Step 2: Add `cv` Fields to the Softc

For each logical condition, add a `struct cv` field. Pick a descriptive name:

```
struct cv  data_cv;  
struct cv  room_cv;
```

Initialize in `attach (cv_init)` and destroy in `detach (cv_destroy)`.

12.15.3 Step 3: Replace `mtx_sleep` with `cv_wait_sig`

For each `mtx_sleep` call, replace with `cv_wait_sig` (or `cv_timedwait_sig`):

```
/* Before: */  
error = mtx_sleep(&sc->cb, &sc->mtx, PCATCH, "myfrd", 0);  
  
/* After: */  
error = cv_wait_sig(&sc->data_cv, &sc->mtx);
```

The `wmesg` argument is gone (the `cv`'s description string takes its place). The `PCATCH` is implicit in the `_sig` suffix. The interlock argument is the same.

12.15.4 Step 4: Replace wakeup with cv_signal or cv_broadcast

For each `wakeup(&channel)` call, decide whether one waiter or all waiters should be woken. Replace with `cv_signal` or `cv_broadcast`:

```
/* Before: */
wakeup(&sc->cb); /* both readers and writers were on this channel */

/* After: */
if (write_succeeded)
    cv_signal(&sc->data_cv); /* only readers care about new data */
if (read_succeeded)
    cv_signal(&sc->room_cv); /* only writers care about new room */
```

This is also the moment to add the per-event correspondence the cv API encourages: signal only when the state actually changed.

12.15.5 Step 5: Update the Detach Path

Detach was waking the channel before destroying state:

```
/* Before: */
sc->is_attached = 0;
wakeup(&sc->cb);

/* After: */
sc->is_attached = 0;
cv_broadcast(&sc->data_cv);
cv_broadcast(&sc->room_cv);
/* later, after all waiters have exited: */
cv_destroy(&sc->data_cv);
cv_destroy(&sc->room_cv);
```

`cv_broadcast` ensures every waiter wakes; the post-wait `is_attached` check returns `ENXIO` for each of them.

12.15.6 Step 6: Update LOCKING.md

Document each new cv: its name, its condition, its interlock, its signaller(s), its waiter(s). The Chapter 12 driver's `LOCKING.md` is the template.

12.15.7 Step 7: Re-run the Stress Kit

The migration should not change observable behaviour; only the internal mechanism. Run the existing tests; they should pass. Run them under `WITNESS`; no new warnings should appear.

The migration in our chapter is a few hundred lines of source; the recipe above scales to drivers of any size. The benefit is that every wait is now self-documenting and every signal is targeted.

12.15.8 When Migration Is and Is Not Worth It

The migration costs a few hours of refactor effort, a careful re-run of the test suite, and an update to the documentation. The benefits are:

- Each wait acquires a name visible in `procstat`, `dtrace`, and source.
- Wakeups become per-condition; the thundering herd shrinks.
- Wakeup-channel mismatches are easier to spot in the source.

For a small driver with a single waiting condition, the costs and benefits roughly cancel; the legacy mechanism is fine. For a driver with two or more distinct conditions, the migration almost always pays off. For a driver maintained by multiple developers, the readability win is large.

12.16 Reference: Pre-Production Audit Checklist

A short audit to perform before promoting a synchronization-heavy driver from development to production. Each item is a question; each item should be answerable with confidence.

12.16.1 Lock Inventory

- ☐ Have I listed every lock the driver owns in `LOCKING.md`?
- ☐ For each lock, have I named what it protects?
- ☐ For each lock, have I named the contexts in which it may be acquired?
- ☐ For each lock, have I documented its lifetime (created where, destroyed where)?

12.16.2 Lock Order

- ☐ Is the global lock order documented in `LOCKING.md`?
- ☐ Does every code path that holds two locks follow the global order?
- ☐ Have I run `WITNESS` for at least 30 minutes under stress and observed no order reversals?
- ☐ If the driver has more than one instance, have I confirmed that intra-instance ordering is consistent with inter-instance ordering?

12.16.3 cv Inventory

- ☐ Have I listed every cv the driver owns?
- ☐ For each cv, have I named the condition it represents?
- ☐ For each cv, have I named the interlock mutex?
- ☐ For each cv, have I confirmed at least one signaller and at least one waiter?
- ☐ For each cv, have I confirmed `cv_broadcast` is called in detach before `cv_destroy`?

12.16.4 Wait Helpers

- ☐ Is every `cv_wait` (or variant) inside a `while (!condition)` loop?
- ☐ Does every wait helper re-check `is_attached` after the wait?
- ☐ Does every wait helper return a sensible error (`ENXIO`, `EINTR`, `EAGAIN`)?

12.16.5 Signal Sites

- ☐ Does every state change that should wake waiters have a corresponding `cv_signal` or `cv_broadcast`?
- ☐ Is `cv_signal` used when only one waiter needs to wake; `cv_broadcast` only when all do?
- ☐ Are signal sites guarded by `if (state_changed)` so empty signals are skipped?

12.16.6 Detach Path

- ☐ Does detach refuse to proceed while `active_fhs > 0`?
- ☐ Does detach clear `is_attached` under the device mutex?
- ☐ Does detach broadcast every `cv` before destroying?
- ☐ Are primitives destroyed in reverse acquisition order (innermost lock destroyed first)?

12.16.7 Static Analysis

- ☐ Has `clang --analyze` been run; new warnings triaged?
- ☐ Has `WARNS=6` build produced no warnings?
- ☐ Has the regression suite been run on a `WITNESS` kernel; all tests pass?

12.16.8 Documentation

- ☐ Is `LOCKING.md` up to date with the code?
- ☐ Is the version string in the source bumped?
- ☐ Is the `CHANGELOG.md` updated?
- ☐ Does the `README.md` describe the new features and their sysctls?

A driver that passes this audit is a driver you can trust under load.

12.17 Reference: Sleep Channel Hygiene

Both the legacy `mtx_sleep/wakeup` channel mechanism and the modern `cv(9)` API rely on a *channel* that identifies which waiters a wakeup affects. The channel is a key into the kernel's sleep-queue hash table. Mistakes around channels are the source of several common bugs.

A few rules of hygiene.

12.17.1 One Channel Per Logical Condition

If your driver has two distinct conditions that block (say, “data available” and “room available”), use two distinct channels. Sharing a single channel forces every wakeup to wake all waiters; some of them will go back to sleep immediately because their condition is still false. The performance cost is real; the readability cost is also real.

In our chapter, this rule manifests as `data_cv` and `room_cv` being separate cvs. The Chapter 11 driver used a shared anonymous channel `&sc->cb` and paid the thundering-herd cost; Chapter 12’s split is the cure.

12.17.2 Channel Pointers Must Be Stable

A channel is an address. The kernel does not interpret it; it uses it as a hash key. The address must not change between the wait and the signal. This usually happens automatically (the address of a `softc` field is stable for the lifetime of the `softc`), but be careful about temporary buffers, stack-allocated structures, or freed memory.

If you see a wait that hangs after a particular code path, suspect a channel-pointer mismatch. The signaller and the waiter must be using the same address.

12.17.3 Channel Pointers Must Be Unique to Their Purpose

If the same address is used for two different purposes (say, the data-available channel and a “completion” channel), wakeups for one purpose may unintentionally wake waiters for the other. Use a different field of the `softc` as the channel for each purpose, or use a `cv` (which has a name and is a separate object).

12.17.4 Wakers Should Hold the Interlock When the State Could Change

Although `wakeup` and `cv_signal` are not strictly required to be called under the interlock, doing so closes a race window in which a waiter checks the condition (false), the state changes, the wakeup fires (no waiters in the queue), and the waiter then enqueues and sleeps forever. Holding the interlock while signalling is the safe default; relax only when you can prove the state cannot revert.

Our Chapter 12 design signals after dropping the mutex, which is safe because the `cv`’s own enqueue-under-interlock contract closes the race for `cv` (not for wakeup). For wakeup, signal under the mutex.

12.17.5 A Signal With No Waiter Is Free

`cv_signal` and `wakeup` on a channel with no waiters do nothing. There is no penalty for an unnecessary signal; the cost is essentially the cost of taking and releasing the `cv`-queue spin lock. Do not avoid signals out of optimization fear; signal when the state changes, even if it sometimes signals nothing.

12.17.6 A Wait With No Signaller Is a Bug

A wait that is never signalled is a hang. Ensure that every wait has at least one matching signal site, and that the matching site is reached in every code path that produces the awaited state change.

This is the most common `cv` bug. The audit checklist asks the question; the discipline of asking it during code review catches most cases.

12.18 Reference: Common cv Idioms

Quick-lookup collection for the cv patterns you will use most often.

12.18.1 Wait for a Condition

```
mtx_lock(&mtx);
while (!condition)
    cv_wait_sig(&cv, &mtx);
/* condition is true; do work */
mtx_unlock(&mtx);
```

The while loop is essential. Spurious wakeups are permitted; signals interrupt the wait; both look like a return from `cv_wait_sig`. Re-check the condition after every return.

12.18.2 Signal One Waiter

```
mtx_lock(&mtx);
/* change state */
mtx_unlock(&mtx);
cv_signal(&cv);
```

`cv_signal` after the unlock saves a context switch (the woken thread does not contend immediately for the mutex). Acceptable when the state change is unambiguous and no concurrent path can revert it.

12.18.3 Broadcast a State Change

```
mtx_lock(&mtx);
state_changed_globally = true;
cv_broadcast(&cv);
mtx_unlock(&mtx);
```

Use `cv_broadcast` when every waiter needs to know about the change. Detach paths and configuration resets are typical examples.

12.18.4 Wait With a Timeout

```
mtx_lock(&mtx);
while (!condition) {
    int ticks = (ms * hz + 999) / 1000;
    int err = cv_timedwait_sig(&cv, &mtx, ticks);
    if (err == EWOULDBLOCK) {
        mtx_unlock(&mtx);
        return (EAGAIN);
    }
    if (err != 0) {
        mtx_unlock(&mtx);
        return (err);
    }
}
/* do work */
mtx_unlock(&mtx);
```

Convert milliseconds to ticks, round up, handle the three return cases (timeout, signal, normal wakeup) explicitly.

12.18.5 Wait With Detach Awareness

```
while (!condition) {
    int err = cv_wait_sig(&cv, &mtx);
    if (err != 0)
        return (err);
    if (!sc->is_attached)
        return (ENXIO);
}
```

The post-sleep `is_attached` check ensures we exit cleanly if the device was detached while we slept. `cv_broadcast` in the detach path makes this work.

12.18.6 Drain Waiters Before Destroy

```
mtx_lock(&mtx);
sc->is_attached = 0;
cv_broadcast(&cv);
mtx_unlock(&mtx);
/* waiters wake, see !is_attached, return ENXIO, exit */
/* by the time active_fhs == 0, no waiters remain */
cv_destroy(&cv);
```

The combination of broadcast and the `is_attached` re-check guarantees no waiter remains in the cv at destroy time.

12.19 Reference: Common sx Idioms

12.19.1 Read-Mostly Field

```
sx_slock(&sx);  
value = field;  
sx_sunlock(&sx);
```

Cheap on a multi-core system; multiple readers do not contend.

12.19.2 Update With Validation

```
sx_slock(&sx);  
old = field;  
sx_sunlock(&sx);  
  
/* validate possibly-new value with sysctl_handle_*, etc. */  
  
sx_xlock(&sx);  
field = new;  
sx_xunlock(&sx);
```

Two acquire-release cycles. The shared lock for the read; the exclusive lock for the write. Drop between them so the validation does not hold either lock.

12.19.3 Snapshot-and-Apply Across Two Locks

```
sx_slock(&cfg_sx);  
local = cfg.value;  
sx_sunlock(&cfg_sx);  
  
mtx_lock(&data_mtx);  
/* use local without holding either lock together */  
mtx_unlock(&data_mtx);
```

Avoids holding both locks simultaneously; relaxes the lock-order constraint.

12.19.4 Try-Upgrade Pattern

```

sx_slock(&sx);
if (need_modify) {
    if (sx_try_upgrade(&sx)) {
        /* exclusive */
        modify();
        sx_downgrade(&sx);
    } else {
        /* drop, reacquire as exclusive, re-validate */
        sx_sunlock(&sx);
        sx_xlock(&sx);
        if (still_need_modify())
            modify();
        sx_downgrade(&sx);
    }
}
sx_sunlock(&sx);

```

Optimistic upgrade. The fallback path must re-validate, because the world changed during the unlock window.

12.19.5 Assert Held

```

sx_assert(&sx, SA_SLOCKED); /* shared */
sx_assert(&sx, SA_XLOCKED); /* exclusive */
sx_assert(&sx, SA_LOCKED);  /* either */

```

Use at the start of helpers that expect a particular lock state.

12.20 Reference: Decision Table for Synchronization Primitives

A compact lookup table.

If you need to...	Use
Update a single word atomically	<code>atomic(9)</code>
Update a per-CPU counter cheaply	<code>counter(9)</code>
Protect composite state in process context	<code>mtx(9)</code> (MTX_DEF)
Protect composite state in interrupt context	<code>mtx(9)</code> (MTX_SPIN)
Protect read-mostly state in process context	<code>sx(9)</code>

If you need to...	Use
Protect read-mostly state where sleep is forbidden	<code>rw(9)</code>
Wait for a specific condition to become true	<code>cv(9)</code> (paired with <code>mtx(9)</code> or <code>sx(9)</code>)
Wait until a deadline	<code>cv_timedwait_sig</code> , or <code>mtx_sleep</code> with <code>timo > 0</code>
Wait such that Ctrl-C can interrupt	The <code>_sig</code> variant of any wait primitive
Run code at a specific time in the future	<code>callout(9)</code> (Chapter 13)
Defer work to a worker thread	<code>taskqueue(9)</code> (Chapter 16)
Read concurrently with no synchronization at all	<code>epoch(9)</code> (later chapters)

If two primitives both fit, use the simpler one.

12.21 Reference: Reading `kern_condvar.c` and `kern_sx.c`

Two files in `/usr/src/sys/kern/` are worth opening once you have used the `cv` and `sx` APIs in your driver.

`/usr/src/sys/kern/kern_condvar.c` is the `cv` implementation. The functions worth seeing:

- `cv_init`: initialization. Trivial.
- `_cv_wait` and `_cv_wait_sig`: the core blocking primitives. Each takes the `cv`-queue spin lock, increments the waiter count, drops the interlock, hands the thread to the sleep queue, yields, and on return reacquires the interlock. The atomicity of “drop interlock, sleep” is provided by the sleep queue layer.
- `_cv_timedwait_sbt` and `_cv_timedwait_sig_sbt`: the timed variants. Same shape, with a callout that wakes the thread if the timeout fires first.
- `cv_signal`: takes the `cv`-queue spin lock, signals one waiter via `sleepq_signal`.
- `cv_broadcastpri`: signals all waiters at the given priority.

The whole file is about 400 lines. An afternoon of reading is more than enough to understand it end to end.

`/usr/src/sys/kern/kern_sx.c` is the `sx` implementation. Larger and denser, because the lock supports both shared and exclusive modes with full priority propagation. The functions worth seeing:

- `sx_init_flags`: initialization. Sets the initial state, registers with `WITNESS`.
- `_sx_xlock_hard` and `_sx_xunlock_hard`: the slow paths for exclusive operations. The fast paths inline in `sx.h`.
- `_sx_slock_int` and `_sx_sunlock_int`: the shared-mode operations. The shared count is incremented via atomic compare-and-swap; if the lock is held exclusively, the thread blocks.
- `sx_try_upgrade_int` and `sx_downgrade_int`: the mode-change operations.

Skim. The internals are intricate, but the public-facing API behaves as documented and the source confirms it.

12.22 Reference: Common Mistakes With cv and sx

Each new primitive comes with a set of mistakes that beginners make until they have been bitten. A short catalogue.

12.22.1 cv Mistakes

Using if instead of while around cv_wait. The condition may not be true on return because of a spurious wakeup. Always loop.

Forgetting to broadcast in detach. Waiters never wake, the cv has lingering waiters at destroy time, the kernel may panic. Always cv_broadcast before cv_destroy.

Signalling the wrong cv. Waking up readers when you meant writers (or vice versa). Easy mistake when refactoring. The cv's name is your defense; if cv_signal(&sc->room_cv) does not feel right at the call site, it probably is not right.

Signalling without the interlock when the state could revert. If two threads can both modify the state, one of them must hold the interlock when signalling, or a wakeup may be lost. Default to signalling under the interlock; relax only when you can prove the state cannot revert.

Missing the post-wait check for detach. A waiter that wakes up because of a cv_broadcast from detach must re-check is_attached and return ENXIO. If the check is missing, the waiter proceeds as if the device is still alive and crashes.

Calling cv_wait while holding multiple locks. Only the interlock is dropped during the sleep. Other locks remain held. If those locks are needed by the waker, you have a deadlock. Drop other locks first.

12.22.2 sx Mistakes

Holding the sx across a sleeping call. Drop before sysctl_handle_*, uiomove, or malloc(M_WAITOK). The sx is sleepable, so the kernel will not panic, but other waiters will be blocked for the duration.

Acquiring shared and then xlock without dropping shared. sx_xlock while holding the same sx in shared mode is a deadlock; the call will block forever waiting for itself. Use sx_try_upgrade or drop and reacquire.

Forgetting that sx is sleepable. Calling sx_xlock from a context where sleeping is illegal (interrupt context, inside a spin lock) panics. Use rw(9) for those contexts.

Holding the sx in shared mode across a long operation. Other readers can proceed, but the sx writer is blocked indefinitely. If the operation is long, drop the shared lock, do the work, and reacquire if you need to commit.

Releasing the wrong mode. sx_xunlock on a shared-mode lock is a bug; sx_sunlock on an exclusive-mode lock is a bug. Use sx_unlock (the polymorphic version) only when you do not know which mode you are in (rare).

12.22.3 Mistakes Specific to Combining Both

Acquiring in the wrong order. The Chapter 12 driver requires mtx first, sx second. Reverse order produces a WITNESS warning under load.

Releasing in the wrong order. Acquire `mtx`, acquire `sx`, release `mtx`, release `sx`. The release order *must* be the reverse of the acquisition order: release `sx` first, then release `mtx`. Otherwise an observer between the two releases sees an unexpected combination.

Snapshot-and-apply where staleness matters. The pattern is correct only when the snapshot can tolerate small staleness. For values that must be current (security flags, hard-quota limits), snapshot-and-apply is wrong; you must hold both locks atomically.

Forgetting to update `LOCKING.md`. Adding a lock or changing the order without updating the documentation produces drift. Three months later, no one remembers what the rule was. Update the document in the same commit.

12.23 Reference: Time Primitives

A brief tour of how the kernel expresses time. Useful when reading or writing the timed-wait variants.

The kernel has three commonly-used time representations:

- `int` ticks. The legacy unit. `hz` ticks equal one second. The default `hz` on FreeBSD 14.3 is 1000, so one tick is one millisecond. `mtx_sleep`, `cv_timedwait`, and `tsleep` all take their timeouts in ticks.
- `sbintime_t`. A 64-bit signed binary fixed-point representation: the upper 32 bits are seconds, the lower 32 bits are a fraction of a second. The unit constants are in `/usr/src/sys/sys/time.h`: `SBT_1S`, `SBT_1MS`, `SBT_1US`, `SBT_1NS`. The newer time API (`msleep_sbt`, `cv_timedwait_sbt`, `callout_reset_sbt`) uses `sbintime`.
- `struct timespec`. POSIX seconds-and-nanoseconds. Used at the user-space boundary; rarely needed in driver internals.

Conversion helpers in `time.h`:

- `tick_sbt`: a global variable holding $1 / \text{hz}$ as an `sbintime`, so `tick_sbt * timo_in_ticks` gives the equivalent `sbintime`.
- `nstosbt(ns)`, `ustosbt(us)`, `sbttous(sbt)`, `sbttons(sbt)`, `tstosbt(ts)`, `sbt tots(ts)`: explicit conversions between the various units.

The `_sbt` time API exists because `hz` granularity is too coarse for some uses. With `hz=1000`, the smallest expressible timeout is 1 ms, and timeouts are aligned to tick boundaries. With `sbintime`, you can express 100 microseconds and ask the kernel to schedule the wakeup as close to that as the hardware timer allows.

For Chapter 12, we use the tick-based API everywhere because the precision is sufficient. The reference is here so you know where to reach when sub-millisecond precision matters.

The `pr` argument to `_sbt` functions deserves a sentence. It is the *precision* the caller is willing to accept: how much wiggle the kernel may add for power-saving timer coalescing. A precision of `SBT_1S` means “I do not care if my 5-second timer fires up to 1 second late; if you can coalesce it with another timer to save power, please do”. A precision of `SBT_1NS` means “fire as close to the deadline as you can”. For driver code, 0 (no slop) or `SBT_1MS` (a millisecond of slop) are the typical values.

The `flags` argument controls how the timer is registered. `C_HARDCLOCK` is the most common: align to the system’s hardclock interrupt for predictable timing. `C_DIRECT_EXEC` runs the callout in the timer interrupt

rather than deferring it to a callout thread. `C_ABSOLUTE` interprets `sbt` as an absolute time rather than a relative timeout. We use `C_HARDCLOCK` everywhere in Chapter 12.

12.24 Reference: Common WITNESS Warnings, Decoded

WITNESS produces several kinds of warning. Each has a recognisable shape.

12.24.1 “lock order reversal”

The signature: two lines naming “1st” and “2nd” lock, plus an “established at” line. We have walked through the diagnosis in Section 6.

Common cause: a path that takes locks in an order that contradicts a previously-observed order. Fix by reordering or restructuring.

12.24.2 “duplicate lock of same name”

The signature: a warning about acquiring a lock with the same `lo_name` as one already held.

Common cause: two instances of the same driver, each with its own lock, both with the same name. WITNESS is conservative and assumes that two locks of the same type belong to the same class. Fix by initializing each lock with a unique name (e.g., include the unit number via `device_get_nameunit(dev)`), or by passing the appropriate per-class “duplicate-acquire OK” flag at init time: `MTX_DUPOK` for mutexes, `SX_DUPOK` for `sx`, `RW_DUPOK` for `rwlocks`. Each of those expands to the lock-object-level `LO_DUPOK` bit; you write the per-class name in driver code.

12.24.3 “sleeping thread (pid N) owns a non-sleepable lock”

The signature: a thread is in a sleeping primitive (`cv_wait`, `mtx_sleep`, `_sleep`) while holding a spin mutex or an `rw` lock.

Common cause: a function that takes a non-sleepable lock and then calls something that may sleep. Fix by dropping the non-sleepable lock first.

12.24.4 “exclusive sleep mutex foo not owned at”

The signature: a thread tried to release or assert a mutex it does not hold.

Common cause: the wrong mutex pointer, or an unlock without a matching lock on this code path. Fix by tracing the lock acquisition.

12.24.5 “lock list reversal”

The signature: similar to lock-order reversal but indicates a more complex inversion involving more than two locks.

Common cause: a chain of acquisitions that taken together violates the global order. Fix by simplifying the acquisition pattern; if the chain is truly necessary, consider whether the design should use fewer locks.

12.24.6 “sleepable acquired while holding non-sleepable”

The signature: a thread tried to acquire a sleepable lock (sx, mtx_def, lockmgr) while holding a non-sleepable one (mtx_spin, rw).

Common cause: confusion about the lock classes. Fix by switching the inner lock to a sleepable variant or by restructuring to avoid the nesting.

12.24.7 Acting on a Warning

When WITNESS fires, the temptation is to suppress the warning. Resist. The warning means the kernel has observed a real situation that violates a real rule. Suppression hides the bug; it does not fix it.

The right responses, in order of preference:

1. Fix the bug (reorder locks, drop a lock, restructure code).
2. Explain why the warning is incorrect for this case and use the appropriate `_DUPOK` flag with a comment in the source.
3. If you cannot do either, escalate. Ask on `freebsd-hackers` or open a PR. A WITNESS warning that nobody can explain is a real bug somewhere.

12.25 Reference: Lock Class Quick Reference

A compact lookup for the differences between the lock classes you have seen so far.

Property	mtx_def	mtx_spin	sx	rw	rmlock	lockmgr
Sleeps when contended	Yes	No (spins)	Yes	No (spins)	No (mostly)	Yes
Multiple holders	No	No	Yes (shared)	Yes (read)	Yes (read)	Yes (shared)
Holder may sleep	Yes	No	Yes	No	No (read)	Yes
Priority propagation	Yes	n/a	No	Yes	n/a	Yes
Signal interruptible	n/a	n/a	_sig	No	No	Yes
Recursion supported	Optional	Yes	Optional	No	No	Yes

Property	mtx_def	mtx_spin	sx	rw	rmlock	lockmgr
WITNESS-tracked	Yes	Yes	Yes	Yes	Yes	Yes
Best driver use	Default	Interrupt context	Read-mostly	Hot read paths	Very hot reads	Filesystems

`rmlock(9)` and `lockmgr(9)` are listed for completeness; this book covers `mtx`, `cv`, `sx`, and `rw` in depth, and treats the others as “known to exist, look up the manual page if needed”.

12.26 Reference: Multi-Primitive Driver Design Patterns

Three patterns recur in drivers that combine several synchronization primitives. Each is worth a sentence so you recognize them in the wild.

12.26.1 Pattern: One Mutex, One Configuration `sx`

The `myfirst` driver of Chapter 12 is this pattern. The mutex protects the data path and a `sx` protects the configuration. The lock order is mutex first, `sx` second. Most simple drivers fit this pattern.

When to use: the data path is process-context, has a composite invariant, and reads configuration occasionally. The configuration is read frequently and written rarely.

When not to use: when the data path runs in interrupt context (mutex must be `MTX_SPIN`, configuration must be `rw`), or when the data path itself has sub-paths that benefit from different locks.

12.26.2 Pattern: Per-Queue Lock With a Configuration Lock

A driver with multiple queues (one per CPU, one per consumer, one per stream) gives each queue its own lock and uses a separate `sx` for configuration. The lock order is per-queue lock first, configuration `sx` second. Lock-order between queues is not defined (you should never hold two queues at once).

When to use: high core counts, the workload partitions naturally per queue.

When not to use: the workload is symmetric and the per-queue locks would not help, or the data crosses queues frequently and would force order rules.

12.26.3 Pattern: Per-Object Lock With a Container Lock

A driver that maintains a list of objects (devices, sessions, descriptors) gives each object its own lock and uses a container lock to protect the list of objects. Walking the list takes the container lock; modifying an object takes that object’s lock; both can be held in the order container first, object second.

When to use: list operations and per-object operations both need protection, with different lifetimes.

When not to use: a single mutex would be enough (small lists, infrequent operations).

The `myfirst` driver does not need this pattern yet; future drivers in the book will.

12.26.4 Pattern: Per-CPU Counters With Mutex-Protected Tail

This is the Chapter 11 pattern, which Chapter 12 inherits. Hot counters (`bytes_read`, `bytes_written`) use `counter(9)` per-CPU storage. The `cbuf`, with its composite invariant, uses a single mutex. The two are independent; updates to the counters do not need the mutex; updates to the `cbuf` still do.

When to use: a high-frequency counter sits next to a structure with a composite invariant.

When not to use: the counter updates need to be consistent with the structure updates (then both need the same lock).

12.26.5 Pattern: Snapshot-and-Apply Across Two Lock Classes

Any time a path needs both lock classes, the snapshot-and-apply pattern reduces the lock-order constraint to a single direction. Read from one lock, drop, then take the other. The snapshot may be slightly stale; for advisory values, that is acceptable.

When to use: the value being snapshot is not strictly current-required; staleness on the order of microseconds is acceptable.

When not to use: the value is a security flag, a hard-budget limit, or anything where staleness could violate a contract.

The `myfirst_write` path uses this for the soft byte limit: snapshot the limit under the `cfg sx`, drop it, take the data mutex, check the limit against the current `cb_used`. The combined operation is not atomic, but it is correct in the sense that any race causes the wrong answer to be a tolerable wrong answer (refuse a write that would have fit, or accept a write that just barely overflows; both are recoverable).

12.27 Reference: Pre-Conditions for Each Primitive

Each primitive has rules about when and how it may be used. Violating a rule is a bug; the rules are listed here for quick reference.

12.27.1 `mtx(9)` (`MTX_DEF`)

Pre-conditions for `mtx_init`: - Memory for the struct `mtx` exists and is not aliased. - The mutex is not already initialized.

Pre-conditions for `mtx_lock`: - Mutex is initialized. - Calling thread is in process context, kernel-thread context, or callout-mpsafe context. - Calling thread does not already hold the mutex (unless `MTX_RECURSE`). - Calling thread does not hold any spin mutex.

Pre-conditions for `mtx_unlock`: - Calling thread holds the mutex.

Pre-conditions for `mtx_destroy`: - Mutex is initialized. - No thread holds the mutex. - No thread is blocked on the mutex.

12.27.2 cv(9)

Pre-conditions for `cv_init`: - Memory for the struct `cv` exists. - The `cv` is not already initialized.

Pre-conditions for `cv_wait` and `cv_wait_sig`: - The interlock mutex is held by the calling thread. - Calling thread is in a context where sleeping is legal.

Pre-conditions for `cv_signal` and `cv_broadcast`: - The `cv` is initialized. - Convention: the interlock mutex is held by the calling thread (not strictly required by the API, but defensive).

Pre-conditions for `cv_destroy`: - The `cv` is initialized. - No thread is blocked on the `cv` (the wait queue must be empty).

12.27.3 sx(9)

Pre-conditions for `sx_init`: - Memory for the struct `sx` exists. - The `sx` is not already initialized (unless `SX_NEW` is used).

Pre-conditions for `sx_xlock` and `sx_xlock_sig`: - The `sx` is initialized. - Calling thread does not already hold the `sx` exclusively (unless `SX_RECURSE`). - Calling thread is in a context where sleeping is legal. - Calling thread does not hold any non-sleepable lock (no spin mutex, no rw lock).

Pre-conditions for `sx_slock` and `sx_slock_sig`: - Same as `sx_xlock` except recursion check applies to shared mode.

Pre-conditions for `sx_xunlock` and `sx_sunlock`: - Calling thread holds the `sx` in the corresponding mode.

Pre-conditions for `sx_destroy`: - The `sx` is initialized. - No thread holds the `sx` in any mode. - No thread is blocked on the `sx`.

12.27.4 rw(9)

Pre-conditions for `rw_init`, `rw_destroy`: same shape as `sx_init`, `sx_destroy`.

Pre-conditions for `rw_wlock` and `rw_rlock`: - The `rw` is initialized. - Calling thread does not currently hold the `rw` in a conflicting mode. - Calling thread is *not* required to be in a sleepable context. The `rw` lock itself does not sleep; however, the calling thread *must not* call any function that may sleep while the `rw` is held.

Pre-conditions for `rw_wunlock` and `rw_runlock`: - Calling thread holds the `rw` in the corresponding mode.

Following these pre-conditions is the difference between a driver that runs cleanly under WITNESS for years and one that produces an unexpected panic on the first uncommon code path.

12.28 Reference: A Chapter 12 Self-Assessment

Use this rubric to confirm you have internalised the Chapter 12 material before turning to Chapter 13. Every question should be answerable without re-reading the chapter.

12.28.1 Conceptual Questions

1. **Name the three primary forms of synchronization.** Mutual exclusion, shared access with restricted writes, coordinated waiting.
2. **Why is a condition variable preferable to an anonymous wakeup channel?** Each cv represents one logical condition; signals do not wake unrelated waiters; the cv has a name visible in procstat and dtrace; the API enforces the atomic drop-and-sleep contract through its types.
3. **What is the difference between cv_signal and cv_broadcast?** cv_signal wakes one waiter (highest priority, FIFO among equal); cv_broadcast wakes all waiters. Use signal for per-event state changes; use broadcast for global changes (detach, reset).
4. **What does cv_wait_sig return when interrupted by a signal?** Either EINTR or ERESTART, depending on the signal's restart disposition. The driver passes the value through unchanged.
5. **What is the difference between sx and rw locks?** sx(9) is sleepable; rw(9) is not. Use sx in process context where the critical section may include sleeping calls; use rw when the critical section may run in interrupt context or must not sleep.
6. **Why does sx_try_upgrade exist instead of an unconditional sx_upgrade?** Because two simultaneous holders both attempting an unconditional upgrade would deadlock. The try variant returns failure when another shared holder is present, letting the caller back off cleanly.
7. **What is the snapshot-and-apply pattern and why is it useful?** Acquire one lock, read needed values into local variables, release; then acquire a different lock and use the local values. Avoids holding two locks simultaneously, relaxing lock-order constraints. Acceptable when the snapshot can tolerate small staleness.
8. **What is the canonical lock order in the Chapter 12 driver?** sc->mtx before sc->cfg_sx. Documented in LOCKING.md; enforced by WITNESS.

12.28.2 Code Reading Questions

Open your Chapter 12 driver source and verify:

1. Every cv_wait_sig is inside a while (!condition) loop.
2. Every cv has at least one signaller and one broadcast caller (broadcast in detach is acceptable).
3. Every sx_xlock has a matching sx_xunlock on every code path, including error returns.
4. The detach path broadcasts each cv before destroying any primitive.
5. The cfg sx is dropped before any potentially-sleeping call (sysctl_handle_*, uiomove, malloc(M_WAITOK)).
6. The lock order rule (mtx first, sx second) is followed in every path that holds both.

12.28.3 Hands-On Questions

1. Load the Chapter 12 driver on a WITNESS kernel and run the composite stress for 30 minutes. Are there any warnings? If yes, investigate.

2. Set `read_timeout_ms` to 100 and run a `read(2)` against an idle device. What does the call return? After how long?
3. Toggle `debug_level` between 0 and 3 with `sysctl -w` while `mp_stress` is running. Does the level take effect promptly? Does anything break?
4. Use `lockstat(1)` to measure the contention on the `sx` lock under a config-writer-heavy workload. What is the wait time?
5. Open the `kern_condvar.c` source and find the function `cv_signal`. Read it. Can you describe what it does in two sentences?

If all five hands-on questions pass and the conceptual questions are easy, your Chapter 12 work is solid.

12.29 Reference: Further Reading on Synchronization

For readers who want to go deeper than this chapter covers:

12.29.1 Manual Pages

- `mutex(9)`: the mutex API (covered fully in Chapter 11; reference here for completeness).
- `condvar(9)`: the condition variable API.
- `sx(9)`: the shared/exclusive lock API.
- `rwlock(9)`: the reader/writer lock API.
- `rmlock(9)`: the read-mostly lock API (advanced).
- `sema(9)`: the counting semaphore API (advanced).
- `epoch(9)`: the deferred-reclamation read-mostly framework (advanced; relevant for network drivers).
- `locking(9)`: an overview of FreeBSD's locking primitives.
- `lock(9)`: the common lock-object infrastructure.
- `witness(4)`: the WITNESS lock-order checker (covered in Chapter 11; revisited in this chapter).
- `lockstat(1)`: the lock-profiling user-space tool.
- `dtrace(1)`: the dynamic tracing framework, covered in more depth in Chapter 15.

12.29.2 Source Files

- `/usr/src/sys/kern/kern_condvar.c`: the `cv` implementation.
- `/usr/src/sys/kern/kern_sx.c`: the `sx` implementation.
- `/usr/src/sys/kern/kern_rwlock.c`: the `rw` implementation.
- `/usr/src/sys/kern/subr_sleepqueue.c`: the sleep-queue machinery underlying `cv` and other sleep primitives.
- `/usr/src/sys/kern/subr_turnstile.c`: the turnstile machinery underlying `rw` and other priority-propagating primitives.
- `/usr/src/sys/sys/condvar.h`, `/usr/src/sys/sys/sx.h`, `/usr/src/sys/sys/rwlock.h`: the public API headers.
- `/usr/src/sys/sys/_lock.h`, `/usr/src/sys/sys/lock.h`: the common lock-object structure and class registry.

12.29.3 External Material

For concurrency theory applicable to any operating system, *The Art of Multiprocessor Programming* by Herlihy and Shavit is excellent. For FreeBSD-specific kernel internals, *The Design and Implementation of the FreeBSD Operating System* by McKusick and others remains the canonical textbook; the chapters on locking and scheduling are particularly relevant.

Neither book is required for this chapter. Both are useful when the time comes for deeper study.

12.30 Looking Ahead: Bridge to Chapter 13

Chapter 13 is titled *Timers and Delayed Work*. Its scope is the kernel's time infrastructure as seen from a driver: how to schedule a callback for some time in the future, how to cancel one cleanly, how to handle the rules around callouts that may run concurrently with the driver's other code paths, and how to use timers for typical driver patterns like watchdogs, deferred work, and periodic polling.

Chapter 12 prepared the ground in three specific ways.

First, you already know how to wait with a timeout. Chapter 13's `callout(9)` mechanism is the same idea seen from the other side: instead of "wake me at time T", it is "run this function at time T". The synchronization rules around callouts (callouts run on a kernel thread, can race with your other code, must be drained before destruction) build on the discipline Chapter 12 established for `cvs` and `sxs`.

Second, you already know how to design a multi-primitive driver. Chapter 13's callouts add another execution context to the driver: the callout handler runs concurrently with `myfirst_read`, `myfirst_write`, and the `sysctl` handlers. That means callout handlers participate in the lock order. The `LOCKING.md` you wrote in Chapter 12 will absorb the addition with one new entry.

Third, you already know how to debug under load. Chapter 13 introduces a new bug class (callout races at unload time) that benefits from the same `WITNESS`, `lockstat`, and `dtrace` workflow Chapter 12 taught.

Specific topics Chapter 13 will cover include:

- The `callout(9)` API: `callout_init`, `callout_init_mtx`, `callout_reset`, `callout_stop`, `callout_drain`.
- The lock-aware callout (`callout_init_mtx`) and why it is the right default for driver code.
- Callout reuse: scheduling the same callout multiple times safely.
- The unload race: how a callout that fires after `kldunload` can crash the kernel, and how to prevent it with `callout_drain`.
- Periodic patterns: the watchdog, the heartbeat, the deferred reaper.
- The `tick_sbt` and `sbintime_t` time abstractions, useful for sub-millisecond timing.
- Comparison with `timeout(9)` (the older interface, deprecated for new code).

You do not need to read ahead. Chapter 12's material is sufficient preparation. Bring your Chapter 12 driver, your test kit, and your `WITNESS`-enabled kernel. Chapter 13 starts where Chapter 12 ended.

A small closing reflection. You started this chapter with one mutex, one anonymous channel, and a clear idea of what synchronization meant. You leave with three primitives, a documented lock order, a richer vocabulary, and the experience of debugging real coordination problems with real kernel tools. That progression is the heart

of Part 3 of this book. From here, Chapter 13 expands the driver's awareness of *time*, Chapter 14 expands its awareness of *interrupts*, and the remaining chapters of Part 3 prepare you for the hardware-touching chapters of Part 4.

Take a moment. The driver you started Part 3 with knew only how to handle one thread at a time. The driver you have now coordinates many threads across two lock classes, can be reconfigured at runtime without disrupting its data path, and respects the user's signals and deadlines. That is a real, qualitative leap. Then turn the page.

When you do open Chapter 13, the first thing you will see is `callout(9)`, the kernel's timed-callback infrastructure. The discipline you learned here for `cvs`, `sxs`, and the lock-order-aware design transfers directly. Callouts are simply another concurrent execution context that participates in the lock order; the patterns from Chapter 12 absorb them without growing brittle. The synchronization vocabulary is the same; the time vocabulary is what is new.

13 Timers and Delayed Work

13.1 Reader Guidance & Outcomes

So far, every line of driver code we have written has been *reactive*: the user calls `read(2)`, the kernel calls our handler, we do work, we return. Chapter 12’s blocking primitives extended that model with the ability to *wait* for something we did not initiate. But the driver itself never reached out to the world on its own. It had no way to say “in 100 milliseconds from now, please do this”. It had no way to count time at all, except as something it observed passing while it was already inside a syscall.

That changes here. Chapter 13 introduces *time* as a first-class concept in your driver. The kernel has an entire subsystem dedicated to running your code at a specific moment in the future, repeatedly if you ask, with precise lock-handling rules and clean teardown semantics. It is called `callout(9)`, and it is small, regular, and deeply useful. By the end of this chapter, your `myfirst` driver will have learned to schedule its own work, to act on the world without being prodded, and to give back its scheduled work safely when the device is unloaded.

The shape of the chapter mirrors Chapter 12’s. Each new primitive comes with a motivation, a precise API tour, a worked refactor of the running driver, a WITNESS-verified design, and a documented locking story in `LOCKING.md`. The chapter does not stop at “you can call `callout_reset`”; it walks you through every contract the callout subsystem cares about, so that the timers you write keep working when the rest of the driver evolves.

13.1.1 Why This Chapter Earns Its Own Place

You could try to fake timers. A kernel thread that sleeps in a loop, calling `cv_timedwait_sig` and doing work each time it wakes, is technically a timer. So is a user-space process that opens the device once a second and pokes a `sysctl`. Neither is wrong, but both are clumsy compared to what the kernel offers, and both create new resources (the kernel thread, the user-space process) that have their own lifetime to manage.

`callout(9)` is the right answer in nearly every case where you want a function to run “later”. It is built on the kernel’s hardware-clock infrastructure, costs essentially nothing at rest, scales to thousands of pending callouts per system, integrates with WITNESS and INVARIANTS, and provides clear rules for how to interact with locks and how to drain pending work at teardown. Most drivers in `/usr/src/sys/dev/` use it. Once you know it, the pattern transfers across every kind of driver you will meet: USB, network, storage, watchdog, sensor, anything whose physical world has a clock in it.

The cost of *not* knowing `callout(9)` is high. A driver that re-invents timing creates a private subsystem that nobody else knows how to debug. A driver that uses `callout(9)` correctly slots into the kernel’s existing observability tools (`procstat`, `dtrace`, `lockstat`, `ddb`) and behaves predictably at unload time. The chapter pays for itself the first time you have to extend a driver someone else wrote.

13.1.2 Where Chapter 12 Left the Driver

A brief recap of where you should stand, because Chapter 13 builds directly on the Chapter 12 deliverables. If any of the following is missing or feels uncertain, return to Chapter 12 before starting this chapter.

- Your `myfirst` driver compiles cleanly and is at version 0.6-sync.
- It uses `MYFIRST_LOCK(sc)` / `MYFIRST_UNLOCK(sc)` macros around `sc->mtx` (the data-path mutex).
- It uses `MYFIRST_CFG_SLOCK(sc)` / `MYFIRST_CFG_XLOCK(sc)` around `sc->cfg_sx` (the configuration sx).
- It uses two named condition variables (`sc->data_cv`, `sc->room_cv`) for blocking reads and writes.
- It supports timed reads via `cv_timedwait_sig` and the `read_timeout_ms` sysctl.
- The lock order `sc->mtx -> sc->cfg_sx` is documented in `LOCKING.md` and enforced by `WITNESS`.
- `INVARIANTS` and `WITNESS` are enabled in your test kernel; you have built and booted it.
- The Chapter 12 stress kit (Chapter 11 testers plus `timeout_tester` and `config_writer`) builds and runs cleanly.

That driver is what we extend in Chapter 13. We will add a periodic callout, then a watchdog callout, then a configurable tick-source, and finally consolidate them with a refactor pass and a documentation update. The driver's data path stays as it was; the new code lives next to the existing primitives.

13.1.3 What You Will Learn

By the time you move on to the next chapter, you will be able to:

- Explain when a callout is the right primitive for the job and when a kernel thread, a `cv_timedwait`, or a user-space helper would serve better.
- Initialize a callout with the appropriate lock-awareness using `callout_init`, `callout_init_mtx`, `callout_init_rw`, or `callout_init_rm`, and choose between the lock-managed and `mpsafe` variants for your driver's context.
- Schedule a one-shot timer with `callout_reset` (tick-based) or `callout_reset_sbt` (sub-tick precision), using the `tick_sbt`, `SBT_1S`, `SBT_1MS`, `SBT_1US` time constants where appropriate.
- Schedule a periodic timer by having the callback re-arm itself, with the right pattern that survives `callout_drain`.
- Choose between `callout_reset` and `callout_schedule` and understand when each is the right tool.
- Describe the lock contract `callout(9)` enforces when you initialize it with a lock pointer: the kernel acquires that lock before your function runs, releases it after (unless you set `CALLOUT_RETURNUNLOCKED`), and serialises the callout with respect to other lock holders.
- Read and interpret the `c_iflags` and `c_flags` fields of a callout, and use `callout_pending`, `callout_active`, and `callout_deactivate` correctly.
- Use `callout_stop` to cancel a pending callout in normal driver code, and `callout_drain` at teardown to wait for an in-flight callback to finish.
- Recognise the unload race (a callout fires after `kldunload` and crashes the kernel) and describe the standard cure: drain at detach, refuse detach until the device is quiet.
- Apply the `is_attached` pattern (which we built for `cv` waiters in Chapter 12) to callout callbacks, so a callback that fires during teardown returns cleanly without rescheduling.
- Build a watchdog timer that detects a stuck condition and acts on it.

- Build a debounce timer that ignores rapid repeated events.
- Build a periodic tick-source that injects synthetic data into the cbuf for testing.
- Verify the callout-enabled driver against WITNESS, lockstat(1), and a long-running stress test that includes timer activity.
- Extend LOCKING.md with a “Callouts” section that names every callout, its callback, its lock, and its lifetime.
- Refactor the driver into a shape where timer code is grouped, named, and obviously safe to maintain.

That is a substantial list. None of it is optional for a driver that uses time at all. All of it transfers directly to drivers that will appear in Part 4 and beyond, where real hardware brings its own clocks and demands its own watchdogs.

13.1.4 What This Chapter Does Not Cover

Several adjacent topics are deliberately deferred:

- **Taskqueues (taskqueue(9)).** Chapter 16 introduces the kernel’s general deferred-work framework. Taskqueues and callouts are complementary: callouts run a function at a specific time; taskqueues run a function as soon as a worker thread can pick it up. Many drivers use both: a callout fires at the right moment, the callout enqueues a task, the task runs the actual work in process context where sleeping is permitted. Chapter 13 stays inside the callout’s own callback for simplicity; the deferred-work pattern belongs to Chapter 16.
- **Hardware interrupt handlers.** Chapter 14 introduces interrupts. A real driver may install an interrupt handler that runs without process context. The callout(9) rules around lock classes are similar to the rules for interrupt handlers (you may not sleep), but the framing is different. We will revisit timer-and-interrupt interaction in Chapter 14.
- **epoch(9).** A read-mostly synchronization framework used by network drivers. Out of scope for Chapter 13.
- **High-resolution event scheduling.** The kernel exposes sbintime_t and the _sbt variants for sub-tick precision; we touch on the sbintime-based variants of the callout API briefly, but the full story of event timer drivers (/usr/src/sys/kern/kern_clocksource.c) belongs in a kernel-internals book, not a driver book.
- **Real-time and deadline scheduling.** Out of scope. We rely on the general scheduler.
- **Periodic workloads via the scheduler tick (hardclock).** The kernel itself uses hardclock(9) for system-wide periodic work; drivers do not interact with hardclock directly. We mention it for context.

Staying inside those lines keeps the chapter focused. The reader of Chapter 13 should finish with confident control of callout(9) and a working sense of when to reach for taskqueue(9) instead. Chapter 14 and Chapter 16 fill in the rest.

13.1.5 Estimated Time Investment

- **Reading only:** about three hours. The API surface is small but the lock and lifetime rules deserve careful attention.
- **Reading plus typing the worked examples:** six to eight hours over two sessions. The driver evolves in four small stages; each stage adds one timer pattern.

- **Reading plus all labs and challenges:** ten to fourteen hours over three or four sessions, including time for stress tests with timers active and lockstat measurements.

If you find yourself confused midway through Section 5 (the lock-context rules), that is normal. The interaction between callouts and locks is the most surprising part of the API, and even experienced kernel programmers occasionally get it wrong. Stop, re-read Section 5's worked example, and continue when the model has settled.

13.1.6 Prerequisites

Before starting this chapter, confirm:

- Your driver source matches Chapter 12 Stage 4 (`stage4-final`). The starting point assumes the cv channels, bounded reads, sx-protected configuration, and reset sysctl are all in place.
- Your lab machine is running FreeBSD 14.3 with `/usr/src` on disk and matching the running kernel.
- A debug kernel with `INVARIANTS`, `WITNESS`, `WITNESS_SKIPSPIN`, `DDB`, `KDB`, and `KDB_UNATTENDED` is built, installed, and booting cleanly.
- You read Chapter 12 carefully. The lock-order discipline, the cv pattern, and the snapshot-and-apply pattern are assumed knowledge here.
- You have run the Chapter 12 composite stress kit at least once and seen it pass cleanly.

If any of the above is shaky, fixing it now is a much better investment than pushing through Chapter 13 and trying to debug from a moving foundation.

13.1.7 How to Get the Most Out of This Chapter

Three habits will pay off quickly.

First, keep `/usr/src/sys/kern/kern_timeout.c` and `/usr/src/sys/sys/callout.h` bookmarked. The header is short and contains the API the chapter teaches. The implementation file is long but well-commented; we will point at specific functions a few times during the chapter. Two minutes spent at each pointer pays off.

A note on line numbers. When this chapter names a specific function in `kern_timeout.c` or a macro in `callout.h`, treat the name as the fixed address and any line we happen to mention as scenery around it. `callout_init_mtx` and `callout_reset_sbt_on` will still carry those names in future FreeBSD 14.x revisions; the lines where they sit will have moved by the time your editor opens the file. Jump to the symbol and let your editor report the rest.

Second, run every code change you make under `WITNESS`. The callout subsystem has its own rules that `WITNESS` checks at runtime. The most common Chapter 13 mistake is to schedule a callout whose function tries to acquire a sleepable lock from a context where sleeping is illegal; `WITNESS` catches it immediately on a debug kernel and silently corrupts on a production kernel. Do not run Chapter 13 code on the production kernel until it passes the debug kernel.

Third, type the changes by hand. The companion source under `examples/part-03/ch13-timers-and-delayed-work/` is the canonical version, but the muscle memory of typing `callout_init_mtx(&sc->heartbeat_co, &sc->mtx, 0)` once is worth more than reading it ten times. The chapter shows the changes incrementally; mirror that incremental pace in your own copy of the driver.

13.1.8 Roadmap Through the Chapter

The sections in order are:

1. Why use timers in a driver. Where time enters the picture in real driver work, and which patterns map onto `callout(9)` rather than something else.
2. Introduction to FreeBSD's `callout(9)` API. The structure, the lifecycle, the four initialization variants, and what each one buys you.
3. Scheduling one-shot and repeating events. `callout_reset`, `callout_reset_sbt`, `callout_schedule`, and the periodic-callback re-arm pattern.
4. Integrating timers into your driver. The first refactor: a heartbeat callout that periodically logs stats or fires a synthetic event.
5. Handling locking and context in timers. The lock-aware initialization, the `CALLOUT_RETURNUNLOCKED` and `CALLOUT_SHAREDLOCK` flags, the rules around what a callout function may and may not do.
6. Timer cleanup and resource management. The unload race, `callout_stop` versus `callout_drain`, the standard detach pattern with timers.
7. Use cases and extensions for timed work. Watchdogs, debouncing, periodic polling, delayed retries, statistics rollovers, all framed as small recipes you can lift into other drivers.
8. Refactoring and versioning. A clean `LOCKING.md` extension, a bumped version string, an updated changelog, and a regression run that includes timer-related testing.

Hands-on labs and challenge exercises follow, then a troubleshooting reference, a wrapping-up section, and a bridge to Chapter 14.

If this is your first pass, read linearly and do the labs in order. If you are revisiting, the cleanup section (Section 6) and the refactoring section (Section 8) stand alone and make good single-sitting reads.

13.2 Section 1: Why Use Timers in a Driver?

Most of what a driver does is reactive. A user opens the device, the open handler runs. A user issues `read(2)`, the read handler runs. A signal arrives, a sleeper wakes. The kernel hands control to the driver in response to something the world did. Each invocation has a clear cause, and once the work is done, the driver returns and waits for the next cause.

Real hardware does not always cooperate with that model. A network card may need a heartbeat sent every few seconds even when nothing else is happening, just to convince the switch on the other end that the link is alive. A storage controller may need a watchdog reset every five hundred milliseconds, or it will assume the host has gone away and reset the channel. A USB hub poll has to happen on a timer because the USB bus does not interrupt for the kind of state change the driver wants to see. A button on a development board needs debouncing because the spring contacts produce many events in quick succession when the user only meant one. A driver that retries a transient failure should back off, not loop tightly.

All of those are reasons to schedule code for the future. The kernel has a single primitive for the job, `callout(9)`, and Chapter 13 teaches it from the ground up. Before we get to the API, this section sets the conceptual stage. We look at what “later” means in a driver, what shapes a “later” callback typically takes, and where `callout(9)` fits relative to the other ways a driver could express the idea of time.

13.2.1 The Three Shapes of “Later”

Driver code that wants to do something in the future falls into one of three shapes. Knowing which shape you are in is half of choosing the right primitive.

One-shot. “Do this once, X milliseconds from now, then forget about it.” Examples: schedule a watchdog timeout that fires only if no activity has been observed in the next second; debounce a button by ignoring all subsequent presses for fifty milliseconds; defer a teardown step until the current operation completes. The callback runs once and the driver does not re-arm it.

Periodic. “Do this every X milliseconds, until I tell you to stop.” Examples: poll a hardware register that does not interrupt; emit a heartbeat to a peer; refresh a cached value; sample a sensor; rotate a statistics window. The callback runs once, then re-arms itself for the next interval, and continues until the driver stops it.

Bounded wait. “Do this when condition Y becomes true, but give up if Y has not happened in X milliseconds.” Examples: wait for a hardware response with a timeout; wait for the buffer to drain or the deadline to fire; allow Ctrl-C to interrupt a wait. We met this shape in Chapter 12 with `cv_timedwait_sig`. The driver thread is the one that waits, not a callback.

`callout(9)` is the primitive for the first two shapes. The third uses `cv_timedwait_sig` (Chapter 12), `mtx_sleep` with a non-zero `timo`, or one of the `_sbt` variants for sub-tick precision. The two are complementary, not alternatives: many drivers use both. A bounded wait suspends the calling thread; a callout runs in a separate context after a delay.

13.2.2 Real-World Patterns

A short tour of patterns that recur across drivers in `/usr/src/sys/dev/`. Recognising them early gives you a vocabulary for the rest of the chapter.

The heartbeat. A periodic callout that fires every N milliseconds and emits some trivial state (a counter increment, a log line, a packet on the wire). Useful for debugging and for protocols that need a liveness signal.

The watchdog. A one-shot callout scheduled at the start of an operation. If the operation completes normally, the driver cancels the callout. If the operation hangs, the callout fires and the driver takes corrective action (reset the hardware, log a warning, kill a stuck request). Almost every storage and network driver has at least one watchdog.

The debounce. A one-shot callout scheduled when an event arrives. Subsequent identical events within the timeout are ignored. When the callout fires, the driver acts on the most recent event. Used for hardware events that bounce (mechanical switches, optical sensors).

The poll. A periodic callout that reads a hardware register and acts on the value. Used when the hardware does not produce interrupts for the events the driver cares about, or when interrupts are too noisy to be useful.

The retry-with-backoff. A one-shot callout scheduled with an increasing delay after each failed attempt. The first failure schedules a 10 ms retry; the second schedules a 20 ms retry; and so on. Bounds the rate at which the driver pesters the hardware after a failure.

The statistics rollover. A periodic callout that takes a snapshot of internal counters at regular intervals, computes a per-interval rate, and stores it in a circular buffer for later inspection.

The deferred reaper. A one-shot callout that completes a teardown after some grace period. Used when an object cannot be freed immediately because some other code path may still hold a reference; the callout waits long enough for those references to drain, then frees the object.

We will implement the first three (heartbeat, watchdog, deferred tick-source) in `myfirst` over the course of this chapter. The others are the same shape; once you know the pattern, the variations are mechanical.

13.2.3 Why Not Just Use a Kernel Thread?

A reasonable question for a beginner: why is `callout(9)` a separate API at all? Could the same effects be achieved by a kernel thread that loops, sleeps, and acts?

In principle, yes. In practice, no driver should reach for a kernel thread when a callout will do.

A kernel thread is a heavyweight resource. It has its own stack (typically 16 KB on amd64), its own scheduler entry, its own priority, its own state. Spinning one up for a periodic action that takes 10 microseconds every second is wasteful: 16 KB of memory plus scheduler overhead just to wake up, do trivial work, and sleep again. Multiplied across many drivers, the kernel ends up with hundreds of mostly-idle threads.

A callout is essentially free at rest. The data structure is a few pointers and integers (see `struct callout` in `/usr/src/sys/sys/_callout.h`). There is no thread, no stack, no scheduler entry. The kernel's hardware-clock interrupt walks a callout wheel and runs every callout that has come due, then returns. Thousands of pending callouts cost essentially nothing until they fire.

A callout also slots into the kernel's existing observability tools. `dtrace`, `lockstat`, and `procstat` all understand callouts. A custom kernel thread does not have any of that for free; you would have to instrument it yourself.

The exception, of course, is when the work the timer needs to do is genuinely long and would benefit from being in a sleepable thread context. A callout function may not sleep; if your work requires sleeping, the callout's job is to *enqueue* the work onto a taskqueue or wake a kernel thread that can do it safely. Chapter 16 covers that pattern. For Chapter 13, the work the callout does is short, non-sleeping, and lock-aware.

13.2.4 Why Not Just Use `cv_timedwait` In a Loop?

Another reasonable alternative: a kernel thread that loops on `cv_timedwait_sig` would also produce periodic behaviour. So would a user-space helper that polls a `sysctl`. Why callout?

The kernel-thread answer is the resource argument from the previous subsection: callouts are vastly cheaper than threads.

The user-space-helper answer is correctness: a driver whose timing depends on a user-space process is a driver that fails when that process crashes, gets paged out, or is denied CPU by an unrelated workload. A driver should be self-sufficient for its own correctness, even if user-space tooling provides additional features on top.

There is one situation where `cv_timedwait_sig` is the right answer: when the *calling thread itself* needs to wait. Chapter 12's `read_timeout_ms` `sysctl` uses `cv_timedwait_sig` because the reader is the one waiting; it has work to do as soon as data arrives or the deadline fires. A callout would be wrong because the reader's `syscall` thread cannot be the one running the callback (the callback runs in a different context).

Use `cv_timedwait_sig` when the syscall thread waits. Use `callout(9)` when something independent of any syscall thread must happen at a specific time. The two coexist comfortably in the same driver; Chapter 13 will end with a driver that uses both.

13.2.5 A Brief Note on Time Itself

The kernel exposes time through several units, each with its own conventions. We met them in Chapter 12; a recap helps before we wade into the API.

- **int ticks.** The legacy unit. `hz` ticks equal one second. The default `hz` on FreeBSD 14.3 is 1000, so one tick is one millisecond. `callout_reset` takes its delay in ticks.
- **sbintime_t.** A 64-bit signed binary fixed-point representation: upper 32 bits seconds, lower 32 bits a fraction of a second. The unit constants are in `/usr/src/sys/sys/time.h`: `SBT_1S`, `SBT_1MS`, `SBT_1US`, `SBT_1NS`. `callout_reset_sbt` takes its delay in sbintime.
- **tick_sbt.** A global variable holding `1 / hz` as an sbintime. Useful when you have a tick count and want the equivalent sbintime: `tick_sbt * tmo_in_ticks`.
- **The precision argument.** `callout_reset_sbt` takes an additional precision argument. It tells the kernel how much wiggle is acceptable when scheduling, which lets the callout subsystem coalesce nearby timers for power efficiency. A precision of zero means “fire as close to the deadline as possible”. A precision of `SBT_1MS` means “anywhere within a millisecond of the deadline is fine”.

For most driver work the tick-based API is the right level of precision. We use `callout_reset` (ticks) throughout the early sections of the chapter and reach for `callout_reset_sbt` only when sub-millisecond precision matters or when we want to tell the kernel about acceptable slop.

13.2.6 When a Callout Is the Wrong Tool

For completeness, three situations where `callout(9)` is *not* the right answer.

- **The work needs to sleep.** Callout functions run in a context that may not sleep. If the work involves `uiomove`, `copyin`, `malloc(M_WAITOK)`, or any other potentially-blocking call, the callout must enqueue a task on a taskqueue or wake a kernel thread that can do the work in process context. Chapter 16.
- **The work needs to run on a specific CPU for cache reasons.** `callout_reset_on` lets you bind a callout to a specific CPU, which is useful, but if the requirement is “run on the same CPU that submitted the request” the answer might be a per-CPU primitive instead. We touch on `callout_reset_on` briefly and defer the deeper CPU-affinity discussion.
- **The work is event-driven, not time-driven.** If the trigger is “data arrived” rather than “100 ms have elapsed”, you want a `cv` or a wakeup, not a callout. Mixing the two often leads to unnecessary complexity.

13.2.7 A Mental Model: The Callout Wheel

To make the cost argument from the previous subsections concrete, here is what the kernel actually does to manage callouts. You do not need this to use `callout(9)` correctly, but knowing it makes several later sections easier to follow.

The kernel maintains a *callout wheel* per CPU. Conceptually, the wheel is a circular array of buckets. Each bucket corresponds to a small range of time. When you call `callout_reset(co, ticks_in_future, fn, arg)`, the kernel computes which bucket “now plus ticks_in_future” lands in and adds the callout to that bucket’s list. The arithmetic is $(\text{current\_tick} + \text{ticks\_in\_future}) \bmod \text{wheel\_size}$.

A periodic timer interrupt (the hardware clock) fires at every tick. The interrupt handler increments a global tick counter, looks at the current bucket of every CPU’s wheel, and walks the list. For each callout that has reached its deadline, the kernel pulls it off the wheel and either runs the callback inline (for `C_DIRECT_EXEC` callouts) or hands it to a callout-processing thread.

Three properties of this mechanism matter for the chapter.

First, scheduling a callout is cheap: it is essentially “compute a bucket index and link the structure into a list”. A few atomic operations. No allocation. No context switch.

Second, an unscheduled callout costs nothing: it is just a `struct callout` somewhere in your softc. The kernel does not know about it until you call `callout_reset`. There is no per-callout overhead at rest.

Third, the granularity of the wheel is one tick. A 1.7-tick delay rounds up to 2 ticks. The `precision` argument to `callout_reset_sbt` lets you trade exactness for the kernel’s freedom to coalesce nearby firings, which is a power-saving optimization on systems with many concurrent timers. For driver work, the default precision is almost always fine.

There is a great deal more in the actual implementation: per-CPU wheels for cache locality, deferred migration when a callout is rescheduled to a different CPU, special handling for `C_DIRECT_EXEC` callouts that run in the timer interrupt itself, and so on. The implementation is in `/usr/src/sys/kern/kern_timeout.c` if you are curious. Reading it once is worthwhile; you do not need to memorize it.

13.2.8 What “Now” Means in the Kernel

A small but recurring confusion: there are several time bases in the kernel, and they measure different things.

`ticks` is a global variable that counts hardware-clock interrupts since boot. It increments by one every clock tick. It is fast to read (one memory load), wraps around every few weeks at typical `hz=1000`, and is the time base `callout_reset` uses. Always express callout deadlines as “now plus N ticks”, which is what `callout_reset(co, N, ...)` does.

`time_uptime` and `time_second` are `time_t` values that count seconds since boot or since the epoch (respectively). Less precise; useful for log timestamps and human-readable elapsed times.

`sbinuptime()` returns an `sbintime_t` representing seconds-and-fractions since boot. This is the time base `callout_reset_sbt` works in. It does not wrap (well, it would in some hundreds of years).

`getmicrouptime()` and `getnanouptime()` are coarse but fast accessors for “now”; they may be a tick or two old. `microuptime()` and `nanouptime()` are precise but more expensive (they read the hardware timer directly).

For a driver doing typical timer work, `ticks` (for tick-based callout work) and `getsbinuptime()` (for sbintime-based work) are the two that come up. We use them in the labs without commentary; if you wonder where they come from, this is the answer.

13.2.9 Wrapping Up Section 1

Time enters driver work in three shapes: one-shot, periodic, and bounded wait. The first two are exactly what `callout(9)` is for; the third is what Chapter 12's `cv_timedwait_sig` is for. Real-world patterns (heartbeat, watchdog, debounce, poll, retry, rollover, deferred reaper) are all instances of one-shot or periodic callouts; recognising them lets you reuse the same primitive across many situations.

Underneath the API, the kernel maintains a per-CPU callout wheel that costs essentially nothing at rest and very little to schedule. The granularity is one tick (one millisecond on a typical FreeBSD 14.3 system). The implementation handles thousands of pending callouts per CPU without breaking a sweat.

Section 2 introduces the API: the callout structure, the four initialization variants, and the lifecycle every callout follows.

13.3 Section 2: Introduction to FreeBSD's `callout(9)` API

`callout(9)` is, like most of the synchronization primitives, a small API on top of a careful implementation. The data structure is short, the lifecycle is regular (init, schedule, fire, stop or drain, destroy), and the rules are explicit enough that you can verify your usage by reading the source. This section walks through the structure, names the variants of init, and lays out the lifecycle stages so the rest of the chapter has a vocabulary to reuse.

13.3.1 The Callout Structure

The data structure lives in `/usr/src/sys/sys/_callout.h`:

```
struct callout {
    union {
        LIST_ENTRY(callout) le;
        SLIST_ENTRY(callout) sle;
        TAILQ_ENTRY(callout) tqe;
    } c_links;
    sbintime_t c_time;           /* ticks to the event */
    sbintime_t c_precision;     /* delta allowed wrt opt */
    void *c_arg;                /* function argument */
    callout_func_t *c_func;     /* function to call */
    struct lock_object *c_lock; /* lock to handle */
    short c_flags;              /* User State */
    short c_iflags;             /* Internal State */
    volatile int c_cpu;         /* CPU we're scheduled on */
};
```

It is one structure per callout, embedded in the softc or wherever else you need it. The fields you touch directly are: none. Every interaction goes through API calls. The fields you can read for diagnostic purposes are `c_flags` (via `callout_active / callout_pending`) and `c_arg` (rarely useful from outside).

The two flag fields deserve a sentence each.

`c_iflags` is internal. The kernel sets and clears bits in it under the callout subsystem's own lock. The bits encode whether the callout is on a wheel or processing list, whether it is pending, and a handful of internal book-keeping states. Driver code uses `callout_pending(c)` to read it; nothing else.

`c_flags` is external. The caller (your driver) is supposed to manage two bits in it: `CALLOUT_ACTIVE` and `CALLOUT_RETURNUNLOCKED`. The active bit is meant to track "I have asked this callout to be scheduled and have not cancelled it yet". The returnunlocked bit changes the lock-handling contract; we will get to that in Section 5. Driver code reads the active bit via `callout_active(c)` and clears it via `callout_deactivate(c)`.

The `c_lock` field deserves its own paragraph. When you initialize the callout with `callout_init_mtx`, `callout_init_rw`, or `callout_init_rm`, the kernel records the lock pointer here. Later, when the callout fires, the kernel acquires that lock before calling your callback function and releases it after the callback returns (unless you specifically asked otherwise). This means your callback runs as if the caller had acquired the lock for it. The lock-managed callout is almost always what you want for driver code; we will say more in Section 5.

13.3.2 The Callback Function Signature

A callout's callback function has a single argument: a `void *`. The kernel passes whatever you registered with `callout_reset` (or its variants). The function returns `void`. Its full signature, from `/usr/src/sys/sys/_callout.h`:

```
typedef void callout_func_t(void *);
```

Convention: pass a pointer to your `softc` (or to whatever per-instance state the callback needs). The first line of the callback casts the void pointer back to the struct pointer:

```
static void
myfirst_heartbeat(void *arg)
{
    struct myfirst_softc *sc = arg;
    /* ... do timer work ... */
}
```

The argument is fixed at registration time and does not change between firings. If you need to pass changing context to the callback, store it somewhere the callback can find via the `softc`.

13.3.3 The Four Initialization Variants

`callout(9)` offers four ways to initialize a callout, distinguished by what kind of lock (if any) the kernel will acquire for you before the callback runs.

```
void callout_init(struct callout *c, int mpsafe);

#define callout_init_mtx(c, mtx, flags) \
```

```
_callout_init_lock((c), &(mtx)->lock_object, (flags))
#define callout_init_rw(c, rw, flags) \
    _callout_init_lock((c), &(rw)->lock_object, (flags))
#define callout_init_rm(c, rm, flags) \
    _callout_init_lock((c), &(rm)->lock_object, (flags))
```

`callout_init(c, mpsafe)` is the legacy, lock-naive variant. The `mpsafe` argument is now badly named; it really means “may run without acquiring Giant for me”. Pass 1 for any modern driver code; pass 0 only if you genuinely want the kernel to acquire Giant before your callback (almost never, and only in very old code paths). New drivers should not use this variant. The chapter mentions it for completeness because you will see it in old code.

`callout_init_mtx(c, mtx, flags)` registers a sleep mutex (`MTX_DEF`) as the callout’s lock. Before each firing, the kernel acquires the mutex and releases it after the callback returns. This is the variant you will use for almost all driver code. It pairs naturally with the `MTX_DEF` mutex you already have on the data path.

`callout_init_rw(c, rw, flags)` registers an `rw(9)` reader/writer lock. The kernel takes the write lock unless you set `CALLOUT_SHAREDLOCK`, in which case it takes the read lock. Less common in driver code; useful when the callback needs to read a piece of read-mostly state and several callouts share the same lock.

`callout_init_rm(c, rm, flags)` registers an `rmlock(9)`. Specialized; used in network drivers with hot read paths that should not contend.

For the `myfirst` driver, every callout we add will use `callout_init_mtx(&sc->some_co, &sc->mtx, 0)`. The kernel acquires `sc->mtx` before the callback runs, the callback can manipulate the `cbuf` and other mutex-protected state without taking the lock itself, and the kernel releases `sc->mtx` after. The pattern is clean, the rules are explicit, and `WITNESS` will yell if you violate them.

13.3.4 The flags Argument

The flags argument to `_callout_init_lock` is one of two values for driver code:

- 0: the callout’s lock is acquired before the callback and released after. This is the default and the right answer almost every time.
- `CALLOUT_RETURNUNLOCKED`: the callout’s lock is acquired before the callback. The callback is responsible for releasing it (or it may have already been released by something the callback called). This is occasionally useful when the callback’s last action is to drop the lock and do something the lock cannot cover.
- `CALLOUT_SHAREDLOCK`: only valid for `callout_init_rw` and `callout_init_rm`. The lock is acquired in shared mode rather than exclusive.

For Chapter 13 we use 0 everywhere. `CALLOUT_RETURNUNLOCKED` is mentioned in Section 5 for completeness; the chapter does not need it.

13.3.5 The Five Lifecycle Stages

Every callout follows the same five-stage lifecycle. Knowing the stages by name will make the rest of the chapter much easier to read.

Stage 1: initialized. The struct `callout` has been initialized with one of the init variants. It has a lock association (or `mpsafe`). It has not been scheduled. Nothing will fire until you tell it to.

Stage 2: pending. You have called `callout_reset` or `callout_reset_sbt`. The kernel has placed the callout on its internal wheel and noted the time at which it should fire. `callout_pending(c)` returns true. The callback has not yet run. You can cancel by calling `callout_stop(c)`, which removes it from the wheel.

Stage 3: firing. The deadline has arrived and the kernel is now running the callback. If the callout has a registered lock, the kernel has acquired it. Your callback function is executing. During this stage `callout_active(c)` is true, `callout_pending(c)` may be false (it has been removed from the wheel). The callback is free to call `callout_reset` to re-arm itself (this is the periodic pattern).

Stage 4: completed. The callback has returned. If the callback re-armed via `callout_reset`, the callout is back in stage 2. Otherwise it is now idle: `callout_pending(c)` is false. If the kernel acquired a lock for the callback, it has released it.

Stage 5: destroyed. The callout's underlying memory is no longer needed. There is no `callout_destroy` function; instead, you must ensure that the callout is not pending and not firing, then free the containing structure. The standard tool for the “wait for the callout to be safely idle” job is `callout_drain`. Section 6 covers this in detail.

The cycle is: init once, alternate between pending and (firing + completed) any number of times, drain, free.

13.3.6 A First Look at the API

We have not yet scheduled anything. Let us read the four most important calls, with one-line summaries each:

```
int callout_reset(struct callout *c, int to_ticks,
                  void (*fn)(void *), void *arg);
int callout_reset_sbt(struct callout *c, sbintime_t sbt,
                      sbintime_t prec, void (*fn)(void *), void *arg, int flags);
int callout_stop(struct callout *c);
int callout_drain(struct callout *c);
```

`callout_reset` schedules the callout. The first argument is the callout to schedule. The second is the delay in ticks (multiply seconds by `hz` to convert; on FreeBSD 14.3, `hz=1000` typically, so a tick is a millisecond). The third is the callback function. The fourth is the argument to pass to the callback. Returns nonzero if the callout was previously pending and got cancelled (so the new schedule is replacing the old).

`callout_reset_sbt` is the same but takes the delay as `sbintime_t` and accepts a precision and flags. Used for sub-tick precision or when you want to tell the kernel about acceptable slop. Most drivers use `callout_reset` and reach for `_sbt` only when needed.

`callout_stop` cancels a pending callout. If the callout was pending, it is removed from the wheel and never fires. Returns nonzero if a pending callout was cancelled. If the callout was not pending (already fired, or never scheduled), the call is a no-op and returns zero. Critically: `callout_stop` does *not* wait for an in-flight callback to finish. If the callback is currently executing on another CPU, `callout_stop` returns before the callback returns.

`callout_drain` is the safe-for-teardown variant. It cancels the callout if pending, *and* waits for any currently-executing callback to return before itself returning. After `callout_drain` returns, the callout is guaranteed to be idle and not running anywhere. This is the function you call at detach time. Section 6 walks through why this matters.

13.3.7 Reading the Source

If you have ten minutes, open `/usr/src/sys/sys/callout.h` and `/usr/src/sys/kern/kern_timeout.c` and skim. Three things to look for:

The header defines the public API in less than 130 lines. Every function the chapter mentions is declared there. The macros that wrap `_callout_init_lock` are clearly visible.

The implementation file is long (around 1550 lines in FreeBSD 14.3), but the function names match the API. `callout_reset_sbt_on` is the core scheduling function; everything else is a wrapper. `_callout_stop_safe` is the unified stop-and-maybe-drain function; `callout_stop` and `callout_drain` are macros that call it with different flags. `callout_init` and `_callout_init_lock` sit near the bottom of the file.

The chapter cites FreeBSD functions and tables by name rather than line number, because line numbers drift between releases while the function and symbol names survive. If you need approximate line numbers for `kern_timeout.c` in FreeBSD 14.3: `callout_reset_sbt_on` near 936, `_callout_stop_safe` near 1085, `callout_init` near 1347. Open the file and jump to the symbol; the line is whatever your editor reports.

The KASSERTs scattered through the source are the rules in code form. For example, the assertion in `_callout_init_lock` that “you may not give me a sleepable lock” enforces the rule that callouts may not block on a lock that could sleep. Reading those assertions builds confidence that the API guarantees what it says.

13.3.8 A Worked Lifecycle Walkthrough

Putting the lifecycle stages on a timeline makes them concrete. Imagine a heartbeat callout that is initialized at attach, enabled at $t=0$, and disabled at $t=2.5$ seconds.

- **$t=-1s$ (attach time):** The driver calls `callout_init_mtx(&sc->heartbeat_co, &sc->mtx, 0)`. The callout is now in stage 1 (initialized). `callout_pending(c)` returns false. The kernel knows about the callout’s lock association.
- **$t=0s$:** The user enables the heartbeat by writing the sysctl. The handler acquires `sc->mtx`, sets `interval_ms = 1000`, and calls `callout_reset(&sc->heartbeat_co, hz, myfirst_heartbeat, sc)`. The callout transitions to stage 2 (pending). `callout_pending(c)` returns true. The kernel has placed it on a wheel bucket corresponding to $t+1$ second.
- **$t=1s$:** The deadline arrives. The kernel pulls the callout off the wheel (`callout_pending(c)` becomes false). The kernel acquires `sc->mtx`. The kernel calls `myfirst_heartbeat(sc)`. The callout is now in stage 3 (firing). The callback runs, emits a log line, calls `callout_reset` to re-arm. The re-arm puts the callout back on a wheel bucket for $t+2$ seconds. `callout_pending(c)` is true again. The callback returns. The kernel releases `sc->mtx`. The callout is now in stage 2 (pending) again, waiting for its next firing.
- **$t=2s$:** Same sequence. The callback fires, re-arms, the callout is pending for $t+3$ seconds.

- **t=2.5s:** The user disables the heartbeat by writing the sysctl. The handler acquires `sc->mtx`, sets `interval_ms = 0`, and calls `callout_stop(&sc->heartbeat_co)`. The kernel removes the callout from the wheel. `callout_stop` returns 1 (it cancelled a pending callout). `callout_pending(c)` becomes false. The callout is now back in stage 1 (initialized but idle).
- **t=∞ (later, at detach time):** The detach path calls `callout_drain(&sc->heartbeat_co)`. The callout is already idle; `callout_drain` returns immediately. The driver may now safely free the surrounding state.

Notice three things about the timeline.

The cycle of pending → firing → pending repeats indefinitely as long as the callback re-arms. There is no hard limit on iterations.

A `callout_stop` can intercept the pending-firing-pending cycle at any point. If the callout is in stage 2 (pending), `callout_stop` cancels it. If the callout is in stage 3 (firing) on another CPU, `callout_stop` does *not* cancel it (the callback will run to completion); the next iteration of the cycle will not happen because the callback's re-arm condition (`interval_ms > 0`) is now false.

The `is_attached` check in the callback (which we will introduce in Section 4) provides a similar interception point during teardown. If the callback fires after detach has cleared `is_attached`, the callback exits without re-arming, and the next iteration does not happen.

This timeline is the entire shape of `callout(9)` use in driver code. Variations involve adding a one-shot pattern (no re-arm), a watchdog pattern (cancel on success), or a debounce pattern (only schedule if not already pending). The lifecycle stages are the same.

13.3.9 A Word About “active” Versus “pending”

Two related concepts that beginners sometimes conflate.

`callout_pending(c)` is set by the kernel when the callout is on the wheel waiting to fire. It is cleared by the kernel when the callout fires (the callback is about to run) or when `callout_stop` cancels it.

`callout_active(c)` is set by the kernel when `callout_reset` succeeds. It is cleared by `callout_deactivate` (a function you call) or by `callout_stop`. Crucially, the kernel does *not* clear `callout_active` when the callback fires. The active bit is a flag that says “I scheduled this callout and have not actively cancelled it”; whether the callback has fired since then is a separate question.

A callout can be in any of four states:

- not active and not pending: never scheduled, or cancelled via `callout_stop`, or deactivated via `callout_deactivate` after firing.
- active and pending: scheduled, on the wheel, waiting to fire.
- active and not pending: scheduled, fired (or about to fire), and the callback has not yet called `callout_deactivate`.
- not active and pending: rare, but possible if the driver calls `callout_deactivate` while the callout is still scheduled. Most drivers never reach this state because they call `callout_deactivate` only inside the callback, after the pending bit has already been cleared.

For most drivers you only need `callout_pending` (used in patterns like the debounce). The `active` flag matters more in code that wants to know “did we schedule a callout, even if it has already run?”. For Chapter 13, we use `pending` once and never use `active`.

13.3.10 Wrapping Up Section 2

Callouts are small structures with a small API and a regular lifecycle. The four initialization variants pick the kind of lock the kernel will acquire for you (or none). The four functions you will use most are `callout_reset`, `callout_reset_sbt`, `callout_stop`, and `callout_drain`. Section 3 puts them to work, scheduling one-shot and periodic timers and showing how the periodic re-arm pattern actually works.

13.4 Section 3: Scheduling One-shot and Repeating Events

A timer in `callout(9)` is always conceptually one-shot. There is no `callout_reset_periodic` function. Periodic behaviour is built by having the callback re-arm itself at the end of each firing. Both the one-shot and the periodic patterns use the same API call (`callout_reset`); the difference is whether the callback decides to schedule the next firing.

This section walks through both patterns with worked examples that compile and run. We will not yet integrate them into `myfirst`; that is Section 4. Here we focus on the timing primitives and the patterns you will use.

13.4.1 The One-Shot Pattern

The simplest possible callout: schedule a callback to fire once, in the future.

```
static void
my_oneshot(void *arg)
{
    device_printf((device_t)arg, "one-shot fired\n");
}

void
schedule_a_one_shot(device_t dev, struct callout *co)
{
    callout_reset(co, hz / 10, my_oneshot, dev);
}
```

`hz / 10` means “100 milliseconds from now” on a system with `hz=1000`. The callback receives the device pointer we registered. It runs once, prints, and returns. The callout is now idle. To run it again, you would call `callout_reset` again.

Three things to notice. First, the callback’s argument is whatever we passed to `callout_reset`, untyped, recovered with a cast. Second, the callback emits one log line and returns; it does not re-schedule. This is the one-shot pattern. Third, we used `hz / 10` rather than a hard-coded value. Always express callout delays in terms of `hz` so the code is portable across systems with different clock rates.

If you wanted a 250 ms delay, you would write `hz / 4` (or `hz * 250 / 1000` for clarity). For a 5 second delay, `hz * 5`. The arithmetic is integer; for fractional values, multiply before dividing to preserve precision.

13.4.2 The Periodic Pattern

For periodic behaviour, the callback re-arms itself at the end:

```
static void
my_periodic(void *arg)
{
    struct myfirst_softc *sc = arg;
    device_printf(sc->dev, "tick\n");
    callout_reset(&sc->heartbeat_co, hz, my_periodic, sc);
}

void
start_periodic(struct myfirst_softc *sc)
{
    callout_reset(&sc->heartbeat_co, hz, my_periodic, sc);
}
```

The first call to `callout_reset` (in `start_periodic`) arms the callout for one second from now. When it fires, `my_periodic` runs, emits a log line, and re-arms for one second after the present moment. The next firing happens, the cycle continues. To stop the periodic firing, call `callout_stop(&sc->heartbeat_co)` (or `callout_drain` at teardown). Once the callout has been stopped, `my_periodic` will not fire again until `start_periodic` is called again.

Three subtleties.

First, the re-arm happens at the *end* of the callback. If the callback's work takes a long time, the next firing is delayed by that work. The actual interval between firings is roughly `hz` ticks plus the time the callback took. For most driver use cases this is fine. If you need an exact period, use `callout_schedule` or `callout_reset_sbt` with a calculated absolute deadline.

Second, the callback is called with the callout's lock acquired (we will see why in Section 5). When the callback calls `callout_reset` to re-arm, the callout subsystem handles the re-arm correctly even though it is being called from inside the firing of the same callout. The kernel's internal bookkeeping is designed for exactly this pattern.

Third, if the driver is being torn down at the same moment the callback re-arms, you have a race: the re-arm puts the callout back on the wheel after the cancel/drain has run. Section 6 explains how to handle this. The short answer is: at detach time, set a "shutting down" flag in the `softc` under the mutex, then `callout_drain` the callout. The callback checks the flag at entry and returns without re-arming if it sees the flag set. The drain waits for the in-flight callback to return.

13.4.3 `callout_schedule` for Re-Arming Without Repeating Arguments

For periodic callouts, the callback re-arms with the same function and argument every time. `callout_reset` requires you to pass them again. `callout_schedule` is a convenience that uses the function and argument from the last `callout_reset`:

```
int callout_schedule(struct callout *c, int to_ticks);
```

Inside the periodic callback:

```
static void
my_periodic(void *arg)
{
    struct myfirst_softc *sc = arg;
    device_printf(sc->dev, "tick\n");
    callout_schedule(&sc->heartbeat_co, hz);
}
```

The kernel uses the function pointer and argument it remembers from the last `callout_reset` call. Less typing, and the code reads slightly cleaner. Both `callout_reset` and `callout_schedule` work for the periodic pattern; pick whichever you prefer.

13.4.4 Sub-Tick Precision With `callout_reset_sbt`

When you need precision finer than a tick, or you want to tell the kernel about acceptable slop, use the sbintime variant:

```
int callout_reset_sbt(struct callout *c, sbintime_t sbt,
                    sbintime_t prec,
                    void (*fn)(void *), void *arg, int flags);
```

Example: schedule a 250-microsecond timer:

```
sbintime_t sbt = 250 * SBT_1US;
callout_reset_sbt(&sc->fast_co, sbt, SBT_1US,
    my_callback, sc, C_HARDCLOCK);
```

The `prec` argument is the precision the caller is willing to accept. `SBT_1US` says “anywhere within one microsecond of the deadline is fine”; the kernel may coalesce this timer with other timers up to one microsecond apart. `0` means “fire as close to the deadline as possible”. The flags include `C_HARDCLOCK` (align to the system clock interrupt, default for most cases), `C_DIRECT_EXEC` (run in the timer interrupt context, only useful with a spin lock), `C_ABSOLUTE` (interpret `sbt` as absolute time rather than a relative delay), and `C_PRECALC` (used internally; don’t set it).

For the chapter, we use `callout_reset` (tick-based) almost everywhere. `callout_reset_sbt` is mentioned for completeness; the lab section has one exercise using it.

13.4.5 Cancellation: `callout_stop`

To cancel a pending callout, call `callout_stop`:

```
int callout_stop(struct callout *c);
```

If the callout is pending, the kernel removes it from the wheel and returns 1. If the callout is not pending (already fired, or never scheduled), the call is a no-op and returns 0.

Critically: `callout_stop` does *not* wait. If the callback is currently executing on another CPU when `callout_stop` is called, the call returns immediately. The callback continues to run on the other CPU and finishes when it finishes. If the callback re-arms itself, the callout is back on the wheel after `callout_stop` returns.

This means `callout_stop` is the right tool for normal operation (cancel a pending callout because the condition that motivated it has been resolved) but the *wrong* tool for teardown (where you must wait for any in-flight callback to finish before freeing the surrounding state). For teardown, use `callout_drain`. Section 6 covers this distinction in depth.

The standard pattern in normal operation:

```
/* Decided we don't need this watchdog any more */
if (callout_stop(&sc->watchdog_co)) {
    /* The callout was pending; we just cancelled it. */
    device_printf(sc->dev, "watchdog cancelled\n");
}
/* If callout_stop returned 0, the callout had already fired
   or was never scheduled; nothing to do. */
```

A small point of caution: between `callout_stop` returning 1 and the next statement running, no other thread can re-arm the callout because we hold the lock that protects the surrounding state. Without the lock, `callout_stop` would still cancel correctly, but the meaning of the return value would become racy.

13.4.6 Cancellation: `callout_drain`

`callout_drain` is the teardown-safe variant:

```
int callout_drain(struct callout *c);
```

Like `callout_stop`, it cancels a pending callout. *Unlike* `callout_stop`, if the callback is currently executing on another CPU, `callout_drain` waits for it to return before itself returning. After `callout_drain` returns, the callout is guaranteed to be idle: not pending, not firing, and (if the callback has not re-armed) it will not fire again.

Two important rules.

First, the caller of `callout_drain` *must not* hold the callout's lock. If the callout is currently executing (it has acquired the lock and is running the callback), `callout_drain` needs to wait for the callback to return, which

means the callback needs to release the lock, which means the caller of `callout_drain` cannot be holding it. Holding the lock would deadlock.

Second, `callout_drain` may sleep. The thread waits on a sleep queue for the callback to finish. Therefore `callout_drain` is only legal in contexts where sleeping is allowed (process context or kernel thread; not interrupt or spin-lock context).

The standard teardown pattern:

```
static int
myfirst_detach(device_t dev)
{
    struct myfirst_softc *sc = device_get_softc(dev);

    /* mark "going away" so a re-arming callback will not re-schedule */
    MYFIRST_LOCK(sc);
    sc->is_attached = 0;
    MYFIRST_UNLOCK(sc);

    /* drain the callout: cancel pending, wait for in-flight */
    callout_drain(&sc->heartbeat_co);
    callout_drain(&sc->watchdog_co);

    /* now safe to destroy other primitives and free state */
    /* ... */
}
```

Section 6 expands this pattern in detail.

13.4.7 `callout_pending` and `callout_active`

Two diagnostic accessors that are useful when you want to know what state a callout is in:

```
int callout_pending(const struct callout *c);
int callout_active(const struct callout *c);
void callout_deactivate(struct callout *c);
```

`callout_pending(c)` returns nonzero if the callout is currently scheduled and waiting to fire. False if the callout has already fired (or never been scheduled, or was cancelled).

`callout_active(c)` returns nonzero if `callout_reset` has been called on this callout since the last `callout_deactivate`. The “active” bit is something *you* manage. The kernel never sets or clears it on its own (with one small exception: a successful `callout_stop` clears it). The convention is that the callback clears the bit at the start, the rest of the driver sets it when scheduling, and code that wonders “do I have a pending or just-fired callout?” can check `callout_active`.

For most driver work you do not need either accessor. We mention them because real driver source uses them and you should recognize the pattern. The `myfirst` driver in Chapter 13 uses `callout_pending` once, in the watchdog cancellation path; the rest of the chapter does not need them.

13.4.8 A Worked Example: A Two-Stage Schedule

Putting the pieces together: a small worked example that schedules a callback to fire 100 ms from now, then re-schedules it for 500 ms after that, then runs it once and stops.

```
static int g_count = 0;
static struct callout g_co;
static struct mtx g_mtx;

static void
my_callback(void *arg)
{
    printf("callback fired (count=%d)\n", ++g_count);
    if (g_count == 1) {
        /* Reschedule for 500 ms later. */
        callout_reset(&g_co, hz / 2, my_callback, NULL);
    } else if (g_count == 2) {
        /* Done; do nothing, callout becomes idle. */
    }
}

void
start_test(void)
{
    mtx_init(&g_mtx, "test_co", NULL, MTX_DEF);
    callout_init_mtx(&g_co, &g_mtx, 0);
    callout_reset(&g_co, hz / 10, my_callback, NULL);
}

void
stop_test(void)
{
    callout_drain(&g_co);
    mtx_destroy(&g_mtx);
}
```

Ten lines of substance. The callback decides whether to re-arm based on the count. After two firings, it stops re-arming and the callout becomes idle. `stop_test` drains the callout (waiting if necessary for any in-flight firing), then destroys the mutex.

This pattern, with variations, is the entire shape of `callout(9)` use in driver code. Section 4 puts it inside `myfirst` and gives it real work to do.

13.4.9 Wrapping Up Section 3

Callouts are scheduled with `callout_reset` (tick-based) or `callout_reset_sbt` (sbintime-based). One-shot behaviour comes from a callback that does not re-arm; periodic behaviour comes from a callback that re-arms itself at the end. Cancellation is `callout_stop` for normal operation and `callout_drain` for teardown. The accessors `callout_pending`, `callout_active`, and `callout_deactivate` are for diagnostic inspection.

Section 4 takes the patterns of this section and integrates a real callout into the `myfirst` driver: a heartbeat that periodically logs a stat line.

13.5 Section 4: Integrating Timers into Your Driver

Theory is comfortable; integration is where the rough edges show up. This section walks through adding a heartbeat callout to `myfirst`. The heartbeat fires once per second, logs a short statistics line, and re-arms itself. We will see how the callout integrates with the existing mutex, how the lock-aware initialization removes a class of race, how the `is_attached` flag from Chapter 12 protects the callback during teardown, and how `WITNESS` confirms the design is correct.

Consider this Stage 1 of the chapter's driver evolution. By the end of this section, the `myfirst` driver has its first timer.

13.5.1 Adding a Heartbeat Callout

Add two fields to struct `myfirst_softc`:

```
struct myfirst_softc {
    /* ... existing fields ... */
    struct callout      heartbeat_co;
    int                 heartbeat_interval_ms; /* 0 = disabled */
    /* ... rest ... */
};
```

`heartbeat_co` is the callout itself. `heartbeat_interval_ms` is a sysctl-tunable that lets the user enable, disable, and adjust the heartbeat at runtime. A value of zero disables the heartbeat. A positive value is the interval in milliseconds.

Initialize the callout in `myfirst_attach`. Place the call after the mutex is initialized and before the `cdev` is created (so the callout is ready to schedule but no user can yet trigger anything):

```
static int
myfirst_attach(device_t dev)
{
```

```

    /* ... existing setup ... */

    mtx_init(&sc->mtx, device_get_nameunit(dev), "myfirst", MTX_DEF);
    cv_init(&sc->data_cv, "myfirst data");
    cv_init(&sc->room_cv, "myfirst room");
    sx_init(&sc->cfg_sx, "myfirst cfg");
    callout_init_mtx(&sc->heartbeat_co, &sc->mtx, 0);

    /* ... rest of attach ... */
}

```

`callout_init_mtx(&sc->heartbeat_co, &sc->mtx, 0)` registers `sc->mtx` as the callout's lock. From this point on, any time the heartbeat callout fires, the kernel will acquire `sc->mtx` before calling our callback and release it after the callback returns. This is exactly the contract we want: the callback may freely manipulate `cbuf` state and per-softc fields without taking the lock itself.

Drain the callout in `myfirst_detach`, before destroying the primitives:

```

static int
myfirst_detach(device_t dev)
{
    struct myfirst_softc *sc = device_get_softc(dev);

    /* ... refuse detach while active_fhs > 0 ... */
    /* ... clear is_attached and broadcast cvs under sc->mtx ... */

    seldrain(&sc->rsl);
    seldrain(&sc->wsl);

    callout_drain(&sc->heartbeat_co);

    if (sc->cdev_alias != NULL) { destroy_dev(sc->cdev_alias); /* ... */ }
    /* ... rest of detach as before ... */
}

```

The `callout_drain` call must come *after* `is_attached` is cleared and the cvs are broadcast (so a callback that fires during the drain sees the cleared flag), and *before* any primitive the callback might touch is destroyed. The cleared `is_attached` flag prevents the callback from rescheduling; the drain waits for any in-flight callback to finish. After `callout_drain` returns, no callback can be running and none is pending; the rest of detach can safely free state.

13.5.2 The Heartbeat Callback

Now the callback itself:

```

static void
myfirst_heartbeat(void *arg)
{
    struct myfirst_softc *sc = arg;
    size_t used;
    uint64_t br, bw;
    int interval;

    MYFIRST_ASSERT(sc);

    if (!sc->is_attached)
        return; /* device going away; do not re-arm */

    used = cbuf_used(&sc->cb);
    br = counter_u64_fetch(sc->bytes_read);
    bw = counter_u64_fetch(sc->bytes_written);
    device_printf(sc->dev,
        "heartbeat: cb_used=%zu, bytes_read=%ju, bytes_written=%ju\n",
        used, (uintmax_t)br, (uintmax_t)bw);

    interval = sc->heartbeat_interval_ms;
    if (interval > 0)
        callout_reset(&sc->heartbeat_co,
            (interval * hz + 999) / 1000,
            myfirst_heartbeat, sc);
}

```

Ten lines that capture the entire periodic-heartbeat pattern. Let us walk through them.

`MYFIRST_ASSERT(sc)` confirms that `sc->mtx` is held. The callout was initialized with `callout_init_mtx(&sc->heartbeat_co, &sc->mtx, 0)`, so the kernel acquired `sc->mtx` before calling us; the assertion is a sanity check that catches the case where someone (perhaps a future maintainer) accidentally changes the init to `callout_init` without paying attention.

`if (!sc->is_attached) return;` is the teardown guard. If the detach path has cleared `is_attached`, we exit immediately without doing any work and without re-arming. The drain in `myfirst_detach` will see the callout idle and complete cleanly.

The `cbuf-used` and counter reads happen under the lock. We call `cbuf_used` (which expects `sc->mtx` held) and `counter_u64_fetch` (which is lockless and safe everywhere). The `device_printf` call is potentially expensive but is conventional for log lines; we tolerate the cost because it happens at most once per second.

The re-arm at the end uses the current value of `heartbeat_interval_ms`. If the user has set it to zero (disabled the heartbeat), we do not re-arm, and the callout becomes idle until something else schedules it. If the user has changed the interval, the next firing will use the new value. This is a small but significant feature: the heartbeat's frequency is dynamically configurable without restarting the driver.

The $(\text{interval} * \text{hz} + 999) / 1000$ arithmetic converts milliseconds to ticks, rounding up. Same formula as Chapter 12's bounded waits, for the same reason: never round below the requested duration.

13.5.3 Starting the Heartbeat From a Sysctl

The user enables the heartbeat by writing a non-zero value to `dev.myfirst.<unit>.heartbeat_interval_ms`. We need a sysctl handler that schedules the first firing:

```
static int
myfirst_sysctl_heartbeat_interval_ms(SYSCTL_HANDLER_ARGS)
{
    struct myfirst_softc *sc = arg1;
    int new, old, error;

    old = sc->heartbeat_interval_ms;
    new = old;
    error = sysctl_handle_int(oidp, &new, 0, req);
    if (error || req->newptr == NULL)
        return (error);

    if (new < 0)
        return (EINVAL);

    MYFIRST_LOCK(sc);
    sc->heartbeat_interval_ms = new;
    if (new > 0 && old == 0) {
        /* Enabling: schedule the first firing. */
        callout_reset(&sc->heartbeat_co,
            (new * hz + 999) / 1000,
            myfirst_heartbeat, sc);
    } else if (new == 0 && old > 0) {
        /* Disabling: cancel any pending heartbeat. */
        callout_stop(&sc->heartbeat_co);
    }
    MYFIRST_UNLOCK(sc);
    return (0);
}
```

The handler:

1. Reads the current value (so a read-only query returns the current interval).
2. Lets `sysctl_handle_int` validate and update the local `new` variable.
3. Validates that the new value is non-negative.
4. Acquires `sc->mtx` to commit the change atomically against any racing callout activity.
5. If the heartbeat was disabled and is now enabled, schedules the first firing.

6. If the heartbeat was enabled and is now disabled, cancels the pending callout.
7. Drops the lock and returns.

Note the symmetric handling. If the user toggles the heartbeat off and on rapidly, the handler does the right thing each time. A re-arm in the callback would not fire a new heartbeat after the user disables it (the callback checks `heartbeat_interval_ms` before re-arming). A schedule from the sysctl would not double-schedule (the callback re-arms only if `interval_ms > 0`, and the sysctl only schedules if `old == 0`).

A subtle point: the callout is initialized with `sc->mtx` as its lock, and the sysctl handler acquires `sc->mtx` before calling `callout_reset`. The kernel acquires `sc->mtx` for callbacks too. This means the sysctl handler and any in-flight callback are serialized: the sysctl waits if a callback is currently running, and the callback cannot run while the sysctl holds the lock. The race “user disables the heartbeat just as the callback re-arms” is closed by the lock.

Register the sysctl in attach:

```
SYSCALL_ADD_PROC(&sc->sysctl_ctx,
    SYSCTL_CHILDREN(device_get_sysctl_tree(dev)),
    OID_AUTO, "heartbeat_interval_ms",
    CTLTYPE_INT | CTLFLAG_RW | CTLFLAG_MPSAFE,
    sc, 0, myfirst_sysctl_heartbeat_interval_ms, "I",
    "Heartbeat interval in milliseconds (0 = disabled)");
```

And initialize `heartbeat_interval_ms = 0` in attach so the heartbeat is disabled by default. A user opts in by setting the sysctl; the driver is silent until then.

13.5.4 Verifying the Refactor

Build the new driver and load it on a WITNESS kernel. Three tests:

Test 1: heartbeat off by default.

```
$ kldload ./myfirst.ko
$ dmesg | tail -3  # attach line shown; no heartbeat logs
$ sleep 5
$ dmesg | tail -3  # still no heartbeat logs
```

Expected: the attach line, then nothing. The heartbeat is disabled by default.

Test 2: heartbeat on.

```
$ sysctl -w dev.myfirst.0.heartbeat_interval_ms=1000
$ sleep 5
$ dmesg | tail -10
```

Expected: about five heartbeat lines, one per second:

```
myfirst0: heartbeat: cb_used=0, bytes_read=0, bytes_written=0
myfirst0: heartbeat: cb_used=0, bytes_read=0, bytes_written=0
myfirst0: heartbeat: cb_used=0, bytes_read=0, bytes_written=0
```

Test 3: heartbeat under load.

In one terminal:

```
$ ../../part-02/ch10-handling-io-efficiently/userland/producer_consumer
```

While that runs, observe dmesg in another terminal. The heartbeat lines should now show non-zero byte counts:

```
myfirst0: heartbeat: cb_used=0, bytes_read=1048576, bytes_written=1048576
```

Test 4: disable cleanly.

```
$ sysctl -w dev.myfirst.0.heartbeat_interval_ms=0
$ sleep 5
$ dmesg | tail -3  # nothing new
```

The heartbeat stops; no further lines are emitted.

Test 5: detach with heartbeat active.

```
$ sysctl -w dev.myfirst.0.heartbeat_interval_ms=1000
$ kldunload myfirst
```

Expected: detach succeeds. The drain in `myfirst_detach` cancels the pending callout and waits for any in-flight firing to complete. No WITNESS warnings, no panics.

If any of these tests fails, the most likely cause is either (a) the `is_attached` check is missing in the callback (so the callback re-arms during teardown and the `callout_drain` never returns) or (b) the lock initialization is wrong (so the callback runs without the expected mutex held and `MYFIRST_ASSERT` fires).

13.5.5 A Note on the Heartbeat's Overhead

A 1-second heartbeat is essentially free: one callout per second, three counter reads, one log line, one re-arm. Total CPU: microseconds per firing. Memory: zero beyond the struct `callout` already in the softc.

A 1-millisecond heartbeat is a different story. A thousand log lines per second will saturate the dmesg buffer in seconds and dominate the driver's CPU usage. Use short intervals only when the work is genuinely fast and the log is gated by a debug level.

For demonstration purposes, 1000 (one per second) is reasonable. For a real-world heartbeat the reasonable range is probably 100 ms to 10 s. The chapter does not enforce a minimum; the user's choice is their own.

13.5.6 A Mental Model: How a Heartbeat Plays Out

A step-by-step picture of what the kernel and the driver do during a single heartbeat firing. Useful for cementing the lifecycle vocabulary in concrete terms.

- **t=0:** User runs `sysctl -w dev.myfirst.0.heartbeat_interval_ms=1000`.
- **t=0+δ:** The `sysctl` handler runs. It reads the current value (0), validates the new value (1000), and acquires `sc->mtx`. Inside the lock: it sets `sc->heartbeat_interval_ms = 1000`. Detecting the 0 → non-zero transition, it calls `callout_reset(&sc->heartbeat_co, hz, myfirst_heartbeat, sc)`. The kernel computes the wheel bucket for “now plus 1000 ticks” and links the callout into that bucket. The handler releases `sc->mtx` and returns 0.
- **t=0 to t=1s:** The kernel does other work. The callout sits on the wheel.
- **t=1s:** The hardware-clock interrupt fires. The callout subsystem walks the current wheel bucket and finds `sc->heartbeat_co` waiting. The kernel removes it from the wheel and dispatches it.
- **t=1s+δ:** A callout-processing thread wakes (or, if the system is idle, the timer interrupt itself runs the callback). The kernel acquires `sc->mtx` (this may briefly block if another thread holds it; for our typical workload, the lock is free). Once `sc->mtx` is held, the kernel calls `myfirst_heartbeat(sc)`.
- **Inside the callback:** `MYFIRST_ASSERT(sc)` confirms the lock is held. The `is_attached` check passes. The callback reads `cbuf_used` (lock held; safe), reads the per-CPU counters (lockless; always safe), emits one `device_printf` line. It checks `sc->heartbeat_interval_ms` (1000); since it is positive, it calls `callout_reset(&sc->heartbeat_co, hz, myfirst_heartbeat, sc)` to schedule the next firing. The kernel re-links the callout into the wheel bucket for “now plus 1000 ticks”. The callback returns.
- **t=1s+ε:** The kernel releases `sc->mtx`. The callout is now back on the wheel, waiting for t=2s.
- **t=2s:** The cycle repeats.

Three observations.

First, the kernel manages the lock on your behalf. Your callback runs as if some unseen caller had acquired `sc->mtx` for it. There is no `mtx_lock/mtx_unlock` in the callback because the kernel handles them.

Second, the re-arm is just another `callout_reset` call. It is allowed because the callout’s lock is held; the kernel’s internal bookkeeping handles the case “this callout is currently firing, and is being re-armed inside its own callback”.

Third, the time between firings is approximately `hz` ticks, but is slightly more than that: the callback’s work time plus any scheduling latency adds to the interval. For a 1-second heartbeat the drift is microseconds; for a 1-millisecond heartbeat it could be measurable. If exact period matters, use `callout_reset_sbt` and compute the next deadline as “previous deadline + interval”, not “now + interval”.

13.5.7 Visualizing the Timer With `dtrace`

A useful sanity check: confirm the heartbeat fires at the rate you configured.

```
# dtrace -n 'fbt::myfirst_heartbeat:entry { @ = count(); } tick-1sec { printa(@);  
↵ trunc(@); }'
```

This dtrace one-liner counts how many times `myfirst_heartbeat` is entered each second. With `heartbeat_interval_ms=1000`, the count should be 1 per second. With `heartbeat_interval_ms=100`, the count should be 10. With `heartbeat_interval_ms=10`, the count should be 100.

If the count is wildly off from the expected value, the configuration did not take effect. Common causes: the `sysctl` handler did not commit the change (bug in the handler), the callback is exiting early due to `is_attached == 0` (bug in the teardown flow), or the system is so loaded that the callout is firing late and pile-up is occurring. In normal operation the count should be stable to within one count per second.

A more elaborate dtrace recipe: histogram of the time spent in the callback.

```
# dtrace -n '  
fbt::myfirst_heartbeat:entry { self->ts = timestamp; }  
fbt::myfirst_heartbeat:return /self->ts/ {  
    @ = quantize(timestamp - self->ts);  
    self->ts = 0;  
}  
tick-30sec { exit(0); }'
```

Each callback typically takes a few microseconds (the time to read counters and emit a log line). If the histogram shows callbacks taking milliseconds or more, something is wrong; investigate.

13.5.8 Wrapping Up Section 4

The driver has its first timer. The heartbeat callout fires periodically, logs a stat line, and re-arms. The `is_attached` flag (introduced in Chapter 12 for `cv` waiters) plays exactly the same role here: it lets the callback exit cleanly when the device is being torn down. The lock-aware initialization (`callout_init_mtx` with `sc->mtx`) means the callback runs with the data-path mutex held, and the kernel handles the lock acquisition for us.

Section 5 examines the lock contract more carefully. The contract is the most important rule in the callout API; getting it right makes the rest easy, and getting it wrong creates bugs that are hard to find.

13.6 Section 5: Handling Locking and Context in Timers

Section 4 used `callout_init_mtx` and trusted the kernel to acquire `sc->mtx` before each firing. This section opens that black box. We look at exactly what the kernel does with the lock pointer you registered, what guarantees you can assume inside the callback, and what you may and may not do during a firing.

The lock contract is the most important rule in `callout(9)`. A driver that respects it is correct by construction. A driver that violates it produces races that are hard to reproduce and even harder to diagnose. Spend the time on this section now; the rest of the chapter is easier when the model is solid.

13.6.1 What the Kernel Does Before Your Callback Runs

When a callout's deadline arrives, the kernel's callout-processing code (in `/usr/src/sys/kern/kern_timeout.c`) finds the callout on the wheel and prepares to fire it. The preparation depends on what lock

you registered:

- **No lock (`callout_init` with `mpsafe=1`).** The kernel sets `c_iflags`, marks the callout as no longer pending, and calls your function directly. Your function must do all its own locking.
- **A mutex (`callout_init_mtx` with an `MTX_DEF` mutex).** The kernel acquires the mutex with `mtx_lock`. If the mutex is contended, the firing thread blocks until it can acquire. Once the mutex is held, the kernel calls your function. After your function returns, the kernel releases the mutex with `mtx_unlock` (unless you set `CALLOUT_RETURNUNLOCKED`).
- **An rw lock (`callout_init_rw`).** Same as the mutex case, but with `rw_wlock` (or `rw_rlock` if you set `CALLOUT_SHAREDLOCK`).
- **An rmlock (`callout_init_rm`).** Same shape with rmlock primitives.
- **Giant (the default of `callout_init` with `mpsafe=0`).** The kernel acquires Giant. Avoid this for new code.

The lock is acquired in the firing thread's context. From your callback's point of view, the lock is just held: the same invariants apply as if any other thread had called `mtx_lock` and then called your function.

13.6.2 Why the Lock Is Acquired By the Kernel

A natural question: why doesn't the callback acquire the lock itself? The kernel-acquires-the-lock model has three subtle benefits.

It cooperates correctly with `callout_drain`. When `callout_drain` waits for an in-flight callback to finish, it must know whether a callback is currently running. The kernel's callout subsystem tracks exactly that, but only because it is the code that acquired the lock and began the callback. If the callback acquired its own lock, the subsystem would not know the difference between "callback is currently blocked trying to acquire the lock" and "callback has returned", and a clean drain would be impossible to implement without exposing kernel-private state. The kernel-acquires model keeps the subsystem firmly in control of the firing timeline.

It enforces the lock-class rule. The kernel checks at registration time that the lock you supply is not sleepable beyond what callouts can tolerate. A sleepable `sx` lock or `lockmgr` lock would let the callback call `cv_wait`, which in callout context is illegal. The init function (`_callout_init_lock` in `kern_timeout.c`) has the assertion: `KASSERT(lock == NULL || !(LOCK_CLASS(lock)->lc_flags & LC_SLEEPABLE), ...)` to catch this.

It serializes the callback against `callout_reset` and `callout_stop`. When the callback is firing, the lock is held. When you call `callout_reset` or `callout_stop` from your driver code, you must hold the same lock (the kernel checks). Therefore the cancel/reschedule and the firing are mutually exclusive: at any moment, either the callback is firing (lock held by the kernel-acquired path) or the driver code is reacting to a state change (lock held by the driver code). They never run concurrently.

This third property is what makes the heartbeat `sysctl` handler in Section 4 race-free. The handler acquires `sc->mtx`, decides to cancel or schedule, and the cancel/schedule completes atomically against any in-flight callback. No special precautions needed; the lock does the work.

13.6.3 What You May Do Inside a Callback

The callback runs with the registered lock held. The lock determines what is legal.

For an MTX_DEF mutex (our case), the rules are the same as any other code that holds a sleep mutex:

- You may read and write any state protected by the mutex.
- You may call `cbuf_*` helpers and other mutex-internal operations.
- You may call `cv_signal` and `cv_broadcast` (the `cv` API does not require the interlock to be dropped first).
- You may call `callout_reset`, `callout_stop`, or `callout_pending` on the same callout (re-arm, cancel, or check).
- You may call `callout_reset` on a *different* callout if you hold its lock too (or if it's mpsafe).

13.6.4 What You May Not Do Inside a Callback

The same rules: no sleeping while holding the mutex.

- You may **not** call `cv_wait`, `cv_wait_sig`, `mtx_sleep`, or any other sleeping primitive directly. (The mutex is held; sleeping with it held would be a sleep-with-mutex violation that WITNESS catches.)
- You may **not** call `uiomove`, `copyin`, or `copyout` (each may sleep).
- You may **not** call `malloc(..., M_WAITOK)`. Use `M_NOWAIT` instead, with proper error handling for the allocation failure case.
- You may **not** call `selwakeup` (it takes its own locks that may produce ordering violations).
- You may **not** call any function that may sleep.

The callback should be short. A few microseconds of work is typical. If you need to do something long-running, the callback should *enqueue* the work to a taskqueue or wake a kernel thread that can do it in process context. Chapter 16 covers the taskqueue pattern.

13.6.5 What If You Do Need To Sleep?

The standard answer: do not sleep in the callback. Defer the work. Two patterns are common.

Pattern 1: Set a flag, signal a kernel thread. The callback sets a flag in the softc and signals a `cv`. A kernel thread (created by `kproc_create` or `kthread_add`, both topics for later chapters) is waiting on the `cv`; it wakes, does the long-running work in process context, and goes back to waiting. The callback is short; the work is unconstrained.

Pattern 2: Enqueue a task on a taskqueue. The callback calls `taskqueue_enqueue` to defer the work to a taskqueue worker thread. The worker runs in process context and may sleep. Again, the callback is short; the work is unconstrained. Chapter 16 introduces this in depth.

For Chapter 13 we keep all our timer work short and lock-friendly; we do not yet need to defer. The pattern is mentioned so you know the option exists.

13.6.6 The `CALLOUT_RETURNUNLOCKED` Flag

`CALLOUT_RETURNUNLOCKED` changes the lock contract. Without it, the kernel acquires the lock before calling the callback and releases it after the callback returns. With it, the kernel acquires the lock before calling the callback and the *callback* is responsible for releasing it (or the callback may call something that drops the lock).

Why would you want this? Two reasons.

The callback drops the lock to do something that cannot be done under it. For example, the callback finishes its locked work, drops the lock, then enqueues a task on a taskqueue. The enqueue does not need the lock and could even violate ordering if held. Setting `CALLOUT_RETURNUNLOCKED` lets you write the drop in the natural place.

The callback hands the lock off to another function. If the callback calls a helper that takes ownership of the lock and is responsible for releasing it, `CALLOUT_RETURNUNLOCKED` documents the handoff to `WITNESS` so the assertion check passes.

Without `CALLOUT_RETURNUNLOCKED`, the kernel will assert that the lock is still held by the firing thread when the callback returns. The flag tells the assertion that the callback is allowed to leave the function with the lock dropped.

For Chapter 13 we do not need `CALLOUT_RETURNUNLOCKED`. All our callbacks acquire no additional locks, drop no locks, and return with the same lock state as on entry. The flag is mentioned so you recognize it in real-driver source.

13.6.7 The `CALLOUT_SHAREDLOCK` Flag

`CALLOUT_SHAREDLOCK` is only valid for `callout_init_rw` and `callout_init_rm`. It tells the kernel to acquire the lock in shared (read) mode rather than exclusive (write) mode before calling the callback.

Used when the callback only reads state and there are many callouts that share the same lock. With `CALLOUT_SHAREDLOCK`, multiple callbacks can run concurrently as long as no writer holds the lock.

For Chapter 13 we use `callout_init_mtx` with `MTX_DEF`, where shared mode does not exist. The flag is mentioned for completeness.

13.6.8 The “Direct Execution” Mode

The kernel offers a “direct” mode where the callout function runs in the timer interrupt context itself, rather than being deferred to a thread. The flag is `C_DIRECT_EXEC`, passed to `callout_reset_sbt`. It is documented in `/usr/src/sys/sys/callout.h` and only valid for callouts whose lock is a spin mutex (or no lock at all).

Direct execution is fast (no context switch, no thread wakeup) but the rules are stricter than ordinary callout context: no sleeping (already true), no acquiring sleep mutexes, no calling functions that might. The function runs in interrupt context, with all the constraints that implies (Chapter 14).

For Chapter 13 we never use `C_DIRECT_EXEC`. Our callouts are not time-critical to that degree. We mention it because you will see it in some hardware drivers (especially network drivers with hot RX paths).

13.6.9 A Worked Example: Lock Contract in the Heartbeat

Recall the heartbeat callback from Section 4:

```
static void
myfirst_heartbeat(void *arg)
{
    struct myfirst_softc *sc = arg;
    size_t used;
    uint64_t br, bw;
    int interval;

    MYFIRST_ASSERT(sc);

    if (!sc->is_attached)
        return;

    used = cbuf_used(&sc->cb);
    br = counter_u64_fetch(sc->bytes_read);
    bw = counter_u64_fetch(sc->bytes_written);
    device_printf(sc->dev,
        "heartbeat: cb_used=%zu, bytes_read=%ju, bytes_written=%ju\n",
        used, (uintmax_t)br, (uintmax_t)bw);

    interval = sc->heartbeat_interval_ms;
    if (interval > 0)
        callout_reset(&sc->heartbeat_co,
            (interval * hz + 999) / 1000,
            myfirst_heartbeat, sc);
}
```

Walk through the lock contract:

- The callout was initialized with `callout_init_mtx(&sc->heartbeat_co, &sc->mtx, 0)`. The kernel holds `sc->mtx` before calling us.
- The `MYFIRST_ASSERT(sc)` confirms `sc->mtx` is held. Sanity check.
- `sc->is_attached` is read under the lock. Safe.
- `cbuf_used(&sc->cb)` is called. The `cbuf` helper expects `sc->mtx` held; we have it.
- `counter_u64_fetch(sc->bytes_read)` is called. `counter(9)` is lockless and safe everywhere.
- `device_printf` is called. `device_printf` does not take any of our locks; it's safe under our mutex.
- `sc->heartbeat_interval_ms` is read under the lock. Safe.
- `callout_reset` is called to re-arm. The callout API requires the callout's lock to be held when calling `callout_reset`; we have it.

Every operation in the callback respects the lock contract. The kernel will release `sc->mtx` after the callback returns.

A specific check: the callback does *not* call anything that may sleep. `device_printf` does not sleep. `cbuf_used` does not sleep. `counter_u64_fetch` does not sleep. `callout_reset` does not sleep. The callback respects the mutex's no-sleep convention.

If we accidentally added a sleep, WITNESS would catch it on a debug kernel: “sleeping thread (pid X) owns a non-sleepable lock” or similar. The lesson: trust the kernel to enforce the rules; just keep the callback short.

13.6.10 What Happens If Two Callouts Share a Lock

A single lock can be the interlock for many callouts. Consider:

```
callout_init_mtx(&sc->heartbeat_co, &sc->mtx, 0);
callout_init_mtx(&sc->watchdog_co, &sc->mtx, 0);
callout_init_mtx(&sc->tick_source_co, &sc->mtx, 0);
```

Three callouts, all using `sc->mtx`. When any one of them fires, the kernel acquires `sc->mtx` and runs the callback. While that callback is running, the lock is held; no other callback (or other thread acquiring `sc->mtx`) can proceed.

This is the right pattern: the data-path mutex protects all per-softc state, and any callout that needs to read or modify that state shares the same lock. The serialization is automatic and free.

The downside: if the heartbeat callback is slow, it delays the watchdog callback. Keep callbacks short.

13.6.11 What If the Callback Is Currently Firing When You Call `callout_reset`?

A subtle but important question: what happens if the callback is in the middle of executing on one CPU, and you call `callout_reset` on another CPU to reschedule it?

The kernel handles this case correctly. Let us walk through it.

The callback is firing on CPU 0. It holds `sc->mtx` (the kernel acquired it before calling). On CPU 1, you call `callout_reset(&sc->heartbeat_co, hz, fn, arg)` (perhaps because the user changed the interval). The callout API requires the caller to hold the same lock the callout uses; you do, on CPU 1.

But CPU 0 is already inside the callback, holding `sc->mtx`. Therefore CPU 1 cannot have just acquired it. Either CPU 1 acquired the lock long before CPU 0 got it (in which case CPU 0 is currently blocked waiting for the lock and is not in the callback), or CPU 1 is somehow about to acquire the lock and CPU 0 is about to release it.

The kernel handles the case correctly via the same mechanism it uses for ordinary `mtx_lock` synchronization. There is exactly one holder of `sc->mtx` at any given instant. If CPU 0 is firing, CPU 1's `callout_reset` is blocked waiting for the lock. When CPU 0's callback finishes and the kernel releases the lock, CPU 1 acquires the lock and proceeds with the reschedule. The callout is now scheduled for the new deadline.

If the callback re-armed itself before CPU 0 released the lock (the periodic pattern), the callout is currently pending. CPU 1's `callout_reset` cancels the pending and replaces with the new schedule. The return value is 1 (cancelled).

If the callback did not re-arm (one-shot, or interval was 0), the callout is idle. CPU 1's `callout_reset` schedules it. The return value is 0 (no previous schedule cancelled).

Either way, the result is correct: after `callout_reset` returns, the callout is scheduled for the new deadline, with the new function and argument.

13.6.12 What If the Callback Is Currently Firing When You Call `callout_stop`?

Similar question: callback firing on CPU 0, caller on CPU 1 wants to cancel.

CPU 1 calls `callout_stop`. It needs to hold the callout's lock; it does. CPU 0 is firing the callback while holding the same lock; CPU 1's lock acquisition blocks. When CPU 0's callback returns and releases the lock, CPU 1 acquires it.

At this point, the callback may have re-armed (if it was a periodic). `callout_stop` cancels the pending schedule. Return value is 1.

If the callback did not re-arm, the callout is idle. `callout_stop` is a no-op. Return value is 0.

After `callout_stop` returns, the callout will not fire again unless something else schedules it. Importantly, the callback that was running on CPU 0 has *already finished* by the time `callout_stop` returns; the lock was held through the duration. So `callout_stop` does effectively wait for the in-flight callback, but only because of the lock-acquisition wait, not because of any explicit waiting in the callout subsystem.

This is why `callout_stop` is safe to use in normal driver operation when you hold the lock, and why `callout_drain` is needed only when you are about to free the surrounding state (where you cannot hold the lock during the wait).

13.6.13 `callout_stop` From a Context Without the Lock

What if you call `callout_stop` without holding the callout's lock? The kernel's `_callout_stop_safe` function will detect the missing lock and assert (under INVARIANTS). On a non-INVARIANTS kernel, the call may produce incorrect results or race conditions.

The rule: when calling `callout_stop` or `callout_reset`, you must hold the same lock the callout was initialized with. The kernel enforces this; a violation is a WITNESS warning or an INVARIANTS panic.

For Chapter 13 we always hold `sc->mtx` when calling `callout_reset` or `callout_stop` from `sysctl` handlers. The detach path is the exception: it drops the lock before calling `callout_drain`. `callout_drain` does not require the lock to be held; in fact it requires it to *not* be held.

13.6.14 A Pattern: Conditional Re-Arm

A useful pattern for periodic callouts: only re-arm if some condition is true. In our heartbeat:

```
interval = sc->heartbeat_interval_ms;
if (interval > 0)
    callout_reset(&sc->heartbeat_co, ..., myfirst_heartbeat, sc);
```

The conditional re-arm gives the user fine control over the periodic firing. A user who sets `interval_ms = 0` disables the heartbeat at the next firing. The callback exits without re-arming; the callout becomes idle.

A more elaborate version: re-arm at variable intervals based on activity. A heartbeat that fires more often when the buffer is busy and less often when it is idle:

```
if (cbuf_used(&sc->cb) > 0)
    interval = sc->heartbeat_busy_interval_ms; /* short */
else
    interval = sc->heartbeat_idle_interval_ms; /* long */

if (interval > 0)
    callout_reset(&sc->heartbeat_co, ..., myfirst_heartbeat, sc);
```

The variable interval lets the heartbeat sample the device adaptively. When activity is high, it fires often (catching state changes quickly); when activity is low, it fires rarely (saving CPU and log space).

13.6.15 Wrapping Up Section 5

The lock contract is the heart of `callout(9)`. The kernel acquires the registered lock before each firing, runs your callback, and releases the lock after. This serializes the callback against other holders of the lock and eliminates a class of race that would otherwise require explicit handling. The rules inside the callback are the same as the lock's normal rules: for an `MTX_DEF` mutex, no sleeping, no `uiomove`, no `malloc(M_WAITOK)`. The callback should be short; if it needs to do long work, defer to a taskqueue (Chapter 16) or a kernel thread.

Reschedule and stop work correctly even when the callback is firing on another CPU; the lock-acquisition mechanism ensures atomicity. The conditional re-arm pattern (re-arm only if some condition is true) is the natural way to give a periodic callout a graceful disable path.

Section 6 deals with the corollary of all this: at unload time, you must not free the surrounding state while a callback is in progress or pending. `callout_drain` is the tool, and the unload race is the problem it solves.

13.7 Section 6: Timer Cleanup and Resource Management

Every callout has a destruction problem. Between the moment you decide to remove the driver and the moment the surrounding memory is freed, you have to make sure no callback is running and no callback is scheduled to run. If a callback fires after the memory is freed, the kernel crashes. If a callback is running when you free the memory, the kernel crashes. The crash is reliable, immediate, and fatal; it is the kind of bug that hangs the test machine and is hard to debug because the backtrace points at code that has already been freed.

`callout(9)` provides the tools to solve this cleanly: `callout_stop` for normal cancellation, `callout_drain` for teardown, and `callout_async_drain` for the rare cases where you want to schedule cleanup without blocking. This section walks through each, names the unload race precisely, and presents the standard pattern for safe driver detach.

13.7.1 The Unload Race

Imagine the driver as Stage 1 of Chapter 13 (heartbeat enabled, `kldunload` called). Without `callout_drain`, the sequence might be:

1. User runs `kldunload myfirst`.
2. The kernel calls `myfirst_detach`.
3. `myfirst_detach` clears `is_attached`, broadcasts `cvs`, drops the mutex, and calls `mtx_destroy(&sc->mtx)`.
4. The driver module is unloaded; the memory containing `sc->mtx`, `sc->heartbeat_co`, and `myfirst_heartbeat`'s code is freed.
5. The hardware-clock interrupt fires, the callout subsystem walks the wheel, finds `sc->heartbeat_co` (still on the wheel because we never cancelled it), and calls `myfirst_heartbeat` with `sc` as the argument.
6. `myfirst_heartbeat` is no longer in memory. The kernel jumps to a now-invalid address. Panic.

The race is not theoretical. Even if step 5 happens microseconds after step 4, the kernel still crashes. The window is small but non-zero.

The cure is to ensure that by step 4, no callout is pending and no callback is in flight. Two actions:

- **Cancel pending callouts.** If the callout is on the wheel, remove it. `callout_stop` does this.
- **Wait for in-flight callbacks.** If a callback is currently running on another CPU, wait for it to return. `callout_drain` does this.

`callout_drain` does both: it cancels pending and waits for in-flight. It is what you call at detach time.

13.7.2 `callout_stop` vs `callout_drain`

The distinction is whether the call waits.

`callout_stop`: cancels pending, returns immediately. Does not wait for an in-flight callback. Returns 1 if the callout was pending and got cancelled; 0 otherwise.

`callout_drain`: cancels pending, *and* waits for any in-flight callback to return before itself returning. Returns 1 if the callout was pending and got cancelled; 0 otherwise. After `callout_drain` returns, the callout is guaranteed to be idle.

Use `callout_stop` in normal driver operation when you want to cancel the timer because the condition that motivated it has been resolved. The watchdog use case: schedule a watchdog at the start of an operation; cancel it (with `callout_stop`) when the operation completes successfully. If the watchdog is already firing on another CPU, `callout_stop` returns and the watchdog will run to completion; that is fine because the watchdog handler will see the operation has completed and do nothing (or take some recovery action that is now unnecessary but harmless).

Use `callout_drain` at detach time, where waiting is required to prevent the unload race. Do not use `callout_stop` at detach time; the callback might be running on another CPU and the surrounding memory could be freed before it returns.

13.7.3 Two Critical Rules for `callout_drain`

`callout_drain` has two rules that are easy to violate.

Rule 1: do not hold the callout's lock when calling `callout_drain`. If the callout is currently executing, the callback is holding the lock (the kernel acquired it for the callback). `callout_drain` waits for the callback to return; the callback returns when its work is done; the work includes the lock being released. If the caller of `callout_drain` is *also* holding the lock, the caller would block waiting for itself to release it. Deadlock.

Rule 2: `callout_drain` may sleep. It waits on a sleep queue for the in-flight callback to finish. Therefore `callout_drain` is only legal in contexts where sleeping is allowed: process context (typical detach path) or kernel-thread context. Not interrupt context. Not while holding a spin lock. Not while holding any other non-sleepable lock.

These rules together imply that the standard detach path drops `sc->mtx` (and any other non-sleepable lock) before calling `callout_drain`. The chapter's detach pattern follows this:

```
MYFIRST_LOCK(sc);
sc->is_attached = 0;
cv_broadcast(&sc->data_cv);
cv_broadcast(&sc->room_cv);
MYFIRST_UNLOCK(sc);    /* drop the mutex before draining */

seldrain(&sc->rsl);
seldrain(&sc->wsl);

callout_drain(&sc->heartbeat_co);    /* now safe to call */
```

The mutex is dropped after clearing `is_attached`. The `callout_drain` runs without the mutex held; it is free to wait on a sleep queue. Any callback that fires during the drain sees `is_attached == 0` and exits without re-arming. After the drain, the callout is idle.

13.7.4 The `is_attached` Pattern, Revisited

In Chapter 12 we used `is_attached` as a signal to cv waiters: “the device is going away; return `ENXIO`”. In Chapter 13 we use it for the same purpose with callouts: “the device is going away; do not re-arm”.

The pattern is identical:

```
static void
myfirst_some_callback(void *arg)
{
    struct myfirst_softc *sc = arg;

    MYFIRST_ASSERT(sc);

    if (!sc->is_attached)
```

```

        return; /* device going away; do not re-arm */

/* ... do the work ... */

/* re-arm if periodic */
if (some_condition)
    callout_reset(&sc->some_co, ticks, myfirst_some_callback, sc);
}

```

The check is at the top, before any work. If `is_attached == 0`, the callback exits immediately without doing work and without re-arming. The drain in detach will see the callout idle (no pending firing) and complete cleanly.

A subtle point: the check happens *under the lock* (the kernel acquired it for us). The detach path clears `is_attached` *under the lock*. So the callback always sees the current value of `is_attached`; there is no race. This is the same property we relied on in Chapter 12 for cv waiters.

13.7.5 Why Not `callout_stop` Instead?

A natural question: instead of `callout_drain`, why not `callout_stop` followed by some manual wait?

The implementation of `callout_drain` (in `_callout_stop_safe` in `/usr/src/sys/kern/kern_timeout.c`) does exactly that, but inside the kernel where it can use internal sleep queues without exposing them. Trying to do the same thing in driver code is fragile: you would need to know whether the callback is currently running, which you cannot tell from outside without inspecting kernel-private fields.

Just call `callout_drain`. It is what the API is for.

13.7.6 `callout_async_drain`

For the rare case where you want to drain without blocking, the kernel offers `callout_async_drain`:

```
#define callout_async_drain(c, d) _callout_stop_safe(c, 0, d)
```

It cancels pending and arranges for a “drain done” callback (the `d` function pointer) to be called when the in-flight callback finishes. The caller does not block; control returns immediately. Useful in contexts where you cannot sleep but need to know when the drain has completed.

For the chapter’s purposes, `callout_async_drain` is overkill. We do detach in process context where blocking is fine. We mention it because you will see it in some real-driver source.

13.7.7 The Standard Detach Pattern with Timers

Putting it all together, the canonical detach pattern for a driver with one or more callouts:

Reading this example. The listing below is a composite view of the canonical `callout(9)` teardown sequence, distilled from real drivers such as `/usr/src/sys/dev/re/if_re.c` (where `callout_drain(&sc->rl_stat_callout)` runs at detach time) and `/usr/src/sys/dev/watchdog/watchdog.c` (where two callouts are drained in sequence). We have kept the phase ordering, the mandatory `callout_drain()` calls, and the lock discipline intact; a production driver adds per-device bookkeeping that the real detach function interleaves with each step. Every symbol the listing names, from `callout_drain` to `seldrain` to `mtx_destroy`, is a real FreeBSD API; the `myfirst_softc` fields are this chapter's evolving driver.

```
static int
myfirst_detach(device_t dev)
{
    struct myfirst_softc *sc = device_get_softc(dev);

    /* 1. Refuse detach if the device is in use. */
    MYFIRST_LOCK(sc);
    if (sc->active_fhs > 0) {
        MYFIRST_UNLOCK(sc);
        return (EBUSY);
    }

    /* 2. Mark the device as going away. */
    sc->is_attached = 0;
    cv_broadcast(&sc->data_cv);
    cv_broadcast(&sc->room_cv);
    MYFIRST_UNLOCK(sc);

    /* 3. Drain the selinfo readiness machinery. */
    seldrain(&sc->rsel);
    seldrain(&sc->wsel);

    /* 4. Drain every callout. Each takes its own line. */
    callout_drain(&sc->heartbeat_co);
    callout_drain(&sc->watchdog_co);
    callout_drain(&sc->tick_source_co);

    /* 5. Destroy cdevs (no new opens after this). */
    if (sc->cdev_alias != NULL) {
        destroy_dev(sc->cdev_alias);
        sc->cdev_alias = NULL;
    }
    if (sc->cdev != NULL) {
        destroy_dev(sc->cdev);
        sc->cdev = NULL;
    }
}
```



```

    }

    /* 6. Free other resources. */
    sysctl_ctx_free(&sc->sysctl_ctx);
    cbuf_destroy(&sc->cb);
    counter_u64_free(sc->bytes_read);
    counter_u64_free(sc->bytes_written);

    /* 7. Destroy primitives in reverse acquisition order:
     *   cvs first, then sx, then mutex. */
    cv_destroy(&sc->data_cv);
    cv_destroy(&sc->room_cv);
    sx_destroy(&sc->cfg_sx);
    mtx_destroy(&sc->mtx);

    return (0);
}

```

Seven phases. Each one is a hard requirement. Let us walk through them.

Phase 1: refuse detach while the device is in use (`active_fhs > 0`). Without this, a user with the device open could close their descriptor in the middle of detach, hitting code paths that no longer have valid state.

Phase 2: mark the device as going away. The `is_attached` flag is the signal to every blocked or future code path that the device is being removed. The cv broadcasts wake any cv waiters; they re-check `is_attached` and exit with `ENXIO`. The lock is held during this phase to make the change atomic against any thread that just entered a handler.

Phase 3: drain `selinfo`. This ensures that `selrecord(9)` and `selwakeup(9)` callers no longer reference the device's `selinfo` structures.

Phase 4: drain every callout. Each `callout_drain` cancels pending and waits for in-flight. The mutex is dropped before the first drain (it was dropped at the end of phase 2). After phase 4, no callout can be running.

Phase 5: destroy cdevs. After this, no new `open(2)` can reach the driver. (The ones that snuck in just before would have already been refused in phase 1, but that's the safety net.)

Phase 6: free auxiliary resources (sysctl context, cbuf, counters).

Phase 7: destroy primitives in reverse order. The order matters for the same reason discussed in Chapter 12: cvs use the mutex as their interlock; if we destroy the mutex first, a callback in the middle of releasing the mutex would crash.

This is a lot. It is also what every driver has to do if it has callouts and primitives. The companion source for Chapter 13 (`stage4-final/myfirst.c`) follows this pattern exactly.

13.7.8 A Note on Kernel-Module Unload

`kldunload myfirst` triggers the detach path through the kernel's module-event handling. The `MOD_UNLOAD` event causes the kernel to call the driver's detach function. If the detach function returns an error (typically

EBUSY), the unload fails and the module remains loaded.

The standard pattern we just walked through returns EBUSY if `active_fhs > 0`. A user who wants to unload the driver must first close every open descriptor. From a shell:

```
# List processes holding the device open.
$ fstat | grep myfirst
USER      CMD          PID    FD      ... NAME
root      cat           12345   3       ... /dev/myfirst
$ kill 12345
$ kldunload myfirst
```

This is conventional UNIX behavior; the user is expected to close descriptors before unloading. The driver enforces it.

13.7.9 Initializing After Drain

A subtle point: after `callout_drain`, the callout is idle but is *not* in the same state as a freshly-initialized callout. The `c_func` and `c_arg` fields still point at the last callback and argument, in case a later `callout_schedule` wants to reuse them. The internal flags are cleared.

If you wanted to reuse the same `struct callout` for a different purpose (different lock, different callback signature), you would need to call `callout_init_mtx` (or one of the variants) again to re-initialize. In the detach path, we never re-init; the surrounding memory is about to be freed. The state at drain time is sufficient.

13.7.10 A Worked Walkthrough: Catching the Unload Race in DDB

To make the unload race visceral, walk through what happens when a careless driver omits `callout_drain` and the next callout firing crashes the kernel.

Imagine a buggy driver that disables the heartbeat sysctl in detach but does not call `callout_drain`. The detach path looks like this:

```
static int
buggy_detach(device_t dev)
{
    struct myfirst_softc *sc = device_get_softc(dev);

    MYFIRST_LOCK(sc);
    sc->is_attached = 0;
    sc->heartbeat_interval_ms = 0; /* hope the callback won't re-arm */
    MYFIRST_UNLOCK(sc);

    /* No callout_drain here! */

    destroy_dev(sc->cdev);
}
```

```
    mtx_destroy(&sc->mtx);  
    return (0);  
}
```

The `is_attached = 0` and `heartbeat_interval_ms = 0` are intended to make the callback exit without re-arming. But:

- The callback might be already in the middle of executing when detach starts. The lock is held by the kernel-acquired path. The detach path's `MYFIRST_LOCK(sc)` blocks until the callback releases the lock. Once the lock is acquired by detach, `is_attached` and `heartbeat_interval_ms` are set. The detach drops the lock. So far so good.
- *But:* the callback that was just running has already entered the re-arm path before checking `interval_ms`. It calls `callout_reset` to schedule the next firing, with the just-cleared `interval_ms` value of 0... no, wait, the callback re-reads `sc->heartbeat_interval_ms`, sees 0, and does not re-arm. OK, that case is safe.
- *Or:* the callback completed cleanly with no re-arm. The callout is now idle. The detach path proceeds. It destroys `sc->mtx` and the surrounding state. Everything seems fine.
- *Then:* a different invocation of the callback starts firing. The callout was not on the wheel (no re-arm), so this should not happen, right?

It can happen if there was a concurrent firing on a different CPU. Imagine: the callout fires on CPU 0 and CPU 1 in close succession. CPU 0 starts the callback (acquires the lock). CPU 1 enters the firing path, tries to acquire the lock, blocks. CPU 0 finishes the callback and re-arms (puts the callout back on the wheel for the next firing). CPU 0 releases the lock. CPU 1 acquires the lock and runs the callback. The callback re-arms. CPU 1 releases the lock.

Now suppose the detach path runs between the CPU-0 release and the CPU-1 acquisition. The detach takes the lock (which is now free), clears the flags, drops the lock. CPU 1 acquires the lock and calls the callback. The callback re-reads the flags, sees the cleared values, and exits without re-arming. OK, still safe.

But now consider: the detach path has destroyed the mutex. CPU 1's callback execution is done. The kernel releases the now-destroyed mutex. The release operation operates on freed memory. Panic.

This is the unload race. The fix is straightforward but absolutely required: call `callout_drain(&sc->heartbeat_co)` after dropping the mutex and before destroying primitives. The drain waits for all in-flight callbacks (on any CPU) to return before it itself returns.

Walk through with the drain in place:

- Detach acquires the lock, clears flags, drops the lock.
- Detach calls `callout_drain(&sc->heartbeat_co)`. The drain notices any in-flight callback and waits.
- All callbacks that were firing return cleanly (they re-read the flags, exit without re-arming).
- The drain returns.
- Detach destroys the `cdev`, then destroys the mutex.
- No callback can be running at this point. No callback can fire later because the wheel does not have the callout.

The drain is the safety net. Skipping it produces a panic that may not happen on every unload, but will happen eventually under load. The drain is mandatory.

13.7.11 What Happens If You Forget Drain on a Production Kernel

A production kernel without `INVARIENTS` or `WITNESS` does not catch the unload race in advance. The first time a callout fires after the freed module's memory has been reused, the kernel reads garbage instructions, jumps to a random location, and crashes with whatever pattern the random bytes happen to produce. The crash backtrace points at code that was never the bug; the actual bug is several seconds in the past, in the detach path that did not drain.

This is exactly why the standard advice is “test on a debug kernel before promoting to production”. `WITNESS` catches some forms of the race (it warns about callbacks being called with non-sleepable locks held in unexpected ways); `INVARIENTS` catches some others (the `mtx_destroy` of a destroyed mutex). The production kernel sees only the panic and the wrong backtrace.

13.7.12 What `callout_drain` Returns

`callout_drain` returns the same value as `callout_stop`: 1 if a pending callout was cancelled, 0 if not. Callers usually do not look at the return value; the function is called for its side effect (waiting for in-flight callbacks to finish).

If you want to be sure that the callout is fully idle after a particular code path completed, the discipline is: call `callout_drain` and ignore the return value. Whether the callout was pending or not, after the drain it is idle.

13.7.13 Detach Order with Multiple Callouts

If your driver has three callouts (heartbeat, watchdog, tick source) and you `callout_drain` each in turn, the total wait time is at most the longest of any single in-flight callback (not the sum). The drains are independent: each waits for its own callback. They can effectively run in parallel because each only blocks on its specific callout.

For the chapter's pseudo-device, callbacks are short (microseconds). The drain time is dominated by the cost of the wakeup on the sleep queue, not by the callback work. In total, all three drains complete in much less than a millisecond, even under load.

For drivers with longer callback work, the wait time can be longer. A watchdog callback that takes 10 ms means the worst-case drain is 10 ms (if you happened to call `callout_drain` while it was firing). Most of the time the callout is idle and the drain is instant. Either way, the drain is bounded; it does not loop indefinitely.

13.7.14 The Same Bug, Different Primitive: A Taskqueue Sketch

The “callback ran after detach” bug is not unique to callouts. Every kernel deferred-work primitive has the same pitfall, and the standard answer is always a drain routine that waits for in-flight callbacks to finish. A brief sibling walkthrough drives the point home without pulling Chapter 13 off-topic.

Suppose a driver enqueues work on a taskqueue instead of using a callout. The softc holds a `struct task` and a `struct taskqueue *`, and something in the driver calls `taskqueue_enqueue(sc->tq, &sc->work)` when work is needed. Now imagine a buggy detach that clears `is_attached` and tears down the softc but forgets to drain the task:

```
static int
buggy_tq_detach(device_t dev)
{
    struct myfirst_softc *sc = device_get_softc(dev);

    MYFIRST_LOCK(sc);
    sc->is_attached = 0;
    MYFIRST_UNLOCK(sc);

    /* No taskqueue_drain here! */

    destroy_dev(sc->cdev);
    free(sc->buf, M_MYFIRST);
    mtx_destroy(&sc->mtx);
    return (0);
}
```

The outcome is the same shape as the callout case. If the task was pending when detach ran, the worker thread pops it after detach has freed `sc->buf` and destroyed `sc->mtx`. The task handler dereferences `sc`, finds stale memory, and either reads garbage or panics on the first locked operation. If the task was already running on another CPU, the worker thread is still inside the handler when detach frees the memory underneath it, with the same ending.

The fix is structurally identical to `callout_drain`:

```
taskqueue_drain(sc->tq, &sc->work);
```

`taskqueue_drain(9)` waits until the specified task is neither pending nor currently executing on any worker thread. After it returns, that task cannot fire again unless something re-enqueues it, which is exactly what detach is trying to prevent by clearing `is_attached` first. For drivers that use many tasks on the same queue, `taskqueue_drain_all(9)` waits for every task currently queued or running on that taskqueue, which is the usual call in a module-unload path where nothing on the queue will be re-enqueued.

The takeaway is not a new rule but a wider one: any deferred-work primitive in the kernel, whether `callout(9)`, `taskqueue(9)`, or the network-stack epoch callbacks you will meet in Part 6, needs a corresponding drain before the memory it reads is freed. Chapter 16 walks through `taskqueue(9)` in depth, including how drain interacts with task enqueue ordering; for now, remember that the mental model is identical. Clear the flag, drop the lock, drain the primitive, destroy the storage. The word changes with the primitive, but the shape of the pattern does not.

13.7.15 Wrapping Up Section 6

The unload race is real. `callout_drain` is the cure. The standard detach pattern is: refuse if busy, clear `is_attached` under the lock, broadcast cvs, drop the lock, drain selinfo, drain every callout, destroy cdevs, free auxiliary resources, destroy primitives in reverse order. Every phase is necessary; skipping any of them creates a race that crashes the kernel under load.

Section 7 puts the framework to work on real timer use cases: watchdogs, debouncing, periodic tick sources.

13.8 Section 7: Use Cases and Extensions for Timed Work

Sections 4 through 6 introduced the heartbeat callout: periodic, lock-aware, drained at teardown. The same pattern handles a wide range of real driver problems with small variations. This section walks through three more callouts that we add to `myfirst`: a watchdog that detects buffer staleness, a tick source that injects synthetic events, and (briefly) a debounce shape used in many hardware drivers. Together with the heartbeat, the four cover the bulk of what driver timers are used for in practice.

Treat this section as a recipe collection. Each subsection is a self-contained pattern that you can lift into other drivers.

13.8.1 Pattern 1: A Watchdog Timer

A watchdog detects a stuck condition and acts on it. The classic shape: schedule a callout at the start of an operation; if the operation completes successfully, cancel the callout; if the callout fires, the operation is presumed stuck and the driver takes recovery action.

For `myfirst`, a useful watchdog is “the buffer has not made progress for too long”. If `cb_used > 0` and the value has not changed for `N` seconds, no reader is draining the buffer. This is unusual; we will log a warning.

Add fields to the `softc`:

```
struct callout      watchdog_co;
int                 watchdog_interval_ms; /* 0 = disabled */
size_t              watchdog_last_used;
```

`watchdog_interval_ms` is a sysctl tunable. `watchdog_last_used` records the value of `cbuf_used` from the previous tick; the next tick compares.

Initialize in `attach`:

```
callout_init_mtx(&sc->watchdog_co, &sc->mtx, 0);
sc->watchdog_interval_ms = 0;
sc->watchdog_last_used = 0;
```

Drain in `detach`:

```
callout_drain(&sc->watchdog_co);
```

The callback:

```
static void
myfirst_watchdog(void *arg)
{
    struct myfirst_softc *sc = arg;
    size_t used;
    int interval;

    MYFIRST_ASSERT(sc);

    if (!sc->is_attached)
        return;

    used = cbuf_used(&sc->cb);
    if (used > 0 && used == sc->watchdog_last_used) {
        device_printf(sc->dev,
            "watchdog: buffer has %zu bytes, no progress in last "
            "interval; reader stuck?\n", used);
    }
    sc->watchdog_last_used = used;

    interval = sc->watchdog_interval_ms;
    if (interval > 0)
        callout_reset(&sc->watchdog_co,
            (interval * hz + 999) / 1000,
            myfirst_watchdog, sc);
}
```

The structure mirrors the heartbeat: assert, check `is_attached`, do work, re-arm if interval is non-zero. The work this time is the staleness check: compare current `cbuf_used` to last recorded; if they match and are non-zero, no progress has been made.

The sysctl handler is symmetric to the heartbeat's:

```
static int
myfirst_sysctl_watchdog_interval_ms(SYSCTL_HANDLER_ARGS)
{
    struct myfirst_softc *sc = arg1;
    int new, old, error;

    old = sc->watchdog_interval_ms;
```

```

    new = old;
    error = sysctl_handle_int(oidp, &new, 0, req);
    if (error || req->newptr == NULL)
        return (error);
    if (new < 0)
        return (EINVAL);

    MYFIRST_LOCK(sc);
    sc->watchdog_interval_ms = new;
    if (new > 0 && old == 0) {
        sc->watchdog_last_used = cbuf_used(&sc->cb);
        callout_reset(&sc->watchdog_co,
            (new * hz + 999) / 1000,
            myfirst_watchdog, sc);
    } else if (new == 0 && old > 0) {
        callout_stop(&sc->watchdog_co);
    }
    MYFIRST_UNLOCK(sc);
    return (0);
}

```

The only addition: when enabling, we initialize `watchdog_last_used` to the current `cbuf_used`, so the first comparison has a sensible baseline.

To test: enable the watchdog with a 2-second interval, write some bytes into the buffer, and do not read them. After two seconds, `dmesg` should show the watchdog warning.

```

$ sysctl -w dev.myfirst.0.watchdog_interval_ms=2000
$ printf 'hello' > /dev/myfirst
$ sleep 5
$ dmesg | tail
myfirst0: watchdog: buffer has 5 bytes, no progress in last interval; reader stuck?
myfirst0: watchdog: buffer has 5 bytes, no progress in last interval; reader stuck?

```

Now drain the buffer:

```

$ cat /dev/myfirst
hello

```

The watchdog stops warning because `cbuf_used` is now zero (the comparison used `> 0` fails).

This is a contrived watchdog. Real watchdogs do more: reset a hardware engine, kill a stuck request, log to the kernel ringbuffer with a specific format that monitoring tools can grep. The shape is the same: detect, act, re-arm.

13.8.2 Pattern 2: A Tick Source for Synthetic Events

A tick source is a callout that periodically generates events as if the hardware did. Useful for drivers that simulate something or that want a stable test workload independent of user-space activity.

For myfirst, a tick source can periodically write a single byte into the cbuf. With the heartbeat enabled, the byte counts will rise visibly without any external producer.

Add fields:

```
struct callout      tick_source_co;
int                 tick_source_interval_ms; /* 0 = disabled */
char                tick_source_byte;        /* the byte to write */
```

Initialize in attach:

```
callout_init_mtx(&sc->tick_source_co, &sc->mtx, 0);
sc->tick_source_interval_ms = 0;
sc->tick_source_byte = 't';
```

Drain in detach:

```
callout_drain(&sc->tick_source_co);
```

The callback:

```
static void
myfirst_tick_source(void *arg)
{
    struct myfirst_softc *sc = arg;
    size_t put;
    int interval;

    MYFIRST_ASSERT(sc);

    if (!sc->is_attached)
        return;

    if (cbuf_free(&sc->cb) > 0) {
        put = cbuf_write(&sc->cb, &sc->tick_source_byte, 1);
        if (put > 0) {
            counter_u64_add(sc->bytes_written, put);
            cv_signal(&sc->data_cv);
            /* selwakeup omitted on purpose: it may sleep
             * and we are inside a callout context with the
```

```

        * mutex held. Defer to a taskqueue if real-time
        * poll(2) wakeups are needed. */
    }

}

interval = sc->tick_source_interval_ms;
if (interval > 0)
    callout_reset(&sc->tick_source_co,
        (interval * hz + 999) / 1000,
        myfirst_tick_source, sc);
}

```

The structure is the same as the heartbeat. The work is different: write one byte to the cbuf, increment the counter, signal data_cv so any reader wakes.

Note the deliberate omission of `se wakeup` from the callback. `se wakeup` may sleep and may take other locks, which is illegal under our mutex. Calling it from a callout context with the mutex held would be a WITNESS violation. The `cv_signal` is enough to wake blocking readers; `poll(2)` waiters will not be woken in real time, but they will pick up the next state change at their normal poll interval. For a real driver that needs immediate `poll(2)` wakeups from a callout, the answer is to defer the `se wakeup` to a taskqueue (Chapter 16). For Chapter 13, omitting it is acceptable.

A sysctl handler enables and disables, mirroring the others:

```

static int
myfirst_sysctl_tick_source_interval_ms(SYSCTL_HANDLER_ARGS)
{
    struct myfirst_softc *sc = arg1;
    int new, old, error;

    old = sc->tick_source_interval_ms;
    new = old;
    error = sysctl_handle_int(oidp, &new, 0, req);
    if (error || req->newptr == NULL)
        return (error);
    if (new < 0)
        return (EINVAL);

    MYFIRST_LOCK(sc);
    sc->tick_source_interval_ms = new;
    if (new > 0 && old == 0)
        callout_reset(&sc->tick_source_co,
            (new * hz + 999) / 1000,
            myfirst_tick_source, sc);
    else if (new == 0 && old > 0)

```

```
        callout_stop(&sc->tick_source_co);
    MYFIRST_UNLOCK(sc);
    return (0);
}
```

To test:

```
$ sysctl -w dev.myfirst.0.tick_source_interval_ms=100
$ cat /dev/myfirst
ttttttttttttttttttttttttttttttttt      # ten 't's per second
^C
$ sysctl -w dev.myfirst.0.tick_source_interval_ms=0
```

The tick source produces ten ‘t’ characters per second, which cat reads and prints. Disable by setting the sysctl back to zero.

13.8.3 Pattern 3: A Debounce Shape

A debounce ignores rapid repeated events. The shape: when an event arrives, check whether a “debounce timer” is already pending; if so, ignore the event; if not, schedule a debounce timer for N milliseconds, and act on the event when the timer fires.

For myfirst, we do not have a hardware event source, so we will not implement a full debounce. The shape, in pseudo-code:

```
static void
some_event_callback(struct myfirst_softc *sc)
{
    MYFIRST_LOCK(sc);
    sc->latest_event_time = ticks;
    if (!callout_pending(&sc->debounce_co)) {
        callout_reset(&sc->debounce_co,
                      DEBOUNCE_DURATION_TICKS,
                      myfirst_debounce_handler, sc);
    }
    MYFIRST_UNLOCK(sc);
}
```

```
static void
myfirst_debounce_handler(void *arg)
{
    struct myfirst_softc *sc = arg;

    MYFIRST_ASSERT(sc);
}
```

```

    if (!sc->is_attached)
        return;

    /* Act on the latest event seen. */
    process_event(sc, sc->latest_event_time);
    /* Do not re-arm; one-shot. */
}

```

When the first event arrives, the debounce timer is scheduled. Subsequent events update the recorded “latest event time” but do not re-schedule the timer (because it is still pending). When the debounce timer fires, the handler processes the most recent event. After the handler returns, the timer is no longer pending; the next event will reschedule.

This is a one-shot pattern, not periodic. The callback does not re-arm. The `callout_pending` check in `some_event_callback` is the gate.

Lab 13.5 walks through implementing a similar debounce as a stretch exercise. The chapter does not add it to `myfirst` because we have no hardware event to debounce, but the shape is one to remember.

13.8.4 Pattern 4: A Retry With Exponential Backoff

A retry-with-backoff shape: an operation fails; schedule a retry after N milliseconds; if the retry also fails, schedule the next retry after $2N$ milliseconds; and so on, capped at some maximum.

For `myfirst`, no operations fail in a way that demands retry. The shape:

```

struct callout      retry_co;
int                 retry_attempt;           /* 0, 1, 2, ... */
int                 retry_base_ms;          /* base interval */
int                 retry_max_attempts;     /* cap */

static void
some_operation_failed(struct myfirst_softc *sc)
{
    int next_delay_ms;

    MYFIRST_LOCK(sc);
    if (sc->retry_attempt < sc->retry_max_attempts) {
        next_delay_ms = sc->retry_base_ms * (1 << sc->retry_attempt);
        callout_reset(&sc->retry_co,
            (next_delay_ms * hz + 999) / 1000,
            myfirst_retry, sc);
        sc->retry_attempt++;
    } else {
        /* Give up. */
    }
}

```

```

        device_printf(sc->dev, "retry: exhausted attempts; failing\n");
        some_failure_action(sc);
    }
    MYFIRST_UNLOCK(sc);
}

static void
myfirst_retry(void *arg)
{
    struct myfirst_softc *sc = arg;

    MYFIRST_ASSERT(sc);
    if (!sc->is_attached)
        return;

    if (some_operation(sc)) {
        /* success */
        sc->retry_attempt = 0;
    } else {
        /* failure: schedule next retry */
        some_operation_failed(sc);
    }
}

```

The callback retries the operation. Success resets the attempt counter. Failure schedules the next retry with an exponentially-increasing delay, capped at `retry_max_attempts`.

This pattern is in many real drivers, particularly storage and network drivers handling transient hardware errors. Chapter 13 does not add it to `myfirst` because we have no failures to retry. The shape is in your toolbox.

13.8.5 Pattern 5: A Deferred Reaper

A deferred reaper is a one-shot callout that frees something after a grace period. Used when an object cannot be freed immediately because some other code path may still hold a reference, but we know that after some time has passed, all references will have drained.

The shape, sketched as pseudo-code (the `some_object` type stands in for whatever deferred-free object your driver actually uses):

```

struct some_object {
    TAILQ_ENTRY(some_object) link;
    /* ... per-object fields ... */
};

TAILQ_HEAD(some_object_list, some_object);

```

```

struct myfirst_softc {
    /* ... existing fields ... */
    struct callout      reaper_co;
    struct some_object_list pending_free;
    /* ... */
};

static void
schedule_free(struct myfirst_softc *sc, struct some_object *obj)
{
    MYFIRST_LOCK(sc);
    TAILQ_INSERT_TAIL(&sc->pending_free, obj, link);
    if (!callout_pending(&sc->reaper_co))
        callout_reset(&sc->reaper_co, hz, myfirst_reaper, sc);
    MYFIRST_UNLOCK(sc);
}

static void
myfirst_reaper(void *arg)
{
    struct myfirst_softc *sc = arg;
    struct some_object *obj, *tmp;

    MYFIRST_ASSERT(sc);
    if (!sc->is_attached)
        return;

    TAILQ_FOREACH_SAFE(obj, &sc->pending_free, link, tmp) {
        TAILQ_REMOVE(&sc->pending_free, obj, link);
        free(obj, M_DEVBUF);
    }

    /* Do not re-arm; new objects scheduled later will re-arm us. */
}

```

The reaper runs once a second (or whatever interval makes sense), frees everything on the pending list, and stops. New scheduling adds to the list and re-arms only if the reaper is not currently pending.

Used in network drivers where receive buffers cannot be freed immediately because the network layer still has references; the buffer is queued for the reaper, which frees it after a grace period.

myfirst does not need this pattern. It is in the toolbox.

13.8.6 Pattern 6: A Polling Loop Replacement

Some hardware does not interrupt for events the driver cares about. A typical example: a sensor that has a status register the driver must check every few milliseconds to learn about new readings. Without callouts, the driver would either spin (waste CPU) or run a kernel thread that sleeps and polls (waste a thread). With callouts, the polling loop is a periodic callback that reads the register, takes appropriate action, and re-arms.

```
static void
myfirst_poll(void *arg)
{
    struct myfirst_softc *sc = arg;
    uint32_t status;
    int interval;

    MYFIRST_ASSERT(sc);
    if (!sc->is_attached)
        return;

    status = bus_read_4(sc->res, REG_STATUS);  /* hypothetical */
    if (status & STATUS_DATA_READY) {
        /* Pull data from the device into the cbuf. */
        myfirst_drain_hardware(sc);
    }
    if (status & STATUS_ERROR) {
        /* Recover from the error. */
        myfirst_handle_error(sc);
    }

    interval = sc->poll_interval_ms;
    if (interval > 0)
        callout_reset(&sc->poll_co,
            (interval * hz + 999) / 1000,
            myfirst_poll, sc);
}
```

The callback reads a hardware register (not implemented in our pseudo-device, but the shape is clear), checks bits, acts, and re-arms. The interval determines how often the driver checks; shorter means more responsive but more CPU. Real polling drivers typically use 1-10 ms intervals when active; longer when idle.

The code comment about `bus_read_4` is a forward reference to Chapter 19, which introduces bus space access. For Chapter 13, treat it as pseudo-code that demonstrates the pattern; the polling logic is what matters.

13.8.7 Pattern 7: A Statistics Window

A periodic callout that takes a snapshot of internal counters at regular intervals and computes per-interval rates. Useful for monitoring; the driver can answer “how many bytes per second am I currently moving?” without the

user having to sample manually.

```

struct myfirst_stats_window {
    uint64_t      last_bytes_read;
    uint64_t      last_bytes_written;
    uint64_t      rate_bytes_read;      /* bytes/sec, latest interval */
    uint64_t      rate_bytes_written;
};

struct myfirst_softc {
    /* ... existing fields ... */
    struct callout      stats_window_co;
    int                stats_window_interval_ms;
    struct myfirst_stats_window      stats_window;
    /* ... */
};

static void
myfirst_stats_window(void *arg)
{
    struct myfirst_softc *sc = arg;
    uint64_t cur_br, cur_bw;
    int interval;

    MYFIRST_ASSERT(sc);
    if (!sc->is_attached)
        return;

    cur_br = counter_u64_fetch(sc->bytes_read);
    cur_bw = counter_u64_fetch(sc->bytes_written);
    interval = sc->stats_window_interval_ms;

    if (interval > 0) {
        /* bytes-per-second over this interval */
        sc->stats_window.rate_bytes_read = (cur_br -
            sc->stats_window.last_bytes_read) * 1000 / interval;
        sc->stats_window.rate_bytes_written = (cur_bw -
            sc->stats_window.last_bytes_written) * 1000 / interval;
    }

    sc->stats_window.last_bytes_read = cur_br;
    sc->stats_window.last_bytes_written = cur_bw;

    if (interval > 0)
        callout_reset(&sc->stats_window_co,

```



```
        (interval * hz + 999) / 1000,  
        myfirst_stats_window, sc);  
}
```

Expose the rates as sysctls. A user can `sysctl dev.myfirst.0.stats.rate_bytes_read` and see the per-interval rate, computed live without having to manually sample and difference.

This pattern is in many monitoring-friendly drivers. The granularity (the interval) is configurable; longer intervals smooth out short bursts; shorter intervals respond more quickly. Choose to match what the user wants to measure.

13.8.8 Pattern 8: A Timed Status Refresh

A periodic callout that refreshes a cached value the rest of the driver reads. Useful when the underlying value is expensive to compute every time but acceptable to be slightly stale.

For our `myfirst`, we do not have an expensive computation to cache. The shape, in pseudo-code:

```
static void  
myfirst_refresh_status(void *arg)  
{  
    struct myfirst_softc *sc = arg;  
  
    MYFIRST_ASSERT(sc);  
    if (!sc->is_attached)  
        return;  
  
    sc->cached_status = expensive_compute(sc);  
    callout_reset(&sc->refresh_co, hz, myfirst_refresh_status, sc);  
}  
  
/* Other code reads sc->cached_status freely; it may be up to 1s stale. */
```

Used in drivers where the computation is expensive (parsing a hardware status table, communicating with a remote subsystem) but the consumer can tolerate stale values. The callback runs periodically and refreshes; the consumer gets the cached value.

`myfirst` does not need this pattern. It is in your toolbox.

13.8.9 Pattern 9: A Periodic Reset

Some hardware needs a periodic reset (a write to a specific register) to keep an internal watchdog from triggering. The pattern:

```
static void
myfirst_periodic_reset(void *arg)
{
    struct myfirst_softc *sc = arg;

    MYFIRST_ASSERT(sc);
    if (!sc->is_attached)
        return;

    bus_write_4(sc->res, REG_KEEPLIVE, KEEPALIVE_VALUE);
    callout_reset(&sc->keepalive_co, hz / 2,
        myfirst_periodic_reset, sc);
}
```

The hardware expects the keepalive write at least every second; we send it every 500 ms to have margin. If we miss a few writes (system load, reschedule), the hardware does not panic.

Used in storage controllers, network controllers, and embedded systems where the device has a per-side watchdog the host driver must satisfy.

13.8.10 Combining Patterns

A driver typically uses several callouts at once. `myfirst` (Stage 4 of this chapter) uses three: heartbeat, watchdog, tick source. Each has its own callout and its own sysctl tunable. They share the same lock (`sc->mtx`), which means only one fires at a time; serialization is automatic.

In a more complex driver, you might have ten or twenty callouts, each for a specific purpose. The pattern scales: each callout has its own struct callout, its own callback, its own sysctl (if user-facing), and its own line in the detach callout_drain block. The disciplines from this chapter (lock-aware init, `is_attached` check, drain at detach) apply to every one of them.

13.8.11 Wrapping Up Section 7

Nine patterns cover most of what driver timers do: heartbeat, watchdog, debounce, retry-with-backoff, deferred reaper, statistics rollover, polling loop, statistics window, timed status refresh, and periodic reset. Each is a small variation on the periodic or one-shot shape. The disciplines from Sections 4 through 6 (lock-aware initialization, `is_attached` check, drain at detach) apply uniformly. A driver that adds new timers follows the same recipe; the surface area expands without the maintenance burden growing.

Section 8 closes the chapter with the housekeeping pass: documentation, version bump, regression test, pre-commit checklist.

13.9 Section 8: Refactoring and Versioning Your Timer-Enhanced Driver

The driver now has three callouts (heartbeat, watchdog, tick source), four sysctls (the three intervals plus the existing config), and a detach path that drains every callout safely. The remaining work is the housekeeping pass: tidying the source for clarity, updating the documentation, bumping the version, running static analysis, and verifying the regression suite passes.

This section follows the same shape as the equivalent sections in Chapters 11 and 12. None of it is glamorous. All of it is what makes the difference between a driver delivered once and a driver that keeps working as it grows.

13.9.1 Cleaning Up the Source

After this chapter's focused additions, three small reorganizations are worth doing.

Group callout-related code. Move all the callout callbacks (myfirst_heartbeat, myfirst_watchdog, myfirst_tick_source) into a single section of the source file, after the wait helpers and before the cdevsw handlers. Move the corresponding sysctl handlers next to them. The compiler does not care about ordering; the reader does.

Standardise macro vocabulary. Add a small set of macros to make callout operations consistent across the driver. The existing pattern with MYFIRST_LOCK and MYFIRST_CFG_* extends naturally:

```
#define MYFIRST_CO_INIT(sc, co)  callout_init_mtx((co), &(sc)->mtx, 0)
#define MYFIRST_CO_DRAIN(co)    callout_drain((co))
```

The MYFIRST_CO_INIT macro takes sc explicitly so it works in any function, not just those where a local variable named sc happens to be in scope. MYFIRST_CO_DRAIN only needs the callout itself, because draining does not require the softc.

The macros are thin, but they document the convention: every callout in the driver uses sc->mtx as its lock and is drained at detach. A future maintainer who adds a callout sees the macro and knows the rule.

Comment the detach order. The detach function is short on its own but the order of operations is critical. Add comments at each phase:

```
static int
myfirst_detach(device_t dev)
{
    struct myfirst_softc *sc = device_get_softc(dev);

    /* Phase 1: refuse if in use. */
    MYFIRST_LOCK(sc);
    if (sc->active_fhs > 0) {
        MYFIRST_UNLOCK(sc);
        return (EBUSY);
    }
}
```

```
/* Phase 2: signal "going away" to all waiters and callbacks. */
sc->is_attached = 0;
cv_broadcast(&sc->data_cv);
cv_broadcast(&sc->room_cv);
MYFIRST_UNLOCK(sc);

/* Phase 3: drain selinfo. */
seldrain(&sc->rsel);
seldrain(&sc->wsel);

/* Phase 4: drain every callout (no lock held; safe to sleep). */
MYFIRST_CO_DRAIN(&sc->heartbeat_co);
MYFIRST_CO_DRAIN(&sc->watchdog_co);
MYFIRST_CO_DRAIN(&sc->tick_source_co);

/* Phase 5: destroy cdevs (no new opens after this). */
if (sc->cdev_alias != NULL) {
    destroy_dev(sc->cdev_alias);
    sc->cdev_alias = NULL;
}
if (sc->cdev != NULL) {
    destroy_dev(sc->cdev);
    sc->cdev = NULL;
}

/* Phase 6: free auxiliary resources. */
sysctl_ctx_free(&sc->sysctl_ctx);
cbuf_destroy(&sc->cb);
counter_u64_free(sc->bytes_read);
counter_u64_free(sc->bytes_written);

/* Phase 7: destroy primitives in reverse order. */
cv_destroy(&sc->data_cv);
cv_destroy(&sc->room_cv);
sx_destroy(&sc->cfg_sx);
mtx_destroy(&sc->mtx);

return (0);
}
```

In attach, the matching initialization uses the two-argument form so that `sc` is passed explicitly:

```
MYFIRST_CO_INIT(sc, &sc->heartbeat_co);
MYFIRST_CO_INIT(sc, &sc->watchdog_co);
MYFIRST_CO_INIT(sc, &sc->tick_source_co);
```

The comments turn the function from a sequence of seemingly-arbitrary calls into a documented checklist.

13.9.2 Updating LOCKING.md

The Chapter 12 LOCKING.md documented three primitives, two lock classes, and one lock order. Chapter 13 adds three callouts. The new sections to add:

Callouts Owned by This Driver

sc->heartbeat_co (callout(9), MYFIRST_CO_INIT)

Lock: sc->mtx (registered via callout_init_mtx).
 Callback: myfirst_heartbeat.
 Behaviour: periodic; re-arms itself at the end of each firing if
 sc->heartbeat_interval_ms > 0.
 Started by: the heartbeat sysctl handler (transition 0 -> non-zero).
 Stopped by: the heartbeat sysctl handler (transition non-zero -> 0)
 via callout_stop, and by myfirst_detach via callout_drain.
 Lifetime: initialised in attach via MYFIRST_CO_INIT; drained in detach
 via MYFIRST_CO_DRAIN.

sc->watchdog_co (callout(9), MYFIRST_CO_INIT)

Lock: sc->mtx.
 Callback: myfirst_watchdog.
 Behaviour: periodic; emits a warning if cb_used has not changed and
 is non-zero between firings.
 Started/stopped: via the watchdog sysctl handler and detach, parallel
 to the heartbeat.

sc->tick_source_co (callout(9), MYFIRST_CO_INIT)

Lock: sc->mtx.
 Callback: myfirst_tick_source.
 Behaviour: periodic; injects a single byte into the cbuf each firing
 if there is room.
 Started/stopped: via the tick_source sysctl handler and detach,
 parallel to the heartbeat.

Callout Discipline

1. Every callout uses `sc->mtx` as its lock via `callout_init_mtx`.
2. Every callout callback asserts `MYFIRST_ASSERT(sc)` at entry.
3. Every callout callback checks `!sc->is_attached` at entry and returns early without re-arming.
4. The detach path clears `sc->is_attached` under `sc->mtx`, broadcasts both cvs, drops the mutex, and then calls `callout_drain` on every callout.
5. `callout_stop` is used to cancel pending callouts in normal driver operation (sysctl handlers); `callout_drain` is used at detach.
6. NEVER call `selwakeup`, `uiomove`, `copyin`, `copyout`, `malloc(M_WAITOK)`, or any sleeping primitive from a callout callback. The mutex is held during the callback, and these calls would violate the sleep-with-mutex rule.

History (extended)

- 0.7-timers (Chapter 13): added heartbeat, watchdog, and tick-source callouts; documented callout discipline; standardised callout detach pattern.
- 0.6-sync (Chapter 12, Stage 4): combined version with cv channels, bounded reads, sx-protected configuration, reset sysctl.
- ... (earlier history as before) ...

Add this to the existing `LOCKING.md` rather than replacing the existing content. The new sections sit alongside the existing “Locks Owned by This Driver”, “Lock Order”, “Locking Discipline”, and so on.

13.9.3 Bumping the Version

Update the version string:

```
#define MYFIRST_VERSION "0.7-timers"
```

Update the changelog entry:

0.7-timers (Chapter 13)

- Added struct `callout heartbeat_co`, `watchdog_co`, `tick_source_co` to the softc.
- Added sysctls `dev.myfirst.<unit>.heartbeat_interval_ms`, `watchdog_interval_ms`, `tick_source_interval_ms`.
- Added callbacks `myfirst_heartbeat`, `myfirst_watchdog`, `myfirst_tick_source`, each lock-aware via `callout_init_mtx`.

- Updated detach to drain every callout under the documented seven-phase pattern.
- Added MYFIRST_CO_INIT and MYFIRST_CO_DRAIN macros for callout init and teardown.
- Updated LOCKING.md with a Callouts section and callout discipline rules.
- Updated regression script to include callout tests.

13.9.4 Updating the README

Two new features in the README:

Features (additions)

- Callout-based heartbeat that periodically logs cbuf usage and byte counts.
- Callout-based watchdog that detects stalled buffer drainage.
- Callout-based tick source that injects synthetic data for testing.

Configuration (additions)

- dev.myfirst.<unit>.heartbeat_interval_ms: periodic heartbeat in milliseconds (0 = disabled).
- dev.myfirst.<unit>.watchdog_interval_ms: watchdog interval in milliseconds (0 = disabled).
- dev.myfirst.<unit>.tick_source_interval_ms: tick-source interval in milliseconds (0 = disabled).

13.9.5 Running Static Analysis

Run `clang --analyze` against the new code. The exact flags depend on your kernel configuration; the same recipe the Chapter 11 regression section used still works, with the added knowledge that callout init macros expand into function calls that `clang` can now see through:

```
$ make WARNS=6 clean all
$ clang --analyze -D_KERNEL -DKLD_MODULE \
  -I/usr/src/sys -I/usr/src/sys/contrib/ck/include \
  -fno-builtin -nostdinc myfirst.c
```

Triage the output as before. A few false positives may appear around the callout init macros (the analyser does not always track the lock association embedded in `_callout_init_lock`); document each so the next maintainer does not re-triage them.

13.9.6 Running the Regression Suite

The Chapter 12 regression script extends naturally. Two design points are worth noting before the script: each subtest clears the kernel message buffer with `dmesg -c` so that `grep -c` only counts lines produced *during* that subtest; and reads use `dd` with a fixed count= rather than `cat`, so an unexpectedly empty buffer cannot make the script hang.

```
#!/bin/sh
# regression.sh: full Chapter 13 regression.

set -eu

die() { echo "FAIL: $" ">&2; exit 1; }
ok()  { echo "PASS: $*"; }

[ $(id -u) -eq 0 ] || die "must run as root"
kldstat | grep -q myfirst && kldunload myfirst
[ -f ./myfirst.ko ] || die "myfirst.ko not built; run make first"

# Clear any stale dmesg contents so per-subtest greps are scoped.
dmesg -c >/dev/null

kldload ./myfirst.ko
trap 'kldunload myfirst 2>/dev/null || true' EXIT

sleep 1
[ -c /dev/myfirst ] || die "device node not created"
ok "load"

# Chapter 7-12 tests (abbreviated; see prior chapters' scripts).
printf 'hello' > /dev/myfirst || die "write failed"
# dd with bs and count avoids blocking if the buffer is shorter
# than expected; if the read returns short, the test still proceeds.
ROUND=$(dd if=/dev/myfirst bs=5 count=1 2>/dev/null)
[ "$ROUND" = "hello" ] || die "round-trip mismatch (got '$ROUND')"
ok "round-trip"

# Chapter 13-specific tests. Each subtest clears dmesg first so the
# subsequent grep counts only the lines produced during that test.

# Heartbeat enable/disable.
dmesg -c >/dev/null
sysctl -w dev.myfirst.0.heartbeat_interval_ms=100 >/dev/null
sleep 1
HB_LINES=$(dmesg | grep -c "heartbeat:" || true)
```



```

[ "$HB_LINES" -ge 5 ] || die "expected >=5 heartbeat lines, got $HB_LINES"
sysctl -w dev.myfirst.0.heartbeat_interval_ms=0 >/dev/null
ok "heartbeat enable/disable"

# Watchdog: enable, write, wait, expect warning, then drain via dd
# (not cat, which would block once the 7 bytes are gone).
dmesg -c >/dev/null
sysctl -w dev.myfirst.0.watchdog_interval_ms=500 >/dev/null
printf 'wd_test' > /dev/myfirst
sleep 2
WD_LINES=$(dmesg | grep -c "watchdog:" || true)
[ "$WD_LINES" -ge 1 ] || die "expected >=1 watchdog line, got $WD_LINES"
sysctl -w dev.myfirst.0.watchdog_interval_ms=0 >/dev/null
dd if=/dev/myfirst bs=7 count=1 of=/dev/null 2>/dev/null # drain
ok "watchdog warns on stuck buffer"

# Tick source: enable, read, expect synthetic bytes.
dmesg -c >/dev/null
sysctl -w dev.myfirst.0.tick_source_interval_ms=50 >/dev/null
TS_BYTES=$(dd if=/dev/myfirst bs=1 count=10 2>/dev/null | wc -c | tr -d ' ')
[ "$TS_BYTES" -eq 10 ] || die "expected 10 tick bytes, got $TS_BYTES"
sysctl -w dev.myfirst.0.tick_source_interval_ms=0 >/dev/null
ok "tick source produces bytes"

# Detach with callouts active. The trap will not fire after the
# explicit unload because the unload succeeds.
sysctl -w dev.myfirst.0.heartbeat_interval_ms=100 >/dev/null
sysctl -w dev.myfirst.0.tick_source_interval_ms=100 >/dev/null
sleep 1 # allow each callout to fire at least a few times
dmesg -c >/dev/null
kldunload myfirst
trap - EXIT # the driver is now unloaded
ok "detach with active callouts"

# WITNESS check. Confined to events since the unload above.
WITNESS_HITS=$(dmesg | grep -ci "witness\|lor" || true)
if [ "$WITNESS_HITS" -gt 0 ]; then
    die "WITNESS warnings detected ($WITNESS_HITS lines)"
fi
ok "witness clean"

echo "ALL TESTS PASSED"

```

A few notes on portability and robustness.

The `dmesg -c` calls flush the kernel message buffer between subtests; on FreeBSD `dmesg -c` is documented to clear the buffer after printing it. Without these, a test that runs after the heartbeat subtest could see heartbeat lines from earlier in the run and miscount.

`dd` is used in place of `cat` for the round-trip and watchdog-drain reads. `cat` blocks until EOF, which a character device never returns; `dd` exits after `count=` blocks have been read. The driver is blocking-by-default, so an over-eager `cat` on an empty buffer would simply hang and break the script.

The detach step does not call `kldload` again at the end, because the only later test (`witness clean`) does not need the driver loaded. The trap is cleared after the successful unload so `EXIT` does not try to unload an already-unloaded module.

A green run after each commit is the minimum bar. A green run on a WITNESS kernel after the long-duration composite stress (Chapter 12 plus the Chapter 13 callouts active) is the higher bar.

13.9.7 Pre-Commit Checklist

The Chapter 12 checklist gets three new items for Chapter 13:

1. Have I updated `LOCKING.md` with any new callouts, intervals, or detach changes?
2. Have I run the full regression suite on a WITNESS kernel?
3. Have I run the long-duration composite stress for at least 30 minutes with all timers enabled?
4. Have I run `clang --analyze` and triaged every new warning?
5. Have I added a `MYFIRST_ASSERT(sc)` and `if (!sc->is_attached) return;` to every new callout callback?
6. Have I bumped the version string and updated `CHANGELOG.md`?
7. Have I verified that the test kit builds and runs?
8. Have I checked that every `cv` has both a signaller and a documented condition?
9. Have I checked that every `sx_xlock` has a paired `sx_xunlock` on every code path?
10. **(New)** Have I added a `MYFIRST_CO_DRAIN` for every new callout in the detach path?
11. **(New)** Have I confirmed no callout callback calls `se wakeup`, `uiomove`, or any sleeping primitive?
12. **(New)** Have I verified that disabling a callout via its `sysctl` actually stops the periodic firing?

The new items capture the most common Chapter 13 mistakes. A callout that is initialised but never drained is a `kldunload` crash waiting to happen. A callback that calls a sleeping function is a WITNESS warning waiting to happen. A `sysctl` that fails to stop a callout is a confusing user experience.

13.9.8 A Note on Backward Compatibility

A reasonable concern: the Chapter 13 driver adds three new `sysctls`. Will existing scripts that interact with `myfirst` (perhaps the Chapter 12 stress kit) break?

The answer is no, for two reasons.

First, the new `sysctls` all default to disabled (`interval = 0`). The driver's behaviour is unchanged unless the user enables one of them.

Second, the Chapter 12 `sysctls` (`debug_level`, `soft_byte_limit`, `nickname`, `read_timeout_ms`, `write_timeout_ms`) and stats (`cb_used`, `cb_free`, `bytes_read`, `bytes_written`) are unchanged. Existing scripts read and write the same values. The Chapter 13 additions are purely additive.

This is the discipline of *non-breaking changes*: when you add a feature, do not change the meaning of existing features. The cost is small (think about the change before making it); the benefit is that existing users see no regression.

For Chapter 13, the heartbeat, watchdog, and tick source are all opt-in. A user who does not know about Chapter 13 sees the same driver as before. A user who reads the chapter and enables one of the timers gets the new behaviour. Both groups are happy.

13.9.9 A Note on Sysctl Naming

The chapter uses sysctl names like `dev.myfirst.0.heartbeat_interval_ms`. The `_ms` suffix is intentional: it documents the unit. A user who sees `heartbeat_interval` could reasonably guess seconds, milliseconds, or microseconds; the suffix removes the ambiguity.

Other conventions:

- `_count` for counters (always non-negative).
- `_max`, `_min` for bounds.
- `_threshold` for switches.
- `_ratio` for percentages or fractions.

Following these conventions makes the sysctl tree self-describing. A user inspecting `sysctl dev.myfirst.0` can guess the meaning of every entry from its name and unit.

13.9.10 Wrapping Up Section 8

The driver is now version `0.7-timers`. It has:

- A documented callout discipline in `LOCKING.md`.
- A standardised macro pair (`MYFIRST_CO_INIT`, `MYFIRST_CO_DRAIN`) for callout lifecycle.
- A seven-phase detach pattern documented in code comments.
- A regression script that exercises every callout.
- A pre-commit checklist that catches the Chapter 13-specific failure modes.
- Three new sysctls with self-describing names and a default-disabled posture.

That is the close of the chapter's main teaching arc. The labs and challenges follow.

13.10 Hands-on Labs

These labs consolidate the Chapter 13 concepts through direct hands-on experience. They are ordered from least to most demanding.

13.10.1 Pre-Lab Setup Checklist

Before starting any lab, confirm:

1. **Debug kernel running.** `sysctl kern.ident` reports the kernel with `INVARIANTS` and `WITNESS`.

2. **WITNESS active.** `sysctl debug.witness.watch` returns a non-zero value.
3. **Driver source matches Chapter 12 Stage 4.** The Chapter 13 examples build on top of this.
4. **A clean dmesg.** `dmesg -c >/dev/null` once before the first lab.
5. **Companion userland built.** From `examples/part-03/ch12-synchronization-mechanisms/userland/`, the `timeout/config` testers should be present.
6. **Backup of Stage 4.** Copy your Chapter 12 Stage 4 driver to a safe location before starting any lab that modifies the source.

13.10.2 Lab 13.1: Add a Heartbeat Callout

Goal. Convert your Chapter 12 Stage 4 driver into a Chapter 13 Stage 1 driver by adding the heartbeat callout.

Steps.

1. Copy your Stage 4 driver into `examples/part-03/ch13-timers-and-delayed-work/stage1-heartbeat/`.
2. Add `struct callout heartbeat_co` and `int heartbeat_interval_ms` to `struct myfirst_softc`.
3. In `myfirst_attach`, call `callout_init_mtx(&sc->heartbeat_co, &sc->mtx, 0)` and initialize `heartbeat_interval_ms = 0`.
4. In `myfirst_detach`, drop the mutex and add `callout_drain(&sc->heartbeat_co)`; before destroying primitives.
5. Implement `myfirst_heartbeat` callback as shown in Section 4.
6. Implement `myfirst_sysctl_heartbeat_interval_ms` and register it.
7. Build and load on a WITNESS kernel.
8. Verify by setting the `sysctl`: `sysctl -w dev.myfirst.0.heartbeat_interval_ms=1000` and watching `dmesg` for heartbeat lines.

Verification. Heartbeat lines appear once per second when enabled. They stop when the `sysctl` is set to 0. Detach succeeds even with the heartbeat enabled. No WITNESS warnings.

Stretch goal. Use `dtrace` to count heartbeat callbacks per second:

```
# dtrace -n 'fbt::myfirst_heartbeat:entry { @ = count(); } tick-1sec { printa(@);  
↵  trunc(@); }'
```

The count should match the configured interval (1 per second for 1000 ms).

13.10.3 Lab 13.2: Add a Watchdog Callout

Goal. Add the watchdog callout that detects stalled buffer drainage.

Steps.

1. Copy Lab 13.1 into `stage2-watchdog/`.
2. Add `struct callout watchdog_co`, `int watchdog_interval_ms`, `size_t watchdog_last_used` to the `softc`.
3. Initialize and drain in `attach/detach` as for the heartbeat.

4. Implement `myfirst_watchdog` and the corresponding `sysctl` handler from Section 7.
5. Build and load.
6. Test: enable a 1-second watchdog, write some bytes, do not drain, observe warning.

Verification. The watchdog warning appears every second while the buffer has unconsumed bytes. The warning stops when the buffer is drained.

Stretch goal. Make the watchdog log the time since the last change in the warning message: “no progress for X.Y seconds”.

13.10.4 Lab 13.3: Add a Tick Source

Goal. Add the tick source callout that injects synthetic bytes into the `cbuf`.

Steps.

1. Copy Lab 13.2 into `stage3-tick-source/`.
2. Add struct callout `tick_source_co`, `int tick_source_interval_ms`, `char tick_source_byte` to the `softc`.
3. Initialize and drain as before.
4. Implement `myfirst_tick_source` as shown in Section 7. Note the deliberate omission of `selwakeup` from the callback.
5. Implement the `sysctl` handler.
6. Build and load.
7. Enable a 100 ms tick source, read with `cat`, observe the synthetic bytes.

Verification. `cat /dev/myfirst` produces approximately 10 bytes per second of the configured tick byte (default `'t'`).

Stretch goal. Add a `sysctl` that lets the user change the tick byte at runtime. Verify that changing the byte takes effect immediately on the next firing.

13.10.5 Lab 13.4: Verify Detach With Active Callouts

Goal. Confirm that detach works correctly even when all three callouts are firing.

Steps.

1. Load the Stage 3 (tick source) driver.
2. Enable all three callouts:

```
sysctl -w dev.myfirst.0.heartbeat_interval_ms=500
sysctl -w dev.myfirst.0.watchdog_interval_ms=500
sysctl -w dev.myfirst.0.tick_source_interval_ms=100
```

3. Confirm activity in `dmesg`.
4. Run `kldunload myfirst`.
5. Verify no panic, no WITNESS warning, no hang.

Verification. The unload completes within a few hundred milliseconds. `dmesg` shows no warnings related to the unload.

Stretch goal. Time the unload with `time kldunload myfirst`. The drain should be the dominant contributor to the time; expect a few hundred milliseconds depending on the intervals.

13.10.6 Lab 13.5: Build a Debounce Timer

Goal. Implement a debounce shape (not used by `myfirst`, but a useful exercise).

Steps.

1. Create a scratch directory for an experimental driver.
2. Implement a `sysctl dev.myfirst.0.event_count` that increments by 1 each time it is written. (The user write triggers the “event”.)
3. Add a debounce callout that fires 100 ms after the most recent event and prints the total count of events seen during the window.
4. Test: write the `sysctl` rapidly five times. Observe that one debounce log line appears 100 ms after the last write, reporting the count.

Verification. Multiple rapid events produce only one log line, with a count equal to the number of events.

13.10.7 Lab 13.6: Detect a Deliberate Race

Goal. Introduce a deliberate bug (a callout callback that calls something sleepable) and observe `WITNESS` catching it.

Steps.

1. In a scratch directory, modify the heartbeat callback to call something that may sleep, such as `pause("test", hz / 100)`.
2. Build and load on a `WITNESS` kernel.
3. Enable the heartbeat with a 1-second interval.
4. Observe `dmesg` for the warning: “Sleeping on ”test” with the following non-sleepable locks held: ...” or similar.
5. Revert the change.

Verification. `WITNESS` produces a warning that names the sleeping operation and the held mutex. The warning includes the source line.

13.10.8 Lab 13.7: Long-Running Composite Stress with Timers

Goal. Run the Chapter 12 composite stress kit for 30 minutes with the new Chapter 13 callouts enabled.

Steps.

1. Load the Stage 4 driver.
2. Enable all three callouts at 100 ms intervals.
3. Run the Chapter 12 composite stress script for 30 minutes.

4. After completion, check:

- `dmesg | grep -ci witness` returns 0.
- All loop iterations completed.
- `vmstat -m | grep cbuf` shows the expected static allocation.

Verification. All criteria met; no warnings, no panics, no memory growth.

13.10.9 Lab 13.8: Profile Callout Activity With `dtrace`

Goal. Use `dtrace` to observe callout firing patterns.

Steps.

1. Load the Stage 4 driver.
2. Enable all three callouts at 100 ms intervals.
3. Run a `dtrace` one-liner to count callout firings per callback per second:

```
# dtrace -n '
fbt::myfirst_heartbeat:entry,
fbt::myfirst_watchdog:entry,
fbt::myfirst_tick_source:entry { @[probefunc] = count(); }
tick-1sec { printa(@); trunc(@); }'
```

4. Observe the per-second counts.

Verification. Each callback fires approximately 10 times per second (1000 ms / 100 ms).

Stretch goal. Modify the `dtrace` script to report the time spent inside each callback (using `quantize` and `timestamp`).

13.10.10 Lab 13.9: Cancel a Watchdog Inline

Goal. Make the watchdog a one-shot timer that the read path cancels on success, demonstrating the cancel-on-progress pattern.

Steps.

1. Copy Lab 13.4 (stage3-tick-source plus heartbeat/watchdog) into a scratch directory.
2. Modify `myfirst_watchdog` to be one-shot: do not re-arm at the end.
3. Schedule the watchdog from `myfirst_write` after each successful write.
4. Cancel the watchdog (using `callout_stop`) from `myfirst_read` after a successful drain.
5. Test: write some bytes; do not read; observe the watchdog warning fire once.
6. Test: write some bytes; read them; observe no warning (because the read cancelled the watchdog).

Verification. The watchdog warning fires only when the buffer is left undrained. Successful drains cancel the pending watchdog.

Stretch goal. Add a counter that tracks how often the watchdog fired versus how often it was cancelled. Expose as a `sysctl`. The ratio is a quality metric for buffer drainage.

13.10.11 Lab 13.10: Schedule From Inside a Sysctl Handler

Goal. Verify that scheduling a callout from a sysctl handler produces correct timing.

Steps.

1. Add a sysctl `dev.myfirst.0.schedule_one-shot_ms` to the Stage 4 driver. Writing N to it schedules a one-shot callback to fire N milliseconds later.
2. The callback simply logs “one-shot fired”.
3. Test: write 100 to the sysctl. Observe the log line about 100 ms later.
4. Test: write 1000 to the sysctl. Observe about 1 second later.
5. Test: write 1 to the sysctl five times in quick succession. Observe how the kernel handles the rapid rescheduling.

Verification. Each write produces one log line at approximately the configured interval. Rapid writes either schedule new firings (cancelling the previous) or are coalesced; observe which.

Stretch goal. Use `dtrace` to measure the delta between the sysctl write and the actual firing. The histogram should be tight around the configured interval.

13.11 Challenge Exercises

The challenges extend Chapter 13 beyond the baseline labs. Each is optional; each is designed to deepen your understanding.

13.11.1 Challenge 1: Sub-Millisecond Tick Source

Modify the tick source callout to use `callout_reset_sbt` with a sub-millisecond interval (say, 250 microseconds). Test it. What happens to the heartbeat output (which logs counters)? What does `lockstat` show for the data mutex?

13.11.2 Challenge 2: Watchdog with Adaptive Intervals

Make the watchdog reduce its interval each time it fires (signal of trouble), and increase its interval when it sees forward progress. Cap both ends at reasonable values.

13.11.3 Challenge 3: Defer the Selwakeup to a Taskqueue

The tick source omits `selwakeup` because it cannot be called from the callout context. Read `taskqueue(9)` (Chapter 16 will introduce this in depth) and use a taskqueue to defer the `selwakeup` to a worker thread. Verify that `poll(2)` waiters now wake correctly.

13.11.4 Challenge 4: Multi-CPU Callout Distribution

By default, callouts run on a single CPU. Use `callout_reset_on` to bind each of the three callouts to a different CPU. Use `dtrace` to verify the binding. Discuss the trade-offs.

13.11.5 Challenge 5: Bound the Maximum Interval

Add validation to each interval sysctl to enforce a minimum (say, 10 ms) and a maximum (say, 60000 ms). Below the minimum, refuse with `EINVAL`. Above the maximum, also refuse. Document the choice.

13.11.6 Challenge 6: Callout-Based Read Timeout

Replace the Chapter 12 `cv_timedwait_sig`-based read timeout with a callout-based mechanism: schedule a one-shot callout when the reader starts blocking; the callout fires `cv_signal` on the data `cv` to wake the reader. Compare the two approaches.

13.11.7 Challenge 7: A Statistics Rollover

Add a callout that takes a snapshot of `bytes_read` and `bytes_written` every 5 seconds and stores the per-interval rates in a circular buffer (separate from the `cbuf`). Expose the most recent rates via a sysctl.

13.11.8 Challenge 8: Drain Without Hold

Verify experimentally that calling `callout_drain` while holding the callout's lock deadlocks. Write a small driver variant that does this deliberately, observe the deadlock with DDB, and document the symptom.

13.11.9 Challenge 9: Reuse a Callout Structure

Use the same `struct callout` for two different callbacks at different times: schedule with callback A, wait for it to fire, schedule with callback B. What happens if A is still pending when you call `callout_reset` with B's function? Write a test to verify the kernel's behaviour.

13.11.10 Challenge 10: Callout-Based Hello-World Module

Write a minimal module (no `myfirst` involved) that does nothing except install a single callout that prints "tick" every second. Use this as a sanity check for the callout subsystem on your test machine.

13.11.11 Challenge 11: Verify Lock Serialization

Demonstrate that two callouts sharing the same lock are serialized. Write a driver with two callouts; have each callback sleep briefly (with `DELAY()` if you must, since `DELAY()` does not sleep but spins). Confirm via `dt race` that the callbacks never overlap.

13.11.12 Challenge 12: Coalesce Latency

Use `callout_reset_sbt` with various precision values (0, `SBT_1MS`, `SBT_1S`) for a 1-second timer. Use `dt race` to measure the actual firing times. How much does the kernel coalesce when given more slop? When does coalescing reduce CPU usage?

13.11.13 Challenge 13: Callout Wheel Inspection

The kernel exposes the callout wheel state through `kern.callout_stat` and `kern.callout_*` sysctls. Read them on a busy system. Can you identify the callouts your driver has scheduled?

13.11.14 Challenge 14: Callout Function Pointer Replacement

Schedule a callout with one function. Before it fires, schedule it again with a different function. What happens? Does the second function replace the first? Document the behaviour with a small experiment.

13.11.15 Challenge 15: Adaptive Heartbeat

Make the heartbeat fire faster when there is recent activity (writes in the last second) and slower when idle. The interval should range from 100 ms (active) to 5 seconds (idle). Test it under a stress workload to verify it adapts as expected.

13.12 Troubleshooting

This reference catalogues the bugs you are most likely to encounter while working through Chapter 13.

13.12.1 Symptom: Callout never fires

Cause. Either the interval is zero (callout was disabled), or the sysctl handler did not actually call `callout_reset`.

Fix. Check the sysctl handler logic. Confirm that the transition from 0 to non-zero is detected. Add a `device_printf` at the call site to verify.

13.12.2 Symptom: kldunload panics shortly after

Cause. A callout was not drained at detach. The callout fired after the module was unloaded.

Fix. Add `callout_drain` for every callout in the detach path. Confirm the order: drain *after* clearing `is_attached`, *before* destroying primitives.

13.12.3 Symptom: WITNESS warns “sleeping thread (pid X) owns a non-sleepable lock”

Cause. A callout callback called something that sleeps (`uiomove`, `copyin`, `malloc(M_WAITOK)`, `pause`, or any `cv_wait` variant) while holding the kernel-acquired mutex.

Fix. Remove the sleeping operation from the callback. If the work requires sleeping, defer to a taskqueue or kernel thread.

13.12.4 Symptom: Heartbeat fires once and then never again

Cause. The callback re-arm code is missing or guarded by a condition that becomes false.

Fix. Check the re-arm at the end of the callback. Confirm that `interval_ms > 0` and that the call to `callout_reset` actually executes.

13.12.5 Symptom: Callout fires more often than the configured interval

Cause. Two paths are scheduling the same callout. Either the `sysctl` handler and the callback are both calling `callout_reset`, or two callbacks share a callout struct.

Fix. Audit the call sites. The `sysctl` handler should `callout_reset` only on the 0 -> non-zero transition; the callback re-arms only at its own end.

13.12.6 Symptom: Detach hangs

Cause. A callout callback re-armed itself between the `is_attached = 0` and the `callout_drain`. The drain is now waiting for the callback to finish; the callback (which checked `is_attached` before the assignment took effect) is not exiting.

Fix. Confirm that the `is_attached = 0` happens under the same lock as the callout's lock. Confirm that the drain happens after the assignment, not before. The check inside the callback must see the cleared flag.

13.12.7 Symptom: WITNESS warns about lock-order issues with the callout's lock

Cause. The callout lock is being acquired in conflicting orders by different paths.

Fix. The callout's lock is `sc->mtx`. Confirm that every path that acquires `sc->mtx` follows the canonical order (mtx first, then any other lock). The callback runs with `sc->mtx` already held; the callback must not acquire any lock that should be acquired before `sc->mtx`.

13.12.8 Symptom: Callout-Drain Sleeps Forever

Cause. `callout_drain` was called with the callout's lock held. Deadlock: the drain waits for the callback to release the lock, the callback is waiting because the drain is the lock holder.

Fix. Drop the lock before calling `callout_drain`. The standard detach pattern does this.

13.12.9 Symptom: Callback runs but data is stale

Cause. The callback is using cached values from before the firing. Either it stored data in a local variable that became stale, or it dereferenced a structure that was modified.

Fix. The callback runs with the lock held. Re-read fields each time the callback fires; do not cache across firings.

13.12.10 Symptom: `procstat -kk` shows no thread waiting on the callout

Cause. Callouts do not have associated threads. The callback runs in a kernel thread context (the callout subsystem manages a small pool), but no specific thread is “the callout’s thread” the way a kernel thread might own a wait condition.

Fix. None needed; this is by design. To see callout activity, use `dtrace` or `lockstat` instead.

13.12.11 Symptom: `callout_reset` returns 1 unexpectedly

Cause. The callout was previously pending and got cancelled by this `callout_reset`. The return value is informational, not an error.

Fix. None needed; this is normal. Use the return value if you care whether the previous schedule was overwritten.

13.12.12 Symptom: Sysctl handler reports `EINVAL` for valid input

Cause. The handler’s validation rejects the value. Common cause: the user passed a negative number that the validation correctly rejects, or the handler has a too-strict bound.

Fix. Inspect the validation code. Confirm the user’s input meets the documented constraints.

13.12.13 Symptom: Two callbacks for different callouts run concurrently and deadlock

Cause. Both callouts are bound to the same lock, so they cannot run concurrently. If they appear to deadlock, check whether either callback acquires another lock that the other path already holds.

Fix. Audit the lock acquisition order. The callback-running thread holds `sc->mtx`; if it tries to acquire `sc->cfg_sx`, the order must be `mtx-then-sx` (which our canonical order is).

13.12.14 Symptom: `tick_source` produces the wrong byte

Cause. The callback reads `tick_source_byte` at firing time. If a sysctl just changed it, the callback may see either the old or the new value depending on timing.

Fix. This is correct behaviour; the byte change takes effect on the next firing. If immediate effect is required, use the snapshot-and-apply pattern from Chapter 12.

13.12.15 Symptom: `lockstat` shows the data mutex held for unusually long during heartbeats

Cause. The heartbeat callback is doing too much work while the lock is held.

Fix. The heartbeat does only counter reads and one log line; if the hold time is long, it is probably the `device_printf` (which acquires global locks for the message buffer). For low-overhead heartbeats, gate the log line behind a debug level.

13.12.16 Symptom: Heartbeat continues after sysctl is set to 0

Cause. The `callout_stop` did not actually cancel because the callback was already running. The callback re-armed before checking the new value.

Fix. The race is closed if the sysctl handler holds `sc->mtx` while updating `interval_ms` and calling `callout_stop`. The callback runs under the same lock; it cannot run between the update and the stop. Verify the lock is held in the right places.

13.12.17 Symptom: WITNESS warns about acquiring the callout's lock during init

Cause. Some earlier path in `attach` has not yet established the lock order rules. Adding a callout's lock association makes WITNESS notice the inconsistency.

Fix. Move the `callout_init_mtx` to after the mutex is initialized. The order must be: `mtx_init`, then `callout_init_mtx`.

13.12.18 Symptom: A single fast callout causes high CPU usage

Cause. A 1 ms callout that does even a small amount of work fires 1000 times per second. If each firing takes 100 microseconds, that is 10% of one CPU.

Fix. Increase the interval. Sub-second intervals should be used only when truly required.

13.12.19 Symptom: dtrace cannot find the callback function

Cause. `dtrace`'s `fbt` provider needs the function to be present in the kernel's symbol table. If the function was inlined or optimized away, the probe is not available.

Fix. Confirm the function is not declared `static inline` or wrapped in a way that prevents external linkage. The standard `static void myfirst_heartbeat(void *arg)` is fine; `dtrace` can probe it.

13.12.20 Symptom: heartbeat_interval_ms reads back as 0 after setting it

Cause. The sysctl handler updates a local copy and never commits to the `softc` field, or the field is overwritten elsewhere.

Fix. Confirm the handler assigns `sc->heartbeat_interval_ms = new` after validation, before returning.

13.12.21 Symptom: WITNESS warns "callout_init: lock has sleepable lock_class"

Cause. You called `callout_init_mtx` with an `sx` lock or other sleepable primitive instead of an `MTX_DEF` mutex. Sleepable locks are forbidden as callout interlocks because callouts run in a context where sleeping is illegal.

Fix. Use `callout_init_mtx` with an `MTX_DEF` mutex, or `callout_init_rw` with an `rw(9)` lock, or `callout_init_rm` with an `rmlock(9)`. Do not use `sx`, `lockmgr`, or any other sleepable lock.

13.12.22 Symptom: Detach takes seconds even when callouts seem idle

Cause. A callout has a long interval (say, 30 seconds) and is currently pending. `callout_drain` waits for the next firing or for the explicit cancellation. If the deadline is far in the future, the wait can be long.

Fix. `callout_drain` actually does not wait for the deadline; it cancels the pending and returns once any in-flight callback completes. If your detach takes seconds, something else is wrong (a callback is genuinely taking that long, or a different sleep is involved). Use `dtrace` on `_callout_stop_safe` to investigate.

13.12.23 Symptom: `callout_pending` returns true after `callout_stop`

Cause. Race: another path scheduled the callout between the `callout_stop` and your check of `callout_pending`. Or: the callout was previously firing on another CPU and has just re-armed.

Fix. Always hold the callout's lock when calling `callout_stop` and checking `callout_pending`. The lock makes the operations atomic.

13.12.24 Symptom: A callout function appears in `dmesg` long after the driver was unloaded

Cause. The unload race. The callout fired after detach destroyed state. If the kernel did not panic immediately, the printed line is from the freed callback's code, executing in a kernel that has lost track of the original module.

Fix. This should not happen if you called `callout_drain` correctly. If it does, your detach path is broken; review every callout to confirm each is drained.

13.12.25 Symptom: Multiple callouts fire all at once after a long pause

Cause. The system was under load (a long-running interrupt, a stuck callout-processing thread) and could not service the callout wheel. When it recovers, it processes all the deferred callouts in quick succession.

Fix. This is normal under unusual load. If it happens routinely, investigate why the system could not service callouts on time. `dtrace -n 'callout-end'` (using the `callout` provider, if your kernel exposes it) shows actual firing times.

13.12.26 Symptom: A periodic callout drifts: each firing is slightly later than the previous

Cause. The re-arm is `callout_reset(&co, hz, ..., ...)`, which schedules "1 second from now". Each firing's "now" is slightly later than the previous firing's deadline, so the actual interval grows by the callback's execution time.

Fix. For exact periodicity, compute the next deadline as "previous deadline + interval", not "now + interval". Use `callout_reset_sbt` with `C_ABSOLUTE` and an absolute sbintime computed from the original schedule.

13.12.27 Symptom: Callout never fires even though `callout_pending` returns true

Cause. Either the callout is stuck on a CPU that is offline (rare, but possible during CPU hot-unplug), or the system's clock interrupt is not firing on that CPU.

Fix. Check `kern.hz` and `kern.eventtimer` sysctls. The default `hz=1000` should produce regular firings. If a CPU is offline, the callout subsystem migrates pending callouts to a working CPU, but there is a window. For most drivers, this is not a real concern.

13.12.28 Symptom: Stress test causes intermittent panic in `callout_process`

Cause. Almost certainly the unload race or an incorrect lock association on a callout. The callout subsystem itself is well-tested; bugs at this level are usually in the calling code.

Fix. Audit every callout's init and drain. Check that the lock association is correct (no sleepable locks). Run with `INVARIANTS` to catch invariant violations.

13.12.29 Symptom: `kern.callout.busy` counter grows under load

Cause. The callout subsystem detected callbacks taking too long. Each "busy" event is a callback that did not complete within an expected window.

Fix. Inspect the slow callbacks with `dtrace`. Long callbacks indicate either too much work (split into multiple callouts or defer to a taskqueue) or a lock-contention issue (the callback is waiting for the lock to become available).

13.12.30 Symptom: Driver logs show "callout_drain detected migration" or similar

Cause. A callout was bound to a specific CPU (via `callout_reset_on`) and the binding migration overlapped with the drain. The kernel resolves this internally; the log message is informational.

Fix. None usually needed. If the message is frequent, consider whether per-CPU binding is necessary at all.

13.12.31 Symptom: `callout_reset_sbt` gives unexpected timing

Cause. The precision argument is too loose: the kernel coalesced your callout with others into a window much wider than expected.

Fix. Set precision to a smaller value (or 0 for "fire as close to deadline as possible"). The default is `tick_sbt` (one tick of slop), which is fine for most timer work.

13.12.32 Symptom: A callout that was working stops firing after a power-management event

Cause. The system's clock interrupt may have been reconfigured (transition between event-timer modes during sleep/wake). The callout subsystem reschedules pending callouts after such transitions, but the timing may be slightly off.

Fix. Verify with `dtrace` that the callout's callback is being invoked. If not, the callout has been migrated or discarded; reschedule from a known-good code path.

13.12.33 Symptom: All callouts in the driver fire on the same CPU

Cause. This is the default. Callouts are bound to the CPU that scheduled them; if all your `callout_reset` calls run on CPU 0 (because the user's syscall was dispatched there), all the callouts fire on CPU 0.

Fix. This is correct for most drivers. If you want load distribution, use `callout_reset_on` to bind explicitly to different CPUs. Most drivers do not need this; the per-CPU wheels naturally balance over time as different syscalls hit different CPUs.

13.12.34 Symptom: `callout_drain` returns but the next syscall sees stale state

Cause. The callback completed and returned, but a subsequent code path observed state that the callback had set. This is correct behaviour, not a bug.

Fix. None. The drain only guarantees the callback is no longer running; any state changes the callback made are still in effect. If the changes are unwanted, the callback should not have made them.

13.12.35 Symptom: Re-arm in callback fails silently

Cause. The condition `interval > 0` is false because the user just disabled the timer. The callback exits without re-arming; the callout becomes idle.

Fix. This is correct behaviour. If you want to know when the callback declined to re-arm, add a counter or log line.

13.12.36 Symptom: Callout fires but `device_printf` is silent

Cause. The driver's `dev` field is NULL or the device has been detached and the `cdev` destroyed. `device_printf` may suppress output in those states.

Fix. Add an explicit `printf("%s: ...\n", device_get_nameunit(dev), ...)` to bypass the wrapper. Or confirm that `sc->dev` is valid via `KASSERT`.

13.13 Reference: The Driver's Stage Progression

Chapter 13 evolves the `myfirst` driver in four discrete stages, each of which is its own directory under `examples/part-03/ch13-timers-and-delayed-work/`. The progression mirrors the chapter's narrative; it lets a reader build the driver one timer at a time and see what each addition contributes.

13.13.1 Stage 1: heartbeat

Adds the heartbeat callout that periodically logs `cbuf` usage and byte counts. The new `sysctl dev.myfirst.<unit>.heartbeat` enables, disables, and adjusts the heartbeat at runtime.

What changes: one new callout, one new callback, one new `sysctl`, and the corresponding `init/drain` in `attach/detach`.

What you can verify: setting the sysctl to a positive value produces periodic log lines; setting it to 0 stops them; detach succeeds even with the heartbeat enabled.

13.13.2 Stage 2: watchdog

Adds the watchdog callout that detects stalled buffer drainage. The new sysctl `dev.myfirst.<unit>.watchdog_interval` enables, disables, and adjusts the interval.

What changes: one new callout, one new callback, one new sysctl, and the corresponding init/drain.

What you can verify: enabling the watchdog and writing bytes (without reading them) produces warning lines; reading the buffer stops the warnings.

13.13.3 Stage 3: tick-source

Adds the tick source callout that injects synthetic bytes into the cbuf. The new sysctl `dev.myfirst.<unit>.tick_source_interval` enables, disables, and adjusts the interval.

What changes: one new callout, one new callback, one new sysctl, and the corresponding init/drain.

What you can verify: enabling the tick source and reading from `/dev/myfirst` produces bytes at the configured rate.

13.13.4 Stage 4: final

The combined driver with all three callouts, plus a `LOCKING.md` extension, the version bump to `0.7-timers`, and the standardised `MYFIRST_CO_INIT` and `MYFIRST_CO_DRAIN` macros.

What changes: integration. No new primitives.

What you can verify: the regression suite passes; the long-duration stress test with all callouts active runs cleanly; `WITNESS` is silent.

This four-stage progression is the canonical Chapter 13 driver. The companion examples mirror the stages exactly so a reader can compile and load any one of them.

13.14 Reference: Anatomy of a Real Watchdog

A real production watchdog does more than the chapter's example. A short tour of what real watchdogs typically include, useful when you write or read driver source.

13.14.1 Per-Request Tracking

Real I/O watchdogs track each pending request individually. The watchdog callback walks a list of pending requests, finds those that have been outstanding too long, and acts on each.

```
struct myfirst_request {
    TAILQ_ENTRY(myfirst_request) link;
    sbintime_t    submitted_sbt;
    int           op;
    /* ... other request state ... */
};

TAILQ_HEAD(, myfirst_request) pending_requests;
```

The watchdog walks `pending_requests`, computes the age of each, and acts on the stale ones.

13.14.2 Threshold-Based Action

Different ages get different actions. Up to T1, ignore (the request is still working). T1 to T2, log a warning. T2 to T3, attempt soft recovery (send a reset to the request). Beyond T3, hard recovery (reset the channel, fail the request).

```
age_sbt = now - req->submitted_sbt;
if (age_sbt > sc->watchdog_hard_sbt) {
    /* hard recovery */
} else if (age_sbt > sc->watchdog_soft_sbt) {
    /* soft recovery */
} else if (age_sbt > sc->watchdog_warn_sbt) {
    /* log warning */
}
```

13.14.3 Statistics

A real watchdog tracks how often each threshold was hit, what percentage of requests exceeded each threshold, and so on. The statistics are exposed as sysctls for monitoring.

13.14.4 Configurable Thresholds

Each threshold (T1, T2, T3) is a sysctl. Different deployments need different bounds; hard-coding is wrong.

13.14.5 Recovery Logging

The recovery action logs to `dmesg` with a recognizable prefix that monitoring tools can `grep`. A detailed message with the request's identity, the action taken, and any kernel state that might help diagnose the underlying issue.

13.14.6 Coordination With Other Subsystems

A hard recovery often involves cooperation with other parts of the driver: the I/O layer must know that the channel is being reset, queued requests must be re-queued or failed, and the driver's "is operational" state must be updated.

For Chapter 13, our watchdog is much simpler. It detects one specific condition (no progress on the cbuf), logs a warning, and re-arms. This captures the essential pattern. Real-world watchdogs add the pieces above incrementally.

13.15 Reference: Periodic vs Event-Driven Driver Architecture

A small architectural digression. Some drivers are dominated by events (an interrupt arrives, the driver responds). Others are dominated by polling (the driver wakes periodically to check). Understanding which one your driver is helps choose primitives.

13.15.1 Event-Driven

In an event-driven design, the driver is mostly idle. Activity is triggered by:

- User syscalls (open, read, write, ioctl).
- Hardware interrupts (Chapter 14).
- Wake-ups from other subsystems (cv signals, taskqueue runs).

Callouts in an event-driven design are typically watchdogs (track an event, fire if it does not happen) and reapers (clean up after events).

The `myfirst` driver was originally event-driven (read/write triggered everything). Chapter 13 adds some polling-flavoured behaviour (heartbeat, tick source) for demonstration, but the underlying design is still event-driven.

13.15.2 Polling-Driven

In a polling-driven design, the driver wakes periodically to do work, regardless of whether anyone is asking. This is appropriate for hardware that does not interrupt for the events the driver cares about.

Callouts in a polling-driven design are the heartbeat of the driver: every firing, the callback checks the hardware and processes whatever it finds.

The polling-loop pattern (Section 7) is the basic shape. Real polling drivers extend it with adaptive intervals (poll faster when busy, slower when idle), error counting (give up after too many failed polls), and so on.

13.15.3 Hybrid

Most real drivers are hybrid: events drive most activity, but a periodic callout catches what events miss (timeouts, slow polling, statistics). The patterns from this chapter apply to either side; the choice of which to use where is a design decision.

For `myfirst`, our hybrid uses: - Event-driven syscall handlers for the main I/O. - A heartbeat callout for periodic logging. - A watchdog callout for stuck-state detection. - An optional tick source callout for synthetic event generation.

A real driver would have many more callouts, but the shape is the same.

13.16 Wrapping Up

Chapter 13 took the driver you built in Chapter 12 and gave it the ability to act on its own schedule. Three callouts now sit alongside the existing primitives: a heartbeat that periodically logs state, a watchdog that detects stalled drainage, and a tick source that injects synthetic bytes. Each is lock-aware, drained at detach, configurable through a `sysctl`, and documented in `LOCKING.md`. The driver's data path is unchanged; the new code is purely additive.

We learned that `callout(9)` is small, regular, and well-integrated with the rest of the kernel. The lifecycle is the same five stages every time: `init`, `schedule`, `fire`, `complete`, `drain`. The lock contract is the same model every time: the kernel acquires the registered lock before each firing and releases it after, serializing the callback against any other holder. The detach pattern is the same seven phases every time: `refuse if busy`, `mark going-away under the lock`, `drop the lock`, `drain selinfo`, `drain every callout`, `destroy cdevs`, `free state`, `destroy primitives in reverse order`.

We also learned a small handful of recipes that recur across drivers: heartbeat, watchdog, debounce, retry-with-backoff, deferred reaper, statistics rollover. Each is a small variation on the periodic or one-shot shape; once you know the patterns, the variations are mechanical.

Four closing reminders before moving on.

The first is to *drain every callout at detach*. The unload race is reliable, immediate, and fatal. The cure is mechanical: one `callout_drain` per callout, after `is_attached` is cleared and before primitives are destroyed. There is no excuse to skip this.

The second is to *keep callbacks short and lock-aware*. The callback runs with the registered lock held, in a context that may not sleep. Treat it like a hardware-interrupt handler: do the minimum, defer the rest. If the work needs to sleep, enqueue it on a taskqueue (Chapter 16) or wake a kernel thread.

The third is to *use sysctls to make timer behaviour configurable*. Hard-coding intervals is a maintenance burden. Letting users tune the heartbeat, the watchdog, or the tick source from `sysctl -w` makes the driver useful in environments you did not anticipate. The cost is small (one `sysctl` handler per knob) and the benefit is large.

The fourth is to *update `LOCKING.md` in the same commit as any code change*. A driver whose documentation drifts from the code accumulates subtle bugs because no one knows what the rules are supposed to be. The discipline is one minute per change; the benefit is years of clean maintenance.

These four disciplines together produce drivers that compose well with the rest of FreeBSD, that survive long-term maintenance, and that behave predictably under load. They are also the disciplines Chapter 14 will assume; the patterns from this chapter transfer directly to interrupt handlers.

13.16.1 What You Should Now Be Able to Do

A short self-checklist before turning to Chapter 14:

- Choose between `callout(9)` and `cv_timedwait_sig` for any “wait until X happens, or until Y time has passed” requirement.
- Initialize a callout with the appropriate lock variant for your driver’s needs.
- Schedule one-shot and periodic callouts using `callout_reset` (or `callout_reset_sbt` for sub-tick precision).
- Cancel callouts with `callout_stop` in normal operation; drain with `callout_drain` at detach.
- Write a callout callback that respects the lock contract and the no-sleeping rule.
- Use the `is_attached` pattern to make callbacks safe during teardown.
- Document every callout in `LOCKING.md`, including its lock, its callback, its lifecycle.
- Recognize the unload race and avoid it through the standard seven-phase detach pattern.
- Build watchdog, heartbeat, debounce, and tick-source patterns as needed.
- Use `dtrace` to verify callout firing rates, latency, and lifecycle behaviour.
- Read real-driver source (led, uart, network drivers) and recognize the patterns from this chapter at work.

If any of those feels uncertain, the labs in this chapter are the place to build the muscle memory. None requires more than an hour or two; together they cover every primitive and every pattern the chapter introduced.

13.16.2 A Note On the Companion Examples

The companion source under `examples/part-03/ch13-timers-and-delayed-work/` mirrors the chapter’s stages. Each stage builds on the previous one, so you can compile and load any stage to see exactly the driver state the chapter describes at that point.

If you prefer to type the changes by hand (recommended for the first read), use the chapter’s worked examples as your guide and the companion source as a reference. If you prefer to read finished code, the companion source is canonical.

A note on the `LOCKING.md` document: the chapter’s text explains what `LOCKING.md` should contain. The actual file is in the example tree alongside the source. Keep both in sync as you make changes; the discipline of updating `LOCKING.md` in the same commit as the code change is the most reliable way to keep the documentation accurate.

13.17 Reference: `callout(9)` Quick Reference

A compact API summary for everyday lookup.

13.17.1 Initialization

```
callout_init(&co, 1)                /* mpsafe; no lock */
callout_init_mtx(&co, &mtx, 0)      /* lock is mtx (default) */
callout_init_mtx(&co, &mtx, CALLOUT_RETURNUNLOCKED)
callout_init_rw(&co, &rw, 0)        /* lock is rw, exclusive */
callout_init_rw(&co, &rw, CALLOUT_SHAREDLOCK)
callout_init_rm(&co, &rm, 0)        /* lock is rmlock */
```

13.17.2 Scheduling

```
callout_reset(&co, ticks, fn, arg)           /* tick-based delay */
callout_reset_sbt(&co, sbt, prec, fn, arg, flags)
callout_reset_on(&co, ticks, fn, arg, cpu) /* bind to CPU */
callout_schedule(&co, ticks)                 /* re-use last fn/arg */
```

13.17.3 Cancellation

```
callout_stop(&co)                           /* cancel; do not wait */
callout_drain(&co)                          /* cancel + wait for in-flight */
callout_async_drain(&co, drain_fn)           /* drain async */
```

13.17.4 Inspection

```
callout_pending(&co)                        /* is the callout scheduled? */
callout_active(&co)                        /* user-managed active flag */
callout_deactivate(&co)                    /* clear the active flag */
```

13.17.5 Common Flags

```
CALLOUT_RETURNUNLOCKED /* callback releases the lock itself */
CALLOUT_SHAREDLOCK     /* acquire rw/rm in shared mode */
C_HARDCLOCK            /* align to hardclock() */
C_DIRECT_EXEC          /* run in timer interrupt context */
C_ABSOLUTE              /* sbt is absolute time */
```

13.18 Reference: Standard Detach Pattern

The seven-phase detach pattern for a driver with callouts:

```
static int
myfirst_detach(device_t dev)
{
    struct myfirst_softc *sc = device_get_softc(dev);

    /* Phase 1: refuse if in use. */
```

```
MYFIRST_LOCK(sc);
if (sc->active_fhs > 0) {
    MYFIRST_UNLOCK(sc);
    return (EBUSY);
}

/* Phase 2: mark going away; broadcast cvs. */
sc->is_attached = 0;
cv_broadcast(&sc->data_cv);
cv_broadcast(&sc->room_cv);
MYFIRST_UNLOCK(sc);

/* Phase 3: drain selinfo. */
seldrain(&sc->rsl);
seldrain(&sc->wsl);

/* Phase 4: drain every callout (no lock held). */
callout_drain(&sc->heartbeat_co);
callout_drain(&sc->watchdog_co);
callout_drain(&sc->tick_source_co);

/* Phase 5: destroy cdevs (no new opens). */
if (sc->cdev_alias != NULL) {
    destroy_dev(sc->cdev_alias);
    sc->cdev_alias = NULL;
}
if (sc->cdev != NULL) {
    destroy_dev(sc->cdev);
    sc->cdev = NULL;
}

/* Phase 6: free auxiliary resources. */
sysctl_ctx_free(&sc->sysctl_ctx);
cbuf_destroy(&sc->cb);
counter_u64_free(sc->bytes_read);
counter_u64_free(sc->bytes_written);

/* Phase 7: destroy primitives in reverse order. */
cv_destroy(&sc->data_cv);
cv_destroy(&sc->room_cv);
sx_destroy(&sc->cfg_sx);
mtx_destroy(&sc->mtx);

return (0);
```

}

Skipping any phase creates a class of bug that crashes the kernel under load.

13.19 Reference: When to Use Each Timing Primitive

A compact decision table.

Need	Primitive
One-shot callback at time T	<code>callout_reset</code>
Periodic callback every T ticks	<code>callout_reset</code> with re-arm
Sub-millisecond callback timing	<code>callout_reset_sbt</code>
Bind callback to a specific CPU	<code>callout_reset_on</code>
Run callback in timer-interrupt context	<code>callout_reset_sbt</code> with <code>C_DIRECT_EXEC</code>
Wait until a condition, with deadline	<code>cv_timedwait_sig</code> (Chapter 12)
Wait until a condition, no deadline	<code>cv_wait_sig</code> (Chapter 12)
Defer work to a worker thread	<code>taskqueue_enqueue</code> (Chapter 16)
Long-running periodic work	Kernel thread + <code>cv_timedwait</code>

The first four are `callout(9)` use cases; the others use other primitives.

13.20 Reference: Callout Mistakes to Avoid

A compact list of the most common mistakes:

- **Forgetting `callout_drain` at detach.** Causes a panic the next time the callout fires.
- **Calling `callout_drain` while holding the callout's lock.** Causes a deadlock.
- **Calling sleeping functions from the callback.** Causes a `WITNESS` warning or a panic.
- **Using `callout_init` (with `mpsafe=0`) for new code.** Acquires Giant; harms scalability.
- **Forgetting the `is_attached` check at the top of the callback.** Detach can race with re-arm and never complete.
- **Sharing a struct `callout` between two callbacks.** Confusing and rarely what you want; use two structs.
- **Hard-coding intervals in the callback.** No way for users to tune behaviour.
- **Failing to validate `sysctl` input.** Negative or absurd intervals create surprising behaviour.
- **Calling `selwakeup` from a callback.** Takes other locks; can produce ordering violations.
- **Using `callout_stop` at detach.** Does not wait for in-flight; causes the unload race.

13.21 Reference: Reading `kern_timeout.c`

Two functions in `/usr/src/sys/kern/kern_timeout.c` are worth opening once.

`callout_reset_sbt_on` is the core scheduling function. Every other variant of `callout_reset` is a wrapper that ends up here. The function handles the cases of “callout is currently running”, “callout is pending and being rescheduled”, “callout needs to be migrated to a different CPU”, and “callout is fresh”. The complexity is real; the public-facing behaviour is simple.

`_callout_stop_safe` is the unified stop-and-maybe-drain function. Both `callout_stop` and `callout_drain` are macros that call this function with different flags. The `CS_DRAIN` flag is the one that triggers the wait-for-in-flight behaviour. Reading this function once shows exactly how the drain interacts with the firing callback.

The file is around 1550 lines in FreeBSD 14.3. You do not need to read every line. Skim the function names, find the two functions above, and read each carefully. Twenty minutes of reading gives you a working sense of the implementation.

13.22 Reference: The `c_iflags` and `c_flags` Fields

A short look at the two flag fields, useful when you read real driver source that inspects them directly.

`c_iflags` (internal flags, set by the kernel):

- `CALLOUT_PENDING`: the callout is on the wheel waiting to fire. Read via `callout_pending(c)`.
- `CALLOUT_PROCESSED`: internal bookkeeping for which list the callout is on.
- `CALLOUT_DIRECT`: set if `C_DIRECT_EXEC` was used.
- `CALLOUT_DFRMIGRATION`: set during deferred migration to a different CPU.
- `CALLOUT_RETURNUNLOCKED`: set if the lock-handling contract was set up to expect the callback to release the lock.
- `CALLOUT_SHAREDLOCK`: set if the rw/rm lock is to be acquired in shared mode.

`c_flags` (external flags, managed by the caller):

- `CALLOUT_ACTIVE`: a user-managed bit. Set by the kernel during a successful `callout_reset`; cleared by `callout_deactivate` or by a successful `callout_stop`. Driver code reads via `callout_active(c)`.
- `CALLOUT_LOCAL_ALLOC`: deprecated; only used in legacy `timeout(9)` style.
- `CALLOUT_MPSAFE`: deprecated; use `callout_init_mtx` instead.

Driver code touches only `CALLOUT_ACTIVE` (via `callout_active` and `callout_deactivate`) and `CALLOUT_PENDING` (via `callout_pending`). Everything else is internal.

13.23 Reference: A Chapter 13 Self-Assessment

Before turning to Chapter 14, use this rubric. The format mirrors Chapter 11 and Chapter 12: conceptual questions, code-reading questions, and hands-on questions. If any item feels uncertain, the relevant section name is in parentheses.

The questions are not exhaustive; they sample the chapter's core ideas. A reader who can answer all of them confidently is ready for the next chapter. A reader who struggles on a particular item should re-read the relevant section before continuing.

13.23.1 Conceptual Questions

These questions sample the Chapter 13 vocabulary. A reader who can answer them all without re-checking the chapter has internalised the material.

1. **Why use `callout(9)` instead of a kernel thread for periodic work?** Callouts are essentially free at rest; threads have a 16 KB stack and scheduler overhead. For short periodic work, a callout is the right answer.
2. **What is the difference between `callout_stop` and `callout_drain`?** `callout_stop` cancels pending and returns immediately; `callout_drain` cancels pending and waits for any in-flight callback to finish. Use `callout_stop` in normal operation, `callout_drain` at detach.
3. **What does the lock argument to `callout_init_mtx` accomplish?** The kernel acquires that lock before each callback firing and releases it after. The callback runs with the lock held.
4. **What may a callout callback not do?** Anything that may sleep, including `cv_wait`, `mtx_sleep`, `uiomove`, `copyin`, `copyout`, `malloc(M_WAITOK)`, `selwakeup`.
5. **Why does the standard detach pattern call `callout_drain` after dropping the mutex?** `callout_drain` may sleep waiting for the in-flight callback. Sleeping with the mutex held is illegal. Dropping the mutex before draining is mandatory.
6. **What is the unload race?** A callout fires after `kldunload` has run, finding its function and surrounding state freed. The kernel jumps to invalid memory and panics.
7. **What is the periodic callback re-arm pattern?** The callback does its work and calls `callout_reset` at the end to schedule the next firing.
8. **Why does the callback check `is_attached` before doing work?** Detach clears `is_attached`; if the callback fires during the brief window between clearing and `callout_drain`, the check prevents the callback from doing work that depends on state being torn down.
9. **What happens if you call `callout_drain` while holding the callout's lock?** Deadlock: the drain waits for the callback to release the lock, but the callback cannot release the lock that the drainer is holding. Always drop the lock before calling `callout_drain`.
10. **What is the purpose of `MYFIRST_CO_INIT` and `MYFIRST_CO_DRAIN`?** They are macro wrappers around `callout_init_mtx` and `callout_drain` that document the convention: every callout uses `sc->mtx` and is drained at detach. Standardising via macros makes new callouts mechanical to add and easy to review.
11. **Why is `device_printf` safe to call from a callout callback while the mutex is held?** It does not acquire any of the driver's locks and it does not sleep; it writes to a global ringbuffer with its own internal locking. It is one of the few "safe to call from a callout context" output functions.
12. **What is the difference between scheduling a callout for hz ticks and scheduling it for `tick_sbt * hz`?** Conceptually nothing; both represent one second. The first uses `callout_reset` (tick-based API); the second uses `callout_reset_sbt` (sbintime-based API). Choose the API that matches the precision you need.

13.23.2 Code Reading Questions

Open your Chapter 13 driver source and verify:

1. Every `callout_init_mtx` is paired with a `callout_drain` in `detach`.
2. Every callback starts with `MYFIRST_ASSERT(sc)` and `if (!sc->is_attached) return;`.
3. No callback calls `selwakeup`, `uiomove`, `copyin`, `copyout`, `malloc(M_WAITOK)`, or `cv_wait`.
4. The `detach` path drops the mutex before calling `callout_drain`.
5. Every callout has a `sysctl` that allows the user to enable, disable, or change its interval.
6. Every callout's lifecycle (init in `attach`, drain in `detach`) is documented in `LOCKING.md`.
7. Every periodic callback re-arms only when its interval is positive.
8. Every `sysctl` handler holds the mutex when calling `callout_reset` or `callout_stop`.

13.23.3 Hands-On Questions

These should all be quick to run; if any fails, the relevant lab in this chapter walks through the setup.

1. Load the Chapter 13 driver. Enable the heartbeat with a 1-second interval. Confirm `dmesg` shows one log line per second.
2. Enable the watchdog with a 1-second interval. Write some bytes. Wait. Confirm the warning appears.
3. Enable the tick source with a 100 ms interval. Read with `cat`. Confirm 10 bytes per second.
4. Enable all three callouts. Run `kldunload myfirst`. Verify no panic, no warning.
5. Open `/usr/src/sys/kern/kern_timeout.c`. Find `callout_reset_sbt_on`. Read the first 50 lines. Can you describe what it does in two sentences?
6. Use `dttrace` to confirm that the heartbeat fires at the expected rate. With `heartbeat_interval_ms=200`, the rate should be 5 firings per second.
7. Modify the watchdog to log additional information (e.g., how many callbacks have fired since boot). Verify the new field appears in `dmesg`.
8. Open `/usr/src/sys/dev/led/led.c`. Find the calls to `callout_init_mtx`, `callout_reset`, and `callout_drain`. Compare to the patterns in this chapter. Are there differences?

If all eight hands-on questions pass and the conceptual questions are easy, your Chapter 13 work is solid. You are ready for Chapter 14.

A note on pacing: Part 3 of this book has been dense. Three chapters (11, 12, 13) on synchronization-related topics is a lot of new vocabulary to absorb. If the labs in this chapter felt easy, that is a good sign. If they felt hard, take a day or two before starting Chapter 14; the material will compound nicely after a short break, and starting Chapter 14 with fresh attention is better than pushing through tired.

A note on testing: the regression script in Section 8 covers the basic functionality. For long-term confidence, run the composite stress kit from Chapter 12 with all three Chapter 13 callouts active, on a `WITNESS` kernel, for at least 30 minutes. A clean run is the bar to clear before declaring the driver production-ready. Anything less risks the unload race or a subtle lock-order issue that the basic regression does not catch.

If the stress run finds problems, the troubleshooting reference earlier in this chapter is the first stop. Most issues fall into one of the symptom patterns there. If the symptom does not match anything in the reference, the next step is the in-kernel debugger; the DDB recipes in the troubleshooting section get you started.

When you have a clean stress run and a clean review, the chapter's work is done. The driver is now stage-4 final, version 0.7-timers, with a documented callout discipline and a regression suite that proves the discipline holds under load. Take a moment to appreciate that. Then proceed.

13.24 Reference: Callouts in Real FreeBSD Drivers

A short tour of how `callout(9)` is used in actual FreeBSD source. The patterns you have learned in this chapter map directly onto the patterns these drivers use.

13.24.1 `/usr/src/sys/dev/led/led.c`

A simple but instructive example. The `led(4)` driver lets user-space scripts blink LEDs on hardware that supports them. The driver schedules a callout every `hz / 10` (100 ms) to step through the blink pattern.

The key call inside `led_timeout` and `led_state` is:

```
callout_reset(&led_ch, hz / 10, led_timeout, p);
```

The callout is initialized in `led_drvinit`:

```
callout_init_mtx(&led_ch, &led_mtx, 0);
```

A periodic callout, lock-aware via the driver's mutex. Exactly the pattern this chapter teaches.

13.24.2 `/usr/src/sys/dev/uart/uart_core.c`

The serial-port (UART) driver uses a callout in some configurations to poll for input on hardware that does not interrupt for character receipt. The pattern is the same: `callout_init_mtx`, periodic callback, drain at detach.

13.24.3 Network Driver Watchdogs

Most network drivers (`ixgbe`, `em`, `mlx5`, etc.) install a watchdog callout at attach time. The watchdog fires every few seconds; it checks whether the hardware has produced an interrupt recently and resets the chip if not. The callback is short, lock-aware, and deferred to the rest of the driver if it needs to do something complex (typically by enqueueing a task on a taskqueue).

13.24.4 Storage Driver I/O Timeouts

ATA, NVMe, and SCSI drivers use callouts as I/O timeouts. When a request is sent to the hardware, the driver schedules a callout for some bounded time in the future. If the request completes normally, the driver cancels the callout. If the callout fires, the driver assumes the request is stuck and takes recovery action (reset the channel, retry, fail the request to the user).

This is the watchdog pattern (Section 7) applied to per-request operations rather than to the device as a whole.

13.24.5 USB Hub Polling

The USB hub driver polls hub status every few hundred milliseconds (configurable). The poll discovers attached/detached devices, port-status changes, and transfer completions that the hub does not interrupt for. The pattern is the polling-loop pattern (Section 7).

13.24.6 What These Drivers Do Differently

The drivers above use additional primitives beyond what Chapter 13 covers, especially taskqueues. Many of them schedule a callout that, instead of doing the actual work, enqueues a task on a taskqueue. The task runs in process context and may sleep. Chapter 16 introduces this pattern in depth.

For Chapter 13 we keep all the work inside the callout's callback, lock-aware and non-sleeping. Real drivers extend this pattern by deferring long work; the underlying timing infrastructure (callout) is the same.

13.25 Reference: Comparing callout to Other Time Primitives

A more detailed comparison than the decision table earlier in the chapter.

13.25.1 `callout(9)` vs `cv_timedwait_sig`

The same primitive at the syscall layer is `cv_timedwait_sig` (Chapter 12): “wait until condition X becomes true, but give up after T milliseconds”. The caller is the one waiting; the cv is the one that gets signalled.

Compare to `callout(9)`: a callback runs at time T, regardless of whether anyone is waiting for it. The callback is the one acting; it does its own work, possibly signalling other things.

The two differ in *who waits* and *who acts*. In `cv_timedwait_sig`, the syscall thread is both waiter and (after wakeup) actor. In `callout(9)`, the syscall thread is uninvolved; an independent context fires the callback.

Use `cv_timedwait_sig` when the syscall thread has work to do as soon as the wait completes. Use `callout(9)` when something independent must happen at a deadline.

13.25.2 `callout(9)` vs `taskqueue(9)`

`taskqueue(9)` runs a function “as soon as possible” by enqueueing it on a worker thread. There is no time delay; the work runs as soon as the worker can pick it up.

Compare to `callout(9)`: the function runs at a specific time in the future.

A common pattern is to combine the two: a callout fires at time *T*, decides that work is needed, and enqueues a task. The task runs in process context and does the actual work (which may include sleeping). Chapter 16 will cover this combination.

13.25.3 `callout(9)` vs Kernel Threads

A kernel thread can loop and call `cv_timedwait_sig` to produce periodic behaviour. The thread is heavyweight: 16 KB of stack, scheduler entry, priority assignment.

Compare to `callout(9)`: no thread; the kernel's timer interrupt mechanism handles the firing, and a small callout-processing pool runs the callbacks.

Use a kernel thread when the work is genuinely long-running (the worker thread waits, does substantial work, waits again). Use a callout when the work is short and you just need it to fire on a schedule.

13.25.4 `callout(9)` vs Periodic `SIGALRM` in User Space

A user-space process can install a `SIGALRM` handler and use `alarm(2)` for periodic behaviour. The signal handler runs in the process; it is short and constrained.

Compare to `callout(9)`: kernel-side, lock-aware, integrated with the rest of the driver.

User-space alarms are appropriate for user-space code. They have no role in driver work; the kernel does its own thing.

13.25.5 `callout(9)` vs Hardware Timers

Some hardware has its own timer registers (a “GP timer” or “watchdog timer” that the host driver programs). These hardware timers fire interrupts directly to the host. They are fast, precise, and bypass the kernel's callout subsystem.

Use the hardware timer when: - The hardware provides one and you have an interrupt handler. - The precision required exceeds what `callout_reset_sbt` can deliver.

Use `callout(9)` when: - The hardware does not have a usable timer for your purpose. - The precision the kernel can deliver (down to `tick_sbt` or sub-tick with `_sbt`) is sufficient.

For our pseudo-device, no hardware exists; `callout(9)` is the right and only choice.

13.26 Reference: Common Callout Vocabulary

A glossary of terms used in the chapter, useful when you encounter them in driver source.

Callout: an instance of `struct callout`; a scheduled-or-unscheduled timer.

Wheel: the kernel's per-CPU array of callout buckets, organized by deadline.

Bucket: one element of the wheel, holding a list of callouts that should fire in a small range of time.

Pending: state in which the callout is on the wheel waiting to fire.

Active: a user-managed bit indicating “I scheduled this callout and have not actively cancelled it”; not the same as pending.

Firing: state in which the callout’s callback is currently executing.

Idle: state in which the callout is initialized but not pending; either never scheduled or has fired and not re-armed.

Drain: the operation of waiting for an in-flight callback to complete (typically in detach); `callout_drain`.

Stop: the operation of cancelling a pending callout without waiting; `callout_stop`.

Direct execution: an optimization where the callback runs in the timer interrupt context itself, set with `C_DIRECT_EXEC`.

Migration: the kernel’s relocation of a callout to a different CPU (typically because the originally-bound CPU is offline).

Lock-aware: a callout initialized with one of the lock variants (`callout_init_mtx`, `_rw`, or `_rm`); the kernel acquires the lock for each firing.

Mpsafe: a legacy term for “may be called without acquiring Giant”; in modern use it appears as the `mpsafe` argument to `callout_init`.

Re-arm: the action a callback takes to schedule the next firing of the same callout.

13.27 Reference: Timer Anti-Patterns to Avoid

A short catalogue of patterns that look reasonable but are wrong.

Anti-pattern 1: Polling in a tight loop. Some drivers, especially ones written by beginners, busy-wait on a hardware register: `while (!(read_reg() & READY)) ; /* keep checking */`. This burns CPU and produces a system that is unresponsive under load. The callout-based polling pattern is the cure: schedule a callback that checks the register and re-arms.

Anti-pattern 2: Hard-coded intervals. A driver that hard-codes “wait 100 ms” everywhere is a driver that is hard to tune. Make the interval a `sysctl` or a `softc` field that the user can adjust.

Anti-pattern 3: Missing drain at detach. The most common Chapter 13 mistake. The unload race crashes the kernel. Always drain.

Anti-pattern 4: Sleeping in the callback. The callback runs with a non-sleepable lock held; sleeping is forbidden. If the work needs to sleep, defer to a kernel thread or taskqueue.

Anti-pattern 5: Using `callout_init` (the legacy variant) for new code. The lock-naive variant requires you to do all your own locking inside the callback, which is more error-prone than letting the kernel do it. Use `callout_init_mtx` for new code.

Anti-pattern 6: Sharing a `struct callout` between multiple callbacks. A `struct callout` is not a queue. If you need to fire two different callbacks, use two `struct callouts`.

Anti-pattern 7: Calling `callout_drain` while holding the callout’s lock. Causes a deadlock. Drop the lock first.

Anti-pattern 8: Setting the same lock as multiple unrelated subsystems' callout interlocks. The serialization can produce surprising lock contention. Each subsystem should generally have its own lock; share only when the work is genuinely related.

Anti-pattern 9: Re-using a struct callout after callout_drain without re-initializing. After drain, the callout's internal state is reset, but the function and argument from the last callout_reset are still there. If you callout_schedule next, you reuse those. This is subtle. For clarity, call callout_init_mtx again before reuse.

Anti-pattern 10: Forgetting that callout_stop does not wait. In normal operation this is correct; in detach it is wrong. Use callout_drain for detach.

These patterns recur often enough to be worth memorizing. A driver that avoids all ten will have a much easier time.

13.28 Reference: Tracing Callouts With dtrace

A short collection of dtrace recipes useful for inspecting callout behaviour. Each is one or two lines; together they cover most diagnostic needs.

13.28.1 Count Firings of a Specific Callback

```
# dtrace -n 'fbt::myfirst_heartbeat:entry { @ = count(); } tick-1sec { printa(@);
↪   trunc(@); }'
```

Per-second count of how many times the heartbeat callback ran. Useful for confirming the configured rate.

13.28.2 Histogram Time Spent In Callback

```
# dtrace -n '
fbt::myfirst_heartbeat:entry { self->ts = timestamp; }
fbt::myfirst_heartbeat:return /self->ts/ {
    @ = quantize(timestamp - self->ts);
    self->ts = 0;
}
tick-30sec { exit(0); }'
```

Distribution of callback durations, in nanoseconds. Useful for spotting unusually slow firings.

13.28.3 Trace All Callout Resets


```
# dtrace -n 'fbt::callout_reset_sbt_on:entry { printf("co=%p, fn=%p, arg=%p", arg0,  
↪ arg3, arg4); }'
```

Every `callout_reset` (and its variants) call. Useful for confirming what code paths are scheduling callouts.

13.28.4 Trace Callout Drains

```
# dtrace -n 'fbt::_callout_stop_safe:entry /arg1 == 1/ { printf("drain co=%p", arg0);  
↪ stack(); }'
```

Every call to the drain path (`flags == CS_DRAIN`). Useful for confirming detach calls drain on every callout.

13.28.5 Per-CPU Callout Activity

```
# dtrace -n 'fbt::callout_process:entry { @[cpu] = count(); } tick-1sec { printa(@);  
↪ trunc(@); }'
```

Per-second count of callout-processing invocations on each CPU. Tells you which CPUs are doing the timer work.

13.28.6 Identify Slow Callouts

```
# dtrace -n '  
fbt::callout_process:entry { self->ts = timestamp; }  
fbt::callout_process:return /self->ts/ {  
    @ = quantize(timestamp - self->ts);  
    self->ts = 0;  
}  
tick-30sec { exit(0); }'
```

Distribution of how long the callout-processing loop takes. Long durations indicate either many callouts firing at the same time or slow individual callbacks.

13.28.7 A Combined Diagnostic Script

For reading per-second:

```
# dtrace -n '
fbt::callout_reset_sbt_on:entry { @resets = count(); }
fbt::_callout_stop_safe:entry /arg1 == 1/ { @drains = count(); }
fbt::myfirst_heartbeat:entry { @hb = count(); }
fbt::myfirst_watchdog:entry { @wd = count(); }
fbt::myfirst_tick_source:entry { @ts = count(); }
tick-1sec {
    printa("resets=%@u drains=%@u hb=%@u wd=%@u ts=%@u\n",
        @resets, @drains, @hb, @wd, @ts);
    trunc(@resets); trunc(@drains);
    trunc(@hb); trunc(@wd); trunc(@ts);
}'
```

A condensed diagnostic line per second. Useful as a sanity check during development.

13.29 Reference: Inspecting Callout State From DDB

When a system hangs and you need to inspect callout state from the debugger, several DDB commands help.

13.29.1 show callout <addr>

If you know a callout's address, this shows its current state: pending or not, scheduled deadline, callback function pointer, argument. Useful when you know which callout to inspect.

13.29.2 show callout_stat

Dumps overall callout statistics: how many are scheduled, how many fired since boot, how many are pending. Useful for system-wide overview.

13.29.3 ps

The standard process listing. Threads inside callout-processing are typically named `clock` or similar. They are usually in `mi_switch` or in the callback being executed.

13.29.4 bt <thread>

Backtrace of a specific thread. If the thread is inside a callout callback, the backtrace shows the call chain: the kernel's callout subsystem at the bottom, the callback at the top. This tells you which callback is running.

13.29.5 show all locks

If a callout's callback is currently executing, the backtrace will show `mtx_lock` (the kernel acquiring the callout's lock). `show all locks` confirms which lock is held and by which thread.

13.29.6 Combined: Inspecting a Hung Callout

```
db> show all locks
... shows myfirst0 mutex held by thread 1234

db> ps
... 1234 is "myfirst_heartbeat" (or similar)

db> bt 1234
... backtrace shows _cv_wait or similar; the callback is sleeping (which it should not!)
```

If you see this, the callback is doing something illegal (sleeping with a non-sleepable lock held). The fix is to remove the sleeping operation from the callback.

13.30 Reference: Comparing Tick-Based and SBT-Based APIs

The two callout APIs (tick-based and sbintime-based) deserve a side-by-side comparison.

13.30.1 Tick-Based API

```
callout_reset(&co, ticks, fn, arg);
callout_schedule(&co, ticks);
```

The delay is in ticks: integer count of clock interrupts. On a 1000 Hz kernel, one tick is one millisecond. Multiply seconds by hz to convert; e.g., $5 * \text{hz}$ for five seconds, $\text{hz} / 10$ for 100 ms.

Pros: simple, well-known, fast (no sbintime arithmetic). Cons: precision limited to one tick (1 ms typical); cannot express sub-tick delays.

Use for: most callout work. Watchdogs at second-level intervals, heartbeats at hundred-millisecond intervals, periodic polling at tens-of-milliseconds intervals.

13.30.2 SBT-Based API

```
callout_reset_sbt(&co, sbt, prec, fn, arg, flags);
callout_schedule_sbt(&co, sbt, prec, flags);
```

The delay is an `sbintime_t`: high-precision binary fixed-point time. Use `SBT_1S`, `SBT_1MS`, `SBT_1US`, `SBT_1NS` constants to construct values.

Pros: sub-tick precision; explicit precision/coalescing argument; explicit flags for absolute vs relative time. Cons: more arithmetic; must understand `sbintime_t`.

Use for: callouts that need sub-millisecond precision (network protocols, hardware controllers with tight timing requirements). Most driver work does not need this.

13.30.3 Conversion Helpers

```
sbintime_t  ticks_to_sbt = tick_sbt * timo_in_ticks;  /* tick_sbt is global */
sbintime_t  ms_to_sbt  = ms_value * SBT_1MS;
sbintime_t  us_to_sbt  = us_value * SBT_1US;
```

The `tick_sbt` global gives you the sbintime equivalent of one tick; multiply by your tick count to convert.

13.31 Reference: Pre-Production Callout Audit

A short audit to perform before promoting a callout-using driver from development to production. Each item is a question; each item should be answerable with confidence.

13.31.1 Callout Inventory

- ☐ Have I listed every callout the driver owns in `LOCKING.md`?
- ☐ For each callout, have I named its callback function?
- ☐ For each callout, have I named the lock it uses (if any)?
- ☐ For each callout, have I documented its lifetime (init in `attach`, drain in `detach`)?
- ☐ For each callout, have I documented its trigger (what causes it to be scheduled)?
- ☐ For each callout, have I documented whether it re-arms (periodic) or fires once (one-shot)?

13.31.2 Initialization

- ☐ Does every callout init use `callout_init_mtx` (or `_rw/_rm`) rather than the bare `callout_init`?
- ☐ Is the init called after the lock it references is initialized?
- ☐ Is the lock kind correct (sleep mutex for sleepable contexts, etc.)?

13.31.3 Scheduling

- ☐ Does every `callout_reset` happen with the appropriate lock held?
- ☐ Is the interval reasonable for the work the callback does?
- ☐ Is the conversion from milliseconds to ticks correct ($(ms * hz + 999) / 1000$ for ceil)?
- ☐ If the callout is periodic, does the callback re-arm only under a documented condition?

13.31.4 Callback Hygiene

- ☐ Does every callback start with `MYFIRST_ASSERT(sc)` (or equivalent)?
- ☐ Does every callback check `is_attached` before doing work?
- ☐ Does every callback exit early if `is_attached == 0`?
- ☐ Does the callback avoid sleeping operations (`uiomove`, `cv_wait`, `mtx_sleep`, `malloc(M_WAITOK)`, `sel-wakeup`)?
- ☐ Is the callback's total work time bounded?

13.31.5 Cancellation

- ☐ Does the `sysctl` handler use `callout_stop` to disable the timer?
- ☐ Does the `sysctl` handler hold the lock when calling `callout_stop` and `callout_reset`?
- ☐ Are there any code paths that can race with the `sysctl` handler?

13.31.6 Detach

- ☐ Does the detach path drop the mutex before calling `callout_drain`?
- ☐ Does the detach path drain every callout?
- ☐ Are the callouts drained in the correct phase (after `is_attached` is cleared)?

13.31.7 Documentation

- ☐ Is every callout documented in `LOCKING.md`?
- ☐ Are the discipline rules (lock-aware, no-sleep, drain-at-detach) documented?
- ☐ Is the callout subsystem mentioned in the `README`?
- ☐ Are there `sysctls` exposed that let users tune the behaviour?

13.31.8 Testing

- ☐ Have I run the regression suite with `WITNESS` enabled?
- ☐ Have I tested detach with all callouts active?
- ☐ Have I run a long-duration stress test?
- ☐ Have I used `dt race` to verify the firing rates match the configured intervals?

A driver that passes this audit is a driver you can trust under load.

13.32 Reference: Standardising Timers Across a Driver

For a driver with several callouts, consistency matters more than cleverness. A short discipline.

13.32.1 One Naming Convention

Pick a convention and follow it. The chapter's convention:

- The callout struct is named `<purpose>_co` (e.g., `heartbeat_co`, `watchdog_co`, `tick_source_co`).
- The callback is named `myfirst_<purpose>` (e.g., `myfirst_heartbeat`, `myfirst_watchdog`, `myfirst_tick_source`).
- The interval sysctl is named `<purpose>_interval_ms` (e.g., `heartbeat_interval_ms`, `watchdog_interval_ms`, `tick_source_interval_ms`).
- The sysctl handler is named `myfirst_sysctl_<purpose>_interval_ms`.

A new maintainer can add a new callout following the convention without thinking about names. Conversely, a code review immediately catches deviations.

13.32.2 One Init/Drain Pattern

Every callout uses the same initialization and drain:

```
/* In attach: */
callout_init_mtx(&sc-><purpose>_co, &sc->mtx, 0);

/* In detach (after dropping the mutex): */
callout_drain(&sc-><purpose>_co);
```

Or, with the macros:

```
MYFIRST_CO_INIT(sc, &sc-><purpose>_co);
MYFIRST_CO_DRAIN(&sc-><purpose>_co);
```

The macros document the pattern in their definition; the call sites are short and uniform.

13.32.3 One Sysctl Handler Pattern

Every interval sysctl handler follows the same structure:

```
static int
myfirst_sysctl_<purpose>_interval_ms(SYSCTL_HANDLER_ARGS)
{
    struct myfirst_softc *sc = arg1;
    int new, old, error;

    old = sc-><purpose>_interval_ms;
    new = old;
    error = sysctl_handle_int(oidp, &new, 0, req);
```

```

    if (error || req->newptr == NULL)
        return (error);
    if (new < 0)
        return (EINVAL);

    MYFIRST_LOCK(sc);
    sc-><purpose>_interval_ms = new;
    if (new > 0 && old == 0) {
        /* enabling */
        callout_reset(&sc-><purpose>_co,
            (new * hz + 999) / 1000,
            myfirst_<purpose>, sc);
    } else if (new == 0 && old > 0) {
        /* disabling */
        callout_stop(&sc-><purpose>_co);
    }
    MYFIRST_UNLOCK(sc);
    return (0);
}

```

The handler's shape is the same for every interval sysctl. A new sysctl is mechanical to add.

13.32.4 One Callback Pattern

Every periodic callback follows the same structure:

```

static void
myfirst_<purpose>(void *arg)
{
    struct myfirst_softc *sc = arg;
    int interval;

    MYFIRST_ASSERT(sc);
    if (!sc->is_attached)
        return;

    /* ... do the per-firing work ... */

    interval = sc-><purpose>_interval_ms;
    if (interval > 0)
        callout_reset(&sc-><purpose>_co,
            (interval * hz + 999) / 1000,
            myfirst_<purpose>, sc);
}

```

Assert, check `is_attached`, do work, conditionally re-arm. Every callback in the driver has this shape; deviations stand out.

13.32.5 One Documentation Pattern

Every callout is documented in `LOCKING.md` with the same fields:

- Lock used.
- Callback function.
- Behaviour (periodic or one-shot).
- Started by (which code path schedules it).
- Stopped by (which code path stops it).
- Lifetime (init in attach, drain in detach).

A new callout's documentation is mechanical. A code review can verify the documentation against the code.

13.32.6 Why Standardise

Standardisation has costs: a new contributor must learn the conventions; deviations require a special reason. The benefits are larger:

- Reduced cognitive load. A reader who knows the pattern instantly understands every callout.
- Fewer mistakes. The standard pattern handles the common cases (lock acquisition, `is_attached` check, drain) correctly; a deviation is more likely to be wrong.
- Easier review. Reviewers can scan for shape rather than reading every line.
- Easier handoff. A maintainer who has not seen the driver can add a new callout following the existing template.

The cost of standardisation is paid once at design time. The benefits accrue forever. Always worth it.

13.33 Reference: Further Reading on Timers

For readers who want to go deeper:

13.33.1 Manual Pages

- `callout(9)`: the canonical API reference.
- `timeout(9)`: the legacy interface (deprecated; mentioned for historical reading).
- `microtime(9)`, `getmicrouptime(9)`, `getsbinuptime(9)`: time-reading primitives that callouts often use.
- `eventtimers(4)`: the event-timer subsystem that drives callouts.
- `kern.eventtimer`: the sysctl tree that exposes event-timer state.

13.33.2 Source Files

- `/usr/src/sys/kern/kern_timeout.c`: the callout implementation.
- `/usr/src/sys/kern/kern_clocksource.c`: the event-timer driver layer.
- `/usr/src/sys/sys/callout.h`, `/usr/src/sys/sys/_callout.h`: the public API and structure.
- `/usr/src/sys/sys/time.h`: the sbintime constants and conversion macros.
- `/usr/src/sys/dev/led/led.c`: a small driver that exemplifies the callout pattern.
- `/usr/src/sys/dev/uart/uart_core.c`: a more elaborate use, including a polling fallback for hardware that does not interrupt for input.

13.33.3 Manual Pages To Read in Order

For a reader new to FreeBSD's time subsystem, a sensible reading order:

1. `callout(9)`: the canonical API reference.
2. `time(9)`: units and primitives.
3. `eventtimers(4)`: the event-timer subsystem that drives callouts.
4. `kern.eventtimer` and `kern.hz sysctls`: runtime controls.
5. `microuptime(9)`, `getmicrouptime(9)`: time-reading primitives.
6. `kproc(9)`, `kthread(9)`: for when you genuinely need a kernel thread.

Each builds on the previous; reading in order takes a couple of hours and gives you a solid mental model of the kernel's time infrastructure.

13.33.4 External Material

The chapter on timers in *The Design and Implementation of the FreeBSD Operating System* (McKusick et al.) covers the historical evolution of timer subsystems and the reasoning behind the current design. Useful as context; not required.

The FreeBSD developers' mailing list (`freebsd-hackers@`) occasionally discusses callout improvements and edge cases. Searching the archive for "callout" returns relevant historical context for how the API has evolved.

For deeper understanding of how the kernel schedules events at the lowest level, the `eventtimers(4)` manual page and the source under `/usr/src/sys/kern/kern_clocksource.c` are worth a careful read. They are below the level of this chapter (we do not interact with event timers directly), but they explain why the callout subsystem can deliver the precision it does.

Finally, real driver source. Pick any driver in `/usr/src/sys/dev/` that uses callouts (most do), read its callout-related code, and compare it to the patterns in this chapter. The translation is direct; you will recognize the shapes immediately. That kind of reading turns the chapter's abstractions into working knowledge.

13.34 Reference: Callout Cost Analysis

A short discussion of what callouts actually cost, useful when deciding intervals or designing a high-frequency timer.

13.34.1 Cost At Rest

A `struct callout` that has not been scheduled costs nothing beyond the size of the structure (about 80 bytes on amd64). The kernel does not know about it. It sits in your softc, doing nothing.

A `struct callout` that has been scheduled but not yet fired costs slightly more: the kernel has linked it into a wheel bucket. The link entries cost a few bytes. The kernel does not poll the structure; it only looks at it when the relevant bucket comes due.

The hardware-clock interrupt (which drives the wheel) fires `hz` times per second (typically 1000). It is essentially zero-cost in the empty case (no callouts due) and proportional to the number of due callouts in the busy case.

13.34.2 Cost Per Firing

When a callout fires, the kernel does roughly:

1. Walk the wheel bucket; find the callout. Constant time per callout in the bucket.
2. Acquire the callout's lock (if any). Cost depends on contention; typically nanoseconds.
3. Call the callback function. Cost depends on the callback.
4. Release the lock. Microseconds.

For a typical short callback (a few microseconds of work), the per-firing cost is dominated by the callback itself plus the lock acquisition. The kernel's overhead is negligible.

13.34.3 Cost At Cancel/Drain

`callout_stop` is fast: linked-list removal plus an atomic flag update. Microseconds.

`callout_drain` is fast if the callout is idle (just like `callout_stop`). If the callback is currently firing, the drain waits via the sleep-queue mechanism; the wait time depends on how long the callback takes.

13.34.4 Practical Implications

Hundreds of pending callouts: no problem. The wheel handles them efficiently.

Thousands of pending callouts: still no problem in normal operation. Walking a wheel bucket of dozens of callouts is fast.

A single callout that fires at 1 Hz: essentially free. One hardware interrupt out of a thousand walks the bucket and finds the callout.

A single callout that fires at 1 kHz: starts to be measurable. One thousand callbacks per second adds up. If the callback takes 10 microseconds, that is 1% of one CPU. If the callback is heavier, more.

A callout at 10 kHz or faster: probably the wrong design. Reach for a busy-poll or a hardware timer or a specialised mechanism.

13.34.5 Comparison To Other Approaches

A kernel thread that loops on `cv_timedwait` and does work each wake costs:

- Memory: ~16 KB stack.
- Per wake: scheduler entry, context switch, callback, context switch back.

For a 1 Hz workload, the kernel-thread cost (one wake per second) is about the same as the callout cost. For a 1 kHz workload, both are similar. For a 10 kHz workload, both are getting expensive; consider whether you really need that frequency.

A user-space loop polling a `sysctl`:

- Memory: a whole user process (megabytes).
- Per poll: syscall round-trip, `sysctl` handler invocation, return to user space.

Always more expensive than a kernel callout. Only appropriate when the polling logic genuinely belongs in user space (a monitoring tool, an external probe).

13.34.6 When To Worry About Cost

Most drivers do not have to. Callouts are cheap; the kernel is well-tuned. Worry about cost only when:

- Profiling shows callouts dominate CPU usage. (Use `dtrace` to confirm.)
- You are writing a high-frequency driver (network or storage with tight latency requirements).
- The system has thousands of callouts active and you want to understand the load.

In all other cases, write the callout naturally and trust the kernel to handle the load.

13.35 Looking Ahead: Bridge to Chapter 14

Chapter 14 is titled *Taskqueues and Deferred Work*. Its scope is the kernel's deferred-work framework as seen from a driver: how to move work out of a context that cannot safely run it (a callout callback, an interrupt handler, an epoch section) and into a context that can.

Chapter 13 prepared the ground in three specific ways.

First, you already know that callout callbacks run under a strict context contract: no sleeping, no sleepable-lock acquisition, no `uiomove`, no `copyin`, no `copyout`, and no `selwakeup` with a driver mutex held. You saw that contract enforced at the line in `myfirst_tick_source` where `selwakeup` was deliberately omitted because the callout context could not legally make the call. Chapter 14 introduces `taskqueue(9)`, which is the primitive the kernel offers for exactly this kind of hand-off: the callout enqueues a task, and the task runs in thread context where the omitted call is legal.

Second, you already know the drain-at-detach discipline. `callout_drain` ensures no callback is running when detach proceeds. Tasks have a matching primitive: `taskqueue_drain` waits until a specific task is neither pending nor running. The mental model is the same; the ordering grows by one step (callouts first, tasks second, then everything they affect).

Third, you already know the shape of `LOCKING.md` as a living document. Chapter 14 extends it with a Tasks section that names every task, its callback, its lifetime, and its place in the detach order. The discipline is the same; the vocabulary is a little wider.

Specific topics Chapter 14 will cover:

- The `taskqueue(9)` API: `struct task`, `TASK_INIT`, `taskqueue_create`, `taskqueue_start_threads`, `taskqueue_enqueue`, `taskqueue_drain`, `taskqueue_free`.
- The predefined system taskqueues (`taskqueue_thread`, `taskqueue_swi`, `taskqueue_fast`, `taskqueue_bus`) and when a private taskqueue is preferable.
- The coalescing rule: what happens when a task is enqueued while it is already pending.
- `struct timeout_task` and `taskqueue_enqueue_timeout` for deferred-and-scheduled work.
- Patterns that recur in real FreeBSD drivers, and the debugging story for when they go wrong.

You do not need to read ahead. Chapter 13 is sufficient preparation. Bring your `myfirst` driver (Stage 4 of Chapter 13), your test kit, and your WITNESS-enabled kernel. Chapter 14 starts where Chapter 13 ended.

A small closing reflection. You started this chapter with a driver that could not act on its own: every line of work was triggered by something the user did. You leave with a driver that has internal time, that periodically logs its state, that detects stalled drainage, that injects synthetic events for testing, and that tears down all of that infrastructure cleanly when the module is unloaded. That is a real qualitative leap, and the patterns transfer directly to every kind of driver Part 4 will introduce.

Take a moment. The driver you started Part 3 with knew how to handle one thread at a time. The driver you have now coordinates many threads, supports configurable timed work, and tears down without a race. From here, Chapter 14 adds the *task*, which is the missing piece for any driver whose timer callbacks need to trigger work that callouts cannot safely perform. Then turn the page.

13.35.1 A Final Aside on Time

One last thought before Chapter 14. You have spent two chapters on synchronization (Chapter 12) and one chapter on time (Chapter 13). The two are deeply related: synchronization is, at its core, about *when* events happen relative to each other, and time is the explicit measure of that. Locks serialize accesses; cvs coordinate waits; callouts fire at deadlines. All three are different ways of slicing the same underlying question: how do independent execution streams agree on order?

Chapter 14 adds a fourth piece: *context*. A callout fires at a precise moment, but the context in which it fires (no sleeping, no sleepable locks, no user-space copy) is narrower than what most real work needs. Deferred work via `taskqueue(9)` is the bridge from that narrow context into a thread context where the full set of kernel operations is legal.

The patterns transfer. The lock-aware initialization that callouts use for their callbacks is the same shape you will apply when deciding which lock a task callback acquires. The drain pattern that callouts use at detach is the same shape that tasks use at teardown. The “do little here, defer the rest” discipline that callouts demand is the discipline that Chapter 14 gives you a concrete tool to follow.

So when you reach Chapter 14, the framework will already be familiar. You will be adding one more primitive to your driver’s toolkit. Its rules compose cleanly with the callout rules you now know. The tools you have built (`LOCKING.md`, the seven-phase detach, the assert-and-check-attached pattern) will absorb the new primitive without growing brittle.

That is what makes Part 3 of this book work as a unit. Each chapter adds one more dimension to the driver's awareness of the world (concurrency, synchronization, time, deferred work), and each builds on the previous chapter's infrastructure. By the end of Part 3, your driver will be ready for Part 4 and the real hardware that lies beyond.

14 Taskqueues and Deferred Work

14.1 Reader Guidance & Outcomes

At the end of Chapter 13 your `myfirst` driver gained a small but real sense of internal time. It could schedule periodic work with `callout(9)`, emit a heartbeat, notice stalled drainage with a watchdog, and inject synthetic bytes with a tick source. Every callback obeyed a strict discipline: acquire the registered mutex, check `is_attached`, do short bounded work, maybe re-arm, release the mutex. That discipline is what made the timers safe. It is also what made them narrow.

Chapter 14 confronts the narrowness directly. A callout callback runs in a context that cannot sleep. It cannot call `uiomove(9)`, it cannot call `copyin(9)`, it cannot acquire a sleepable `sx(9)` lock, it cannot allocate memory with `M_WAITOK`, it cannot call `selwakeup(9)` while the sleep mutex is held. If the work a timer wants to trigger needs any of those, the timer has to hand the work off. The same limitation applies to the interrupt handlers you will meet in Part 4, and to several other constrained contexts that appear throughout the kernel. The kernel exposes a single primitive for the hand-off: `taskqueue(9)`.

A taskqueue is, at its simplest, a queue of small work items paired with one or more kernel threads that consume the queue. Your constrained context enqueues a task; the taskqueue's thread wakes up and runs the task's callback in process context, where the ordinary kernel rules apply. The task can sleep, it can allocate freely, it can touch sleepable locks. The taskqueue subsystem also knows how to coalesce bursts of enqueues, cancel pending work, wait for in-flight work at teardown, and schedule a task for a specific future moment. All of that fits on a small API surface, and all of it is exactly what a callout-using or interrupt-using driver needs.

This chapter teaches `taskqueue(9)` with the same care Chapter 13 gave to `callout(9)`. We begin with the shape of the problem, walk through the API, then evolve the `myfirst` driver through four stages that add task-based deferral to the existing timer infrastructure. By the end the driver will use a private taskqueue to move every piece of work that cannot run in a callout or interrupt context out of those contexts, and it will tear the taskqueue down at detach without leaking a stale task, waking a dead thread, or corrupting anything.

14.1.1 Why This Chapter Earns Its Own Place

You could pretend taskqueues do not exist. Instead of enqueueing a task, your callout could try to do the deferred work inline, accept the consequences of panicking the kernel the first time `WITNESS` notices a sleep with a spin lock held, and hope nobody ever loads your driver on a debug kernel. That is not a real option, and we will not humor it. The purpose of this chapter is to give you the honest alternative, the one the rest of the kernel actually uses.

You could also roll your own deferred-work framework with `kproc_create(9)` and a custom condition variable. That is technically possible and occasionally unavoidable, but it is almost always the wrong first choice. A custom thread is a weightier resource than a task, and it misses the observability that comes for free when you

use the shared framework. `ps(1)`, `procstat(1)`, `dtrace(1)`, `ktr(4)`, `wchan` traces, and `ddb(4)` all understand taskqueue threads. They do not understand your one-off helper unless you instrument it yourself.

Taskqueues are the right answer in almost every case where a driver needs to move work out of a constrained context. The cost of not knowing them is higher than the cost of learning them, and the cost of learning them is modest: the API is smaller than `callout(9)`'s, the rules are regular, and the idioms transfer directly across drivers. Once the mental model clicks, you will start recognising the pattern in nearly every driver under `/usr/src/sys/dev/`.

14.1.2 Where Chapter 13 Left the Driver

A quick checkpoint before we go further. Chapter 14 extends the driver produced at the end of Chapter 13 Stage 4, not any earlier stage. If any of the items below feels uncertain, return to Chapter 13 before starting this chapter.

- Your `myfirst` driver compiles cleanly and identifies itself as version `0.7-timers`.
- It has three callouts declared in the `softc`: `heartbeat_co`, `watchdog_co`, and `tick_source_co`.
- Each callout is initialised with `callout_init_mtx(&co, &sc->mtx, 0)` in `myfirst_attach` and drained with `callout_drain` in `myfirst_detach` after `is_attached` has been cleared.
- Each callout has an interval `sysctl` (`heartbeat_interval_ms`, `watchdog_interval_ms`, `tick_source_interval_ms`) that defaults to zero (disabled) and reflects the enable/disable transitions in its handler.
- The detach path runs in the documented order: refuse on `active_fhs`, clear `is_attached`, broadcast both `cvs`, drain `selinfo`, drain all callouts, destroy devices, free `sysctls`, destroy `cbuf` and counters and `cvs` and the `sx` and the mutex.
- Your `LOCKING.md` has a Callouts section that names each callout, its callback, its lock, and its lifetime.
- The Chapter 13 stress kit (Chapter 12 testers plus the Chapter 13 timer exercisers) builds and runs cleanly with `WITNESS` and `INVARIANTS` enabled.

That driver is the shape we extend. Chapter 14 does not rework any of those structures. It adds a new column to the `softc`, a new initialisation call, a new teardown call, and small changes to the three callout callbacks and one or two other places where the driver would benefit from moving work out of a constrained context.

14.1.3 What You Will Learn

By the end of this chapter you will be able to:

- Explain why some work cannot be done inside a callout callback or an interrupt handler, and recognise the operations that force a hand-off to a different context.
- Describe the three things a taskqueue is: a queue of `struct task`, a thread (or small pool of threads) that consumes the queue, and an enqueue/dispatch policy that ties the two together.
- Initialise a task with `TASK_INIT(&sc->foo_task, 0, myfirst_foo_task, sc)`, understand what each argument means, and place the call in the correct stage of attach.
- Enqueue a task with `taskqueue_enqueue(tq, &sc->foo_task)` from a callout callback, from a `sysctl` handler, from a read or write path, or from any other driver code where deferral is the right answer.
- Choose between the predefined system taskqueues (`taskqueue_thread`, `taskqueue_swi`, `taskqueue_swi_giant`, `taskqueue_fast`, `taskqueue_bus`) and a private taskqueue you create with `taskqueue_create` and populate with `taskqueue_start_threads`.

- Understand the coalescing contract: when a task is enqueued while it is already pending, the kernel increments `ta_pending` instead of linking it twice, and the callback is handed the final pending count so it can batch.
- Use the struct `timeout_task` variant with `taskqueue_enqueue_timeout` to schedule a task for a specific future moment, and drain it correctly with `taskqueue_drain_timeout`.
- Block and unblock a taskqueue around delicate shutdown steps, and quiesce a taskqueue when you need a guarantee that no task is currently running anywhere in the queue.
- Drain every task a driver owns at detach, in the right order, without deadlocking against the callouts and cvs you already drain.
- Separate the concerns of timer code and task code inside your driver source, so a new reader can tell which work runs in which context just by looking at the file.
- Recognise and apply the network-driver pattern that uses `epoch(9)` and `grouptaskqueues` for lockless read paths, at the level of knowing when to reach for them and when not to.
- Debug a taskqueue-using driver with `procstat -t, ps ax, dtrace -l`, and `ktr(4)`, and interpret what each tool shows you.
- Tag the driver as version `0.8-taskqueues` and document the deferral policy in `LOCKING.md` so that the next person who inherits the driver can read it.

That is a long list. Most items build on each other, so the progression inside the chapter is the natural path.

14.1.4 What This Chapter Does Not Cover

Several adjacent topics are explicitly deferred so Chapter 14 stays focused.

- **Interrupt handlers as a primary topic.** Part 4 introduces `bus_setup_intr(9)` and the split between `FILTER` and `ITHREAD` handlers. Chapter 14 mentions interrupt context when it explains why deferred work matters, and the patterns it teaches transfer directly from callouts to real interrupt handlers, but the interrupt API itself is Part 4's job.
- **The full condition-variable and semaphore story.** Chapter 15 extends the synchronisation vocabulary with counting semaphores, signal-interruptible blocking, and cross-component handshakes that coordinate timers, tasks, and user threads. Chapter 14 uses the existing `cv` infrastructure as-is and does not add new synchronisation primitives beyond what taskqueues themselves bring.
- **Deep coverage of grouptaskqueues and iflib.** The `taskqgroup` family exists and this chapter explains when it is the right answer, but the complete story belongs to the network drivers in Part 6 (Chapter 28). The introduction here is intentionally light.
- **Hardware-driven DMA completion paths.** A taskqueue is the natural place to finish a DMA transfer after the hardware signalled completion via an interrupt, and we mention that pattern, but the mechanics of DMA buffer management wait until the bus-space and DMA chapters.
- **Workloops, kthreads for per-CPU polling, and the advanced scheduler hooks.** These are real parts of the kernel's deferred-work landscape, but they are specialised, and a driver hits them rarely. When they matter, the chapters that need them introduce them.

Staying inside those lines keeps the chapter's mental model coherent. Chapter 14 gives you one good tool, taught carefully. Later chapters give you the neighbouring tools and the real hardware contexts that justify using them.

14.1.5 Estimated Time Investment

- **Reading only:** about three hours. The API surface is smaller than `callout(9)`'s, but the interaction between a taskqueue and the rest of the driver's locking story takes a little time to settle.
- **Reading plus typing the worked examples:** six to eight hours over two sessions. The driver evolves in four stages; each stage changes roughly one concern.
- **Reading plus all labs and challenges:** ten to fourteen hours over three or four sessions, including the time needed to observe taskqueue threads under load with `procstat`, `dtrace`, and a stress workload.

If you find the ordering rules at the start of Section 4 disorienting, that is normal. The detach sequence with callouts, cvs, sel handlers, and now tasks has four pieces that must compose. We will walk through the order once, state it, justify it, and reuse it.

14.1.6 Prerequisites

Before starting this chapter, confirm:

- Your driver source matches Chapter 13 Stage 4 (`stage4-final`). The starting point assumes the three callouts, the three interval sysctls, the `is_attached` discipline in every callback, and the documented detach order.
- Your lab machine runs FreeBSD 14.3 with `/usr/src` on disk and matching the running kernel. Several of the source references in this chapter are things you should actually open and read.
- A debug kernel with `INVARIANTS`, `WITNESS`, `WITNESS_SKIPSPIN`, `DDB`, `KDB`, and `KDB_UNATTENDED` is built, installed, and booting cleanly.
- Chapter 13 feels comfortable. The lock-aware callout, the `is_attached` discipline in callbacks, and the detach ordering are assumed knowledge here.
- You have run the Chapter 13 stress kit at least once with every timer enabled and seen it pass cleanly.

If any of the above is shaky, fixing it now is a better investment than pushing through Chapter 14 and trying to debug from a moving foundation. The Chapter 14 patterns are specifically designed to compose with the Chapter 13 patterns; starting from a Chapter 13 driver that is not quite right makes every Chapter 14 step harder.

14.1.7 How to Get the Most Out of This Chapter

Three habits will pay off quickly.

First, keep `/usr/src/sys/kern/subr_taskqueue.c` and `/usr/src/sys/sys/taskqueue.h` bookmarked. The header is short, about two hundred lines, and it is the canonical summary of the API. The implementation file is about a thousand lines, well commented, and reading `taskqueue_run_locked` carefully pays for itself the first time you have to reason about what a task's pending count actually means. Ten minutes with the header now buys ten hours of confidence later.

Second, run every code change under `WITNESS`. The taskqueue subsystem has its own lock (a spin mutex or a sleep mutex, depending on whether the queue was created with `taskqueue_create` or `taskqueue_create_fast`), and it interacts with your driver's locks in ways that `WITNESS` understands. A misplaced lock acquisition inside a task callback is exactly the kind of bug `WITNESS` catches instantly on

a debug kernel and corrupts silently on a production kernel. Do not run Chapter 14 code on the production kernel until it passes the debug kernel.

Third, type the changes by hand. The companion source under `examples/part-03/ch14-taskqueues-and-deferred-work/` is the canonical version, but muscle memory is worth more than reading. The chapter introduces small incremental edits; mirror that small-step rhythm in your own copy of the driver. When the test environment passes at one stage, commit that version and move on; when a step breaks, the previous commit is your recovery point.

14.1.8 Roadmap Through the Chapter

The sections in order are:

1. Why use deferred work in a driver. The shape of the problem: what cannot be done in callouts, interrupts, and other constrained contexts; the real-world cases that force the hand-off.
2. Introduction to taskqueue(9). The structures, the API, the predefined queues, and the comparison with callouts.
3. Deferring work from a timer or simulated interrupt. The first refactor, Stage 1: add a single task that a callout enqueues.
4. Taskqueue setup and cleanup. Stage 2: create a private taskqueue, wire the detach sequence, and audit the result against WITNESS.
5. Prioritisation and coalescing of work. Stage 3: use the `ta_pending` coalescing behaviour deliberately for batching, introduce `taskqueue_enqueue_timeout` for scheduled tasks, and discuss priorities.
6. Real-world patterns using taskqueues. A tour of patterns that recur in real FreeBSD drivers, presented as small recipes you can lift into your own code.
7. Debugging taskqueues. The tools, the common mistakes, and a guided break-and-fix on a realistic scenario.
8. Refactoring and versioning. Stage 4: consolidate the driver into a coherent whole, bump the version to `0.8-taskqueues`, and extend `LOCKING.md`.

After the eight main sections, we cover epoch(9), group taskqueues, and per-CPU taskqueues at a light, introductory level, then hands-on labs, challenge exercises, a troubleshooting reference, a wrapping-up section, and a bridge to Chapter 15.

If this is your first pass, read linearly and do the labs in order. If you are revisiting, Sections 4, 6, and 8 stand alone and make good single-sitting reads.

14.2 Section 1: Why Use Deferred Work in a Driver?

Chapter 13 ended with a driver whose callouts did every job the callback could safely do. The heartbeat printed a one-line status report. The watchdog recorded a single count and optionally printed a warning. The tick source wrote a single byte to the circular buffer and signalled a condition variable. Each callback took microseconds, held the mutex for those microseconds, and returned. That is the callout contract at its best: small, predictable, lock-aware, and cheap.

Real driver work does not always fit inside that contract. Some tasks want to run at the same pace as the timer that notices the need for them, but they want to do things the timer cannot safely do. Other tasks are triggered

by a different constrained context (an interrupt handler, for example, or a filter routine in the network stack) but have the same “cannot do that here” problem. This section surveys the shape of the problem: what cannot be done in constrained contexts, what kinds of work drivers want to defer, and what options the kernel offers for getting the work to a place where it can actually run.

14.2.1 The Callout Contract, Re-Read

A short re-reading of the callout rule, because the taskqueue story is entirely about the things callouts cannot do.

A `callout(9)` callback runs in one of two modes. The default mode is `dispatch-from-callout-thread`: the kernel’s dedicated callout thread for that CPU wakes up on a hardware-clock boundary, walks the callout wheel, finds the callbacks whose deadlines have arrived, and calls them one by one. The alternative mode, `C_DIRECT_EXEC`, runs the callback directly inside the hardware-clock interrupt handler itself. Your driver rarely picks the alternative; the default is what almost all drivers use.

In both modes the callback runs with the callout’s registered lock held (for the `callout_init_mtx` family) and cannot cross certain context boundaries. It must not sleep. Sleeping means calling any primitive that can deschedule the thread and block for an indefinite period waiting for a condition. `mtx_sleep`, `cv_wait`, `msleep`, `sx_slock`, `malloc(..., M_WAITOK)`, `uiomove`, `copyin`, and `copyout` all sleep on their slow path. `selwakeup(9)` does not sleep per se, but it takes the per-selinfo mutex, which can be the wrong mutex for the context the callout is running in, and standard practice is to call it with no driver mutex held anyway. None of those calls belong inside a callout callback.

Those are hard rules at the kernel level. `INVARIANTS` and `WITNESS` catch many of the violations at runtime. A few of them corrupt the kernel silently, in ways that are hard to debug later. In all cases, a driver that wants the effect of one of those operations must make the call from a context that allows it. That context is the context a taskqueue provides.

The rest of this section expands on the same observation from different angles: what sort of work a driver wants to defer, why the constrained contexts deserve the constraints, and which FreeBSD facilities compete for the job.

14.2.2 Constrained Contexts, Not Just Callouts

The callout is the first constrained context a `myfirst`-style driver meets, but it is not the only one. Several other places in the kernel run code that cannot sleep or cannot do certain kinds of allocation. A driver that wants to take action from any of them faces the same “defer it” decision.

Hardware interrupt filters. When a real device raises an interrupt, the kernel runs a filter routine synchronously on the CPU that took the interrupt. Filters cannot sleep, cannot acquire sleep mutexes, and cannot call most of the kernel’s normal APIs. They are usually split into a tiny filter (read a status register, decide whether the interrupt is ours) that runs in hardware context, plus an associated `ithread` (interrupt thread) that runs the real work in a full thread context. We will meet the precise filter/`ithread` split when Part 4 introduces `bus_setup_intr(9)`, but the structural lesson is clear even now: interrupt filters are another place where work must be handed off somewhere else.

Network packet-input paths. Parts of the `ifnet(9)` reception path run under `epoch(9)` protection, which restricts the kind of lock acquisitions and sleeping operations that are safe. Network drivers frequently enqueue a task when they want to do non-trivial work that belongs in process context.

taskqueue_fast and taskqueue_swi callbacks. Even when you are already inside a task callback, if the task is running on a spin-mutex-backed queue (`taskqueue_fast`) or a software-interrupt queue (`taskqueue_swi`), the same no-sleeping rule applies as for the originating context. Task callbacks on the default `taskqueue_thread` have no such restriction; they run in full thread context and can sleep freely. That distinction matters and we will return to it in Section 2.

epoch(9) read sections. A code path bracketed by `epoch_enter()` and `epoch_exit()` cannot sleep. Network drivers use this pattern heavily to make read paths lock-free; write-side work is deferred to outside the epoch. Chapter 14 covers epoch at an introductory level in the later “Additional topics” part of the chapter.

The common thread across all those contexts is that something about the surrounding environment forbids thread-context operations. The “something” differs (a spin lock, a filter context, an epoch section, a software-interrupt dispatch), but the remedy is the same: enqueue a task to be run later by a thread that is not in the constrained context.

14.2.3 Real-World Reasons to Defer

Short tours of the kinds of work drivers push out of constrained contexts. Recognising the shapes now gives you vocabulary for the patterns Section 6 will develop.

Non-trivial selwakeup(9). `selwakeup` is the kernel’s notify-all-select/poll-waiters call. Traditional wisdom says to call it with no driver mutex held, and never from a context that has a spin lock held. A callout callback holds a mutex; an interrupt filter holds nothing, but is itself in a bad place. Drivers that want to notify pollers from those contexts typically enqueue a task whose only job is to call `selwakeup`.

copyin and copyout following a hardware event. After an interrupt signals that a DMA transfer has completed, the driver may want to copy data to or from a user-space buffer whose address was registered with a previous `ioctl`. Neither `copyin` nor `copyout` is legal in interrupt context. The driver schedules a task whose callback does the copy in process context.

Reconfiguration that requires a sleepable lock. The driver’s configuration is often protected by an `sx(9)` lock, which can sleep. A callout or interrupt cannot acquire a sleepable lock directly. If a timer-driven decision implies a configuration change, the timer enqueues a task; the task acquires the `sx` lock and performs the change.

Retrying a failed operation after a backoff. Hardware operations sometimes fail transiently. The sensible response is to wait some interval and retry. An interrupt handler cannot block; it enqueues a `timeout_task` with a delay equal to the backoff interval. The timeout task fires later in thread context, retries the operation, and if it fails again reschedules itself with a longer delay.

Logging a non-trivial event. Kernel `printf(9)` is surprisingly tolerant of weird contexts, but `log(9)` and its friends are not. A driver that wants to emit a multi-line diagnostic from an interrupt context writes the bare minimum in the handler (a flag, a counter increment) and schedules a task to do the real logging later.

Draining or reconfiguring a long hardware queue. A network driver that detects head-of-line blocking may want to walk its transmit ring, free completed descriptors, and reset per-descriptor state. The work is bounded but nontrivial. Doing it inline in the interrupt path monopolises a CPU in a bad context. Doing it in a task lets the interrupt return immediately and the real work happen on a thread.

Deferred teardown. When a driver detaches while some object still has outstanding references, the driver cannot free the object immediately. A common pattern: defer the free to a task that fires after the reference count is known to be zero, or after a grace period that is long enough for any in-flight references to drain.

All those cases share the structure: the constrained context detects a need, possibly records a small amount of state, and enqueues a task. The task runs later in thread context, does the real work, and optionally re-enqueues itself or schedules a follow-up.

14.2.4 Polling Versus Deferred Execution

A reasonable question at this point: if the constrained context cannot do the work, why not arrange for the work to happen somewhere entirely outside the constrained context? Why not have a single dedicated kernel thread that polls for “anything to do” and wakes up when it sees state that requires action?

That is, in effect, what a taskqueue does. A taskqueue thread sleeps until work is available and wakes up to process it. The difference between “a taskqueue” and “a hand-rolled polling thread” is that the taskqueue framework solves the surrounding logistics for you. Enqueue is a single atomic operation. The task structure holds the callback and context directly, so you do not have to design a “work queue entry” type. Coalescing of redundant enqueues is automatic. Draining a task is a single call. Teardown is a single call. Observability via the standard tools comes for free.

A hand-rolled polling thread can do the same work, and in extreme cases it is the right choice (if, for example, the work has hard real-time constraints, or if it is part of a subsystem that needs a dedicated priority). For ordinary driver work, reaching past `taskqueue(9)` is almost always a mistake.

A separate but related question: why not just spin up a new kernel thread for each deferred operation? That is extremely expensive: creating a thread takes time, allocates a full kernel stack, and hands the new thread to the scheduler. For work that happens repeatedly, the sensible design is to reuse a thread, which is exactly what a taskqueue provides. For work that happens once, you might `kproc_create(9)` and have the new thread exit when done, but even then a taskqueue with `taskqueue_drain` is usually simpler and nearly as cheap.

14.2.5 FreeBSD’s Solutions

The kernel offers a small family of facilities for deferred work. Chapter 14 focuses on one of them (`taskqueue(9)`) and mentions the others at the right level of detail for a driver writer to know when they are appropriate. A short tour now; the sections that follow expand on each one as it becomes relevant.

taskqueue(9). A queue of `struct task` entries and one or more kernel threads (or a software-interrupt context) that consume the queue. The dominant choice for deferred driver work. Covered in depth throughout this chapter.

epoch(9). A lockless read-synchronisation mechanism that network drivers use to allow readers to walk shared data structures without locks. Writers defer cleanup via `epoch_call` or `epoch_wait`. Not a general deferred-work mechanism for arbitrary drivers, but important enough to introduce later in this chapter so that you recognise it when you see it in network-driver code.

Grouptaskqueues. A scalable variation on taskqueues where a group of related tasks share a pool of per-CPU worker threads. Network drivers use this heavily; most other drivers do not. Introduced later in this chapter.

kproc_create(9) / kthread_add(9). Direct creation of a kernel thread. Useful when the deferred work is a long-running loop that does not fit the “short task” shape, and when the work deserves a dedicated priority or CPU affinity. Almost always overkill for simple deferral; a taskqueue is preferred.

Dedicated SWI (software interrupt) handlers via swi_add(9). A way to register a function that runs in a software-interrupt context. The system taskqueues (taskqueue_swi, taskqueue_swi_giant, taskqueue_fast) are built on top of this mechanism. Driver code rarely calls swi_add directly; the taskqueue layer is the right abstraction.

The callout itself, rescheduled for “zero-from-now”. A pattern that does not work: you cannot “escape” the callout context by scheduling another callout, because the next callout still runs in callout context. Recognising that this is a dead end is itself useful. Callouts schedule the moment; taskqueues provide the context.

For the rest of Chapter 14, unless we say otherwise, “defer to a task” or “enqueue a task” means “enqueue a struct task onto a struct taskqueue”.

14.2.6 When Deferral Is the Wrong Answer

Deferral is a tool, not a default. Several situations benefit from doing the work in place rather than deferring it.

The work is genuinely short and safe for the current context. Logging a one-line statistic with `device_printf(9)` from a callout is fine. So is incrementing a counter. So is signalling a cv. Deferring these trivialities to a task costs more than doing them. Only defer when the work actually does not belong in the current context.

Timing matters and the deferral introduces variance. A task does not run instantly. It runs when the taskqueue’s thread is next scheduled, which may be microseconds or milliseconds away depending on system load. If the work has tight timing requirements (acknowledging a hardware event within a deadline, for example), deferral may miss the deadline. For such work you need a faster mechanism (a hardware-level completion, a `C_DIRECT_EXEC` callout, or an SWI) or a different design.

The deferral would add a hop for no benefit. If your interrupt handler’s only work is already safe to do in interrupt context, adding a task round-trip doubles the latency without improving anything. Only defer the parts of the work that need to be deferred.

The work requires a specific thread. If the work needs to run as a specific user process (for example, to use that process’s file-descriptor table), a generic taskqueue thread is the wrong place. That situation is rare in drivers, but it exists.

For everything else, deferral via a taskqueue is the right answer, and the rest of the chapter is about how to do it well.

14.2.7 A Worked Example: Why the Tick Source Cannot Wake Pollers

A concrete example from the Chapter 13 driver, worth slowing down for because it is the first place Chapter 14 makes a real change.

The Chapter 13 `tick_source` callback, from `stage4-final/myfirst.c`, looks like this:

```

static void
myfirst_tick_source(void *arg)
{
    struct myfirst_softc *sc = arg;
    size_t put;
    int interval;

    MYFIRST_ASSERT(sc);
    if (!sc->is_attached)
        return;

    if (cbuf_free(&sc->cb) > 0) {
        put = cbuf_write(&sc->cb, &sc->tick_source_byte, 1);
        if (put > 0) {
            counter_u64_add(sc->bytes_written, put);
            cv_signal(&sc->data_cv);
            /* selwakeup omitted: cannot be called from a
             * callout callback while sc->mtx is held. */
        }
    }
    ...
}

```

That comment at the bottom is not hypothetical. `selwakeup(9)` enters the per-selinfo mutex and may call into the kqueue subsystem, which is not safe to do inside a callout callback that holds a different driver mutex. A `select(2)/poll(2)` user program waiting on `/dev/myfirst` for readability therefore does not get notified when the tick source deposits a byte. The program only wakes when some other path calls `selwakeup`, for example when another thread's `write(2)` arrives.

That is a real bug in the Chapter 13 driver. We left it unfixed in Chapter 13 because fixing it required a primitive we had not yet introduced. Chapter 14 introduces that primitive and fixes the bug.

The fix is small. Add a `struct task` to the `_softc`. Initialise it in `attach`. Instead of omitting `selwakeup` from the `tick_source` callback, enqueue the task; the task runs in thread context, with no driver mutex held, and calls `selwakeup` safely. Drain the task in `detach` after `is_attached` has been cleared, before freeing the selinfo.

We will walk through every step of that change in Section 3. For now the point is that the change is mechanical, and its necessity is not contrived. Chapter 14's first real job is to give you the tool this kind of bug needs.

14.2.8 A Small Mental Model

A useful picture, offered once and referenced later.

Think of your driver as being made of two kinds of code. The first kind is the code that runs because somebody asked for it: the `read(2)` handler, the `write(2)` handler, the `ioctl(2)` handler, the `sysctl` handlers, the `open` and `close` handlers. That code runs in thread context, with ordinary rules, and it can sleep, allocate, and touch any lock. Call it “thread-context code”.

The second kind is the code that runs because time or the hardware said so: a callout callback, an interrupt filter, an epoch-protected read. That code runs in a constrained context, with a narrower set of rules, and it must keep its work short and non-sleeping. Call it “edge-context code”.

Most real work belongs in thread-context code. Most of what edge-context code actually needs to do is: notice the edge, record a tiny amount of state, and hand the work to thread-context code. A taskqueue is the hand-off. The task callback runs in thread-context code, because the taskqueue’s thread is in thread context. Everything the callback does follows the ordinary rules.

This mental model lets you read every subsequent section as variations on a single idea: edge-context code detects, thread-context code acts, and the taskqueue is the seam between them. Once you see the driver that way, the rest of the chapter is engineering details.

14.2.9 Wrapping Up Section 1

Some work must run in a constrained context (callouts, interrupt filters, epoch sections). The rules of those contexts forbid sleeping, heavy allocation, sleepable-lock acquisition, and several other common operations. Drivers with real responsibilities frequently need to do exactly those operations in response to events that arrive in constrained contexts. The remedy is to enqueue a task and let a worker thread do the real work in a context where the rules allow it.

The kernel exposes that pattern as `taskqueue(9)`. The API is small, the idioms are regular, and the tool composes cleanly with the callouts and synchronisation primitives you already know. Section 2 introduces the primitive.

14.3 Section 2: Introduction to `taskqueue(9)`

`taskqueue(9)` is, like most of the kernel’s well-aged subsystems, a small API on top of a careful implementation. The data structures are short, the lifecycle is regular (init, enqueue, run, drain, free), and the rules are explicit enough that you can verify your usage by reading the source. This section walks through the structures, names the API, lists the predefined queues the kernel provides for free, and compares taskqueues with the callouts from Chapter 13 so you can see when each is the right tool.

14.3.1 The Task Structure

The data structure is in `/usr/src/sys/sys/_task.h`:

```
typedef void task_fn_t(void *context, int pending);

struct task {
    STAILQ_ENTRY(task) ta_link;    /* (q) link for queue */
    uint16_t ta_pending;          /* (q) count times queued */
    uint8_t ta_priority;          /* (c) Priority */
    uint8_t ta_flags;             /* (c) Flags */
    task_fn_t *ta_func;           /* (c) task handler */
};
```



```
void      *ta_context;          /* (c) argument for handler */
};
```

The fields split into two groups. The (q) fields are managed by the taskqueue under its own internal lock; driver code does not touch them directly. The (c) fields are const-after-init; driver code sets them once via an initialiser and never modifies them again.

`ta_link` is the list linkage used when the task is enqueued. Unused when the task is idle.

`ta_pending` is the coalescing counter. When the task is first enqueued it goes from zero to one and the task is placed on the list. If it is enqueued again before the callback runs, the counter simply increments and the task stays on the list once. When the callback eventually runs, the final pending count is passed as the second argument to the callback, and the counter is reset to zero. The worst mistake you can make with `ta_pending` is to assume that a task will run *N* times if you enqueue it *N* times; it will not. It will run once and the callback will know it was enqueued *N* times. Section 5 covers the design implications in detail.

`ta_priority` orders tasks inside a single queue. Higher priority tasks run before lower priority tasks. For most drivers the value is zero (ordinary priority) and the queue is effectively FIFO.

`ta_flags` is a small bitfield. The kernel uses it to record whether the task is currently enqueued and, for network tasks, whether the task should be run inside the network epoch. Driver code does not touch it after `TASK_INIT` or `NET_TASK_INIT` has set it.

`ta_func` is the callback function. Its signature is `void (*)(void *context, int pending)`. The first argument is whatever you stored in `ta_context` at init time; the second is the coalescing count.

`ta_context` is the callback's argument. For a device-driver task this is almost always the softc pointer.

The structure is 32 bytes on amd64, plus or minus padding. You embed one per deferred-work pattern into your softc. A driver with three deferral paths has three `struct task` members.

14.3.2 Initialising a Task

The canonical macro is `TASK_INIT`, in `/usr/src/sys/sys/taskqueue.h`:

```
#define TASK_INIT_FLAGS(task, priority, func, context, flags) do {      \
    (task)->ta_pending = 0;                                           \
    (task)->ta_priority = (priority);                                \
    (task)->ta_flags = (flags);                                       \
    (task)->ta_func = (func);                                         \
    (task)->ta_context = (context);                                   \
} while (0)

#define TASK_INIT(t, p, f, c)    TASK_INIT_FLAGS(t, p, f, c, 0)
```

A typical call from a driver's attach routine looks like:

```
TASK_INIT(&sc->selwake_task, 0, myfirst_selwake_task, sc);
```

The arguments read as: “initialise this task, at ordinary priority zero, to run `myfirst_selwake_task(sc, pending)` when it fires”. That is the whole initialisation ritual. There is no corresponding “destroy” call; a task becomes idle again when its callback finishes, and goes out of scope when the surrounding softc is freed.

For network-path tasks there is a variant, `NET_TASK_INIT`, that sets the `TASK_NETWORK` flag so the taskqueue knows to run the callback inside the `net_epoch_preempt` epoch:

```
#define NET_TASK_INIT(t, p, f, c) TASK_INIT_FLAGS(t, p, f, c, TASK_NETWORK)
```

Unless you are writing a network driver, `TASK_INIT` is the one you use. Chapter 14 uses `TASK_INIT` throughout and returns to `NET_TASK_INIT` only in the “Additional topics” section.

14.3.3 The Taskqueue Structure, From a Driver’s Point of View

From a driver’s point of view, a taskqueue is a `struct taskqueue *`. The pointer is either a predefined global (`taskqueue_thread`, `taskqueue_swi`, `taskqueue_bus`, etc.) or one the driver created with `taskqueue_create` and stored in its softc. In both cases the pointer is opaque. All interactions go through API calls. The only internal we care about in the chapter is the fact that the taskqueue holds its own lock, which it acquires on enqueue and when its worker thread pulls tasks off the list.

For completeness, the definition (from `/usr/src/sys/kern/subr_taskqueue.c`):

```
struct taskqueue {
    STAILQ_HEAD(, task)    tq_queue;
    LIST_HEAD(, taskqueue_busy) tq_active;
    struct task            *tq_hint;
    u_int                  tq_seq;
    int                    tq_callouts;
    struct mtx_padalign    tq_mutex;
    taskqueue_enqueue_fn    tq_enqueue;
    void                   *tq_context;
    char                   *tq_name;
    struct thread          **tq_threads;
    int                    tq_tcount;
    int                    tq_spin;
    int                    tq_flags;
    ...
};
```

`tq_queue` is the pending-task list. `tq_active` records which tasks are currently running, which the drain logic uses to wait for completion. `tq_mutex` is the taskqueue’s own lock. `tq_threads` is the array of worker threads, of size `tq_tcount`. `tq_spin` records whether the mutex is a spin mutex (for taskqueues created with

`taskqueue_create_fast`) or a sleep mutex (for taskqueues created with `taskqueue_create`). `tq_flags` records shutdown state.

You do not touch any of those fields from driver code. They are shown here once so that the API calls in the rest of the section have a concrete referent. The rest of the chapter treats the taskqueue as opaque.

14.3.4 The API, One Function at a Time

The public functions are declared in `/usr/src/sys/sys/taskqueue.h`. A driver typically uses fewer than a dozen of them. We walk through the important ones now, grouped by purpose.

Creating and destroying a taskqueue.

```
struct taskqueue *taskqueue_create(const char *name, int mflags,
    taskqueue_enqueue_fn enqueue, void *context);

struct taskqueue *taskqueue_create_fast(const char *name, int mflags,
    taskqueue_enqueue_fn enqueue, void *context);

int taskqueue_start_threads(struct taskqueue **tqp, int count, int pri,
    const char *name, ...);

void taskqueue_free(struct taskqueue *queue);
```

`taskqueue_create` creates a taskqueue that uses a sleep mutex internally. Tasks enqueued on it run in a context where sleeping is legal (assuming they are dispatched via `taskqueue_thread_enqueue` and `taskqueue_start_threads`). This is the right choice for almost every driver taskqueue.

`taskqueue_create_fast` creates a taskqueue that uses a spin mutex internally. Required only if you intend to enqueue from a context where a sleep mutex would be wrong (for example, from inside a spin mutex or a filter interrupt). Driver code rarely needs this; the predefined `taskqueue_fast` exists for the cases that do.

The enqueue callback is called by the taskqueue layer when a task is added to an otherwise-empty queue, and is the way the layer “wakes” a consumer. For queues serviced by kernel threads the enqueue function is `taskqueue_thread_enqueue`, which the kernel provides. For software-interrupt-serviced queues the kernel provides `taskqueue_swi_enqueue`. Driver code almost always passes `taskqueue_thread_enqueue` here.

The context argument is passed back to the enqueue callback. When using `taskqueue_thread_enqueue` the convention is to pass `&your_taskqueue_pointer`, so the function can find the taskqueue it is waking. The Chapter 14 examples follow this convention literally.

`taskqueue_start_threads` creates count kernel threads that run the `taskqueue_thread_loop` dispatcher, each sleeping on the queue until a task arrives. The `pri` argument is the thread’s priority. `PWAIT` (defined in `/usr/src/sys/sys/priority.h`, numerically 76) is the ordinary choice for driver taskqueues; network drivers often pass `PI_NET` (numerically 4) to run at interrupt-adjacent priority. Chapter 14’s worker threads use `PWAIT`.

`taskqueue_free` shuts down the taskqueue. It drains all pending and running tasks, terminates the worker threads, and frees the internal state. It must be called with no tasks pending that have not yet been drained; after it returns, the `struct taskqueue *` is invalid and must not be used.

Initialising a task. TASK_INIT as shown above. There is no counterpart “destroy” because the task structure is caller-owned.

Enqueuing a task.

```
int taskqueue_enqueue(struct taskqueue *queue, struct task *task);
int taskqueue_enqueue_flags(struct taskqueue *queue, struct task *task,
    int flags);
int taskqueue_enqueue_timeout(struct taskqueue *queue,
    struct timeout_task *timeout_task, int ticks);
int taskqueue_enqueue_timeout_sbt(struct taskqueue *queue,
    struct timeout_task *timeout_task, sbintime_t sbt, sbintime_t pr,
    int flags);
```

taskqueue_enqueue is the workhorse. It links the task onto the queue and wakes the worker thread. If the task is already pending, it increments ta_pending and returns. Returns zero on success; rarely fails.

taskqueue_enqueue_flags is the same, with optional flags:

- TASKQUEUE_FAIL_IF_PENDING makes the enqueue return EEXIST instead of coalescing if the task is already pending.
- TASKQUEUE_FAIL_IF_CANCELING makes the enqueue return EAGAIN if the task is currently being cancelled.

The default taskqueue_enqueue silently coalesces; the flag variant lets you detect the situation when that matters.

taskqueue_enqueue_timeout schedules a struct timeout_task to fire after the given number of ticks. Behind the scenes it uses an internal callout whose callback enqueues the underlying task on the taskqueue when the delay elapses. The sbt variant takes sbintime for sub-tick precision.

Cancelling a task.

```
int taskqueue_cancel(struct taskqueue *queue, struct task *task,
    u_int *pendp);
int taskqueue_cancel_timeout(struct taskqueue *queue,
    struct timeout_task *timeout_task, u_int *pendp);
```

taskqueue_cancel removes a pending task from the queue if it has not yet started running, and writes the previous pending count to *pendp if that pointer is non-NULL. If the task is currently running, the function returns EBUSY and does not wait; you must follow up with taskqueue_drain if you need to wait.

taskqueue_cancel_timeout is the same for timeout tasks.

Draining a task.

```
void taskqueue_drain(struct taskqueue *queue, struct task *task);
void taskqueue_drain_timeout(struct taskqueue *queue,
    struct timeout_task *timeout_task);
void taskqueue_drain_all(struct taskqueue *queue);
```

`taskqueue_drain(tq, task)` blocks until the given task is no longer pending and no longer running. If the task was pending, the drain waits for it to run and complete. If the task was running, the drain waits for the current invocation to return. If the task was idle, the drain returns immediately. This is the call you use at detach for every task your driver owns.

`taskqueue_drain_timeout` is the same for timeout tasks.

`taskqueue_drain_all` drains every task and every timeout task in the taskqueue. Useful when you own a private taskqueue and want to be sure it is entirely quiet before freeing it. `taskqueue_free` itself does the equivalent work internally, so `taskqueue_drain_all` is not strictly required before `taskqueue_free`, but it is useful when you want to quiesce a taskqueue without destroying it.

Blocking and unblocking.

```
void taskqueue_block(struct taskqueue *queue);
void taskqueue_unblock(struct taskqueue *queue);
void taskqueue_quiesce(struct taskqueue *queue);
```

`taskqueue_block` stops the queue from running new tasks. Already-running tasks complete; newly enqueued tasks accumulate but do not run until `taskqueue_unblock` is called. The pair is useful for temporarily freezing a queue during a delicate transition without tearing it down.

`taskqueue_quiesce` waits for the currently running task (if any) to finish and for the queue to be empty of pending tasks. Equivalent to “drain everything, but do not destroy”. Safe to call with the queue running.

Membership check.

```
int taskqueue_member(struct taskqueue *queue, struct thread *td);
```

Returns true if the given thread is one of the taskqueue’s worker threads. Useful inside a task callback when you want to branch on “am I running on my own taskqueue”, though the more common idiom is to use `curthread` against a stored thread pointer.

That is the whole API a driver ordinarily uses. A handful of less-common functions exist (`taskqueue_set_callback` for init/shutdown hooks, `taskqueue_poll_is_busy` for polling-style checks) but most drivers never touch them.

14.3.5 The Predefined Taskqueues

The kernel provides a small set of preconfigured taskqueues for drivers that do not need a private one. They are declared in `/usr/src/sys/sys/taskqueue.h` with `TASKQUEUE_DECLARE`, which expands to an extern pointer. A driver uses them by name:

```
TASKQUEUE_DECLARE(thread);
TASKQUEUE_DECLARE(swi);
TASKQUEUE_DECLARE(swi_giant);
TASKQUEUE_DECLARE(fast);
TASKQUEUE_DECLARE(bus);
```

taskqueue_thread is the generic thread-context queue. One kernel thread, priority P`WAIT`. The thread's name shows up in `ps` as `thread taskq`. Safe for any task that wants a full thread context and does not need special properties. The easiest predefined queue to use; a very reasonable first choice if you are not sure which queue you need.

taskqueue_swi is dispatched by a software interrupt handler, not a kernel thread. Tasks on this queue run with no driver mutex held but in SWI context, which still has restrictions (no sleeping). Useful for short non-sleeping work that wants to run promptly after an enqueue without the scheduling latency of waking a kernel thread. Driver use is uncommon.

taskqueue_swi_giant is the same as `taskqueue_swi` but runs with the historical Giant lock held. Essentially never used in new code. Mentioned only for completeness.

taskqueue_fast is a spin-mutex-backed software-interrupt queue, used for tasks that must be enqueueable from contexts where a sleep mutex would be wrong (for example, from inside another spin mutex). The taskqueue itself uses a spin mutex for its internal list, so enqueue is legal from any context. The task callback, however, runs in SWI context, which still has the no-sleeping restriction. Driver use is rare; filter-interrupt contexts that need to enqueue work typically use `taskqueue_fast` or, more commonly today, a private `taskqueue_create_fast` queue.

taskqueue_bus is a dedicated queue for `newbus(9)` device events (hot-plug insertion, removal, child-bus notifications). Ordinary drivers do not enqueue onto this queue.

For a driver like `myfirst`, the realistic choices are `taskqueue_thread` (the shared queue) or a private taskqueue you own and tear down at detach. Section 4 discusses the trade-off; Stage 1 of the refactor uses `taskqueue_thread` for simplicity and Stage 2 moves to a private queue.

14.3.6 Comparing Taskqueues and Callouts

A short side-by-side comparison, because a new reader asks this question first.

Property	<code>callout(9)</code>	<code>taskqueue(9)</code>
Fires at	A specific time	As soon as a worker thread picks it up
Callback context	Callout thread (default) or hardclock IRQ (<code>C_DIRECT_EXEC</code>)	Kernel thread (for <code>taskqueue_thread</code> , private queues) or SWI (for <code>taskqueue_swi</code> , <code>taskqueue_fast</code>)
May sleep	No	Yes, for thread-backed queues; no, for SWI-backed queues
May acquire sleepable locks	No	Yes, for thread-backed queues
May call <code>uiomove</code> , <code>copyin</code> , <code>copyout</code>	No	Yes, for thread-backed queues

Property	<code>callout(9)</code>	<code>taskqueue(9)</code>
Coalesces redundant submissions	No, each reset replaces the previous deadline	Yes, <code>ta_pending</code> increments
Cancellable before firing	<code>callout_stop(co)</code>	<code>taskqueue_cancel(tq, task, &pendp)</code>
Waits for in-flight callback	<code>callout_drain(co)</code>	<code>taskqueue_drain(tq, task)</code>
Periodic	Callback reschedules itself	No; enqueue again from somewhere else, or use a callout to enqueue
Scheduled for the future	<code>callout_reset(co, ticks, ...)</code>	<code>taskqueue_enqueue_timeout(tq, tt, ticks)</code>
Cost per firing	Microseconds	Microseconds plus thread wake (can be larger under load)

The table illustrates the division. A callout is the right primitive when you need to fire at a particular moment and the work is safe for callout context. A taskqueue is the right primitive when you need thread-context work and are willing to accept whatever scheduling latency the taskqueue introduces. Many drivers use both together: a callout fires at the deadline, the callout enqueues a task, the task does the real work in thread context.

14.3.7 Comparing Taskqueues and a Private Kernel Thread

Another comparison the chapter owes you, because a reader asking “why not just make a kernel thread” deserves a straight answer.

A kernel thread created with `kproc_create(9)` is a full-fledged scheduled entity: its own stack (typically 16 KB on amd64), its own priority, its own proc entry, its own state. A driver that wants to run a loop “every second, do X” could create such a thread and have it `kproc_kthread_add` plus `cv_timedwait` its way through the loop. The code works, but it costs more than the job usually deserves. A taskqueue with one thread that sits idle most of the time and wakes on enqueue is cheaper per pending work item and easier to tear down.

There are legitimate cases for `kproc_create`. A long-running subsystem with its own tuning (priority, CPU affinity, process group) is one. A periodic job that genuinely needs a thread of its own for observability is another. A driver’s deferred-work pattern almost never is. Use a taskqueue until a specific requirement forces you to do something else.

14.3.8 The Enqueue-Already-Pending Rule

One rule worth calling out early, because it is the single most common source of surprise for newcomers to the API: a task cannot be pending twice. If you call `taskqueue_enqueue(tq, &sc->t)` while `sc->t` is already pending, the kernel increments `sc->t.ta_pending` and returns success without linking the task a second time.

This has two implications. First, your callback will run once, not twice, even if you enqueued twice. Second, the pending argument the callback receives is the number of times the task was enqueued before the callback got dispatched; your callback may use that count to batch accumulated work.

If you want to run the callback N times for N enqueues, a single task is the wrong model. Use N separate tasks, or enqueue a sentinel into a driver-owned queue and process each sentinel in the callback. Almost always the coalescing behaviour is what you want; Section 5 walks through how to exploit it deliberately.

14.3.9 A Minimal Example, End to End

A hello-world task, for concreteness. If you type this in a scratch module and load it, you will see the `device_printf` line in `dmesg`:

```
#include <sys/param.h>
#include <sys/kernel.h>
#include <sys/module.h>
#include <sys/systm.h>
#include <sys/taskqueue.h>

static struct task example_task;

static void
example_task_fn(void *context, int pending)
{
    printf("example_task_fn: pending=%d\n", pending);
}

static int
example_modevent(module_t m, int event, void *arg)
{
    int error = 0;

    switch (event) {
    case MOD_LOAD:
        TASK_INIT(&example_task, 0, example_task_fn, NULL);
        taskqueue_enqueue(taskqueue_thread, &example_task);
        break;
    case MOD_UNLOAD:
        taskqueue_drain(taskqueue_thread, &example_task);
        break;
    default:
        error = EOPNOTSUPP;
        break;
    }
    return (error);
}
```



```
}

static moduledata_t example_mod = {
    "example_task", example_modevent, NULL
};
DECLARE_MODULE(example_task, example_mod, SI_SUB_DRIVERS, SI_ORDER_MIDDLE);
MODULE_VERSION(example_task, 1);
```

The module does five things, each a single line. At load, `TASK_INIT` prepares the task structure. `taskqueue_enqueue` asks the shared `taskqueue_thread` to run the callback. The callback prints a message. At unload, `taskqueue_drain` waits for the callback to finish if it has not already. The whole lifecycle is compact.

If you type this and load it, `dmesg` shows:

```
example_task_fn: pending=1
```

The `pending=1` reflects the fact that the task was enqueued once before the callback fired.

Now try a coalescing demonstration: change `MOD_LOAD` to enqueue the task five times in a row, then add a brief spin so the `taskqueue` thread gets a chance to wake up:

```
for (int i = 0; i < 5; i++)
    taskqueue_enqueue(taskqueue_thread, &example_task);
pause("example", hz / 10);
```

Run it again and `dmesg` shows:

```
example_task_fn: pending=5
```

One invocation, pending of five. That is the coalescing rule, live.

That is enough shape to make the worked refactors in the next sections meaningful. The rest of this chapter scales the same structure up to the real `myfirst` driver, replaces the scratch module with four integration stages, adds the teardown, adds a private `taskqueue`, and walks through the debugging story.

14.3.10 Wrapping Up Section 2

A `struct task` holds a callback and its context. A `struct taskqueue` manages a queue of such tasks and one or more threads (or an SWI context) that consume them. The API is small: create, start threads, enqueue (optionally with a delay), cancel, drain, free, block, unblock, quiesce. The kernel provides a handful of predefined queues that every driver can use without creating its own. The enqueue-already-pending rule folds redundant submissions into a coalesced invocation whose pending count is the final tally.

Section 3 takes those tools to the `myfirst` driver and deposits the first task in the code, under the `tick_source` callout that has been quietly skipping `selwakeup`. The fix is small; the mental model is the important part.

14.4 Section 3: Deferring Work from a Timer or Simulated Interrupt

Chapter 13 left the `myfirst` driver with three callouts that all obeyed the callout contract strictly. None of them tried to do anything that does not belong in a callout callback. The `tick_source` callback in particular omitted the `se wakeup` call that a real driver would want to make when new bytes appear in the buffer, and the file even carried a comment saying so. Chapter 14 removes that omission.

Section 3 is the first worked refactor. It introduces Stage 1 of the Chapter 14 driver: the driver gains one `struct task`, one task callback, an enqueue from `tick_source`, and a drain at detach. The private-taskqueue work is saved for Section 4; for Stage 1 we use the shared `taskqueue_thread`. Using the shared queue first keeps the first step small and isolates the change to the deferred-work pattern itself.

14.4.1 The Change in One Sentence

When `tick_source` has just deposited a byte into the circular buffer, instead of silently omitting `se wakeup`, enqueue a task whose callback runs `se wakeup` in thread context.

That is the whole change. Everything else is the surrounding setup.

14.4.2 The Softc Addition

Add two members to `struct myfirst_softc`:

```
struct task          selwake_task;
int                  selwake_pending_drops;
```

The `selwake_task` is the task we will enqueue. The `selwake_pending_drops` is a debug counter we will increment whenever the task coalesces two or more enqueues into one firing; the difference between “number of enqueue calls” and “number of callback invocations” tells us how often the tick source produced data faster than the taskqueue thread drained it. That is pure diagnostic; you can omit it if you prefer, but seeing a real coalescing count in action is valuable.

Add a read-only `sysctl` so we can observe the counter from userspace without a debug build:

```
SYSCALL_ADD_INT(&sc->sysctl_ctx, SYSCTL_CHILDREN(sc->sysctl_tree),
    OID_AUTO, "selwake_pending_drops", CTLFLAG_RD,
    &sc->selwake_pending_drops, 0,
    "Times selwake_task coalesced two or more enqueues into one firing");
```

Placement matters only in the sense that it must come after `sc->sysctl_tree` has been created and before the function returns success; the Chapter 13 attach sequence already has the right structure, so the addition slots in naturally alongside the other stats.

14.4.3 The Task Callback

Add a function:

```
static void
myfirst_selwake_task(void *arg, int pending)
{
    struct myfirst_softc *sc = arg;

    if (pending > 1) {
        MYFIRST_LOCK(sc);
        sc->selwake_pending_drops++;
        MYFIRST_UNLOCK(sc);
    }

    /*
     * No driver mutex held. Safe to call selwakeup(9) here.
     */
    selwakeup(&sc->rsel);
}
```

Several things to notice.

The callback takes the `_softc` via the `arg` pointer, exactly like the callout callbacks do. It does not need `MYFIRST_ASSERT` at the top because the task callback does not run with any driver lock held; the taskqueue framework does not hold your lock for you. This is different from the callout lock-aware pattern from Chapter 13, and is worth pausing on. A callout initialised with `callout_init_mtx(&co, &sc->mtx, 0)` runs with `sc->mtx` held. A task never does. Inside the task callback, if you want to touch state that the mutex protects, you acquire the mutex yourself, do the work, release it, and continue.

The callback conditionally updates `selwake_pending_drops` under the mutex. The condition `pending > 1` means “this callback is handling at least two coalesced enqueues”. Incrementing a counter under the mutex is fast and safe; doing it unconditionally would make the common case (`pending == 1`, no coalescing) pay the lock cost needlessly.

The `selwakeup(&sc->rsel)` call itself is the reason we are here. It runs without any driver lock held, which is what `selwakeup` wants, and it runs in thread context, which is what `selwakeup` requires. The bug from Chapter 13 is fixed.

The callback does not check `is_attached`. It does not need to. The detach path drains the task before freeing the `selinfo`; by the time `is_attached` would be zero, the task callback is guaranteed not to be running, and `selwakeup` will see valid state. The drain ordering is what makes the omission safe, which is why we discuss the ordering so carefully in Section 4.

14.4.4 The `tick_source` Edit

Change the `tick_source` callback from:

```
static void
myfirst_tick_source(void *arg)
{
    struct myfirst_softc *sc = arg;
    size_t put;
    int interval;

    MYFIRST_ASSERT(sc);
    if (!sc->is_attached)
        return;

    if (cbuf_free(&sc->cb) > 0) {
        put = cbuf_write(&sc->cb, &sc->tick_source_byte, 1);
        if (put > 0) {
            counter_u64_add(sc->bytes_written, put);
            cv_signal(&sc->data_cv);
            /* selwakeup omitted: cannot be called from a
             * callout callback while sc->mtx is held. */
        }
    }

    interval = sc->tick_source_interval_ms;
    if (interval > 0)
        callout_reset(&sc->tick_source_co,
            (interval * hz + 999) / 1000,
            myfirst_tick_source, sc);
}
```

To:

```
static void
myfirst_tick_source(void *arg)
{
    struct myfirst_softc *sc = arg;
    size_t put;
    int interval;
    bool wake_sel = false;

    MYFIRST_ASSERT(sc);
    if (!sc->is_attached)
        return;

    if (cbuf_free(&sc->cb) > 0) {
        put = cbuf_write(&sc->cb, &sc->tick_source_byte, 1);
```

```

        if (put > 0) {
            counter_u64_add(sc->bytes_written, put);
            cv_signal(&sc->data_cv);
            wake_sel = true;
        }
    }

    if (wake_sel)
        taskqueue_enqueue(taskqueue_thread, &sc->selwake_task);

    interval = sc->tick_source_interval_ms;
    if (interval > 0)
        callout_reset(&sc->tick_source_co,
            (interval * hz + 999) / 1000,
            myfirst_tick_source, sc);
}

```

Two edits. A local `wake_sel` flag records whether a byte was written; the `taskqueue_enqueue` call happens after the `cbuf` work. The comment about “selwakeupt omitted” becomes obsolete and is removed.

Why use a flag instead of calling `taskqueue_enqueue` inline in the `if (put > 0)` block? Because `taskqueue_enqueue` is safe to call while holding `sc->mtx` (it acquires its own internal mutex; there is no lock-order problem for the taskqueue’s own mutex because it is not ordered against `sc->mtx`), but it is good hygiene to keep mutex-held sections tight and to name the reason for the enqueue with a local variable. The version with the flag is easier to read and easier to extend if later stages add more conditions that should trigger the wake.

Is `taskqueue_enqueue` actually safe to call from a callout callback with `sc->mtx` held? Yes. The taskqueue uses its own internal mutex (`tq_mutex`) which is entirely separate from `sc->mtx`; no lock order between them is established, so `WITNESS` has nothing to complain about. We will verify this in the lab at the end of this section. For future reference, the relevant guarantee in `/usr/src/sys/kern/subr_taskqueue.c` is that `taskqueue_enqueue` acquires `TQ_LOCK(tq)` (a sleep mutex for `taskqueue_create`, a spin mutex for `taskqueue_create_fast`), performs the list manipulation, and releases the lock. No sleeping, no recursion into the caller’s lock, no cross-lock dependency.

14.4.5 The Attach Change

In `myfirst_attach`, add one line after the existing callout initialisations:

```
TASK_INIT(&sc->selwake_task, 0, myfirst_selwake_task, sc);
```

Place it alongside the callout init calls. The conceptual grouping (“here is where we prepare the driver’s deferred-work primitives”) makes the file easier to scan.

Initialise `selwake_pending_drops` to zero in the same block where other counters are zeroed:

```
sc->selwake_pending_drops = 0;
```

14.4.6 The Detach Change

This is the critical part of the stage. The Chapter 13 detach sequence, simplified, is:

1. Refuse detach if `active_fhs > 0`.
2. Clear `is_attached`.
3. Broadcast `data_cv` and `room_cv`.
4. Drain `rsl` and `wsl` via `seldrain`.
5. Drain the three callouts.
6. Destroy devices, free `sysctls`, destroy `cbuf`, free counters, destroy `cvs`, destroy `sx`, destroy `mtx`.

Chapter 14 Stage 1 adds one step: drain `selwake_task` between the callout drains (step 5) and the `seldrain` calls (step 4). Actually, the ordering subtlety is more careful than that. Let's think through it.

The `selwake_task` callback calls `selwakeup(&sc->rsl)`. If `sc->rsl` is being drained concurrently, the callback could race. The rule is: ensure the task callback is guaranteed not to be running before calling `seldrain`. That means `taskqueue_drain(taskqueue_thread, &sc->selwake_task)` must happen before `seldrain(&sc->rsl)`.

However, the task can still be enqueued by an in-flight callout callback until we have drained the callouts. If we drain the task first and then the callouts, an in-flight callout could re-enqueue the task after we drained it, and the re-enqueued task would then try to run after `seldrain`.

The only safe ordering is: drain callouts first (which guarantees no more enqueues will happen), then drain the task (which guarantees the last enqueue has completed), then call `seldrain`. But we also must clear `is_attached` before draining the callouts so an in-flight callback exits early instead of re-arming.

Putting it all together, the Stage 1 detach ordering is:

1. Refuse detach if `active_fhs > 0`.
2. Clear `is_attached` (under the mutex).
3. Broadcast `data_cv` and `room_cv` (release the mutex first).
4. Drain the three callouts (no mutex held; `callout_drain` may sleep).
5. Drain `selwake_task` (no mutex held; `taskqueue_drain` may sleep).
6. Drain `rsl` and `wsl` via `seldrain`.
7. Destroy devices, free `sysctls`, destroy `cbuf`, free counters, destroy `cvs`, destroy `sx`, destroy `mtx`.

Steps 4 and 5 are the new ordering constraint. Callouts first, tasks second, `sel` third. Violating that order on a debug kernel typically trips an assertion inside `seldrain`; on a production kernel it is a use-after-free waiting to happen.

The code in `myfirst_detach` becomes:

```
/* Chapter 13: drain every callout. No lock held; safe to sleep. */
MYFIRST_CO_DRAIN(&sc->heartbeat_co);
MYFIRST_CO_DRAIN(&sc->watchdog_co);
```

```
MYFIRST_CO_DRAIN(&sc->tick_source_co);

/* Chapter 14: drain every task. No lock held; safe to sleep. */
taskqueue_drain(taskqueue_thread, &sc->selwake_task);

seldrain(&sc->rsel);
seldrain(&sc->wsel);
```

Two lines of code plus a comment. The ordering is visible in the source.

14.4.7 The Makefile

No change. `bsd.kmod.mk` picks up the taskqueue API headers from the system tree; no additional source files are needed for Stage 1.

14.4.8 Building and Loading

At this point your working copy should have:

- The two new softc members (`selwake_task`, `selwake_pending_drops`).
- The `myfirst_selwake_task` function.
- The edit to `myfirst_tick_source`.
- The `TASK_INIT` call and the counter zeroing in `attach`.
- The `taskqueue_drain` call in `detach`.
- The new `selwake_pending_drops` `sysctl`.

Build from the Stage 1 directory:

```
# cd /path/to/examples/part-03/ch14-taskqueues-and-deferred-work/stage1-first-task
# make clean && make
```

Load:

```
# kldload ./myfirst.ko
```

Verify:

```
# kldstat | grep myfirst
 7    1 0xffffffff82f30000    ... myfirst.ko
# sysctl dev.myfirst.0
dev.myfirst.0.stats.selwake_pending_drops: 0
...
```

14.4.9 Observing the Fix

To observe Stage 1 doing its job, start a `poll(2)` waiter on the device and have the tick source generate data. A simple poller lives under `examples/part-03/ch14-taskqueues-and-deferred-work/labs/poll_waiter.c`:

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <poll.h>
#include <err.h>

int
main(int argc, char **argv)
{
    int fd, n;
    struct pollfd pfd;
    char c;

    fd = open("/dev/myfirst", O_RDONLY);
    if (fd < 0)
        err(1, "open");
    pfd.fd = fd;
    pfd.events = POLLIN;

    for (;;) {
        n = poll(&pfd, 1, -1);
        if (n < 0) {
            if (errno == EINTR)
                continue;
            err(1, "poll");
        }
        if (pfd.revents & POLLIN) {
            n = read(fd, &c, 1);
            if (n > 0)
                write(STDOUT_FILENO, &c, 1);
        }
    }
}
```

Compile it with `cc poll_waiter.c -o poll_waiter` (no special libraries). Run it in one terminal:

```
# ./poll_waiter
```

In a second terminal, enable the tick source at a slow pace to make the output easy to watch:


```
# sysctl dev.myfirst.0.tick_source_interval_ms=500
```

The Chapter 13 driver, without the Stage 1 fix, would leave `poll_waiter` stuck. The reader byte would accumulate in the buffer, but `poll(2)` would never return because `selwakeup` was never called. You would see nothing.

The Stage 1 driver does call `selwakeup`, via the task. You should see `t` characters appearing in the `poll_waiter` terminal every half second. When you stop the test, `poll_waiter` exits cleanly via `Ctrl-C`.

Now speed up the tick source to stress the taskqueue:

```
# sysctl dev.myfirst.0.tick_source_interval_ms=1
```

You should see a continuous stream of `t` characters. Check the coalescing counter:

```
# sysctl dev.myfirst.0.stats.selwake_pending_drops
dev.myfirst.0.stats.selwake_pending_drops: <some number, growing slowly>
```

The number is the count of times the task callback handled a pending count greater than one. On a lightly loaded machine it may stay small (the taskqueue thread wakes quickly enough to handle each enqueue individually). Under contention the number grows, and you can watch the coalescing behaviour directly.

If the counter stays at zero even under load, the machine is fast enough that every enqueue drains before the next one arrives. That is not a bug; it is a sign that the coalescing is present but not triggered. Section 5 introduces a deliberate workload that forces coalescing.

14.4.10 Unloading

Stop the tick source:

```
# sysctl dev.myfirst.0.tick_source_interval_ms=0
```

Close the `poll_waiter` with `Ctrl-C`. Unload:

```
# kldunload myfirst
```

The unload should be clean. If it fails with `EBUSY`, you still have an open descriptor somewhere; the close and the next `kldunload` should succeed.

If the unload hangs, something in the detach path is blocked. The most likely cause is that the `taskqueue_drain` is waiting for a task that cannot complete. That would indicate a bug, and the debugging section (Section 7) shows how to identify it. For the normal flow, the unload completes in milliseconds.

14.4.11 What We Just Did

A short summary before Section 4 scales up.

Stage 1 added one task to the driver, initialised it in `attach`, enqueued it from a callout callback, drained it in `detach` in the correct order, and observed coalescing in action. The task runs on the shared `taskqueue_thread`; it shares that queue with every other driver in the system that also uses it. For a low-rate workload that is entirely fine. For a driver that will eventually do substantial work in its tasks, or that wants to isolate its task-processing latency from whatever else the system is doing, a private taskqueue is the right answer. Section 4 takes that step.

14.4.12 Common Mistakes to Avoid

A short list of mistakes that beginners make when writing their first task. Each has bitten real drivers; each has a simple rule that prevents it.

Forgetting to drain at detach. If you enqueue tasks but do not drain them, an in-flight task may run after the softc is freed, and the kernel crashes in the task callback with a dereference of freed memory. Always drain every task your driver owns before freeing anything the task touches.

Draining in the wrong order relative to the state the task uses. The task-then-sel ordering we discussed above is a specific case. The general rule: drain every producer of enqueues, then drain the task, then free the state the task uses. Violating the order is a race, even if the race is rare.

Assuming the task runs immediately after enqueue. It does not. The taskqueue thread wakes on enqueue and then the scheduler decides when to run it. Under load this can be milliseconds. Drivers that assume zero latency break under load.

Assuming the task runs once per enqueue. It does not. Coalescing folds redundant submissions. If you need “exactly once per event” semantics, you need per-event state inside the softc (a queue of work items, for example), not one task per event.

Acquiring a driver lock in the wrong order in the task callback. The task callback is ordinary thread-context code. It obeys your driver’s established lock order. If the driver’s order is `sc->mtx -> sc->cfg_sx`, the task callback must take the mutex before the sx. Violations of this order are WITNESS errors the same as they would be anywhere else.

Using `taskqueue_enqueue` from inside a filter-interrupt context without a fast taskqueue. `taskqueue_enqueue(taskq, ...)` acquires a sleep mutex on the taskqueue’s internal lock. That is illegal from a filter-interrupt context. Filter interrupts must enqueue onto `taskqueue_fast` or a `taskqueue_create_fast` queue. Callout callbacks do not hit this restriction because they run in thread context; the issue is specific to filter interrupts. Part 4 revisits this when it introduces `bus_setup_intr`.

Each of those mistakes can be caught by a review, by WITNESS, or by a carefully written stress test. The first two in particular are the kind of bug that looks fine until the first detach under load.

14.4.13 Wrapping Up Section 3

The `myfirst` driver now has one task. It uses the task to move `selwakeup` out of the callout callback and into thread context, fixing a real bug from Chapter 13. The task is initialised in `attach`, enqueued from the

`tick_source` callback, and drained in detach in the correct order relative to the callouts and the `selinfo` drain.

The shared `taskqueue_thread` is the first taskqueue we used because it was already there. For a driver that is going to grow more tasks and more responsibility, a private taskqueue gives better isolation and a cleaner teardown story. Section 4 creates that private taskqueue.

14.5 Section 4: Taskqueue Setup and Cleanup

Stage 1 used the shared `taskqueue_thread`. That choice kept the first change small: one task, one enqueue, one drain, and a detach ordering to respect. Stage 2 creates a private taskqueue owned by the driver. The change is small in code terms, but it buys a handful of properties that become important once the driver grows.

This section teaches Stage 2 of the refactor, walks through the setup and teardown of a private taskqueue, audits the detach ordering carefully, and finishes with a pre-production checklist you can reuse on every taskqueue-using driver you write.

14.5.1 Why a Private Taskqueue

Three reasons for a private taskqueue.

First, **isolation**. A private taskqueue's thread runs only your driver's tasks. If some other driver in the system misbehaves on `taskqueue_thread` (blocking for too long in its task callback, for example), your driver's tasks are not affected. Conversely, if your driver misbehaves, the misbehaviour is contained.

Second, **observability**. `procstat -t` and `ps ax` show each taskqueue thread with a distinct name. A private queue is easy to spot: it shows up with the name you gave it (`myfirst taskq` by convention). The shared `taskqueue_thread` shows up just as `thread taskq`, which is shared with every other driver.

Third, **teardown is self-contained**. When you detach, you drain and free your own taskqueue. You do not have to reason about whether some other driver has a pending task that your drain might wait for. (You would not actually wait for another driver's task on the shared queue, but the mental model of "we own our teardown" is easier to reason about.)

The cost is small. A taskqueue and one kernel thread, created at attach and torn down at detach. A few pages of memory and a couple of scheduler entries. Nothing measurable on any realistic system.

For a driver that will eventually have multiple tasks, a private taskqueue is the right default. For a driver with a single trivial task on a rare code path, the shared queue is fine. `myfirst` is the former: we already have one task, and the chapter will add more.

14.5.2 The Softc Addition

Add one member to `struct myfirst_softc`:

```
struct taskqueue      *tq;
```

No other `softc` changes for Stage 2.

14.5.3 Creating the Taskqueue in Attach

In `myfirst_attach`, between the `mutex / cv / sx` initialisations and the callout initialisations, add:

```
sc->tq = taskqueue_create("myfirst taskq", M_WAITOK,
    taskqueue_thread_enqueue, &sc->tq);
if (sc->tq == NULL) {
    error = ENOMEM;
    goto fail_sx;
}
error = taskqueue_start_threads(&sc->tq, 1, PWAIT,
    "%s taskq", device_get_nameunit(dev));
if (error != 0)
    goto fail_tq;
```

The call reads as: create a taskqueue called “myfirst taskq”, allocate with `M_WAITOK` so the allocation cannot fail (we are in attach, which is a sleepable context), use `taskqueue_thread_enqueue` as the dispatcher so the queue is serviced by kernel threads, and pass `&sc->tq` as the context so the dispatcher can find the queue.

The name “myfirst taskq” is the human-readable label that shows up in `procstat -t`. The convention in the Chapter 14 examples is “<driver> taskq” for a driver with a single private queue; drivers with multiple queues should use more specific names like “myfirst rx taskq” and “myfirst tx taskq”.

`taskqueue_start_threads` creates the worker threads. The first argument is `&sc->tq`, a double pointer so the function can find the taskqueue. The second argument is the thread count; we use one thread for `myfirst`. A driver with heavy, parallelisable work might use more. The third argument is the priority; `PWAIT` is the ordinary choice and equivalent to what the predefined `taskqueue_thread` uses. The variadic name is a format string for each thread’s name; `device_get_nameunit(dev)` gives a per-instance name so multiple `myfirst` instances have distinguishable threads.

The failure paths deserve attention. If `taskqueue_create` returns `NULL` (it normally does not with `M_WAITOK`, but be defensive), we jump to `fail_sx`. If `taskqueue_start_threads` fails, we jump to `fail_tq`, which must call `taskqueue_free` before continuing with the other cleanup. The Chapter 14 Stage 2 source (see the examples tree) has the labels in the right order.

14.5.4 Updating the Enqueue Call Sites

Every `taskqueue_enqueue(taskqueue_thread, ...)` call becomes `taskqueue_enqueue(sc->tq, ...)`. Same for drains: `taskqueue_drain(taskqueue_thread, ...)` becomes `taskqueue_drain(sc->tq, ...)`.

After Stage 1, the driver has two such call sites: the enqueue in `myfirst_tick_source` and the drain in `myfirst_detach`. Both change in a single search-and-replace pass.

14.5.5 The Teardown Sequence

The detach ordering grows by two lines. The full sequence for Stage 2 is:

1. Refuse detach if `active_fhs > 0`.
2. Clear `is_attached` under the mutex, broadcast both cvs, release the mutex.
3. Drain the three callouts.
4. Drain `selwake_task` on the private taskqueue.
5. Drain `rsel` and `wsel` via `seldrain`.
6. Free the private taskqueue with `taskqueue_free`.
7. Destroy devices, free `sysctls`, destroy `cbuf`, free counters, destroy cvs, destroy sx, destroy mtx.

The new steps are 4 (which already existed in Stage 1, now targeting `sc->tq`) and 6 (which is new in Stage 2).

A natural question: do we need the explicit `taskqueue_drain` at step 4 if step 6 is going to drain everything anyway? Technically no. `taskqueue_free` drains all pending tasks before destroying the queue. But keeping the explicit drain has two benefits. First, it makes the ordering explicit: you see that the task drain happens before `seldrain`, which is the ordering we care about. Second, it separates the “wait for this specific task to finish” question from the “tear down the whole queue” question. If later stages add more tasks on the same queue, each gets its own explicit drain, and the code tells the reader what is happening.

The relevant code in `myfirst_detach`:

```
/* Chapter 13: drain every callout. No lock held; safe to sleep. */
MYFIRST_CO_DRAIN(&sc->heartbeat_co);
MYFIRST_CO_DRAIN(&sc->watchdog_co);
MYFIRST_CO_DRAIN(&sc->tick_source_co);

/* Chapter 14 Stage 1: drain every task. */
taskqueue_drain(sc->tq, &sc->selwake_task);

seldrain(&sc->rsel);
seldrain(&sc->wsel);

/* Chapter 14 Stage 2: destroy the private taskqueue. */
taskqueue_free(sc->tq);
sc->tq = NULL;
```

Setting `sc->tq` to `NULL` after freeing is defensive: a later bug that tries to use a pointer after free will dereference `NULL` and panic at the call site, rather than corrupt unrelated memory. It costs nothing and occasionally saves an afternoon of debugging.

14.5.6 The Attach Failure Path

Walk through the attach failure path carefully. The Chapter 13 attach had labels for the `cbuf` and mutex failure paths. Stage 2 adds taskqueue-related labels:

```

static int
myfirst_attach(device_t dev)
{
    struct myfirst_softc *sc;
    struct make_dev_args args;
    int error;

    sc = device_get_softc(dev);
    sc->dev = dev;
    sc->unit = device_get_unit(dev);

    mtx_init(&sc->mtx, device_get_nameunit(dev), "myfirst", MTX_DEF);
    cv_init(&sc->data_cv, "myfirst data");
    cv_init(&sc->room_cv, "myfirst room");
    sx_init(&sc->cfg_sx, "myfirst cfg");

    sc->tq = taskqueue_create("myfirst taskq", M_WAITOK,
        taskqueue_thread_enqueue, &sc->tq);
    if (sc->tq == NULL) {
        error = ENOMEM;
        goto fail_sx;
    }
    error = taskqueue_start_threads(&sc->tq, 1, PWAIT,
        "%s taskq", device_get_nameunit(dev));
    if (error != 0)
        goto fail_tq;

    MYFIRST_CO_INIT(sc, &sc->heartbeat_co);
    MYFIRST_CO_INIT(sc, &sc->watchdog_co);
    MYFIRST_CO_INIT(sc, &sc->tick_source_co);

    TASK_INIT(&sc->selwake_task, 0, myfirst_selwake_task, sc);

    /* ... rest of attach as in Chapter 13 ... */

    return (0);

fail_cb:
    cbuf_destroy(&sc->cb);
fail_tq:
    taskqueue_free(sc->tq);
fail_sx:
    cv_destroy(&sc->data_cv);
    cv_destroy(&sc->room_cv);

```

```
    sx_destroy(&sc->cfg_sx);
    mtx_destroy(&sc->mtx);
    sc->is_attached = 0;
    return (error);
}
```

The failure labels chain: `fail_cb` calls `cbuf_destroy` then falls through to `fail_tq` which calls `taskqueue_free` then falls through to `fail_sx` which destroys the cvs, the sx, and the mutex. Each label undoes everything up to the point the corresponding init call succeeded. If `taskqueue_start_threads` fails, we go straight to `fail_tq` (the taskqueue was allocated but has no threads; `taskqueue_free` still handles that correctly because a just-created-and-not-started taskqueue has zero threads to reap).

Note also: `TASK_INIT` does not have a failure mode (it is a macro that sets fields), and does not need a destroy counterpart. The task becomes idle once `taskqueue_drain` has been called on it, and the storage is simply reclaimed with the `softc`.

14.5.7 Thread Naming Conventions

`taskqueue_start_threads` takes a format string and a variable argument list, so each thread gets its own name. The naming convention has a real effect on debuggability, so a short paragraph about conventions is worth it.

The format string we use is `"%s taskq"` with `device_get_nameunit(dev)` as the argument. For a first myfirst instance the thread shows up as `myfirst0 taskq`. For a second instance it shows up as `myfirst1 taskq`. That makes the thread identifiable in `procstat -t` and `ps ax`.

A driver with multiple private queues should pick names that distinguish the queues:

```
taskqueue_start_threads(&sc->tx_tq, 1, PWAIT,
    "%s tx", device_get_nameunit(dev));
taskqueue_start_threads(&sc->rx_tq, 1, PWAIT,
    "%s rx", device_get_nameunit(dev));
```

Network drivers often name per-queue threads more specifically still (`"%s tx%d"` with the queue index) so that `procstat -t` shows every hardware queue's dedicated worker.

14.5.8 Choosing Thread Count

Most drivers create a single-threaded private taskqueue. One worker means tasks run sequentially, which simplifies the locking story: inside the task callback you can assume no other invocation of the same callback is running concurrently, without any explicit exclusion.

A driver with multiple hardware channels that need parallel processing might create multiple worker threads on the same taskqueue. The taskqueue guarantees that a single task is running on at most one thread at a time (that is what `tq_active` tracks), but different tasks on the same queue can run in parallel on different threads. For `myfirst` the single-threaded configuration is correct.

Multi-threaded taskqueues have implications for lock contention: two workers on the same queue, each running a different task, may contend for the same driver mutex. If the workload is naturally parallelisable, a multi-threaded queue speeds things up. If the workload is serialised anyway on the driver mutex, multiple threads add complexity without benefit. For a first taskqueue, single-threaded is the right default.

14.5.9 Choosing a Thread Priority

The `pri` argument to `taskqueue_start_threads` is the thread's scheduling priority. We use `PWAIT` in the Chapter 14 examples. The options in practice are:

- `PWAIT` (numerically 76): ordinary driver priority, equivalent to the priority of `taskqueue_thread`.
- `PI_NET` (numerically 4): network-adjacent priority, used by many ethernet drivers.
- `PI_DISK`: historical constant; `PRI_MIN_KERN` territory. Used by storage drivers.
- `PRI_MIN_KERN` (numerically 48): generic kernel-thread priority, used when the above constants do not fit.

For a driver whose task work is not latency-sensitive, `PWAIT` is fine. For a driver that must run its task callbacks promptly even under load, raising the priority closer to the interrupt threads is sometimes justified. `myfirst` uses `PWAIT`.

If you are writing a driver and are not sure what priority the convention expects, look at drivers of the same kind in `/usr/src/sys/dev/`. A storage driver that uses taskqueues probably uses `PRI_MIN_KERN` or `PI_DISK`; a network driver probably uses `PI_NET`. Pattern-matching on existing drivers is better than making up a priority.

14.5.10 A Worked Source Excerpt: `ale(4)`

A real driver using the exact pattern this section teaches. From `/usr/src/sys/dev/ale/if_ale.c`:

```
/* Create local taskq. */
sc->ale_tq = taskqueue_create_fast("ale_taskq", M_WAITOK,
    taskqueue_thread_enqueue, &sc->ale_tq);
taskqueue_start_threads(&sc->ale_tq, 1, PI_NET, "%s taskq",
    device_get_nameunit(sc->ale_dev));
```

The `ale` ethernet driver creates a fast taskqueue (`taskqueue_create_fast`) with a spin mutex, because it wants to be able to enqueue from its filter interrupt handler. It runs one thread at `PI_NET` priority, with the per-unit name convention. The shape is exactly what we use in `myfirst`, with the choice of fast vs. regular and the priority reflecting the driver's context.

From the same file, the matching teardown path:

```
taskqueue_drain(sc->ale_tq, &sc->ale_int_task);
/* ... */
taskqueue_free(sc->ale_tq);
```


taskqueue_drain on the specific task, then taskqueue_free on the queue. The same idiom we use.

Reading a `le(4)`'s setup and teardown once is worthwhile. It is a real driver, doing real work, using the pattern you are about to write in your own driver. Every driver under `/usr/src/sys/dev/` that uses taskqueues has very similar shape.

14.5.11 Regressing Chapter 13 Behaviour

Stage 2 must not break anything Chapter 13 established. Before proceeding, rerun the Chapter 13 stress kit with the Stage 2 driver loaded:

```
# cd /path/to/examples/part-03/ch13-timers-and-delayed-work/stage4-final
# ./test-all.sh
```

The test should pass exactly as it did at the end of Chapter 13. If it does not, the regression is in something Stage 2 changed; roll back to the pre-Stage-2 source and find the difference. The common cause is a missed enqueue-call-site update (an enqueue still targets `taskqueue_thread` instead of `sc->tq`). Those compile fine because the API is the same; they produce unrelated bugs at runtime.

14.5.12 Observing the Private Taskqueue

With the Stage 2 driver loaded, `procstat -t` shows the new thread:

```
# procstat -t | grep myfirst
<PID> <THREAD>      0 100 myfirst0 taskq      sleep  -      -      0:00
```

The name `myfirst0 taskq` is the per-instance thread name we asked for in `taskqueue_start_threads`. The state is `sleep` because the thread is blocked waiting for a task. The `wchan` is empty because the thread is sleeping on its own `cv`, which `procstat` may show differently across releases.

Enable the tick source and watch again:

```
# sysctl dev.myfirst.0.tick_source_interval_ms=100
# procstat -t | grep myfirst
<PID> <THREAD>      0 100 myfirst0 taskq      run    -      -      0:00
```

Briefly you may catch the thread in `run` state as it processes a task. Most of the time it sits in `sleep`. Both are expected.

`ps ax` shows the same thread:

```
# ps ax | grep 'myfirst.*taskq'
50  -  IL      0:00.00 [myfirst0 taskq]
```

The brackets indicate a kernel thread. The thread is always there while the driver is attached; it goes away at detach.

14.5.13 The Pre-Production Checklist for Stage 2

A short checklist to run through before declaring Stage 2 done. Each item is a question; each should be answerable with confidence.

- ☐ Does `attach` create the taskqueue before any code that might enqueue onto it?
- ☐ Does `attach` start at least one worker thread on the taskqueue before any code that expects a task to actually run?
- ☐ Does `attach` have failure labels that `taskqueue_free` the queue if a subsequent init fails?
- ☐ Does `detach` drain every task the driver owns before freeing any state those tasks touch?
- ☐ Does `detach` call `taskqueue_free` after every task drain and before destroying the mutex?
- ☐ Does `detach` set `sc->tq = NULL` after free, for defensive clarity?
- ☐ Is the taskqueue's thread priority chosen deliberately, with a rationale that matches the driver's kind?
- ☐ Is the taskqueue's thread name informative enough that `procstat -t` output is useful?
- ☐ Does every `taskqueue_enqueue` call target `sc->tq`, not `taskqueue_thread` (unless the enqueue is on a genuinely shared path for a specific reason)?
- ☐ Does every `taskqueue_drain` call match a `taskqueue_enqueue` on the same queue with the same task?

A driver that answers every item cleanly is a driver that handles its private taskqueue correctly. A driver that cannot is probably one step from a use-after-free at detach.

14.5.14 Common Mistakes in Stage 2

Three mistakes that beginners make when adding a private taskqueue. Each is preventable with a habit.

Creating the taskqueue after something that enqueues. If the enqueue happens before `taskqueue_create` has returned, the enqueue dereferences a NULL pointer. Always put `taskqueue_create` early in `attach`, before any code that can trigger an enqueue.

Forgetting `taskqueue_start_threads`. A taskqueue without worker threads is a queue that accepts enqueues but never runs the callbacks. Tasks pile up silently. If you think “my task never fires”, check that you called `taskqueue_start_threads`.

Calling `taskqueue_free` without first having cleared `is_attached`. If the taskqueue is freed while a callout callback is still running and could enqueue, the callout's enqueue will crash on the freed taskqueue. Always clear `is_attached`, drain callouts, drain the task, then free. The ordering is what makes it safe.

Section 7 will walk through each of these mistakes live, with a break-and-fix on a deliberately-misordered driver. For now the rule is: follow the ordering in this section, and the taskqueue lifecycle will be correct.

14.5.15 Wrapping Up Section 4

The driver now owns its own taskqueue. One thread, one name, one lifetime. Attach creates it and starts a worker; detach drains the task, frees the taskqueue, and unwinds. The ordering relative to callouts and `selinfo` is respected. `procstat -t` shows the thread by a recognisable name. The driver is self-contained in its deferred-work story.

Section 5 takes the next step: we exploit the coalescing behaviour deliberately for batching, introduce the `timeout_task` variant for scheduled tasks, and discuss how priority applies when a queue holds multiple task kinds.

14.6 Section 5: Prioritization and Coalescing of Work

Every task your driver owns enters a queue. The queue has a policy for deciding the order in which tasks run and what happens when the same task is enqueued twice. Section 5 makes that policy explicit, then shows how to use the coalescing contract deliberately to batch work, and finally introduces `timeout_task` as the taskqueue-side analogue of the callout.

This section is dense in ideas but short in code. The two driver changes for Stage 3 are a new task on the same taskqueue and a timeout task that drives a periodic batched write. The value is in the rules you internalise.

14.6.1 The Priority Ordering Rule

The `ta_priority` field in `struct task` orders tasks inside a single queue. Higher priority tasks run before lower priority tasks. A task with priority 5 that is enqueued after a task with priority 0 runs before the priority-0 task, even if the priority-0 task was enqueued first.

Priority is a small unsigned integer (`uint8_t`, so range 0-255). Most drivers use priority 0 for everything, in which case the queue is effectively FIFO. A driver with genuinely different urgencies for different tasks can assign different priorities and let the taskqueue reorder.

A quick example. Suppose a driver has two tasks: a `reset_task` that recovers from a hardware error, and a `stats_task` that rolls up accumulated statistics. If both are enqueued in a short window, the reset should run first. Giving `reset_task` priority 10 and `stats_task` priority 0 achieves that. The reset task runs first even if it was enqueued last.

Use priorities sparingly. A driver with ten different task kinds and ten different priorities is harder to reason about than a driver with ten task kinds that all run in enqueue order. Priorities exist for real differentiation, not for aesthetic ordering.

14.6.2 The Coalescing Rule, Restated

From Section 2, and worth saying again: if a task is enqueued while it is already pending, the kernel increments `ta_pending` and does not link the task a second time. The callback runs once, with the pending count in the second argument.

The precise code, from `/usr/src/sys/kern/subr_taskqueue.c`:

```
if (task->ta_pending) {
    if (__predict_false((flags & TASKQUEUE_FAIL_IF_PENDING) != 0)) {
        TQ_UNLOCK(queue);
        return (EEXIST);
    }
}
```

```

    if (task->ta_pending < USHRT_MAX)
        task->ta_pending++;
    TQ_UNLOCK(queue);
    return (0);
}

```

The counter saturates at `USHRT_MAX` (65535), which is a hard cap on how high the coalescing count can go. Past that, repeated enqueues are lost from the counter’s perspective, though they still return success. In practice nobody hits that cap, because a task that backs up 65535 times has deeper problems.

The coalescing rule has three consequences you design around.

First, **a task handles at most “one run per scheduler wakeup”**. If your work model needs “one callback per event”, a single task is wrong. You need per-event state.

Second, **the callback must be able to handle multiple events in one firing**. Writing the callback as if pending is always 1 is a bug that only shows up under load. Use the pending argument deliberately, or structure the callback so it processes whatever is in a driver-owned queue until the queue is empty.

Third, **you can exploit coalescing for batching**. If a producer enqueues a task once per event and the consumer drains a batch per firing, the system naturally converges on the rate the consumer can sustain. Under light load the coalescing never triggers (one event, one firing). Under heavy load the coalescing folds bursts into single firings with larger batches. The behaviour is self-tuning.

14.6.3 A Deliberate Batching Pattern: Stage 3

Stage 3 adds a second task to the driver: a `bulk_writer_task` that writes a fixed number of tick bytes to the buffer in a single firing, driven by a callout that enqueues the task periodically. The pattern is contrived (the real driver would just use a faster tick source), but it is the simplest demonstration of deliberate batching.

The softc addition:

```

struct task          bulk_writer_task;
int                  bulk_writer_batch;    /* bytes per firing */

```

The default `bulk_writer_batch` is zero (disabled). A `sysctl` exposes it for tuning.

The callback:

```

static void
myfirst_bulk_writer_task(void *arg, int pending)
{
    struct myfirst_softc *sc = arg;
    int batch, written;
    char buf[64];

    MYFIRST_LOCK(sc);

```

```

    batch = sc->bulk_writer_batch;
    MYFIRST_UNLOCK(sc);

    if (batch <= 0)
        return;

    batch = MIN(batch, (int)sizeof(buf));
    memset(buf, 'B', batch);

    MYFIRST_LOCK(sc);
    written = (int)cbuf_write(&sc->cb, buf, batch);
    if (written > 0) {
        counter_u64_add(sc->bytes_written, written);
        cv_signal(&sc->data_cv);
    }
    MYFIRST_UNLOCK(sc);

    if (written > 0)
        selwakeup(&sc->rsel);
}

```

A few remarks.

The callback acquires `sc->mtx`, reads the batch size, releases. Acquiring and releasing twice is fine; the work in between (`memset`) does not need the lock. The second acquisition wraps the actual `cbuf` operation and the counter update. The `selwakeup` happens with no lock held, as always.

The pending argument is unused in this simple callback. For a different batching design, pending would tell the callback how many times the task was enqueued and therefore how much work accumulated. Here the batching policy is “always write exactly `bulk_writer_batch` bytes per firing, no matter how many times enqueued”, so pending does not come into it.

The callback does not check `is_attached`. It does not need to. The detach drains the task before freeing anything the task touches, and `sc->mtx` protects `sc->cb` until the drain completes.

14.6.4 The Coalescing in Action

To demonstrate coalescing deliberately, Stage 3 adds a `sysctl bulk_writer_flood` whose writer attempts to enqueue `bulk_writer_task` a thousand times in a tight loop:

```

static int
myfirst_sysctl_bulk_writer_flood(SYSCTL_HANDLER_ARGS)
{
    struct myfirst_softc *sc = arg1;
    int flood = 0;

```

```

    int error, i;

    error = sysctl_handle_int(oidp, &flood, 0, req);
    if (error || req->newptr == NULL)
        return (error);
    if (flood < 1 || flood > 10000)
        return (EINVAL);

    for (i = 0; i < flood; i++)
        taskqueue_enqueue(sc->tq, &sc->bulk_writer_task);
    return (0);
}

```

Run it:

```

# sysctl dev.myfirst.0.bulk_writer_batch=32
# sysctl dev.myfirst.0.bulk_writer_flood=1000

```

Immediately after, observe the byte count. Without coalescing, a thousand enqueues at 32 bytes each would produce 32000 bytes. With coalescing, the actual number is one firing of 32 bytes, because the thousand enqueues collapsed into a single pending task. The driver's bytes_written counter should increase by 32, not 32000.

This is the coalescing contract working as designed. The producer asked for a thousand task runs; the taskqueue delivered one. The callback's single firing reflected all thousand requests but did the fixed amount of work the batching policy specified.

14.6.5 Using pending for Adaptive Batching

A more sophisticated pattern uses the pending argument to adapt batch size to queue depth. Suppose a driver wants to write pending bytes per firing: one byte per enqueue, folded into coalesced runs. The callback becomes:

```

static void
myfirst_adaptive_task(void *arg, int pending)
{
    struct myfirst_softc *sc = arg;
    char buf[64];
    int n;

    n = MIN(pending, (int)sizeof(buf));
    memset(buf, 'A', n);

    MYFIRST_LOCK(sc);
}

```

```
(void)cbuf_write(&sc->cb, buf, n);
counter_u64_add(sc->bytes_written, n);
cv_signal(&sc->data_cv);
MYFIRST_UNLOCK(sc);

selwakeup(&sc->rsel);
}
```

The callback writes pending bytes (up to the buffer size). At low load, pending is 1 and the callback writes one byte. At high load, pending is the queue depth at the moment the callback started, and the callback writes that many bytes in a single pass. The batching naturally scales with load.

This design is useful when each enqueue corresponds to a real event that wants one unit of work, and the batching is a performance optimisation rather than a semantic change. A network driver’s “transmit completion” handler is a classic example: each transmitted packet generates an interrupt that enqueues a task; the task’s job is to reclaim completed descriptors; at high packet rates, many interrupts fold into a single task firing that reclaims many descriptors at once.

We will not add the adaptive-batching task to `myfirst` in Stage 3, because the fixed-batch version already demonstrates coalescing. The adaptive pattern is worth keeping in mind for real driver work; the real FreeBSD drivers you will read use it often.

14.6.6 The Enqueue Flags

`taskqueue_enqueue_flags` extends `taskqueue_enqueue` with two flag bits:

- `TASKQUEUE_FAIL_IF_PENDING`: if the task is already pending, return `EEXIST` instead of coalescing.
- `TASKQUEUE_FAIL_IF_CANCELING`: if the task is currently being cancelled, return `EAGAIN` instead of waiting.

`TASKQUEUE_FAIL_IF_PENDING` is useful when you want to know whether the enqueue actually produced a new pending state, for accounting or debugging. A driver that counts “how many times was this task enqueued” can use the flag, get `EEXIST` on the redundant calls, and count only the non-redundant enqueues.

`TASKQUEUE_FAIL_IF_CANCELING` is useful during shutdown. If you are tearing the driver down and some code path would enqueue a task, you can pass the flag and check for `EAGAIN` to avoid re-adding a task that is in the middle of being cancelled. Most drivers do not need this in practice; the `is_attached` check usually handles the equivalent condition.

Neither flag is used in `myfirst`. Both exist, and a driver with specific needs can reach for them. For ordinary work, the plain `taskqueue_enqueue` is correct.

14.6.7 The `timeout_task` Variant

Sometimes you want a task to fire after a specific delay. The callout would be the natural primitive for that, but if the work the delayed callback wants to do requires thread context, you need the task’s context, not the callout’s. The kernel offers `struct timeout_task` for exactly this case.

timeout_task is defined in /usr/src/sys/sys/_task.h:

```
struct timeout_task {
    struct taskqueue *q;
    struct task t;
    struct callout c;
    int f;
};
```

The structure wraps a struct task, a struct callout, and an internal flag. When you schedule a timeout task with taskqueue_enqueue_timeout, the kernel starts the callout; when the callout fires, its callback enqueues the underlying task on the taskqueue. The task then runs in thread context, with all the usual guarantees.

The initialisation uses TIMEOUT_TASK_INIT:

```
TIMEOUT_TASK_INIT(queue, timeout_task, priority, func, context);
```

The macro expands to a function call _timeout_task_init that initialises both the task and the callout with appropriate linkage. You must pass the taskqueue at init time because the callout is set up to enqueue on that specific queue.

Scheduling uses taskqueue_enqueue_timeout(tq, &tt, ticks):

```
int taskqueue_enqueue_timeout(struct taskqueue *queue,
    struct timeout_task *timeout_task, int ticks);
```

The ticks argument is the same convention as callout_reset: hz ticks equals one second.

Draining uses taskqueue_drain_timeout(tq, &tt), which waits for the callout to expire (or cancels it if still pending) and then waits for the underlying task to complete. The drain is a single call, but it handles both the callout and the task phases.

Cancelling uses taskqueue_cancel_timeout(tq, &tt, &pendp):

```
int taskqueue_cancel_timeout(struct taskqueue *queue,
    struct timeout_task *timeout_task, u_int *pendp);
```

Returns zero if the timeout was cancelled cleanly, or EBUSY if the task is currently running. In the EBUSY case you typically follow up with taskqueue_drain_timeout.

14.6.8 A Stage 3 Timeout Task: Delayed Reset

Stage 3 adds a timeout task to the driver: a delayed reset that fires reset_delay_ms milliseconds after the reset sysctl is written. The existing reset sysctl runs synchronously; the delayed variant schedules the reset for later. Useful for testing and for situations where the reset should not happen until after the current IO drains.

The softc addition:


```
struct timeout_task    reset_delayed_task;
int                    reset_delay_ms;
```

The initialisation in attach:

```
TIMEOUT_TASK_INIT(sc->taskqueue, &sc->reset_delayed_task, 0,
    myfirst_reset_delayed_task, sc);
sc->reset_delay_ms = 0;
```

TIMEOUT_TASK_INIT takes the taskqueue as its first argument because the callout inside the timeout_task needs to know which queue to enqueue onto when it fires.

The callback:

```
static void
myfirst_reset_delayed_task(void *arg, int pending)
{
    struct myfirst_softc *sc = arg;

    MYFIRST_LOCK(sc);
    MYFIRST_CFG_XLOCK(sc);

    cbuf_reset(&sc->cb);
    sc->cfg.debug_level = 0;
    counter_u64_zero(sc->bytes_read);
    counter_u64_zero(sc->bytes_written);

    MYFIRST_CFG_XUNLOCK(sc);
    MYFIRST_UNLOCK(sc);

    cv_broadcast(&sc->room_cv);
    device_printf(sc->dev, "delayed reset fired (pending=%d)\n", pending);
}
```

Same logic as the synchronous reset from Chapter 13, but in task context. It can acquire the sleepable cfg_sx without the complications a callout would face. The pending count is logged for diagnostic purposes.

The sysctl handler that arms the delayed reset:

```
static int
myfirst_sysctl_reset_delayed(SYSCTL_HANDLER_ARGS)
{
    struct myfirst_softc *sc = arg1;
    int ms = 0;
```

```

    int error;

    error = sysctl_handle_int(oidp, &ms, 0, req);
    if (error || req->newptr == NULL)
        return (error);
    if (ms < 0)
        return (EINVAL);
    if (ms == 0) {
        (void)taskqueue_cancel_timeout(sc->tq,
            &sc->reset_delayed_task, NULL);
        return (0);
    }

    sc->reset_delay_ms = ms;
    taskqueue_enqueue_timeout(sc->tq, &sc->reset_delayed_task,
        (ms * hz + 999) / 1000);
    return (0);
}

```

A zero writes cancel the pending delayed reset. Any positive value schedules the task to fire after the given number of milliseconds. The tick conversion $(ms * hz + 999) / 1000$ is the same ceiling conversion we use for callouts.

The detach path drains the timeout task:

```
taskqueue_drain_timeout(sc->tq, &sc->reset_delayed_task);
```

Placement of the drain is the same as the plain task drain: after the callouts are drained, after `is_attached` is clear, before `sel drain` and before `taskqueue_free`.

14.6.9 Observing the Delayed Reset

With Stage 3 loaded, arm the delayed reset for three seconds in the future:

```
# sysctl dev.myfirst.0.reset_delayed=3000
```

Three seconds later `dmesg` shows:

```
myfirst0: delayed reset fired (pending=1)
```

The `pending=1` confirms that the timeout task fired once. Now arm it multiple times in rapid succession:

```
# sysctl dev.myfirst.0.reset_delayed=1000
# sysctl dev.myfirst.0.reset_delayed=1000
# sysctl dev.myfirst.0.reset_delayed=1000
```

One second later, only one reset fires. `dmesg` shows:

```
myfirst0: delayed reset fired (pending=1)
```

Why only one firing? Because `taskqueue_enqueue_timeout` behaves consistently with `callout_reset`: arming a pending timeout task replaces the previous deadline. The three successive arms produce one scheduled firing. The same behaviour would apply if we used `callout_reset` on a plain callout.

14.6.10 When to Use `timeout_task` vs. Callout Plus Task

A timeout task is the right primitive when you want a delayed action in thread context and the delay is the primary parameter. A plain callout with a task enqueue is the right primitive when you want a delayed action and the delay is an implementation detail: for example, when the delay is recomputed dynamically each time. Both work.

The two patterns have slightly different shapes in the source:

```
/* timeout_task pattern */
TIMEOUT_TASK_INIT(tq, &tt, 0, fn, ctx);
...
taskqueue_enqueue_timeout(tq, &tt, ticks);
...
taskqueue_drain_timeout(tq, &tt);
```

```
/* callout + task pattern */
callout_init_mtx(&co, &sc->mtx, 0);
TASK_INIT(&t, 0, fn, ctx);
...
callout_reset(&co, ticks, myfirst_co_fn, sc);
/* in the callout callback: taskqueue_enqueue(tq, &t); */
...
callout_drain(&co);
taskqueue_drain(tq, &t);
```

The `timeout_task` version is shorter because the kernel has bundled the pattern for you. The `callout+task` version is more flexible because the callout callback can decide dynamically whether to enqueue the task (for example, based on state conditions that do not exist at schedule time).

For `myfirst`'s delayed reset, the `timeout_task` is the right choice because the decision to fire is taken at schedule time (the `sysctl` writer requested it) and nothing in between changes that decision.

14.6.11 Priority Ordering Across Task Kinds

A driver with multiple tasks on the same taskqueue can use priorities to order them. For `myfirst` we do not need this; all tasks are equal-priority. But the pattern is worth understanding for when you do.

Suppose we had a `high_priority_reset_task` that must run ahead of any other pending tasks. We would initialise it with a priority greater than zero:

```
TASK_INIT(&sc->high_priority_reset_task, 10,  
         myfirst_high_priority_reset_task, sc);
```

And enqueue it normally:

```
taskqueue_enqueue(sc->tq, &sc->high_priority_reset_task);
```

If the queue has several tasks pending, including the new one and several priority-0 tasks, the new one runs first because of its higher priority. Priority is a property of the task (set at init), not of the enqueue (set at each call); if a task should sometimes be urgent and sometimes not, you need two task structures with two priorities, not one task you retune.

14.6.12 A Note on Fairness

A taskqueue with a single worker thread runs tasks strictly in priority order, with ties broken by enqueue order. A taskqueue with multiple worker threads can run several tasks in parallel; priority still orders the list, but the parallel workers may dispatch tasks out of strict order at the margin. For most drivers this does not matter.

If strict fairness or strict priority ordering is required, a single worker is the right choice. If throughput at the cost of occasional reordering is acceptable, multiple workers are fine. `myfirst` uses a single worker.

14.6.13 Wrapping Up Section 5

Stage 3 added a deliberate batching task and a timeout task. The batching task demonstrates coalescing by collapsing a thousand enqueues into a single firing; the timeout task demonstrates delayed execution in thread context. Both share the private taskqueue from Stage 2, both are drained at detach in the established order, and both obey the locking discipline the rest of the driver uses.

The priority and coalescing rules are now explicit. A task's priority orders it inside the queue; a task's `ta_pending` counter folds redundant enqueues into a single firing whose pending argument carries the tally.

Section 6 steps back from the `myfirst` refactor to survey the patterns that show up in real FreeBSD drivers. The mental models accumulate; the driver does not change again until Section 8.

14.7 Section 6: Real-World Patterns Using Taskqueues

Up to now Chapter 14 has developed a single driver through three stages. Real FreeBSD drivers use taskqueues in a handful of recurring shapes. This section catalogs the patterns, shows where each one appears in `/usr/src-`

c/sys/dev/, and explains when to reach for which. Recognising the patterns turns reading driver source from a puzzle into a vocabulary exercise.

Each pattern is presented as a short recipe: the problem, the taskqueue shape that solves it, a code sketch, and a real-driver reference you can read for the production version.

14.7.1 Pattern 1: Deferred Logging or Notification From an Edge Context

Problem. An edge-context callback (callout, interrupt filter, epoch section) detects a condition that should produce a log message or a notification to user-space. The logging call is too heavy for the edge context: `selwakeup`, `log(9)`, `kqueue_user_event`, or a multi-line `printf` that holds a lock the edge context cannot afford.

Solution. One `struct task` per condition, initialised in `attach` with a callback that performs the heavy call in thread context. The edge-context callback records the condition in softc state (a flag, a counter, a small piece of data), enqueues the task, and returns. The task runs in thread context, reads the condition from softc state, performs the call, clears the condition.

Code sketch.

```
struct my_softc {
    struct task log_task;
    int log_flags;
    struct mtx mtx;
    ...
};

#define MY_LOG_UNDERRUN 0x01
#define MY_LOG_OVERRUN 0x02

static void
my_log_task(void *arg, int pending)
{
    struct my_softc *sc = arg;
    int flags;

    mtx_lock(&sc->mtx);
    flags = sc->log_flags;
    sc->log_flags = 0;
    mtx_unlock(&sc->mtx);

    if (flags & MY_LOG_UNDERRUN)
        log(LOG_WARNING, "%s: buffer underrun\n",
            device_get_nameunit(sc->dev));
    if (flags & MY_LOG_OVERRUN)
        log(LOG_WARNING, "%s: buffer overrun\n",
```

```
        device_get_nameunit(sc->dev));
}

/* In an interrupt or callout callback: */
if (some_condition) {
    sc->log_flags |= MY_LOG_UNDERRUN;
    taskqueue_enqueue(sc->tq, &sc->log_task);
}
```

The flags field lets the edge context accumulate multiple distinct conditions before the task runs. When the task fires, it snapshots the flags, clears them, and emits one log line per condition. Coalescing folds repeated-same-condition enqueues into one callback invocation, which is what you want for log-spam prevention.

Real example. `/usr/src/sys/dev/ale/if_ale.c` uses an interrupt task (`sc->ale_int_task`) to handle deferred work from the filter interrupt, including conditions that want to log or notify.

14.7.2 Pattern 2: Delayed Reset or Reconfiguration

Problem. The driver detects a condition that calls for a hardware reset or a configuration change, but the reset should not happen immediately. Reasons for the delay include “give in-flight IO a chance to complete”, “batch multiple causes into one reset”, or “throttle resets to avoid a reset storm”.

Solution. A struct `timeout_task` (or a struct `callout` paired with a struct `task`). The detector enqueues the timeout task with the chosen delay. If the condition clears before the delay elapses, the detector cancels the timeout task. If the condition persists, the task fires in thread context and performs the reset.

Code sketch. Same shape as the `myfirst` Stage 3 delayed-reset task. The only variation is that the detector typically cancels the pending task whenever the “need to reset” state changes, so the reset happens only when the condition has persisted for the full delay.

Real example. Many storage and network drivers use this pattern for recovery. The `/usr/src/sys/dev/bge/if_bge.c` Broadcom driver uses timeout tasks for link-state re-evaluation after a physical-layer event.

14.7.3 Pattern 3: Post-Interrupt Processing (the Filter + Task Split)

Problem. A hardware interrupt arrives. The interrupt filter’s job is to decide “is this our interrupt and did the hardware really need attention”. The filter must run quickly and must not sleep. The actual processing (read registers, service completion queues, possibly copyout results to user-space) does not belong in the filter.

Solution. A two-level split. The filter handler runs synchronously, reads a status register, decides whether the interrupt belongs to us, and if so enqueues a task. The task runs in thread context and performs the real work. This is the standard filter-plus-ithread split that `bus_setup_intr(9)` supports natively, but the taskqueue variant is useful when the driver wants more control over the deferred context than `ithread` provides.

Code sketch.

```
static int
my_intr_filter(void *arg)
{
    struct my_softc *sc = arg;
    uint32_t status;

    status = CSR_READ_4(sc, STATUS_REG);
    if (status == 0)
        return (FILTER_STRAY);

    /* Mask further interrupts from the hardware. */
    CSR_WRITE_4(sc, INTR_MASK_REG, 0);

    taskqueue_enqueue(sc->tq, &sc->intr_task);
    return (FILTER_HANDLED);
}

static void
my_intr_task(void *arg, int pending)
{
    struct my_softc *sc = arg;

    mtx_lock(&sc->mtx);
    my_process_completions(sc);
    mtx_unlock(&sc->mtx);

    /* Unmask interrupts again. */
    CSR_WRITE_4(sc, INTR_MASK_REG, ALL_INTERRUPTS);
}
```

Several subtleties. The filter masks the interrupt at the hardware level before enqueueing the task, so the hardware does not keep firing while the task is pending. The task runs in thread context, processes completions, and re-enables interrupts at the end. Coalescing folds multiple interrupts into a single task firing; the masking prevents the hardware from firing unboundedly. Part 4 will walk through real interrupt setup; the pattern shown here is the shape the chapter prepares you for.

Real example. `/usr/src/sys/dev/ale/if_ale.c`, `/usr/src/sys/dev/age/if_age.c`, and most Ethernet drivers use this pattern or a close variant.

14.7.4 Pattern 4: Async copyin/copyout After Hardware Completion

Problem. The driver has a queued user-space request that supplied addresses for input or output data. Hardware completion arrives as an interrupt. The driver must copy the data between user-space and kernel buffers to finalise the request. `copyin` and `copyout` sleep on their slow path, so they cannot run in interrupt context.

Solution. The interrupt path records the request identifier and enqueues a task. The task runs in thread context, identifies the user-space addresses from the stored request state, performs the copyin or copyout, and wakes the waiting user thread.

Code sketch.

```
struct my_request {
    struct task finish_task;
    struct proc *proc;
    void *uaddr;
    void *kaddr;
    size_t len;
    int done;
    struct cv cv;
    /* ... */
};

static void
my_finish_task(void *arg, int pending)
{
    struct my_request *req = arg;

    (void)copyout(req->kaddr, req->uaddr, req->len);

    mtx_lock(&req->sc->mtx);
    req->done = 1;
    cv_broadcast(&req->cv);
    mtx_unlock(&req->sc->mtx);
}

/* In the interrupt task: */
taskqueue_enqueue(sc->tq, &req->finish_task);
```

The user-space thread waits on req->cv after submitting the request; it wakes when the task marks done and broadcasts.

Real example. Character-device drivers that implement ioctls with large data transfers sometimes use this pattern. USB bulk-transfer completion in `/usr/src/sys/dev/usb/` frequently defers the user-space data copy via tasks.

14.7.5 Pattern 5: Retry-With-Backoff After Transient Failure

Problem. A hardware operation failed, but the failure is known to be transient. The driver wants to retry after a backoff interval, with increasing backoff on repeated failures.

Solution. A struct `timeout_task` that is rearmed with increasing delay on each failure. The task callback performs the retry; on success the driver clears the backoff; on failure the task re-enqueues with a larger delay.

Code sketch.

```

struct my_softc {
    struct timeout_task retry_task;
    int retry_interval_ms;
    int retry_attempts;
    /* ... */
};

static void
my_retry_task(void *arg, int pending)
{
    struct my_softc *sc = arg;
    int err;

    err = my_attempt_operation(sc);
    if (err == 0) {
        sc->retry_attempts = 0;
        sc->retry_interval_ms = 10;
        return;
    }

    sc->retry_attempts++;
    if (sc->retry_attempts > MAX_RETRIES) {
        device_printf(sc->dev, "giving up after %d attempts\n",
            sc->retry_attempts);
        return;
    }

    sc->retry_interval_ms = MIN(sc->retry_interval_ms * 2, 5000);
    taskqueue_enqueue_timeout(sc->tq, &sc->retry_task,
        (sc->retry_interval_ms * hz + 999) / 1000);
}

```

The initial interval is 10 ms, doubling on each failure, capped at 5 seconds, with a maximum attempt count. The retry keeps firing until success or giveup. A separate code path can cancel the retry (with `taskqueue_cancel_timeout`) if the conditions that motivated it change.

Real example. `/usr/src/sys/dev/iwm/if_iwm.c` and other wireless drivers use timeout tasks for firmware-load retries and link recalibration.

14.7.6 Pattern 6: Deferred Teardown

Problem. An object inside the driver must be freed, but some other code path may still hold a reference. Freeing immediately would be a use-after-free. The driver needs to free later, after the references are known to be

gone.

Solution. A `struct task` whose callback frees the object. The code path that wants to free the object enqueues the task; the task runs in thread context, after any outstanding references have had a chance to complete.

In more elaborate forms, the pattern uses reference counting: the task decrements a reference count and frees the object only when the count reaches zero. In simpler forms, the taskqueue's FIFO ordering is enough: all earlier-enqueued tasks complete before the teardown task runs, so if references are always taken inside tasks, they are all gone by the time the teardown task fires.

Code sketch.

```
static void
my_free_task(void *arg, int pending)
{
    struct my_object *obj = arg;

    /* All earlier tasks on this queue have completed. */
    free(obj, M_DEVBUF);
}

/* When we want to free the object: */
static struct task free_task;
TASK_INIT(&free_task, 0, my_free_task, obj);
taskqueue_enqueue(sc->tq, &free_task);
```

Caveat: the `struct task` itself must live until the callback fires, which means either embedding it in the object (and freeing the containing structure) or allocating it separately.

Real example. `/usr/src/sys/dev/usb/usb_hub.c` uses deferred teardown when a USB device is removed while still being used by a driver further up the stack.

14.7.7 Pattern 7: Scheduled Statistics Rollover

Problem. The driver keeps accumulated statistics that must be rolled into a rate-per-interval at regular boundaries. The rollover involves snapshotting counters, computing the delta, and storing the result in a ring buffer. This can be done in a timer callback, but the computation touches data structures protected by a sleepable lock.

Solution. A periodic `timeout_task` that handles the rollover in thread context. The task re-enqueues itself at the end of each firing for the next interval.

This is, in effect, a “taskqueue-based callout”. It is slightly heavier than a plain callout because it rides a callout-plus-task combo, but it can do things a plain callout cannot. Useful only when the callout alone would not suffice.

Code sketch.

```
static void
my_stats_rollover_task(void *arg, int pending)
{
    struct my_softc *sc = arg;

    sx_xlock(&sc->stats_sx);
    my_rollover_stats(sc);
    sx_xunlock(&sc->stats_sx);

    taskqueue_enqueue_timeout(sc->tq, &sc->stats_task, hz);
}
```

The self-reenqueue at the end keeps the task firing once per second. A control path that wants to stop the rollover cancels the timeout task.

Real example. Several network drivers use this pattern for watchdog-adjacent timers whose work needs a sleepable lock.

14.7.8 Pattern 8: taskqueue_block During Delicate Configuration

Problem. The driver is performing a configuration change that must not be interrupted by task execution. A task firing in the middle of the configuration could observe inconsistent state.

Solution. `taskqueue_block(sc->tq)` before the configuration change; `taskqueue_unblock(sc->tq)` after. While blocked, new enqueues accumulate but no tasks are dispatched. The already-running task (if any) completes naturally before the block takes effect.

Code sketch.

```
taskqueue_block(sc->tq);
/* ... reconfigure ... */
taskqueue_unblock(sc->tq);
```

`taskqueue_block` is fast. It does not drain running tasks; it only prevents dispatch of new ones. For a guarantee that no task is currently running you combine it with `taskqueue_quiesce`:

```
taskqueue_block(sc->tq);
taskqueue_quiesce(sc->tq);
/* ... reconfigure ... */
taskqueue_unblock(sc->tq);
```

`taskqueue_quiesce` waits for the currently-running task to finish and for the pending queue to drain. Combined with `block`, you have a guarantee that no task is running and no task will start until you unblock.

Real example. Some Ethernet drivers use this pattern during interface state transitions (link up, link down, media change).

14.7.9 Pattern 9: `taskqueue_drain_all` at a Subsystem Boundary

Problem. A complex subsystem wants to be fully quiet at a specific point. All its pending tasks, including ones that may have been enqueued by other pending tasks, must complete before the subsystem proceeds.

Solution. `taskqueue_drain_all(tq)` drains every task on the queue, waits for every in-flight task to complete, and returns when the queue is quiet.

`taskqueue_drain_all` is not a substitute for per-task `taskqueue_drain` at detach (because the queue might have tasks from other paths that should not be drained), but it is useful for internal synchronisation points where you want “everything is done, full stop”.

Real example. `/usr/src/sys/dev/wg/if_wg.c` uses `taskqueue_drain_all` on its per-peer taskqueue during peer cleanup.

14.7.10 Pattern 10: Simulation-Grade Synthetic Event Generation

Problem. During testing, the driver wants to generate synthetic events that exercise the full event-processing path. A direct function call would bypass the scheduler, miss the race conditions, and not stress the taskqueue mechanism. A real hardware event is of course unavailable in a test rig.

Solution. A `sysctl` handler that enqueues a task. The task callback invokes the same driver routine a real event would have invoked. Because the task goes through the taskqueue, the synthetic event has the same execution shape as a real one: it runs in thread context, observes the same locking, and passes through the same coalescing.

This is exactly what `myfirst's bulk_writer_flood` `sysctl` does. The pattern transfers to any driver that wants to self-test its deferred-work paths without needing real hardware to generate the triggering event.

14.7.11 A Selection From Real Drivers

The patterns above are not invented for the chapter. A short tour of `/usr/src/sys/dev/` that you should explore for yourself, with a suggested order:

- `/usr/src/sys/dev/ale/if_ale.c`: A small, readable Ethernet driver that uses a private taskqueue, a filter-plus-task split, and a single interrupt task. Good first reading.
- `/usr/src/sys/dev/age/if_age.c`: Similar pattern, slightly different driver family. Reading both reinforces the pattern.
- `/usr/src/sys/dev/bge/if_bge.c`: A larger Ethernet driver with multiple tasks (interrupt task, link task, reset task). Shows how multiple tasks compose on one queue.
- `/usr/src/sys/dev/usb/usb_process.c`: USB's dedicated per-device process queue (`usb_proc_*`). Demonstrates how a subsystem wraps task-style deferred work for its own domain.
- `/usr/src/sys/dev/wg/if_wg.c`: WireGuard uses group taskqueues for per-peer encryption. Advanced reading, but useful once the basic patterns click.
- `/usr/src/sys/dev/iwm/if_iwm.c`: Wireless driver with multiple timeout tasks for calibration, scanning, and firmware management.
- `/usr/src/sys/kern/subr_taskqueue.c`: The implementation itself. Reading `taskqueue_run_locked` once makes everything else concrete.

Twenty minutes with any of those files is an hour of chapter explanations you can skip. The patterns are visible at a glance once you know what to look for.

14.7.12 Wrapping Up Section 6

The same small API composes into a large family of patterns. Deferred logging, filter-plus-task interrupts, async copyin/copyout, retry-with-backoff, deferred teardown, statistics rollover, block during reconfiguration, `drain_all` at subsystem boundaries, synthetic event generation: each pattern is a variation on “edge detects, task acts”, and each is productive whenever the driver you are writing or reading fits the shape.

Section 7 turns to the other side of the same coin: when the pattern goes wrong, how do you see it? What tools does FreeBSD provide for inspecting a taskqueue’s state, and what are the common bugs that those tools help you diagnose?

14.8 Section 7: Debugging Taskqueues

Most taskqueue code is short. The common bugs are not subtle: tasks that never fire, tasks that fire too often, tasks that fire after the softc is freed, deadlocks against the driver mutex, and draining at the wrong point in the detach sequence. This section names the bugs, shows how to observe them, and walks through a deliberate break-and-fix on `myfirst` so you can practise the debugging workflow with something in front of you.

14.8.1 The Tools

A short survey of the tools you will reach for.

procstat -t: lists every kernel thread with its name, priority, state, and wait-channel. A private taskqueue’s worker thread shows up as `<name> taskq`, where `<name>` is what you passed to `taskqueue_start_threads`. A thread stuck in a non-trivial wait channel is a clue: the name of the channel often tells you what the thread is waiting for.

ps ax: equivalent for most of what `procstat -t` shows, with a less taskqueue-specific output. The kernel thread name appears in brackets.

sysctl dev.<driver>: the driver’s own sysctl tree. If you added a counter like `selwake_pending_drops`, its value is visible here. Diagnostic sysctls are the cheapest form of observability; add them whenever the question “how often does this path fire” might matter later.

dtrace(1): the kernel tracing framework. Taskqueue activity is traceable via FBT (function boundary tracing) probes on `taskqueue_enqueue` and `taskqueue_run_locked`. A short D script can count enqueues, measure delays between enqueue and dispatch, and so on.

ktr(4): the kernel event tracer. Compile-time enabled in debug kernels, provides a ring buffer of kernel events that can be dumped after a crash or inspected live. Useful for post-mortem analysis.

ddb(4): the in-kernel debugger. Breakpoints, stack traces, memory inspection. Reachable via `kgdb` after a kernel panic, or interactively after a `sysctl debug.kdb.enter=1` if you built a kernel with KDB enabled.

INVARIANTS and WITNESS: compile-time assertions and lock-order checker. Not tools you invoke, but the first line of defence. A debug kernel catches most taskqueue bugs the first time you hit them.

The Chapter 14 labs exercise `procstat -t`, `sysctl`, and `dtrace` explicitly. `ktr` and `ddb` are mentioned for completeness.

14.8.2 Common Bug 1: The Task Never Runs

Symptoms. You enqueue a task from a `sysctl` handler or callout callback; the task callback's `device_printf` never appears; the driver otherwise seems to work.

Likely cause. `taskqueue_start_threads` was not called, or the taskqueue pointer you enqueued onto is `NULL`.

How to check.

```
# procstat -t | grep myfirst
```

If no `myfirst taskq` thread is listed, the taskqueue either does not exist or has no threads. Check the attach path: is `taskqueue_create` called? Is its return value stored? Is `taskqueue_start_threads` called after?

```
# dtrace -n 'fbt::taskqueue_enqueue:entry /arg0 != 0/ { @[stack()] = count(); }'
```

If the stack trace shows enqueues onto the driver's taskqueue, the task is being submitted. If nothing shows, the code path that should enqueue is not being reached. Trace back and find why.

14.8.3 Common Bug 2: The Task Runs Too Often

Symptoms. The task callback does more work than expected, or the driver logs a strange count.

Likely cause. The callback does not respect the pending argument, or the callback self-enqueues without a condition, so once started it loops forever.

How to check. Add a counter to the callback:

```
static void
myfirst_task(void *arg, int pending)
{
    struct myfirst_softc *sc = arg;
    static int invocations;

    atomic_add_int(&invocations, 1);
    if ((invocations % 1000) == 0)
        device_printf(sc->dev, "task invocations=%d\n", invocations);
    /* ... */
}
```

If the counter grows faster than the expected trigger rate, the task is self-looping or coalescing is not happening. Inspect the enqueue call sites.

14.8.4 Common Bug 3: Use-After-Free In the Task Callback

Symptoms. Kernel panic with a stack trace ending in your task callback, at a location that accesses softc state. The panic may happen during or shortly after detach.

Likely cause. The detach path freed the softc (or something the task touches) before draining the task. A trailing enqueue from a callout or other edge context fired after the drain, and the task ran against freed state.

How to check. Review the detach path against the ordering from Section 4. Specifically:

1. `is_attached` must be cleared before callouts are drained, so callout callbacks exit without re-enqueueing.
2. Callouts must be drained before tasks are drained, so no more enqueues can happen after the task drain.
3. Tasks must be drained before the state they touch is freed.
4. `taskqueue_free` must be called after all tasks on the queue are drained.

A mismatch in any of those is a potential use-after-free.

The debug kernel catches many of these cases via INVARIANTS assertions in `cbuf_*`, `cv_*`, and `mtx_*` routines. Run the detach path under load with WITNESS enabled; a bug often surfaces immediately.

14.8.5 Common Bug 4: Deadlock Between Task and Driver Mutex

Symptoms. A `read(2)` or `write(2)` to the device hangs forever. The task callback sits in a lock wait. The driver mutex is held by a different thread.

Likely cause. The task callback tries to acquire a lock that the thread enqueueing the task already holds, creating a cycle. For example:

- Thread A holds `sc->mtx` and calls a function that enqueues a task.
- The task callback acquires `sc->mtx` before doing its work.
- The task cannot proceed because Thread A still holds the mutex.
- Thread A waits for the task to complete.

The “Thread A waits for the task to complete” part does not fit the `myfirst` architecture (the driver does not explicitly wait for tasks from inside mutex-held paths), but it is a common shape in other drivers. Avoid it by not draining tasks while holding a lock they need.

How to check.

```
# procstat -kk <pid of stuck read/write thread>
# procstat -kk <pid of taskqueue thread>
```

Compare the stack traces. If one shows `mtx_lock/sx_xlock` in the task callback and the other shows `msleep_sbt/sleepqueue` at a point that holds the same lock, you have a deadlock.

14.8.6 Common Bug 5: Drain Hangs Forever

Symptoms. Detach hangs, `kldunload` does not return. The taskqueue thread is stuck somewhere.

Likely cause. The task callback is waiting on a condition that cannot be satisfied because the drain path is blocking the producer. Or the task callback is waiting on a lock that the detach path holds.

How to check.

```
# procstat -kk <pid of kldunload>
# procstat -kk <pid of taskqueue thread>
```

The drain is in `taskqueue_drain`, which is in `msleep`. The task is in some wait. Identify the wait channel; the name often tells you what the task is blocked on. If the task is blocked on something the detach path is holding, the design has a cycle.

A common specific case: the task callback calls `seldrain`, the detach path also calls `seldrain`, and the two collide. Avoid by ensuring `seldrain` is called exactly once, in the detach path, after the task drain.

14.8.7 The Break-and-Fix Exercise

A deliberate bug-and-fix walkthrough. The Stage 1 driver is correct; we modify it to introduce each of the bugs above, observe the symptom, and fix it.

14.8.7.1 Broken variant 1: missing `taskqueue_start_threads`

Remove the `taskqueue_start_threads` call from `attach`. Rebuild, load, enable the tick source, and run the `poll_waiter`. You will observe: no data appears in `poll_waiter`, even though `sysctl dev.myfirst.0.tick_source_interval_ms` is set.

Check `procstat -t`:

```
# procstat -t | grep myfirst
```

No `myfirst taskq` thread appears. The taskqueue exists (you created it) but has no workers. The enqueued `selwake_task` sits on the queue forever.

Fix: put the `taskqueue_start_threads` call back. Rebuild. Confirm the thread appears in `procstat -t` and the `poll_waiter` sees data.

14.8.7.2 Broken variant 2: draining in the wrong order

Move the `taskqueue_drain` call in `detach` to happen before the callout drains:

```
/* WRONG ORDER: */
taskqueue_drain(sc->tq, &sc->selwake_task);
MYFIRST_CO_DRAIN(&sc->heartbeat_co);
MYFIRST_CO_DRAIN(&sc->watchdog_co);
MYFIRST_CO_DRAIN(&sc->tick_source_co);
seldrain(&sc->rsl);
seldrain(&sc->wsl);
```


Rebuild, load, enable the tick source at a high rate, let data flow for a few seconds, then unload. Most times the unload works. Occasionally, the unload panics with a stack in `selwakeup` being called after `sel drain`. The race is rare but real.

The issue: `taskqueue_drain` returned, but then an in-flight `tick_source` callout fired (it had not been drained yet) and re-enqueued the task. The new task fired after `sel drain` had run, and tried to `selwakeup` on a drained `selinfo`.

Fix: restore the correct order (callouts first, then tasks, then `sel drain`). Rebuild, verify the race is gone under the same stress.

14.8.7.3 Broken variant 3: task callback holds the mutex too long

Change `myfirst_selwake_task` to hold the mutex across `selwakeup`:

```
static void
myfirst_selwake_task(void *arg, int pending)
{
    struct myfirst_softc *sc = arg;

    MYFIRST_LOCK(sc);          /* WRONG: holds mutex across selwakeup */
    selwakeup(&sc->rsl);
    MYFIRST_UNLOCK(sc);
}
```

Rebuild. Load under a debug kernel. Enable the tick source. Within seconds the kernel panics with a WITNESS complaint about locking order (or in some configurations with an assertion failure in `selwakeup` itself).

The issue: `selwakeup` acquires a lock that is not in the driver's documented lock order. WITNESS notices and complains.

Fix: the correct `myfirst_selwake_task` calls `selwakeup` with no driver mutex held. Restore that, rebuild, verify no WITNESS warnings.

14.8.7.4 Broken variant 4: forget to drain the task in detach

Remove the `taskqueue_drain(sc->tq, &sc->selwake_task)` line from `detach`. Rebuild. Load, enable the tick source at a high rate, run the `poll_waiter`, and immediately unload the driver.

Most times the unload completes. Occasionally, a task that was in flight at unload time runs against a `softc` whose `selinfo` has been drained and freed. The symptom is usually a kernel panic or a memory corruption that shows up later as an unrelated crash.

Fix: restore the drain. Rebuild, verify that repeated load-and-unload under load is stable.

14.8.7.5 Broken variant 5: wrong taskqueue pointer

A subtle Stage-2 bug. After moving to the private taskqueue, forget to update the `taskqueue_drain` call in `detach`. It still targets `taskqueue_thread`:

```
/* WRONG: enqueue on sc->tq but drain on taskqueue_thread */
taskqueue_enqueue(sc->tq, &sc->selwake_task);
/* ... in detach ... */
taskqueue_drain(taskqueue_thread, &sc->selwake_task);
```

Rebuild. Load, enable the tick source, run the waiter, unload. The unload typically completes without error, but `taskqueue_drain(taskqueue_thread, ...)` does not actually wait for the task that is running on `sc->tq`. If the task is in flight when detach proceeds, use-after-free.

Fix: match enqueue and drain on the same taskqueue pointer. Rebuild, test.

14.8.8 A DTrace One-Liner

A useful one-liner for any taskqueue-using driver. It measures the time between enqueue and dispatch for every task on the system:

```
# dtrace -n '
  fbt::taskqueue_enqueue:entry { self->t = timestamp; }
  fbt::taskqueue_run_locked:entry /self->t/ {
    @[execname] = quantize(timestamp - self->t);
    self->t = 0;
  }
'
```

The output is a distribution of enqueue-to-dispatch latency per process. Run it while your driver is producing tasks, then hit Ctrl-C to see the quantised histogram. Typical results on a lightly-loaded machine: tens of microseconds. Under load: milliseconds. If you see seconds, something is wrong.

A second useful one-liner measures task callback duration:

```
# dtrace -n '
  fbt::taskqueue_run_locked:entry { self->t = timestamp; }
  fbt::taskqueue_run_locked:return /self->t/ {
    @[execname] = quantize(timestamp - self->t);
    self->t = 0;
  }
'
```

Same structure, different timing. Tells you how long each `taskqueue_run_locked` invocation takes (which is the callback duration plus a small constant overhead).

14.8.9 Diagnostic Sysctls to Add

Useful counters to add to any taskqueue-using driver, with minimal cost and high diagnostic value.

```

int enqueues;           /* Total enqueues attempted. */
int pending_drops;      /* Enqueues that coalesced. */
int callback_runs;      /* Total callback invocations. */
int largest_pending;    /* Peak pending count observed. */

```

Update the counters in the enqueue path and the callback:

```

static void
myfirst_task(void *arg, int pending)
{
    struct myfirst_softc *sc = arg;

    sc->callback_runs++;
    if (pending > sc->largest_pending)
        sc->largest_pending = pending;
    if (pending > 1)
        sc->pending_drops += pending - 1;
    /* ... */
}

/* Enqueue site: */
sc->enqueues++;
taskqueue_enqueue(sc->tq, &sc->task);

```

Expose each as a read-only sysctl. Under normal load, `enqueues == callback_runs + pending_drops`. `largest_pending` tells you the worst coalescing moment; if it grows, the taskqueue is falling behind the producer.

These counters cost a few atomic adds per enqueue. On any realistic workload the cost is unmeasurable. The diagnostic value is substantial.

14.8.10 The Debug-Kernel Obligation

A reminder worth repeating: run every Chapter 14 change under a kernel with `INVARIANTS`, `WITNESS`, `WITNESS_SKIPSPIN`, `DDB`, `KDB`, and `KDB_UNATTENDED`. Most taskqueue bugs that are hard to find on a production kernel are caught instantly on a debug kernel. The cost of running a debug kernel is a small performance hit and a slightly larger build; the cost of not running one is an afternoon of debugging whenever anything goes wrong.

14.8.11 Wrapping Up Section 7

Debugging taskqueues is a small skill that composes with the tools you already have for debugging callouts, mutexes, and cvs. `procstat -t` and `ps ax` show the thread. `sysctl` exposes diagnostic counters. `dtrace`

measures enqueue-to-dispatch latency and callback duration. WITNESS catches locking-order violations at run-time. The common bugs (task never runs, wrong drain order, wrong lock discipline in callback, forgotten drain) are each catchable with a checklist and a debug kernel.

Section 8 consolidates the Chapter 14 work into Stage 4, the final driver. We extend `LOCKING.md`, bump the version string, and audit the driver against a full stress pass.

14.9 Section 8: Refactoring and Versioning Your Taskqueue Driver

Stage 4 is the consolidation stage. It does not add new functionality beyond what Stage 3 established; it sharpens the organisation of the code, updates the documentation, bumps the version, and runs the full regression sweep. If Stages 1 through 3 are where you built the driver, Stage 4 is where you ship it.

This section walks through the consolidation. The driver source is unified into a single, well-structured file; `LOCKING.md` gains a Tasks section; the version string advances to `0.8-taskqueues`; and the final regression pass confirms that every Chapter 12 and Chapter 13 behaviour still works correctly alongside the new Chapter 14 additions.

14.9.1 File Organisation

The chapter does not split the driver into multiple `.c` files. `myfirst.c` stays as the single translation unit, with one added responsibility (the tasks) grouped next to the corresponding callouts. If the driver grew much larger, a natural split would be `myfirst_timers.c` for callout code and `myfirst_tasks.c` for task code, with shared declarations in `myfirst.h`. For the current size the single file is easier to read.

Inside `myfirst.c`, the Stage 4 organisation is:

1. Includes and global macros.
2. Softc structure.
3. File-handle structure.
4. `cdevsw` declaration.
5. Buffer helpers.
6. Condition-variable wait helpers.
7. Sysctl handlers, grouped:
 - Configuration sysctls (debug level, soft byte limit, nickname).
 - Timer interval sysctls (heartbeat, watchdog, tick source).
 - Task sysctls (reset delayed, bulk writer batch, bulk writer flood).
 - Read-only stats sysctls.
8. Callout callbacks.
9. Task callbacks.
10. Cdev handlers (open, close, read, write, poll, handle destructor).
11. Device methods (identify, probe, attach, detach).
12. Module glue (driver, `DRIVER_MODULE`, version).

A block comment at the top of the file lists the major sections, so a new reader can jump to the right area without grep. Inside each section, the order is established-first: heartbeat before watchdog before tick source, for the callouts that share that order in the outline.

14.9.2 The LOCKING.md Update

The Chapter 13 LOCKING.md had sections for the mutex, the cvs, the sx, and the callouts. Chapter 14 adds a Tasks section.

Tasks

The driver owns one private taskqueue (``sc->tq``) and three tasks:

- ``selwake_task`` (plain): calls ``selwakeup(&sc->rsl)``. Enqueued from ``myfirst_tick_source`` when a byte is written. Drained at detach after callouts are drained and before ``seldrain``.
- ``bulk_writer_task`` (plain): writes a configured number of bytes to the cbuf, signals ``data_cv``, calls ``selwakeup(&sc->rsl)``. Enqueued from sysctl handlers and from the tick_source callback when ``bulk_writer_batch`` is non-zero. Drained at detach after callouts.
- ``reset_delayed_task`` (timeout_task): performs a delayed reset of the cbuf, counters, and configuration. Enqueued by the ``reset_delayed`` sysctl. Drained at detach.

The taskqueue is created in ``myfirst_attach`` with ``taskqueue_create`` and one worker thread started at ``PWAIT`` priority via ``taskqueue_start_threads``. It is freed in ``myfirst_detach`` via ``taskqueue_free`` after every task has been drained.

All task callbacks run in thread context. Each callback acquires ``sc->mtx`` explicitly if it needs state protected by the mutex; the taskqueue framework does not acquire driver locks automatically.

All task callbacks call ``selwakeup(9)`` (when they call it at all) with no driver lock held. The rule is the same as for the ``myfirst_read`` / ``myfirst_write`` paths: drop the mutex before ``selwakeup``.

Detach Ordering

The detach sequence is:

1. Refuse detach if ``sc->active_fhs > 0`` (EBUSY).
2. Clear ``sc->is_attached`` under ``sc->mtx``.
3. Broadcast ``data_cv`` and ``room_cv``.
4. Release ``sc->mtx``.

5. Drain ``heartbeat_co`, `watchdog_co`, `tick_source_co``.
6. Drain ``selwake_task`, `bulk_writer_task`, `reset_delayed_task`` (the last via ``taskqueue_drain_timeout``).
7. ``seldrain(&sc->rsel)`, `seldrain(&sc->wsel)``.
8. ``taskqueue_free(sc->tq)``.
9. Destroy cdev and cdev alias.
10. Free sysctl context.
11. Destroy cbuf, free counters.
12. Destroy ``data_cv`, `room_cv`, `cfg_sx`, `mtx``.

Violating the order risks use-after-free in task callbacks, selinfo accesses after drain, or taskqueue teardown while a task is still running.

The update is explicit about the ordering because the ordering is the main thing that can go wrong. A reader who inherits the driver from you and wants to add a new task has the existing discipline spelled out.

14.9.3 The Version Bump

The version string in the source moves from 0.7-timers to 0.8-taskqueues:

```
#define MYFIRST_VERSION "0.8-taskqueues"
```

And the driver's probe string updates:

```
device_set_desc(dev, "My First FreeBSD Driver (Chapter 14 Stage 4)");
```

The version is visible via the `hw.myfirst.version` sysctl, which was established in Chapter 12.

14.9.4 The Final Regression Pass

Stage 4 must pass every test that Stages 1 through 3 passed, plus the Chapter 12 and Chapter 13 test suites. A compact pass order:

1. **Build cleanly** under the debug kernel (`make clean && make`).
2. **Load** with `kldload ./myfirst.ko`.
3. **Chapter 11 unit tests**: basic read, write, open, close, reset.
4. **Chapter 12 synchronisation tests**: bounded blocking reads, bounded blocking writes, timeout reads, sx-protected configuration, cv broadcasts at detach.
5. **Chapter 13 timer tests**: heartbeat fires at configured rate, watchdog detects stalled drainage, tick source injects bytes.
6. **Chapter 14 task tests**:
 - `poll_waiter` sees data when tick source is active.

- `selwake_pending_drops` counter grows under load.
 - `bulk_writer_flood` triggers coalescing into a single callback.
 - `reset_delayed` fires after the configured delay.
 - Re-arming `reset_delayed` replaces the deadline (only one fire).
7. **Detach under load:** tick source at 1 ms, `poll_waiter` running, `bulk_writer_flood` issuing floods, then immediate unload. Should be clean.
 8. **WITNESS pass:** every test above, with no WITNESS warnings in `dmesg`.
 9. **lockstat pass:** run the test suite under `lockstat -s 5` to measure lock contention. The taskqueue's internal mutex should show up only briefly.

Every test should pass. If any fails, the cause is almost certainly a regression introduced between Stage 3 and Stage 4, not a pre-existing issue; Stages 1-3 are each validated independently before Stage 4 starts.

14.9.5 Keeping Documentation in Sync

Three places where documentation should reflect Chapter 14:

- The top-of-file comment in `myfirst.c`. Update the “locking strategy” block to mention the taskqueue.
- `LOCKING.md`. Update per the earlier subsection.
- Any per-chapter `README.md` under `examples/part-03/ch14-taskqueues-and-deferred-work/`. Describe each stage's deliverables and how to build them.

Updating documentation feels like overhead. It is not. Next year's reader (often your future self) depends on the documentation to reconstruct the design. Writing it now, while the design is fresh, is an order of magnitude cheaper than writing it later.

14.9.6 A Final Audit

Before closing Stage 4, run through a short audit.

- ☐ Does every callout drain happen before every task drain in detach?
- ☐ Does every task drain happen before `seldrain`?
- ☐ Does `taskqueue_free` happen after all tasks are drained?
- ☐ Does the attach failure path `taskqueue_free` the queue if a subsequent init step fails?
- ☐ Is every enqueue call site targeting the right taskqueue pointer (private, not shared)?
- ☐ Is every drain call site matched with its enqueue call site (same taskqueue, same task)?
- ☐ Is every task callback free of assumptions that “I run exactly once per enqueue”?
- ☐ Is every task callback free of assumptions that “I hold a driver lock on entry”?
- ☐ Does `LOCKING.md` list every task and its callback, lifetime, and enqueue paths?
- ☐ Does the version string reflect the new stage?

A driver that passes this audit, plus Chapter 13's audit from Section 7 of that chapter, is a driver you can hand to another engineer with confidence.

14.9.7 Wrapping Up Section 8

Stage 4 is the consolidation. The driver code is organised. `LOCKING.md` is current. The version string reflects the new capability. The full regression suite passes under a debug kernel. The audit checklist is clean.

The `myfirst` driver has come a long way. It started Chapter 10 as a single-open character device that moved bytes through a circular buffer. Chapter 11 gave it concurrent access. Chapter 12 gave it bounded blocking, cv channels, and sx-protected configuration. Chapter 13 gave it callouts for periodic and watchdog work. Chapter 14 gives it deferred work, which is the bridge between edge contexts and thread context and the missing piece for a driver that will eventually face real hardware interrupts.

The rest of this chapter expands the view slightly. The additional topics section introduces `epoch(9)`, group-taskqueues, and per-CPU taskqueues at an introductory level. Hands-on labs consolidate the Section 3 through Section 8 material. Challenge exercises stretch the reader. A troubleshooting reference gathers the common issues into one place. Then the wrapping-up and the bridge to Chapter 15.

14.10 Additional Topics: `epoch(9)`, Group-taskqueues, and Per-CPU Taskqueues

The main body of Chapter 14 taught the `taskqueue(9)` patterns a typical driver needs. Three adjacent topics deserve a mention for readers who will eventually write or read network drivers, or whose drivers grow to a scale where the simple private taskqueue is not enough. Each topic is introduced at the “know when to reach for it” level. The full mechanics belong to later chapters, especially those covering network drivers in Part 6 (Chapter 28).

14.10.1 `epoch(9)` in One Page

`epoch(9)` is a lockless read-synchronisation mechanism. Its purpose is to allow many readers to walk a shared data structure concurrently, without acquiring any exclusive lock, while guaranteeing that the data structure will not disappear underneath them.

The shape is this. Code that reads shared data enters an “epoch section” with `epoch_enter(epoch)` and leaves it with `epoch_exit(epoch)`. Inside the section, readers can dereference pointers freely. Writers who want to change or free a shared object do not do so directly; instead, they either call `epoch_wait(epoch)` to block until every current reader has left the epoch section, or they register a callback via `epoch_call(epoch, cb, ctx)` that runs asynchronously after all current readers have left.

The benefit is scalability. Readers pay no atomic-operation cost; they simply record thread-local state on entry and exit. Writers pay the synchronisation cost, but writes are rare compared to reads, so the amortised cost is low. For data structures that are walked by many threads and changed only occasionally, `epoch(9)` beats reader-writer locks substantially.

The cost is discipline. Code inside an epoch section must not sleep, must not acquire sleepable locks, and must not call functions that might do either. Writers who use `epoch_wait` block until every current reader has left, which means the writer is waiting for potentially a lot of readers.

Network drivers use `epoch(9)` heavily. The `net_epoch_preempt` epoch protects reads of network state (ifnet lists, routing entries, interface flags). A packet-input path enters the epoch, walks the state, exits the epoch.

A writer who wants to remove an interface defers the free via `NET_EPOCH_CALL` and the free happens on a taskqueue-like mechanism once every reader has finished.

For the taskqueue connection: when a task is initialised with `NET_TASK_INIT` instead of `TASK_INIT`, the taskqueue runs the callback inside the `net_epoch_preempt` epoch. The task callback can therefore walk network state without entering the epoch explicitly. From the implementation in `/usr/src/sys/kern/subr_taskqueue.c`:

```
if (!in_net_epoch && TASK_IS_NET(task)) {
    in_net_epoch = true;
    NET_EPOCH_ENTER(et);
} else if (in_net_epoch && !TASK_IS_NET(task)) {
    NET_EPOCH_EXIT(et);
    in_net_epoch = false;
}
task->ta_func(task->ta_context, pending);
```

The taskqueue dispatcher notices the `TASK_NETWORK` flag and enters or exits the epoch around the callback as needed. Consecutive network tasks share a single epoch entry, which is a small optimisation the framework does for free.

For myfirst, this is not relevant. The driver does not touch network state. But if you later write a network driver or read network driver code, `NET_TASK_INIT` and `TASK_IS_NET` are the macros that tell you a task is epoch-aware.

14.10.2 Grouptaskqueues in One Page

A grouptaskqueue is a scalable generalisation of a taskqueue. The basic idea: instead of a single queue with a single (or small) pool of workers, distribute tasks across many per-CPU queues, each serviced by its own worker thread. A “grouptask” is a task that binds to one of those queues.

The header is `/usr/src/sys/sys/gtaskqueue.h`:

```
#define GROUPTASK_INIT(gtask, priority, func, context) \
    GTASK_INIT(&(gtask)->gt_task, 0, priority, func, context)

#define GROUPTASK_ENQUEUE(gtask) \
    grouptaskqueue_enqueue((gtask)->gt_taskqueue, &(gtask)->gt_task)

void    taskqgroup_attach(struct taskqgroup *qgroup,
                        struct grouptask *grptask, void *uniq, device_t dev,
                        struct resource *irq, const char *name);
int     taskqgroup_attach_cpu(struct taskqgroup *qgroup,
                        struct grouptask *grptask, void *uniq, int cpu, device_t dev,
                        struct resource *irq, const char *name);
void     taskqgroup_detach(struct taskqgroup *qgroup, struct grouptask *gtask);
```

A driver that uses grouptasks does the following at attach:

1. Initialise each grouptask with `GROUPTASK_INIT`.
2. Attach each grouptask to a taskqgroup with `taskqgroup_attach` or `taskqgroup_attach_cpu`. The attachment assigns the grouptask to a specific per-CPU queue and worker.
3. At event time, enqueue with `GROUPTASK_ENQUEUE`.
4. At detach, `taskqgroup_detach` disassociates the grouptask.

Why use grouptasks instead of plain tasks? Two reasons.

First, **scalability with CPU count**. A single-threaded taskqueue is a bottleneck when many producers on different CPUs enqueue concurrently. The taskqueue's internal mutex becomes contended. A grouptaskqueue with per-CPU queues lets each CPU enqueue onto its own queue with no cross-CPU contention.

Second, **cache locality**. When an interrupt fires on CPU N and enqueues a grouptask that is bound to CPU N, the task runs on the same CPU that saw the interrupt. The task's data is already in that CPU's caches. For high-rate network drivers this is a substantial performance win.

The cost is complexity. Grouptaskqueues require more setup, more teardown, and more thinking about which queue a task belongs to. For most drivers this cost is not justified. For a high-end Ethernet driver that processes millions of packets per second, the cost pays for itself.

`myfirst` does not use grouptasks. It would not benefit. We mention them so that when you read a driver like `/usr/src/sys/dev/wg/if_wg.c` or `/usr/src/sys/net/iflib.c`, the macros look familiar.

14.10.3 Per-CPU Taskqueues in One Page

A per-CPU taskqueue is the simple version of the grouptaskqueue idea: one taskqueue per CPU, each with its own worker thread. The driver creates N taskqueues (one per CPU), binds each to a specific CPU with `taskqueue_start_threads_cpuset`, and dispatches tasks to the appropriate queue based on whatever locality rule the driver wants.

The key primitive is `taskqueue_start_threads_cpuset`:

```
int taskqueue_start_threads_cpuset(struct taskqueue **tqp, int count,
    int pri, cpuset_t *mask, const char *name, ...);
```

It is like `taskqueue_start_threads` but with a `cpuset_t` describing which CPUs the threads may run on. For a single-CPU binding the mask has exactly one bit set. For multi-CPU flexibility the mask has multiple bits.

A driver that uses per-CPU taskqueues typically keeps an array of taskqueue pointers indexed by CPU:

```
struct my_softc {
    struct taskqueue *per_cpu_tq[MAXCPU];
    ...
};

for (int i = 0; i < mp_ncpus; i++) {
    CPU_SETOF(i, &mask);
```

```

    sc->per_cpu_tq[i] = taskqueue_create("per_cpu", M_WAITOK,
        taskqueue_thread_enqueue, &sc->per_cpu_tq[i]);
    taskqueue_start_threads_cpuset(&sc->per_cpu_tq[i], 1, PWAIT,
        &mask, "%s cpu%d", device_get_nameunit(sc->dev), i);
}

```

And at enqueue time, pick the queue corresponding to the current CPU:

```

int cpu = curcpu;
taskqueue_enqueue(sc->per_cpu_tq[cpu], &task);

```

The benefits are the same as grouptasks, without the grouptask framework: work stays on the CPU it was produced on, CPU-local contention is eliminated, caches stay warm. The cost is that the driver manages its own per-CPU data structure.

For `myfirst` this is overkill. For a driver whose event rate exceeds tens of thousands of events per second, per-CPU taskqueues are worth considering. Grouptaskqueues are more general and usually preferred when the scalability story matters; per-CPU taskqueues are the lighter-weight alternative.

14.10.4 When to Use Which

A short decision tree.

- **Low-rate, thread-context, shared-queue is fine:** use `taskqueue_thread`. Easiest.
- **Low-rate, thread-context, isolation matters:** private taskqueue with `taskqueue_create` and `taskqueue_start_threads`. What `myfirst` uses.
- **High-rate, contention is the bottleneck:** per-CPU taskqueues or grouptaskqueues. Start with per-CPU; reach for grouptasks if you need the extra scalability features.
- **Network-path data:** `NET_TASK_INIT` and grouptaskqueues, following the patterns in network drivers.
- **Filter-interrupt context, must enqueue without sleeping:** `taskqueue_create_fast` or `taskqueue_fast`, since filter interrupts cannot use a sleep mutex.

Most drivers you write or read will fit one of the first two lines. The remainder are specialised cases that their chapters will walk through.

14.10.5 Wrapping Up Additional Topics

`epoch(9)`, grouptaskqueues, and per-CPU taskqueues are the scaling story for taskqueues. They share the same mental model as the basic API: enqueue from a producer, dispatch on a worker, obey the lock discipline, drain at teardown. The differences are in how many queues there are and how tasks are dispatched across them. For most drivers the basic API is enough; these advanced variants exist for when it is not.

The chapter now moves to hands-on labs.

14.11 Hands-On Labs

The labs consolidate the chapter's material into four practical exercises. Each lab uses the driver you have been evolving through Stages 1 through 4, plus a few small userspace helpers provided under `examples/part-03/ch14-taskqueues-and-deferred-work/labs/`.

Allocate one session per lab. If time is limited, Labs 1 and 2 are the most important; Labs 3 and 4 are worthwhile but more involved.

14.11.1 Lab 1: Observing a Taskqueue Worker Thread

Objective. Confirm that your driver's private taskqueue has a worker thread, that the thread sleeps when there is no work, and that the thread wakes and runs the callback when work is enqueued.

Setup. Load the Stage 2 driver (or Stage 4, either works). Make sure no other taskqueue-using process is flooding the system; the quieter the system, the easier the observation.

Steps.

1. Run `procstat -t | grep myfirst`. Record the PID and TID shown. The thread should be in state `sleep`.
2. Run `sysctl dev.myfirst.0.heartbeat_interval_ms=1000`. Wait a few seconds.
3. Run `procstat -t | grep myfirst` again. The thread may briefly appear in `run` state during a heartbeat firing; most of the time it is still `sleep` because the heartbeat does not enqueue a task. Confirm this is what you see. Note that the heartbeat runs in the callout thread, not the driver's taskqueue thread.
4. Run `sysctl dev.myfirst.0.tick_source_interval_ms=100`. Wait a few seconds.
5. Run `procstat -t | grep myfirst` again. The thread should now oscillate between `sleep` and `run` because the tick source is enqueueing a task ten times per second.
6. Stop the tick source with `sysctl dev.myfirst.0.tick_source_interval_ms=0`. Confirm the thread returns to permanent `sleep`.
7. Stop the heartbeat. Unload the driver. Confirm the thread disappears from `procstat -t`.

Expected outcome. You have directly observed the thread's lifecycle: created at `attach`, sleeps when idle, runs when dispatched, destroyed at `detach`. The observation is worth more than the two pages of explanation that produced it.

14.11.2 Lab 2: Measuring Coalescing Under Load

Objective. Produce a workload that stresses the taskqueue enough to trigger coalescing, then measure the coalescing rate using the `selwake_pending_drops` `sysctl`.

Setup. Load the Stage 4 driver. Compile `poll_waiter` as in Section 3.

Steps.

1. Start the `poll_waiter` in one terminal: `./poll_waiter > /dev/null`. Redirecting to `/dev/null` prevents the terminal from becoming a bottleneck.
2. In a second terminal, set the tick source to a fast rate: `sysctl dev.myfirst.0.tick_source_interval_ms=1`.

3. Wait ten seconds.
4. Read `sysctl dev.myfirst.0.stats.selwake_pending_drops`. Record the value.
5. Wait ten more seconds and read again. Compute the per-second rate.
6. Increase the tick source rate to see if coalescing increases: the minimum tick source interval is 1 ms, but you can combine with the `bulk_writer_flood` `sysctl` to produce burstier load:

```
# for i in $(seq 1 100); do sysctl dev.myfirst.0.bulk_writer_flood=1000; done
```

7. Read `selwake_pending_drops` after the flood.

Expected outcome. The number grows over time, more under burstier load, less under steady load. If the number stays at zero even under aggressive load, the taskqueue thread is fast enough to keep up; this is a good state, not a bug.

Variation. Run the same workload with a debug kernel (WITNESS enabled) and observe whether `dmesg` shows any WITNESS warnings. It should not.

14.11.3 Lab 3: Verify the Detach Ordering

Objective. Confirm that the detach path correctly drains tasks before freeing the state those tasks touch. Deliberately introduce the Section 7 bug (task drain after `seldrain`) and observe the race.

Setup. Start from Stage 4. Make a working copy of `myfirst.c`.

Steps.

1. In your working copy, reorder the drains in `myfirst_detach` so `seldrain` comes before `taskqueue_drain`:

```
/* BROKEN ORDER: */
MYFIRST_CO_DRAIN(&sc->heartbeat_co);
MYFIRST_CO_DRAIN(&sc->watchdog_co);
MYFIRST_CO_DRAIN(&sc->tick_source_co);
seldrain(&sc->rsl);
seldrain(&sc->wsl);
taskqueue_drain(sc->tq, &sc->selwake_task);
/* ... rest ... */
```

This is intentionally wrong.

2. Rebuild with the broken order.
3. Load the driver. Enable the tick source at 1 ms. Run the `poll_waiter`.
4. After a few seconds of data flow, unload the driver: `kldunload myfirst`.
5. Most often the unload succeeds. Occasionally, especially under load, the kernel panics. The panic stack typically includes `selwakeup` called from `myfirst_selwake_task`, after `seldrain` has run.
6. Restore the correct order. Rebuild. Run the same stress and repeat the unload many times.
7. Confirm that the correct order never panics.

Expected outcome. You have experienced the race directly. The takeaway is that “usually works” is not “works”. The correct order is an invariant you preserve even when the incorrect order seems to work on casual testing.

Note. On a production kernel the panic may not happen; the memory corruption can hide until something else crashes. Always run such experiments on a debug kernel with INVARIANTS and WITNESS enabled.

14.11.4 Lab 4: Coalescing vs. Adaptive Batching

Objective. Build a small modification that uses the pending argument to drive adaptive batching, and compare its behaviour to the fixed-batch `bulk_writer_task` from Stage 3.

Setup. Start from Stage 4.

Steps.

1. Add a new task to the driver: `adaptive_writer_task`. Its callback writes pending bytes (capped at 64) to the buffer. Use the pattern from Section 5.
2. Add a `sysctl` that enqueues `adaptive_writer_task` on demand:

```
static int
myfirst_sysctl_adaptive_enqueue(SYSCTL_HANDLER_ARGS)
{
    struct myfirst_softc *sc = arg1;
    int n = 0, i, error;

    error = sysctl_handle_int(oidp, &n, 0, req);
    if (error || req->newptr == NULL)
        return (error);
    for (i = 0; i < n; i++)
        taskqueue_enqueue(sc->tq, &sc->adaptive_writer_task);
    return (0);
}
```

3. Initialise the task in `attach`, drain it in `detach`.
4. Rebuild, load.
5. Issue 1000 enqueues via the `sysctl`: `sysctl dev.myfirst.0.adaptive_enqueue=1000`.
6. Read `sysctl dev.myfirst.0.stats.bytes_written`. Observe how many bytes were written.
7. Compare to Stage 3’s `bulk_writer_flood` with `bulk_writer_batch=1`. The fixed batch would write 1 byte (coalesced into one firing). The adaptive batch writes whatever pending was, up to 64.

Expected outcome. The adaptive task writes more bytes under burst load because it uses the coalescing information the kernel has already computed. For workloads where the per-event work should scale with the event count, this pattern is preferable to a fixed batch size.

Variation. Add a counter that records the largest pending value ever seen. Expose it as a `sysctl`. Under stress, you will see the peak pending grow as load increases.

14.12 Challenge Exercises

The challenges are optional stretches. They push the patterns this chapter established into territory the main text did not cover. Take your time with them; they are meant to consolidate understanding, not to introduce new material.

14.12.1 Challenge 1: Per-File-Handle Task

Modify the driver so that each open file handle has its own task. The task's job, when enqueued, is to emit a log line identifying the handle. Write a sysctl that enqueues every handle's task simultaneously.

Hints: - The struct `myfirst_fh` allocated in `myfirst_open` is the natural home for the per-handle task. - Initialise the task in `myfirst_open` after `malloc`. - Drain the task in `myfirst_fh_dtor` before `free`. - Enqueuing "every handle's task" requires a list of open handles. `devfs_set_cdevpriv` does not maintain such a list; you will have to build one in the softc, protected by the mutex.

Expected outcome: a demonstration of task ownership at a granularity finer than the driver. The challenge tests your understanding of lifetime ordering.

14.12.2 Challenge 2: Two-Tier Task Pipeline

Add a pipeline of two tasks. Task A receives data from the `write(2)` handler, transforms it (for simplicity, uppercases each byte), and enqueues Task B. Task B writes the transformed data to a secondary buffer and notifies waiters.

Hints: - The transformation work happens in the task callback, in thread context. The `write(2)` handler should not block waiting for the transformation. - You will need a small queue of pending transformations, protected by the mutex. - Task A pulls from the queue, transforms, and enqueues Task B with per-item state. Alternatively, Task B runs once per enqueue of A and handles whatever is in the queue.

Expected outcome: a mental model of how taskqueues can form pipelines, with each stage running in its own invocation. This is how complex drivers split work.

14.12.3 Challenge 3: Priority-Driven Task Ordering

Add two tasks with different priorities to the driver. The `urgent_task` has priority 10 and prints "URGENT". The `normal_task` has priority 0 and prints "normal". Write a sysctl handler that enqueues both tasks, normal first, urgent second.

Expected outcome: the `dmesg` output shows URGENT before normal, confirming that priority overrides enqueue order inside the queue.

14.12.4 Challenge 4: Blocking Reconfiguration

Implement a reconfiguration path that uses `taskqueue_block` and `taskqueue_quiesce`. The path should:

1. Block the taskqueue.

2. Quiesce (wait for running tasks to finish).
3. Perform the reconfiguration (say, resize the circular buffer).
4. Unblock.

Verify with `dttrace` that no task runs during the reconfiguration window.

Expected outcome: experience with `taskqueue_block` and `taskqueue_quiesce`, and an understanding of when those primitives are appropriate.

14.12.5 Challenge 5: Multi-Threaded Taskqueue

Modify Stage 4 to use a multi-threaded private taskqueue (say, four worker threads instead of one). Run the Lab 2 coalescing test. Observe what changes.

Expected outcome: Under load, the coalescing rate decreases because multiple worker threads drain the queue faster. Under very light load, nothing visible changes. The challenge shows how taskqueue configuration trades off under different workloads.

14.12.6 Challenge 6: Implementing a Watchdog Using a Timeout Task

Reimplement the Chapter 13 watchdog using a `timeout_task` instead of a plain callout. Each watchdog firing re-enqueues itself with the configured interval. A “kick” operation (another `sysctl`, perhaps `watchdog_kick`) cancels and re-enqueues the timeout task to reset the timer.

Expected outcome: understanding how the `timeout_task` primitive can replace a callout for periodic work, and when each is preferable. (Answer: `timeout_task` when the work needs thread context; callout otherwise.)

14.12.7 Challenge 7: Load a Real Driver and Read Its Code

Pick one of the drivers listed in Section 6 (`/usr/src/sys/dev/ale/if_ale.c`, `/usr/src/sys/dev/age/if_age.c`, `/usr/src/sys/dev/bge/if_bge.c`, or `/usr/src/sys/dev/iwm/if_iwm.c`). Read through its taskqueue usage. Identify:

- Which tasks the driver owns.
- Where each task is initialised.
- Where each task is enqueued.
- Where each task is drained.
- Whether the driver uses `taskqueue_create` or `taskqueue_create_fast`.
- What thread priority the driver uses.

Write a short summary (a page or so) of how the driver uses the taskqueue API. Keep it as a reference.

Expected outcome: Real driver reading turns pattern recognition from abstract to concrete. After doing this once with one driver, reading the next one is dramatically faster.

14.13 Troubleshooting Reference

A flat reference list of symptoms and remedies, for the moment when a bug surfaces and you need the answer fast. Pair this reference with the common-mistakes lists inside each section; together they cover most real problems.

14.13.1 The task never runs

- **Did you call `taskqueue_start_threads` after `taskqueue_create`?** Without threads, the queue accepts enqueues but never dispatches them.
- **Is the `taskqueue` pointer `NULL` at enqueue time?** Check the attach path; `taskqueue_create` may have failed silently if you did not check its return value.
- **Is the driver's `is_attached` false when the enqueue fires?** Some code paths (like Chapter 13's callout callbacks) exit early if `is_attached` is false; if the exit happens before the enqueue, the task does not run.
- **Is the `taskqueue` blocked via `taskqueue_block`?** If so, it accepts enqueues but does not dispatch. Unblock it.

14.13.2 The task runs twice when you expected once

- **Is the task self-reenqueuing?** A task callback that calls `taskqueue_enqueue` on itself will loop indefinitely unless the callback exits early under some condition.
- **Is a different code path also enqueueing?** Check every `taskqueue_enqueue` call site for the task. Two sources enqueueing will produce the expected double-run under some timing.

14.13.3 The task runs once when you expected twice or more

- **Did your enqueues coalesce?** The enqueue-already-pending rule folds redundant submissions. If you need exactly-one-per-event semantics, use separate tasks or a per-event queue.
- **Is the pending argument reported as greater than one?** If so, the framework coalesced.

14.13.4 Kernel panic in the task callback

- **Is the callback accessing freed state?** The most common cause of a panic in a task callback is a use-after-free. Check the detach ordering: every producer of enqueues must be drained before the task is drained; the state the task touches must not be freed until the task is drained.
- **Is the callback holding a lock it should not?** `WITNESS` catches most of these. Run under a debug kernel and read `dmesg`.
- **Is the callback calling `selwakeup` with a driver mutex held?** Do not. `selwakeup` acquires its own lock and must not be called with unrelated driver locks held.

14.13.5 Detach hangs

- **Is `taskqueue_drain` waiting for a task that cannot complete?** Check with `procstat -kk` the state of the taskqueue worker. If it is in a wait that depends on something the detach path holds, the design has a cycle.
- **Is `taskqueue_free` waiting for tasks that are still being enqueued?** Check `is_attached`: if callouts are still running and still enqueueing, the drain will not terminate. Ensure callouts are drained first.

14.13.6 `kldunload` returns `EBUSY` immediately

- **Is a file descriptor still open?** The detach path in `myfirst_detach` rejects with `EBUSY` if `active_fhs > 0`. Close any open descriptors and retry.

14.13.7 Coalescing count stays at zero

- **Is the workload too light?** Coalescing happens only when the producer outruns the consumer. On a lightly-loaded machine this rarely happens.
- **Is your measurement correct?** Coalescing is counted in the callback, not in the enqueue path. Check your counter logic.
- **Is the taskqueue multi-threaded?** More threads mean faster consumption, less coalescing.

14.13.8 The private taskqueue's thread does not appear in `procstat -t`

- **Did `taskqueue_start_threads` return zero?** If it returned an error, the threads were not created. Check the return value.
- **Is the driver actually loaded?** `kldstat` confirms.
- **Is the thread's name different from what you expected?** The format string to `taskqueue_start_threads` controls the name; make sure you are grepping for the right thing.

14.13.9 Deadlock between task and driver mutex

- **Does the task callback acquire a lock that a different thread holds while waiting for the task?** That is the textbook deadlock shape. Break it by moving the task enqueue outside the lock-held section, or by restructuring the wait so it does not block on the task.

14.13.10 `taskqueue_enqueue` fails with `EEXIST`

- **You passed `TASKQUEUE_FAIL_IF_PENDING` and the task was already pending.** The failure is intentional; check whether the flag is what you want.

14.13.11 `taskqueue_enqueue_timeout` does not seem to fire

- **Is the taskqueue blocked?** Blocked queues do not dispatch timeout tasks either.

- **Is the tick count reasonable?** A tick count of zero fires immediately, but a non-integer millisecond-to-tick conversion can produce unexpectedly long delays. Use $(\text{ms} * \text{hz} + 999) / 1000$ for ceil.
- **Has the timeout task been cancelled by a `taskqueue_cancel_timeout`?** If so, re-enqueue.

14.13.12 Re-arming a `timeout_task` is not replacing the deadline

- **Each `taskqueue_enqueue_timeout` replaces the pending deadline.** If your driver calls it multiple times and only the first seems to take effect, you may have an ordering issue: are you sure the subsequent calls are happening?

14.13.13 WITNESS complains about a lock order involving `tq_mutex`

- **The taskqueue's internal mutex is entering your driver's lock order.** Usually because a task callback acquires a driver lock, and some other code path acquires that driver lock first and then enqueues.
- **The resolution is usually to enqueue before acquiring the driver lock, or to restructure the code so the two locks are never held in the same thread in the wrong order.**

14.13.14 `procstat -kk` shows the taskqueue thread sleeping on a lock

- **A task callback is blocked on a sleepable lock.** Identify the lock from the wait channel. Check whether the holder of that lock is also waiting for something; if so, you have a dependency chain.

14.13.15 Task callbacks are slow

- **Profile with `dtrace`.** The one-liner from Section 7 measures callback duration.
- **Is the callback holding a lock during long operations?** Move the long operation outside the lock.
- **Is the callback performing synchronous IO?** That belongs in `read(2)` / `write(2)` / `ioctl(2)` handlers, not in taskqueue callbacks, unless the IO is genuinely the point of the task.

14.13.16 Taskqueue deadlock during boot

- **Are you enqueueing tasks from an `SI_SUB` that runs before `SI_SUB_TASKQ`?** The predefined taskqueues are initialised at `SI_SUB_TASKQ`. Earlier `SI_SUB` handlers cannot enqueue onto them.

14.14 Wrapping Up

Chapter 14 taught one primitive in depth. The primitive is `taskqueue(9)`. Its purpose is to move work out of contexts that cannot do the work and into a context that can. Its API is small: initialise a task, enqueue it, drain it, free the queue when you are done. The mental model is equally small: edge contexts detect, thread-context tasks act.

The `myfirst` driver absorbed the new mechanism gracefully because every earlier chapter had prepared the scaffolding. Chapter 11 gave it concurrency. Chapter 12 gave it `cv` channels and `sx` configuration. Chapter 13

gave it callouts and the drain-at-detach discipline. Chapter 14 added tasks as a fifth primitive in the same shape: initialise in attach, drain in detach, obey the established locking rules, and compose with what came before. The driver is now version 0.8-taskqueues, with three tasks and one private taskqueue, and it tears down cleanly under load.

Underneath those concrete changes, the chapter made several larger points. A short recap of each.

Deferred work is the bridge between edge contexts and thread contexts. Callouts, interrupt filters, and epoch sections all face the same constraint: they cannot do work that requires sleeping or sleepable-lock acquisition. A taskqueue solves the problem uniformly by accepting a small submission from the edge context and running the real work on a thread.

The taskqueue framework handles the logistics so you do not have to. Allocation, locking of the internal queue, dispatch, coalescing, cancellation, drain: every one of those is handled by the framework. Your driver provides a callback and an enqueue site. The rest is a short setup in attach and a short teardown in detach.

Coalescing is a feature, not a bug. A task coalesces redundant enqueues into a single firing whose pending argument carries the tally. This lets bursts of events fold into a single callback invocation, which is almost always what you want for performance. A design that needs per-event invocation needs a per-event task or a per-event queue, not one task enqueued many times.

Detach ordering is the biggest new discipline the chapter added. Callouts first, tasks second, selinfo third, taskqueue_free last. Violating the order is a race that may not appear on quiet testing and will appear under load. The `LOCKING.md` document is where you write the order down; following it is where you avoid the race.

Real drivers all use the same handful of patterns. Deferred logging from edge context; filter-plus-task interrupt splits; async copyin/copyout; retry-with-backoff; deferred teardown; scheduled rollover; block-during-reconfiguration. Every one of them is a variation on the edge-detects, task-acts shape. Reading `/usr/src/sys/dev/` is the fastest way to absorb the patterns in depth.

The taskqueue story scales up. epoch(9), group taskqueues, and per-CPU taskqueues handle the scalability cases that a simple private taskqueue cannot. They share the mental model of the basic API; the differences are in the number of queues, the dispatch strategy, and the scaffolding around the workers. For most drivers the basic API is enough; for the high-end cases, the advanced variants are there when you need them.

14.14.1 A Reflection Before Chapter 15

You started Chapter 14 with a driver that could act on its own through time (callouts) but whose actions were constrained to what callouts allow. You leave with a driver that can act through time and also act through work handed off to a worker thread. Those two combine to cover almost every kind of deferred action a driver needs. The third piece is the synchronisation that coordinates the producers and consumers safely, and that piece is what Chapter 15 develops.

The mental model is cumulative. Chapter 12 introduced cvs, mutexes, and sx. Chapter 13 introduced callouts and the no-sleep context. Chapter 14 introduced tasks and the defer-to-thread context. Chapter 15 will introduce the advanced coordination primitives (semaphores, cv_timedwait_sig, cross-component handshakes) that compose the earlier pieces into richer patterns. Each chapter adds a small primitive and the discipline that goes with it.

The cumulative shape of the driver is visible in `LOCKING.md`. A Chapter 10 driver had no `LOCKING.md`. A Chapter 11 driver had a single paragraph. A Chapter 14 driver has a multi-page document with sections for the

mutex, the cvs, the sx, the callouts, and the tasks, plus a detach-ordering section that names every drain step in the correct order. That document is an artefact you carry forward into every future chapter. When Chapter 15 adds semaphores, `LOCKING.md` grows a Semaphores section. When Part 4 adds interrupts, it grows an Interrupts section. The driver's lifetime is its `LOCKING.md`.

14.14.2 A Second Reflection: The Discipline

One habit the chapter wants you to internalise above all others: every new primitive in your driver earns an entry in `LOCKING.md`, a destructor pair in attach/detach, and a place in the documented detach ordering. Skipping any of those creates a bug waiting to happen. The discipline pays for itself the first time you hand the driver to someone else.

The inverse is also true: every time you read someone else's driver, look at their `LOCKING.md` first. If it is missing, read the attach and detach functions to reconstruct the ordering from code. If you see a primitive in attach with no corresponding drain in detach, that is a bug. If you see a drain without a clear predecessor, that is probably an ordering error. The discipline is the same for writing and for reading.

14.14.3 A Short Note on Simplicity

Taskqueues look simple. They are. The API is small, the patterns are regular, and the idioms transfer across drivers. The simplicity is deliberate; it is what makes the API usable in practice. The same simplicity also makes the rules non-negotiable: a rule that is skipped produces a race that is hard to debug. Follow the discipline, and taskqueues will stay simple for you. Improvise, and they will not.

14.14.4 What to Do If You Are Stuck

If something in your driver is not behaving as expected, work the troubleshooting reference from Section 7 in order. Check the first item that matches your symptom. If no item matches, re-read Section 4 (setup and cleanup) and audit your detach ordering against `LOCKING.md`. If the ordering is correct, `dtrace` the enqueue path and see whether the expected events are happening.

If the driver panics, use `gdb` on the crash dump. A bt shows the stack. A stack that includes your task callback is a good place to start; compare it against the patterns in Section 7.

If all else fails, re-read the real driver you picked for Challenge 7 in the previous section. Sometimes the pattern that seems confusing in your own code reads naturally in a driver someone else wrote. The patterns are universal; the driver you are reading is not special.

14.15 Bridge to Chapter 15

Chapter 15 is titled *More Synchronization: Conditions, Semaphores, and Coordination*. Its scope is the advanced coordination primitives that compose the mutexes, cvs, sx locks, callouts, and tasks you now have into more sophisticated patterns.

Chapter 14 prepared the ground in four specific ways.

First, **you now have a working producer/consumer pair in the driver**. The `tick_source` callout (and the other Chapter 14 enqueue sites) is a producer; the taskqueue thread is a consumer. The cv channels from Chapter 12 are another producer/consumer pair: `write(2)` produces, `read(2)` consumes. Chapter 15 generalises the pattern and adds primitives (counting semaphores, interruptible cv waits) that handle more elaborate versions of the same shape.

Second, **you know the discipline of drain-at-detach**. Every primitive you have added so far has a matching drain. Chapter 15 introduces semaphores, which have their own “drain” pattern (release all waiters, then destroy), and the discipline transfers directly.

Third, **you know how to think about context boundaries**. The callout context, the task context, the syscall context: each has its own rules, and your driver’s design respects them. Chapter 15 adds signal-interruptible waits, which add user-interaction context to the mix. The habit of “which context am I in and what can I do here” transfers.

Fourth, **your `LOCKING.md` is in the rhythm of “one section per primitive, plus an ordering section at the end”**. Chapter 15 will add a Semaphores section and possibly a Coordination section. The structure is established; only the content changes.

Specific topics Chapter 15 will cover:

- Counting semaphores and binary semaphores via the `sema(9)` API.
- `cv_timedwait_sig` and signal-interruptible blocking.
- Readers-writers patterns via `sx(9)` in their fully-general form.
- Cross-component handshakes that coordinate timers, tasks, and user threads.
- State flags and memory barriers for lock-free coordination (at an introductory level).
- Testing harnesses for concurrency: stress scripts, fault-injection, replay of races.

You do not need to read ahead. Chapter 14 is sufficient preparation. Bring your `myfirst` driver at Stage 4 of Chapter 14, your `LOCKING.md`, your WITNESS-enabled kernel, and your test kit. Chapter 15 starts where Chapter 14 ended.

A small closing reflection. The driver you started Part 3 with understood one syscall at a time. The driver you have now has three callouts, three tasks, two cvs, two locks, and a full detach ordering. It handles concurrent readers and writers, timed events, deferred work across context boundaries, coalesced bursts of events, and clean teardown under load. It has a documented locking story and a validated regression suite. It is starting to look like a real FreeBSD driver.

Chapter 15 closes out Part 3 by adding the coordination primitives that let the pieces compose into richer patterns. Then Part 4 begins: hardware and platform-level integration. Real interrupts. Real memory-mapped registers. Real hardware that might fail, misbehave, or refuse to cooperate. The discipline you have built through Part 3 is what will carry you through.

Take a moment before moving on. The leap from Chapter 13 to Chapter 14 was a qualitative one: the driver gained the ability to defer work to a thread context, and the patterns you learned along the way (detach ordering, coalescing, the edge/thread mental model) are patterns you will reuse in every chapter that follows. From here, Chapter 15 consolidates the synchronisation story, and Part 4 begins the hardware story. The work you have done is not lost; it compounds.

14.15.1 A Final Aside on the Shape of the Kernel

One last thought before Chapter 15. You have now met five of the kernel's synchronisation-and-deferral primitives: the mutex, the condition variable, the sx lock, the callout, and the taskqueue. Each exists because earlier, simpler primitives could not address the same problem. A mutex cannot express “wait for a condition”; that is what cvs do. A sleep mutex cannot be held across a sleep; that is what sx locks allow. A callout cannot run work that needs to sleep; that is what taskqueues allow.

The pattern is recognisable across the kernel. Every synchronisation primitive exists because of a specific gap in the earlier ones. When you read kernel code, you can often guess why a particular primitive was chosen by asking what its neighbours cannot do. A file descriptor cannot suffer a use-after-free because the reference-count primitive prevents it. A network packet cannot be freed while a reader is walking the list because the epoch primitive prevents it. A task cannot run after detach because the drain primitive prevents it.

The kernel is a catalogue of these primitives, each deliberate, each a response to a specific class of problem. Your driver accumulates its own catalogue as it grows. Chapter 14 added one more item to the list. Chapter 15 adds several. Part 4 begins the hardware-facing catalogue. From here on the primitives multiply, but the shape of the discipline does not change. Define the problem, pick the primitive, initialise and tear down cleanly, document the ordering, verify under load.

That is the craft. The rest of the book walks you through it.

14.16 Reference: Pre-Production Taskqueue Audit

A short audit to perform before promoting a taskqueue-using driver from development to production. Each item is a question; each should be answerable with confidence.

14.16.1 Task Inventory

- ☐ Have I listed every task the driver owns in `LOCKING.md`?
- ☐ For each task, have I named its callback function?
- ☐ For each task, have I documented its lifetime (init in `attach`, drain in `detach`)?
- ☐ For each task, have I documented its trigger (what causes it to be enqueued)?
- ☐ For each task, have I documented whether it self-reenqueues or runs once per external trigger?
- ☐ For each timeout task, have I named the interval it is scheduled for and the cancellation path?

14.16.2 Taskqueue Inventory

- ☐ Is the taskqueue a private queue or a predefined one? Is the choice justified?
- ☐ If private, does `attach` call `taskqueue_create` (or `taskqueue_create_fast`) before any code that could enqueue?
- ☐ If private, does `attach` call `taskqueue_start_threads` before any code that expects a callback to fire?
- ☐ Is the worker thread count appropriate for the workload?
- ☐ Is the worker thread priority appropriate for the workload?
- ☐ Is the worker thread name informative enough that `procstat -t` output is useful?

14.16.3 Initialisation

- ☐ Does every `TASK_INIT` happen after the `softc` is zeroed and before the task can be enqueued?
- ☐ Does every `TIMEOUT_TASK_INIT` reference the correct taskqueue and a valid callback?
- ☐ Does `attach` handle `taskqueue_create` failure by unwinding earlier initialisations?
- ☐ Does `attach` handle `taskqueue_start_threads` failure by freeing the taskqueue?

14.16.4 Enqueue Sites

- ☐ Does every enqueue site target the right taskqueue pointer?
- ☐ Does every enqueue from an edge context (callout, interrupt filter) confirm the taskqueue exists before enqueueing?
- ☐ Is the enqueue call safe in the context it happens from (not inside a spin mutex if the taskqueue is `taskqueue_create`, for example)?
- ☐ Is the coalescing behaviour intentional at every enqueue site?

14.16.5 Callback Hygiene

- ☐ Does every callback have the correct signature (`void *context, int pending`)?
- ☐ Does every callback acquire driver locks explicitly where needed?
- ☐ Does every callback release driver locks before calling `selwakeup`, `log`, or other functions that acquire unrelated locks?
- ☐ Does every callback avoid `M_NOWAIT` allocations where `M_WAITOK` is safe?
- ☐ Is the callback's total work time bounded?

14.16.6 Cancellation

- ☐ Does every `taskqueue_cancel` / `taskqueue_cancel_timeout` call happen under the right mutex if the cancellation race matters?
- ☐ Are the cases where `cancel` returns `EBUSY` handled (usually by a follow-up drain)?

14.16.7 Detach

- ☐ Does `detach` clear `is_attached` before draining callouts?
- ☐ Does `detach` drain every callout before draining any task?
- ☐ Does `detach` drain every task before calling `sel drain`?
- ☐ Does `detach` call `sel drain` before `taskqueue_free`?
- ☐ Does `detach` call `taskqueue_free` before destroying the mutex?
- ☐ Does `detach` set `sc->tq = NULL` after free?

14.16.8 Documentation

- ☐ Is every task documented in `LOCKING.md`?
- ☐ Are the discipline rules (enqueue-safe, callback-lock, drain-order) documented?

- ☐ Is the taskqueue subsystem mentioned in the README?
- ☐ Are there sysctls exposed that let users observe the behaviour?

14.16.9 Testing

- ☐ Have I run the regression suite with WITNESS enabled?
- ☐ Have I tested detach with all tasks in flight?
- ☐ Have I run a long-duration stress test with high enqueue rates?
- ☐ Have I used dtrace to verify the enqueue-to-dispatch latency is within expectations?
- ☐ Have I used procstat -kk under load to confirm the taskqueue thread is not stuck?

A driver that passes this audit is a driver you can trust under load.

14.17 Reference: Standardising Tasks Across a Driver

For a driver with several tasks, consistency matters more than cleverness. A short discipline.

14.17.1 One Naming Convention

Pick a convention and follow it. The chapter's convention:

- The task struct is named `<purpose>_task` (e.g., `selwake_task`, `bulk_writer_task`).
- The timeout-task struct is named `<purpose>_delayed_task` (e.g., `reset_delayed_task`).
- The callback is named `myfirst_<purpose>_task` (e.g., `myfirst_selwake_task`, `myfirst_bulk_writer_task`).
- The sysctl that enqueues the task (if one exists) is named `<purpose>_enqueue` or `<purpose>_flood` for bulk variants.
- The sysctl that configures the task (if one exists) is named `<purpose>_<parameter>` (e.g., `bulk_writer_batch`).

A new maintainer can add a new task following the convention without thinking about names. Conversely, a code review immediately catches deviations.

14.17.2 One Init/Drain Pattern

Every task uses the same initialisation and drain:

```
/* In attach, after taskqueue_start_threads: */
TASK_INIT(&sc-><purpose>_task, 0, myfirst_<purpose>_task, sc);

/* In detach, after callout drains, before seldrain: */
taskqueue_drain(sc->tq, &sc-><purpose>_task);
```

For timeout tasks:

```

/* In attach: */
TIMEOUT_TASK_INIT(sc->tq, &sc-><purpose>_delayed_task, 0,
    myfirst_<purpose>_delayed_task, sc);

/* In detach: */
taskqueue_drain_timeout(sc->tq, &sc-><purpose>_delayed_task);

```

The call sites are short and uniform. A reviewer can scan for the pattern and flag deviations instantly.

14.17.3 One Callback Pattern

Every task callback follows the same structure:

```

static void
myfirst_<purpose>_task(void *arg, int pending)
{
    struct myfirst_softc *sc = arg;

    /* Optional: record coalescing for diagnostics. */
    if (pending > 1) {
        MYFIRST_LOCK(sc);
        sc-><purpose>_drops += pending - 1;
        MYFIRST_UNLOCK(sc);
    }

    /* ... do the work, acquiring locks as needed ... */
}

```

The optional coalescing record makes the coalescing behaviour visible through sysctl. Drop it if the task rarely coalesces or if the counter would not be useful.

14.17.4 One Documentation Pattern

Every task is documented in `LOCKING.md` with the same fields:

- Task name and kind (plain or `timeout_task`).
- Callback function.
- Which code paths enqueue it.
- Which code paths cancel it (if any).
- Where it is drained at detach.
- What lock, if any, the callback acquires.
- Why this task is deferred (i.e., why it cannot run in the enqueue context).

A new task's documentation is mechanical. A code review can verify the documentation against the code.

14.17.5 Why Standardise

Standardisation has costs: a new contributor must learn the conventions; deviations require a special reason. The benefits are larger:

- Reduced cognitive load. A reader who knows the pattern instantly understands every task.
- Fewer mistakes. The standard pattern handles the common cases (init in attach, drain in detach, drop lock before selwakeup) correctly; a deviation is more likely to be wrong.
- Easier review. Reviewers can scan for shape rather than reading every line.
- Easier handoff. A maintainer who has not seen the driver can add a new task following the existing template.

The cost of standardisation is paid once at design time. The benefits accrue forever. Always worth it.

14.18 Reference: Further Reading on Taskqueues

For readers who want to go deeper.

14.18.1 Manual Pages

- `taskqueue(9)`: the canonical API reference.
- `epoch(9)`: the epoch synchronisation framework, relevant for network tasks.
- `callout(9)`: the companion primitive; `timeout_task` is built on top of it.
- `swi_add(9)`: the software-interrupt registration used by `taskqueue_swi` and friends.
- `kproc(9)`, `kthread(9)`: direct kernel-thread creation, for when a taskqueue is not enough.

14.18.2 Source Files

- `/usr/src/sys/kern/subr_taskqueue.c`: the taskqueue implementation. Read `taskqueue_run_locked` carefully; it is the heart of the subsystem.
- `/usr/src/sys/sys/taskqueue.h`, `/usr/src/sys/sys/_task.h`: the public API and structures.
- `/usr/src/sys/kern/subr_gtaskqueue.c`, `/usr/src/sys/sys/gtaskqueue.h`: the group taskqueue layer.
- `/usr/src/sys/sys/epoch.h`, `/usr/src/sys/kern/subr_epoch.c`: the epoch framework.
- `/usr/src/sys/dev/ale/if_ale.c`: a clean Ethernet driver using taskqueues.
- `/usr/src/sys/dev/bge/if_bge.c`: a larger Ethernet driver with multiple tasks.
- `/usr/src/sys/dev/wg/if_wg.c`: WireGuard's group taskqueue usage.
- `/usr/src/sys/dev/iwm/if_iwm.c`: a wireless driver with timeout tasks.
- `/usr/src/sys/dev/usb/usb_process.c`: USB's dedicated per-device process queue (`usb_proc_*`).

14.18.3 Manual Pages To Read in Order

For a reader new to FreeBSD's deferred-work subsystem:

1. `taskqueue(9)`: the canonical API.

2. `epoch(9)`: the lockless read-synchronisation framework.
3. `callout(9)`: the sibling timed-execution primitive.
4. `swi_add(9)`: the software-interrupt layer beneath some taskqueues.
5. `kthread(9)`: the direct-thread-creation alternative.

Each builds on the previous; reading in order takes a couple of hours and gives you a solid mental model of the kernel's deferred-work infrastructure.

14.18.4 External Material

The chapter on synchronization in *The Design and Implementation of the FreeBSD Operating System* (McKusick et al.) covers the historical evolution of deferred-work subsystems. Useful as context; not required.

The FreeBSD developers' mailing list (`freebsd-hackers@`) occasionally discusses taskqueue improvements and edge cases. Searching the archive for "taskqueue" returns relevant historical context.

For deeper understanding of the network stack's use of `epoch(9)` and group taskqueues, the `iflib(9)` framework documentation and the source under `/usr/src/sys/net/iflib.c` are worth a read. They are beyond the level of this chapter, but they explain why modern network drivers are structured the way they are.

Finally, real driver source. Pick any driver in `/usr/src/sys/dev/` that uses taskqueues (most do), read its taskqueue-related code, and compare it to the patterns in this chapter. The translation is direct; you will recognize the shapes immediately. That kind of reading turns the chapter's abstractions into working knowledge.

14.19 Reference: Taskqueue Cost Analysis

A short discussion of what taskqueues actually cost, useful when deciding whether to defer or whether to create a private queue.

14.19.1 Cost At Rest

A `struct task` that has not been enqueued costs nothing beyond the size of the structure (32 bytes on amd64). The kernel does not know about it. It sits in your softc, doing nothing.

A `struct taskqueue` allocated and idle costs: - The taskqueue structure itself (a few hundred bytes). - One or more worker threads (16 KB stack each on amd64, plus scheduler state). - No per-enqueue cost at rest.

14.19.2 Cost Per Enqueue

When you call `taskqueue_enqueue(tq, &task)`, the kernel does:

1. Acquire the taskqueue's internal mutex. Microseconds.
2. Check whether the task is already pending. Constant time.
3. If not pending, link into the list and wake the worker (via wakeup on the queue). Constant time plus a scheduler event.
4. If already pending, increment `ta_pending`. Single arithmetic.
5. Release the mutex.

The total cost is microseconds on an uncontended queue. Under contention the mutex acquisition can take longer, but the framework uses a padalign mutex to minimise false sharing, and the mutex is rarely held for more than a few instructions.

14.19.3 Cost Per Dispatch

When the worker thread wakes and runs `taskqueue_run_locked`, the per-task cost is:

1. Walk to the head of the queue. Constant time.
2. Pull the task off. Constant time.
3. Record the pending count, reset it. Constant time.
4. Drop the mutex.
5. Enter any required epoch (for network tasks).
6. Call the callback. Cost depends on the callback.
7. Exit epoch if entered.
8. Re-acquire the mutex for the next iteration.

For a typical short callback (microseconds of work), the per-dispatch overhead is dominated by the callback itself plus one mutex round-trip and one wakeup round-trip.

14.19.4 Cost At Cancel/Drain

`taskqueue_cancel` is fast: mutex acquisition, list removal if pending, mutex release. Microseconds.

`taskqueue_drain` is fast if the task is idle. If the task is pending, the drain waits for it to run and complete; duration depends on queue depth and callback duration. If the task is running, the drain waits for the current invocation to return.

`taskqueue_drain_all` is more expensive: it must wait for every task in the queue. Duration is proportional to the total work remaining.

`taskqueue_free` drains the queue, terminates the threads, and frees the state. The thread termination involves signalling each thread to exit and waiting for it to finish its current task. Microseconds to milliseconds depending on queue depth.

14.19.5 Practical Implications

A few practical notes.

Single-thread taskqueues are cheap. The per-instance cost is a few hundred bytes plus a 16-KB thread stack. On any realistic system, this is negligible.

Shared taskqueues are cheaper per driver but contended. `taskqueue_thread` is used by every driver that does not create its own. Under heavy load it becomes a serial bottleneck. For drivers with significant task traffic, a private queue avoids the contention.

Multi-thread taskqueues trade memory for parallelism. Four threads is four 16-KB stacks plus four scheduler entries. Worthwhile when the workload is naturally parallel; wasted when the workload serialises on a single driver mutex.

Coalescing is free performance. When enqueues come faster than the taskqueue can dispatch, coalescing folds the bursts into single firings. The driver pays one callback invocation for whatever work the pending count implies.

14.19.6 Comparison To Other Approaches

A kernel thread created with `kproc_create` and managed by the driver costs: - 16 KB stack plus scheduler entry (same as a taskqueue worker). - No built-in enqueue/dispatch framework: the driver rolls its own queue and wakeup. - No built-in coalescing or cancellation.

For work that fits the task model (enqueue, dispatch, drain), a taskqueue is always the right choice. For work that does not (a long-running loop with its own rhythm), a `kproc_create` thread may fit better.

A callout that enqueues a task combines the costs of both primitives. Worthwhile when the work needs both a specific deadline and thread context.

14.19.7 When To Worry About Cost

Most drivers do not have to. Taskqueues are cheap; the kernel is well-tuned. Worry about cost only when:

- Profiling shows taskqueue operations dominate CPU usage. (Use `dtrace` to confirm.)
- You are writing a high-rate driver (thousands of events per second or more) and the taskqueue is the serialisation point.
- The system has many drivers competing for `taskqueue_thread` and contention is measurable.

In all other cases, write the taskqueue naturally and trust the kernel to handle the load.

14.20 Reference: The Task Coalescing Semantics, Precisely

Coalescing is the feature that most often surprises newcomers. A precise statement of the semantics, with worked examples, is worth its own reference subsection.

14.20.1 The Rule

When `taskqueue_enqueue(tq, &task)` is called on a task that is already pending (`task->ta_pending > 0`), the kernel increments `task->ta_pending` and returns success. The task is not linked onto the queue a second time. When the callback eventually runs, it runs exactly once, with the accumulated `ta_pending` value passed as its second argument (and the field is reset to zero before the callback is called).

The rule has edge cases worth naming.

The cap. `ta_pending` is a `uint16_t`. It saturates at `USHRT_MAX` (65535). Enqueues past that point still return success but the counter does not grow further. In practice, reaching 65535 coalesced enqueues is a design problem, not a performance problem.

The `TASKQUEUE_FAIL_IF_PENDING` flag. If you pass this flag to `taskqueue_enqueue_flags`, the function returns `EEXIST` instead of coalescing. Useful when you want to know whether the enqueue produced a new pending state.

Timing. The coalescing happens at enqueue time. If enqueue A and enqueue B both happen while the task is pending, both coalesce. If enqueue A causes the task to start running, and enqueue B happens while the callback is executing, enqueue B makes the task pending again (`pending=1`) and the callback will be invoked again after the current invocation returns. The second invocation sees `pending=1` because only B has accumulated. Both the first and second invocations happen; no enqueue is lost.

Priority. If two different tasks are pending on the same queue and one has higher priority, the higher-priority one runs first regardless of enqueue order. Within a single task, priority is not a factor; all invocations of a given task run in sequence.

14.20.2 Worked Examples

Example 1: Simple single enqueue.

```
taskqueue_enqueue(tq, &task);  
/* Worker fires the callback. */  
/* Callback sees pending == 1. */
```

Example 2: Coalesced enqueue before dispatch.

```
taskqueue_enqueue(tq, &task);  
taskqueue_enqueue(tq, &task);  
taskqueue_enqueue(tq, &task);  
/* (Worker has not yet woken up.) */  
/* Worker fires the callback. */  
/* Callback sees pending == 3. */
```

Example 3: Enqueue during callback execution.

```
taskqueue_enqueue(tq, &task);  
/* Callback starts; pending is reset to 0. */  
/* While callback is running: */  
taskqueue_enqueue(tq, &task);  
/* Callback finishes its first invocation. */  
/* Worker notices pending == 1; fires callback again. */  
/* Second callback invocation sees pending == 1. */
```

Example 4: Cancel before dispatch.

```
taskqueue_enqueue(tq, &task);  
taskqueue_enqueue(tq, &task);  
/* Cancel: */  
taskqueue_cancel(tq, &task, &pendp);  
/* pendp == 2; callback does not run. */
```

Example 5: Cancel during execution.

```
taskqueue_enqueue(tq, &task);
/* Callback starts. */
/* During callback: */
taskqueue_cancel(tq, &task, &pendp);
/* Returns EBUSY; pending (if any future enqueues came in) may or may not be zeroed. */
/* The currently executing invocation completes; the cancellation affects only future
↳ runs. */
```

14.20.3 Design Implications

A few design implications follow from the rule.

Your callback must be idempotent over pending. Writing a callback that assumes `pending==1` breaks under load. Always use `pending` deliberately, either by looping `pending` times or by doing a single pass that handles whatever state has accumulated.

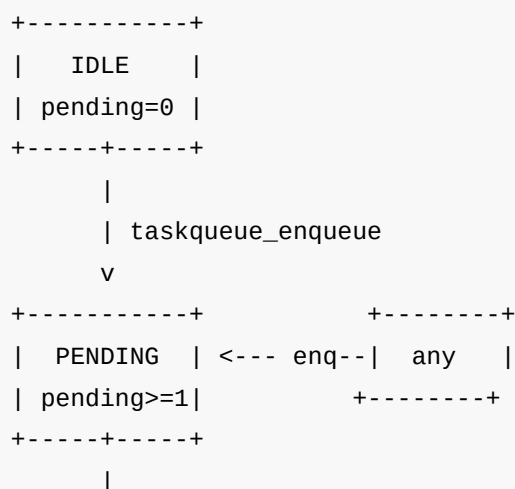
Do not use “number of callback invocations” as a count of events. Use `pending` from each invocation, summed. Or, better, use a per-event state structure (a queue inside the softc) that the callback drains.

Coalescing changes per-event work into per-burst work. A callback that does $O(1)$ work per invocation, with `pending` discarded, handles the same amount of work regardless of enqueue rate. That is usually fine for “notify waiters” kinds of work; it is wrong for “process each event” kinds of work.

Coalescing lets you enqueue freely from edge contexts. A callout that fires every millisecond can enqueue a task every millisecond; if the callback takes 10 ms to run, nine enqueues coalesce into one invocation per callback. The system converges naturally on the throughput the callback can sustain.

14.21 Reference: The Taskqueue State Diagram

A short state diagram for a single task, as an aid to reasoning about the lifecycle.




```

        | worker picks up
        v
+-----+
|  RUNNING  |
| (callback |
| executing)|
+-----+
        |
        | callback returns
        v
+-----+
|  IDLE     |
| pending=0 |
+-----+

```

A task is always in exactly one of three states: IDLE, PENDING, or RUNNING.

IDLE. Not on any queue. `ta_pending == 0`. Enqueue moves it to PENDING.

PENDING. On the taskqueue's pending list. `ta_pending >= 1`. Coalescing increments `ta_pending` without leaving PENDING. Cancellation moves it back to IDLE.

RUNNING. In `tq_active`, with the callback executing. `ta_pending` has been reset to zero, and the callback has received the previous value. New enqueues transition back to PENDING (so after the callback returns, the worker fires it again). Cancellation returns `EBUSY` in this state.

The state transitions are all serialised by `tq_mutex`. At any instant, the kernel can tell you which state a task is in, and the transitions are atomic.

`taskqueue_drain(tq, &task)` waits until the task is IDLE and no new enqueues arrive before returning. That is the precise guarantee the drain provides.

14.22 Reference: Observability Cheat Sheet

For quick reference during debugging.

14.22.1 List Every Taskqueue Thread

```
# procstat -t | grep taskq
```

14.22.2 List Every Task Submission Rate With DTrace

```
# dtrace -n 'fbt::taskqueue_enqueue:entry { @[caddr_t)arg1 = count(); }' -c 'sleep
↳ 10'
```

The script counts enqueues per task pointer over ten seconds. The task pointers map back to drivers via `addr2line` or `kgdb`.

14.22.3 Measure Dispatch Latency

```
# dtrace -n '
fbt::taskqueue_enqueue:entry { self->t = timestamp; }
fbt::taskqueue_run_locked:entry /self->t/ {
    @[execname] = quantize(timestamp - self->t);
    self->t = 0;
}
' -c 'sleep 10'
```

14.22.4 Measure Callback Duration

```
# dtrace -n '
fbt::taskqueue_run_locked:entry { self->t = timestamp; }
fbt::taskqueue_run_locked:return /self->t/ {
    @[execname] = quantize(timestamp - self->t);
    self->t = 0;
}
' -c 'sleep 10'
```

14.22.5 Stack Trace of the Taskqueue Thread When Stuck

```
# procstat -kk <pid>
```

14.22.6 Active Tasks Inside ddb

From the ddb prompt:

```
db> show taskqueues
```

Lists every taskqueue, its active task (if any), and the pending queue.

14.22.7 Sysctl Knobs Your Driver Should Provide

For every task the driver owns, consider exposing:

- `<purpose>_enqueues`: total enqueues attempted.
- `<purpose>_coalesced`: count of times coalescing fired.
- `<purpose>_runs`: total callback invocations.
- `<purpose>_largest_pending`: peak pending count.

Under normal conditions: `enqueues == runs + coalesced`. Under coalescing: `runs < enqueues`. Under no load: `largest_pending == 1`. Under heavy load: `largest_pending` grows.

These counters turn opaque driver behaviour into a readable sysctl display. The cost is a few atomic adds; the value is high.

14.23 Reference: A Minimal Working Task Template

For copy-and-adapt convenience. Every piece has been introduced in the chapter; the template assembles them into a ready-to-use skeleton.

```
#include <sys/param.h>
#include <sys/kernel.h>
#include <sys/module.h>
#include <sys/system.h>
#include <sys/bus.h>
#include <sys/taskqueue.h>
#include <sys/mutex.h>
#include <sys/lock.h>

struct example_softc {
    device_t      dev;
    struct mtx    mtx;
    struct taskqueue *tq;
    struct task    work_task;
    int           is_attached;
};

static void
example_work_task(void *arg, int pending)
{
    struct example_softc *sc = arg;

    mtx_lock(&sc->mtx);
    /* ... do work under the mutex if state protection is needed ... */
    mtx_unlock(&sc->mtx);
}
```

```
        /* ... do lock-free work or calls like selwakeup here ... */
    }

static int
example_attach(device_t dev)
{
    struct example_softc *sc = device_get_softc(dev);
    int error;

    sc->dev = dev;
    mtx_init(&sc->mtx, device_get_nameunit(dev), "example", MTX_DEF);

    sc->tq = taskqueue_create("example taskq", M_WAITOK,
        taskqueue_thread_enqueue, &sc->tq);
    if (sc->tq == NULL) {
        error = ENOMEM;
        goto fail_mtx;
    }
    error = taskqueue_start_threads(&sc->tq, 1, PWAIT,
        "%s taskq", device_get_nameunit(dev));
    if (error != 0)
        goto fail_tq;

    TASK_INIT(&sc->work_task, 0, example_work_task, sc);
    sc->is_attached = 1;
    return (0);

fail_tq:
    taskqueue_free(sc->tq);
fail_mtx:
    mtx_destroy(&sc->mtx);
    return (error);
}

static int
example_detach(device_t dev)
{
    struct example_softc *sc = device_get_softc(dev);

    mtx_lock(&sc->mtx);
    sc->is_attached = 0;
    mtx_unlock(&sc->mtx);
}
```

```

        taskqueue_drain(sc->tq, &sc->work_task);
        taskqueue_free(sc->tq);
        mtx_destroy(&sc->mtx);
        return (0);
}

/* Elsewhere, a code path that wants to defer work: */
static void
example_trigger_work(struct example_softc *sc)
{
    if (sc->is_attached)
        taskqueue_enqueue(sc->tq, &sc->work_task);
}

```

Every element is essential. Dropping any of them reintroduces a bug this chapter warned against.

14.24 Reference: Comparison With Linux Workqueues

A short comparison for readers coming from Linux kernel work. Both systems solve the same problem; the differences are in naming, granularity, and defaults.

14.24.1 Naming

Concept	FreeBSD	Linux
Unit of deferred work	struct task	struct work_struct
Queue	struct taskqueue	struct workqueue_struct
Shared queue	taskqueue_thread	system_wq
Unbound queue	taskqueue_thread with many threads	system_unbound_wq
Create a queue	taskqueue_create	alloc_workqueue
Enqueue	taskqueue_enqueue	queue_work
Enqueue with delay	taskqueue_enqueue_timeout	queue_delayed_work
Wait for work to finish	taskqueue_drain	flush_work
Destroy a queue	taskqueue_free	destroy_workqueue
Priority	ta_priority	WQ_HIGHPRI flag
Coalescing behaviour	Automatic, pending count exposed	work_pending check, no count

14.24.2 Semantic Differences

Coalescing visibility. FreeBSD exposes the pending count to the callback; Linux does not. A Linux callback knows the work fired but not how many times it was requested.

Timeout task vs. delayed work. FreeBSD's `timeout_task` embeds a callout; Linux's `delayed_work` embeds a `timer_list`. Both behave the same from the user's point of view.

Group taskqueues vs. percpu workqueues. FreeBSD's `taskqgroup` is explicit and separate; Linux's `alloc_workqueue(..., WQ_UNBOUND | WQ_CPU_INTENSIVE)` has similar semantics with different knobs.

Epoch integration. FreeBSD has `NET_TASK_INIT` for tasks that run inside the network epoch; Linux does not have a direct analogue (the RCU framework is similar but not identical).

A driver ported from Linux to FreeBSD (or vice versa) can usually translate the deferred-work pattern nearly one-to-one. The structural differences are in the surrounding APIs (device registration, memory allocation, locking) more than in the taskqueue itself.

14.25 Reference: When Not To Use a Taskqueue

A short list of scenarios where another primitive is preferable.

The work has tight timing requirements. Taskqueues add scheduling latency. For microsecond-scale deadlines, a `C_DIRECT_EXEC` callout or a `taskqueue_swi` is faster. For nanosecond-scale deadlines, none of the deferred-work mechanisms is fast enough; the work needs to happen inline.

The work is a one-shot cleanup that has no producer to associate with. A simple `free` inside a teardown path does not need a taskqueue; just call it directly. Deferral for deferral's sake adds no value.

The work must run at a specific scheduler priority higher than `PWAIT`. If the work is genuinely high-priority (a real-time driver, an interrupt-threshold task), use `kthread_add` with an explicit priority rather than a generic taskqueue.

The work requires a specific thread context that a generic worker cannot provide. Tasks run in a kernel thread with no specific user-process context. Work that needs a specific user's credentials, file-descriptor table, or address space must happen inside that process, not in a task.

The driver has only one task and it runs rarely. A single `kthread_add` with a `cv_timedwait` loop may be clearer than the full taskqueue setup. Use judgment; for three or more tasks, taskqueues are almost always clearer.

For everything else, use a taskqueue. The default is "use `taskqueue(9)`"; the exceptions are narrow.

14.26 Reference: A Worked Reading of `subr_taskqueue.c`

One more reading exercise, because understanding the implementation makes the API's behaviour predictable.

The file is `/usr/src/sys/kern/subr_taskqueue.c`. The structure, briefly:

struct taskqueue. Defined near the top of the file. Holds the pending queue (`tq_queue`), the active task list (`tq_active`), the internal mutex (`tq_mutex`), the enqueue callback (`tq_enqueue`), the worker threads (`tq_threads`), and flags.

TQ_LOCK / TQ_UNLOCK macros. Just after the structure. Acquire the mutex (spin or sleep, depending on `tq_spin`).

taskqueue_create and _taskqueue_create. Allocate the structure, initialise the mutex (`MTX_DEF` or `MTX_SPIN`), return.

taskqueue_enqueue and taskqueue_enqueue_flags. Acquire the mutex, check `task->ta_pending`, coalesce or link, wake the worker (via the enqueue callback), release the mutex.

taskqueue_enqueue_timeout. Schedule the internal callout; the callout's callback will later call `taskqueue_enqueue` on the underlying task.

taskqueue_cancel and taskqueue_cancel_timeout. Remove from queue if pending; return `EBUSY` if running.

taskqueue_drain and variants. `msleep` on a condition that is set when the task is idle and not pending.

taskqueue_run_locked. The heart of the subsystem. In a loop: take a task from the pending queue, record `ta_pending`, clear it, move to active, drop the mutex, optionally enter the net epoch, call the callback, re-acquire the mutex, signal any drainers. Loop until the queue is empty.

taskqueue_thread_loop. The worker thread's main loop. Acquire the taskqueue mutex, wait for work (`msleep`) if the queue is empty, call `taskqueue_run_locked` when work arrives, loop.

taskqueue_free. Set the "draining" flag, wake every worker, wait for every worker to exit, drain any remaining tasks, free the structure.

This reading cites each function by name rather than by line number, because line numbers drift between FreeBSD releases while symbol names survive. If you want approximate coordinates for `subr_taskqueue.c` in FreeBSD 14.3, the main entry points live near these lines: `_taskqueue_create` 141, `taskqueue_create` 178, `taskqueue_free` 217, `taskqueue_enqueue_flags` 305, `taskqueue_enqueue` 317, `taskqueue_enqueue_timeout` 382, `taskqueue_run_locked` 485, `taskqueue_cancel` 579, `taskqueue_cancel_timeout` 591, `taskqueue_drain` 612, `taskqueue_thread_loop` 820. Treat those numbers as a scroll hint; open the file and jump to the symbol.

Reading these functions once is a good investment. Everything Chapter 14 taught about the API's behaviour is visible in the implementation.

14.27 A Final Tour: Five Common Shapes

Five shapes that account for most taskqueue usage in the FreeBSD tree. Recognising them turns reading driver source from parsing to pattern-matching.

14.27.1 Shape A: The Solo Task

One task, enqueued from one place, drained at detach. Simplest. Used by drivers that need to defer exactly one kind of work.

```
TASK_INIT(&sc->task, 0, sc_task, sc);
/* ... */
taskqueue_enqueue(sc->tq, &sc->task);
/* ... */
taskqueue_drain(sc->tq, &sc->task);
```

14.27.2 Shape B: The Filter-Plus-Task Split

Interrupt filter does the minimum, enqueues a task for the rest.

```
static int
sc_filter(void *arg)
{
    struct sc *sc = arg;
    taskqueue_enqueue(sc->tq, &sc->intr_task);
    return (FILTER_HANDLED);
}
```

14.27.3 Shape C: The Callout-Driven Periodic Task

Callout fires periodically, enqueues a task that does the work.

```
static void
sc_periodic_callout(void *arg)
{
    struct sc *sc = arg;
    taskqueue_enqueue(sc->tq, &sc->periodic_task);
    callout_reset(&sc->co, hz, sc_periodic_callout, sc);
}
```

14.27.4 Shape D: The Timeout Task

timeout_task for delayed work in thread context.

```
TIMEOUT_TASK_INIT(sc->tq, &sc->delayed, 0, sc_delayed, sc);
/* ... */
taskqueue_enqueue_timeout(sc->tq, &sc->delayed, delay_ticks);
/* ... */
taskqueue_drain_timeout(sc->tq, &sc->delayed);
```


14.27.5 Shape E: The Self-Re-Enqueueing Task

A task that schedules itself again from its own callback.

```
static void
sc_self(void *arg, int pending)
{
    struct sc *sc = arg;
    /* work */
    if (sc->keep_running)
        taskqueue_enqueue_timeout(sc->tq, &sc->self_tt, hz);
}
```

Every driver you read will use some combination of these five shapes. Once they are familiar, the rest is implementation detail.

14.28 Closing Summary: What The Chapter Delivered

A short inventory, for a reader who wants the compressed version after working through the full chapter.

Concepts introduced.

- Deferred work as the bridge from edge contexts to thread context.
- The `struct task` / `struct timeout_task` data structures and their lifecycle.
- Taskqueues as the queue + worker-thread pair.
- Private versus predefined taskqueues and when each is right.
- Coalescing via `ta_pending` and the `pending` argument.
- Priority ordering within a queue.
- The `block/unblock/quiesce/drain_all` primitives.
- Detach ordering with callouts, tasks, `selinfo`, and taskqueue teardown.
- Debugging via `procstat`, `dtrace`, `sysctl` counters, and `WITNESS`.
- Introductory exposure to `epoch(9)`, `grouptaskqueues`, and per-CPU taskqueues.

Driver changes.

- Stage 1: one task enqueued from `tick_source`, drained at detach.
- Stage 2: a private taskqueue owned by the driver.
- Stage 3: a bulk-writer task demonstrating deliberate coalescing, a `timeout_task` for delayed reset.
- Stage 4: consolidation, version bump to 0.8- taskqueues, full regression.

Documentation changes.

- A Tasks section in `LOCKING.md`.
- A detach-ordering section that enumerates every drain step.
- Per-task documentation listing callback, lifetime, enqueue paths, and cancellation paths.

Patterns catalogued.

- Deferred logging.
- Delayed reset.
- Filter-plus-task interrupt split.
- Async copyin/copyout.
- Retry-with-backoff.
- Deferred teardown.
- Statistics rollover.
- Block-during-reconfiguration.
- Drain-all at subsystem boundary.
- Synthetic event generation.

Debugging tools used.

- `procstat -t` for taskqueue thread state.
- `ps ax` for kernel thread inventory.
- `sysctl dev.<driver>` for driver-exposed counters.
- `dtrace` for enqueue latency and callback duration.
- `procstat -kk` for stuck-thread diagnosis.
- WITNESS and INVARIANTS as the debug-kernel safety net.

Deliverables.

- `content/chapters/part3/chapter-14.md` (this file).
- `examples/part-03/ch14-taskqueues-and-deferred-work/stage1-first-task/`.
- `examples/part-03/ch14-taskqueues-and-deferred-work/stage2-private-taskqueue/`.
- `examples/part-03/ch14-taskqueues-and-deferred-work/stage3-coalescing/`.
- `examples/part-03/ch14-taskqueues-and-deferred-work/stage4-final/`.
- `examples/part-03/ch14-taskqueues-and-deferred-work/labs/` with `poll_waiter.c` and small helper scripts.
- `examples/part-03/ch14-taskqueues-and-deferred-work/LOCKING.md` with the Tasks section.
- `examples/part-03/ch14-taskqueues-and-deferred-work/README.md` with per-stage build and test instructions.

That is the end of Chapter 14. Chapter 15 continues the synchronisation story.

14.29 Reference: Reading `taskqueue_run_locked` Line By Line

The heart of the taskqueue subsystem is a short loop inside `taskqueue_run_locked` in `/usr/src/sys/kern/-subr_taskqueue.c`. Reading it once, slowly, pays off whenever you need to reason about the subsystem's behaviour. A narrated pass follows.

The function is called from the worker thread's main loop, `taskqueue_thread_loop`, with the taskqueue mutex held. Its job is to process every pending task, release the mutex around the callback, and return with the mutex still held when the queue is empty.

```
static void
taskqueue_run_locked(struct taskqueue *queue)
{
    struct epoch_tracker et;
    struct taskqueue_busy tb;
    struct task *task;
    bool in_net_epoch;
    int pending;

    KASSERT(queue != NULL, ("tq is NULL"));
    TQ_ASSERT_LOCKED(queue);
    tb.tb_running = NULL;
    LIST_INSERT_HEAD(&queue->tq_active, &tb, tb_link);
    in_net_epoch = false;

```

The function starts by asserting the mutex is held and inserting a local `taskqueue_busy` structure into the active list. The `tb` structure represents this invocation of `taskqueue_run_locked`; later code uses it to track what this invocation is currently running. The `in_net_epoch` flag tracks whether we are currently inside the network epoch, so we do not enter it redundantly when consecutive tasks are all network-flagged.

```
while ((task = STAILQ_FIRST(&queue->tq_queue)) != NULL) {
    STAILQ_REMOVE_HEAD(&queue->tq_queue, ta_link);
    if (queue->tq_hint == task)
        queue->tq_hint = NULL;
    pending = task->ta_pending;
    task->ta_pending = 0;
    tb.tb_running = task;
    tb.tb_seq = ++queue->tq_seq;
    tb.tb_canceling = false;
    TQ_UNLOCK(queue);

```

The main loop. Pull the head off the pending queue. Snapshot the pending count into a local variable, reset the field to zero (so new enqueues that arrive during the callback increment from zero again). Record the task in the `tb` structure so drain callers can see what is running. Increment a sequence counter for stale-drain detection. Drop the mutex.

Notice that between here and the next `TQ_LOCK`, the mutex is not held. This is the window where the callback runs; the rest of the kernel can enqueue more tasks (which will coalesce or queue up), drain other tasks (which will see `tb.tb_running == task` and wait), or run its own business.

```
KASSERT(task->ta_func != NULL, ("task->ta_func is NULL"));
if (!in_net_epoch && TASK_IS_NET(task)) {
    in_net_epoch = true;
    NET_EPOCH_ENTER(et);

```

```

        } else if (in_net_epoch && !TASK_IS_NET(task)) {
            NET_EPOCH_EXIT(et);
            in_net_epoch = false;
        }
        task->ta_func(task->ta_context, pending);

        TQ_LOCK(queue);
        wakeup(task);
    }
    if (in_net_epoch)
        NET_EPOCH_EXIT(et);
    LIST_REMOVE(&tb, tb_link);
}

```

The epoch bookkeeping: enter the net epoch if this task is net-flagged and we are not already in it; exit the epoch if we entered it for an earlier task but this one is not net-flagged. This lets consecutive net tasks share a single epoch entry, which is an optimisation the framework does for free.

Call the callback with the context and pending count. Re-acquire the mutex. Wake up any drain caller who is waiting for this specific task. Loop.

After the loop, if we are still in the net epoch, exit it. Remove the tb structure from the active list.

Seven observations from reading this function.

Observation 1. The mutex is dropped for exactly as long as the callback runs. No taskqueue-internal code runs with the callback; if the callback takes milliseconds, the taskqueue mutex is free for milliseconds.

Observation 2. `ta_pending` is reset before the callback runs, not after. A new enqueue during the callback makes the task pending again (`pending=1`). After the callback returns, the loop sees the new pending, pulls it off, and runs the callback a second time with `pending=1`. No enqueue is lost.

Observation 3. The pending value passed to the callback is the count at the moment the task was pulled off the queue, not at the moment the enqueue calls happened. If enqueues arrive during the callback, they do not contribute to this invocation's pending; they contribute to the next invocation's pending.

Observation 4. The wakeup at the bottom of the loop wakes drain callers sleeping on the task's address. Drain uses `msleep(&task, &tq->mutex, ...)` and waits for the task to be off the queue and not currently running. The wakeup here is what makes the drain terminate.

Observation 5. The sequence counter `tq_seq` and the `tb.tb_seq` allow drain-all to detect whether new tasks have been added since the drain started. Without the sequence, drain-all would race with new enqueues.

Observation 6. `tb.tb_canceling` is a flag that `taskqueue_cancel` sets to tell a waiter "this task is currently being cancelled"; its purpose is to let concurrent cancel/drain calls coordinate. We have not discussed it in the main text because most drivers never see it.

Observation 7. Multiple worker threads can each be inside `taskqueue_run_locked` simultaneously, each dispatching a different task. The `tq_active` list holds all their `tb` structures. Different tasks on the same queue run in parallel; the same task cannot run in parallel with itself, because only one worker pulls it off at a time.

Those observations together describe exactly what the taskqueue guarantees and what it does not. Every behaviour the chapter described earlier is a consequence of this short loop.

14.30 Reference: A Walkthrough of `taskqueue_drain`

Equally illuminating, equally short. From `/usr/src/sys/kern/subr_taskqueue.c`, roughly:

```
void
taskqueue_drain(struct taskqueue *queue, struct task *task)
{
    if (!queue->tq_spin)
        WITNESS_WARN(WARN_GIANTOK | WARN_SLEEPOK, NULL, ...);

    TQ_LOCK(queue);
    while (task->ta_pending != 0 || task_is_running(queue, task))
        TQ_SLEEP(queue, task, "taskqueue_drain");
    TQ_UNLOCK(queue);
}
```

The function acquires the taskqueue mutex, then loops until the task is neither pending nor running. Each iteration sleeps on the task's address; each wakeup at the bottom of `taskqueue_run_locked` wakes the drainer to re-check.

`task_is_running(queue, task)` walks the active list (`tq_active`) and returns true if any `tb.tb_running == task`. It is $O(N)$ in the number of workers, but for most drivers N is 1 and this is $O(1)$.

The function does not hold the lock across the sleep; `TQ_SLEEP` (which expands to `msleep` or `msleep_spin`) drops the mutex during the sleep and re-acquires it on wakeup, which is the standard condition-variable pattern.

Observations from reading `taskqueue_drain`.

Observation 1. Drain is a condition-variable wait, using the task pointer as the wakeup channel. The wake comes from the `wakeup(task)` at the bottom of `taskqueue_run_locked`.

Observation 2. Drain does not stop new enqueues from happening. If the task is enqueued again while the drain is waiting, the drain will continue waiting until that new enqueue fires and completes. This is why detach discipline requires draining every producer (callouts, other tasks, interrupt handlers) before draining the victim task.

Observation 3. Drain on an idle task (never enqueued, or enqueued and completed) returns immediately. It is safe to call drain unconditionally in detach.

Observation 4. Drain holds the taskqueue mutex across the initial check and before the sleep, which means drain cannot race with enqueue in a way that misses a newly pending task. If an enqueue arrives between the check and the sleep, `ta_pending` becomes non-zero and the drain loop re-iterates.

Observation 5. The `WITNESS_WARN` at the top asserts that the caller is in a context where sleeping is legal. If you try to call `taskqueue_drain` from a context that cannot sleep (a callout callback, for example), `WITNESS` complains.

Two complementary functions are `taskqueue_cancel` (which removes the task from the queue if it is pending and returns `EBUSY` if it is running) and `taskqueue_drain_timeout` (which also cancels the embedded callout). Reading their implementations once is worthwhile; they are short.

14.31 Reference: The Lifecycle As Seen From the Softc

One more view, for completeness. The same information, organised around the `softc` instead of around the API.

At **attach time**, the `softc` gains:

- A taskqueue pointer (`sc->tq`), created by `taskqueue_create` and populated by `taskqueue_start_threads`.
- One or more task structures (`sc->foo_task`), initialised by `TASK_INIT` or `TIMEOUT_TASK_INIT`.
- Counters and flags for observability (optional but recommended).

At **runtime**, the `softc`'s taskqueue state is:

- `sc->tq` is an opaque pointer; drivers never read its fields.
- `sc->foo_task` may be `IDLE`, `PENDING`, or `RUNNING` at any instant.
- The taskqueue's worker threads sleep most of the time, wake on enqueue, run callbacks, sleep again.

At **detach time**, the `softc` is torn down in this order:

1. Clear `sc->is_attached` under the mutex, broadcast cvs, release mutex.
2. Drain every callout.
3. Drain every task.
4. Drain selinfo.
5. Free the taskqueue.
6. Destroy the `cdev` and its alias.
7. Free the `sysctl` context.
8. Destroy the `cbuf` and counters.
9. Destroy the `cvs`, `sx`, mutex.

After `taskqueue_free`, `sc->tq` becomes invalid. After the task drains, the `sc->foo_task` structures are idle and their storage can be reclaimed along with the `softc`.

The `softc`'s lifetime is determined by the device's attach/detach. Tasks cannot outlive their `softc`. The drain at detach is what guarantees that property.

14.32 Reference: A Glossary of Terms

For quick lookup.

Task. An instance of `struct task`; a callback-plus-context bundled for enqueue onto a taskqueue.

Taskqueue. An instance of `struct taskqueue`; a queue of pending tasks paired with one or more worker threads.

Timeout task. An instance of `struct timeout_task`; a task plus an internal callout, used for scheduled-in-the-future work.

Enqueue. To add a task to a taskqueue. If the task is already pending, increment its pending count instead.

Drain. To wait until a task is neither pending nor running.

Dispatch. The act of the taskqueue worker taking a task off the pending list and running its callback.

Coalesce. To fold redundant enqueues into a single pending-state increment rather than two list entries.

Pending count. The value of `ta_pending`, representing how many coalesced enqueues are accumulated on this task.

Idle task. A task that is not pending and not running. `ta_pending == 0` and no worker has it.

Worker thread. A kernel thread (usually one per taskqueue) whose job is to wait for work and run task callbacks.

Edge context. A constrained context (callout, interrupt filter, epoch section) where some operations are not permitted.

Thread context. An ordinary kernel thread context where sleeping, sleepable-lock acquisition, and all standard operations are permitted.

Detach ordering. The sequence in which primitives are drained and freed at device detach, such that no primitive is freed while something can still reference it.

Drain race. A bug where a primitive is freed while a callback or handler is still potentially running, caused by incorrect detach ordering.

Pending-drop counter. A diagnostic counter incremented when the callback's pending argument is greater than one, indicating coalescing occurred.

Private taskqueue. A taskqueue owned by the driver, created and freed with attach/detach, not shared with other drivers.

Shared taskqueue. A kernel-provided taskqueue (`taskqueue_thread`, `taskqueue_swi`, etc.) used by multiple drivers simultaneously.

Fast taskqueue. A taskqueue created with `taskqueue_create_fast` that uses a spin mutex internally, safe for enqueue from filter-interrupt context.

Grouptaskqueue. A scalable variation where tasks are distributed across per-CPU queues. Used by high-rate network drivers.

Epoch. A lockless read-synchronisation mechanism. The `net_epoch_preempt` epoch protects network state.

Chapter 14 ends here. The next chapter takes the synchronisation story further.

15 More Synchronization: Conditions, Semaphores, and Coordination

15.1 Reader Guidance & Outcomes

At the end of Chapter 14 your `myfirst` driver reached a qualitatively different state from where it started Part 3. It has a documented data-path mutex, two condition variables, a configuration `sx` lock, three callouts, a private taskqueue with three tasks, and a detach path that drains every primitive in the correct order. The driver is, for the first time in the book, more than a collection of handlers. It is a composition of synchronisation primitives that cooperate to deliver bounded, safe behaviour under load.

Chapter 15 is about pushing that composition further. Most real drivers eventually discover that mutexes and basic condition variables, even combined with `sx` locks and taskqueues, are not always the primitives that make a particular problem easy to express. A driver might need to cap the number of concurrent writers, enforce a bounded pool of reusable hardware slots, coordinate a handshake between a callout and a task, let a slow read wake up for a signal without losing the partial progress it made, or surface shutdown state across several subsystems in a way that every piece of code can check cheaply. Each of those shapes is solvable with what you already know, but each has a primitive or idiom that makes the solution direct and the code readable. This chapter teaches those primitives and idioms one at a time, applies them to the driver, and ties the result together with a small wrapper layer that turns scattered calls into a named vocabulary.

The chapter is also the last chapter of Part 3. After Chapter 15, Part 4 begins and the book turns to hardware. Every primitive Part 3 taught you, from the first `mtx_init` in Chapter 11 to the last `taskqueue_drain` in Chapter 14, stays with you into Part 4. The coordination patterns in this chapter are not a bonus topic. They are the closing piece of the synchronisation toolbox your driver carries forward into the hardware-facing chapters.

15.1.1 Why This Chapter Earns Its Own Place

You could skip this chapter. The driver as it stands at the end of Chapter 14 is functional, tested, and technically correct. Its mutex-and-cv discipline is sound. Its detach ordering works. Its taskqueue is clean.

What the driver lacks, and what Chapter 15 adds, is a small set of sharper tools for specific shapes of coordination that mutexes and basic condition variables express awkwardly. A counting semaphore is a few lines of code that says “at most *N* participants at once”; expressing the same invariant with a mutex and a counter and a cv requires more lines and hides the intent. A refined `sx` pattern with `sx_try_upgrade` lets a read path occasionally promote to a writer without releasing its slot and racing with other would-be writers; without the primitive, you write awkward retry loops. A proper `cv_timedwait_sig` usage distinguishes between `EINTR` and `ERESTART` and between “the caller was interrupted” and “the deadline fired”; a naive wait leaves the caller hanging or abandons partial work on any signal.

The payoff of learning these tools is not just that the current chapter's refactor will be cleaner. It is that when you read a production FreeBSD driver a year from now, you will recognise these shapes immediately. When `/usr/src/sys/dev/hyperv/storvsc/hv_storvsc_drv_freebsd.c` calls `sema_wait` on a per-request semaphore to block until hardware completion, you will know what the author was thinking. When a network driver reaches for `sx_try_upgrade` in a statistics-update path, you will know why that was the correct call. Without Chapter 15 those calls are opaque. With Chapter 15 they are obvious.

The other payoff is maintainability. A driver that scatters its synchronisation vocabulary across a hundred places is hard to change. A driver that encapsulates its synchronisation in a small named layer (even just a set of inline functions in a header) is easy to change. Section 6 walks through the encapsulation explicitly; by the end of the chapter, your driver will have a small `myfirst_sync.h` that names every coordination primitive it uses. Adding a new synchronised state later becomes an exercise in extending the header, not in spreading new `mtx_lock/mtx_unlock` calls across the file.

15.1.2 Where Chapter 14 Left the Driver

A few prerequisites to verify before starting. Chapter 15 extends the driver produced at the end of Chapter 14 Stage 4 (version 0.8-taskqueues). If any of the items below feels uncertain, return to Chapter 14 before starting this chapter.

- Your `myfirst` driver compiles cleanly and identifies itself as version 0.8-taskqueues.
- It uses `MYFIRST_LOCK/MYFIRST_UNLOCK` macros around `sc->mtx` (the data-path mutex).
- It uses `MYFIRST_CFG_SLOCK/MYFIRST_CFG_XLOCK` around `sc->cfg_sx` (the configuration `sx`).
- It uses two named condition variables (`sc->data_cv`, `sc->room_cv`) for blocking reads and writes.
- It supports timed reads via `cv_timedwait_sig` and the `read_timeout_ms` sysctl.
- It has three callouts (`heartbeat_co`, `watchdog_co`, `tick_source_co`) with their interval sysctls.
- It has a private taskqueue (`sc->tq`) with three tasks (`selwake_task`, `bulk_writer_task`, `reset_delayed_task`).
- The lock order `sc->mtx -> sc->cfg_sx` is documented in `LOCKING.md` and enforced by `WITNESS`.
- `INVARIANTS`, `WITNESS`, `WITNESS_SKIPSPIN`, `DDB`, `KDB`, and `KDB_UNATTENDED` are enabled in your test kernel; you have built and booted it.
- The Chapter 14 stress kit runs cleanly under the debug kernel.

That driver is what we extend in Chapter 15. The additions are modest in volume but substantial in what they enable. The driver's data path is unchanged at the mechanical level; what changes is the vocabulary it uses to talk about concurrency.

15.1.3 What You Will Learn

After finishing this chapter you will be able to:

- Recognise when a mutex plus a condition variable is not the right primitive for a particular invariant, and name the alternative (a semaphore, an `sx` upgrade pattern, an atomic flag with a memory barrier, a per-cpu counter, or an encapsulated coordinator function).
- Explain what a counting semaphore is, how it differs from a mutex and from a binary semaphore, and why FreeBSD's `sema(9)` API is specifically a counting-semaphore API with no ownership concept.

- Use `sema_init`, `sema_wait`, `sema_post`, `sema_trywait`, `sema_timedwait`, `sema_value`, and `sema_destroy` correctly, including the lifetime contract that no waiters may be present when `sema_destroy` is called.
- Describe the known limitations of FreeBSD's kernel semaphore: no priority inheritance, no signal-interruptible wait, and the guidance in `/usr/src/sys/kern/kern_sema.c` about why they are not a general replacement for a mutex plus a condition variable.
- Refine the driver's `sx` usage with `sx_try_upgrade`, `sx_downgrade`, `sx_xlocked`, and `sx_slock` patterns that express read-mostly workloads cleanly.
- Distinguish `cv_wait`, `cv_wait_sig`, `cv_timedwait`, and `cv_timedwait_sig`, and know what each returns on timeout, signal, and normal wakeup.
- Handle the `EINTR` and `ERESTART` return values from signal-interruptible waits correctly, so that `read(2)` and `write(2)` on the driver respond sensibly to `SIGINT` and friends.
- Build cross-component handshakes between a callout, a task, and a user thread using a small state flag protected by the driver mutex.
- Introduce a `myfirst_sync.h` header that names every synchronisation primitive the driver uses, so future contributors can change the locking strategy in one place.
- Use the `atomic(9)` API correctly for small lock-free coordination steps, especially shutdown flags that need to be visible across contexts without a lock.
- Write stress tests that deliberately trigger race conditions in the driver's synchronisation and confirm the primitives handle them.
- Refactor the driver to version 0.9-coordination and update `LOCKING.md` with a Semaphores section and a Coordination section.

That is a long list. Each item in it is small; the value of the chapter is in the composition.

15.1.4 What This Chapter Does Not Cover

Several adjacent topics are explicitly deferred so Chapter 15 stays focused.

- **Hardware interrupt handlers and the full split between `FILTER` and `IThread` execution contexts.** Part 4 introduces `bus_setup_intr(9)` and the actual interrupt story. Chapter 15 mentions interrupt-adjacent contexts only when they illustrate a synchronisation pattern you might reuse.
- **Lock-free data structures at scale.** The `atomic(9)` family covers small coordination flags; it does not cover SMR, hazard pointers, RCU analogues, or full lock-free queues. Chapter 15 touches on atomic operations and epoch briefly; the deeper lock-free story belongs to a specialised kernel-internals discussion.
- **Detailed scheduler tuning.** Thread priorities, RT classes, priority inheritance, CPU affinity: out of scope. We pick sensible defaults and move on.
- **Userland POSIX semaphores and SysV IPC.** `sema(9)` in the kernel is a different beast. Chapter 15 focuses on the kernel primitive.
- **Performance micro-benchmarks.** Lockstat and DTrace lock profiling get a mention, not a full treatment. A dedicated performance chapter later in the book, when it exists, will carry that load.
- **Cross-process coordination primitives.** Some drivers need to coordinate with userland helpers; that problem is fundamentally different and belongs to a later chapter about ioctl-based protocols.

Staying inside those lines keeps the chapter's mental model coherent. Chapter 15 adds the coordination toolkit; Part 4 and later chapters apply the toolkit to hardware-facing scenarios.

15.1.5 Estimated Time Investment

- **Reading only:** about three to four hours. The API surface is small but the composition takes some thought.
- **Reading plus typing the worked examples:** seven to nine hours over two sessions. The driver evolves in four stages.
- **Reading plus all labs and challenges:** twelve to sixteen hours over three or four sessions, including time to run stress tests against race-prone code paths.

If you find Section 5 (cross-subsystem coordination) disorienting on first pass, that is normal. The material is conceptually simple but requires holding several parts of the driver in your head at once. Stop, re-read the worked handshake in Section 5, and continue when the diagram has settled.

15.1.6 Prerequisites

Before starting this chapter, confirm:

- Your driver source matches Chapter 14 Stage 4 (`stage4-final`). The starting point assumes every Chapter 14 primitive, every Chapter 13 callout, every Chapter 12 `cv` and `sx`, and the Chapter 11 concurrent-IO model.
- Your lab machine runs FreeBSD 14.3 with `/usr/src` on disk and matching the running kernel.
- A debug kernel with `INVARIANTS`, `WITNESS`, `WITNESS_SKIPSPIN`, `DDB`, `KDB`, and `KDB_UNATTENDED` is built, installed, and booting cleanly.
- You understand the Chapter 14 detach ordering well enough to extend it without getting lost.
- You have a comfortable mental model of `cv_wait_sig` and `cv_timedwait_sig` from Chapters 12 and 13.

If any of the above is shaky, fix it now rather than pushing through Chapter 15 and trying to reason from a moving foundation. The Chapter 15 primitives are sharper than what came before, and they amplify whatever discipline (or lack of it) the driver already has.

15.1.7 How to Get the Most Out of This Chapter

Three habits will pay off quickly.

First, keep `/usr/src/sys/kern/kern_sema.c` and `/usr/src/sys/sys/sema.h` bookmarked. The implementation is short, under two hundred lines, and it is the shortest path to understanding what a FreeBSD semaphore actually does. Read `_sema_wait`, `_sema_post`, and `_sema_timedwait` carefully once. Knowing that a semaphore is “a counter plus a mutex plus a `cv`, wrapped in an API” makes the rest of the chapter feel obvious.

A note on line numbers. Every pointer into the source in this chapter hangs on a function, macro, or structure name, not on a numeric line. `sema_init` and `_sema_wait` in `kern_sema.c`, and `sx_try_upgrade` in `/usr/src/sys/kern/kern_sx.c`, will remain findable by those names across FreeBSD 14.x point releases; the line each one occupies can drift as the surrounding code is revised. When in doubt, `grep` for the symbol.

Second, compare every new primitive to what you would have written with the old ones. The exercise “if I did not have `sema(9)`, how would I express this?” is instructive. Writing the alternative with `mtx` and `cv` is usually possible, but the semaphore version is often half as long and substantially clearer. Seeing the contrast is how the value of the primitive becomes concrete.

Third, type the changes by hand and run each stage under `WITNESS`. Advanced synchronisation bugs are almost always detected by `WITNESS` on first contact if you run a debug kernel; they are almost always silent on a production kernel until the first crash. The companion source under `examples/part-03/ch15-more-synchronization/` is the reference version, but the muscle memory of typing `sema_init(&sc->writers_sema, 4, "myfirst writers")` once is worth more than reading it ten times.

15.1.8 Roadmap Through the Chapter

The sections in order are:

1. **When mutexes and condition variables are not enough.** A survey of the shapes of problem that benefit from a different primitive.
2. **Using semaphores in the FreeBSD kernel.** The `sema(9)` API in depth, with the writer-cap refactor as Stage 1 of the Chapter 15 driver.
3. **Read-mostly scenarios and shared access.** Refined `sx(9)` patterns, including `sx_try_upgrade` and `sx_downgrade`, with a small statistics-cache refactor as Stage 2.
4. **Condition variables with timeout and interruption.** A careful treatment of `cv_timedwait_sig`, `EINTR` vs `ERESTART`, partial-progress handling, and the `sysctl` tuning that lets readers observe the behaviour. Stage 3 of the driver.
5. **Synchronisation between modules or subsystems.** Handshakes between callouts, tasks, and user threads via small state flags. Atomic operations and memory ordering at an introductory level. Stage 4 begins here.
6. **Synchronisation and modular design.** The `myfirst_sync.h` header, the naming discipline, and how the driver changes shape when synchronisation is encapsulated.
7. **Testing advanced synchronisation.** Stress kits, fault injection, and the observability `sysctls` that let you see the primitives working.
8. **Refactoring and versioning.** Stage 4 complete, version bump to 0.9-coordination, `LOCKING.md` extended, and the Part 3 closing regression pass.

After the eight sections come hands-on labs, challenge exercises, a troubleshooting reference, a Wrapping Up that closes Part 3, and a bridge to Chapter 16 that opens Part 4. The same reference-and-cheat-sheet material that Chapters 13 and 14 ended with returns here at the end.

If this is your first pass, read linearly and do the labs in order. If you are revisiting, Sections 5 and 6 stand alone and make good single-sitting reads.

15.2 Section 1: When Mutexes and Condition Variables Are Not Enough

The Chapter 11 mutex and the Chapter 12 condition variable are the default primitives of FreeBSD driver synchronisation. Nearly every driver uses them. Many drivers use nothing else. For a large class of problems the combination is exactly right: a mutex protects shared state, a condition variable lets a waiter sleep until the

state matches a predicate, and the two together express “wait for state to become acceptable” and “tell others that state has changed” cleanly.

This section is about the problems where that default is awkward. Not because the mutex-and-cv combination cannot express the invariant, but because it expresses it more verbosely and more error-prone than a different primitive. Recognising those shapes is the first step toward using the right tool.

15.2.1 The Shape of the Mismatch

Every synchronisation primitive has an underlying model of what it protects. A mutex protects mutual exclusion: at most one thread executes inside the lock at a time. A condition variable protects a predicate: a waiter sleeps until the predicate becomes true, and the signaller asserts that the predicate has changed. The two compose because the condition variable’s wait drops and reacquires the mutex automatically, which lets the waiter observe the predicate under the lock, release the lock during the sleep, and regain the lock on wakeup.

The mismatch appears when the invariant you are protecting is not best described as “at most one” or “wait for a predicate”. A handful of common shapes recur.

Bounded admission. The invariant is “at most N of a thing at once”. For N equal to one, a mutex is natural. For N greater than one, the mutex-plus-cv-plus-counter version requires you to write an explicit counter, test it under the mutex, sleep on a cv if the counter is at N, decrement on entry, re-signal on exit, and rediscover the right wakeup policy. The semaphore primitive expresses the same invariant in three calls: `sema_init(&s, N, ...)`, `sema_wait(&s)` at entry, `sema_post(&s)` at exit.

Read-mostly state with occasional promotion. The invariant is “many readers concurrently, or one writer; when a reader detects a need to write, promote”. The `sx(9)` lock handles the many-readers-or-one-writer part natively. The promotion part (`sx_try_upgrade`) is a primitive that a mutex-plus-cv version has to simulate with a rwlock-like counter and retry logic.

Predicate that must survive signal interruption with partial-progress preservation. A `read(2)` that copied half its requested bytes and is now sleeping for more should, on a signal, return the copied bytes rather than `EINTR`. `cv_timedwait_sig` gives you the `EINTR` and `ERESTART` distinction; writing the equivalent with `raw_cv_wait` plus a periodic signal check is possible but error-prone.

Cross-component shutdown coordination. Several parts of the driver (callouts, tasks, user threads) need to observe “the driver is shutting down” consistently. A mutex-protected flag is an option. An atomic flag with a seq-cst fence on the writer and acquire loads on readers is often cheaper and clearer for this specific pattern, and the chapter will show when to pick which.

Rate-limited retries. “Do this at most once per 100 ms, skip if already in progress.” Expressible with a mutex and a timer, but a taskqueue plus a timeout task plus an atomic test-and-set on an “already scheduled” flag is often cleaner. This pattern came up at the end of Chapter 14; Chapter 15 refines it.

For each shape, Chapter 15 picks a primitive that fits and shows the refactor side by side. The purpose is not to argue that the semaphore or the `sx` upgrade or the atomic flag is “better”. The purpose is to let you pick the tool that matches the problem, so your driver reads cleanly to the next person who opens it.

15.2.2 A Concrete Motivating Example: Too Many Writers

A motivating example to make the mismatch concrete. Suppose the driver wants to limit the number of concurrent writers. “Concurrent writers” means user threads that are simultaneously inside the `myfirst_write` handler past the initial validation. The limit is a small integer, say four, exposed as a `sysctl` tuning knob.

The mutex-plus-counter version looks like this:

```
/* In the softc: */
int writers_active;
int writers_limit; /* Configurable via sysctl. */
struct cv writer_cv;

/* In myfirst_write, at entry: */
MYFIRST_LOCK(sc);
while (sc->writers_active >= sc->writers_limit) {
    int error = cv_wait_sig(&sc->writer_cv, &sc->mtx);
    if (error != 0) {
        MYFIRST_UNLOCK(sc);
        return (error);
    }
    if (!sc->is_attached) {
        MYFIRST_UNLOCK(sc);
        return (ENXIO);
    }
}
sc->writers_active++;
MYFIRST_UNLOCK(sc);

/* At exit: */
MYFIRST_LOCK(sc);
sc->writers_active--;
cv_signal(&sc->writer_cv);
MYFIRST_UNLOCK(sc);
```

Every line is necessary. The loop handles spurious wakeups and signal returns. The signal check preserves partial progress (if any). The `is_attached` check ensures we do not proceed after detach. The `cv_signal` wakes the next waiter. The caller must remember to decrement.

The semaphore version looks like this:

```
/* In the softc: */
struct sema writers_sema;

/* In attach: */
sema_init(&sc->writers_sema, 4, "myfirst writers");
```

```
/* In destroy: */
sema_destroy(&sc->writers_sema);

/* In myfirst_write, at entry: */
sema_wait(&sc->writers_sema);
if (!sc->is_attached) {
    sema_post(&sc->writers_sema);
    return (ENXIO);
}

/* At exit: */
sema_post(&sc->writers_sema);
```

Five lines of runtime logic, including the attached check. The primitive expresses the invariant directly. A reader who sees `sema_wait(&sc->writers_sema)` understands the intent in one glance.

Note what the semaphore version gives up. `sema_wait` is not signal-interruptible (as we will see in Section 2, FreeBSD's `sema_wait` uses `cv_wait` internally, not `cv_wait_sig`). If you need interruptibility, you fall back to the mutex-plus-cv version or combine `sema_trywait` with a separate interruptible wait. Every primitive has its trade-offs; Section 2 names them.

The broader point is that neither version is “wrong”. The mutex-plus-counter version is correct and has been used in drivers for decades. The semaphore version is correct and clearer for this specific invariant. Knowing both lets you pick the right one for the specific constraints at hand.

15.2.3 The Rest of the Section Previews the Chapter

Section 1 is deliberately short. The rest of the chapter unfolds each shape in its own section with its own refactor of the `myfirst` driver:

- Section 2 does the writer-cap semaphore refactor as Stage 1.
- Section 3 does the read-mostly `sx` refinement as Stage 2.
- Section 4 does the interruptible-wait refinement as Stage 3.
- Section 5 does the cross-component handshake as part of Stage 4.
- Section 6 extracts the synchronisation vocabulary into `myfirst_sync.h`.
- Section 7 writes the stress tests.
- Section 8 ties it all together and ships `0.9-coordination`.

Before diving in, one general observation. The Chapter 15 changes are small in lines of code. The whole chapter probably adds fewer than two hundred lines to the driver. What it adds in mental model is larger. Each primitive we introduce expresses an invariant that was implicit in the Chapter 14 driver; making it explicit is most of the value.

15.2.4 Wrapping Up Section 1

A mutex and a condition variable cover most driver synchronisation. When the invariant is “at most N”, “many readers or one writer with occasional promotion”, “interruptible wait with partial progress”, “cross-component shutdown”, or “rate-limited retries”, a different primitive expresses the intent more directly and leaves less room for bugs. Section 2 introduces the first of those primitives, the counting semaphore.

15.3 Section 2: Using Semaphores in the FreeBSD Kernel

A counting semaphore is a small primitive. Internally it is a counter, a mutex, and a condition variable; the API wraps those three into operations that expose the counter-and-wait-for-positive semantics as its main interface. FreeBSD’s kernel semaphore lives in `/usr/src/sys/sys/semaphore.h` and `/usr/src/sys/kern/kern_semaphore.c`. The whole implementation is under two hundred lines. Reading it once is the fastest way to understand what the API guarantees.

This section covers the API in depth, compares semaphores with mutexes and condition variables, walks through the writer-cap refactor as Stage 1 of the Chapter 15 driver, and names the trade-offs that come with the primitive.

15.3.1 The Counting Semaphore, Precisely

A counting semaphore holds a non-negative integer. The API exposes two core operations:

- `sem_post(&s)` increments the counter. If anyone was waiting because the counter was zero, one of them is woken up.
- `sem_wait(&s)` decrements the counter if it is positive. If the counter is zero, the caller sleeps until `sem_post` increments it, then decrements and returns.

Those two operations, composed, give you bounded admission. Initialise the semaphore with N. Each participant calls `sem_wait` on entry and `sem_post` on exit. The invariant “at most N participants are between their wait and their post” is preserved automatically.

The FreeBSD counting semaphore differs from a binary semaphore (which can only be 0 or 1) in that the counter can go higher than 1. A binary semaphore is effectively a mutex, with one important difference: a semaphore has no ownership concept. Any thread may call `sem_post`; any thread may call `sem_wait`. A mutex, by contrast, must be released by the same thread that acquired it. This lack of ownership is important for exactly the use cases semaphores are best at: a producer that posts and a consumer that waits, which might be different threads.

15.3.2 The Data Structure

The data structure, from `/usr/src/sys/sys/semaphore.h`:

```
struct sema {  
    struct mtx      sema_mtx;      /* General protection lock. */
```



```
    struct cv      sema_cv;      /* Waiters. */
    int            sema_waiters;  /* Number of waiters. */
    int            sema_value;    /* Semaphore value. */
};
```

Four fields. `sema_mtx` is the semaphore's own internal mutex. `sema_cv` is the condition variable that waiters block on. `sema_waiters` counts the number of currently-blocked waiters (for diagnostic purposes and to avoid unnecessary broadcasts). `sema_value` is the counter itself.

You never touch these fields directly. The API is the contract; the structure is shown here once so you can visualise what the primitive is.

15.3.3 The API

From `/usr/src/sys/sys/sema.h`:

```
void sema_init(struct sema *sema, int value, const char *description);
void sema_destroy(struct sema *sema);
void sema_post(struct sema *sema);
void sema_wait(struct sema *sema);
int  sema_timedwait(struct sema *sema, int timo);
int  sema_trywait(struct sema *sema);
int  sema_value(struct sema *sema);
```

sema_init: initialises the semaphore with the given initial value and human-readable description. The description is used by kernel tracing facilities. The value must be non-negative; `sema_init` asserts this with `KASSERT`.

sema_destroy: tears the semaphore down. You must ensure no waiters are present when you call `sema_destroy`; the implementation asserts this. Typically you guarantee it by design: the destroy happens in `detach`, after every path that could `sema_wait` has been quiesced.

sema_post: increments the counter. If there are waiters, wakes one of them. Always succeeds.

sema_wait: if the counter is positive, decrements it and returns. Otherwise sleeps on the internal cv until `sema_post` increments the counter, at which point it decrements and returns. **sema_wait is not signal-interruptible**; it uses `cv_wait` under the hood, not `cv_wait_sig`. A signal will not wake the waiter. If you need interruptibility, `sema_wait` is the wrong tool; use a mutex-plus-cv pattern directly.

sema_timedwait: same as `sema_wait` but bounded by `timo` ticks. Returns 0 on success (value was decremented), `EWOULDBLOCK` on timeout. Internally uses `cv_timedwait`, so also not signal-interruptible.

sema_trywait: non-blocking variant. Returns 1 if the value was successfully decremented, 0 if the value was already zero. Note the unusual convention: 1 means success, 0 means failure. Most FreeBSD kernel APIs return 0 on success; `sema_trywait` is an exception. Be careful when reading or writing code that uses it.

sema_value: returns the current counter value. Useful for diagnostics; not useful for making synchronisation decisions, because the value can change immediately after the call returns.

15.3.4 What a Semaphore Is Not

Three properties the FreeBSD kernel semaphore does not have. Each is important.

No priority inheritance. The comment at the top of `/usr/src/sys/kern/kern_sema.c` is explicit:

Priority propagation will not generally raise the priority of semaphore “owners” (a misnomer in the context of semaphores), so should not be relied upon in combination with semaphores.

If you are protecting a resource and a high-priority thread is waiting on a semaphore held by a low-priority thread, the low-priority thread does not inherit the high priority. This is a consequence of the no-ownership design: there is no “holder” to raise. For resources where priority inheritance matters, use a mutex or a `lockmgr(9)` lock instead.

Not signal-interruptible. `sema_wait` and `sema_timedwait` are not interrupted by signals. A `read(2)` or `write(2)` that blocks in `sema_wait` will not return `EINTR` or `ERESTART` when the user sends `SIGINT`. If your syscall needs to respond to signals, you cannot block in `sema_wait` unconditionally. The two usual workarounds: structure the wait as `sema_trywait` plus an interruptible sleep on a separate `cv`, or keep `sema_wait` but arrange for the producer (the code that `sema_posts`) to also post when shutdown is in progress.

No ownership. Any thread can post; any thread can wait. This is a feature, not a bug, for the producer-consumer shape where one thread signals completion and another waits for it. It is a surprise if you were expecting mutex-like ownership semantics.

Knowing what a primitive is not is as important as knowing what it is. The FreeBSD kernel semaphore is a small, focused tool. Use it where it fits; reach for different primitives where it does not.

15.3.5 A Real-World Example: Hyper-V storvsc

Before the driver refactor, a short look at a real FreeBSD driver that uses `sema(9)` heavily. The Hyper-V storage driver lives at `/usr/src/sys/dev/hyperv/storvsc/hv_storvsc_drv_freebsd.c`. It uses per-request semaphores to block a thread waiting for hardware completion. The pattern:

```
/* In the request submission path: */
sema_init(&request->synch_sema, 0, "stor_synch_sema");
/* ... send command to hypervisor ... */
sema_wait(&request->synch_sema);
/* At this point the completion handler has posted; work is done. */
sema_destroy(&request->synch_sema);
```

And in the completion callback (run from a different context):

```
sema_post(&request->synch_sema);
```

The semaphore is initialised to zero, so `sema_wait` blocks. When hardware completes and the driver’s completion handler runs, it posts, and the submitting thread unblocks. The ownership-free nature of the semaphore is

exactly what makes this pattern work: a different thread (the completion handler) does the post than the one that does the wait.

The same driver uses a second semaphore (`hs_drain_sema`) for drain coordination during shutdown. The shutdown path waits on the semaphore; the request-completion path posts when all outstanding requests have finished.

These patterns are not inventions. They are the canonical uses of `sema(9)` in the FreeBSD tree. The Chapter 15 refactor uses a variation for the “at most N writers” invariant. The underlying idea is the same.

15.3.6 The Writer-Cap Refactor: Stage 1

The first Chapter 15 change to the driver adds a counting semaphore that caps the number of concurrent `myfirst_write` callers. The cap is configurable via a `sysctl`, with a default of 4.

The change is not about performance. The driver can already handle many concurrent writers; the `cbuf` is protected by the mutex and writes serialise there anyway. The change is about expressing the invariant “at most N writers” as a first-class primitive. A real driver might use this pattern for more substantive reasons (a fixed-size DMA descriptor pool, a hardware command queue with bounded depth, a serial device with a transmit window); the refactor is a didactic vehicle for learning the primitive in a context you can run and observe.

15.3.7 The Softc Addition

Add three members to `struct myfirst_softc`:

```
struct sema    writers_sema;
int            writers_limit;           /* Current configured limit. */
int            writers_trywait_failures; /* Diagnostic counter. */
```

`writers_sema` is the semaphore itself. `writers_limit` records the current configured value so the `sysctl` handler can detect changes. `writers_trywait_failures` counts the number of times a writer tried to enter, could not, and returned `EAGAIN` (for `O_NONBLOCK` opens) or `EWOULDBLOCK` (for bounded waits).

15.3.8 Initialising and Destroying the Semaphore

In `myfirst_attach`, before any code that could call `sema_wait` (so typically alongside the other `sema_init/cv_init` calls early in `attach`):

```
sema_init(&sc->writers_sema, 4, "myfirst writers");
sc->writers_limit = 4;
sc->writers_trywait_failures = 0;
```

The initial value of 4 matches the default limit. If we later raise the limit dynamically, we will adjust the semaphore’s value to match; Section 2 shows how.

In `myfirst_detach`, after every path that could `sema_wait` has been quiesced (which, at Stage 1, means after `is_attached` is cleared and all user syscalls have returned or failed with `ENXIO`):

```
sema_destroy(&sc->writers_sema);
```

A subtle point here, and a genuinely tricky one worth slowing down for. `sema_destroy` asserts that no waiters are present; more importantly, it then calls `mtx_destroy` on the semaphore's internal mutex and `cv_destroy` on its internal cv. If any thread is still executing inside any `sema_*` function, that thread may be about to re-acquire the internal mutex when `mtx_destroy` races ahead and frees it. That is a use-after-free, not just an assertion failure.

The naive cure “just post `writers_limit` slots to wake blocked waiters, then destroy” is *almost* correct but has a real race. A woken thread returns from `cv_wait` with the internal `sema_mtx` held, then needs to execute `sema_waiters--` and the final `mtx_unlock`. If the detach thread runs `sema_destroy` before the woken thread reaches its final unlock, the internal mutex is destroyed underneath it.

In practice that window is short (the woken thread typically runs within microseconds of `cv_signal`), but correctness means we cannot rely on “usually works”. The cure is a small extension: track every thread that might currently be inside `sema_*` code and wait for that count to reach zero before calling `sema_destroy`.

We add `sc->writers_inflight`, an int the driver treats as atomic. The write path increments it before calling `sema_wait` and decrements it after calling the matching `sema_post`. The detach path, after posting the wake-up slots, waits for the counter to reach zero:

```
/* In the write path, early: */
atomic_add_int(&sc->writers_inflight, 1);
if (!sc->is_attached) {
    atomic_subtract_int(&sc->writers_inflight, 1);
    return (ENXIO);
}
... sema_wait / work / sema_post ...
atomic_subtract_int(&sc->writers_inflight, 1);

/* In detach, after the posts: */
while (atomic_load_acq_int(&sc->writers_inflight) > 0)
    pause("myfwrdr", 1);
sema_destroy(&sc->writers_sema);
```

Why this works: any thread that could possibly be using the `sema`'s internal state has been counted. Detach waits until every counted thread has finished its final `sema_post`, which by the time the decrement fires has already returned from every `sema_*` function. No thread is still holding or about to acquire the internal mutex when `sema_destroy` runs.

The pattern is worth remembering because it is general: any external primitive whose destroy races with in-flight callers can be drained the same way. `sema(9)` is the immediate example; you will see variants of this counter in real drivers whenever a primitive without a built-in drain needs to be torn down cleanly.

The chapter's Stage 1 through Stage 4 drivers all implement this pattern. Section 6 encapsulates the logic in `myfirst_sync_writer_enter/myfirst_sync_writer_leave` so the call sites read naturally; the inflight bookkeeping hides in the wrapper.

15.3.9 Using the Semaphore in the Write Path

Add `sema_wait/sema_post` around the body of `myfirst_write`:

```
static int
myfirst_write(struct cdev *dev, struct uio *uio, int ioflag)
{
    struct myfirst_softc *sc = dev->si_drv1;
    struct myfirst_fh *fh;
    char bounce[MYFIRST_BOUNCE];
    size_t want, put, room;
    ssize_t nbefore;
    int error;

    error = devfs_get_cdevpriv((void **)&fh);
    if (error != 0)
        return (error);
    if (sc == NULL || !sc->is_attached)
        return (ENXIO);

    /* Chapter 15: enforce the writer cap. */
    if (ioflag & IO_NDELAY) {
        if (!sema_trywait(&sc->writers_sema)) {
            MYFIRST_LOCK(sc);
            sc->writers_trywait_failures++;
            MYFIRST_UNLOCK(sc);
            return (EAGAIN);
        }
    } else {
        sema_wait(&sc->writers_sema);
    }
    if (!sc->is_attached) {
        sema_post(&sc->writers_sema);
        return (ENXIO);
    }

    nbefore = uio->uio_resid;
    while (uio->uio_resid > 0) {
        /* ... same body as Chapter 14 ... */
    }

    sema_post(&sc->writers_sema);
    return (0);
}
```

Several things to notice.

The `IO_NDELAY` (non-blocking) case uses `sema_trywait`, which returns 1 on success and 0 on failure. Note the inverted convention: `if (!sema_trywait(...))` means “if we failed to acquire”. Beginners miss this regularly; read the return value with care every time.

On `sema_trywait` failure the non-blocking caller gets `EAGAIN`. A diagnostic counter is incremented under the mutex (brief mutex acquire/release, unrelated to the semaphore).

The blocking case uses `sema_wait`. It is not signal-interruptible, so a blocking `write(2)` waiting for the semaphore cannot be interrupted by `SIGINT`. This is an important property; users must know it. For the current driver the semaphore is rarely contended in practice (the default limit of 4 is generous), so the interruptibility concern is largely theoretical. If the limit were 1 and writers genuinely queued, you might want to revisit using a semaphore here and instead use an interruptible primitive. Section 4 returns to this trade-off.

After the wait returns we check `is_attached`. If detach happened while we were blocked, we must not proceed with the write; we post the semaphore (restoring the count) and return `ENXIO`.

The `sema_post` at the exit path runs on every successful path. A common mistake is to forget it on an early return (for example, if an intermediate validation fails). The usual discipline is to make the post unconditional via a cleanup pattern: acquire, then all subsequent exits go through one common cleanup.

15.3.10 The Sysctl Handler for the Limit

Users of the driver may want to adjust the writer cap at runtime. The sysctl handler:

```
static int
myfirst_sysctl_writers_limit(SYSCTL_HANDLER_ARGS)
{
    struct myfirst_softc *sc = arg1;
    int new, old, error, delta;

    old = sc->writers_limit;
    new = old;
    error = sysctl_handle_int(oidp, &new, 0, req);
    if (error || req->newptr == NULL)
        return (error);
    if (new < 1 || new > 64)
        return (EINVAL);

    MYFIRST_LOCK(sc);
    delta = new - sc->writers_limit;
    sc->writers_limit = new;
    MYFIRST_UNLOCK(sc);

    if (delta > 0) {
        /* Raised the limit: post the extra slots. */
        int i;
```

```

        for (i = 0; i < delta; i++)
            sema_post(&sc->writers_sema);
    }
    /*
     * Lowering is best-effort: we cannot reclaim posted slots from
     * threads already in the write path. New entries will observe
     * the lower limit once the counter drains below the new cap.
     */
    return (0);
}

```

Interesting details.

Raising the limit requires posting extra slots to the semaphore. If the old limit was 4 and the new limit is 6, we need to post twice so two more writers can enter simultaneously.

Lowering the limit is harder. A semaphore has no way to “consume” excess slots back out of it. If the current counter is 4 and we want a limit of 2, we cannot reduce the counter except by waiting for writers to enter and not posting on their exit. That is complicated and rarely worth the code. Instead, the simple approach: lower the `writers_limit` field, and let the semaphore drain naturally to the new level as writers enter without replacement. The `sysctl` handler comment documents this behaviour.

The mutex is held only for the `writers_limit` read/write, not for the `sema_post` loop. Taking the mutex around `sema_post` would be incorrect anyway: `sema_post` acquires its own internal mutex, and we would be introducing a lock order `sc->mtx -> sc->writers_sema.sema_mtx` that nothing else uses. Since `writers_limit` is the only field we actually protect, the mutex window is small.

15.3.11 Observing the Effect

With Stage 1 loaded, a few experiments.

Start many concurrent writers using a small shell loop:

```

# for i in 1 2 3 4 5 6 7 8; do
    (yes "writer-$i" | dd of=/dev/myfirst bs=512 count=100 2>/dev/null) &
done

```

Eight writers start simultaneously. With `writers_limit=4` (the default), four enter the write loop and the other four block in `sema_wait`. As one finishes and calls `sema_post`, a blocked one wakes up. The throughput is slightly lower than unconstrained (because only four writers actively progress at any moment), but the `cbuf` never has more than four writers contending for the mutex.

Observe the semaphore value live:

```

# sysctl dev.myfirst.0.stats.writers_sema_value
dev.myfirst.0.stats.writers_sema_value: 0

```

During the stress test the value should be near zero. When no writers are present it should equal `writers_limit`.

Tune the limit dynamically:

```
# sysctl dev.myfirst.0.writers_limit=2
```

Rerun the eight-writer stress. Two writers progress; six block. Throughput drops accordingly.

Tune the limit back up:

```
# sysctl dev.myfirst.0.writers_limit=8
```

All eight writers proceed concurrently.

Check the trywait-failure counter by using non-blocking writers (via `open` with `O_NONBLOCK`):

```
# ./nonblock_writer_stress.sh
# sysctl dev.myfirst.0.stats.writers_trywait_failures
```

The count grows whenever a non-blocking writer is turned away because the semaphore was at zero.

15.3.12 Common Mistakes

A short list of mistakes beginners make with `sema(9)`. Each has bitten real drivers; each has a simple rule.

Forgetting to `sema_post` on an error path. If the write path has a `return (error)` that bypasses the `sema_post`, the semaphore leaks a slot. After enough leaks the semaphore is permanently at zero and all writers block. The fix is either to place the `sema_post` in a single cleanup block that all exits flow through, or to audit every `return` statement to confirm it posts.

`sema_wait` in a context that cannot sleep. `sema_wait` blocks. It cannot be called from a callout callback, an interrupt filter, or any other non-sleeping context. The WITNESS assertion catches this on a debug kernel; the production kernel may deadlock or panic silently.

Destroying a semaphore with waiters. `sema_destroy` asserts that no waiters exist. In a driver's detach, the careful thing to do is drain every path that could wait before destroying. If the detach ordering is wrong (destroying before the waiters have woken), the assertion fires on a debug kernel and the destroy silently corrupts on a production kernel.

Using `sema_wait` where signal interruptibility is needed. Users expect `read(2)` and `write(2)` to respond to `SIGINT`. If the syscall blocks in `sema_wait`, it does not. Either pick a different primitive or structure the code so `sema_wait` is short enough that signal latency is acceptable.

Confusing `sema_trywait`'s return value. Returns 1 on success, 0 on failure. Most FreeBSD kernel APIs return 0 on success. A misreading of the return value produces the opposite of the intended behaviour. Always double-check this one.

Assuming priority inheritance. If the invariant requires that a high-priority waiter raise the effective priority of the thread that will post, `sema(9)` will not do that. Use a mutex or a `lockmgr(9)` lock instead.

15.3.13 A Note on When Not to Use Semaphores

For completeness, a short list of situations where `sema(9)` is the wrong tool.

- **When the invariant is “exclusive ownership of a resource”.** That is a mutex. A semaphore initialised to 1 approximates it but loses ownership semantics and priority inheritance.
- **When the waiter must be signal-interruptible.** Use `cv_wait_sig` or `cv_timedwait_sig` with your own counter.
- **When the work is short and contention is high.** The semaphore’s internal mutex is a single point of serialisation. For very short critical sections, the overhead may dominate.
- **When priority inheritance is required.** Use a mutex or `lockmgr(9)`.
- **When you need more than counting.** If the invariant is “wait until this specific complex predicate holds”, a mutex and a `cv` that tests the predicate is the right tool.

For the driver’s writer-cap use case, none of these disqualifications apply. The semaphore is the right tool, the refactor is small, and the resulting code is readable. Stage 1 of the Chapter 15 driver keeps the new vocabulary and moves on.

15.3.14 Wrapping Up Section 2

A counting semaphore is a counter, a mutex, and a condition variable wrapped into a small API. `sema_init`, `sema_wait`, `sema_post`, `sema_trywait`, `sema_timedwait`, `sema_value`, and `sema_destroy` cover the whole surface. The primitive is ideal for bounded admission and for producer-consumer completion shapes where the producer and consumer are different threads. It lacks priority inheritance, signal interruptibility, and ownership, and those limitations are real. Stage 1 of the Chapter 15 driver applied a writer-cap semaphore; the next section applies a read-mostly `sx` refinement.

15.4 Section 3: Read-Mostly Scenarios and Shared Access

The `sx` lock from Chapter 12 is already in the driver. `sc->cfg_sx` protects the `myfirst_config` structure, and the configuration sysctls acquire it in shared mode for reads and exclusive mode for writes. That pattern is correct and, for the configuration use case, sufficient. This section refines the `sx` pattern to cover a slightly different shape: a read-mostly cache where readers occasionally notice that the cache needs updating and must promote to a writer briefly.

This section also introduces a small number of `sx` operations the driver has not used yet: `sx_try_upgrade`, `sx_downgrade`, `sx_xlocked`, and a few introspection macros. The Stage 2 driver refactor adds a tiny statistics cache protected by its own `sx` and uses the upgrade pattern to refresh the cache under light contention.

15.4.1 The Read-Mostly Cache Problem

A concrete motivating problem for the Stage 2 refactor. Suppose the driver wants to expose a computed statistic, “average bytes written per second over the last 10 seconds”. The statistic is expensive to compute (it requires a walk over a per-second history buffer) and is read often (every sysctl read, every heartbeat log line). A naive

implementation recomputes on every read. A better implementation caches the result and invalidates the cache periodically.

The cache has three properties:

1. Reads vastly outnumber writes. Any number of threads may read simultaneously; only the occasional cache refresh needs to write.
2. Readers sometimes detect that the cache is stale. When that happens, the reader wants to promote to a writer briefly, refresh the cache, and return to reading.
3. Refreshing the cache takes a few microseconds. A reader that promotes and refreshes still wants to release the exclusive lock quickly.

The `sx` lock handles properties 1 and 3 natively: many readers can hold `sx_s` lock simultaneously; a writer with `sx_x` lock excludes readers. Property 2 requires `sx_try_upgrade`.

15.4.2 `sx_try_upgrade` and `sx_downgrade`

Two operations on the `sx` lock that Chapter 12 did not introduce.

`sx_try_upgrade(&sx)` attempts to atomically promote a shared lock to an exclusive lock. Returns non-zero on success, zero on failure. A failure means another thread also holds the shared lock (exclusive-with-other-readers is not representable; the upgrade can only succeed if the calling thread is the sole shared holder). On success the shared lock is gone and the caller now holds the exclusive lock.

`sx_downgrade(&sx)` atomically demotes an exclusive lock to a shared lock. Always succeeds. The exclusive holder becomes a shared holder; other shared lockers can then join.

The pattern for read-with-occasional-upgrade:

```
sx_slock(&sx);
if (cache_stale(&cache)) {
    if (sx_try_upgrade(&sx)) {
        /* Promoted to exclusive. */
        refresh_cache(&cache);
        sx_downgrade(&sx);
    } else {
        /*
         * Upgrade failed: another reader holds the lock.
         * Release the shared lock, take the exclusive lock,
         * refresh, downgrade.
         */
        sx_sunlock(&sx);
        sx_xlock(&sx);
        if (cache_stale(&cache))
            refresh_cache(&cache);
        sx_downgrade(&sx);
    }
}
```

```
use_cache(&cache);  
sx_sunlock(&sx);
```

Three things to notice.

The happy path is the `sx_try_upgrade` success. The upgrade is atomic: at no point is the lock released and re-acquired, so no other writer can slip in between. For a read-mostly workload where readers rarely contend with each other, this path dominates.

The fallback path when `sx_try_upgrade` fails drops the shared lock entirely, acquires the exclusive lock from scratch, and re-checks the stale predicate. The re-check is essential: between dropping the shared lock and acquiring the exclusive lock, another thread may have refreshed the cache. Without the re-check you would refresh redundantly.

The final `sx_sunlock` after `sx_downgrade` is always correct because the downgraded state is shared.

This pattern is surprisingly common in the FreeBSD source tree. Search for `sx_try_upgrade` under `/usr/src/sys/` and you find it in several subsystems, including VFS and the routing table updates.

15.4.3 A Worked Application: Stage 2 Driver

Stage 2 of the Chapter 15 driver adds a small statistics cache protected by its own `sx` lock. The cache holds a single integer, “bytes_written in the last 10 seconds, as of the last refresh”, and a timestamp recording when the cache was last refreshed.

The softc addition:

```
struct sx      stats_cache_sx;  
uint64_t      stats_cache_bytes_10s;  
uint64_t      stats_cache_last_refresh_ticks;
```

The cache validity is based on a timestamp. If the current ticks differs from `stats_cache_last_refresh_ticks` by more than `hz` (one second’s worth), the cache is considered stale. Any `sysctl` read of the cached value triggers a staleness check; if stale, the reader promotes and refreshes.

15.4.4 The Cache Refresh Function

The refresh function is trivial for the didactic version: it just reads the current counter and records the current time.

```
static void  
myfirst_stats_cache_refresh(struct myfirst_softc *sc)  
{  
    KASSERT(sx_xlocked(&sc->stats_cache_sx),  
            ("stats cache not exclusively locked"));  
    sc->stats_cache_bytes_10s = counter_u64_fetch(sc->bytes_written);  
}
```

```

        sc->stats_cache_last_refresh_ticks = ticks;
    }

```

The KASSERT documents the contract: this function must be called with the sx held exclusively. A debug kernel catches violations at runtime.

15.4.5 The Sysctl Handler

The sysctl handler that reads the cached value:

```

static int
myfirst_sysctl_stats_cached(SYSCTL_HANDLER_ARGS)
{
    struct myfirst_softc *sc = arg1;
    uint64_t value;
    int stale;

    sx_slock(&sc->stats_cache_sx);
    stale = (ticks - sc->stats_cache_last_refresh_ticks) > hz;
    if (stale) {
        if (sx_try_upgrade(&sc->stats_cache_sx)) {
            myfirst_stats_cache_refresh(sc);
            sx_downgrade(&sc->stats_cache_sx);
        } else {
            sx_sunlock(&sc->stats_cache_sx);
            sx_xlock(&sc->stats_cache_sx);
            if ((ticks - sc->stats_cache_last_refresh_ticks) > hz)
                myfirst_stats_cache_refresh(sc);
            sx_downgrade(&sc->stats_cache_sx);
        }
    }
    value = sc->stats_cache_bytes_10s;
    sx_sunlock(&sc->stats_cache_sx);

    return (sysctl_handle_64(oidp, &value, 0, req));
}

```

The shape matches the pattern from the previous subsection. Worth reading once carefully. The staleness check happens twice in the fallback path: once to decide whether to take the exclusive lock at all, and once after acquisition to confirm the staleness still applies.

15.4.6 Attach and Detach

Initialise the sx in myfirst_attach, alongside the existing cfg_sx:

```
sx_init(&sc->stats_cache_sx, "myfirst stats cache");
sc->stats_cache_bytes_10s = 0;
sc->stats_cache_last_refresh_ticks = 0;
```

Destroy in `myfirst_detach`, after the taskqueue and mutex are torn down (the `sx` is ordered below the mutex in the lock graph; destroy it after the mutex for symmetry with init):

```
sx_destroy(&sc->stats_cache_sx);
```

No waiters should be present at destroy time. Readers may still be in progress if detach races with a `sysctl`, but the detach path does not access the cache `sx` so the two do not directly conflict. If a `sysctl` is executing when detach proceeds, the `sysctl` framework holds its own reference and the context will tear down in order.

15.4.7 Observing the Effect

Read the cached stat a thousand times quickly:

```
# for i in $(jot 1000 1); do
    sysctl -n dev.myfirst.0.stats.bytes_written_10s >/dev/null
done
```

Most reads hit the cache without promoting. Only the first read after a cache expiry refreshes. The result: near-zero contention on the stats cache `sx` under read-mostly load.

Observe refresh rate via `DTrace`:

```
# dtrace -n '
    fbt::myfirst_stats_cache_refresh:entry {
        @[execname] = count();
    }
' -c 'sleep 10'
```

Should show roughly ten refreshes per second (one per cache expiry), regardless of how many read requests arrived.

15.4.8 The `sx` Macro Vocabulary

A few macros and helpers the chapter has not used yet that are worth knowing.

`sx_xlocked(&sx)` returns non-zero if the current thread holds the `sx` exclusively. Useful inside assertions. Does not tell you whether a different thread holds it; there is no equivalent query for that.

`sx_xholder(&sx)` returns the thread pointer of the exclusive holder, or `NULL` if no one holds it exclusively. Useful in debugging output.

`sx_assert(&sx, what)` asserts a property of the lock state. `SX_LOCKED`, `SX_SLOCKED`, `SX_XLOCKED`, `SX_UNLOCKED`, `SX_XLOCKED | SX_NOTRECURSED`, and others are valid. Panics on mismatch when `INVARIANTS` is enabled.

For the Chapter 15 refactor we use `sx_xlocked` in the cache refresh `KASSERT`. The other macros are available when you need them.

15.4.9 Trade-offs and Cautions

Some trade-offs worth naming.

Shared locks have overhead. An `sx` in shared mode still takes a spin on the internal spin lock plus a couple of atomic operations. For extremely hot paths (tens of millions of operations per second), this can be measurable. `atomic(9)` with a seq-cst fence is sometimes cheaper. For the driver's workloads, `sx` is fine.

Upgrade failures are a real possibility. A workload with many concurrent readers will see `sx_try_upgrade` fail often. The fallback path (drop shared, acquire exclusive, re-check) does the right thing but has slightly higher latency. For true read-mostly workloads where upgrades are rare, the success path dominates.

Sx locks can sleep. Unlike a mutex, `sx`'s slow path blocks. Do not call `sx_slock`, `sx_xlock`, `sx_try_upgrade`, or `sx_downgrade` from a context that cannot sleep (callouts initialised without a sleepable lock, interrupt filters, etc.). Chapter 13 explains that the older `CALLOUT_MPSAFE` flag is deprecated; the modern test is whether the callout was set up through `callout_init(, 0)` or `callout_init_mtx(, &mtx, 0)`.

Lock order still matters. Adding an `sx` to the driver means adding a new node to the lock graph. Every code path that holds multiple locks must respect a consistent order. The Chapter 15 driver's final lock order is `sc->mtx -> sc->cfg_sx -> sc->stats_cache_sx`; `WITNESS` enforces it.

15.4.10 Wrapping Up Section 3

An `sx` lock covers many-readers-or-one-writer naturally. `sx_try_upgrade` and `sx_downgrade` extend that to read-mostly-with-promotion. The pattern with a happy-path upgrade and a fallback re-check is the canonical way to express "a reader noticed a need to write, briefly". Stage 2 of the driver added a small statistics cache with this pattern; Stage 3 will refine signal-interruptible waits.

15.5 Section 4: Condition Variables with Timeout and Interruption

The `cv_wait_sig` and `cv_timedwait_sig` primitives are already in the driver. Chapter 12 introduced them; Chapter 13 refined them for the tick-source driver. This section takes the next step: it distinguishes the return values these primitives produce, shows how to handle `EINTR` and `ERESTART` correctly, and refactors the driver's read path to preserve partial progress across signal interruption. This is Stage 3 of the Chapter 15 driver.

Unlike the previous sections, this one does not introduce a new primitive. It introduces a discipline for using a primitive you already know.

15.5.1 What the Return Values Mean

`cv_wait_sig`, `cv_timedwait_sig`, `mtx_sleep` with the `PCATCH` flag, and similar signal-aware waits can return several values:

- **0**: normal wakeup. The caller was woken by a matching `cv_signal` or `cv_broadcast`. Re-check the predicate; if true, proceed; if false, wait again.
- **EINTR**: interrupted by a signal that has a handler installed. The caller should abandon the wait, do any appropriate cleanup, and return `EINTR` to its own caller.
- **ERESTART**: interrupted by a signal whose handler specifies automatic restart. The kernel will re-invoke the syscall. The driver should return `ERESTART` to the syscall layer, which arranges for the restart.
- **EWOULDBLOCK**: only from timed waits. The timeout fired before any wakeup or signal arrived.

The distinction between `EINTR` and `ERESTART` matters because the driver returns these values back through the syscall path and userland handles them differently:

- If the syscall returns `EINTR`, userland's `read(2)` or `write(2)` returns -1 with `errno` set to `EINTR`. User code that did not install a `SA_RESTART` signal handler sees this explicitly.
- If the syscall returns `ERESTART`, the syscall machinery restarts the syscall transparently. Userland never sees the signal delivery at this level; the signal handler ran, but the read call continues.

The practical consequence: if your `cv_wait_sig` returns `EINTR`, the user will see an `EINTR` from their `read(2)` and any partial progress they might have expected must be explicit (the read returns the bytes copied before the signal, not an error, by convention). If it returns `ERESTART`, the restart happens and the read continues where the kernel saw fit.

15.5.2 The Partial-Progress Convention

UNIX convention for `read(2)` and `write(2)`: if a signal arrives after some data has been transferred, the syscall returns the number of bytes transferred, not an error. If no data has been transferred, the syscall returns `EINTR` (or restarts, depending on signal disposition).

Translating to the driver: at the entry of the read path, record the initial `uio_resid`. When the blocking wait returns a signal error, compare the current `uio_resid` with the recorded one. If progress was made, return 0 (which the syscall layer translates to “return the number of bytes copied”). If no progress was made, return the signal error.

The Chapter 12 driver already implements this convention for `myfirst_read` via the `nbefore` local and the “return -1 to caller to indicate partial progress” trick. Chapter 15 refines the handling, makes it explicit, and extends it to the write path.

15.5.3 The Refactored Read Path

The Chapter 14 Stage 4 read path has this shape:

```
while (uio->uio_resid > 0) {  
    MYFIRST_LOCK(sc);
```

```

        error = myfirst_wait_data(sc, ioflag, nbefore, uio);
        if (error != 0) {
            MYFIRST_UNLOCK(sc);
            return (error == -1 ? 0 : error);
        }
        ...
    }

```

And `myfirst_wait_data` returns -1 to signal “partial progress; return 0 to the user”. That convention is correct but cryptic. The Stage 3 refactor replaces the -1 magic value with a named sentinel and documents the convention in a comment:

```

#define MYFIRST_WAIT_PARTIAL    (-1)    /* partial progress already made */

static int
myfirst_wait_data(struct myfirst_softc *sc, int ioflag, ssize_t nbefore,
    struct uio *uio)
{
    int error, timo;

    MYFIRST_ASSERT(sc);
    while (cbuf_used(&sc->cb) == 0) {
        if (uio->uio_resid != nbefore) {
            /*
             * Some bytes already delivered on earlier loop
             * iterations. Do not block further; return
             * "partial progress" so the caller returns 0
             * to the syscall layer, which surfaces the
             * partial byte count.
             */
            return (MYFIRST_WAIT_PARTIAL);
        }
        if (ioflag & IO_NDELAY)
            return (EAGAIN);

        timo = sc->read_timeout_ms;
        if (timo > 0) {
            int ticks_total = (timo * hz + 999) / 1000;
            error = cv_timedwait_sig(&sc->data_cv, &sc->mtx,
                ticks_total);
        } else {
            error = cv_wait_sig(&sc->data_cv, &sc->mtx);
        }
        switch (error) {

```



```

        case 0:
            break;
        case EWOULDBLOCK:
            return (EAGAIN);
        case EINTR:
        case ERESTART:
            if (uio->uio_resid != nbefore)
                return (MYFIRST_WAIT_PARTIAL);
            return (error);
        default:
            return (error);
    }
    if (!sc->is_attached)
        return (ENXIO);
}
return (0);
}

```

Several changes from the Chapter 14 version.

The magic -1 is now MYFIRST_WAIT_PARTIAL, with a comment explaining what it means.

The error handling after the cv wait is explicit about what each return value means. EWOULDBLOCK becomes EAGAIN (which is the conventional user-facing error for “try again later”). EINTR and ERESTART are checked for partial progress: if any bytes were delivered, we return the partial sentinel; if not, we propagate the signal error.

The default case handles any other error the cv wait might return. Currently the kernel’s cv_timedwait_sig returns only the values listed above, but explicit handling of the unexpected case is a habit worth keeping.

15.5.4 Handling at the Caller

In myfirst_read the handling of the sentinel becomes slightly clearer:

```

while (uio->uio_resid > 0) {
    MYFIRST_LOCK(sc);
    error = myfirst_wait_data(sc, ioflag, nbefore, uio);
    if (error != 0) {
        MYFIRST_UNLOCK(sc);
        if (error == MYFIRST_WAIT_PARTIAL)
            return (0);
        return (error);
    }
    ...
}

```

The reader can see at a glance what “partial progress” means. A maintainer adding a new early-exit reason later knows to check whether it should propagate to the user or be suppressed as partial.

15.5.5 The Write Path Gets the Same Treatment

The Chapter 14 write path already implements partial-progress handling in `myfirst_wait_room`. Stage 3 applies the same refactor there: replace `-1` with `MYFIRST_WAIT_PARTIAL`, make the error-handling switch explicit, and document the convention.

One small additional change for the write path. The write path’s `sema_wait` from Section 2 is not signal-interruptible. Before the semaphore change, a blocked write would be interruptible via `cv_wait_sig` inside `myfirst_wait_room`. After adding the semaphore, a write that blocks on `sema_wait` (waiting for a writer slot) is not interruptible.

Is that acceptable? For most workloads yes, because the writer cap is typically not contended. For a workload where the cap is 1 and writers genuinely queue for long periods, users would expect `SIGINT` to work. The trade-off is an explicit trade-off; Section 5 will show how to make the wait interruptible by layering a signal-aware wait around `sema_trywait`.

For Stage 3, we accept the non-interruptible `sema_wait` for the default case and note the trade-off in a comment:

```
/*
 * The writer-cap semaphore wait is not signal-interruptible. For a
 * workload where the cap is rarely contended this is acceptable. If
 * you set writers_limit=1 and create a real queue of writers, consider
 * the interruptible alternative in Section 5.
 */
sema_wait(&sc->writers_sema);
```

15.5.6 The Interruptible Wait Pattern

For readers who want the fully interruptible version now: combine `sema_trywait` with a retry loop that uses a `cv` for interruptible sleep. The code is moderately verbose, which is why Chapter 15 defers it to an optional subsection.

```
static int
myfirst_writer_enter_interruptible(struct myfirst_softc *sc)
{
    int error;

    MYFIRST_LOCK(sc);
    while (!sema_trywait(&sc->writers_sema)) {
        if (!sc->is_attached) {
            MYFIRST_UNLOCK(sc);
```

```

        return (ENXIO);
    }
    error = cv_wait_sig(&sc->writers_wakeup_cv, &sc->mtx);
    if (error != 0) {
        MYFIRST_UNLOCK(sc);
        return (error);
    }
}
MYFIRST_UNLOCK(sc);
return (0);
}

```

This requires a second cv (`writers_wakeup_cv`) that the exit path signals after every `sema_post`:

```

sema_post(&sc->writers_sema);
/* Wake one interruptible waiter so they can retry sema_trywait. */
cv_signal(&sc->writers_wakeup_cv);

```

The interruptible version preserves `EINTR/ERESTART` handling correctly. It is longer than the plain `sema_wait` version, and for most drivers the trade-off is not worth the extra code. But the pattern exists when it is needed.

15.5.7 Common Mistakes

Treating `EWOULDBLOCK` as a normal wakeup. The timed wait returns `EWOULDBLOCK` when the timer fires. Treating this as 0 and re-testing the predicate is wrong: the predicate is probably still false and the loop spins indefinitely.

Treating `EINTR` as a recoverable wakeup. `EINTR` means the caller should abandon the wait. A loop that does `while (... != 0) cv_wait_sig(...)` with no `EINTR` handling never propagates signals back to user space.

Forgetting the partial-progress check. A read that copied half its bytes and gets interrupted should return the half; a naive implementation returns `EINTR` with zero bytes copied, losing the partial data.

Mixing `cv_wait` with signal-interruptible callers. `cv_wait` (without `_sig`) blocks even during signal delivery. A syscall that uses `cv_wait` cannot be interrupted; users' `SIGINT` does nothing until the predicate is satisfied. Always use `cv_wait_sig` in syscall contexts.

Forgetting to re-check the predicate after wakeup. Both signals and `cv_signal` wake the waiter. The predicate may not be true on wakeup (spurious wakeups are allowed by the API). Always check the predicate in a loop.

15.5.8 Wrapping Up Section 4

Signal-interruptible waits have four distinct return values: 0 (normal), `EINTR` (signal without restart), `ERESTART` (signal with restart), `EWOULDBLOCK` (timeout). Each has a specific meaning and the driver must

handle each explicitly. The partial-progress convention (return the bytes copied so far, not an error) is the UNIX standard for reads and writes. Stage 3 of the driver applied this discipline and made the partial-progress sentinel explicit. Section 5 takes the coordination story further.

15.6 Section 5: Synchronization Between Modules or Subsystems

Up to now every primitive in the driver has been local to a single function or file section. A mutex protects a buffer; a cv signals readers; a sema limits writers; an sx caches statistics. Each primitive solves one problem in one place.

Real drivers have coordination that spans across subsystems. A callout fires, needs a task to finish its work, needs a user thread to see the resulting state, and needs another subsystem to notice that shutdown is in progress. The primitives you already know are sufficient; the difficulty is in composing them so the cross-component handshake is explicit and maintainable.

This section teaches the composition. It introduces a small atomic-flag discipline for cross-context shutdown visibility, a state-flag handshake between the callout and the task, and the beginnings of a wrapper layer that Section 6 will formalise. Stage 4 of the Chapter 15 driver begins here.

15.6.1 The Shutdown-Flag Problem

A recurring problem in driver detach: several contexts need to know that shutdown is in progress. The Chapter 14 driver uses `sc->is_attached` as this flag, read under the mutex in most places and occasionally read unprotected (with the comment “read at handler entry may be unprotected”). This works, but it has two subtle issues.

First, the unprotected read is technically undefined behaviour in pure C. A concurrent writer and an unsynchronised reader is a data race; the compiler is free to transform the code in ways that assume no concurrent access. Current kernel compilers rarely do, but the code is not strictly portable and a future compiler could break it.

Second, the mutex-protected reads serialise across a lock even when the only thing you want is a quick “is it shutdown yet” peek. In a hot path this cost is measurable.

The modern discipline: use `atomic_load_int` for reads and `atomic_store_int` (or `atomic_store_rel_int`) for writes. These operations are defined by the C memory model to be well-ordered and race-free. They are also very cheap: on x86 a plain load or store with the right barriers; on other architectures a single atomic instruction.

15.6.2 The Atomic API in One Page

`/usr/src/sys/sys/atomic_common.h` and architecture-specific headers define the atomic operations. The ones you will use most:

- `atomic_load_int(p)`: reads `*p` atomically. No memory barrier.
- `atomic_load_acq_int(p)`: reads `*p` atomically with acquire semantics. Subsequent memory accesses cannot be reordered before the load.
- `atomic_store_int(p, v)`: writes `v` to `*p` atomically. No memory barrier.

- `atomic_store_rel_int(p, v)`: writes `v` to `*p` atomically with release semantics. Preceding memory accesses cannot be reordered after the store.
- `atomic_fetchadd_int(p, v)`: returns old `*p` and sets `*p = *p + v` atomically.
- `atomic_cmpset_int(p, old, new)`: if `*p == old`, sets `*p = new` and returns 1; otherwise returns 0.

For the shutdown flag the pattern is:

- Writer (detach): `atomic_store_rel_int(&sc->is_attached, 0)`. The release ensures any prior state changes (draining, cv broadcasts) are visible before the flag becomes 0.
- Readers (any context): `if (atomic_load_acq_int(&sc->is_attached) == 0) { ... }`. The acquire ensures any subsequent checks see the state the writer intended.

The chapter uses `atomic_load_acq_int` in read-side shutdown checks and `atomic_store_rel_int` in the detach path. This makes the shutdown visibility correct across every context without introducing a mutex cost in the hot path.

15.6.3 Why Not Just a Mutex-Protected Flag?

A fair question. The answer is “because the atomic pattern is cheaper and equally correct for this specific invariant”. The flag has exactly two states (1 and 0), the transition is one-way (from 1 to 0, then never back to 1 in this lifetime), and no reader needs atomicity-with-other-state-changes; each reader only wants “is it still attached?”.

For an invariant with multiple fields or bidirectional transitions a mutex is the right tool. For a monotonic one-bit flag, atomics win.

15.6.4 Applying the Atomic Flag

The Stage 4 refactor converts `sc->is_attached` reads to atomic loads where they currently happen outside the mutex. The places to change are:

- `myfirst_open`: the entry check `if (sc == NULL || !sc->is_attached)`.
- `myfirst_read`: the entry check after `devfs_get_cdevpriv`.
- `myfirst_write`: the entry check after `devfs_get_cdevpriv`.
- `myfirst_poll`: the entry check.
- Every callout callback: `if (!sc->is_attached) return;`
- Every task callback’s equivalent check (if any).
- `myfirst_tick_source` after acquiring the mutex (this one is under the mutex; it could be an atomic load but does not have to be).

The mutex-held checks inside `myfirst_wait_data`, `myfirst_wait_room`, and the blocking re-checks after cv wakeup stay as-is: they are already serialised by the mutex.

The detach write becomes:

```
MYFIRST_LOCK(sc);
if (sc->active_fhs > 0) {
    MYFIRST_UNLOCK(sc);
```

```

        return (EBUSY);
    }
    atomic_store_rel_int(&sc->is_attached, 0);
    cv_broadcast(&sc->data_cv);
    cv_broadcast(&sc->room_cv);
    MYFIRST_UNLOCK(sc);

```

The store-release pairs with the atomic-load-acquire reads in the other contexts. Any state change that happened before the store (for example, any prior shutdown preparation) is visible to any thread that later does an acquire-read.

The handler entry checks become:

```

if (sc == NULL || atomic_load_acq_int(&sc->is_attached) == 0)
    return (ENXIO);

```

For the callout callbacks the check was under the mutex; we leave it under the mutex for consistency with the rest of the callback's serialisation. Some drivers convert even the callout check to an atomic read for performance; the Chapter 15 driver does not, because the mutex cost is negligible at the callout firing rate.

15.6.5 The Callout-to-Task Handshake

A different cross-component coordination problem. Suppose the watchdog callout detects a stall and wants to trigger a recovery action in a task. The callout cannot do the recovery itself (it might sleep, call user-space, etc.). The current driver solves this by enqueueing a task from the callout. What it does not solve is “do not enqueue the task if the previous recovery is still in progress”.

A small state flag solves it. Add to the softc:

```

int recovery_in_progress;    /* 0 or 1; protected by sc->mtx */

```

The callout:

```

static void
myfirst_watchdog(void *arg)
{
    struct myfirst_softc *sc = arg;
    /* ... existing watchdog logic ... */

    if (stall_detected && !sc->recovery_in_progress) {
        sc->recovery_in_progress = 1;
        taskqueue_enqueue(sc->tq, &sc->recovery_task);
    }
}

```

```

        /* ... re-arm as before ... */
    }

```

The task:

```

static void
myfirst_recovery_task(void *arg, int pending)
{
    struct myfirst_softc *sc = arg;

    /* ... recovery work ... */

    MYFIRST_LOCK(sc);
    sc->recovery_in_progress = 0;
    MYFIRST_UNLOCK(sc);
}

```

The flag is protected by the mutex (both writes happen under the mutex; the read in the callout happens under the mutex because callouts hold the mutex via `callout_init_mtx`). The invariant “at most one recovery task at a time” is preserved. A watchdog firing during recovery sees the flag set and does not enqueue.

This is a minimal example, but the pattern generalises. Any time the driver needs to coordinate “do X only if Y is not already happening”, a state flag protected by an appropriate lock is the right tool.

15.6.6 The Stage 4 Softc

Pulling it together. Stage 4 adds these fields:

```

/* Semaphore and its diagnostic fields (from Stage 1). */
struct sema    writers_sema;
int            writers_limit;
int            writers_trywait_failures;

/* Stats cache (from Stage 2). */
struct sx      stats_cache_sx;
uint64_t       stats_cache_bytes_10s;
uint64_t       stats_cache_last_refresh_ticks;

/* Recovery coordination (new in Stage 4). */
int            recovery_in_progress;
struct task    recovery_task;
int            recovery_task_runs;

```

All three fields form a coherent subsystem-coordination substrate. Section 6 encapsulates the vocabulary. Section 8 ships the final version.

15.6.7 Memory Ordering in One Subsection

Memory ordering can feel abstract; a concrete summary helps.

On strongly-ordered architectures (x86, amd64), plain loads and stores of aligned int-sized values are atomic with respect to other aligned int-sized values. A plain `int flag = 0` write is visible to all other CPUs promptly. You rarely need barriers.

On weakly-ordered architectures (arm64, riscv, powerpc), the compiler and CPU are free to reorder loads and stores as long as the sequence would appear correct to a single thread. A plain write by one CPU may be delayed from being visible to another CPU, and reads on the other CPU may be reordered relative to each other.

The `atomic(9)` API papers over the difference. `atomic_store_rel_int` and `atomic_load_acq_int` produce the correct barriers on every architecture. You do not need to know which architecture is weak or strong; you use the API and the right thing happens.

For the Chapter 15 driver, using `atomic_store_rel_int` on the detach write and `atomic_load_acq_int` on the entry checks gives you a driver that works correctly on both x86 and arm64. If the driver ever ships on arm64 systems (and FreeBSD 14.3 supports arm64 well), the discipline pays off.

15.6.8 Wrapping Up Section 5

Cross-component coordination in a driver uses the same primitives as local synchronisation, composed. The atomic API covers cheap shutdown flags with correct memory ordering. State flags protected by the appropriate lock coordinate “at most one” invariants across callouts, tasks, and user threads. Stage 4 of the Chapter 15 driver added both patterns. Section 6 takes the next step and encapsulates the synchronisation vocabulary in a dedicated header.

15.7 Section 6: Synchronization and Modular Design

The driver now uses five kinds of synchronisation primitives: a mutex, two condition variables, two sx locks, a counting semaphore, and atomic operations. Each one appears in several places across the source. A maintainer reading the file for the first time must reconstruct the synchronisation strategy from the scattered call sites.

This section encapsulates the synchronisation vocabulary in a small header, `myfirst_sync.h`, that names every operation the driver performs. The header does not add new primitives; it gives the existing primitives readable names and documents their contracts in one place. Stage 4 of the Chapter 15 driver introduces the header and updates the main source to use it.

A note on status before we go further. The `myfirst_sync.h` wrapper is **a recommendation, not a FreeBSD convention**. Most drivers under `/usr/src/sys/dev` call `mtx_lock`, `sx_xlock`, `cv_wait`, and friends directly; they do not ship a private synchronisation header. If you skim the tree, you will not find a community expectation that every driver provides such a layer. What the FreeBSD community *does* expect is a clear, documented lock order and a `LOCKING.md`-style comment block that a reviewer can follow, and that expectation we satisfy in every chapter of Part 3. The wrapper header is a stylistic extension that has worked well for several medium-sized drivers inside and outside the tree; it is valuable for this book because it turns the synchronisation vocabulary into something you can name, audit, and change in one place. If your future driver does not need the extra readability, skipping the header and keeping the primitive calls in the source is a perfectly normal choice. The

underlying discipline, lock order, drain on detach, explicit contracts, is what matters; the wrapper is one way to keep that discipline visible, not the only way.

15.7.1 Why Encapsulate

Three concrete benefits.

Readability. A code path that reads `myfirst_sync_writer_enter(sc)` tells the reader exactly what the call does. The same code path written as `if (ioflag & IO_NDELAY) { if (!sema_trywait(&sc->writers_sema)) ...` is correct but tells the reader less.

Changeability. If the synchronisation strategy changes (say, the writer-cap semaphore is replaced by an interruptible cv-based wait from Section 4), the change happens in one place in the header. The call sites in `myfirst_write` do not change.

Verifiability. The header is the single place where the synchronisation contracts are documented. A code review can verify “does every enter have a matching leave?” by grepping the header. Without the header the review must walk every call site.

The cost of encapsulation is minimal. A header of 100 to 200 lines. A half-hour of refactoring. A slight layer of indirection that a modern compiler inlines away.

15.7.2 The Shape of `myfirst_sync.h`

The header names every synchronisation operation. It does not define new structures; the structures stay in the softc. It provides inline functions or macros that wrap the primitives.

A sketch:

```
#ifndef MYFIRST_SYNC_H
#define MYFIRST_SYNC_H

#include <sys/lock.h>
#include <sys/mutex.h>
#include <sys/sx.h>
#include <sys/sema.h>
#include <sys/condvar.h>

struct myfirst_softc;      /* Forward declaration. */

/* Data-path mutex operations. */
static __inline void      myfirst_sync_lock(struct myfirst_softc *sc);
static __inline void      myfirst_sync_unlock(struct myfirst_softc *sc);
static __inline void      myfirst_sync_assert_locked(struct myfirst_softc *sc);

/* Configuration sx operations. */
static __inline void      myfirst_sync_cfg_read_begin(struct myfirst_softc *sc);
```

```

static __inline void    myfirst_sync_cfg_read_end(struct myfirst_softc *sc);
static __inline void    myfirst_sync_cfg_write_begin(struct myfirst_softc *sc);
static __inline void    myfirst_sync_cfg_write_end(struct myfirst_softc *sc);

/* Writer-cap semaphore operations. */
static __inline int     myfirst_sync_writer_enter(struct myfirst_softc *sc,
                                                int ioflag);
static __inline void    myfirst_sync_writer_leave(struct myfirst_softc *sc);

/* Stats cache sx operations. */
static __inline void    myfirst_sync_stats_cache_read_begin(
                        struct myfirst_softc *sc);
static __inline void    myfirst_sync_stats_cache_read_end(
                        struct myfirst_softc *sc);
static __inline int     myfirst_sync_stats_cache_try_promote(
                        struct myfirst_softc *sc);
static __inline void    myfirst_sync_stats_cache_downgrade(
                        struct myfirst_softc *sc);
static __inline void    myfirst_sync_stats_cache_write_begin(
                        struct myfirst_softc *sc);
static __inline void    myfirst_sync_stats_cache_write_end(
                        struct myfirst_softc *sc);

/* Attach-flag atomic operations. */
static __inline int     myfirst_sync_is_attached(struct myfirst_softc *sc);
static __inline void    myfirst_sync_mark_detaching(struct myfirst_softc *sc);

#endif /* MYFIRST_SYNC_H */

```

Each function wraps exactly one primitive call plus any convention the call site needs. For example, `myfirst_sync_writer_enter` takes the `ioflag` parameter and picks between `sema_trywait` (for `IO_NDELAY`) and `sema_wait`. The caller does not have to know the trywait-vs-wait logic; the header does.

15.7.3 The Implementation

Each function is a simple inline wrapper. Example implementations (for the most interesting ones):

```

static __inline void
myfirst_sync_lock(struct myfirst_softc *sc)
{
    mtx_lock(&sc->mtx);
}

static __inline void

```

```
myfirst_sync_unlock(struct myfirst_softc *sc)
{
    mtx_unlock(&sc->mtx);
}

static __inline void
myfirst_sync_assert_locked(struct myfirst_softc *sc)
{
    mtx_assert(&sc->mtx, MA_OWNED);
}

static __inline int
myfirst_sync_writer_enter(struct myfirst_softc *sc, int ioflag)
{
    if (ioflag & IO_NDELAY) {
        if (!sema_trywait(&sc->writers_sema)) {
            mtx_lock(&sc->mtx);
            sc->writers_trywait_failures++;
            mtx_unlock(&sc->mtx);
            return (EAGAIN);
        }
    } else {
        sema_wait(&sc->writers_sema);
    }
    if (!myfirst_sync_is_attached(sc)) {
        sema_post(&sc->writers_sema);
        return (ENXIO);
    }
    return (0);
}

static __inline void
myfirst_sync_writer_leave(struct myfirst_softc *sc)
{
    sema_post(&sc->writers_sema);
}

static __inline int
myfirst_sync_is_attached(struct myfirst_softc *sc)
{
    return (atomic_load_acq_int(&sc->is_attached));
}

static __inline void
```

```
myfirst_sync_mark_detaching(struct myfirst_softc *sc)
{
    atomic_store_rel_int(&sc->is_attached, 0);
}
```

The `writer_enter` wrapper is the most complex; everything else is a one-liner. A header of this shape produces zero runtime overhead (the compiler inlines every call) and adds substantial readability.

15.7.4 How the Source Changes

Every `mtx_lock(&sc->mtx)` in the main source becomes `myfirst_sync_lock(sc)`. Every `sema_wait(&sc->writers_sema)` becomes `myfirst_sync_writer_enter(sc, ioflag)` or a variant. Every `atomic_load_acq_int(&sc->is_attached)` becomes `myfirst_sync_is_attached(sc)`.

The main source reads more clearly:

```
/* Before: */
if (ioflag & IO_NDELAY) {
    if (!sema_trywait(&sc->writers_sema)) {
        MYFIRST_LOCK(sc);
        sc->writers_trywait_failures++;
        MYFIRST_UNLOCK(sc);
        return (EAGAIN);
    }
} else {
    sema_wait(&sc->writers_sema);
}
if (!sc->is_attached) {
    sema_post(&sc->writers_sema);
    return (ENXIO);
}

/* After: */
error = myfirst_sync_writer_enter(sc, ioflag);
if (error != 0)
    return (error);
```

Five lines of intent become one. A reader who wants to know what `myfirst_sync_writer_enter` does opens the header and reads the implementation. A reader who accepts the interface reads on.

15.7.5 Naming Conventions

A short discipline for picking names in a synchronisation wrapper layer.

Name operations, not primitives. `myfirst_sync_writer_enter` describes what the caller is doing (entering the writer section). `myfirst_sync_sema_wait` would describe the primitive (calling `sema_wait`), which is less useful.

Use enter/leave pairs for scoped acquisition. Every `enter` has a matching `leave`. This makes it visually obvious whether the driver always releases what it acquires.

Use read/write pairs for shared/exclusive access. `cfg_read_begin/cfg_read_end` for shared; `cfg_write_begin/cfg_write_end` for exclusive. The `begin/end` suffix mirrors the call-site structure.

Use `is_` for predicates that return bool-like values. `myfirst_sync_is_attached` reads like English.

Use `mark_` for atomic state transitions. `myfirst_sync_mark_detaching` describes the transition.

15.7.6 What Not to Put in the Header

The header should wrap synchronisation primitives, not business logic. A function that acquires a lock and also does “interesting” work should stay in the main source; only the pure lock manipulation belongs in the header.

The header also should not hide important details. For example, `myfirst_sync_writer_enter` returns `EAGAIN` or `ENXIO` or `0`; the caller must check. A header that silently “returned” on `ENXIO` would hide an important error path. The wrapper’s contract must be explicit.

15.7.7 A Related Discipline: Assertions

The header is a good place to put the assertions that document invariants. A function that must be called under the mutex can call `myfirst_sync_assert_locked(sc)` at entry:

```
static void
myfirst_some_helper(struct myfirst_softc *sc)
{
    myfirst_sync_assert_locked(sc);
    /* ... */
}
```

On a debug kernel (with `INVARIANTS`) the assertion fires if the helper is ever called without the mutex. On a production kernel the assertion is elided.

The Chapter 14 code uses `MYFIRST_ASSERT`; the Chapter 15 refactor keeps this as `myfirst_sync_assert_locked` with the same behaviour.

15.7.8 A Short WITNESS Walkthrough: Sleeping Under a Non-Sleepable Lock

Chapter 34 walks through a lock-order reversal between two mutexes. A separate class of `WITNESS` warning, equally common and just as cheap to prevent, deserves a short mention here because it lands squarely in the middle of Chapter 15’s territory: the interaction between mutexes and `sx` locks.

Imagine a first-attempt refactor of the configuration read path. The author has just added a new `sx_slock` over the config blob for read-mostly access, and, without thinking, calls it from inside a code path that still holds the data-path mutex:

```
static int
myfirst_read(struct cdev *dev, struct uio *uio, int ioflag)
{
    struct myfirst_softc *sc = dev->si_drv1;
    int error;

    MYFIRST_LOCK(sc);           /* mtx_lock: non-sleepable */
    sx_slock(&sc->cfg_sx);      /* sx_slock: sleepable */
    error = myfirst_copy_out_locked(sc, uio);
    sx_sunlock(&sc->cfg_sx);
    MYFIRST_UNLOCK(sc);
    return (error);
}
```

The code compiles and, in light testing on a non-debug kernel, appears to run correctly. Load it on a kernel built with options `WITNESS` and run the same lab, and the console reports something close to this:

```
lock order reversal: (sleepable after non-sleepable)
 1st 0xfffff800... myfirst_sc_mtx (mutex) @ /usr/src/sys/modules/myfirst/myfirst.c:...
 2nd 0xfffff800... myfirst_cfg_sx (sx) @ /usr/src/sys/modules/myfirst/myfirst.c:...
stack backtrace:
 #0 witness_checkorder+0x...
 #1 _sx_slock+0x...
 #2 myfirst_read+0x...
```

Two things are worth reading carefully in this report. The parenthetical “(sleepable after non-sleepable)” tells you exactly what the reversal is: the thread took a non-sleepable lock first (the mutex) and then asked for a sleepable one (the `sx` lock). `WITNESS` rejects this because `sx_slock` can sleep, and sleeping while a non-sleepable lock is held is a defined kernel bug class: the scheduler cannot take the thread off-CPU without also migrating the mutex’s waiters, and the invariants that make `MTX_DEF` cheap stop holding. The second thing is that `WITNESS` reports it the very first time the path runs, long before any real contention. You do not have to reproduce a race; the warning fires on the ordering itself.

The fix is ordering discipline, not a different primitive. Take the `sx` lock first, then the mutex:

```
sx_slock(&sc->cfg_sx);
MYFIRST_LOCK(sc);
error = myfirst_copy_out_locked(sc, uio);
MYFIRST_UNLOCK(sc);
sx_sunlock(&sc->cfg_sx);
```

Or, better for most drivers, read the config under the sx lock into a local snapshot and release the sx lock before touching the data-path mutex at all. The encapsulation in `myfirst_sync.h` helps here because the lock-order contract is named and documented in one place; a review that sees `myfirst_sync_cfg_slock` followed by `myfirst_sync_lock` can confirm the ordering at a glance.

This walkthrough deliberately stays shorter than Chapter 34's. The category of bug is different, and the lesson is specific to the sleepable/non-sleepable distinction Chapter 15 is built around. The broader lock-order reversal walkthrough belongs to the debugging chapter; this one belongs where the reader first composes sx locks and mutexes in the same code path.

15.7.9 Wrapping Up Section 6

A small synchronisation header names every operation the driver performs and centralises the contracts. The main source reads more clearly; the header is the one place a maintainer looks to understand or change the strategy. Stage 4's `myfirst_sync.h` adds no new primitives; it encapsulates the ones from Sections 2 through 5. Section 7 writes the tests that validate the whole composition.

15.8 Section 7: Testing Advanced Synchronization

Every primitive this chapter introduced has a failure mode. A semaphore with a missing `sema_post` leaks slots. An sx upgrade that does not re-check the predicate refreshes redundantly. A signal-interruptible wait that ignores EINTR deadlocks the caller. A state flag read without the right lock or atomic discipline silently reads stale values.

This section is about writing tests that surface those failure modes before users find them. The tests are not unit tests in the pure sense; they are stress harnesses that exercise the driver under concurrent load and check invariants. The Chapter 15 companion source includes three test programs; this section walks through each.

15.8.1 Why Stress Tests Matter

Synchronisation bugs rarely show up in single-threaded testing. A forgotten `sema_post` is invisible until enough writers have passed through that the semaphore runs out. A misplaced atomic read is invisible until a specific interleaving occurs. A detach race is invisible until detach and unload happen with real concurrency.

Stress tests find these bugs by running the driver in configurations that expose the interleavings. Many concurrent readers and writers. Fast tick source. Frequent detach/reload cycles. Simultaneous `sysctl` writes. The harder the driver works, the more likely a latent bug is to surface.

The tests do not replace `WITNESS` or `INVARIANTS`. `WITNESS` catches lock-order violations at any load. `INVARIANTS` catches structural violations. The stress tests catch logic errors that neither static nor lightweight dynamic checks can detect.

15.8.2 Test 1: Writer-Cap Correctness

The writer-cap semaphore's invariant is "at most `writers_limit` writers in `myfirst_write` at once". A test program starts many concurrent writers, each of which writes a few bytes, records its process ID in a tiny marker

at the start of its writes, and proceeds. A monitor process reads the cbuf in the background and counts concurrent markers.

The test is in `examples/part-03/ch15-more-synchronization/tests/writer_cap_test.c`:

```
/*
 * writer_cap_test: start N writers and verify no more than
 * writers_limit are simultaneously inside the write path.
 */
#include <sys/param.h>
#include <sys/time.h>
#include <err.h>
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/sysctl.h>
#include <unistd.h>

#define N_WRITERS 16

int
main(int argc, char **argv)
{
    int fd, i;
    char buf[64];
    int writers = (argc > 1) ? atoi(argv[1]) : N_WRITERS;

    for (i = 0; i < writers; i++) {
        if (fork() == 0) {
            fd = open("/dev/myfirst", O_WRONLY);
            if (fd < 0)
                err(1, "open");
            snprintf(buf, sizeof(buf), "w%d\n", i);
            for (int j = 0; j < 100; j++) {
                write(fd, buf, strlen(buf));
                usleep(1000);
            }
            close(fd);
            _exit(0);
        }
    }
    while (wait(NULL) > 0)
        ;
}
```



```
    return (0);  
}
```

The test fires `N_WRITERS` processes, each of which writes 100 short messages with a 1 ms delay. A reader process reads `/dev/myfirst` and observes interleaving.

A simple invariant check: the reader reads 100 bytes at a time and records how many distinct writer prefixes appear in that window. If `writers_limit` is 4, the reader should see at most 4 prefixes in any 100-byte window (plus or minus). More than 4 indicates the cap is not being enforced.

A more rigorous check uses `sysctl dev.myfirst.0.stats.writers_trywait_failures` to observe the failure rate in the `O_NONBLOCK` mode. If you set `writers_limit=2` and run 16 non-blocking writers, most of them should see `EAGAIN`; the failure count should grow rapidly.

15.8.3 Test 2: Stats Cache Concurrency

The stats cache invariant is “many readers concurrently, refreshed at most once per second”. A test:

- Start 32 concurrent reader processes, each reading `dev.myfirst.0.stats.bytes_written_10s` in a tight loop.
- Start 1 writer that keeps writing to the device.
- Watch the cache refresh rate via `DTrace`:

```
# dtrace -n '  
fbt::myfirst_stats_cache_refresh:entry {  
    @["refreshes"] = count();  
}  
tick-10s { printa(@); exit(0); }  
,
```

Expected: approximately 10 refreshes per 10 seconds (one per stale expiry). Not 32 per 10 seconds; the cache eliminates the per-reader recomputation.

If the refresh rate spikes under contention, the `sx_try_upgrade` fast path is failing too often and the fallback (drop-and-reacquire) is introducing races. The driver code should handle this correctly; if it does not, the test exposes the bug.

15.8.4 Test 3: Detach Under Load

The detach invariant is “detach completes cleanly even when every Chapter 14 and 15 primitive is under load”. A detach test:

```
# ./stress_all.sh &  
# STRESS_PID=$!  
# sleep 5
```

```
# kldunload myfirst
# kill -TERM $STRESS_PID
```

Where `stress_all.sh` runs:

- Several concurrent writers.
- Several concurrent readers.
- The tick source enabled at 1 ms.
- The heartbeat enabled at 100 ms.
- The watchdog enabled at 1 s.
- Occasional `bulk_writer_flood` sysctl writes.
- Occasional `writers_limit` sysctl adjustments.
- Occasional stats-cache reads.

The detach should complete. If it hangs or panics, the ordering discipline has a bug. With the Chapter 14 and 15 code correctly ordered, the test should pass reliably.

15.8.5 Observing With DTrace

DTrace is the scalpel for synchronisation debugging. A few useful one-liners:

Count cv wakeups per cv name.

```
# dtrace -n 'fbt::cv_signal:entry { @[stringof(arg0)] = count(); }'
```

Interpreting the output requires knowing your cv names; the driver's are "myfirst_data" and "myfirst_room".

Count semaphore operations.

```
# dtrace -n '
  fbt::_sema_wait:entry { @[probecount] = count(); }
  fbt::_sema_post:entry { @[probecount] = count(); }
'
```

In a balanced driver the post count equals the wait count over a long run (plus the initial value from `sema_init`).

Observe task firings.

```
# dtrace -n 'fbt::taskqueue_run_locked:entry { @[execname] = count(); }'
```

Shows which taskqueue threads are running, which is useful for confirming the private taskqueue is getting work.

Observe callout firings.

```
# dtrace -n 'fbt::callout_reset:entry { @[probefunc] = count(); }'
```

Shows how often callouts are being re-armed, which should match the configured interval rate.

Run each one-liner under your stress workload. The counts should match your expectations. Unexpected imbalance is a debugging starting point.

15.8.6 WITNESS, INVARIANTS, and the Debug Kernel

The first line of defence remains the debug kernel. If WITNESS is unhappy about a lock order, fix the order before shipping. If INVARIANTS asserts in `cbuf_*` or `sema_*`, fix the caller before shipping. These checks are cheap; running without them in development is a false economy.

A few WITNESS outputs to expect or avoid:

- **“acquiring duplicate lock of same type”**: you acquired a lock you already hold, probably by accident. Review the call path.
- **“lock order reversal”**: two locks were acquired in different orders on different paths. Pick one order, enforce it, update `LOCKING.md`.
- **“blockable sleep from an invalid context”**: you called something that can sleep from a context where sleeping is not allowed. Check whether the context is a callout or interrupt.

Every WITNESS warning in `dmesg` from your driver is a bug. Treat them as panic-equivalent; fix each one.

15.8.7 Regression Discipline

After every Chapter 15 stage, run:

1. The Chapter 11 IO smoke tests (basic read, write, open, close).
2. The Chapter 12 synchronisation tests (bounded reads, sx-protected config).
3. The Chapter 13 timer tests (heartbeat, watchdog, tick source at various rates).
4. The Chapter 14 taskqueue tests (poll waiter, bulk writer flood, delayed reset).
5. The Chapter 15 tests (writer-cap, stats cache, detach under load).

The whole suite should pass on a debug kernel. If any test fails, the regression is recent; roll back to the previous stage, find the delta, and debug.

15.8.8 Wrapping Up Section 7

Advanced synchronisation needs advanced testing. Stress tests that exercise concurrent writers, readers, and sysctls surface bugs that single-threaded testing misses. DTrace makes the internals of the synchronisation primitives observable. WITNESS and INVARIANTS catch what remains. Running the whole stack under a debug kernel is the closest thing to “good enough” testing that a driver can get. Section 8 closes Part 3.

15.9 Section 8: Refactoring and Versioning Your Coordinated Driver

Stage 4 is the consolidation stage for Chapter 15. It does not add new functionality; it reorganises and documents the Chapter 15 additions, updates `LOCKING.md`, bumps the version to 0.9-coordination, and runs the full regression suite across Chapters 11 through 15.

This section walks through the consolidation, extends `LOCKING.md` with the Chapter 15 sections, and closes Part 3.

15.9.1 File Organisation

The Chapter 15 refactor introduces `myfirst_sync.h`. The file list becomes:

- `myfirst.c`: the main driver source.
- `cbuf.c`, `cbuf.h`: unchanged from Chapter 13.
- `myfirst_sync.h`: new header with synchronisation wrappers.
- `Makefile`: unchanged except for the addition of `myfirst_sync.h` to the headers the source depends on (for make dependency tracking if needed).

Inside `myfirst.c`, the Chapter 15 organisation follows the same pattern as Chapter 14 with a few additions:

1. Includes (now includes `myfirst_sync.h`).
2. `Softc` structure (extended with Chapter 15 fields).
3. File-handle structure (unchanged).
4. `cdevsw` declaration (unchanged).
5. Buffer helpers (unchanged).
6. Cache helpers (new; `myfirst_stats_cache_refresh`).
7. Condition-variable wait helpers (revised with explicit `EINTR/ERESTART` handling).
8. `Sysctl` handlers (extended with `writers_limit`, `stats_cache`, `recovery`).
9. Callout callbacks (revised to use `atomic_load_acq_int` for `is_attached` where appropriate).
10. Task callbacks (extended with `recovery_task`).
11. `Cdev` handlers (revised to use `myfirst_sync_*` wrappers).
12. Device methods (`attach/detach` extended with `sema`, `sx`, and atomic-flag discipline).
13. Module glue (version bump).

The key change across every section is the use of `myfirst_sync.h` wrappers where the Chapter 14 code used direct primitive calls. This is visible in `attach`, `detach`, and every handler.

15.9.2 The `LOCKING.md` Update

The Chapter 14 `LOCKING.md` had sections for the mutex, cvs, sx, callouts, and tasks. Chapter 15 adds Semaphores, Coordination, and an updated Lock Order section.

Semaphores

The driver owns one counting semaphore:

- `writers_sema`: caps concurrent writers at `sc->writers_limit`.
Default limit: 4. Range: 1-64. Configurable via the
`dev.myfirst.N.writers_limit` sysctl.

Semaphore operations happen outside `sc->mtx`. The internal `sema_mtx` is not in the documented lock order because it does not conflict with `sc->mtx`; the driver never holds `sc->mtx` across a `sema_wait` or `sema_post`.

Lowering `writers_limit` below the current semaphore value is best-effort: the handler lowers the target and lets new entries observe the lower cap as the value drains. Raising posts additional slots immediately.

Sema Drain Discipline

The driver tracks `writers_inflight` as an atomic int. It is incremented before any `sema_*` call (specifically at the top of `myfirst_sync_writer_enter`) and decremented after the matching `sema_post` (in `myfirst_sync_writer_leave` or on every error return).

Detach waits for `writers_inflight` to reach zero before calling `sema_destroy`. This closes the use-after-free race where a woken waiter is between `cv_wait` return and its final `mtx_unlock(&sema->sema_mtx)` when `sema_destroy` tears down the internal mutex.

`sema_destroy` itself is called only after:

1. `is_attached` has been cleared atomically.
2. `writers_limit` wake-up slots have been posted to the sema.
3. `writers_inflight` has been observed to reach zero.
4. Every callout, task, and selinfo has been drained.

Coordination

The driver uses three cross-component coordination mechanisms:

1. **Atomic is_attached flag.** Read via `atomic_load_acq_int`, written via `atomic_store_rel_int` in detach. Allows every context (callout, task, user thread) to check shutdown state without acquiring `sc->mtx`.

2. ****recovery_in_progress state flag.**** Protected by ``sc->mtx``. Set by the watchdog callout, cleared by the recovery task. Ensures at most one recovery task is pending or running at a time.
3. ****Stats cache sx.**** Shared reads, occasional upgrade-promote-downgrade for refresh. See the Stats Cache section.

Stats Cache

The ``stats_cache_sx`` protects a small cached statistic. The refresh pattern is:

```

```c
sx_slock(&sc->stats_cache_sx);
if (stale) {
 if (sx_try_upgrade(&sc->stats_cache_sx)) {
 refresh();
 sx_downgrade(&sc->stats_cache_sx);
 } else {
 sx_sunlock(&sc->stats_cache_sx);
 sx_xlock(&sc->stats_cache_sx);
 if (still_stale)
 refresh();
 sx_downgrade(&sc->stats_cache_sx);
 }
}
value = sc->stats_cache_bytes_10s;
sx_sunlock(&sc->stats_cache_sx);
```text

```

Lock Order

The complete driver lock order is:

```

```text
sc->mtx -> sc->cfg_sx -> sc->stats_cache_sx
```text

```

``WITNESS`` enforces this order. The writer-cap semaphore's **internal** mutex is not in the graph because the driver never holds ``sc->mtx`` (or any other driver lock) across a ``sema_wait``/``sema_post`` call.

Detach Ordering (updated)

1. Refuse detach **if** ``sc->active_fhs > 0``.

```

2. Clear `sc->is_attached` under `sc->mtx` via
   `atomic_store_rel_int`.
3. `cv_broadcast(&sc->data_cv)`; `cv_broadcast(&sc->room_cv)`.
4. Release `sc->mtx`.
5. Post `writers_limit` wake-up slots to `writers_sema`.
6. Wait for `writers_inflight == 0` (sema drain).
7. Drain the three callouts.
8. Drain every task including recovery_task.
9. `seldrain(&sc->rsl)`; `seldrain(&sc->wsl)`.
10. `taskqueue_free(sc->tq)`; `sc->tq = NULL`.
11. `sema_destroy(&sc->writers_sema)` (safe: drain completed).
12. `sx_destroy(&sc->stats_cache_sx)`.
13. Destroy cdev, free sysctl context, destroy cbuf, counters,
    cvs, cfg_sx, mtx.

```

15.9.3 The Version Bump

The version string advances from 0.8-taskqueues to 0.9-coordination:

```
#define MYFIRST_VERSION "0.9-coordination"
```

And the driver's probe string:

```
device_set_desc(dev, "My First FreeBSD Driver (Chapter 15 Stage 4)");
```

15.9.4 The Final Regression Pass

The Chapter 15 regression suite adds its own tests but also re-runs every earlier chapter's tests. A compact ordering:

1. **Build cleanly** on a debug kernel with all the usual options enabled.
2. **Load** the driver.
3. **Chapter 11 tests**: basic read, write, open, close, reset.
4. **Chapter 12 tests**: bounded reads, timed reads, cv broadcasts, sx config.
5. **Chapter 13 tests**: callouts at various rates, watchdog detection, tick source.
6. **Chapter 14 tests**: poll waiter, coalescing flood, delayed reset, detach under load.
7. **Chapter 15 tests**: writer-cap correctness, stats cache concurrency, signal interruption with partial progress, detach under full load.
8. **WITNESS pass**: every test, zero warnings in dmesg.
9. **DTrace verification**: wakeup counts and task firings match expectations.
10. **Long-duration stress**: hours of load with periodic detach-reload cycles.

Every test passes. Every WITNESS warning is resolved. The driver is at 0.9-coordination and Part 3 is done.

15.9.5 Documentation Audit

A final documentation pass.

- `myfirst.c` top-of-file comment is updated with the Chapter 15 vocabulary.
- `myfirst_sync.h` has a top-of-file comment summarising the design.
- `LOCKING.md` has the Semaphores, Coordination, and Stats Cache sections.
- The chapter's `README.md` under `examples/part-03/ch15-more-synchronization/` describes each stage.
- Every `sysctl` has a description string.

Updating documentation feels like overhead. It is the difference between a driver the next maintainer can change and a driver they have to rewrite.

15.9.6 A Final Audit Checklist

- ☐ Does every `sema_wait` have a matching `sema_post`?
- ☐ Does every `sx_slock` have a matching `sx_sunlock`?
- ☐ Does every `sx_xlock` have a matching `sx_xunlock`?
- ☐ Does every atomic read use `atomic_load_acq_int` and every atomic write use `atomic_store_rel_int` where ordering matters?
- ☐ Does every `cv_*_sig` call handle `EINTR`, `ERESTART`, and `EWOULDBLOCK` explicitly?
- ☐ Does every blocking-wait caller record and check partial progress?
- ☐ Is every synchronisation primitive wrapped in `myfirst_sync.h`?
- ☐ Does every primitive appear in `LOCKING.md`?
- ☐ Is the detach ordering in `LOCKING.md` accurate?
- ☐ Does the driver pass the full regression suite?

A driver that answers every item cleanly is a driver you can hand to another engineer with confidence.

15.9.7 Wrapping Up Section 8

Stage 4 consolidates. The header is in place, `LOCKING.md` is current, the version reflects the new capability, the regression suite passes, the audit is clean. The driver is `0.9-coordination` and the synchronisation story is complete.

Part 3 is complete as well. Five chapters, five primitives added one at a time, each composed with what came before. The Wrapping Up section after the labs and challenges frames what Part 3 accomplished and how Part 4 will use it.

15.10 Additional Topics: Atomic Operations, epoch(9) Revisited, and Memory Ordering

The main body of Chapter 15 taught the essentials. Three adjacent topics deserve a somewhat deeper mention because they recur in real driver code and because the underlying ideas round out the synchronisation story.

15.10.1 Atomic Operations in More Depth

The Chapter 15 driver used three atomic primitives: `atomic_load_acq_int`, `atomic_store_rel_int`, and implicitly `atomic_fetchadd_int` via `counter(9)`. The `atomic(9)` family is larger and more structured than these three operations suggest.

Read-modify-write primitives.

- `atomic_fetchadd_int(p, v)`: returns the old value of `*p`, sets `*p += v`. Useful for free-running counters.
- `atomic_cmpset_int(p, old, new)`: if `*p == old`, sets `*p = new` and returns 1; otherwise returns 0 and does not modify. The classic compare-and-swap. Useful for implementing lock-free state machines.
- `atomic_cmpset_acq_int`, `atomic_cmpset_rel_int`: variants with acquire or release semantics.
- `atomic_readandclear_int(p)`: returns the old value and sets `*p = 0`. Useful for “take the current value, reset”.
- `atomic_set_int(p, v)`: sets bits `*p |= v`. Useful for flag-set coordination.
- `atomic_clear_int(p)`: clears bits `*p &= ~v`. Useful for flag-clear coordination.
- `atomic_swap_int(p, v)`: returns old `*p`, sets `*p = v`. Useful for taking ownership of a pointer.

Width variants. `atomic_load_int`, `atomic_load_long`, `atomic_load_32`, `atomic_load_64`, `atomic_load_ptr`. The integer size in the name matches the C type. Use the one that matches your variable’s type.

Barrier variants. `atomic_thread_fence_acq`, `atomic_thread_fence_rel`, `atomic_thread_fence_acq_rel`, `atomic_thread_fence_seq_cst`. Pure barriers that order preceding and following memory accesses without atomically modifying a specific location. Occasionally useful.

Picking the right variant is a small discipline. For a flag that readers poll, use `atomic_load_acq`. For a writer that commits a flag after preceding setup, use `atomic_store_rel`. For a counter that is free-running and readers never synchronise on exactly, use `atomic_fetchadd` (no barriers). For a CAS-based state machine, use `atomic_cmpset`.

15.10.2 epoch(9) In One Page

Chapter 14 introduced `epoch(9)` briefly; here is a slightly deeper sketch, within the boundary of what a driver writer usefully knows.

An epoch is a short-lived synchronisation barrier that protects read-mostly data structures without locks. Code that reads shared data enters the epoch with `epoch_enter(epoch)` and leaves with `epoch_exit(epoch)`. The epoch guarantees that any object freed via `epoch_call(epoch, cb, ctx)` is not actually reclaimed until every reader that was inside the epoch when the free was requested has exited.

This is similar in spirit to Linux’s RCU (read-copy-update) but with different ergonomics. The FreeBSD epoch is a coarser tool; it protects large, rarely-changing structures like the `ifnet` list.

A driver that wants to use epoch typically does not create its own. It uses one of the kernel-provided epochs, most commonly `net_epoch_preempt` for network state. Driver writers outside of network code rarely reach for epoch directly.

What a driver writer should know is how to recognise the pattern in other code, and when a taskqueue’s `NET_TASK_INIT` is creating a task that runs inside an epoch. Section 14 covered this.

15.10.3 Memory Ordering, A Little Deeper

The atomic API hides the architecture-specific details of memory ordering. When you use `atomic_store_rel_int` and `atomic_load_acq_int` in matching pairs, the right barrier is inserted on each architecture. You do not need to know the details.

But a one-page intuition helps.

On x86, every load is an acquire-load and every store is a release-store, at the hardware level. The CPU's memory model is "total store order". So `atomic_load_acq_int` on x86 is just a plain `MOV` with no extra instructions. `atomic_store_rel_int` is a plain `MOV` too.

On arm64, loads and stores have weaker default ordering. The compiler inserts `LDAR` (load-acquire) for `atomic_load_acq_int` and `STLR` (store-release) for `atomic_store_rel_int`. These are cheap (a few cycles) but not free.

The implication: correctness-wise, you write the same code on both architectures. Performance-wise, x86 "pays" nothing for the barriers while arm64 pays a small cost. For a rare operation like a shutdown flag, the cost is negligible on both.

The further implication: testing on x86 alone is insufficient for validating memory ordering. Code that works on x86 with plain loads might deadlock or misbehave on arm64 if the atomic barriers were omitted. FreeBSD 14.3 supports arm64 well; drivers shipped to users who run on arm64 hardware need to be correct on the weaker memory model. Using the `atomic(9)` API consistently is how you guarantee this without thinking about architecture per-call.

15.10.4 When to Reach for Each Tool

A small decision tree to close the section.

- **Protect a small invariant read by many, written rarely?** `atomic_load_acq_int/atomic_store_rel_int`.
- **Protect a small invariant with an intricate relation to other state?** `Mutex`.
- **Wait for a predicate?** `Mutex` plus condition variable.
- **Wait for a predicate with signal handling?** `Mutex` plus `cv_wait_sig` or `cv_timedwait_sig`.
- **Admit at most N participants?** `sema(9)`.
- **Many readers or one writer with occasional promotion?** `sx(9)` with `sx_try_upgrade`.
- **Run code later in thread context?** `taskqueue(9)`.
- **Run code at a deadline?** `callout(9)`.
- **Coordinate shutdown across contexts?** Atomic flag + `cv` broadcast (for blocked waiters).
- **Protect a read-mostly structure in network code?** `epoch(9)`.

That decision tree is the mental map Part 3 has been building. Chapter 15 added the last few branches. Chapter 16 will apply the map to hardware.

15.10.5 Wrapping Up Additional Topics

Atomic operations, `epoch(9)`, and memory ordering round out the synchronisation toolkit. For most drivers the common case is a `mutex` plus a `cv` plus occasional atomics; the other primitives are there for specific shapes. Knowing the whole set lets you pick the right tool without the guesswork.

15.11 Hands-On Labs

Four labs apply the Chapter 15 material to concrete tasks. Allocate one session per lab. Labs 1 and 2 are the most important; Labs 3 and 4 stretch the reader.

15.11.1 Lab 1: Observe Writer-Cap Enforcement

Objective. Confirm that the writer-cap semaphore limits concurrent writers and that the limit is configurable at runtime.

Setup. Build and load the Stage 4 driver. Compile the `writer_cap_test` helper from the companion source.

Steps.

1. Verify the default limit: `sysctl dev.myfirst.0.writers_limit`. Should be 4.
2. Check the semaphore value: `sysctl dev.myfirst.0.stats.writers_sema_value`. Should also be 4 (no writers active).
3. Start sixteen blocking writers in the background:

```
# for i in $(jot 16 1); do
    (cat /dev/urandom | head -c 10000 > /dev/myfirst) &
done
```

4. Observe the semaphore value while they run. Should be near zero most of the time (all slots in use).
5. Lower the limit to 2:

```
# sysctl dev.myfirst.0.writers_limit=2
```

6. Observe that the semaphore value eventually drops to 0 and stays there (writers drain faster than they re-enter).
7. Raise the limit to 8:

```
# sysctl dev.myfirst.0.writers_limit=8
```

8. The driver posts four additional slots immediately; writers that were blocked in `sema_wait` wake up and enter.
9. Wait for all writers to finish; verify the final semaphore value equals the current limit.

Expected outcome. The semaphore acts as the admission controller; reconfiguring it at runtime reconfigures the limit. Under heavy load the sema drains to zero; when load eases, it refills to the configured limit.

Variation. Try with non-blocking writers using a helper that opens with `O_NONBLOCK`. Watch `sysctl dev.myfirst.0.stats.writers_trywait_failures` grow when the sema is exhausted.

15.11.2 Lab 2: Stats Cache Contention

Objective. Observe that the stats cache serves many reads with few refreshes under a read-mostly workload.

Setup. Stage 4 driver loaded. DTrace available.

Steps.

1. Start 32 concurrent reader processes, each reading the cached stat in a tight loop:

```
# for i in $(jot 32 1); do
    (while ;; do
        sysctl -n dev.myfirst.0.stats.bytes_written_10s >/dev/null
    done) &
done
```

2. In a separate terminal, observe the cache refresh rate via DTrace:

```
# dtrace -n 'fbt::myfirst_stats_cache_refresh:entry { @ = count(); }'
```

3. Let the workload run for 30 seconds, then Ctrl-C DTrace. Record the count.

Expected outcome. Approximately 30 refreshes over 30 seconds: one per second, regardless of how many readers are reading. If you see substantially more refreshes, the cache is being invalidated too aggressively; if substantially fewer, the readers are not actually triggering the stale path.

Variation. Run the test with a write workload in parallel (also using `/dev/myfirst`). The refresh rate should not change: the refresh is triggered by cache staleness, not by writes.

15.11.3 Lab 3: Signal Handling and Partial Progress

Objective. Confirm that a `read(2)` interrupted by a signal returns the bytes copied so far, not `EINTR`.

Setup. Stage 4 driver loaded. Stopped tick source, empty buffer.

Steps.

1. Start a reader that asks for 4096 bytes with no timeout:

```
# dd if=/dev/myfirst bs=4096 count=1 > /tmp/out 2>&1 &
# READER=$!
```

2. Enable the tick source at a slow rate:

```
# sysctl dev.myfirst.0.tick_source_interval_ms=500
```

The reader slowly accumulates bytes, one every 500 ms.

3. After about 2 seconds, send `SIGINT` to the reader:

```
# kill -INT $READER
```

4. `dd` reports the number of bytes copied before the signal.

Expected outcome. `dd` reports a partial result (say, 4 bytes copied out of 4096 requested) and exits 0 (partial success), not an error. The driver returned the partial byte count to the syscall layer; the syscall layer surfaced it as a normal short read.

Variation. Set `read_timeout_ms` to 1000 and repeat. The driver should return either a partial result (if any bytes arrived) or `EAGAIN` (if the timeout fired first with zero bytes). Signal handling should still preserve partial bytes.

15.11.4 Lab 4: Detach Under Maximum Load

Objective. Confirm that the detach ordering is correct under full concurrent load.

Setup. Stage 4 driver loaded. All the Chapter 14 and 15 stress tools compiled.

Steps.

1. Start the full stress kit:
 - 8 concurrent writers.
 - 4 concurrent readers.
 - Tick source at 1 ms.
 - Heartbeat at 100 ms.
 - Watchdog at 1 s.
 - Sysctl flood every 100 ms: `bulk_writer_flood=1000`.
 - Sysctl flood every 500 ms: adjust `writers_limit` between 1 and 8.
 - Concurrent sysctl reads of `stats.bytes_written_10s`.
2. Let the stress run for 30 seconds to ensure maximum load.
3. Unload the driver:

```
# kldunload myfirst
```

4. Kill the stress processes. Observe that the unload completed cleanly.

Expected outcome. The unload succeeds (not `EBUSY`, not a panic, no hang). All stress processes fail gracefully (open returning `ENXIO`, outstanding reads/writes returning `ENXIO` or short results). `dmesg` has no `WITNESS` warnings.

Variation. Repeat the cycle 20 times (load, stress, unload). Every cycle should behave identically. If one cycle panics or hangs, there is a race; investigate.

15.12 Challenge Exercises

The challenges go beyond the chapter's body. They are optional; each consolidates a specific Chapter 15 idea.

15.12.1 Challenge 1: Replace sema with Interruptible Wait

The writer-cap uses `sema_wait`, which is not signal-interruptible. Rewrite the write path's admission control to use `sema_trywait` plus an interruptible `cv_wait_sig` loop. Preserve the partial-progress convention.

Expected outcome: a writer blocked waiting for a slot can be `SIGINT`'d cleanly. Bonus: use the encapsulation in `myfirst_sync.h` so `myfirst_sync_writer_enter` has the same signature but different internals.

15.12.2 Challenge 2: Read-Mostly with a Background Refresher

The stats cache refreshes on-demand when a reader notices staleness. Change the design so the cache is refreshed by a periodic callout instead, and readers never trigger a refresh. Compare the resulting code to the upgrade-promote-downgrade pattern.

Expected outcome: understand when on-demand caching is better than background caching. Answer: on-demand is simpler (no callout lifetime to manage) but wastes a refresh on a cache that nobody reads. Background is more predictable but requires the extra primitive. Most drivers pick on-demand for small caches and background for large ones.

15.12.3 Challenge 3: Multiple Writer-Cap Semaphores

Imagine the driver is layered over a storage backend that has separate pools for different classes of IO: “small writes” and “large writes”. Add a second semaphore that caps large writes separately, with its own limit. Writers entering the write path choose which semaphore to acquire based on their `uio_resid`.

Expected outcome: practical experience with multiple semaphores. Think through: if a writer acquires the “large” sema, then the `uio` turns out to be small, does the write path want to release and re-acquire? Or keep the acquired slot? Document your choice.

15.12.4 Challenge 4: Atomic-Based Recovery Flag Elimination

Replace the `recovery_in_progress` state flag with an atomic compare-and-swap. The watchdog does `atomic_cmpset_int(&sc->recovery_in_progress, 0, 1)`; on success it enqueues the task. The task clears the flag with `atomic_store_rel_int`.

Expected outcome: the recovery mechanism no longer requires the mutex. Compare the two implementations in terms of correctness, complexity, and observability.

15.12.5 Challenge 5: Epoch for a Hypothetical Read Path

Study `/usr/src/sys/sys/epoch.h` and `/usr/src/sys/kern/subr_epoch.c`. Sketch (in comments, not in code) how you would convert the read path to use a private epoch that protects a “current configuration” pointer that writers occasionally update.

Expected outcome: a written proposal with the trade-offs. Since the driver’s current config is small and `sx` already handles it well, this is a thought exercise rather than a practical refactor. The point is to understand when epoch would be a win.

15.12.6 Challenge 6: Stress Test a Specific Race

Pick one race condition from Section 7. Write a script that triggers it reliably. Verify that the Stage 4 driver does not exhibit the bug. Verify that a deliberately broken version (revert the fix) does exhibit the bug.

Expected outcome: the satisfaction of seeing a race condition produced on demand, and the reassurance that the production code handles it.

15.12.7 Challenge 7: Read a Real Driver's Synchronization Vocabulary

Open `/usr/src/sys/dev/bge/if_bge.c` (or a similar mid-size driver with many primitives). Walk through its attach and detach paths. Count:

- Mutexes.
- Condition variables.
- Sx locks.
- Semaphores (if any; many drivers have none).
- Callouts.
- Tasks.
- Atomic operations.

Write a one-page summary of the driver's synchronisation strategy. Compare to `myfirst`. What does the real driver do differently, and why?

Expected outcome: reading a real driver's synchronisation strategy is the fastest way to make your own feel familiar. After one such read, opening any other driver in `/usr/src/sys/dev/` becomes easier.

15.13 Troubleshooting Reference

A flat reference list for common Chapter 15 problems.

15.13.1 Semaphore Deadlock or Leak

- **Writers pile up and never proceed.** The semaphore's counter is zero and no one posts. Check: is every `sema_wait` matched by a `sema_post`? Is there an early-return path that forgets the post?
- **`sema_destroy` panics with "waiters" assertion.** The destroy happened while a thread was still inside `sema_wait`. Fix: ensure the detach path quiesces all potential waiters before destroying. Usually this means clearing `is_attached` and doing a cv broadcast first.
- **`sema_trywait` returns unexpected values.** Remember: 1 on success, 0 on failure. Inverse of most FreeBSD APIs. Recheck the call-site logic.

15.13.2 Sx Lock Issues

- **`sx_try_upgrade` fails always.** The calling thread probably shares the sx with another reader. Check: is there a path that holds `sx_slock` persistently in a different thread?
- **Deadlock between sx and another lock.** Lock order violation. Run under `WITNESS`; the kernel will name the violation.
- **`sx_downgrade` with no `sx_try_upgrade` pair.** Make sure the sx is actually held exclusively before downgrading. `sx_xlocked(&sx)` asserts this.

15.13.3 Signal-Handling Issues

- **`read(2)` does not respond to SIGINT.** The blocking wait is `cv_wait` (not `cv_wait_sig`), or `sema_wait`. Convert to the signal-interruptible variant.

- **read(2) returns EINTR with partial bytes lost.** The partial-progress check is missing. Add the `uio_resid != nbefore` check on the signal-error path.
- **read(2) loops after EINTR.** The loop continues on signal-error instead of returning. Add the EINTR/ERESTART handling.

15.13.4 Atomic and Memory Ordering

- **Shutdown flag check misses.** A context reads `sc->is_attached` with a plain load and sees a stale value. Convert to `atomic_load_acq_int`.
- **Write order not observed on arm64.** Writes were reordered across architectures. Use `atomic_store_rel_int` for the last write in the pre-detach sequence.

15.13.5 Coordination Bugs

- **Recovery task runs multiple times.** The state flag is not protected or the atomic CAS is misused. Either use the mutex-protected flag pattern or audit the CAS logic.
- **Recovery task never runs.** The flag is never cleared, or the watchdog is not enqueueing. Check which side has the bug with a `device_printf` on each path.

15.13.6 Testing Issues

- **Stress test passes sometimes and fails others.** A race with low probability. Run many more iterations, increase concurrency, or add timing noise (`usleep` at random points) to expose it.
- **DTrace probes do not fire.** The kernel was built without FBT probes, or the function was inlined away. Check `dtrace -l | grep myfirst`.
- **WITNESS warnings flood the logs.** Do not ignore them. Every warning is a real bug. Fix one at a time and iterate.

15.13.7 Detach Issues

- **kldunload returns EBUSY.** Open file descriptors still exist. Close them and retry.
- **kldunload hangs.** A drain is waiting for a primitive that cannot complete. Usually a task or callout. Use `procstat -kk` on the `kldunload` thread to find where it is stuck.
- **Kernel panic during unload.** Use-after-free in a task or callout callback; the ordering is wrong. Review the detach sequence in `LOCKING.md` against the actual code.

15.14 Wrapping Up Part 3

Chapter 15 is the last chapter of Part 3. Part 3 had a specific mission: give the `myfirst` driver a full synchronisation story, from the first mutex in Chapter 11 to the last atomic flag in Chapter 15. The mission is complete.

A short inventory of what Part 3 delivered.

15.14.1 The Mutex (Chapter 11)

The first primitive. A sleep mutex protects the driver's shared data from concurrent access. Every path that touches the cbuf, the open count, the active-fh count, or the other mutex-protected fields acquires `sc->mtx` first. `WITNESS` enforces the rule.

15.14.2 The Condition Variable (Chapter 12)

The waiting primitive. Two cvs (`data_cv`, `room_cv`) let readers and writers sleep until the buffer state is acceptable. `cv_wait_sig` and `cv_timedwait_sig` make the waits signal-interruptible and time-bounded.

15.14.3 The Shared/Exclusive Lock (Chapter 12)

The read-mostly primitive. `sc->cfg_sx` protects the configuration structure. Shared acquisition for reads, exclusive for writes. Chapter 15 added `sc->stats_cache_sx` with the upgrade-promote-downgrade pattern.

15.14.4 The Callout (Chapter 13)

The time primitive. Three callouts (heartbeat, watchdog, tick source) give the driver internal time without requiring a dedicated thread. `callout_init_mtx` makes them lock-aware; `callout_drain` makes them tear down safely.

15.14.5 The Taskqueue (Chapter 14)

The deferred-work primitive. A private taskqueue with three tasks (selwake, bulk writer, recovery) moves work from constrained contexts to thread context. The detach sequence drains every task before freeing the queue.

15.14.6 The Semaphore (Chapter 15)

The bounded-admission primitive. `writers_sema` caps concurrent writers. The API is small and ownership-free; the driver uses `sema_trywait` for non-blocking entries and `sema_wait` for blocking.

15.14.7 Atomic Operations (Chapter 15)

The cross-context flag. `atomic_load_acq_int` and `atomic_store_rel_int` on `is_attached` make the shut-down flag visible to every context with correct memory ordering.

15.14.8 Encapsulation (Chapter 15)

The maintenance primitive. `myfirst_sync.h` wraps every synchronisation operation in a named function. A future reader understands the driver's synchronisation strategy by reading one header.

15.14.9 What the Driver Can Now Do

A short inventory of the driver's capabilities at the end of Part 3:

- Serve concurrent readers and writers on a bounded ring buffer.
- Block readers until data arrives, with optional timeout.
- Block writers until room is available, with optional timeout.
- Cap concurrent writers via a configurable semaphore.
- Expose configuration (debug level, nickname, soft byte limit) protected by an sx lock.
- Emit a periodic heartbeat log line.
- Detect stalled buffer drainage via a watchdog.
- Inject synthetic data via a tick source callout.
- Defer `se_lwakeup` out of the callout callback via a task.
- Demonstrate task coalescing via a configurable bulk writer.
- Schedule a delayed reset via a timeout task.
- Expose a cached statistic via an upgrade-aware sx pattern.
- Coordinate detach across every primitive without races.
- Answer signals correctly during blocking operations.
- Respect partial-progress semantics on read and write.

That is a substantial driver. The `myfirst` module at `0.9-coordination` is a compact but complete example of the synchronisation patterns that every real FreeBSD driver uses. The patterns transfer.

15.14.10 What Part 3 Did Not Cover

A short list of topics Part 3 deliberately left for later parts:

- Hardware interrupts (Part 4).
- Memory-mapped register access (Part 4).
- DMA and bus space operations (Part 4).
- PCI device matching (Part 4).
- USB and network-specific subsystems (Part 6).
- Advanced performance tuning (specialised chapters later).

Part 3 focused on the internal synchronisation story. Part 4 will add the hardware-facing story. The synchronisation story does not go away; it becomes the foundation on which the hardware story rests.

15.14.11 A Reflection

You started Chapter 11 with a driver that supported one user at a time. You end Chapter 15 with a driver that supports many users, coordinates several kinds of work, and tears down cleanly under load. Along the way you learned the kernel's main synchronisation primitives, each introduced with a specific invariant in mind, each composed with what came before.

The learning pattern was deliberate. Each chapter introduced one new concept, applied it to the driver in a small refactor, documented it in `LOCKING.md`, and added regression tests. The result is a driver whose synchronisation

is not an accident. Every primitive is there for a reason; every primitive is documented; every primitive is tested.

That discipline is the most durable thing Part 3 teaches. The specific primitives (mutex, cv, sx, callout, taskqueue, sema, atomic) are the currency, but the discipline of “pick the right primitive, document it, test it” is the investment. Drivers built with that discipline survive growth, maintenance handoffs, and surprising load patterns. Drivers built without it accumulate subtle bugs and hard-to-explain crashes.

Part 4 opens the door to hardware. The primitives you now know stay with you. The discipline you have practised is what will let you add the hardware-facing story without getting lost.

Take a moment. This is a real accomplishment. Then move to Chapter 16.

15.15 Part 3 Checkpoint

Five chapters of synchronization is a lot of material. Before Part 4 opens the door to hardware, it is worth confirming that the primitives and the discipline have settled.

By the end of Part 3 you should be able to do each of the following confidently:

- Choose among `mutex(9)`, `sx(9)`, `rw(9)`, `cv(9)`, `callout(9)`, `taskqueue(9)`, `sema(9)`, and the `atomic(9)` family with a clear sense of which invariant each one is appropriate for, rather than by habit or guesswork.
- Document a driver’s locking in a `LOCKING.md` that names every primitive, the invariant it enforces, the data it protects, and the rules callers must follow.
- Implement a sleep-and-wake handshake using either `mtx_sleep/wakeup` or `cv_wait/cv_signal`, and explain why you chose one over the other.
- Schedule timed work with `callout(9)`, including cancellation under detach, without leaving dangling timers behind.
- Defer heavy or ordered work through `taskqueue(9)`, including the detach-time drain that prevents tasks from running against freed state.
- Keep a `myfirst` running cleanly under `INVARIANTS` and `WITNESS` kernels while multi-threaded stress tests hammer every entry point.

If any of those are still vague, revisit the labs that introduced them:

- Locking discipline and regression: Lab 11.2 (Verify the Locking Discipline with `INVARIANTS`), Lab 11.4 (Build a Multi-threaded Tester), and Lab 11.7 (Long-Running Stress).
- Condition variables and `sx`: Lab 12.2 (Add Bounded Reads), Lab 12.5 (Detect a Deliberate Lock Order Reversal), and Lab 12.7 (Verify the Snapshot-and-Apply Pattern Holds Under Contention).
- Callouts and timed work: Lab 13.1 (Add a Heartbeat Callout) and Lab 13.4 (Verify Detach With Active Callouts).
- Taskqueues and deferred work: Lab 2 (Measuring Coalescing Under Load) and Lab 3 (Verify the Detach Ordering) in Chapter 14.
- Semaphores: Lab 1 (Observe Writer-Cap Enforcement) and Lab 4 (Detach Under Maximum Load) in Chapter 15.

Part 4 will layer hardware onto everything Part 3 just built. Specifically, the chapters that follow will expect:

- The synchronization model internalized rather than memorized, so that an interrupt context can be added as one more kind of caller rather than as a new universe of rules.
- Detach ordering treated as a single, shared discipline across primitives, since Part 4 will add interrupt teardown and bus-resource release into the same chain.
- Continued comfort with INVARIANTS and WITNESS as the default development kernel, since Part 4's harder bugs will usually trip one of the two long before they surface as a visible panic.

If those hold, Part 4 is within reach. If one still feels shaky, the fix is a lap through the relevant lab rather than a push forward.

15.16 Bridge to Chapter 16

Chapter 16 opens Part 4 of the book. Part 4 is titled *Hardware and Platform-Level Integration*, and Chapter 16 is *Hardware Basics and Newbus*. Part 4's mission is to give the driver a hardware story: how a driver announces itself to the kernel's bus layer, how it is matched against hardware the kernel has discovered, how it receives interrupts, how it accesses memory-mapped registers, and how it manages DMA.

The synchronisation story from Part 3 does not go away. It becomes the foundation on which Part 4 builds. A hardware interrupt handler runs in a context you now know how to reason about (no sleeping, no sleepable locks, no uimove). It communicates with the rest of the driver through primitives you now know how to use (taskqueues for deferred work, mutexes for serialisation, atomic flags for shutdown). The difference is that the driver now also has to talk to the hardware directly, and the hardware has its own rules.

Chapter 16 prepares the hardware ground in three specific ways.

First, **you already know about context boundaries**. Part 3 taught you that callouts, tasks, and user threads each have their own rules. Interrupts add one more context with stricter rules. The mental model ("which context am I in; what can I safely do here") transfers directly.

Second, **you already know about detach ordering**. Part 3 built up a detach discipline across five primitives. Part 4 adds two more (interrupt teardown, resource release) that slot into the same discipline. The ordering rules grow; the shape does not.

Third, **you already know about LOCKING.md as a living document**. Chapter 16 adds a Hardware Resources section. The discipline is the same; the vocabulary extends.

Specific topics Chapter 16 will cover:

- The newbus(9) framework: how drivers are identified, probed, and attached.
- `device_t`, `devclass`, `driver_t`, and `device_method_t`.
- `bus_alloc_resource` and `bus_release_resource` for memory-mapped regions, IRQ lines, and other resources.
- `bus_setup_intr` and `bus_teardown_intr` for interrupt registration.
- Filter handlers versus interrupt threads.
- The relationship between newbus and the PCI subsystem (preparing for Chapter 17).

You do not need to read ahead. Chapter 15 is sufficient preparation. Bring your `myfirst` driver at 0.9-coordination, your `LOCKING.md`, your WITNESS-enabled kernel, and your test kit. Chapter 16 starts where Chapter 15 ended.

A small closing reflection. The driver you started Part 3 with knew how to serve one syscall. The driver you have now has a full internal synchronisation story, with six kinds of primitives, each selected for a specific shape of invariant, each encapsulated in a readable wrapper layer, each documented, each tested. It is ready to face the hardware.

The hardware is the next chapter. Then the one after that. Then every chapter of Part 4. The foundation is built. The tools are on the bench. The blueprint is ready.

Turn the page.

15.17 Reference: Pre-Production Synchronisation Audit

Before shipping a synchronisation-heavy driver, run through this audit. Each item is a question; each should be answerable with confidence.

15.17.1 Mutex Audit

- ☐ Does every `sc->mtx` hold region end in an `mtx_unlock` on every path?
- ☐ Is the mutex always released before calling `uiomove`, `copyin`, `copyout`, `selwakeup`, or any other sleepable operation?
- ☐ Are there any places the mutex is held across a `cv wait`? If so, is the wait the intended primitive?
- ☐ Is the lock order `sc->mtx -> sc->cfg_sx -> sc->stats_cache_sx` respected everywhere?

15.17.2 Condition Variable Audit

- ☐ Does every `cv wait` call `cv_wait_sig` or `cv_timedwait_sig` in a syscall context?
- ☐ Is `EINTR` handled with partial-progress preservation where appropriate?
- ☐ Is `ERESTART` propagated correctly?
- ☐ Does every `cv` have a matching broadcast on detach?
- ☐ Are wakeups done with the mutex held for correctness (or the mutex released for throughput, with the trade-off documented)?

15.17.3 Sx Audit

- ☐ Does every `sx_slock` have a matching `sx_sunlock`?
- ☐ Does every `sx_xlock` have a matching `sx_xunlock`?
- ☐ Does every `sx_try_upgrade` failure path correctly handle the re-check after dropping and re-acquiring?
- ☐ Does every `sx_downgrade` happen on a lock that is actually held exclusively?

15.17.4 Semaphore Audit

- ☐ Does every `sema_wait` have a matching `sema_post` on every path?
- ☐ Is the semaphore destroyed only after all waiters have been quiesced?
- ☐ Is `sema_trywait`'s return value (1 on success, 0 on failure) read correctly?

- ☐ If the semaphore is used with interruptible syscalls, is the non-interruptibility of `sema_wait` documented?
- ☐ Is there an in-flight counter (e.g., `writers_inflight`) that detach drains before calling `sema_destroy`?
- ☐ Does every path between counter increment and counter decrement actually use the sema (no early returns that bypass the increment)?

15.17.5 Callout Audit

- ☐ Does every callout use `callout_init_mtx` with the appropriate lock?
- ☐ Does every callout callback check `is_attached` and exit early if false?
- ☐ Is every callout drained at detach before the state it touches is freed?

15.17.6 Task Audit

- ☐ Does every task callback hold the appropriate locks when touching shared state?
- ☐ Does every task callback call `se_lwakeup` only with no driver lock held?
- ☐ Is every task drained at detach after the callouts that enqueue it have been drained?
- ☐ Is the private taskqueue freed after every task is drained?

15.17.7 Atomic Audit

- ☐ Does every shutdown flag read use `atomic_load_acq_int`?
- ☐ Does every shutdown flag write use `atomic_store_rel_int`?
- ☐ Are any other atomic operations justified by a specific memory-ordering requirement?

15.17.8 Cross-Component Audit

- ☐ Does every cross-component state flag have a clear ownership (which path sets, which path clears)?
- ☐ Are the flags protected by the appropriate lock or atomic discipline?
- ☐ Is the handshake documented in `LOCKING.md`?

15.17.9 Documentation Audit

- ☐ Does `LOCKING.md` list every primitive?
- ☐ Does `LOCKING.md` document the detach ordering?
- ☐ Does `LOCKING.md` document the lock order?
- ☐ Are any subtle cross-component handshakes explained?

15.17.10 Testing Audit

- ☐ Has the driver been run under `WITNESS` for an extended stress test with no warnings?
- ☐ Have detach cycles been tested under full load?
- ☐ Have signal-interrupt tests confirmed partial-progress preservation?
- ☐ Have configuration changes at runtime (`sysctl` tuning) been tested?

A driver that passes this audit is a driver you can ship.

15.18 Reference: Synchronisation Primitive Cheat Sheet

15.18.1 When to Use Which

Primitive	Best for	Not good for
<code>struct mtx (MTX_DEF)</code>	Short critical sections; mutual exclusion.	Waiting for a condition.
<code>struct cv + mtx</code>	Waiting for a predicate; signalled wakeups.	Bounded admission.
<code>struct sx</code>	Read-mostly state; shared reads with occasional writes.	Hot contention.
<code>struct sema</code>	Bounded admission; producer-consumer completion.	Interruptible waits.
<code>callout</code>	Time-based work.	Work that must sleep.
<code>taskqueue</code>	Deferred thread-context work.	Sub-microsecond latency.
<code>atomic_*</code>	Small cross-context flags; lock-free coordination.	Complex invariants.
<code>epoch</code>	Read-mostly shared structures in network code.	Drivers without shared structures.

15.18.2 API Quick Reference

Mutex. - `mtx_init(&mtx, name, type, MTX_DEF)` - `mtx_lock(&mtx)`, `mtx_unlock(&mtx)` - `mtx_assert(&mtx, MA_OWNED)` - `mtx_destroy(&mtx)`

Condition variable. - `cv_init(&cv, name)` - `cv_wait(&cv, &mtx)`, `cv_wait_sig` - `cv_timedwait(&cv, &mtx, timo)`, `cv_timedwait_sig` - `cv_signal(&cv)`, `cv_broadcast(&cv)` - `cv_destroy(&cv)`

Sx. - `sx_init(&sx, name)` - `sx_slock(&sx)`, `sx_sunlock(&sx)` - `sx_xlock(&sx)`, `sx_xunlock(&sx)` - `sx_try_upgrade(&sx)`, `sx_downgrade(&sx)` - `sx_xlocked(&sx)`, `sx_xholder(&sx)` - `sx_destroy(&sx)`

Semaphore. - `sema_init(&s, value, name)` - `sema_wait(&s)`, `sema_timedwait(&s, timo)`, `sema_trywait(&s)` - `sema_post(&s)` - `sema_value(&s)` - `sema_destroy(&s)`

Callout. - `callout_init_mtx(&co, &mtx, 0)` - `callout_reset(&co, ticks, fn, arg)` - `callout_stop(&co)`, `callout_drain(&co)`

Taskqueue. - `TASK_INIT(&t, 0, fn, ctx)`, `TIMEOUT_TASK_INIT(...)` - `taskqueue_create(name, flags, enqueue, ctx)` - `taskqueue_start_threads(&tq, count, pri, name, ...)` - `taskqueue_enqueue(tq, &t)`, `taskqueue_enqueue_timeout(...)` - `taskqueue_cancel(tq, &t, &pend)`, `taskqueue_drain(tq, &t)` - `taskqueue_free(tq)`

Atomic. - `atomic_load_acq_int(p)`, `atomic_store_rel_int(p, v)` - `atomic_fetchadd_int(p, v)`, `atomic_cmpset_int(p, old, new)` - `atomic_set_int(p, v)`, `atomic_clear_int(p, v)` - `atomic_thread_fence_seq_cst()`

15.18.3 Context Rules

Context	Can sleep?	Sleepable locks?	Notes
Syscall	Yes	Yes	Full thread context.
Callout callback (lock-aware)	No	No	Registered mutex held.
Task callback (thread-backed)	Yes	Yes	No driver lock held.
Task callback (fast/swi)	No	No	SWI context.
Interrupt filter	No	No	Very limited.
Interrupt thread	No	No	Slightly more than filter.
Epoch section	No	No	Very limited.

15.18.4 Partial Progress Convention

A `read(2)` or `write(2)` that copies `N` bytes before being interrupted should return `N` (as a successful partial short read/write), not `EINTR`. The driver's wait helper returns a sentinel (`MYFIRST_WAIT_PARTIAL`) on the partial path; the caller converts it to 0 so the syscall layer returns the byte count.

15.18.5 Detach Ordering

The canonical order for Chapter 15 detach:

1. Refuse if `active_fhs > 0`.
2. Clear `is_attached` atomically.
3. Broadcast all cvs; release mutex.
4. Drain all callouts.
5. Drain all tasks (including timeout tasks and recovery).
6. `seldrain` for `rsl`, `wsl`.
7. Free taskqueue.
8. Destroy semaphore.
9. Destroy stats-cache `sx`.
10. Destroy `cdev`, `sysctls`, `cbuf`, counters.
11. Destroy `cvs`, `cfg sx`, mutex.

Memorise the shape. Adapt the order when you add new primitives.

15.19 Reference: Further Reading

15.19.1 Manual Pages

- `sema(9)`: kernel semaphores.
- `sx(9)`: shared/exclusive locks.

- `mutex(9)`: mutex primitives.
- `condvar(9)`: condition variables.
- `atomic(9)`: atomic operations.
- `epoch(9)`: epoch-based synchronisation.
- `locking(9)`: overview of kernel locking primitives.

15.19.2 Source Files

- `/usr/src/sys/kern/kern_sema.c`: semaphore implementation.
- `/usr/src/sys/sys/sema.h`: semaphore API.
- `/usr/src/sys/kern/kern_sx.c`: sx implementation.
- `/usr/src/sys/sys/sx.h`: sx API.
- `/usr/src/sys/kern/kern_mutex.c`: mutex implementation.
- `/usr/src/sys/kern/kern_condvar.c`: cv implementation.
- `/usr/src/sys/kern/subr_epoch.c`: epoch implementation.
- `/usr/src/sys/sys/epoch.h`: epoch API.
- `/usr/src/sys/dev/hyperv/storvsc/hv_storvsc_drv_freebsd.c`: real-world sema usage.
- `/usr/src/sys/dev/bge/if_bge.c`: rich synchronisation example.

15.19.3 Books and External Material

- *The Design and Implementation of the FreeBSD Operating System* (McKusick et al.): has detailed chapters on the kernel synchronisation subsystems.
- *FreeBSD Handbook*, developer section: sections on kernel locking.
- FreeBSD mailing list archives: searches for primitive names (`taskqueue`, `sema`, `sx`) reveal historical design discussions.

15.19.4 Suggested Reading Order

For a reader new to advanced synchronisation:

1. `mutex(9)`, `condvar(9)`: the basic primitives.
2. `sx(9)`: the read-mostly primitive.
3. `sema(9)`: the bounded-admission primitive.
4. `atomic(9)`: the cross-context tool.
5. `epoch(9)`: the network-drivers' read-lock-free tool.
6. A real driver source: `/usr/src/sys/dev/bge/if_bge.c` or similar.

Reading in order takes a full afternoon and gives you a solid mental map.

15.20 Reference: A Glossary of Chapter 15 Terms

Counting semaphore. A primitive that holds a non-negative integer and supports wait (decrement, blocking if zero) and post (increment, waking one waiter).

Binary semaphore. A counting semaphore that only holds 0 or 1. Behaviourally similar to a mutex but without ownership.

Priority inheritance. A scheduler technique where a high-priority thread waiting on a lock temporarily raises the priority of the current holder. FreeBSD mutexes support it; semaphores do not.

Signal-interruptible wait. A blocking primitive (e.g. `cv_wait_sig`) that returns with `EINTR` or `ERESTART` when a signal arrives. The caller can then abandon the wait and surface the signal.

Partial progress convention. UNIX-standard behaviour: a `read(2)` or `write(2)` that transferred some bytes and was then interrupted returns the byte count as success, not an error.

EINTR vs ERESTART. Two signal return codes. `EINTR` surfaces to user space as `errno EINTR`. `ERESTART` causes the syscall layer to restart the syscall transparently, based on signal disposition.

Acquire barrier. A memory barrier on a load that prevents subsequent memory accesses from being reordered before the load.

Release barrier. A memory barrier on a store that prevents preceding memory accesses from being reordered after the store.

Compare-and-swap (CAS). An atomic operation that writes a new value only if the current value matches an expected one. Basis for lock-free state machines.

Upgrade-promote-downgrade. An sx pattern: acquire shared, detect need to write, try to upgrade to exclusive, write, downgrade back to shared.

Coalescing. Taskqueue property: redundant enqueues of the same task fold into a single pending state with an incremented counter, rather than being linked separately.

Encapsulation layer. A header (`myfirst_sync.h`) that names every synchronisation operation the driver performs, so the strategy can be changed in one place and understood at a glance.

State flag. A small integer in the `softc` that records whether a specific condition is in progress. Protected by the appropriate lock or atomic discipline.

Cross-component handshake. A coordination between multiple execution contexts (callout, task, user thread) using a state flag, `cv`, or atomic.

Chapter 15 ends here. Part 4 begins next.

15.21 Reference: Reading `kern_sema.c` Line by Line

The implementation of `sema(9)` is short enough to read end-to-end. Doing so once cements the mental model of what the primitive actually does. The file is `/usr/src/sys/kern/kern_sema.c`, under two hundred lines. A narrated pass follows.

15.21.1 `sema_init`

```
void
sema_init(struct sema *sema, int value, const char *description)
{
    KASSERT((value >= 0), ("%s(): negative value\n", __func__));

    bzero(sema, sizeof(*sema));
    mtx_init(&sema->sema_mtx, description, "sema backing lock",
        MTX_DEF | MTX_NOWITNESS | MTX_QUIET);
    cv_init(&sema->sema_cv, description);
    sema->sema_value = value;

    CTR4(KTR_LOCK, "%s(%p, %d, \"%s\")", __func__, sema, value, description);
}
```

Six lines of logic. Assert the initial value is non-negative. Zero the structure. Initialise the internal mutex; note the flags `MTX_NOWITNESS | MTX_QUIET`, which tell `WITNESS` not to track the internal mutex (because the semaphore itself is what the user cares about, not its backing mutex). Initialise the internal cv. Set the counter.

The implication: a semaphore is literally a mutex plus a cv plus a counter, assembled into a small package. Understanding this assembly is understanding the primitive.

15.21.2 sema_destroy

```
void
sema_destroy(struct sema *sema)
{
    CTR3(KTR_LOCK, "%s(%p) \"%s\"", __func__, sema,
        cv_wmesg(&sema->sema_cv));

    KASSERT((sema->sema_waiters == 0), ("%s(): waiters\n", __func__));

    mtx_destroy(&sema->sema_mtx);
    cv_destroy(&sema->sema_cv);
}
```

Two lines of logic after the tracing. Assert no waiters exist. Destroy the internal mutex and cv. The assertion is what forces you to quiesce waiters before destroying; violating it panics a debug kernel.

15.21.3 _sema_post

```
void
_sema_post(struct sema *sema, const char *file, int line)
{

    mtx_lock(&sema->sema_mtx);
    sema->sema_value++;
    if (sema->sema_waiters && sema->sema_value > 0)
        cv_signal(&sema->sema_cv);

    CTR6(KTR_LOCK, "%s(%p) \"%s\" v = %d at %s:%d", __func__, sema,
        cv_wmesg(&sema->sema_cv), sema->sema_value, file, line);

    mtx_unlock(&sema->sema_mtx);
}
```

Three lines of logic. Lock, increment, signal one waiter if any. The signal is conditional on two things: the waiters count is non-zero (no need to signal if nobody is waiting), and the value is positive (signals with a zero value would wake a waiter who would immediately sleep again). The second condition is subtle; it protects against a situation where `sema_value` went positive and then became zero again between a post and the current post. In practice on simple usage the conditions are both true.

15.21.4 `_sema_wait`

```
void
_sema_wait(struct sema *sema, const char *file, int line)
{

    mtx_lock(&sema->sema_mtx);
    while (sema->sema_value == 0) {
        sema->sema_waiters++;
        cv_wait(&sema->sema_cv, &sema->sema_mtx);
        sema->sema_waiters--;
    }
    sema->sema_value--;

    CTR6(KTR_LOCK, "%s(%p) \"%s\" v = %d at %s:%d", __func__, sema,
        cv_wmesg(&sema->sema_cv), sema->sema_value, file, line);

    mtx_unlock(&sema->sema_mtx);
}
```

Four lines of logic. Lock. Loop while the value is zero: increment waiters, wait on the cv, decrement waiters. Once value is positive, decrement it and unlock.

Two observations.

The loop is what makes the primitive safe against spurious wakeups. `cv_wait` can return without `cv_signal` having been called. The loop re-checks the value every time, so a spurious wakeup just re-sleeps.

The wait uses `cv_wait`, not `cv_wait_sig`. This is what makes `sema_wait` non-interruptible. A signal to the caller does nothing. The loop continues until a real `sema_post` arrives.

15.21.5 `_sema_timedwait`

```
int
_sema_timedwait(struct sema *sema, int timo, const char *file, int line)
{
    int error;

    mtx_lock(&sema->sema_mtx);

    for (error = 0; sema->sema_value == 0 && error == 0;) {
        sema->sema_waiters++;
        error = cv_timedwait(&sema->sema_cv, &sema->sema_mtx, timo);
        sema->sema_waiters--;
    }
    if (sema->sema_value > 0) {
        sema->sema_value--;
        error = 0;
        /* ... tracing ... */
    } else {
        /* ... tracing ... */
    }

    mtx_unlock(&sema->sema_mtx);
    return (error);
}
```

Slightly more complex. The loop uses `cv_timedwait`, again not `cv_timedwait_sig`, so the timed wait is also not interruptible. The loop exits when the value is positive or when the error becomes non-zero (usually `EWOLDBLOCK`).

After the loop: if the value is positive, we claim it and return 0 (success). Otherwise, we return the error (`EWOLDBLOCK`). Note the `cv` error is preserved from the last loop iteration in the error case.

A subtlety the comment in the source calls out: a spurious wakeup resets the effective timeout interval because each iteration uses a fresh `timo`. This means the actual wait may be slightly longer than the caller requested, but never shorter. The `EWOLDBLOCK` return eventually fires.

15.21.6 _sema_trywait

```
int
_sema_trywait(struct sema *sema, const char *file, int line)
{
    int ret;

    mtx_lock(&sema->sema_mtx);

    if (sema->sema_value > 0) {
        sema->sema_value--;
        ret = 1;
    } else {
        ret = 0;
    }

    mtx_unlock(&sema->sema_mtx);
    return (ret);
}
```

Two lines of logic. Lock. If value is positive, decrement and return 1. Otherwise return 0. No blocking, no cv, no waiter accounting.

15.21.7 sema_value

```
int
sema_value(struct sema *sema)
{
    int ret;

    mtx_lock(&sema->sema_mtx);
    ret = sema->sema_value;
    mtx_unlock(&sema->sema_mtx);
    return (ret);
}
```

One line of logic. Returns the current value. The value can change immediately after the mutex is released, so the result is a snapshot, not a guarantee. Useful for diagnostics.

15.21.8 Observations

Reading the whole file takes ten minutes. At the end you understand:

- Semaphores are built from a mutex and a cv.
- `sema_wait` is not signal-interruptible because it uses `cv_wait`.
- `sema_destroy` asserts no waiters, which is why you must quiesce before destroying.
- Coalescing does not exist (every post decrements the counter by one, no “pending” accounting).
- The primitive is simple; its contracts are precise.

This kind of reading is the shortest path to mastery of any kernel primitive. The `sema(9)` file is a particularly good starter because it is so short.

15.22 Reference: Standardising Synchronization Primitives Across a Driver

As the driver accumulates more primitives, consistency matters more than cleverness.

15.22.1 One Naming Convention

The Chapter 15 convention, which the reader can adopt or modify:

- **Mutex:** `sc->mtx`. Only one per driver. If the driver needs more than one, each has a purpose suffix: `sc->tx_mtx`, `sc->rx_mtx`.
- **Condition variable:** `sc-><purpose>_cv`. E.g., `data_cv`, `room_cv`.
- **Sx lock:** `sc-><purpose>_sx`. E.g., `cfg_sx`, `stats_cache_sx`.
- **Semaphore:** `sc-><purpose>_sema`. E.g., `writers_sema`.
- **Callout:** `sc-><purpose>_co`. E.g., `heartbeat_co`.
- **Task:** `sc-><purpose>_task`. E.g., `selwake_task`.
- **Timeout task:** `sc-><purpose>_delayed_task`. E.g., `reset_delayed_task`.
- **Atomic flag:** `sc-><purpose>` as an int. No suffix; the type tells the story. E.g., `is_attached`, `recovery_in_progress`.
- **State flag under mutex:** same as atomic; the comment in the softc names the lock.

15.22.2 One Init/Destroy Pattern

Each primitive has a canonical init and destroy. The order in attach mirrors the reverse in detach.

Attach order: 1. Mutex. 2. Cvs. 3. Sx locks. 4. Semaphores. 5. Taskqueue. 6. Callouts. 7. Tasks. 8. Atomic flags (no init required; initialised by softc zeroing).

Detach order (roughly reverse): 1. Clear atomic flag. 2. Broadcast cvs. 3. Release mutex. 4. Drain callouts. 5. Drain tasks. 6. Free taskqueue. 7. Destroy semaphores. 8. Destroy sxs. 9. Destroy cvs. 10. Destroy mutex.

The rule of thumb: destroy in the opposite order to init, and drain anything that can still fire before destroying what it touches.

15.22.3 One Encapsulation Pattern

The `myfirst_sync.h` pattern from Section 6 scales. Every primitive has a wrapper. Every wrapper is named for what it does, not for the primitive it wraps.

15.22.4 One LOCKING.md Template

A LOCKING.md section per primitive. Each section names:

- The primitive.
- Its purpose.
- Its lifetime.
- Its contract with other primitives (lock order, ownership, interaction with the atomic flag).

A new primitive added to the driver adds a new section. A modified primitive changes its existing section. The document is always current.

15.22.5 Why Standardise

The benefits are the same as in Chapter 14. Reduced cognitive load. Fewer mistakes. Easier review. Easier handoff. The costs are small and one-time.

15.23 Reference: When Each Primitive Is Wrong

Synchronisation primitives are tools. Every tool has misuses. A short list of anti-patterns to recognise.

15.23.1 Mutex Misuses

- **Holding a mutex across a sleep.** Prevents other threads from making progress; may cause deadlock if the sleep depends on state another thread needs the mutex for.
- **Nesting mutexes in inconsistent order.** Creates a lock-order cycle; WITNESS catches it.
- **Using a mutex where an atomic would do.** For a single-bit flag checked often, a mutex is heavier than needed.
- **Missing `mtx_assert` in helpers that require the mutex.** Without the assertion, the helper can be called in a context where the mutex is not held; the bug may be silent.

15.23.2 Cv Misuses

- **Using `cv_wait` in a syscall context.** Cannot be interrupted by signals; makes the syscall unresponsive.
- **Not re-checking the predicate after wake-up.** Spurious wakeups are allowed; code that assumes a wakeup means the predicate is true is buggy.
- **Signalling without holding the mutex.** Usually allowed by the API but typically unwise; the waiter may miss the signal if the timing is unlucky.
- **Using `cv_signal` where `cv_broadcast` is needed.** A detach path that wakes only one waiter leaves others blocked.

15.23.3 Sx Misuses

- **Using `sx` where a mutex would do.** Sx has higher overhead than a mutex in the uncontended case; if there is no shared-access benefit, mutex is simpler.

- **Forgetting that `sx` can sleep.** `Sx` cannot be acquired from a non-sleeping context. Callouts initialised with `callout_init_mtx(, &mtx, 0)` are fine; filter interrupts are not. (Historical note: the older `CALLOUT_MPSAFE` flag named the same distinction. Chapter 13 walks through its deprecation.)
- **Using `sx_try_upgrade` without the fallback.** A naive upgrade that does not handle failure races against another upgrader.

15.23.4 Sema Misuses

- **Expecting signal interruption.** `sema_wait` does not interrupt.
- **Expecting priority inheritance.** `sema` does not raise the poster's priority.
- **Destroying with waiters.** Panics a debug kernel, silently corrupts a production kernel.
- **Forgetting a post on an error path.** Leaks a slot; eventually the semaphore drains to zero and all waiters block.

15.23.5 Atomic Misuses

- **Using plain atomics where acquire/release is needed.** Correct on x86, broken on arm64. Always think about memory ordering.
- **Protecting a complex invariant with a single atomic.** If the invariant involves multiple fields, atomics alone are insufficient; a lock is needed.
- **Using CAS where a simple load-store would do.** Wastes an atomic instruction.

15.23.6 Pattern Misuses

- **Rolling your own semaphore from mutex-plus-counter-plus-cv.** The kernel's `sema(9)` already does this; reinventing it creates maintenance debt.
- **Rolling your own read-write lock from mutex-plus-counter.** `sx(9)` does this; reinventing it creates maintenance debt.
- **Rolling your own encapsulation in the driver source.** Put it in a header (`myfirst_sync.h`); do not duplicate it inline.

Recognising anti-patterns is half of good synchronisation. The other half is choosing the right primitive in the first place, which the main body of the chapter has covered.

15.24 Reference: Observability Cheat Sheet

15.24.1 Sysctl Knobs Your Driver Should Expose

For Chapter 15, add these to the driver's sysctl tree (at `dev.myfirst.N.stats.*` or `dev.myfirst.N.*` as appropriate):

- `writers_limit`: current writer cap.
- `stats.writers_sema_value`: snapshot of the semaphore value.
- `stats.writers_trywait_failures`: count of non-blocking writers refused.

- stats.stats_cache_refreshes: count of cache refreshes.
- stats.recovery_task_runs: count of recovery invocations.
- stats.is_attached: current atomic flag value.

These read-only counters give an operator a window into the driver's synchronisation behaviour without a debugger.

15.24.2 DTrace Probes

Count cv signals per cv:

```
dtrace -n 'fbt::cv_signal:entry { @[stringof(args[0]->cv_description)] = count(); }'
```

Count sema waits and posts:

```
dtrace -n '
  fbt::_sema_wait:entry { @["wait"] = count(); }
  fbt::_sema_post:entry { @["post"] = count(); }
'
```

Measure sema wait latency:

```
dtrace -n '
  fbt::_sema_wait:entry { self->t = timestamp; }
  fbt::_sema_wait:return /self->t/ {
    @ = quantize(timestamp - self->t);
    self->t = 0;
  }
'
```

Useful for understanding how long writers block on the writer-cap in practice.

Measure sx contention:

```
dtrace -n '
  lockstat:::sx-block-enter /arg0 == (uintptr_t)&sc_addr/ {
    @ = count();
  }
'
```

Replace sc_addr with the actual address of your sx. Shows how often the sx blocked.

Observe taskqueue_run_locked for recovery:

```
dtrace -n 'fbt::myfirst_recovery_task:entry { printf("recovery at %Y", walltimestamp);  
↵ }'
```

Prints a timestamp every time the recovery task fires.

15.24.3 procstat

`procstat -t | grep myfirst`: shows taskqueue worker threads and their state.

`procstat -kk <pid>`: kernel stack of a specific thread. Useful when something is stuck.

15.24.4 ps

`ps ax | grep taskq`: list every taskqueue worker by name.

15.24.5 ddb

`db> show witness`: dump the WITNESS lock graph.

`db> show locks`: list currently held locks.

`db> show sleepchain <tid>`: walk a sleep chain to find deadlocks.

15.25 Reference: A Worked Stage-by-Stage Diff Summary

A compact summary of the driver diff from Chapter 14 Stage 4 to Chapter 15 Stage 4, for readers who want to see the whole change at a glance.

15.25.1 Stage 1 Diff (v0.8 -> v0.8+writers_sema)

Softc additions:

```
struct sema    writers_sema;  
int            writers_limit;  
int            writers_trywait_failures;  
int            writers_inflight;  /* atomic int; drain counter */
```

Attach additions:

```
sema_init(&sc->writers_sema, 4, "myfirst writers");  
sc->writers_limit = 4;  
sc->writers_trywait_failures = 0;  
sc->writers_inflight = 0;
```

Detach additions (after is_attached=0 and all cv broadcasts):

```
sema_destroy(&sc->writers_sema);
```

New sysctl handler: myfirst_sysctl_writers_limit.

Write-path changes: sema acquisition at entry, release at exit, O_NONBLOCK via sema_trywait.

15.25.2 Stage 2 Diff (+stats_cache_sx)

Softc additions:

```
struct sx      stats_cache_sx;
uint64_t      stats_cache_bytes_10s;
uint64_t      stats_cache_last_refresh_ticks;
```

Attach additions:

```
sx_init(&sc->stats_cache_sx, "myfirst stats cache");
sc->stats_cache_bytes_10s = 0;
sc->stats_cache_last_refresh_ticks = 0;
```

Detach additions (after mutex destroy):

```
sx_destroy(&sc->stats_cache_sx);
```

New helper: myfirst_stats_cache_refresh.

New sysctl handler: myfirst_sysctl_stats_cached.

15.25.3 Stage 3 Diff (EINTR/ERESTART + partial-progress)

No softc changes.

Wait-helper refactor: MYFIRST_WAIT_PARTIAL sentinel; explicit switch on error codes from cv waits.

Caller changes: explicit checks for the sentinel.

15.25.4 Stage 4 Diff (coordination + encapsulation)

Softc additions:

```
int      recovery_in_progress;
struct task recovery_task;
int      recovery_task_runs;
```

Attach additions:

```
TASK_INIT(&sc->recovery_task, 0, myfirst_recovery_task, sc);
sc->recovery_in_progress = 0;
sc->recovery_task_runs = 0;
```

Detach additions:

```
taskqueue_drain(sc->tq, &sc->recovery_task);
```

Atomic flag conversions: `is_attached` reads in handlers and callbacks become `atomic_load_acq_int`; the detach write becomes `atomic_store_rel_int`.

New header: `myfirst_sync.h` with inline wrappers.

Source edits: every primitive-specific call in the main source becomes a wrapper call.

Watchdog refactor: enqueue recovery task on stall, guarded by the `recovery_in_progress` flag.

Version bump: `MYFIRST_VERSION "0.9-coordination"`.

15.25.5 Total Lines Added

Rough count across the four stages:

- Softc: ~10 fields.
- Attach: ~15 lines.
- Detach: ~10 lines.
- New functions: ~80 lines (sysctl handlers, `recovery_task`, helpers).
- Modified functions: ~20 line edits.
- Header file: ~150 lines.

Net addition to the driver: approximately 300 lines. Compare to the approximately 100 lines Chapter 14 added and the 400 lines Chapter 13 added. The Chapter 15 additions are modest in volume; large in what they enable.

15.26 Reference: The Chapter 15 Driver Lifecycle

A summary of the lifecycle with the Chapter 15 additions explicit.

15.26.1 Attach Sequence

1. `mtx_init(&sc->mtx, ...)`.
2. `cv_init(&sc->data_cv, ...)`, `cv_init(&sc->room_cv, ...)`.
3. `sx_init(&sc->cfg_sx, ...)`.
4. `sx_init(&sc->stats_cache_sx, ...)`.
5. `sema_init(&sc->writers_sema, 4, ...)`.

6. `sc->tq = taskqueue_create(...); taskqueue_start_threads(...)`.
7. `callout_init_mtx(&sc->heartbeat_co, ...)`, two more.
8. `TASK_INIT(&sc->selwake_task, ...)`, three more (including `recovery_task`).
9. `TIMEOUT_TASK_INIT(sc->tq, &sc->reset_delayed_task, ...)`.
10. Softc fields initialised.
11. `sc->bytes_read = counter_u64_alloc(M_WAITOK)`; same for `bytes_written`.
12. `cbuf_init(&sc->cb, ...)`.
13. `make_dev_s(...)` for the `cdev`.
14. Sysctl tree set up.
15. `sc->is_attached = 1` (initial store is not strictly atomic-ordered because no reader can see the softc until it is attached).

15.26.2 Runtime

- User threads enter/leave via `open/close/read/write`.
- Callouts fire periodically.
- Tasks fire on enqueue.
- Watchdog detects stalls, enqueues recovery task.
- Atomic flag read via `myfirst_sync_is_attached` in entry checks.

15.26.3 Detach Sequence

1. `myfirst_detach` is called.
2. Check `active_fhs > 0`; return `EBUSY` if so.
3. `atomic_store_rel_int(&sc->is_attached, 0)`.
4. `cv_broadcast(&sc->data_cv, cv_broadcast(&sc->room_cv))`.
5. Release `sc->mtx`.
6. `callout_drain` three times.
7. `taskqueue_drain` three times (`selwake`, `bulk_writer`, `recovery`).
8. `taskqueue_drain_timeout` once (`reset_delayed`).
9. `seldrain` twice.
10. `taskqueue_free(sc->tq)`.
11. `sema_destroy(&sc->writers_sema)`.
12. `destroy_dev` twice.
13. `sysctl_ctx_free`.
14. `cbuf_destroy`, `counter_u64_free` twice.
15. `sx_destroy(&sc->stats_cache_sx)`.
16. `cv_destroy` twice.
17. `sx_destroy(&sc->cfg_sx)`.
18. `mtx_destroy(&sc->mtx)`.

15.26.4 Notes on the Sequence

- Every primitive init in `attach` has a matching `destroy` in `detach`, in reverse order.

- Every drain happens before the thing drained is freed.
- The atomic flag is the very first step after the `active_fhs` check, so every subsequent observer sees the shutdown.
- The taskqueue is freed before the semaphore is destroyed because a task might (in some extended design) wait on the semaphore.

Memorising this lifecycle is the single most useful thing a reader can do with Part 3.

15.27 Reference: Costs and Comparisons

A concise table of the synchronisation primitives' costs.

Primitive	Uncontended cost	Contended cost	Sleep?
<code>atomic_load_acq_int</code>	~1 ns on amd64	~1 ns (same)	No
<code>atomic_fetchadd_int</code>	~10 ns	~100 ns	No
<code>mtx_lock</code> (uncontended)	~20 ns	microseconds	Slow path sleeps
<code>cv_wait_sig</code>	n/a	full scheduler wakeup	Yes
<code>sx_slock</code>	~30 ns	microseconds	Slow path sleeps
<code>sx_xlock</code>	~30 ns	microseconds	Slow path sleeps
<code>sx_try_upgrade</code>	~30 ns	n/a (fails fast)	No
<code>sema_wait</code>	~40 ns	wakeup latency	Yes
<code>sema_post</code>	~30 ns	~100 ns	No
<code>callout_reset</code>	~100 ns	n/a	No
<code>taskqueue_enqueue</code>	~50 ns	wakeup latency	No

The figures above are order-of-magnitude estimates on typical FreeBSD 14.3 amd64 hardware, and the microsecond entries in the contended column correspond to low-microsecond wakeup latencies on that same class of machine. Actual numbers depend on cache state, contention, and system load, and can move by a factor of two or more across CPU generations. Use this table to decide where to optimise; a call taking hundreds of nanoseconds is not a bottleneck on a path that runs once per syscall. See Appendix F for a reproducible benchmark of these figures on your own hardware.

15.28 Reference: A Worked Synchronization Walk

For a fully concrete example, here is the complete control-flow of a blocking `read(2)` in the Chapter 15 Stage 4 driver, from syscall entry to data delivery.

1. User calls `read(fd, buf, 4096)`.
2. Kernel's VFS layer routes to `myfirst_read`.
3. `myfirst_read` calls `devfs_get_cdevpriv` to get the fh.

4. `myfirst_read` calls `myfirst_sync_is_attached(sc)`:
 - Expands to `atomic_load_acq_int(&sc->is_attached)`.
 - Returns 1 (attached).
5. Loop enters: `while (uio->uio_resid > 0)`.
6. `myfirst_sync_lock(sc)`:
 - Expands to `mtx_lock(&sc->mtx)`.
 - Acquires the mutex.
7. `myfirst_wait_data(sc, ioflag, nbefore, uio)`:
 - `while (cbuf_used == 0)`: buffer is empty.
 - Not partial (`nbefore == uio_resid`).
 - Not `IO_NDELAY`.
 - `read_timeout_ms` is 0, so uses `cv_wait_sig`.
 - `cv_wait_sig(&sc->data_cv, &sc->mtx)`:
 - Releases the mutex.
 - Thread sleeps on the cv.
 - Time passes. A writer (or the `tick_source`) calls `cv_signal(&sc->data_cv)`.
 - Thread wakes, `cv_wait_sig` re-acquires the mutex, returns 0.
 - `!sc->is_attached`: false.
 - Loop re-iterates: `cbuf_used` is now `> 0`.
 - Exits the loop; returns 0.
8. `myfirst_buf_read(sc, bounce, take)`:
 - Calls `cbuf_read(&sc->cb, bounce, take)`.
 - Copies data into the bounce buffer.
 - Increments `bytes_read` counter.
9. `myfirst_sync_unlock(sc)`.
10. `cv_signal(&sc->room_cv)`: wake a blocked writer if any.
11. `selwakeup(&sc->wsel)`: wake any poller waiting for write.
12. `uiomove(bounce, got, uio)`: copy from kernel to user space.
13. Loop continues: check `uio->uio_resid > 0`; eventually falls out.
14. Returns 0 to the syscall layer.
15. Syscall layer returns the number of bytes copied to user.

Every primitive in Chapter 15 is visible in the walk:

- The atomic flag (step 4).
- The mutex (step 6, 7, 9).
- The cv wait with signal handling (step 7's `cv_wait_sig`).
- The counter (step 8's `counter_u64_add`).
- The cv signal (step 10).
- The `selwakeup` (step 11, done outside the mutex per discipline).

A walk like this is a useful cross-check. Every primitive in the driver's vocabulary is exercised on the read path. If you can narrate the walk from memory, the synchronisation is internalised.

15.29 Reference: A Minimal Working Template

For copy-and-adapt convenience, a template that compiles and demonstrates Chapter 15's core additions in the smallest possible form. Every element has been introduced in the chapter; the template assembles them.

```
#include <sys/param.h>
#include <sys/kernel.h>
#include <sys/module.h>
#include <sys/systm.h>
#include <sys/bus.h>
#include <sys/mutex.h>
#include <sys/lock.h>
#include <sys/sx.h>
#include <sys/sem.h>
#include <sys/taskqueue.h>
#include <sys/priority.h>

struct template_softc {
    device_t      dev;
    struct mtx     mtx;
    struct sx      stats_sx;
    struct sema    admission_sema;
    struct taskqueue *tq;
    struct task     work_task;
    int            is_attached;
    int            work_in_progress;
};

static void
template_work_task(void *arg, int pending)
{
    struct template_softc *sc = arg;
    mtx_lock(&sc->mtx);
    /* Work under mutex if needed. */
    sc->work_in_progress = 0;
    mtx_unlock(&sc->mtx);
    /* Unlocked work here. */
}

static int
template_attach(device_t dev)
{
    struct template_softc *sc = device_get_softc(dev);
    int error;
```

```

    sc->dev = dev;
    mtx_init(&sc->mtx, device_get_nameunit(dev), "template", MTX_DEF);
    sx_init(&sc->stats_sx, "template stats");
    sema_init(&sc->admission_sema, 4, "template admission");

    sc->tq = taskqueue_create("template taskq", M_WAITOK,
        taskqueue_thread_enqueue, &sc->tq);
    if (sc->tq == NULL) { error = ENOMEM; goto fail_sema; }
    error = taskqueue_start_threads(&sc->tq, 1, PWAIT,
        "%s taskq", device_get_nameunit(dev));
    if (error != 0) goto fail_tq;

    TASK_INIT(&sc->work_task, 0, template_work_task, sc);

    atomic_store_rel_int(&sc->is_attached, 1);
    return (0);

fail_tq:
    taskqueue_free(sc->tq);
fail_sema:
    sema_destroy(&sc->admission_sema);
    sx_destroy(&sc->stats_sx);
    mtx_destroy(&sc->mtx);
    return (error);
}

static int
template_detach(device_t dev)
{
    struct template_softc *sc = device_get_softc(dev);

    atomic_store_rel_int(&sc->is_attached, 0);

    taskqueue_drain(sc->tq, &sc->work_task);
    taskqueue_free(sc->tq);
    sema_destroy(&sc->admission_sema);
    sx_destroy(&sc->stats_sx);
    mtx_destroy(&sc->mtx);
    return (0);
}

/* Entry on the hot path: */
static int
template_hotpath_enter(struct template_softc *sc)

```

```

{
    sema_wait(&sc->admission_sema);
    if (!atomic_load_acq_int(&sc->is_attached)) {
        sema_post(&sc->admission_sema);
        return (ENXIO);
    }
    return (0);
}

static void
template_hotpath_leave(struct template_softc *sc)
{
    sema_post(&sc->admission_sema);
}

```

The template is not a complete driver. It shows the shape of the primitives the chapter introduced. A real driver with this template plus the rest of the Chapter 14 and earlier patterns would be a fully functional synchronised device driver.

15.30 Reference: Comparison With POSIX User-Space Synchronization

Many readers come to FreeBSD kernel driver work from user-space systems programming. A short comparison clarifies the mapping.

Concept	POSIX user-space	FreeBSD kernel
Mutex	pthread_mutex_t	struct mtx
Condition variable	pthread_cond_t	struct cv
Read-write lock	pthread_rwlock_t	struct sx
Semaphore	sem_t (or sem_open)	struct sema
Thread creation	pthread_create	kproc_create, kthread_add
Deferred work	No direct analogue; common to roll your own	struct task + struct taskqueue
Periodic execution	timer_create + signal	struct callout
Atomic operations	<stdatomic.h> or __atomic_*	atomic(9)

The primitives are similar in shape but different in detail. Key differences:

- Kernel mutexes have priority inheritance; POSIX mutexes do only with the `PRIO_INHERIT` attribute.
- Kernel cvs have no name; POSIX cvs are anonymous too, so parity here.
- Kernel sx locks are more flexible than pthread_rwlock (try_upgrade, downgrade).

- Kernel semaphores are not signal-interruptible; POSIX semaphores (some variants) are.
- Kernel taskqueues have no direct POSIX analogue; POSIX thread pools are roll-your-own.
- Kernel atomics are more comprehensive (more operations, better barrier control) than older C11 atomics.

A reader comfortable with POSIX synchronisation will find the kernel primitives intuitive with minor adjustments. The main adjustment is the context awareness: kernel code cannot assume the ability to block.

15.31 Reference: A Worked Pattern Catalog

Ten synchronisation patterns that recur in real drivers. Each is a variation on the primitives Chapter 15 introduced.

15.31.1 Pattern 1: Producer/Consumer With Bounded Queue

Chapter 14's `cbuf` plus Chapter 12's `cvs` already implement this. The producer writes, the consumer reads, the mutex protects the queue, the `cvs` signal empty-to-nonempty and full-to-nonfull transitions.

15.31.2 Pattern 2: Completion Semaphore

A submitter initialises a semaphore to 0 and waits. A completion handler posts. The submitter unblocks. Used for request-reply patterns. See `/usr/src/sys/dev/hyperv/storvsc/hv_storvsc_drv_freebsd.c`.

15.31.3 Pattern 3: Admission Control

A semaphore initialised to N. Each participant `sema_waits` on entry and `sema_posts` on exit. Chapter 15 Stage 1 uses this pattern.

15.31.4 Pattern 4: Upgrade-Promote-Downgrade Read Cache

An `sx` lock with a staleness-based refresh. Readers take shared; on stale they `try_upgrade`, `refresh`, `downgrade`. Chapter 15 Stage 2 uses this pattern.

15.31.5 Pattern 5: Atomic Flag Coordination

An atomic flag read by many contexts, written by one. Use `atomic_load_acq` for reads, `atomic_store_rel` for writes. Chapter 15 Stage 4 uses this for `is_attached`.

15.31.6 Pattern 6: At-Most-One State Flag

A state flag protected by a lock or CAS. The “start of operation” path sets the flag; the “end of operation” path clears it. Chapter 15 Stage 4 uses this for `recovery_in_progress`.

15.31.7 Pattern 7: Signal-Interruptible Bounded Wait

`cv_timedwait_sig` with explicit `EINTR/ERESTART/EWOULDBLOCK` handling. Chapter 15 Stage 3 refines this pattern.

15.31.8 Pattern 8: Periodic Refresh via Callout

A callout periodically invokes a refresh function. The refresh holds a lock briefly. Simpler than Pattern 4 when the refresh interval is fixed.

15.31.9 Pattern 9: Deferred Teardown via Reference Count

Object has a reference count. “Free” decrements; actual free happens when the count reaches zero. Atomic decrement ensures correctness.

15.31.10 Pattern 10: Cross-Subsystem Handshake

Two subsystems coordinate via a shared state flag plus a `cv` or `sema`. One signals “my part is done”; the other waits. Useful for staged shutdown.

Knowing these patterns makes reading real driver source faster. Every Chapter 15 primitive is a building block for one or more of these patterns, and every real driver picks the patterns its workload needs.

15.32 Reference: A Glossary of Part 3 Terms

For final reference, a consolidated glossary spanning Chapters 11 through 15.

Atomic operation. A read, write, or read-modify-write primitive that executes without possibility of concurrent interference at the hardware level.

Barrier. A directive that prevents the compiler or CPU from reordering memory operations past a specific point.

Blocking wait. A synchronisation operation that puts the caller to sleep until a condition is met or a timeout fires.

Broadcast. Wake every thread blocked on a `cv` or `sema`.

Callout. A deferred function scheduled to run at a specific tick count in the future.

Coalescing. Folding multiple requests into a single operation (e.g., task enqueues into `ta_pending`).

Condition variable. A primitive that allows a thread to sleep until another thread signals a state change.

Context. The execution environment of a code path (syscall, callout, task, interrupt, etc.) with its own rules about what operations are safe.

Counter. A per-CPU accumulator primitive (`counter(9)`) used for lock-free statistics.

Drain. Wait until a pending operation is no longer pending and not currently executing.

Enqueue. Add a work item to a queue. Typically triggers a wakeup of the consumer.

Epoch. A synchronisation mechanism that allows lock-free reads of shared structures; writers defer reclamation via `epoch_call`.

Exclusive lock. A lock held by at most one thread; writers use exclusive mode.

Filter interrupt. An interrupt handler that runs in hardware context with severe restrictions on what it can do.

Grouptaskqueue. A scalable taskqueue variant with per-CPU worker queues; used by high-rate network drivers.

Interruptible wait. A blocking wait that can be woken by a signal, returning `EINTR` or `ERESTART`.

Memory ordering. Rules about the visibility and order of memory accesses across CPUs.

Mutex. A primitive ensuring mutual exclusion; at most one thread inside the lock at a time.

Partial progress. Bytes already copied on a read or write when an interrupt or timeout fires; conventionally returned as success with a short count.

Priority inheritance. A scheduler mechanism where a high-priority waiter temporarily raises the priority of the current holder.

Release barrier. A barrier on a store that ensures preceding accesses cannot be reordered after the store.

Semaphore. A primitive with a non-negative counter; `post` increments, `wait` decrements (blocking if zero).

Shared lock. A lock held by many threads at once; readers use shared mode.

Spinlock. A lock whose slow path busy-waits rather than sleeping. `MTX_SPIN` mutex.

Sx lock. A shared/exclusive lock; FreeBSD's read-write lock primitive.

Task. A deferred-work item with a callback and context, submitted to a taskqueue.

Taskqueue. A queue of pending tasks serviced by one or more worker threads.

Timeout. A duration beyond which a wait should abandon and return `EWOULDBLOCK`.

Timeout task. A task scheduled for a specific future moment via `taskqueue_enqueue_timeout`.

Upgrade. Promoting a shared `sx` lock to exclusive without releasing it (`sx_try_upgrade`).

Wakeup. Waking a thread blocked on a `cv`, a `sema`, or a sleep queue.

Part 3 ends here. Part 4 begins with Chapter 16, *Hardware Basics and Newbus*.

15.33 Reference: A Worked Debugging Scenario

A narrative walkthrough of a realistic synchronisation bug. Imagine you inherit a driver written by a colleague. The colleague is on holiday. Users report “under heavy load, the driver panics on detach with a stack trace ending in `selwakeup`”.

This section walks through how you would diagnose and fix the problem using the Chapter 15 toolkit.

15.33.1 Step 1: Reproduce

First priority: get a reliable reproduction. Without one, fixes are guesses.

Start by reading the bug report for specifics. “Heavy load” plus “panics on detach” is a strong hint that the race is between a running worker and the detach path. A stack trace ending in `selwakeup` suggests the panic is inside the `selinfo` code.

Write a minimal stress script that reproduces the scenario:

```
#!/bin/sh
kldload ./myfirst.ko
sysctl dev.myfirst.0.tick_source_interval_ms=1
(while ;; do dd if=/dev/myfirst of=/dev/null bs=1 count=100 2>/dev/null; done) &
READER=$!
sleep 5
kldunload myfirst
kill -TERM $READER
```

Run the script in a loop until the panic fires. On a debug kernel, this is the kind of bug that panics within seconds. On a production kernel, it may take longer.

15.33.2 Step 2: Capture the Stack

A debug kernel has KDB enabled. After the panic, you land in the in-kernel debugger. Commands:

- `bt`: stack trace of the panicking thread.
- `show locks`: locks held at the moment of the panic.
- `show witness`: the WITNESS state.

Record everything. On a production crash, a kernel dump is collected; retrieve it with `crashinfo` and analyse offline with `kgdb`.

A typical stack for this bug:

```
panic: selwakeup: freed selinfo
kdb_enter
vpanic
panic
selwakeup
myfirst_selwake_task
taskqueue_run_locked
taskqueue_thread_loop
fork_exit
```

The panic is in `selwakeup`, called from `myfirst_selwake_task`, called from the taskqueue worker. The `selwakeup`’s assertion says the `selinfo` it was given is freed.

15.33.3 Step 3: Identify the Race

The bug is visible in the stack. The task callback called `selwakeup(&sc->rsel)`, but `sc->rsel` had been drained or freed. Two possible causes:

A. The detach path `seldrain`'d the `selinfo` before draining the task. Then the task fired after `seldrain`, against the freed state. B. The detach path freed the `softc` (containing `rsel`) before draining the task. Then the task fired against freed memory.

Both are ordering violations. Look at the actual detach code in the driver:

```
/* Current order in the buggy driver: */
seldrain(&sc->rsel);
seldrain(&sc->wsel);
taskqueue_drain(sc->tq, &sc->selwake_task);
```

The `seldrain` is before the `taskqueue_drain`. That is exactly cause A. The fix is to swap the order.

15.33.4 Step 4: Apply the Fix

Change the detach to:

```
taskqueue_drain(sc->tq, &sc->selwake_task);
seldrain(&sc->rsel);
seldrain(&sc->wsel);
```

The task drain now guarantees the task is not running. Only then does `seldrain` proceed.

But wait. The task could still be enqueued from a running callout. Check: is the callout drained before the task drain? Read the code again:

```
/* Full order after fix: */
MYFIRST_CO_DRAIN(&sc->heartbeat_co);
MYFIRST_CO_DRAIN(&sc->watchdog_co);
MYFIRST_CO_DRAIN(&sc->tick_source_co);
taskqueue_drain(sc->tq, &sc->selwake_task);
seldrain(&sc->rsel);
seldrain(&sc->wsel);
```

Callouts drained, then task drained, then `sel` drained. This is the correct order.

15.33.5 Step 5: Verify

Run the reproduction script again with the fix. The panic should no longer fire. Run it 100 times to gain confidence. On a debug kernel the race surfaces quickly; 100 clean runs is strong evidence the fix is correct.

Add a regression test that exercises this specific scenario. The test should be part of the driver's test kit so the bug does not return.

15.33.6 Step 6: Document

Update `LOCKING.md` with a comment explaining why the order is what it is. A future maintainer who considers reordering the drains for some reason will see the comment and reconsider.

15.33.7 Takeaways

- The bug was visible in the stack trace; the skill was recognising what the trace meant.
- The fix was one line (reorder two calls); the diagnosis was the work.
- The debug kernel made the bug reproducible; without it, the bug would be intermittent and mysterious.
- The test kit prevents regression; without it, a future refactor could reintroduce the bug silently.

Chapter 14 already taught this specific ordering rule. A production driver written by a colleague who had not internalised Chapter 14 could easily have this bug. The chapter's discipline and the Chapter 15 testing chapter together are what keep this bug out of your own code.

This is a short, contrived scenario. Real bugs are subtler. The same methodology applies: reproduce, capture, identify, fix, verify, document. The primitives and the discipline from Part 3 are the toolbox for the “identify” step, which is usually the hardest.

15.34 Reference: Reading a Real Driver's Synchronization

For a concrete, exam-style exercise: pick one Ethernet driver in `/usr/src/sys/dev/` and walk through its synchronisation vocabulary. This section walks through `/usr/src/sys/dev/ale/if_ale.c` briefly as a template for the exercise.

The `ale(4)` driver is a 10/100/1000 Ethernet driver for the Atheros AR8121/AR8113/AR8114. It is not large (a few thousand lines) and has a clean structure.

15.34.1 Primitives It Uses

Open the file and search for the primitives.

```
$ grep -c 'mtx_init\|mtx_lock\|mtx_unlock' /usr/src/sys/dev/ale/if_ale.c
```

The driver uses one mutex (`sc->ale_mtx`). Uniform acquisition via `ALE_LOCK(sc)` and `ALE_UNLOCK(sc)` macros.

It uses callouts: `sc->ale_tick_ch` for periodic link-state polling.

It uses a taskqueue: `sc->ale_tq`, created with `taskqueue_create_fast`, started with `taskqueue_start_threads(..., 1, PI_NET, ...)`. The fast variant (spin-mutex-backed) is used because the interrupt filter enqueues onto it.

It uses a task on the queue: `sc->ale_int_task` for interrupt post-processing.

It does not use sema or sx. The driver's invariants fit a single mutex.

It uses atomics: several `atomic_set_32` and `atomic_clear_32` calls on hardware registers (via `CSR_WRITE_4` and similar). These are for hardware register manipulation, not driver-level coordination.

15.34.2 Patterns It Demonstrates

Filter-plus-task interrupt split. `ale_intr` is the filter, which masks the IRQ at the hardware and enqueues the task. `ale_int_task` is the task, which processes the interrupt work in thread context.

Callout for link polling. `ale_tick` is a periodic callout that re-arms itself, used for link-state polling.

Standard detach ordering. `ale_detach` drains callouts, drains tasks, frees the taskqueue, destroys the mutex. Same pattern as `myfirst`.

15.34.3 What It Does Not Demonstrate

- No `sx` lock. Configuration is protected by the single mutex.
- No semaphore. No bounded admission.
- No epoch. The driver does not touch network state directly from unusual contexts.
- No `sx_try_upgrade`. No read-mostly cache.

15.34.4 Takeaways

The `ale(4)` driver uses the subset of Part 3 primitives that its workload needs. A driver that needed an `sx` lock or a sema would add it; `ale(4)` does not, so it does not.

Reading one real driver like this is worth more than reading the chapter twice. Pick a driver, read it for 30 minutes, write down what primitives it uses and why.

Do the same exercise with a larger driver (`bge(4)`, `iwm(4)`, `mfx5(4)` are good candidates). Notice how the vocabulary scales. A driver with more state needs more primitives; a driver with simpler state uses fewer.

The `myfirst` driver at the end of Chapter 15 uses every primitive Part 3 introduced. Most real drivers use a subset. Both are valid; the choice depends on the workload.

15.35 Reference: The Full `myfirst_sync.h` Design

The companion source under `examples/part-03/ch15-more-synchronization/stage4-final/` includes the complete `myfirst_sync.h`. For reference, here is a full version that can be used as a template.

```
/*-
 * SPDX-License-Identifier: BSD-2-Clause
 *
 * Copyright (c) 2026 Edson Brandi
 *
 * myfirst_sync.h: the named synchronisation vocabulary of the
 * myfirst driver.
 *
 * Every primitive the driver uses has a wrapper here. The main
 * source calls these wrappers; the wrappers are inlined away, so
```

```
* the runtime cost is zero. The benefit is a readable,
* centralised, and easily-changeable synchronisation strategy.
*
* This file depends on the definition of `struct myfirst_softc`
* in myfirst.c, which must be included before this header.
*/

#ifndef MYFIRST_SYNC_H
#define MYFIRST_SYNC_H

#include <sys/param.h>
#include <sys/system.h>
#include <sys/lock.h>
#include <sys/mutex.h>
#include <sys/sx.h>
#include <sys/sem.h>
#include <sys/condvar.h>

/*
 * Data-path mutex. Single per-softc. Protects cbuf, counters, and
 * most of the per-softc state.
 */
static __inline void
myfirst_sync_lock(struct myfirst_softc *sc)
{
    mtx_lock(&sc->mtx);
}

static __inline void
myfirst_sync_unlock(struct myfirst_softc *sc)
{
    mtx_unlock(&sc->mtx);
}

static __inline void
myfirst_sync_assert_locked(struct myfirst_softc *sc)
{
    mtx_assert(&sc->mtx, MA_OWNED);
}

/*
 * Configuration sx. Protects the myfirst_config structure. Read
 * paths take shared; sysctl writers take exclusive.
 */
```

```
static __inline void
myfirst_sync_cfg_read_begin(struct myfirst_softc *sc)
{
    sx_slock(&sc->cfg_sx);
}

static __inline void
myfirst_sync_cfg_read_end(struct myfirst_softc *sc)
{
    sx_sunlock(&sc->cfg_sx);
}

static __inline void
myfirst_sync_cfg_write_begin(struct myfirst_softc *sc)
{
    sx_xlock(&sc->cfg_sx);
}

static __inline void
myfirst_sync_cfg_write_end(struct myfirst_softc *sc)
{
    sx_xunlock(&sc->cfg_sx);
}

/*
 * Writer-cap semaphore. Caps concurrent writers at
 * sc->writers_limit. Returns 0 on success, EAGAIN if O_NONBLOCK
 * and semaphore is exhausted, ENXIO if detach happened while
 * blocked.
 */
static __inline int
myfirst_sync_writer_enter(struct myfirst_softc *sc, int ioflag)
{
    if (ioflag & IO_NDELAY) {
        if (!sema_trywait(&sc->writers_sema)) {
            mtx_lock(&sc->mtx);
            sc->writers_trywait_failures++;
            mtx_unlock(&sc->mtx);
            return (EAGAIN);
        }
    } else {
        sema_wait(&sc->writers_sema);
    }
    if (!atomic_load_acq_int(&sc->is_attached)) {
```

```
        sema_post(&sc->writers_sema);
        return (ENXIO);
    }
    return (0);
}

static __inline void
myfirst_sync_writer_leave(struct myfirst_softc *sc)
{
    sema_post(&sc->writers_sema);
}

/*
 * Stats cache sx. Protects a small cached statistic.
 */
static __inline void
myfirst_sync_stats_cache_read_begin(struct myfirst_softc *sc)
{
    sx_slock(&sc->stats_cache_sx);
}

static __inline void
myfirst_sync_stats_cache_read_end(struct myfirst_softc *sc)
{
    sx_sunlock(&sc->stats_cache_sx);
}

static __inline int
myfirst_sync_stats_cache_try_promote(struct myfirst_softc *sc)
{
    return (sx_try_upgrade(&sc->stats_cache_sx));
}

static __inline void
myfirst_sync_stats_cache_downgrade(struct myfirst_softc *sc)
{
    sx_downgrade(&sc->stats_cache_sx);
}

static __inline void
myfirst_sync_stats_cache_write_begin(struct myfirst_softc *sc)
{
    sx_xlock(&sc->stats_cache_sx);
}
```

```
static __inline void
myfirst_sync_stats_cache_write_end(struct myfirst_softc *sc)
{
    sx_xunlock(&sc->stats_cache_sx);
}

/*
 * Attach-flag atomic operations. Every context that needs to
 * check "are we still attached?" uses these.
 */
static __inline int
myfirst_sync_is_attached(struct myfirst_softc *sc)
{
    return (atomic_load_acq_int(&sc->is_attached));
}

static __inline void
myfirst_sync_mark_detaching(struct myfirst_softc *sc)
{
    atomic_store_rel_int(&sc->is_attached, 0);
}

static __inline void
myfirst_sync_mark_attached(struct myfirst_softc *sc)
{
    atomic_store_rel_int(&sc->is_attached, 1);
}

#endif /* MYFIRST_SYNC_H */
```

The file is under 200 lines including comments. It names every primitive operation. It adds zero runtime overhead. It is the single place a future maintainer looks to understand or change the synchronisation strategy.

15.36 Reference: Extended Lab: The “Break a Primitive” Exercise

An optional lab that stretches Chapter 15’s material. For each of the primitives Chapter 15 introduced, deliberately break it and observe the failure.

This is pedagogically valuable because seeing the failure mode makes the correct usage concrete.

15.36.1 Break the Writer-Cap

In Stage 1, remove one `sema_post` in the write path (say, on the error-return path). Rebuild. Run the writer-cap test. The `writers_sema_value` sysctl should drift downward over time and never recover. Eventually all writers block. This demonstrates why the post must be on every path.

Restore the post. Verify the drift stops and the sema stays balanced under load.

15.36.2 Break the Sx Upgrade

In Stage 2, remove the re-check after the upgrade fallback:

```
sx_sunlock(&sc->stats_cache_sx);
sx_xlock(&sc->stats_cache_sx);
/* re-check removed */
myfirst_stats_cache_refresh(sc);
sx_downgrade(&sc->stats_cache_sx);
```

Rebuild. Under heavy reader load, the refresh happens multiple times in quick succession because multiple readers race into the fallback path. The refresh counter grows much faster than one per second.

Restore the re-check. Verify the counter settles back to one refresh per second.

15.36.3 Break the Partial-Progress Handling

In Stage 3, remove the partial-progress check on the `EINTR` path:

```
case EINTR:
case ERESTART:
    return (error); /* Partial check removed. */
```

Rebuild. Run the signal-handling lab. A `SIGINT` during a partially-completed read now returns `EINTR` instead of the partial byte count. User-space code expecting the UNIX convention is surprised.

Restore the check. Verify partial-progress works again.

15.36.4 Break the Atomic Read

In Stage 4, replace `atomic_load_acq_int(&sc->is_attached)` with a plain read `sc->is_attached` in the read path's entry check. On x86 this still works (strong memory model). On arm64 it may occasionally miss a detach, producing an `ENXIO`-or-not-`ENXIO` race.

If you do not have arm64 hardware, this one is hard to demonstrate experimentally. Understand it intellectually and move on.

Restore the atomic. The discipline is the same regardless of test outcome.

15.36.5 Break the Detach Ordering

Swap the order of `sel drain` and `taskqueue_drain` in Stage 4's detach (as in the earlier debugging scenario). Run the stress test with detach cycles. Observe the eventual panic.

Restore the correct order. Verify stability.

15.36.6 Break the Sema Destroy Lifetime

Call `sema_destroy(&sc->writers_sema)` early, before the taskqueue is fully drained. On a debug kernel this panics with "sema: waiters" the moment a thread is still inside `sema_wait`. The KASSERT fires.

Restore the correct order. The destroy happens after all waiters have drained.

15.36.7 Why This Matters

Deliberately breaking code is uncomfortable. It is also the fastest way to internalise why the correct code is written the way it is. Each Chapter 15 primitive has failure modes; seeing them live makes the correct usage unforgettable.

After running the break-and-observe exercise for each primitive, the chapter's material will feel solid. You will know not just what to do, but why, and what happens if you skip a step.

15.37 Reference: When To Split A Driver

A meta-observation that belongs here rather than in a specific section.

Part 3 has developed the `myfirst` driver to a moderate size: about 1200 lines in the main source, plus the `myfirst_sync.h` header, plus the `cbuf`. That is small for a real driver. Real drivers range from 2000 lines (simple device support) to 30000 or more (network drivers with offload support).

At what size does a driver warrant being split into multiple source files?

A few heuristics.

- **Under 1000 lines:** one file. The overhead of multiple files exceeds the readability benefit.
- **1000 to 5000 lines:** one file is still fine. Use clear section markers in the file.
- **5000 to 15000 lines:** two or three files. Typical split: main attach/detach logic in `foo.c`, dedicated subsystem logic (e.g., a ring buffer manager) in `foo_ring.c`, hardware register definitions in `foo_reg.h`.
- **Over 15000 lines:** modular design is required. A header for shared structures; several implementation files for subsystems; a top-level `foo.c` that ties them together.

The `myfirst` driver at Chapter 15 Stage 4 is comfortable as a single file plus a sync header. As later chapters add hardware-specific logic, a natural split will emerge: `myfirst_reg.h` for register definitions, `myfirst_intr.c` for interrupt-related code, `myfirst_io.c` for the data path. Those splits will happen in Part 4 when the hardware story justifies them.

The rule of thumb: split when a file exceeds what you can mentally hold at once. For most readers that is around 2000 to 5000 lines. Split earlier if the subsystem boundary is natural; split later if the code is so interleaved that a split would feel forced.

15.38 Reference: A Final Summary of Part 3

Part 3 was a walkthrough of five primitives and their composition. A final summary frames what has been accomplished.

Chapter 11 introduced concurrency into the driver. One user became many users. The mutex made that safe.

Chapter 12 introduced blocking. Readers and writers could wait for state changes. Condition variables made the wait efficient; sx locks made configuration access scalable.

Chapter 13 introduced time. Callouts let the driver act on its own at chosen moments. Lock-aware callouts and drain-at-detach made the timers safe.

Chapter 14 introduced deferred work. Tasks let callouts and other edge contexts hand work off to threads that could actually do the work. Private taskqueues and coalescing made the primitive efficient.

Chapter 15 introduced the remaining coordination primitives. Semaphores capped concurrency. Refined sx patterns enabled read-mostly caches. Signal-interruptible waits with partial-progress preserved the UNIX conventions. Atomic operations made cross-context flags cheap. Encapsulation made the whole vocabulary readable.

Together, the five chapters built a driver with a complete internal synchronisation story. The driver at 0.9-coordination has no synchronisation feature missing; every invariant it cares about has a named primitive, a named operation in the wrapper header, a named section in `LOCKING.md`, and a test in the stress kit.

Part 4 adds the hardware story. The synchronisation story stays.

15.39 Reference: Chapter 15 Deliverables Checklist

Before closing Chapter 15, confirm all deliverables are in place.

15.39.1 Chapter Content

- ☐ `content/chapters/part3/chapter-15.md` exists.
- ☐ Section 1 through Section 8 are written.
- ☐ Additional Topics section is written.
- ☐ Hands-On Labs are written.
- ☐ Challenge Exercises are written.
- ☐ Troubleshooting Reference is written.
- ☐ Wrapping Up Part 3 is written.
- ☐ Bridge to Chapter 16 is written.

15.39.2 Examples

- ☐ `examples/part-03/ch15-more-synchronization/` directory exists.
- ☐ `stage1-writers-sema/` has a working driver.
- ☐ `stage2-stats-cache/` has a working driver.
- ☐ `stage3-interruptible/` has a working driver.

- ☐ `stage4-final/` has the consolidated driver.
- ☐ Each stage has a `Makefile`.
- ☐ `stage4-final/` has `myfirst_sync.h`.
- ☐ `labs/` has test programs and scripts.
- ☐ `README.md` describes each stage.
- ☐ `LOCKING.md` has the updated synchronisation map.

15.39.3 Documentation

- ☐ The main source has a top-of-file comment summarising the Chapter 15 additions.
- ☐ Every new `sysctl` has a description string.
- ☐ Every new structure field has a comment.
- ☐ `LOCKING.md` sections match the driver.

15.39.4 Testing

- ☐ The stage 4 driver builds cleanly.
- ☐ The stage 4 driver passes WITNESS.
- ☐ The Chapter 11-14 regression tests still pass.
- ☐ The Chapter 15 specific tests pass.
- ☐ Detach under load is clean.

A driver and chapter that pass this checklist are done.

15.40 Reference: An Invitation to Experiment

Before closing, a final invitation.

The Chapter 15 driver is a vehicle for learning, not a shipping product. Every primitive it uses is real; every technique it demonstrates is used in real drivers. But the driver itself is deliberately contrived to exercise the full range of primitives in one place. A real driver typically uses a subset.

As you close Part 3, consider experimenting beyond the chapter's worked examples.

- Add a second type of admission control: a semaphore that caps concurrent readers as well as writers. Does it improve or hurt the system? Why?
- Add a watchdog that times out a single write operation (not the whole driver). Implement it with a timeout task. What edge cases do you encounter?
- Convert the configuration `sx` to a set of atomic fields. Measure the performance difference with DTrace. Which design would you ship? Why?
- Write a userland test harness in C that exercises the driver in ways the shell can't. What primitives do you reach for?
- Read a real driver in `/usr/src/sys/dev/` and identify a single synchronisation decision it made. Do you agree with the decision? What would you have chosen instead, and why?

Each experiment is a day or two of work. Each teaches more than a chapter of text. The myfirst driver is a lab; the FreeBSD source is a library; your own curiosity is the syllabus.

Part 3 has taught you the primitives. The rest is practice. Good luck with Part 4.

15.41 Reference: `cv_signal` vs `cv_broadcast`, Precisely

One recurring question when reading or writing driver code: should a signal wake one waiter (`cv_signal`) or all of them (`cv_broadcast`)? The answer is not always obvious; this reference digs in.

15.41.1 The Semantic Difference

`cv_signal(&cv)` wakes at most one thread currently blocked on the `cv`. If multiple threads are waiting, exactly one of them wakes; the kernel chooses which (usually FIFO, but this is not guaranteed by the API).

`cv_broadcast(&cv)` wakes all threads currently blocked on the `cv`. Every blocked thread wakes and re-contentends for the mutex.

15.41.2 When `cv_signal` Is Correct

Two conditions must both hold for `cv_signal` to be safe.

Any one waiter can satisfy the state change. If you signal because a slot became available in a bounded buffer, and any waiter can take that slot, signal is appropriate. The single wakeup is enough.

All waiters are equivalent. If every waiter is running the same predicate and would respond the same way to the wakeup, signal is appropriate. Waking just one avoids the thundering-herd effect of waking all and having all but one immediately re-sleep.

Classic example: a producer/consumer with one item freshly produced. Waking one consumer is sufficient; waking all would wake many consumers who would then see an empty queue and re-sleep.

15.41.3 When `cv_broadcast` Is Correct

A few specific cases make broadcast the right choice.

Multiple waiters may succeed. If a state change unblocks more than one waiter (for example, “the bounded buffer went from full to 10 slots free”), broadcast wakes them all and each one can try. Signaling only one would leave the others blocked even though progress is possible.

Different waiters have different predicates. If some waiters are waiting for “bytes > 0” and others for “bytes > 100”, signal of one might wake a waiter whose predicate is not satisfied, while another waiter whose predicate is satisfied stays asleep. Broadcast ensures every waiter re-evaluates its own predicate.

Shutdown or state invalidation. When the driver detaches, every waiter must see the change and exit. `cv_broadcast` is required because every waiter must return, not just one.

The Chapter 12-15 driver uses `cv_broadcast` on detach (`cv_broadcast(&sc->data_cv)`, `cv_broadcast(&sc->room_cv)`) for exactly this reason. It uses `cv_signal` on normal buffer-state transitions because each transition unblocks at most one waiter productively.

15.41.4 A Subtle Case: The Reset Sysctl

Chapter 12 added a reset sysctl that clears the cbuf. After reset, the buffer is empty and has full room. Which wakeup is correct?

```
cv_broadcast(&sc->room_cv);  /* Room is now fully available. */
```

The driver uses cv_broadcast. Why not signal? Because the reset unblocked potentially many writers who were all waiting for room. Waking them all lets all of them re-check. A signal would wake only one; the others would stay blocked until a write path signalled on a per-byte basis later.

This is the “multiple waiters may succeed” case. Broadcast is correct.

15.41.5 Cost Consideration

cv_broadcast is more expensive than cv_signal. Every woken thread makes the scheduler do work, and every thread that wakes and immediately re-sleeps pays context-switch overhead. For a cv with many waiters, broadcast can be expensive.

For a cv with one or two typical waiters, the cost difference is negligible. Use whichever is semantically correct.

15.41.6 Rules of Thumb

- **Blocking read wakeup after a byte arrives:** cv_signal. One byte can unblock at most one reader.
- **Blocking write wakeup after bytes drain:** depends on how many bytes drained. If one byte drained, signal is fine. If the buffer was emptied by a reset, broadcast.
- **Detach:** always cv_broadcast. Every waiter must exit.
- **Reset or state invalidation that could unblock many:** cv_broadcast.
- **Normal incremental state change:** cv_signal.

When in doubt, cv_broadcast is correct (just more expensive). Prefer signal when you can prove it suffices.

15.42 Reference: The Rare Case Where You Write Your Own Semaphore

A thought experiment. If sema(9) did not exist, how would you implement a counting semaphore with only a mutex and a cv?

```
struct my_sema {
    struct mtx    mtx;
    struct cv     cv;
    int          value;
};

static void
```

```
my_sema_init(struct my_sema *s, int value, const char *name)
{
    mtx_init(&s->mtx, name, NULL, MTX_DEF);
    cv_init(&s->cv, name);
    s->value = value;
}

static void
my_sema_destroy(struct my_sema *s)
{
    mtx_destroy(&s->mtx);
    cv_destroy(&s->cv);
}

static void
my_sema_wait(struct my_sema *s)
{
    mtx_lock(&s->mtx);
    while (s->value == 0)
        cv_wait(&s->cv, &s->mtx);
    s->value--;
    mtx_unlock(&s->mtx);
}

static void
my_sema_post(struct my_sema *s)
{
    mtx_lock(&s->mtx);
    s->value++;
    cv_signal(&s->cv);
    mtx_unlock(&s->mtx);
}
```

Compact, correct, and functionally identical to `sema(9)` for the simple cases. Reading this makes clear what `sema(9)` is doing internally: exactly this, wrapped in an API.

Why does `sema(9)` exist if it is so simple to write? A few reasons:

- It factors the code out of every driver that would otherwise reinvent it.
- It provides a documented, tested primitive with tracing support.
- It optimises the post to avoid `cv_signal` when no waiters are present.
- It provides a consistent vocabulary for code review.

The same argument applies to every kernel primitive. You could roll your own mutex, cv, sx, taskqueue, callout. You do not because the kernel's primitives are better-tested, better-documented, and better-understood by the community. Use them.

The exception is a primitive that is not in the kernel. If your driver needs a specific synchronisation idiom that no kernel primitive provides, implementing it is justified. Document it carefully.

15.43 Reference: A Parting Observation on Kernel Synchronization

An observation that spans Part 3.

Every kernel synchronisation primitive is built from simpler primitives. At the bottom is a spinlock (technically, a compare-and-swap on a memory location, plus barriers). Above spinlocks are mutexes (spinlocks plus priority inheritance plus sleeping). Above mutexes are condition variables (sleep queues plus mutex handoff). Above cvs are sx locks (cv plus a reader counter). Above sxs are semaphores (cv plus a counter). Above semaphores are higher-level primitives (taskqueues, gtaskqueues, epochs).

Each layer adds a specific capability and hides the complexity below. When you call `sema_wait`, you do not think about the cv inside it, the mutex inside the cv, the spinlock inside the mutex, the CAS inside the spinlock. The abstraction works.

The payoff of this layering is that you can reason about one layer at a time. The payoff of knowing the layering is that when a layer fails, you can descend to the one below and debug.

Part 3 introduced the primitives at each layer in order. Part 4 uses them. If a Part 4 bug baffles you, the diagnosis might require descending: from “taskqueue is stuck” to “the task callback is blocked on a mutex” to “the mutex is held by a thread waiting on a cv” to “the cv is waiting for a state change that will never happen because of a different bug”. The tools for this descent are the primitives you now know.

That is the real payoff of Part 3. Not a specific driver pattern, though that is valuable. Not a specific set of API calls, though those are necessary. The real payoff is a mental model of synchronisation that scales with the complexity of the problem. That model is what will carry you through Part 4 and the rest of the book.

Chapter 15 is complete. Part 3 is complete.

Continue to Chapter 16.

15.44 Reference: A Final Note on Testing Discipline

Every chapter of Part 3 ended with tests. The discipline was consistent: add a primitive, refactor the driver, write a test that exercises it, run the whole regression suite, update `LOCKING.md`.

That discipline is what turns a sequence of chapters into a maintainable body of code. Without it, the driver would be a patchwork of features that work individually and break in combination. With it, every chapter’s additions compose with what came before.

Keep the discipline in Part 4. Hardware introduces new primitives (interrupt handlers, resource allocations, DMA tags) and new failure modes (hardware-level races, DMA corruption, register-ordering surprises). Each addition deserves its own test, its own documentation entry, its own integration into the existing regression suite.

The cost of the discipline is a small amount of extra work per chapter. The benefit is that the driver, at whatever stage of development, is always shippable. You can hand it to a colleague, and it will work. You can put it aside

for six months, come back, and still understand what it does. You can add one more feature without fearing that something unrelated will break.

That is the ultimate payoff of Part 3. Not just primitives; not just patterns; a working discipline.

16 Accessing Hardware

16.1 Reader Guidance & Outcomes

Part 3 ended with a driver that knew how to coordinate itself. The `myfirst` module at version 0.9-coordination has a mutex protecting its data path, a pair of condition variables that let readers and writers wait patiently for the buffer state they need, a shared and exclusive lock protecting its configuration, three callouts that give it internal time, a private taskqueue with three tasks that defer work out of constrained contexts, a counting semaphore that caps concurrent writers, an atomic flag that carries the shutdown story across every context, and a small header that names every synchronisation primitive the driver uses. The chapters that built it introduced seven primitives, tied them together with one detach ordering, and documented the whole story in a living `LOCKING.md`.

What the driver does not yet have is a hardware story. Every invariant it coordinates is internal. Every byte that flows through it originates in user space through `write(2)` or is produced by a callout from its own imagination. Nothing in the driver reaches outside the kernel's own memory. A real FreeBSD driver usually exists because there is a device to talk to: a network card, a storage controller, a serial port, a sensor, a custom FPGA, a GPU. That conversation is what Part 4 is about, and Chapter 16 is where it begins.

Chapter 16's scope is deliberately narrow. It teaches the mental model of hardware I/O and the vocabulary of `bus_space(9)`, the FreeBSD abstraction that lets a single driver talk to a device the same way on every supported architecture. It walks through a simulated register block so the reader can practise register-style access without owning real hardware, and it integrates that simulation into the `myfirst` driver in a way that evolves naturally from Chapter 15 rather than throwing the driver away. It covers the safety rules that MMIO demands (memory barriers, access ordering, locking around shared registers) and shows how to debug and trace register-level access. It ends with a small refactor that separates hardware-access code from the driver's business logic, preparing the file layout every later Part 4 chapter will rely on.

Several questions are deliberately postponed so that Chapter 16 can dwell on the vocabulary itself. Dynamic register behaviour, callout-driven status changes, and fault injection belong to Chapter 17. Real PCI devices, BAR mapping, vendor and device ID matching, `pciconf` and `pci(4)`, and the `newbus` glue that ties PCI drivers to the bus subsystem belong to Chapter 18. Interrupts and their split between filter handlers and interrupt threads are Chapter 19. DMA and bus master programming are Chapter 20 and Chapter 21. Chapter 16 stays inside the ground it can cover well, and it hands off explicitly when a topic deserves its own chapter.

Part 4 opens here, and an opening deserves a small pause. Part 3 taught you how to behave inside the driver when many actors touch shared state. Part 4 teaches how the driver reaches outside itself. An interrupt handler runs in a context Part 3 taught you how to reason about and accesses memory Part 4 will teach you how to map. A register write in Part 4 must respect a lock Part 3 taught you how to manage. The disciplines stack. Chapter 16 is your first practice with `bus_space(9)`; Part 3's discipline is what keeps the practice honest.

16.1.1 Why `bus_space(9)` Earns a Chapter of Its Own

You may already be wondering whether MMIO really needs a whole chapter. Why not jump into Chapter 17's simulation or Chapter 18's real PCI work and pick up the accessor vocabulary in passing? If you have used `bus_space(9)` before, the primitive calls in this chapter will not be new to you.

What Chapter 16 adds is the mental model. Hardware I/O is a topic where a small set of ideas done well pays off across every subsequent chapter, and a small set of ideas done sloppily produces quiet, persistent bugs that are hard to find later. The distinction between memory-mapped I/O and port-mapped I/O is simple once you have it and confusing until you do. The meaning of a `bus_space_tag_t` and a `bus_space_handle_t` is simple once you see what they stand for and opaque until then. The reason memory barriers matter around register access is obvious once you have thought about cache coherence and compiler reordering and irrelevant until the day a driver misbehaves for reasons you cannot explain.

The chapter also earns its place by being the chapter where the `myfirst` driver gains its first hardware-facing layer. Until now, the `softc` held only internal state: a circular buffer, some locks, some counters, some flags. After Chapter 16, the `softc` will hold a simulated register block and the accessors that read and write it. That change in the driver's shape is small but formative. It sets up the file organisation and the locking discipline that every later Part 4 chapter will extend. Skipping Chapter 16 would leave the reader trying to learn register-access idioms in the middle of learning PCI, interrupts, or DMA. Doing them one at a time is kinder.

16.1.2 Where Chapter 15 Left the Driver

A short checkpoint before we go further. Chapter 16 extends the driver produced at the end of Chapter 15 Stage 4, tagged as version `0.9-coordination`. If any of the items below feels uncertain, return to Chapter 15 before starting this chapter.

- Your `myfirst` driver compiles cleanly and identifies itself as version `0.9-coordination`.
- The `softc` holds a data-path mutex (`sc->mtx`), a configuration `sx` (`sc->cfg_sx`), a stats-cache `sx` (`sc->stats_cache_sx`), two condition variables (`sc->data_cv`, `sc->room_cv`), three callouts (`heartbeat_co`, `watchdog_co`, `tick_source_co`), a private taskqueue (`sc->tq`) with four tasks (`selwake_task`, `bulk_writer_task`, `reset_delayed_task`, `recovery_task`), and a counting semaphore (`writers_sema`) that caps concurrent writers.
- A header `myfirst_sync.h` encapsulates every synchronisation operation under named inline functions.
- The lock order `sc->mtx -> sc->cfg_sx -> sc->stats_cache_sx` is documented in `LOCKING.md` and enforced by `WITNESS`.
- `INVARIANTS`, `WITNESS`, `WITNESS_SKIPSPIN`, `DDB`, `KDB`, and `KDB_UNATTENDED` are enabled in your test kernel, and you have built and booted it.
- The Chapter 15 stress kit runs cleanly under the debug kernel.

That driver is what Chapter 16 extends. The additions are again modest in volume: one new header (`myfirst_hw.h`), one new structure inside the `softc` (a simulated register block), a handful of accessor helpers, a small hardware-driven task, and a set of barriers and locks around register access. The mental model change is larger than the line count suggests.

16.1.3 What You Will Learn

Walking away from this chapter, you should be able to:

- Describe what hardware I/O means in a driver context and why a driver cannot usually touch device memory by simply dereferencing a pointer.
- Distinguish memory-mapped I/O (MMIO) from port-mapped I/O (PIO), and explain why MMIO dominates modern FreeBSD drivers on modern platforms.
- Explain what a register is, what an offset is, and what a control or status field means, using the vocabulary real device datasheets use.
- Read a simple register map table and translate it into a C header of offsets and bitmasks.
- Describe the roles of `bus_space_tag_t` and `bus_space_handle_t` and why they are an abstraction rather than just a pointer.
- Recognise the shape of `bus_space_read_*`, `bus_space_write_*`, `bus_space_barrier`, `bus_space_read_multi`, `bus_space_read_region_*`, and their `bus_*` shorthand counterparts defined over a `struct resource`.
- Simulate a register block in kernel memory, wrap access to it behind accessor helpers, and use those helpers to build a small driver-visible device that behaves like a real MMIO device without touching real hardware.
- Integrate simulated register access into the evolving `myfirst` driver, with the data path reading and writing through register accessors instead of touching a raw buffer.
- Identify when a register read has side effects and when a register write does, and why that matters for caching, compiler reordering, and debugging.
- Use `bus_space_barrier` correctly to enforce access ordering where it is needed, and recognise when it is not needed.
- Protect shared register state with the right kind of lock, and avoid busy-wait loops that starve other threads.
- Log register access in a way that helps you debug a driver without drowning the reader in noise.
- Refactor a driver so that its hardware-access layer is a named, documented, testable unit of code.
- Tag the driver as version `0.9-mmio`, update `LOCKING.md` and `HARDWARE.md`, and run the full regression suite with hardware-access enabled.

The list is long; each item is narrow. The point of the chapter is the composition.

16.1.4 What This Chapter Does Not Cover

Several adjacent topics are explicitly deferred so Chapter 16 stays focused.

- **Full hardware simulation with dynamic behaviour.** The simulation in this chapter is static enough to teach the register-access vocabulary. Chapter 17 makes the simulation dynamic, with timers that change status registers, events that flip ready bits, and fault injection paths.
- **The PCI subsystem.** `pci(4)`, vendor and device ID matching, `pciconf`, `pci_enable_busmaster`, BAR mapping, MSI and MSI-X, and power management quirks belong to Chapter 18. Chapter 16 mentions PCI only when it is useful to say “this is where your BAR would come from if this were real”.
- **Interrupt handlers.** `bus_setup_intr(9)`, filter handlers, interrupt threads, `INTR_MPSAFE`, and the filter-plus-task split are Chapter 19. Chapter 16 hints at them only to explain why a read with side effects matters.

- **DMA.** `bus_dma(9)`, `bus_dma_tag_create`, `bus_dmamap_load`, bounce buffers, cache flushing around DMA, and scatter-gather lists are Chapter 20 and Chapter 21.
- **Architecture-specific register access oddities.** Weakly-ordered memory models, arm64 device memory attributes, big-endian byte swapping on MIPS and PowerPC, and non-coherent caches on some embedded platforms are mentioned in passing; a deep treatment belongs to the portability chapters.
- **Real-world driver case studies.** Chapter 16 points at `if_ale.c`, `if_em.c`, and `uart_bus_pci.c` as examples of the patterns taught here, but it does not dissect them at length. Later chapters do that work where it fits their own themes.

Staying inside those lines keeps Chapter 16 a chapter about the vocabulary of hardware access. The vocabulary is what transfers; the specific subsystems are what Chapters 17 through 22 apply the vocabulary to.

16.1.5 Estimated Time Investment

- **Reading only:** three to four hours. The vocabulary is small but it requires thinking carefully about each term.
- **Reading plus typing the worked examples:** seven to nine hours over two sessions. The driver evolves in four stages and each stage is a small but real refactor.
- **Reading plus all labs and challenges:** twelve to fifteen hours over three or four sessions, including stress testing and reading some real FreeBSD drivers.

Section 2 and Section 3 are the densest. If the abstraction of a tag and a handle feels opaque on first pass, that is normal. Stop, re-read the worked mapping in Section 3, and continue when the shape has settled.

16.1.6 Prerequisites

Before starting this chapter, confirm:

- Your driver source matches Chapter 15 Stage 4 (`stage4-final`). The starting point assumes every Chapter 15 primitive, every Chapter 14 taskqueue, every Chapter 13 callout, every Chapter 12 cv and sx, and the Chapter 11 concurrent IO model.
- Your lab machine runs FreeBSD 14.3 with `/usr/src` on disk and matching the running kernel.
- A debug kernel with `INVARIANTS`, `WITNESS`, `WITNESS_SKIPSPIN`, `DDB`, `KDB`, and `KDB_UNATTENDED` is built, installed, and booting cleanly.
- You understand the Chapter 15 detach ordering well enough to extend it without getting lost.
- You are comfortable reading hexadecimal offsets and bitmasks.

If any item above is shaky, fix it now rather than pushing through Chapter 16 and trying to reason from a moving foundation. Hardware-access code is sensitive to small mistakes, and a debug kernel catches most of them on first contact.

16.1.7 How to Get the Most Out of This Chapter

Three habits will pay off quickly.

First, keep `/usr/src/sys/sys/bus.h` and `/usr/src/sys/x86/include/bus.h` bookmarked. The `bus.h` file in `/usr/src/sys/sys/` defines the shorthand `bus_read_*` and `bus_write_*` macros over a `struct resource *`. The per-architecture `bus.h` in `/usr/src/sys/x86/include/` (or its equivalent for your platform) defines the lower-level `bus_space_read_*` and `bus_space_write_*` functions and shows you exactly what they compile to. Reading those two files once takes about thirty minutes and removes almost all the mystery from the chapter.

Second, compare every new accessor to what you would have written with plain C. The exercise “if I did not have `bus_space_read_4`, how would I express this register read?” is instructive. The answer on x86 is usually “a volatile pointer dereference plus a compiler barrier”. Seeing the contrast is how the value of the abstraction becomes concrete: the abstraction is the same code, wrapped in a name that carries portability, tracing, and documentation.

Third, type the changes by hand and run each stage. Hardware-access code is code where muscle memory matters. Typing `sc->regs[MYFIRST_REG_CONTROL]` and `bus_write_4(sc->res, MYFIRST_REG_CONTROL, value)` a dozen times is how the difference between raw access and abstracted access becomes visible at a glance. The companion source under `examples/part-04/ch16-accessing-hardware/` is the reference version, but the muscle memory comes from typing.

16.1.8 Roadmap Through the Chapter

The sections in order are:

1. **What Is Hardware I/O?** The mental model of register access, how the driver talks to the device without touching its internals, and what kinds of resources drivers care about.
2. **Understanding Memory-Mapped I/O (MMIO).** How devices appear as memory regions, why a raw pointer cast is not how a driver reaches them, and what alignment and endianness mean here.
3. **Introduction to `bus_space(9)`.** The tag and handle abstraction, the shape of the read and write functions, and the difference between the `bus_space_*` family and the `bus_*` shorthand over a `struct resource *`.
4. **Simulating Hardware for Testing.** A register block allocated with `malloc(9)`, shaped to resemble a real device, with accessors that mirror `bus_space` semantics. Stage 1 of the Chapter 16 driver.
5. **Using `bus_space` in a Real Driver Context.** Integrating the simulated block into `myfirst`, exposing a tiny read-write register interface through the driver, and demonstrating how a task can change register state over time. Stage 2 begins here.
6. **Safety and Synchronisation with MMIO.** Memory barriers, access ordering, locking around registers, and why busy-wait loops are a mistake. Stage 3 of the driver adds the safety discipline.
7. **Debugging and Tracing Hardware Access.** Logging, DTrace, `sysctl` probes, and the small observability layer that makes register access visible without cluttering the driver.
8. **Refactoring and Versioning Your MMIO-Ready Driver.** The final split into `myfirst_hw.h` and `myfirst.c`, the documentation update, and the version bump to `0.9-mmio`. Stage 4 of the driver.

After the eight sections come hands-on labs, challenge exercises, a troubleshooting reference, a Wrapping Up that closes Part 3’s habits and opens Part 4’s, and a bridge to Chapter 17. The reference-and-cheat-sheet material at the end of the chapter is meant to be re-read as you work through later Part 4 chapters; the vocabulary of Chapter 16 is reused in every one of them.

If this is your first pass, read linearly and do the labs in order. If you are revisiting, Sections 3 and 6 stand alone and make good single-sitting reads.

16.2 Section 1: What Is Hardware I/O?

The Chapter 15 driver's world is small. Everything it cares about, it allocates for itself. The circular buffer is a `cbuf_t` inside the `softc`. The heartbeat is a `struct callout` inside the `softc`. The writers semaphore is a `struct sema` inside the `softc`. Every piece of state the driver touches is memory the kernel allocator has given it. Reading from or writing to that state is a plain C memory access: a field assignment, a pointer dereference, or a call to a helper that does one of those things under a lock.

Hardware changes that world. A hardware device is not kernel memory. It is a separate piece of silicon, usually on a different chip from the CPU, with its own registers, its own buffers, its own internal state, and its own rules for how the CPU may communicate with it. A driver's job is to translate the kernel's view of the world, which is software, into the device's view of the world, which is hardware. The first step in that translation is learning how the CPU and the device talk to each other at all.

This section introduces the mental model. Later sections build on it. The goal of Section 1 is not to make you write any code yet. The goal is to establish the vocabulary and the model clearly enough that everything that follows lands softly.

16.2.1 The Device as a Cooperating Partner

A useful first picture is to think of a hardware device not as an object the driver controls, but as a cooperating partner the driver talks to. The device does some fixed amount of work autonomously. A disk rotates whether the driver is attentive or not. A network card receives packets off the wire without asking permission. A temperature sensor measures temperature continuously. A keyboard controller scans the keyboard matrix on a timer of its own. In every case, the device has internal behaviour the driver cannot directly influence.

What the driver can do is send the device commands and receive the device's status and data. The device exposes a small interface: a set of registers, each with a specific meaning, each with a specific protocol for how the driver may read or write it. The driver writes a value to a control register to tell the device what to do. The driver reads a value from a status register to find out what the device is doing. The driver reads a value from a data register to get data the device has received. The driver writes a value to a data register to send data the device should transmit.

Registers, in this picture, are the conversation. The driver does not call a method on the device and does not pass arguments in the C sense. The driver writes a specific value at a specific offset, and the device reads that write and responds. The device writes a specific value at a specific offset, and the driver reads that write to see what the device is telling it. The protocol is entirely defined by the device's documentation, which is usually a datasheet. The driver's job is to follow the protocol.

That word "partner" is doing a lot of work. Hardware is famously unforgiving. A device does not document every wrong thing the driver could do; it simply does something wrong, or undefined, if the driver breaks the protocol. A driver that writes the wrong value to a control register may brick a device until the next power cycle. A driver that reads a status register before the device has finished a previous command may see stale data and make a bad decision. A driver that does not clear an interrupt flag before returning from an interrupt handler

may leave the device thinking the interrupt is still pending. The partner metaphor is cooperative in intent; the actual relationship is one where the driver must be very polite and very attentive, because the device has no way to object except by misbehaving.

16.2.2 What “Accessing Hardware” Really Means

The phrase “accessing hardware” shows up in every kernel programming text, and it rewards a closer look. What does the CPU actually do when a driver reads a register from a device?

On modern platforms the most common answer is: the CPU issues a memory access to a specific physical address, and the memory controller routes that access to the device rather than to RAM. From the CPU’s point of view, it looks like an ordinary load or store. From the device’s point of view, it is a message from the CPU addressed to a specific internal register. The wiring in between, the memory controller and the bus fabric, is what makes the routing work.

That is memory-mapped I/O. The CPU uses normal load and store instructions. The address happens to be routed to a device instead of to RAM. The device exposes its registers as a range of physical addresses, and the driver reads and writes within that range the way it would read and write any other memory.

An older, less common answer on x86 CPUs is: the CPU issues a special I/O instruction (`in` or `out`) to a specific I/O port number, and the chipset routes that instruction to the device. The CPU is not using a load or store; it is using a dedicated instruction that operates on a separate address space, called I/O port space or port-mapped I/O. This was the original mechanism on x86 and is still in use for some legacy devices, but modern drivers rarely see it except in compatibility paths.

FreeBSD abstracts both mechanisms behind a single API. The driver does not care, most of the time, whether a register is reached through a memory access or through an I/O instruction. The abstraction is `bus_space(9)`, which Section 3 introduces. For now, notice that “accessing hardware” is a physical operation on the CPU that the OS hides behind a software interface. The driver writes `bus_space_write_4(tag, handle, offset, value)`, and the kernel does the right thing depending on the platform and the resource type.

16.2.3 Why a Raw Pointer Cast Is Not How Drivers Reach Hardware

A new reader might reasonably ask: if a device appears as a range of physical addresses, why not just take the address of the device’s registers, cast it to a `volatile uint32_t *`, and dereference it? Technically, on some platforms, that works. Practically, no FreeBSD driver does this, and several real reasons make the raw cast a poor choice.

First, the driver does not know the physical address of the device at compile time. The address is assigned by the bus enumeration code at boot or hotplug, based on the Base Address Registers (BARs) that the device advertises. The driver’s attach routine asks the bus subsystem for the resource; the bus subsystem returns a handle that carries the mapping. The driver then uses the handle through the `bus_space` API. There is no place in the driver’s source where the physical address is a constant.

Second, physical addresses are not virtual addresses. The CPU runs in virtual address mode; dereferencing a pointer reads from virtual memory, not physical memory. The kernel’s `pmap(9)` layer maintains the translation. A driver that wants to dereference a device’s registers needs a virtual mapping into the device’s physical range, with the right cache and access attributes. `bus_space_map` does this. A raw pointer cast does not.

Third, different architectures require different access attributes for device memory than for RAM. Device memory must usually be marked uncached, or weakly-cached, or marked as “device memory” in the MMU’s page tables, so that the CPU does not reorder or cache the accesses in ways that confuse the device. On arm64, device memory pages use the nGnRnE or nGnRE attributes that disable speculative prefetching. On x86, the PAT and MTRR mechanisms mark device regions as uncached or write-combining. A raw pointer cast uses whatever attributes the surrounding virtual mapping happens to have, which is usually cached, which is usually wrong for device registers.

Fourth, `bus_space` carries extra information beyond just where to read and write. The tag encodes which address space (memory or I/O port) is in use. On architectures where the two are different, the tag chooses the correct CPU instruction. On architectures where a driver might map a region with byte-swapping for endianness, the tag encodes that too. The tag-plus-handle interface is a portable way to express “access this device with the right semantics”, where a raw pointer is just “poke this virtual address and hope”.

Fifth, `bus_space` supports tracing, hardware access auditing, and optional sanity checks through the `bus_san(9)` layer when the kernel is built with sanitisers enabled. A raw pointer cast is invisible to those tools. If you ever want to know when your driver read a particular register, `bus_space` can tell you; a raw dereference cannot.

The short version is: FreeBSD drivers use `bus_space` because it abstracts a real problem the driver needs to solve, and the raw pointer cast works on some platforms by accident rather than by design. Accepting the abstraction is inexpensive; refusing it creates bugs that surface weeks after deployment.

16.2.4 Categories of Resources a Driver Cares About

Most drivers deal with a small handful of resource categories. Each has a different access pattern. Chapter 16 focuses on one of them (memory-mapped registers), and the others are covered in later chapters. Knowing the whole catalogue helps you place the current topic.

Memory-mapped I/O (MMIO) registers. A range of device physical addresses, mapped into the kernel’s virtual address space, used to send commands and receive status. Every modern device has at least one MMIO region; most have several. Chapter 16’s focus.

Port-mapped I/O (PIO) registers. A range of I/O port numbers on x86, accessed through the `in` and `out` CPU instructions. Older devices used this as their primary mechanism. Newer devices sometimes expose a small compatibility window through ports (a legacy serial controller, for example) while putting the main interface in MMIO. The `bus_space` API abstracts both behind the same read and write calls, which is why this chapter treats them together.

Interrupts. A signal from the device to the CPU that something has happened. The driver registers an interrupt handler through `bus_setup_intr(9)`, and the kernel arranges for the handler to run when the interrupt line asserts. Chapter 19 covers interrupts.

DMA channels. The device reads or writes directly into system RAM, bypassing the CPU. The driver prepares a DMA descriptor that tells the device which RAM addresses it may use. FreeBSD’s `bus_dma(9)` API manages the mappings, the cache coherence, and the synchronisation. Chapters 20 and 21 cover DMA.

Configuration space. On PCI, a separate address space per device, used to describe the device to the OS. BARs live here, the vendor and device IDs live here, power management state lives here. Most drivers read

configuration space only once, during attach, to discover the device's capabilities. Chapter 18 covers PCI configuration space.

Bus-specific capabilities. MSI, MSI-X, PCIe extended capabilities, hot-plug events, errata workarounds. Bus-specific chapters cover these.

This chapter lives in the MMIO box. The `bus_space` abstraction also covers PIO, so we will see port-mapped paths in passing, but the working example is MMIO throughout.

16.2.5 The Register, Closer Up

A register, in the device's language, is a unit of communication. It has a name, an offset, a width, a set of fields, and a protocol.

The **name** is how the datasheet refers to it. `CONTROL`, `STATUS`, `DATA_IN`, `DATA_OUT`, `INTR_MASK`. Names are for humans; the device does not know them.

The **offset** is the distance from the start of the device's register block to the start of this specific register. Offsets are usually given in hexadecimal. `CONTROL` at `0x00`. `STATUS` at `0x04`. `DATA_IN` at `0x08`. `DATA_OUT` at `0x0c`. `INTR_MASK` at `0x10`. The driver uses offsets in every read and write.

The **width** is how many bits the register carries. Common widths are 8, 16, 32, and 64 bits. A 32-bit register is accessed with `bus_space_read_4` and `bus_space_write_4`, where the 4 is the width in bytes. Mismatching the width is a surprisingly common bug; reading a 32-bit register with an 8-bit access reads only one byte, and on some platforms with byte-lane restrictions it may return the wrong byte or no byte at all.

The **fields** inside a register are sub-bit-ranges that each mean something specific. A 32-bit `CONTROL` register might have an `ENABLE` bit at bit 0, a `RESET` bit at bit 1, a 4-bit `MODE` field at bits 4 through 7, and a 16-bit `THRESHOLD` field at bits 16 through 31, with the remaining bits reserved. The driver uses bit masks and shifts to extract or set specific fields, and the datasheet defines every mask and shift.

The **protocol** is the rules the driver must follow when reading or writing the register. Some protocols are trivial ("write this register with the value you want"). Some are subtle ("set the `ENABLE` bit, then poll the `READY` bit in `STATUS` for up to 100 microseconds, then write the `DATA_IN` register"). Some have side effects that the driver must know about ("reading `STATUS` clears the error bits"). Getting the protocol right is the bulk of driver development time in many hardware projects.

For Chapter 16 the registers are simple, because the device is simulated. But the vocabulary is the vocabulary of real datasheets, and every term introduced here transfers directly to any real device you will meet later.

16.2.6 A First Mental Model: The Control Panel

A useful analogy, only if you keep it disciplined, is that of a control panel on an industrial machine. The machine does its work on its own schedule. The panel exposes knobs the operator can turn to tell the machine what to do, gauges the operator can read to see what the machine is doing, and a few lights that go on and off to signal events. The operator does not reach inside the machine; the operator reaches the machine through the panel.

The driver is the operator. The register block is the control panel. A knob on the panel is a control field in a register. A gauge is a status field. A light is a status bit. The wiring behind the panel is the device's internal logic, which the driver does not see and cannot directly influence. The cable between the operator and the panel

is `bus_space`: it carries the operator's turns of the knob and the gauge's readings back and forth, in a language the operator and the panel both understand.

The analogy breaks down quickly if pushed. Real hardware has timing constraints the panel does not. Real hardware has side effects that the panel does not. Real hardware speaks a protocol that changes when the machine's internal state changes. But for the first pass, the panel is good enough: the driver writes to a control field, the device reacts, the driver reads a status field, the device tells it what happened.

Later sections replace the analogy with more precise mental models: the register block as a window into device state, the MMIO region as a memory range with side-effects, the `bus_space` interface as a platform-aware messenger. For now, the panel is the on-ramp.

16.2.7 Why Simulating Hardware Is a Good First Practice

Chapter 16's teaching strategy is to simulate a device rather than require the reader to have a specific piece of real hardware. The reasons are practical and deliberate.

A reader who is practising register-style access for the first time benefits enormously from an environment where they can see the register values directly. A real PCI device's registers are hidden behind a BAR; the reader can read them with `pciconf -r`, but only for specific known offsets, and the values change based on device state in ways a datasheet may not fully document. A simulated device, by contrast, is a struct in kernel memory. The reader can print its contents with a `sysctl`. The reader can modify it from user space through an `ioctl`. The reader can inspect it in `ddb`. The simulation closes the loop between action and observation, which is what makes the practice effective.

A simulated device is also safe. A real device that receives a wrong register value may lock up, corrupt data, or require a physical power cycle. A simulated device that receives a wrong register value does nothing worse than set a wrong bit in kernel memory; if the driver leaks that, `INVARIANTS` will complain. Beginners benefit from the safety net.

A simulated device is reproducible. Every reader running the Chapter 16 code sees the same register values in the same order. A real device's behaviour depends on firmware version, hardware revision, and environmental conditions. Teaching over a reproducible target is much easier than teaching over the union of every possible target.

Chapter 17 expands the simulation with timers, events, and fault injection. Chapter 18 introduces a real PCI device (typically a `virtio` device in a VM) so the reader can practise the real-hardware path. Chapter 16 lays the ground by teaching the vocabulary against a static simulation, which is the gentlest version of the material.

16.2.8 A Glance at Real Drivers That Use Hardware I/O

Before moving on, a short tour of real FreeBSD drivers that exercise the patterns Chapter 16 teaches. You do not need to read these files yet; they are waypoints you can return to as the chapter unfolds.

`/usr/src/sys/dev/uart/uart_bus_pci.c` is a PCI glue layer for UART (serial) controllers. It shows how a driver finds its PCI device, claims an MMIO resource, and hands the resource to a lower layer that actually drives the hardware. It is small and readable, and it uses `bus_space` only indirectly.

`/usr/src/sys/dev/uart/uart_dev_ns8250.c` is the real UART driver for the classic 8250-family serial controller. It is the file where register reads and writes happen. The register layout is defined in `uart_bus.h` and `uart_dev_ns8250.h`. The reads use the abstraction the chapter teaches.

`/usr/src/sys/dev/ale/if_ale.c` is an Ethernet driver for the Attansic L1E chipset. Its `if_alevar.h` defines `CSR_READ_4` and `CSR_WRITE_4` macros over `bus_read_4` and `bus_write_4`, which is a pattern you will adopt in your own driver by Stage 4 of this chapter.

`/usr/src/sys/dev/e1000/if_em.c` is the driver for Intel's gigabit Ethernet controllers (e1000 family). It is larger and more complex than `if_ale.c`, but it uses the same `bus_space` vocabulary. Its attach path is a good reference for how a non-trivial driver allocates MMIO resources.

`/usr/src/sys/dev/led/led.c` is the LED driver. It is a pseudo-device driver that does not talk to real hardware at all; it exposes a small interface through `/dev/led.NAME` and delegates the actual LED control to whichever driver registered it. Chapter 16's simulated device borrows the shape of this driver: a small, self-contained module with a clear interface and no dependency on external hardware.

These files will reappear throughout Part 4. Chapter 16 uses them as tour points; later chapters dissect them where their patterns are the chapter's focus.

16.2.9 What Comes Next in This Chapter

Section 2 moves from the abstract picture to the specific mechanism of memory-mapped I/O. It explains what a mapping is, why device memory is accessed with different rules than ordinary memory, and how the driver can think about alignment, endianness, and caching. Section 3 introduces `bus_space(9)` itself. Section 4 builds the simulated device. Section 5 integrates it into `myfirst`. Section 6 adds the safety discipline. Sections 7 and 8 finish the chapter with debugging, refactoring, and versioning.

The pace from here is slower than Section 1. Section 1 was meant to be read linearly and absorbed as a whole; the later sections are meant to be read section by section, with breaks to type the code and run it.

16.2.10 Wrapping Up Section 1

Hardware I/O is the activity through which a driver talks to a device. The driver cannot reach inside the device; it can only send commands and read status through a defined register interface. On modern platforms the interface is usually memory-mapped; on x86 there is a legacy port-mapped path as well. The FreeBSD abstraction `bus_space(9)` hides the difference from the driver most of the time. A register is a named, offset-located, width-specific unit of communication, with fields and a protocol that the device's datasheet defines.

Chapter 16's simulation lets you practise the vocabulary without real hardware. Later Part 4 chapters apply the vocabulary to real subsystems. The vocabulary is what transfers, and the rest of this chapter exists to give you that vocabulary with enough depth to use it comfortably.

Section 2 begins by looking closely at memory-mapped I/O.

16.3 Section 2: Understanding Memory-Mapped I/O (MMIO)

Section 1 introduced the idea that a device's registers can be reached through ordinary-looking memory accesses. That idea is worth slowing down for. Memory-mapped I/O is the dominant mechanism on modern

FreeBSD platforms, and understanding it well is what makes every later chapter in Part 4 feel tractable rather than mysterious.

This section answers three closely related questions. How does a device appear in memory? Why must the CPU access that memory with different rules than ordinary memory? What does a driver need to think about when reading and writing a register?

The section works from the ground up: physical addresses, virtual mappings, cache attributes, alignment, and endianness. Each piece is small. The composition is where the subtlety lives.

16.3.1 Physical Addresses and Device Memory

The CPU executes instructions. Each load and store instruction names a virtual address, which the memory management unit (MMU) translates into a physical address. Physical addresses are what the memory controller sees. The memory controller's job is to route the access to the right destination.

For most physical addresses, the destination is DRAM. The controller reads or writes a location in system RAM and returns the result to the CPU. This is the common case. Every `malloc(9)` allocation the driver makes returns kernel memory whose physical address is backed by DRAM.

Some physical address ranges, however, are routed to devices instead. The memory controller is configured at boot (by firmware, usually by the BIOS or UEFI on x86, or by the device tree on arm and the `acpi` tables on everything) to send accesses in certain ranges to specific devices. A PCI device might live at physical address `0xfebf0000` through `0xfebfffff`, a 64 KiB region. An embedded UART might live at `0x10000000` through `0x10000fff`, a 4 KiB region. Whatever the range, an access inside it routes to the device rather than to RAM.

From the CPU's point of view, the access looks identical to an access to RAM. The instruction is the same; the address happens to be elsewhere. From the device's point of view, the access looks like an incoming message: a read at offset *X* of the device's internal register file, or a write of some value at offset *Y*.

The key property is that the same CPU instruction (a load or a store) is being reused for a different purpose. That is where "memory-mapped" in MMIO comes from: the device's interface is mapped into the CPU's memory address space, so memory-access instructions reach it.

Port-mapped I/O, the x86 alternative, uses separate instructions (`in`, `out`, and their wider variants) that target a different address space. The port space has its own 16-bit address range on x86. Modern FreeBSD drivers rarely reach port space directly, because modern devices prefer MMIO, but the abstraction is the same: the driver writes a value at an address, and the address happens to route to a device.

16.3.2 The Virtual Mapping

The CPU does not access physical memory directly. Every memory access goes through the MMU, which translates a virtual address into a physical address using page tables. The kernel maintains those page tables in its `pmap(9)` layer. For a driver to read device registers, it needs a virtual mapping into the device's physical range.

When the kernel's bus subsystem discovers a device and the driver's attach routine requests an MMIO resource, the bus layer does two things. First, it finds the physical address range the device occupies, which the device's Base Address Register (BAR) or the platform's device tree describes. Second, it establishes a virtual mapping

from a fresh kernel virtual address range to that physical range, with appropriate cache and access attributes. The result is a virtual address that, when dereferenced, produces an access at the corresponding physical address, which the memory controller then routes to the device.

The handle that `bus_alloc_resource` returns is (on most platforms) a wrapper around that kernel virtual address. The driver does not typically see the address; it passes the resource handle to `bus_space_read_*` and `bus_space_write_*`, which extract the virtual address internally. But the underlying mechanism is a plain virtual-to-physical mapping, set up once at attach and torn down at detach.

This matters for two reasons. First, it explains why `bus_alloc_resource` is not something a driver can skip. Without the resource allocation, there is no virtual mapping; without a virtual mapping, any attempt to access the device will fault or access random memory. Second, it explains why the virtual address is not a constant: the kernel picks it at attach time, and two boots of the same system may produce different addresses.

16.3.3 Cache Attributes Matter

Memory pages have cache attributes. Ordinary RAM uses “write-back” caching: the CPU caches reads and writes in its L1, L2, and L3 caches, writing back to RAM only when the cache line is evicted or explicitly flushed. Write-back caching is great for performance on RAM, where the memory controller’s job is to preserve whatever value the CPU most recently wrote.

Device memory is different. A device’s registers usually have side effects on read and on write. Reading a STATUS register may consume an event the device has signalled. Writing a DATA_IN register may queue data for transmission. Caching a read of a status register means the CPU returns a stale value on the second read; caching a write of a data register means the write goes to the cache and never reaches the device until the cache happens to evict the line.

For these reasons, device memory pages are marked with different cache attributes than ordinary memory. On x86, the attributes are controlled through the PAT (Page Attribute Table) and MTRR (Memory Type Range Registers). Device memory is typically marked UC (uncached) or WC (write-combining). On arm64, device memory pages use the Device-nGnRnE or Device-nGnRE attributes, which disable caching and speculation. The specific names are architecture-dependent; the principle is the same: the CPU must treat device memory differently from RAM.

`bus_space_map` (or the equivalent path inside `bus_alloc_resource`) knows to request the right cache attributes when it establishes the virtual mapping. A driver that dereferences a raw pointer into a device region without going through `bus_space` skips this step and gets whatever attributes the surrounding mapping happens to have, which is usually wrong.

This is one of the most concrete reasons to use the FreeBSD abstraction: the abstraction encodes a correctness requirement (uncached access to devices) that a raw pointer cast cannot.

16.3.4 Alignment

Hardware registers have alignment requirements. A 32-bit register must be accessed with a 32-bit load or store at an offset that is a multiple of 4. A 64-bit register must be accessed with a 64-bit load or store at an offset that is a multiple of 8. On most architectures, an unaligned access to device memory is either slower (decomposed into multiple smaller accesses by the hardware) or outright illegal (trapping with an alignment fault).

The rule for drivers is simple: when reading or writing a register, use the function whose width matches the register's width, and use the correct offset. If the register is 32 bits wide at offset 0x10, the access is `bus_space_read_4(tag, handle, 0x10)` or `bus_space_write_4(tag, handle, 0x10, value)`. If the register is 16 bits wide at offset 0x08, it is `bus_space_read_2(tag, handle, 0x08)` or `bus_space_write_2(tag, handle, 0x08, value)`. The byte-wide variants `bus_space_read_1` and `bus_space_write_1` exist for 8-bit registers.

Mismatching the width is a common early-stage bug and often silent on x86, which has very permissive alignment rules. On arm64 the same code may fault on first contact. Drivers that are developed on x86 and then ported to arm64 often trip on exactly this issue, which is why the FreeBSD style guidance favours matching widths strictly from the beginning.

There is also an offset alignment rule. The offset must be a multiple of the access width. A 32-bit read at offset 0x10 is fine (0x10 is a multiple of 4). A 32-bit read at offset 0x11 is wrong, even if the device nominally has a register starting there; the hardware will usually refuse or return garbage. This rule is easy to follow when the offsets come from a well-named header; it becomes a trap when offsets are computed arithmetically and the arithmetic is wrong.

16.3.5 Endianness

Device memory and the CPU's native byte order may disagree. A device that originated in a PowerPC or network context may present 32-bit registers in big-endian order, meaning the most significant byte of the register is at the lowest byte address within the register. An x86 CPU is little-endian, so the lowest byte address holds the least significant byte. When the CPU reads the device's big-endian register and interprets it with little-endian semantics, the bytes are in the wrong order.

FreeBSD's `bus_space` family has stream variants (`bus_space_read_stream_*`) and ordinary variants (`bus_space_read_*`). On architectures where the bus tag encodes an endian swap, the ordinary variants swap bytes to produce a host-order value. The stream variants do not swap; they return the bytes in device order. A driver that is reading a device whose registers are in a different endianness than the CPU will use the ordinary variants most of the time, relying on the tag to handle the swap. A driver that is reading a data payload (a stream of bytes whose interpretation depends on the protocol, not on the register layout) may use the stream variants.

On x86 the distinction often does not matter because the bus tag does not encode an endian swap by default. The stream variants are aliases for the ordinary ones in `/usr/src/sys/x86/include/bus.h`:

```
#define bus_space_read_stream_1(t, h, o) bus_space_read_1((t), (h), (o))
#define bus_space_read_stream_2(t, h, o) bus_space_read_2((t), (h), (o))
#define bus_space_read_stream_4(t, h, o) bus_space_read_4((t), (h), (o))
```

The comment in that file explains: "Stream accesses are the same as normal accesses on x86; there are no supported bus systems with an endianness different from the host one." On other architectures, the two families can differ, and a driver that cares about endianness picks the appropriate variant based on what the device expects.

For Chapter 16, the simulation is designed to be host-endian. The chapter's drivers use the ordinary `bus_space_read_*` and `bus_space_write_*` without worrying about byte swaps. Later chapters that deal with real network controllers will revisit the endianness story.

16.3.6 Read and Write Side Effects

One of the most important properties of device memory, and one that trips up drivers that treat it as ordinary memory, is that reads and writes can have side effects.

A write to a control register is, by design, a side effect: writing 1 to the `ENABLE` bit tells the device to start operating. The driver expects that side effect, because that is what the register is for. The subtlety is that the write has a side effect on the device even though the value written is also remembered; a driver that writes `0x00000001` to `CONTROL` and then reads `CONTROL` may see `0x00000001` (if the register echoes the written value) or some other value (if the register echoes the device's current state, which may differ from the last written value).

A read from a status register may also be a side effect. Some devices implement “read-to-clear” semantics, where reading the register returns the current status and, as part of the read, clears pending error bits or interrupt flags. A driver that reads the status twice in close succession may see different values on the two reads, because the first read changed the device's internal state. This is by design; the datasheet says so.

Some registers are **write-only**. Reading them returns a fixed value (often all zeros) and reveals nothing about the device. Writing them has the intended effect. A driver that tries to read a write-only register to check its current value will be misled.

Some registers are **read-only**. Writing them is either ignored or hazardous. A driver that writes a read-only register may do nothing (if the hardware is defensive) or may cause undefined behaviour (if it is not).

Some registers are **read-modify-write** unsafe. A naive update pattern (read the current value, modify one field, write the value back) is safe on a register where the read returns the current contents and the write replaces them. It is unsafe on a register where the read has side effects, where the write has side effects on unintended fields, or where another agent (another CPU, a DMA engine, an interrupt handler) can modify the register between the read and the write.

For Chapter 16 the simulated device has a simple protocol: reads and writes each affect only the specific field the caller changes, and no read has side effects. This is not realistic; Chapter 17 introduces read-to-clear and write-only behaviours. For now, the simplicity is a feature: the reader can focus on the mechanics of access without also juggling the device's protocol quirks.

16.3.7 A Concrete Picture: A Device's Register Block

A concrete example, though invented, makes the picture stick. Imagine a simple temperature-and-fan controller exposed as a 64-byte MMIO region. The register map might look like this:

Offset	Width	Name	Direction	Description
0x00	32 bit	CONTROL	Read/Write	Global enable, reset, and mode bits.
0x04	32 bit	STATUS	Read-only	Device ready, fault, data-available.

Offset	Width	Name	Direction	Description
0x08	32 bit	TEMP_SAMPLE	Read-only	Most recent temperature reading.
0x0c	32 bit	FAN_PWM	Read/Write	Fan PWM duty cycle (0-255).
0x10	32 bit	INTR_MASK	Read/Write	Per-interrupt enable bits.
0x14	32 bit	INTR_STATUS	Read/Clear	Pending interrupt flags (read-to-clear).
0x18	32 bit	DEVICE_ID	Read-only	Fixed identifier; vendor code.
0x1c	32 bit	FIRMWARE_REV	Read-only	Device firmware revision.
0x20- 0x3f	32 bytes	reserved	-	Must be written as zero; reads undefined.

A driver for this device would read `DEVICE_ID` at attach to confirm the hardware is what the driver expects, write `CONTROL` to enable the device, poll `STATUS` to confirm the device is ready, periodically read `TEMP_SAMPLE` to report the temperature, and periodically write `FAN_PWM` to adjust the fan. The interrupt path would read `INTR_STATUS` to see which events are pending (which also clears them) and write `INTR_MASK` during setup to choose which interrupts to enable.

Chapter 16's simulated device borrows heavily from this shape. The simulation has a `CONTROL`, a `STATUS`, a `DATA_IN`, a `DATA_OUT`, an `INTR_MASK`, and an `INTR_STATUS`. It is deliberately a toy; the fields and protocol are chosen so the reader can manipulate them easily from user space through the driver's existing `read(2)` and `write(2)` paths. The register map is kept simple because Chapter 17 will introduce the complexity that real devices add on top.

16.3.8 The Shape of a Register Access

Putting the pieces together, a single register access consists of:

1. The driver has a `bus_space_tag_t` and a `bus_space_handle_t` that together describe a specific device region with specific cache attributes.
2. The driver picks an offset within the region, corresponding to a register defined in the device's datasheet.
3. The driver picks an access width that matches the register's width.
4. The driver calls `bus_space_read_*` or `bus_space_write_*` with tag, handle, offset, and (for writes) value.
5. The kernel's `bus_space` implementation for the current architecture compiles the call down to the appropriate CPU instruction (a `mov` on x86 MMIO, an `inb/outb` on x86 PIO, a `ldr/str` on arm64, and so on).
6. The memory controller or I/O fabric routes the access to the device.
7. The device responds: for a read, it returns the requested value; for a write, it performs the action the register's protocol defines.

The abstraction hides all of this from the driver, most of the time. The driver writes `bus_space_read_4(tag, handle, 0x04)` and gets a 32-bit value back. The machinery between the C call and the device is the kernel's and the hardware's job.

What the driver must remain aware of is the handful of correctness rules: alignment, width, side effects, and access ordering. The chapter revisits ordering in Section 6.

16.3.9 What MMIO Is Not

A short list of things MMIO is not, to clear up common confusions.

MMIO is not DMA. DMA is when the device reads or writes system RAM on its own. MMIO is when the CPU reads or writes the device's registers. Both may be used in the same driver, for different purposes. DMA is faster for bulk data; MMIO is necessary for commands and status. Chapter 20 and Chapter 21 cover DMA.

MMIO is not shared memory. Shared memory (in the POSIX sense) is RAM accessible to multiple processes. MMIO is device memory accessible to the kernel only. User space cannot (and should not) access MMIO directly; the driver mediates.

MMIO is not a block of RAM with the device living behind it. MMIO is a direct interface to the device's internal registers. Reading MMIO does not return kernel memory; it returns whatever the device decides to return at that offset. Writing MMIO does not store a value in kernel memory; it sends a message to the device at that offset.

MMIO is not free. Each access is a transaction on the CPU's bus. On a deep cache hierarchy with high memory latency, a single uncached MMIO read can take hundreds of cycles, because the CPU cannot use the cache and must wait for the device to respond. Drivers that issue thousands of MMIO accesses per operation are usually doing something wrong; most operations can be batched or eliminated.

16.3.10 Wrapping Up Section 2

Memory-mapped I/O is the mechanism by which a modern CPU reaches a device through ordinary load and store instructions, with the address routed to the device instead of to RAM. The kernel's virtual mapping layer and the `bus_space` abstraction together hide the plumbing, but the driver must still be aware of alignment, endianness, cache attributes, and side effects. A register is accessed with a read or write of the right width at the right offset; the kernel compiles the call into the appropriate CPU instruction for the current architecture.

Section 3 introduces `bus_space(9)` itself: the tag, the handle, the read and write functions, and the shape of the API as it appears in every FreeBSD driver that talks to hardware. After Section 3, you will be ready to simulate a register block in Section 4 and start writing code.

16.4 Section 3: Introduction to `bus_space(9)`

`bus_space(9)` is the FreeBSD abstraction for portable hardware access. Every driver that talks to memory-mapped or port-mapped hardware uses it, directly or through a thin wrapper. The abstraction is small: two opaque types, a dozen read and write functions in several widths, a barrier function, and a few helpers for multi-register and region accesses. Section 3 walks through the whole thing in the order a reader would naturally meet it.

The section starts with the two types, moves to the read and write functions, covers the multi and region helpers, introduces the barrier function, and closes with the `bus_*` shorthand defined over a `struct resource *` that

most real drivers use in practice. By the end you will recognise every `bus_space` call you meet in `/usr/src/sys/dev/`, and you will have a mental model for writing your own.

16.4.1 The Two Types: `bus_space_tag_t` and `bus_space_handle_t`

Every `bus_space` call takes a tag and a handle as its first two arguments, in that order. Understanding what each one represents is the first step.

A `bus_space_tag_t` identifies an address space. “Address space” here is narrower than its general usage; it specifically refers to the combination of a bus and an access method. On x86, there are two address spaces: memory and I/O port. Each has its own tag value. On other architectures, there may be more: a memory space with host-endian access, a memory space with swapped-endian access, and so on. The tag tells the `bus_space` functions which rules to apply.

The tag is architecture-specific. On x86, the tag is an integer: 0 for I/O port space (`X86_BUS_SPACE_IO`) and 1 for memory space (`X86_BUS_SPACE_MEM`). On arm64, the tag is a pointer to a structure that describes the bus’s endian and access behaviour. On MIPS, it is yet another shape. Drivers do not usually see these architecture details; they obtain the tag from the bus subsystem (through `rman_get_bustag(resource)` or equivalent) and pass it through without inspection.

A `bus_space_handle_t` identifies a specific region within the address space. It is effectively a pointer, but the pointer’s meaning depends on the tag. For a memory tag on x86, the handle is the kernel virtual address at which the device’s physical range has been mapped. For an I/O port tag on x86, the handle is the I/O port base address. For more elaborate tags, the handle may be a structure or an encoded value. Drivers treat the handle as opaque and pass it through.

The pairing is important. A tag alone does not identify a specific device; it identifies only the address space. A handle alone does not carry the access rules. The pair (tag, handle) together identifies a specific mappable region with specific access rules, and that pair is what the `bus_space_read_*` and `bus_space_write_*` functions operate on.

In practice a driver obtains a `struct resource *` from the bus subsystem at attach time and extracts the tag and handle from it with `rman_get_bustag` and `rman_get_bushandle`. It stores the pair in the `softc`, or it stores the resource pointer and uses the shorthand `bus_read_*` and `bus_write_*` macros that extract the tag and handle internally. Section 5 walks through the real pattern.

16.4.2 Offsets

Every read and write function takes an **offset** inside the region. The offset is a `bus_size_t`, which is typically a 64-bit unsigned integer, measured in bytes from the start of the region. A 32-bit register at the start of a device’s MMIO region has offset 0. A 32-bit register at the next slot has offset 4. A 32-bit register at offset 0x10 is 16 bytes into the region.

Offsets are expressed in bytes regardless of the access width. `bus_space_read_4(tag, handle, 0x10)` reads a 32-bit value starting at byte offset 0x10. `bus_space_read_2(tag, handle, 0x12)` reads a 16-bit value starting at byte offset 0x12. The function’s suffix names the width in bytes, not the offset granularity.

The driver is responsible for ensuring the offset falls within the mapped region. `bus_space` does not check bounds; an out-of-range access is a driver bug that reads or writes whatever happens to be beyond the device’s

mapping, which on most platforms is either unmapped memory (faulting the kernel) or another device's memory (corrupting that device's state). Keep your offsets in headers, derive them from the datasheet, and never compute them arithmetically without bounds-checking the result.

16.4.3 The Read Functions

The basic read functions come in four widths:

```
u_int8_t  bus_space_read_1(bus_space_tag_t tag, bus_space_handle_t handle,
                           bus_size_t offset);
u_int16_t bus_space_read_2(bus_space_tag_t tag, bus_space_handle_t handle,
                           bus_size_t offset);
u_int32_t bus_space_read_4(bus_space_tag_t tag, bus_space_handle_t handle,
                           bus_size_t offset);
uint64_t  bus_space_read_8(bus_space_tag_t tag, bus_space_handle_t handle,
                           bus_size_t offset);
```

The `_1`, `_2`, `_4`, `_8` suffixes are access widths in bytes. `_1` is an 8-bit read, `_2` is a 16-bit read, `_4` is a 32-bit read, `_8` is a 64-bit read. The return type is the corresponding unsigned integer.

Not all widths are supported on all platforms. On x86, `bus_space_read_8` is defined only for `__amd64__` (the 64-bit x86) and only for memory space, not I/O port space. The definition in `/usr/src/sys/x86/include/bus.h` is explicit:

```
#ifdef __amd64__
static __inline uint64_t
bus_space_read_8(bus_space_tag_t tag, bus_space_handle_t handle,
                 bus_size_t offset)
{
    if (tag == X86_BUS_SPACE_IO)
        return (BUS_SPACE_INVALID_DATA);
    return (*(volatile uint64_t *) (handle + offset));
}
#endif
```

A 64-bit I/O port access returns `BUS_SPACE_INVALID_DATA` (all bits set). A 64-bit memory access dereferences the handle plus offset as a `volatile uint64_t *`. The `volatile` qualifier is what stops the compiler from caching or reordering the access.

The 32-bit case is similar:

```
static __inline u_int32_t
bus_space_read_4(bus_space_tag_t tag, bus_space_handle_t handle,
                 bus_size_t offset)
{
```

```

    if (tag == X86_BUS_SPACE_IO)
        return (inl(handle + offset));
    return (*(volatile u_int32_t *)(handle + offset));
}

```

Memory space compiles to a `volatile` dereference. I/O port space compiles to an `inl` instruction that reads a long from an I/O port.

The 16-bit (`inw, *(volatile u_int16_t *)`) and 8-bit (`inb, *(volatile u_int8_t *)`) cases follow the same pattern. On a 64-bit x86, `bus_space_read_4` on a memory region compiles to a single `mov` instruction from the mapped address. The cost of the abstraction at runtime, on this common platform, is literally one instruction's worth of call-frame setup if the inline expands, which it does in release builds.

16.4.4 The Write Functions

The write functions mirror the read functions:

```

void bus_space_write_1(bus_space_tag_t tag, bus_space_handle_t handle,
                      bus_size_t offset, u_int8_t value);
void bus_space_write_2(bus_space_tag_t tag, bus_space_handle_t handle,
                      bus_size_t offset, u_int16_t value);
void bus_space_write_4(bus_space_tag_t tag, bus_space_handle_t handle,
                      bus_size_t offset, u_int32_t value);
void bus_space_write_8(bus_space_tag_t tag, bus_space_handle_t handle,
                      bus_size_t offset, uint64_t value);

```

On x86 memory space, a write compiles to a `volatile` store:

```

static __inline void
bus_space_write_4(bus_space_tag_t tag, bus_space_handle_t bsh,
                 bus_size_t offset, u_int32_t value)
{
    if (tag == X86_BUS_SPACE_IO)
        outl(bsh + offset, value);
    else
        *(volatile u_int32_t *)(bsh + offset) = value;
}

```

Port-mapped I/O compiles to an `outl`. The driver writes the same source line regardless of platform; the kernel's per-architecture `bus.h` does the rest.

As with reads, `bus_space_write_8` to I/O port space on x86 is not supported; the function silently returns without emitting a write. This reflects the hardware: x86 I/O ports are 32-bit at most.

16.4.5 The Multi and Region Helpers

Sometimes a driver wants to read or write many values from or to a single register, or many values across a range of registers. The `bus_space` API provides two families of helpers.

Multi accesses repeatedly access a single register, transferring a buffer of values through it. The register stays at a fixed offset; the buffer is consumed or produced. The use case is a FIFO-style register, where the device's internal queue is exposed through a single address, and reading or writing that address pops or pushes one entry.

```
void bus_space_read_multi_1(bus_space_tag_t tag, bus_space_handle_t handle,
                           bus_size_t offset, u_int8_t *buf, size_t count);
void bus_space_read_multi_2(bus_space_tag_t tag, bus_space_handle_t handle,
                           bus_size_t offset, u_int16_t *buf, size_t count);
void bus_space_read_multi_4(bus_space_tag_t tag, bus_space_handle_t handle,
                           bus_size_t offset, u_int32_t *buf, size_t count);
```

`bus_space_read_multi_4(tag, handle, 0x20, buf, 16)` reads a 32-bit value from offset 0x20 sixteen times, storing each value in successive entries of `buf`. The offset does not change between reads; only the buffer pointer advances.

The write variants mirror the reads:

```
void bus_space_write_multi_1(bus_space_tag_t tag, bus_space_handle_t handle,
                             bus_size_t offset, const u_int8_t *buf, size_t count);
void bus_space_write_multi_2(bus_space_tag_t tag, bus_space_handle_t handle,
                             bus_size_t offset, const u_int16_t *buf, size_t count);
void bus_space_write_multi_4(bus_space_tag_t tag, bus_space_handle_t handle,
                             bus_size_t offset, const u_int32_t *buf, size_t count);
```

Region accesses transfer across a range of offsets. The offset advances each step; the buffer advances each step. The use case is a memory-like region inside the device, such as a block of configuration data or a frame buffer slice.

```
void bus_space_read_region_1(bus_space_tag_t tag, bus_space_handle_t handle,
                             bus_size_t offset, u_int8_t *buf, size_t count);
void bus_space_read_region_4(bus_space_tag_t tag, bus_space_handle_t handle,
                             bus_size_t offset, u_int32_t *buf, size_t count);
void bus_space_write_region_1(bus_space_tag_t tag, bus_space_handle_t handle,
                             bus_size_t offset, const u_int8_t *buf, size_t count);
void bus_space_write_region_4(bus_space_tag_t tag, bus_space_handle_t handle,
                             bus_size_t offset, const u_int32_t *buf, size_t count);
```

`bus_space_read_region_4(tag, handle, 0x100, buf, 16)` reads 16 consecutive 32-bit values starting at offset 0x100 and ending at offset 0x13c, storing them in `buf[0]` through `buf[15]`.

The distinction between multi and region corresponds to two different hardware patterns. A FIFO register at a single offset is a multi; a configuration block that spans many offsets is a region. Using the wrong family leaves the driver doing the wrong thing, even if the loop count matches, so take care to pick the right one.

Chapter 16's simulated device does not use multi or region accesses; the driver addresses registers one at a time. Chapter 17's richer simulation and later PCI-based chapters introduce the multi and region patterns where they apply.

16.4.6 The Barrier Function

`bus_space_barrier` is the function that most drivers forget exists until they need it, and its correct use is one of the quiet disciplines of solid hardware programming.

```
void bus_space_barrier(bus_space_tag_t tag, bus_space_handle_t handle,
                      bus_size_t offset, bus_size_t length, int flags);
```

The function enforces ordering on `bus_space` reads and writes issued before the call, relative to those issued after. The `flags` argument is a bitmask:

- `BUS_SPACE_BARRIER_READ` makes prior reads complete before subsequent reads.
- `BUS_SPACE_BARRIER_WRITE` makes prior writes complete before subsequent writes.
- The two may be OR-ed together to enforce both directions.

The `offset` and `length` parameters describe the region the barrier applies to. On x86 these are ignored; the barrier applies to the whole CPU. On other architectures, a bus bridge may be able to enforce barriers more narrowly, and the parameters are informative.

On x86 specifically, `bus_space_barrier` compiles to a small and well-defined sequence. From `/usr/src/sys/x86/include/bus.h`:

```
static __inline void
bus_space_barrier(bus_space_tag_t tag __unused, bus_space_handle_t bsh __unused,
                  bus_size_t offset __unused, bus_size_t len __unused, int flags)
{
    if (flags & BUS_SPACE_BARRIER_READ)
#ifdef __amd64__
        __asm __volatile("lock; addl $0,0(%%rsp)" : : : "memory");
    #else
        __asm __volatile("lock; addl $0,0(%%esp)" : : : "memory");
    #endif
    else
        __compiler_membar();
}
```

A read barrier on amd64 emits a `lock addl` on the stack, which is a cheap way to issue a full memory fence on x86. A write barrier emits only a compiler barrier (`__compiler_membar()`), because x86 hardware retires

writes in program order and the only reordering a driver can experience on writes is from the compiler. The distinction between “the CPU might reorder this” and “the compiler might reorder this” matters, and the x86 `bus_space_barrier` encodes it with minimum cost.

On arm64 the barrier compiles to a `dsb` or `dmb` instruction depending on the flags, because arm64’s memory model is weaker and actual CPU reordering is possible. The driver’s source does not change; the same `bus_space_barrier` call picks the right instruction for each platform.

When is a barrier required? The rule of thumb is: when the correctness of one register access depends on another access having completed first. Examples:

- A driver writes a command to `CONTROL` and reads the result from `STATUS`. The read must not be speculated before the write. A `bus_space_barrier(tag, handle, 0, 0, BUS_SPACE_BARRIER_WRITE | BUS_SPACE_BARRIER_READ)` between them enforces the ordering.
- A driver clears an interrupt flag in `INTR_STATUS` and expects the clear to reach the device before re-enabling interrupts. A write barrier after the clear, before the re-enable, is the correct discipline.
- A driver posts a DMA descriptor to memory and then writes a “doorbell” register to tell the device to process it. A write barrier between the memory write and the doorbell write is required on weakly-ordered platforms.

On x86, many of these cases are handled by the platform’s strong ordering model, and a driver written without explicit barriers often works. The same driver ported to arm64 may fail subtly. The rule “use barriers when ordering matters” produces portable code; the rule “barriers do nothing on x86 so skip them” produces code that breaks on half of FreeBSD’s supported platforms.

Section 6 of this chapter revisits barriers with worked examples in the simulated driver.

16.4.7 The `bus_*` Shorthand over a `struct resource *`

The `bus_space_*` family takes a tag and a handle. In practice, drivers do not usually carry those around; they carry a `struct resource *`, which is what `bus_alloc_resource_any` returns. The resource structure contains the tag and the handle, among other things. Passing them separately would be noise.

To eliminate the noise, `/usr/src/sys/sys/bus.h` defines a family of shorthand macros that take a `struct resource *` and extract the tag and handle internally:

```
#define bus_read_1(r, o) \
    bus_space_read_1((r)->r_bustag, (r)->r_bushandle, (o))
#define bus_read_2(r, o) \
    bus_space_read_2((r)->r_bustag, (r)->r_bushandle, (o))
#define bus_read_4(r, o) \
    bus_space_read_4((r)->r_bustag, (r)->r_bushandle, (o))
#define bus_write_1(r, o, v) \
    bus_space_write_1((r)->r_bustag, (r)->r_bushandle, (o), (v))
#define bus_write_4(r, o, v) \
    bus_space_write_4((r)->r_bustag, (r)->r_bushandle, (o), (v))
#define bus_barrier(r, o, l, f) \
    bus_space_barrier((r)->r_bustag, (r)->r_bushandle, (o), (l), (f))
```

There are equivalents for `_multi` and `_region` variants, stream variants, and the barrier. The macros cover the same functionality as the underlying `bus_space_*` family, just with a more compact call shape.

Most drivers in `/usr/src/sys/dev/` use the shorthand. A typical usage looks like this, adapted from `if_alevar.h`:

```
#define CSR_READ_4(sc, reg)      bus_read_4((sc)->res[0], (reg))
#define CSR_WRITE_4(sc, reg, val) bus_write_4((sc)->res[0], (reg), (val))
```

The driver defines its own `CSR_READ_4` and `CSR_WRITE_4` macros in terms of `bus_read_4` and `bus_write_4`, abstracting one more layer on top. The softc holds an array of `struct resource *` pointers, and the macros reach through to the first one (the main MMIO region) without the driver having to write out the resource dereference every time.

This is a deliberate pattern. It makes register access statements short and scanable. It centralises the resource reference in one place, so if the driver later maps a second region, only the macros change. And it gives the driver's code a consistent look that anyone familiar with `/usr/src/sys/dev/` will recognise immediately.

Chapter 16's simulated driver adopts this pattern by Stage 4. The early stages use the `bus_space_*` family directly, to keep the mechanism visible; the final refactor wraps the accesses in `CSR_READ_*` and `CSR_WRITE_*` macros the way a production driver would.

16.4.8 Setup and Teardown

A driver that uses `bus_space` does not call `bus_space_map` directly in most cases. Instead, it asks the bus subsystem for a resource through `bus_alloc_resource_any`:

```
int rid = 0;
struct resource *res;

res = bus_alloc_resource_any(dev, SYS_RES_MEMORY, &rid, RF_ACTIVE);
if (res == NULL) {
    device_printf(dev, "cannot allocate MMIO resource\n");
    return (ENXIO);
}
```

The arguments are:

- `dev` is the `device_t` for the driver's device.
- `SYS_RES_MEMORY` selects a memory-mapped resource. `SYS_RES_IOPORT` selects a port-mapped resource. `SYS_RES_IRQ` selects an IRQ (used in Chapter 19).
- `rid` is the "resource ID", the index of the resource within the device's resources. A PCI device's first BAR is usually `rid 0` (which for a legacy PCI device corresponds to the BAR at PCI config offset `0x10`). `rid` is a pointer because the bus subsystem may update it to reflect the actual `rid` it used, though for `_any` allocations on a known `rid`, the value passed in is usually returned unchanged.
- `RF_ACTIVE` tells the bus to activate the resource immediately, which includes establishing the virtual mapping. Without `RF_ACTIVE`, the resource is reserved but not mapped.

On success, `res` is a valid `struct resource *` whose tag and handle can be extracted with `rman_get_bustag(res)` and `rman_get_bushandle(res)`, or whose tag and handle are used implicitly by the `bus_read_*` and `bus_write_*` shorthand macros.

At detach, the driver releases the resource:

```
bus_release_resource(dev, SYS_RES_MEMORY, rid, res);
```

The release undoes the allocation, including tearing down the virtual mapping and marking the range available for reuse.

This is the boilerplate every driver follows for MMIO resources. Chapter 16’s simulated device skips it entirely, because there is no bus to allocate from; the “resource” is a chunk of kernel memory the driver allocated with `malloc(9)`. Chapter 17 introduces a slightly more sophisticated simulation that mimics the allocation path. Chapter 18, when real PCI enters the picture, uses the full `bus_alloc_resource_any` flow.

16.4.9 A First Stand-Alone Example

Even without real hardware, a simple stand-alone program illustrates the shape of a `bus_space` call. Imagine a driver that wants to read the 32-bit `DEVICE_ID` register at offset `0x18` from a device whose MMIO region has been allocated as `res`:

```
uint32_t devid = bus_read_4(sc->res, 0x18);
```

One line. The `sc->res` holds the `struct resource *`. The offset `0x18` comes from the datasheet. The return value is the 32-bit contents of the register.

To write a control value:

```
bus_write_4(sc->res, 0x00, 0x00000001); /* set ENABLE bit */
```

To enforce ordering between the write and a subsequent read:

```
bus_write_4(sc->res, 0x00, 0x00000001);
bus_barrier(sc->res, 0, 0, BUS_SPACE_BARRIER_WRITE | BUS_SPACE_BARRIER_READ);
uint32_t status = bus_read_4(sc->res, 0x04);
```

The barrier ensures the write reaches the device before the read is issued. On x86 the barrier is cheap; on arm64 it emits a fence instruction. The driver does not know or care which; the abstraction handles it.

These are three-line shapes that will appear, with small variations, in every driver you write in Part 4 and beyond. The patterns look identical whether the target is a real network card, a USB controller, a storage adapter, or a simulated device.

16.4.10 A Look at a Real Driver's `bus_space` Usage

To connect the vocabulary to real code, open `/usr/src/sys/dev/ale/if_alevar.h` and scroll to the `CSR_WRITE_*` / `CSR_READ_*` macro block. You will find:

```
#define CSR_WRITE_4(_sc, reg, val)    \
    bus_write_4((_sc)->ale_res[0], (reg), (val))
#define CSR_WRITE_2(_sc, reg, val)    \
    bus_write_2((_sc)->ale_res[0], (reg), (val))
#define CSR_WRITE_1(_sc, reg, val)    \
    bus_write_1((_sc)->ale_res[0], (reg), (val))
#define CSR_READ_2(_sc, reg)          \
    bus_read_2((_sc)->ale_res[0], (reg))
#define CSR_READ_4(_sc, reg)          \
    bus_read_4((_sc)->ale_res[0], (reg))
```

The softc stores an array `ale_res[]` of resources; the macros reach into the first slot. Everywhere else in the driver, a register access reads as `CSR_READ_4(sc, ALE_SOME_REG)` and reads naturally.

Or open `/usr/src/sys/dev/e1000/if_em.c` and search for `bus_alloc_resource_any`. You will find:

```
sc->memory = bus_alloc_resource_any(dev, SYS_RES_MEMORY, &rid, RF_ACTIVE);
```

The resource goes into the softc's memory field; the rest of the driver uses macros over `sc->memory`. The pattern repeats in every driver you will meet in Part 4.

Chapter 16 builds up to this pattern gradually. Stage 1 uses plain struct access to emphasise the mechanics. Stage 2 introduces `bus_space_*` directly against a simulated handle. Stage 3 adds barriers and locking. Stage 4 wraps everything in `CSR_*` macros over a struct `resource *`-compatible pointer, matching the real-driver idiom.

A note on line numbers. The chapter cites FreeBSD source by function, macro, or structure name rather than by line number, because line numbers drift between releases while symbol names survive. For approximate coordinates in FreeBSD 14.3, for orientation only: the `CSR_WRITE_*` macros in `if_alevar.h` sit near line 228, `em_allocate_pci_resources` in `if_em.c` near line 2415, `ale_attach` in `if_ale.c` near line 451, and the `ale_attach` resource-alloc and register-read block spans roughly lines 463 to 580. Open the file and jump to the symbol; the line is whatever your editor reports.

16.4.11 Wrapping Up Section 3

`bus_space(9)` is a small, focused abstraction over hardware access. A tag identifies an address space; a handle identifies a specific region within it. Read and write functions come in 8, 16, 32, and 64-bit widths. Multi accesses repeat on a single offset; region accesses walk across offsets. Barriers enforce ordering where it matters. The `bus_*` shorthand over a struct `resource *` is what most drivers use day to day.

The mechanism underneath compiles to CPU instructions that match the platform: a `mov` on x86 MMIO, an `in` or `out` on x86 PIO, a `ldr` or `str` on arm64. The driver writes portable code; the compiler does the translation.

Section 4 now takes you from vocabulary to practice. We build a simulated register block in kernel memory, wrap it with accessor helpers, and start Stage 1 of the Chapter 16 driver refactor.

16.5 Section 4: Simulating Hardware for Testing

Real hardware is a tough teacher. It is expensive to buy, fragile to mishandle, inconsistent across revisions, and unkind to beginners. For Chapter 16's purposes we want something different: an environment where the reader can practise register-style access, see the results, break things safely, and observe what happens. The answer is to simulate a device in kernel memory.

This section builds that simulation from scratch. First a mental model (what does “simulate a device” mean?), then a register map for the device we are going to fake, then the allocation, the accessors, and the first integration with the `myfirst` driver. By the end of Section 4 the driver has Stage 1: a softc that carries a register block, accessors that read and write it, and a couple of sysctls that let you poke at the simulation from user space.

16.5.1 What “Simulating Hardware” Means Here

The simulation is deliberately minimal in Section 4. A chunk of kernel memory, allocated once, sized to match a register block, and accessed through functions that look like `bus_space` calls. Reads fetch values from the chunk; writes store values into the chunk. There is no dynamic behaviour yet: no timers changing a status register, no events setting a ready bit, no fault injection. Chapter 17 adds all of that. Section 4 gives you the skeleton.

This narrowness is deliberate. Chapter 16's job is to teach the access mechanism. A richer simulation, where the reader has to reason about both the mechanism and the device's behaviour, would compete for attention with the vocabulary the reader is still learning. Section 4's simulation exists so every register read and write returns a predictable result, which lets the reader focus on correctness of access rather than on whether the device liked the access or not.

A small but important point about the simulation: because the “device” is kernel memory, the reader can inspect it, poke it, and dump it through mechanisms the kernel already provides (`sysctl`, `ddb`, `gdb` on a core dump). This transparency is a pedagogical feature. A real device's registers are only visible through the register interface; the simulated registers are visible through the interface *and* through the allocator. When something goes wrong in the driver, the reader can compare “what the driver thinks the register is” with “what the register actually contains”. That debugging pathway is very educational and will be lost when we eventually point the driver at real hardware.

16.5.2 The Register Map for the Simulated Device

Before allocating anything, decide what the device looks like. Picking a register map up front is exactly what a datasheet does for real hardware, and doing it before writing code is a habit worth building.

The Chapter 16 simulated device is a minimal “widget”: it can accept a command, report a status, receive a single byte of data, and send a single byte back. The register map is:

Offset	Width	Name	Direction	Description
0x00	32 bit	CTRL	Read/Write	Control: enable, reset, mode bits.
0x04	32 bit	STATUS	Read-only	Status: ready, busy, error, data available.
0x08	32 bit	DATA_IN	Write-only	Data written to the device for processing.
0x0c	32 bit	DATA_OUT	Read-only	Data the device has produced.
0x10	32 bit	INTR_MASK	Read/Write	Interrupt enable mask.
0x14	32 bit	INTR_STATUS	Read/Clear	Pending interrupt flags (read-to-clear, Chapter 17).
0x18	32 bit	DEVICE_ID	Read-only	Fixed identifier: 0x4D594649 ('MYFI').
0x1c	32 bit	FIRMWARE_REV	Read-only	Firmware revision: encoded as major«16
0x20	32 bit	SCRATCH_A	Read/Write	Free scratch register. Always echoes writes.
0x24	32 bit	SCRATCH_B	Read/Write	Free scratch register. Always echoes writes.

The total size is 40 bytes of register space, which we round up to 64 bytes to give ourselves room to grow in Chapter 17.

For Chapter 16, all register access is simplified to direct read-and-write on kernel memory. Read-to-clear semantics on `INTR_STATUS`, write-only semantics on `DATA_IN`, and the behaviour of `CTRL` on reset are deferred to Chapter 17. For now, `DATA_IN` echoes whatever the driver wrote; `INTR_STATUS` holds whatever value the driver last set; and the whole block behaves like a plain block of 32-bit slots.

This is deliberate. Chapter 16 is teaching register access. Chapter 17 introduces the protocol layer. Splitting the two keeps each chapter focused.

16.5.3 The Register Offsets Header

A real driver separates register offsets into a header so the datasheet mapping lives in one place. The Chapter 16 driver follows the same discipline. Create a file `myfirst_hw.h` alongside `myfirst.c`:

```
/* myfirst_hw.h -- Chapter 16 simulated register definitions. */
#ifndef _MYFIRST_HW_H
#define _MYFIRST_HW_H

/* Register offsets for the simulated myfirst widget. */
#define MYFIRST_REG_CTRL      0x00
#define MYFIRST_REG_STATUS    0x04
#define MYFIRST_REG_DATA_IN    0x08
#define MYFIRST_REG_DATA_OUT  0x0c
#define MYFIRST_REG_INTR_MASK 0x10
#define MYFIRST_REG_INTR_STATUS 0x14
#define MYFIRST_REG_DEVICE_ID 0x18
```

```

#define MYFIRST_REG_FIRMWARE_REV 0x1c
#define MYFIRST_REG_SCRATCH_A    0x20
#define MYFIRST_REG_SCRATCH_B    0x24

/* Total size of the register block. */
#define MYFIRST_REG_SIZE         0x40

/* CTRL register bits. */
#define MYFIRST_CTRL_ENABLE      0x00000001u /* bit 0: device enabled */
#define MYFIRST_CTRL_RESET       0x00000002u /* bit 1: reset (write 1 to) */
#define MYFIRST_CTRL_MODE_MASK   0x000000f0u /* bits 4..7: operating mode */
#define MYFIRST_CTRL_MODE_SHIFT  4
#define MYFIRST_CTRL_LOOPBACK    0x00000100u /* bit 8: loopback DATA_IN -> OUT */

/* STATUS register bits. */
#define MYFIRST_STATUS_READY      0x00000001u /* bit 0: device ready */
#define MYFIRST_STATUS_BUSY       0x00000002u /* bit 1: device busy */
#define MYFIRST_STATUS_ERROR      0x00000004u /* bit 2: error latch */
#define MYFIRST_STATUS_DATA_AV    0x00000008u /* bit 3: DATA_OUT has data */

/* INTR_MASK and INTR_STATUS bits. */
#define MYFIRST_INTR_DATA_AV      0x00000001u /* bit 0: data available */
#define MYFIRST_INTR_ERROR        0x00000002u /* bit 1: error condition */
#define MYFIRST_INTR_COMPLETE     0x00000004u /* bit 2: operation complete */

/* Fixed identifier values. */
#define MYFIRST_DEVICE_ID_VALUE  0x4D594649u /* 'MYFI' in little-endian */
#define MYFIRST_FW_REV_MAJOR      1
#define MYFIRST_FW_REV_MINOR      0
#define MYFIRST_FW_REV_VALUE \
    ((MYFIRST_FW_REV_MAJOR << 16) | MYFIRST_FW_REV_MINOR)

#endif /* _MYFIRST_HW_H */

```

Every offset is a named constant. Every bit mask has a name. Every fixed value has a constant. Later chapters add more registers and more bits; the header grows incrementally. The discipline of “no magic numbers inside the driver’s code” starts here and pays off throughout Part 4.

A note on the `u` suffix on the numeric constants. The `u` makes each constant an unsigned `int`, which is important when the value has the high bit set (32-bit registers use the full `0x80000000` bit, which a plain `int` constant cannot represent portably). Using `u` everywhere keeps the driver consistent; getting into the habit prevents the class of bug where a signed-vs-unsigned mismatch leads to a sign-extended comparison that silently passes or fails.

16.5.4 Allocating the Register Block

With the offsets defined, the driver needs a register block. For the simulation, the block is kernel memory. Add the following to the softc (in `myfirst.c` where the softc is declared):

```
struct myfirst_softc {
    /* ... all existing Chapter 15 fields ... */

    /* Chapter 16: simulated MMIO register block. */
    uint8_t      *regs_buf;      /* malloc'd register storage */
    size_t       regs_size;      /* size of the register region */
};
```

`regs_buf` is a byte pointer to an allocation. Using `uint8_t *` rather than `uint32_t *` makes the per-byte offset arithmetic in the accessors straightforward; we cast to the appropriate width at each access.

Before the allocation itself, a small but useful improvement. The Chapter 15 driver uses `M_DEVBUF`, the kernel's generic driver-memory bucket, for its allocations. That works, but it blurs our driver's footprint with every other driver on the system: `vmstat -m` reports the aggregated usage under `devbuf`, with no way to tell what came from `myfirst`. Chapter 16 is a good moment to introduce a per-driver malloc type. Near the top of `myfirst.c`, alongside the other file-scoped declarations:

```
static MALLOC_DEFINE(M_MYFIRST, "myfirst", "myfirst driver allocations");
```

`MALLOC_DEFINE` registers a new malloc bucket named `myfirst`, with the long description used by `vmstat -m`. Every allocation the driver makes from this chapter onward is tagged with `M_MYFIRST`, so `vmstat -m` can report the driver's total memory use directly. Chapter 15's allocations that previously used `M_DEVBUF` can be migrated to `M_MYFIRST` in the same pass, or left alone; the practical difference is small and the migration is purely cosmetic.

With the type defined, the allocation happens in `myfirst_attach`, before any code that might access the registers:

```
/* In myfirst_attach, after softc initialisation, before registering /dev nodes. */
sc->regs_size = MYFIRST_REG_SIZE;
sc->regs_buf = malloc(sc->regs_size, M_MYFIRST, M_WAITOK | M_ZERO);

/* Initialise fixed registers to their documented values. */
*(uint32_t *) (sc->regs_buf + MYFIRST_REG_DEVICE_ID) = MYFIRST_DEVICE_ID_VALUE;
*(uint32_t *) (sc->regs_buf + MYFIRST_REG_FIRMWARE_REV) = MYFIRST_FW_REV_VALUE;
*(uint32_t *) (sc->regs_buf + MYFIRST_REG_STATUS) = MYFIRST_STATUS_READY;
```

`M_WAITOK | M_ZERO` produces a zero-filled allocation that can sleep to complete if memory is tight, which is fine at attach time. `M_WAITOK` is the right choice because the caller is the kernel's attach path, which is a process context and can block; `M_NOWAIT` would be required only from a callout or filter-interrupt context.

The initialisation writes three fixed values: the device ID, the firmware revision, and an initial STATUS with the READY bit set. A real device would set these through hardware logic at power-on; the simulation does them explicitly in code.

The teardown is symmetric, in `myfirst_detach`:

```
/* In myfirst_detach, after all consumers of regs_buf have quiesced. */
if (sc->regs_buf != NULL) {
    free(sc->regs_buf, M_MYFIRST);
    sc->regs_buf = NULL;
    sc->regs_size = 0;
}
```

As always in the Chapter 11-15 tradition, the free happens after every code path that could touch the memory has finished. By the time we reach this point in detach, the callouts are drained, the taskqueue is drained, the cdev is destroyed, and no syscall can reach the driver.

A subtle but important point: the allocation uses `malloc(9)` rather than `contigmalloc(9)` or `bus_dmamem_alloc(9)`. For simulation, any kernel memory works. For real hardware with DMA requirements, the allocation would need to be physically contiguous, page-aligned, and bounce-buffered as appropriate; that is Chapter 20's topic, not ours.

16.5.5 The First Accessor Helpers

Direct struct access through raw casts `*(uint32_t *) (sc->regs_buf + MYFIRST_REG_CTRL)` works but is ugly, unsafe (no bounds checking), and inconsistent with the `bus_space` idiom the chapter is teaching. Replace it with named accessors.

In `myfirst_hw.h`, add function prototypes and inline definitions:

```
/* Simulated accessor helpers. Stage 1: direct memory, no barriers. */

static __inline uint32_t
myfirst_reg_read(uint8_t *regs_buf, size_t regs_size, bus_size_t offset)
{
    KASSERT(offset + 4 <= regs_size,
        ("myfirst: register read past end of register block: "
         "offset=%#x size=%zu", (unsigned)offset, regs_size));
    return (*(volatile uint32_t *) (regs_buf + offset));
}

static __inline void
myfirst_reg_write(uint8_t *regs_buf, size_t regs_size, bus_size_t offset,
    uint32_t value)
{
    KASSERT(offset + 4 <= regs_size,
```

```

        ("myfirst: register write past end of register block: "
         "offset=%#x size=%zu", (unsigned)offset, regs_size));
    *(volatile uint32_t *)(regs_buf + offset) = value;
}

```

Two helpers: one read, one write. Each bounds-checks the offset with KASSERT so an out-of-range access is caught immediately on a debug kernel. Each uses `volatile` to prevent the compiler from caching or reordering the access. `bus_size_t` is the same type `bus_space` uses for offsets; using it keeps the accessors compatible with the later transition.

A driver that wants to read STATUS from its softc now writes:

```
uint32_t status = myfirst_reg_read(sc->regs_buf, sc->regs_size, MYFIRST_REG_STATUS);
```

Two arguments of boilerplate per call feels like a lot. Real drivers wrap their accessors in shorter macros that take the softc directly. Let us do the same:

```

#define MYFIRST_REG_READ(sc, offset) \
    myfirst_reg_read((sc)->regs_buf, (sc)->regs_size, (offset))
#define MYFIRST_REG_WRITE(sc, offset, value) \
    myfirst_reg_write((sc)->regs_buf, (sc)->regs_size, (offset), (value))

```

Now the register access reads:

```
uint32_t status = MYFIRST_REG_READ(sc, MYFIRST_REG_STATUS);
```

Short, named, scannable. The macros do not add cost beyond the inline expansion the compiler would have done anyway.

One more helper worth introducing for Stage 1. A common operation is “read a register, modify one field, write it back”:

```

static __inline void
myfirst_reg_update(struct myfirst_softc *sc, bus_size_t offset,
    uint32_t clear_mask, uint32_t set_mask)
{
    uint32_t value;

    value = MYFIRST_REG_READ(sc, offset);
    value &= ~clear_mask;
    value |= set_mask;
    MYFIRST_REG_WRITE(sc, offset, value);
}

```

The helper reads the register, clears the bits named in `clear_mask`, sets the bits named in `set_mask`, and writes the result back. A typical use:

```
/* Clear the ENABLE bit in CTRL. */
myfirst_reg_update(sc, MYFIRST_REG_CTRL, MYFIRST_CTRL_ENABLE, 0);

/* Set the ENABLE bit in CTRL. */
myfirst_reg_update(sc, MYFIRST_REG_CTRL, 0, MYFIRST_CTRL_ENABLE);

/* Change MODE to 0x3, keeping other bits intact. */
myfirst_reg_update(sc, MYFIRST_REG_CTRL, MYFIRST_CTRL_MODE_MASK,
    3 << MYFIRST_CTRL_MODE_SHIFT);
```

A caveat: `myfirst_reg_update` as written is not atomic. Between the read and the write, another context could read the same register, modify it, and write it back; our write would then overwrite the other context's update. For Stage 1 this is acceptable, because the simulated registers are accessed only from the syscall context and are not shared with interrupts or tasks yet. Section 6 revisits the atomicity story and introduces locking around the update.

16.5.6 Exposing the Registers Through Sysctls

To make the Stage 1 register block observable without writing a user-space tool, expose each register as a read-only sysctl. In `myfirst_attach`, alongside the other sysctl definitions:

```
/* Chapter 16, Stage 1: sysctls that read the simulated registers. */
SYSCTL_ADD_PROC(&sc->sysctl_ctx,
    SYSCTL_CHILDREN(sc->sysctl_tree), OID_AUTO, "reg_ctrl",
    CTLTYPE_UINT | CTLFLAG_RD | CTLFLAG_MPSAFE, sc, MYFIRST_REG_CTRL,
    myfirst_sysctl_reg, "IU", "Control register (read-only view)");

SYSCTL_ADD_PROC(&sc->sysctl_ctx,
    SYSCTL_CHILDREN(sc->sysctl_tree), OID_AUTO, "reg_status",
    CTLTYPE_UINT | CTLFLAG_RD | CTLFLAG_MPSAFE, sc, MYFIRST_REG_STATUS,
    myfirst_sysctl_reg, "IU", "Status register (read-only view)");

SYSCTL_ADD_PROC(&sc->sysctl_ctx,
    SYSCTL_CHILDREN(sc->sysctl_tree), OID_AUTO, "reg_device_id",
    CTLTYPE_UINT | CTLFLAG_RD | CTLFLAG_MPSAFE, sc, MYFIRST_REG_DEVICE_ID,
    myfirst_sysctl_reg, "IU", "Device ID register (read-only view)");
```

(Equivalent entries for each interesting register follow the same pattern. The `examples/part-04` source has the full list.)

The sysctl handler translates the `arg1/arg2` pair into a register read:


```
static int
myfirst_sysctl_reg(SYSCTL_HANDLER_ARGS)
{
    struct myfirst_softc *sc = arg1;
    bus_size_t offset = arg2;
    uint32_t value;

    if (sc->regs_buf == NULL)
        return (ENODEV);
    value = MYFIRST_REG_READ(sc, offset);
    return (sysctl_handle_int(oidp, &value, 0, req));
}
```

With these sysctls in place, the reader can type:

```
# sysctl dev.myfirst.0.reg_ctrl
dev.myfirst.0.reg_ctrl: 0
# sysctl dev.myfirst.0.reg_status
dev.myfirst.0.reg_status: 1
# sysctl dev.myfirst.0.reg_device_id
dev.myfirst.0.reg_device_id: 1298498121
```

1298498121 in decimal is 0x4D594649, the fixed device ID. 1 in `reg_status` is the `READY` bit. These are the values the attach path set; the reader can see them from user space. The loop from “the driver writes a register” to “the reader observes the register value” is closed.

16.5.7 A Writeable Sysctl for CTRL and DATA_IN

Reading is half the story. The Stage 1 simulation also benefits from a writeable sysctl that lets the reader poke register values:

```
SYSCTL_ADD_PROC(&sc->sysctl_ctx,
    SYSCTL_CHILDREN(sc->sysctl_tree), OID_AUTO, "reg_ctrl_set",
    CTLTYPE_UINT | CTLFLAG_RW | CTLFLAG_MPSAFE, sc, MYFIRST_REG_CTRL,
    myfirst_sysctl_reg_write, "IU",
    "Control register (writeable, Stage 1 test aid)");
```

With the write handler:

```
static int
myfirst_sysctl_reg_write(SYSCTL_HANDLER_ARGS)
{
    struct myfirst_softc *sc = arg1;
```

```

    bus_size_t offset = arg2;
    uint32_t value;
    int error;

    if (sc->regs_buf == NULL)
        return (ENODEV);
    value = MYFIRST_REG_READ(sc, offset);
    error = sysctl_handle_int(oidp, &value, 0, req);
    if (error != 0 || req->newptr == NULL)
        return (error);
    MYFIRST_REG_WRITE(sc, offset, value);
    return (0);
}

```

The handler reads the current value, accepts a new value from the caller, writes it back. Writes are unrestricted for now; later sections add validation and side effects.

The reader can now experiment:

```

# sysctl dev.myfirst.0.reg_ctrl_set
dev.myfirst.0.reg_ctrl_set: 0
# sysctl dev.myfirst.0.reg_ctrl_set=1
dev.myfirst.0.reg_ctrl_set: 0 -> 1
# sysctl dev.myfirst.0.reg_ctrl
dev.myfirst.0.reg_ctrl: 1

```

Setting the `ctrl_set` sysctl to 1 enables the (notional) device by setting the `ENABLE` bit in `CTRL`. Reading `reg_ctrl` confirms it. The loop is now complete: user space writes, the register updates, user space reads, the value matches.

16.5.8 The Observer Pattern: Coupling Registers to Driver State

At this point the driver has a register block that is inert. Writing to `CTRL` does not make the driver do anything. Reading from `STATUS` returns whatever was last written. The registers exist; the driver ignores them.

Stage 1's closing step couples two small pieces of driver state to the register block, so that user-space observation of registers reflects something real.

The first coupling: clear the `READY` bit in `STATUS` while the driver is in a reset state, and set it when the driver is attached and operational. In `myfirst_attach`, immediately after allocating `regs_buf`:

```
MYFIRST_REG_WRITE(sc, MYFIRST_REG_STATUS, MYFIRST_STATUS_READY);
```

In `myfirst_detach`, before freeing `regs_buf`:

```
MYFIRST_REG_WRITE(sc, MYFIRST_REG_STATUS, 0);
```

(In practice, detach's `free(regs_buf)` makes the clearing pointless, but the explicit clear documents the intent and mirrors how a real driver would signal the device that the driver is going away.)

The second coupling: if the user-space caller sets `CTRL.ENABLE`, set a soft flag in the softc that the driver uses to decide whether to emit heartbeat output. If the user clears it, the flag clears. This needs a small change in the writeable sysctl handler and a short routine that applies the change:

```
static void
myfirst_ctrl_update(struct myfirst_softc *sc, uint32_t old, uint32_t new)
{
    if ((old & MYFIRST_CTRL_ENABLE) != (new & MYFIRST_CTRL_ENABLE)) {
        device_printf(sc->dev, "CTRL.ENABLE now %s\n",
            (new & MYFIRST_CTRL_ENABLE) ? "on" : "off");
    }
    /* Other fields will grow in later stages. */
}
```

The writeable sysctl handler calls it after updating the register:

```
static int
myfirst_sysctl_reg_write(SYSCTL_HANDLER_ARGS)
{
    struct myfirst_softc *sc = arg1;
    bus_size_t offset = arg2;
    uint32_t oldval, newval;
    int error;

    if (sc->regs_buf == NULL)
        return (ENODEV);
    oldval = MYFIRST_REG_READ(sc, offset);
    newval = oldval;
    error = sysctl_handle_int(oidp, &newval, 0, req);
    if (error != 0 || req->newptr == NULL)
        return (error);
    MYFIRST_REG_WRITE(sc, offset, newval);

    /* Apply side effects of specific registers. */
    if (offset == MYFIRST_REG_CTRL)
        myfirst_ctrl_update(sc, oldval, newval);

    return (0);
}
```

Now writing 1 to `reg_ctrl_set` produces a `device_printf` in `dmesg` noting the transition. Writing 0 to `reg_ctrl_set` produces another. The register is no longer inert; it drives an observable behaviour.

This is a tiny example of a pattern that will recur: the register is a control surface, the driver reacts to register changes, user-space (or in real drivers, the device) triggers those changes. In Chapter 17 we automate the device side with a callout that changes registers on a timer; in Chapter 18 we point the driver at a real PCI device.

16.5.9 What Stage 1 Accomplished

At the end of Section 4, the driver has:

- A `myfirst_hw.h` header with register offsets, bit masks, and fixed values.
- A `regs_buf` in the `softc`, allocated at `attach` and freed at `detach`.
- Accessor helpers (`myfirst_reg_read`, `myfirst_reg_write`, `myfirst_reg_update`) and macros (`MYFIRST_REG_READ`, `MYFIRST_REG_WRITE`) that wrap access.
- `Sysctls` that expose several registers for read and a writeable `sysctl` for one of them.
- A small coupling between `CTRL.ENABLE` and a driver-level `printf`.

The version tag becomes `0.9-mmio-stage1`. The driver still does everything Chapter 15 gave it; it has simply grown a register-shaped appendage.

Build, load, and test:

```
# cd examples/part-04/ch16-accessing-hardware/stage1-register-map
# make clean && make
# kldload ./myfirst.ko
# sysctl dev.myfirst.0 | grep reg_
# sysctl dev.myfirst.0.reg_ctrl_set=1
# dmesg | tail
# sysctl dev.myfirst.0.reg_ctrl_set=0
# dmesg | tail
# kldunload myfirst
```

You should see the `ENABLE` transitions in `dmesg` and the register values changing through `sysctl`. If any step fails, Section 4's troubleshooting entries at the end of the chapter catch the most common problems.

16.5.10 A Note on What Stage 1 Is Not

Stage 1 is a register-shaped state container. It is not yet using `bus_space(9)` at the API level. The accessors are plain C memory access behind a named helper. Section 5 takes the next step: replace those helpers with real `bus_space_*` calls that operate on the same kernel memory, so the driver's access pattern matches what a real driver with a real resource would look like.

The reason to introduce the abstraction in two passes is pedagogical. Stage 1 makes the mechanism visible: you can see exactly what `MYFIRST_REG_READ` does under the covers. Stage 2 replaces the visible mechanism with the portable API, and you can compare the two and see that the API is doing nothing the helper was not already doing, only more portably and with platform-appropriate barriers where needed. The two-step teaches both.

16.5.11 Wrapping Up Section 4

Simulating hardware starts with a register map and a chunk of kernel memory. A small header declares the offsets, bit masks, and fixed values. A `softc` field holds the allocation. Accessor helpers wrap read and write. `Sysctls` expose the registers so the reader can observe and poke them from user space. Small couplings between registers and driver-level behaviour make the abstraction concrete.

The driver is now at Stage 1 of Chapter 16. Section 5 integrates `bus_space(9)` into this setup, replacing the direct accessors with the portable API the rest of FreeBSD uses.

16.6 Section 5: Using `bus_space` in a Real Driver Context

Section 4 gave the driver a simulated register block and direct accessors. Section 5 replaces the direct accessors with the `bus_space(9)` API and integrates register access into the `myfirst` data path. The shape of the driver changes in a few small but deliberate ways, all of which look more like a real hardware driver than the Stage 1 version did.

This section starts with the smallest possible change (replace the accessor bodies with `bus_space_*` calls) and builds up. By the end, the driver's `write(2)` path produces a register access as a side effect, the driver's `read(2)` path reflects register state, and a task can change register values on a timer so the reader can watch the device “breathe” without user-space poking.

16.6.1 The Pedagogical Trick: Using `bus_space` on Kernel Memory

A minor sleight-of-hand that makes Section 5 possible: on x86, the `bus_space_read_*` and `bus_space_write_*` functions for memory space compile to a plain `volatile` dereference of `handle + offset`. The `handle` is just a `uintptr_t`-shaped value that the functions cast to a pointer. If we set the `handle` to `(bus_space_handle_t)sc->regs_buf`, and the tag to `X86_BUS_SPACE_MEM`, the `bus_space_read_4(tag, handle, offset)` call will do `*(volatile uint32_t*)(regs_buf + offset)`, which is exactly what our Stage 1 accessor did.

That means, on x86 at least, we can drive our simulated register block through the real `bus_space` API by filling in a tag and handle that point at our `malloc'd` memory. The driver's code then becomes indistinguishable, at the source level, from a driver that is accessing real MMIO. That is the whole point: the vocabulary transfers.

On non-x86 platforms, the trick is slightly less clean. On arm64 and some other architectures, `bus_space_tag_t` is a pointer to a structure describing the bus, and using a manufactured tag requires more setup. For Chapter 16 the simulation path is x86-centric; the chapter acknowledges the architectural limitation and defers portability-across-architectures to the later portability chapter. The lessons Chapter 16 teaches are universally applicable; only this one shortcut for simulation is x86-specific.

16.6.2 Stage 2: Setting Up the Simulated Tag and Handle

Add two fields to the `softc`:

```

struct myfirst_softc {
    /* ... all existing fields ... */

    /* Chapter 16 Stage 2: simulated bus_space tag and handle. */
    bus_space_tag_t  regs_tag;
    bus_space_handle_t regs_handle;
};

```

In `myfirst_attach`, after allocating `regs_buf`:

```

#if defined(__amd64__) || defined(__i386__)
sc->regs_tag = X86_BUS_SPACE_MEM;
#else
#error "Chapter 16 simulation path supports x86 only; see text for portability note."
#endif
sc->regs_handle = (bus_space_handle_t)(uintptr_t)sc->regs_buf;

```

That is all the setup. The handle is the virtual address of the allocation cast through `uintptr_t`; the tag is the architecture's constant for memory space.

The `#error` for non-x86 is a deliberate teaching signal: the chapter explicitly flags what is portable (the vocabulary) and what is not (this specific simulation shortcut). Chapter 17 and the portability chapter will teach a cleaner alternative. Until then, x86 is the supported platform for the chapter's exercises.

16.6.3 Replacing the Accessors

With the tag and handle set up, the accessors become one-liners over `bus_space_*`:

```

static __inline uint32_t
myfirst_reg_read(struct myfirst_softc *sc, bus_size_t offset)
{
    KASSERT(offset + 4 <= sc->regs_size,
        ("myfirst: register read past end of block: offset=%#x size=%zu",
         (unsigned)offset, sc->regs_size));
    return (bus_space_read_4(sc->regs_tag, sc->regs_handle, offset));
}

static __inline void
myfirst_reg_write(struct myfirst_softc *sc, bus_size_t offset, uint32_t value)
{
    KASSERT(offset + 4 <= sc->regs_size,
        ("myfirst: register write past end of block: offset=%#x size=%zu",
         (unsigned)offset, sc->regs_size));
}

```

```
bus_space_write_4(sc->regs_tag, sc->regs_handle, offset, value);
}
```

The signature changes: the helpers now take a `struct myfirst_softc *` rather than `regs_buf` and `regs_size` separately. Internal bounds checking is retained; the `KASSERT` fires if a bug in the driver produces an out-of-range offset. The body uses `bus_space_read_4` and `bus_space_write_4` instead of direct memory access.

The `MYFIRST_REG_READ` and `MYFIRST_REG_WRITE` macros simplify accordingly:

```
#define MYFIRST_REG_READ(sc, offset)      myfirst_reg_read((sc), (offset))
#define MYFIRST_REG_WRITE(sc, offset, value) myfirst_reg_write((sc), (offset), (value))
```

Every register access in the driver, including the `sysctl` handlers and `myfirst_reg_update`, continues to use these macros. None of the call sites change. The driver's behaviour is identical to Stage 1, but the access path now travels through `bus_space`, and the path would work just as well if `regs_tag` and `regs_handle` came from `rman_get_bustag` and `rman_get_bushandle` on a real resource.

Build the driver and confirm it still passes the Stage 1 `sysctl` exercises:

```
# cd examples/part-04/ch16-accessing-hardware/stage2-bus-space
# make clean && make
# kldload ./myfirst.ko
# sysctl dev.myfirst.0.reg_device_id
dev.myfirst.0.reg_device_id: 1298498121
# sysctl dev.myfirst.0.reg_ctrl_set=1
# dmesg | tail
# kldunload myfirst
```

The output matches Stage 1. The driver now uses `bus_space` the way a real driver does.

16.6.4 Exposing DATA_IN Through the Write Path

With the accessor layer in place, Stage 2 couples the driver's `write(2)` syscall to the `DATA_IN` register. Every byte written to the device file `/dev/myfirst0` now ends up at the `DATA_IN` register, where the reader can observe it.

Modify `myfirst_write` (the `d_write` callback). The existing handler reads bytes from the `uio`, copies them into the ring buffer, signals waiters, and returns. The new handler does the same, plus: just before returning, it writes the most recently copied byte to `DATA_IN` and sets the `DATA_AV` bit in `STATUS`:

```
static int
myfirst_write(struct cdev *cdev, struct uio *uio, int flag)
{
    struct myfirst_softc *sc = cdev->si_drv1;
```

```

uint8_t buf[MYFIRST_BOUNCE];
size_t n;
int error = 0;
uint8_t last_byte = 0;
bool wrote_any = false;

/* ... existing writer-cap and lock acquisition ... */

while (uio->uio_resid > 0) {
    n = MIN(uio->uio_resid, sizeof(buf));
    error = uiomove(buf, n, uio);
    if (error != 0)
        break;

    /* Remember the most recent byte for the register update. */
    if (n > 0) {
        last_byte = buf[n - 1];
        wrote_any = true;
    }

    /* ... existing copy into the ring buffer ... */
}

/* ... existing unlock and cv_signal ... */

/* Chapter 16 Stage 2: reflect the last byte in DATA_IN. */
if (wrote_any) {
    MYFIRST_REG_WRITE(sc, MYFIRST_REG_DATA_IN,
        (uint32_t)last_byte);
    myfirst_reg_update(sc, MYFIRST_REG_STATUS,
        0, MYFIRST_STATUS_DATA_AV);
}

return (error);
}

```

Now, after any `echo foo > /dev/myfirst0`, the `DATA_IN` register contains the byte value of 'o' (the last character of "foo\n" is \n, actually, which is 0x0a), and the `DATA_AV` bit in `STATUS` is set. The reader can observe this through `sysctl`:

```

# echo -n "Hello" > /dev/myfirst0
# sysctl dev.myfirst.0.reg_data_in
dev.myfirst.0.reg_data_in: 111
# sysctl dev.myfirst.0.reg_status

```



```
dev.myfirst.0.reg_status: 9
```

111 is the ASCII code for 'o', the last byte of “Hello”. 9 is MYFIRST_STATUS_READY | MYFIRST_STATUS_DATA_AV (1 | 8 = 9). The driver has, for the first time, produced an externally-observable register-level side effect in response to user-space action.

16.6.5 Exposing DATA_OUT Through the Read Path

Symmetrically, every byte read from /dev/myfirst0 can update DATA_OUT to reflect what was last read. Modify myfirst_read:

```
static int
myfirst_read(struct cdev *cdev, struct uio *uio, int flag)
{
    struct myfirst_softc *sc = cdev->si_drv1;
    uint8_t buf[MYFIRST_BOUNCE];
    size_t n;
    int error = 0;
    uint8_t last_byte = 0;
    bool read_any = false;

    /* ... existing blocking logic and lock acquisition ... */

    while (uio->uio_resid > 0) {
        /* ... existing ring-buffer extraction ... */

        if (n > 0) {
            last_byte = buf[n - 1];
            read_any = true;
        }

        error = uiomove(buf, n, uio);
        if (error != 0)
            break;
    }

    /* ... existing unlock and cv_signal ... */

    /* Chapter 16 Stage 2: reflect the last byte in DATA_OUT. */
    if (read_any) {
        MYFIRST_REG_WRITE(sc, MYFIRST_REG_DATA_OUT,
            (uint32_t)last_byte);
        /* If the ring buffer is now empty, clear DATA_AV. */
        if (cbuf_is_empty(&sc->cb))
```

```

        myfirst_reg_update(sc, MYFIRST_REG_STATUS,
                           MYFIRST_STATUS_DATA_AV, 0);
    }

    return (error);
}

```

Now DATA_OUT reflects the last byte the reader read, and DATA_AV clears when the ring buffer drains. The loop from “user writes a byte” to “driver updates register” to “user reads a byte” to “driver updates registers” is closed.

Testing:

```

# echo -n "ABC" > /dev/myfirst0
# sysctl dev.myfirst.0.reg_data_in dev.myfirst.0.reg_status
dev.myfirst.0.reg_data_in: 67
dev.myfirst.0.reg_status: 9
# dd if=/dev/myfirst0 bs=1 count=3 of=/dev/null
# sysctl dev.myfirst.0.reg_data_out dev.myfirst.0.reg_status
dev.myfirst.0.reg_data_out: 67
dev.myfirst.0.reg_status: 1

```

67 is 'C', the last byte written. After the dd consumes all three bytes, DATA_OUT holds 'C' (the last byte read) and STATUS is back to just READY because DATA_AV cleared.

16.6.6 Driving Register State From a Task

The register block so far reflects driver actions triggered by user-space syscalls. To illustrate a task-driven pattern, add a small task that periodically increments SCRATCH_A. This is an artificial example; it exists so the reader can see register values changing autonomously in response to task-triggered events, preparing for Chapter 17 where callouts and timers drive more realistic changes.

In the softc:

```

struct task      reg_ticker_task;
int              reg_ticker_enabled;

```

The task callback:

```

static void
myfirst_reg_ticker_cb(void *arg, int pending)
{
    struct myfirst_softc *sc = arg;

```

```

        if (!myfirst_is_attached(sc))
            return;

        MYFIRST_REG_WRITE(sc, MYFIRST_REG_SCRATCH_A,
            MYFIRST_REG_READ(sc, MYFIRST_REG_SCRATCH_A) + 1);
    }

```

The task is enqueued from the existing tick_source callout (Chapter 14's callout that already fires on a timer). In the callout callback, alongside the selwake task enqueue:

```

if (sc->reg_ticker_enabled)
    taskqueue_enqueue(sc->tq, &sc->reg_ticker_task);

```

And a sysctl to enable it:

```

SYSCTL_ADD_INT(&sc->sysctl_ctx,
    SYSCTL_CHILDREN(sc->sysctl_tree), OID_AUTO, "reg_ticker_enabled",
    CTLFLAG_RW, &sc->reg_ticker_enabled, 0,
    "Enable the periodic register ticker (increments SCRATCH_A each tick)");

```

Initialisation in attach:

```

TASK_INIT(&sc->reg_ticker_task, 0, myfirst_reg_ticker_cb, sc);
sc->reg_ticker_enabled = 0;

```

Drain in detach, in the existing task drain block:

```

taskqueue_drain(sc->tq, &sc->reg_ticker_task);

```

With this in place, enabling the ticker produces a visible register effect:

```

# sysctl dev.myfirst.0.reg_ticker_enabled=1
# sleep 5
# sysctl dev.myfirst.0.reg_scratch_a
dev.myfirst.0.reg_scratch_a: 5
# sleep 5
# sysctl dev.myfirst.0.reg_scratch_a
dev.myfirst.0.reg_scratch_a: 10
# sysctl dev.myfirst.0.reg_ticker_enabled=0

```

The register value climbs at one per second, as the tick_source callout fires. The driver is now exhibiting autonomous register-level behaviour, triggered by a task, mediated by bus_space.

16.6.7 The Full Stage 2 Sysctl Tree

After Stage 2 the full sysctl tree under `dev.myfirst.0` looks roughly like:

```
dev.myfirst.0.debug_level
dev.myfirst.0.soft_byte_limit
dev.myfirst.0.nickname
dev.myfirst.0.heartbeat_interval_ms
dev.myfirst.0.watchdog_interval_ms
dev.myfirst.0.tick_source_interval_ms
dev.myfirst.0.bulk_writer_batch
dev.myfirst.0.reset_delayed
dev.myfirst.0.writers_limit
dev.myfirst.0.writers_sema_value
dev.myfirst.0.writers_trywait_failures
dev.myfirst.0.stats_cache_refresh_count
dev.myfirst.0.reg_ctrl
dev.myfirst.0.reg_status
dev.myfirst.0.reg_data_in
dev.myfirst.0.reg_data_out
dev.myfirst.0.reg_intr_mask
dev.myfirst.0.reg_intr_status
dev.myfirst.0.reg_device_id
dev.myfirst.0.reg_firmware_rev
dev.myfirst.0.reg_scratch_a
dev.myfirst.0.reg_scratch_b
dev.myfirst.0.reg_ctrl_set
dev.myfirst.0.reg_ticker_enabled
```

Ten register views, one writeable register, one ticker toggle, plus every previous sysctl from Chapters 11 through 15. The driver has grown, but every addition is small and named.

16.6.8 A Note on Reads From STATUS While the Driver Is Running

In the Stage 1 and Stage 2 setups, reading STATUS via sysctl returns whatever bits the driver most recently set. No read has side effects. This is intentional for Chapter 16. But notice a subtle consequence: the driver can set the STATUS.DATA_AV bit in the write path and clear it in the read path, and the user-space reader can observe the bit change over time. Running `sysctl -w dev.myfirst.0.reg_status=0` is possible through the writeable sysctl, but the driver's automatic updates will re-set the bit on the next write to the device file.

This is how a “polled” device driver works at a conceptual level: the driver polls the status register periodically, reacts to state changes, and updates driver-visible state accordingly. A real device's STATUS bits change for hardware reasons; the simulated device's bits change for simulated-driver reasons. The mechanism is the same.

Chapter 19 introduces interrupts, which replace the polling model with an event-driven one. Until then, polling is a reasonable pattern for the simulated device.

16.6.9 The Detach Path, Updated

Every chapter in Part 3 added a few lines to the detach ordering. Chapter 16 Stage 2 adds two: drain the `reg_ticker_task` and free the register buffer. The full detach order at Stage 2:

1. Refuse detach if `active_fhs > 0`.
2. Clear `is_attached` (atomically), broadcast cvs.
3. Drain all callouts (heartbeat, watchdog, tick_source).
4. Drain all tasks (selwake, bulk_writer, reset_delayed, recovery, reg_ticker).
5. `seldrain(&sc->rsl)`, `seldrain(&sc->wsl)`.
6. `taskqueue_free(sc->tq)`.
7. Destroy cdev.
8. Release sysctl context.
9. **Free `regs_buf`.** (New in Stage 2.)
10. Destroy cbuf, counters, cvs, sx locks, semaphore, mutex.

The `regs_buf` free happens after the sysctl context is torn down, because a sysctl handler could in principle be running on another CPU during detach. After `sysctl_ctx_free`, no sysctl handler can reach the softc, and the free is safe. Real drivers follow the same discipline for their resource releases.

16.6.10 Updating `LOCKING.md` (Now `HARDWARE.md` Too)

Part 3 established `LOCKING.md` as the driver's synchronisation map. Chapter 16 opens a sibling document: `HARDWARE.md`. It lives alongside `LOCKING.md` and documents the register interface, access patterns, and ownership rules for hardware-facing state.

A first cut:

```
# myfirst Hardware Interface
```

```
## Register Block
```

```
Size: 64 bytes (MYFIRST_REG_SIZE).
```

```
Access: 32-bit reads and writes on 32-bit-aligned offsets.
```

```
Allocated in attach, freed in detach.
```

```
### Register Map
```

Offset	Name	Direction	Owner
0x00	CTRL	R/W	driver
0x04	STATUS	R/W	driver
0x08	DATA_IN	W	syscall

0x0c	DATA_OUT	R	syscall	
0x10	INTR_MASK	R/W	driver	
0x14	INTR_STATUS	R/W	driver	
0x18	DEVICE_ID	R	attach	
0x1c	FIRMWARE_REV	R	attach	
0x20	SCRATCH_A	R/W	ticker	
0x24	SCRATCH_B	R/W	free	

Write Protections

Stage 2 does not lock register access. A syscall writer, a syscall writer, and the ticker task can each access the same register without a lock. See Section 6 for the locking story.

Access Paths

- Sysctl read handlers: MYFIRST_REG_READ
- Sysctl write handlers: MYFIRST_REG_WRITE, with side-effect call
- Syscall write path: MYFIRST_REG_WRITE(DATA_IN), myfirst_reg_update(STATUS)
- Syscall read path: MYFIRST_REG_WRITE(DATA_OUT), myfirst_reg_update(STATUS)
- Ticker task: MYFIRST_REG_WRITE(SCRATCH_A)

The document is short now and will grow as later chapters add more registers and more access paths. The discipline of documenting the register interface alongside the code is the same as Part 3's discipline of documenting locks.

16.6.11 Stage 2 Complete

At the end of Section 5, the driver is at 0.9-mmio-stage2. It has:

- A real bus_space_* access path over a simulated tag and handle.
- DATA_IN reflecting the last written byte, with DATA_AV tracking the ring buffer's state.
- DATA_OUT reflecting the last read byte.
- A task that autonomously increments a scratch register on a timer.
- Full syscall visibility into every register.
- A HARDWARE.md document describing the interface.

Build, load, test, observe, unload:

```
# cd examples/part-04/ch16-accessing-hardware/stage2-bus-space
# make clean && make
# kldload ./myfirst.ko
# echo -n "hello" > /dev/myfirst0
# syscall dev.myfirst.0.reg_data_in dev.myfirst.0.reg_status
```

```
# dd if=/dev/myfirst0 bs=1 count=5 of=/dev/null
# sysctl dev.myfirst.0.reg_data_out dev.myfirst.0.reg_status
# sysctl dev.myfirst.0.reg_ticker_enabled=1 ; sleep 3
# sysctl dev.myfirst.0.reg_scratch_a
# sysctl dev.myfirst.0.reg_ticker_enabled=0
# kldunload myfirst
```

The outputs should tell a consistent story about what the driver and the registers are doing.

16.6.12 A Look at a Real Pattern: The `em` Attach Path

Now that Stage 2 mirrors a real driver's structure, a brief look at what Stage 2 would look like if the register block were a real PCI MMIO region, just to anchor the expectation. In `/usr/src/sys/dev/e1000/if_em.c`, inside `em_allocate_pci_resources`, you will find:

```
sc->memory = bus_alloc_resource_any(dev, SYS_RES_MEMORY, &rid, RF_ACTIVE);
if (sc->memory == NULL) {
    device_printf(dev, "Unable to allocate bus resource: memory\n");
    return (ENXIO);
}
sc->osdep.mem_bus_space_tag = rman_get_bustag(sc->memory);
sc->osdep.mem_bus_space_handle = rman_get_bushandle(sc->memory);
sc->hw.hw_addr = (uint8_t *)&sc->osdep.mem_bus_space_handle;
```

The resource is allocated, the tag and handle are extracted into the softc's `osdep` structure, and an additional `hw_addr` pointer is set up for the hardware-abstraction-layer code that Intel shares between drivers. The rest of the driver uses macros (`E1000_READ_REG`, `E1000_WRITE_REG`) defined over `bus_space_*` to talk to the hardware.

The shape is the same as our Stage 2. The difference is exactly one function call: `bus_alloc_resource_any` for a real driver, `malloc(9)` plus a handcrafted tag for ours. Everything above the allocation layer is identical.

Chapter 18 will swap our `malloc` for `bus_alloc_resource_any` and point the driver at a real PCI device. The driver's upper layers will not change.

16.6.13 Wrapping Up Section 5

Stage 2 replaces the direct accessors of Stage 1 with real `bus_space(9)` calls operating on the simulated tag and handle. The driver's `write(2)` and `read(2)` paths now produce register-level side effects. A task updates a scratch register on a timer. The `HARDWARE.md` document describes the register interface. The driver's shape closely matches that of a real driver like `if_em`.

Section 6 introduces the safety discipline that real MMIO demands: memory barriers, locking around shared registers, and the reasons busy-wait loops are a mistake. Stage 3 of the driver adds that discipline.

16.7 Section 6: Safety and Synchronization with MMIO

Stage 2 works, but it is unsafe in three specific ways that Section 6 names and fixes. The first is that register access is not atomic. The second is that register ordering is not enforced. The third is that there is no discipline around what context may access which registers. Each of these is a category of bug that can hurt a real driver; each is fixable with discipline the chapter has already taught in Part 3, applied to the new hardware-facing state.

This section walks through each problem, explains why it matters, and produces Stage 3 of the driver: a version that is correct under concurrent access, ordering-safe for platforms that need it, and clearly partitioned by context.

16.7.1 Why a Register Access Can Be Unsafe Without a Lock

Consider two user-space threads writing different bytes to `/dev/myfirst0` concurrently. In Stage 2, both call `myfirst_write`, which in turn calls `MYFIRST_REG_WRITE(sc, MYFIRST_REG_DATA_IN, last_byte)`. Without a lock, the two writes race: one finishes first, one finishes second, and `DATA_IN` ends up with whichever value was written last. That is not wrong, exactly; both bytes were really the last byte of their respective writes. But the driver has no way of telling which value in `DATA_IN` came from which writer.

More subtly, consider `myfirst_reg_update`, which does a read-modify-write sequence on a register. Two threads calling it on the same register in parallel can produce a classic lost-update. Thread A reads `CTRL = 0`. Thread B reads `CTRL = 0`. Thread A sets the `ENABLE` bit and writes `CTRL = 1`. Thread B sets the `RESET` bit and writes `CTRL = 2`. The result is `CTRL = 2`, with `ENABLE` lost. On any register where multiple contexts perform read-modify-write operations, this is a data-race bug that can cause real protocol failures.

The solution is familiar: a lock. The only question is which lock. Chapter 11's `sc->mtx` protects the data path; it is the natural choice for register access that happens inside the data path. A separate mutex, `sc->reg_mtx`, can be introduced for register access that happens outside the data path (sysctl handlers, the ticker task). The two can be the same lock, or different locks, depending on the driver's access patterns.

For Stage 3, we take the simpler path: use `sc->mtx` for every register access. This enforces the rule “no register access without the driver mutex” with a single primitive. The cost is that sysctl handlers and the ticker task must acquire the driver mutex briefly, which serialises them with the data path. For a driver this small, the cost is negligible.

16.7.2 Adding the Lock

Modify `myfirst_reg_read` and `myfirst_reg_write` to assert that the driver lock is held, and modify their callers to acquire it. An assertion is cheaper than acquiring the lock inside the accessor, and it makes the locking rule visible at every call site.

```
static __inline uint32_t
myfirst_reg_read(struct myfirst_softc *sc, bus_size_t offset)
{
    MYFIRST_ASSERT(sc); /* Chapter 11: mtx_assert(&sc->mtx, MA_OWNED). */
    KASSERT(offset + 4 <= sc->regs_size, (...));
```



```

        return (bus_space_read_4(sc->regs_tag, sc->regs_handle, offset));
    }

static __inline void
myfirst_reg_write(struct myfirst_softc *sc, bus_size_t offset, uint32_t value)
{
    MYFIRST_ASSERT(sc);
    KASSERT(offset + 4 <= sc->regs_size, (...));
    bus_space_write_4(sc->regs_tag, sc->regs_handle, offset, value);
}

```

The MYFIRST_ASSERT macro from Chapter 11 asserts that `sc->mtx` is held in MA_OWNED mode. A debug kernel catches any caller that forgot to acquire the lock; a production kernel elides the check.

Now every call site must acquire the lock. The sysctl handler becomes:

```

static int
myfirst_sysctl_reg(SYSCTL_HANDLER_ARGS)
{
    struct myfirst_softc *sc = arg1;
    bus_size_t offset = arg2;
    uint32_t value;

    if (!myfirst_is_attached(sc))
        return (ENODEV);

    MYFIRST_LOCK(sc);
    if (sc->regs_buf == NULL) {
        MYFIRST_UNLOCK(sc);
        return (ENODEV);
    }
    value = MYFIRST_REG_READ(sc, offset);
    MYFIRST_UNLOCK(sc);

    return (sysctl_handle_int(oidp, &value, 0, req));
}

```

The lock is acquired before the register read, held briefly, released before the sysctl framework's `sysctl_handle_int`. The framework may sleep (it copies the value to user space), so the lock cannot be held across that call.

Similarly, the writeable sysctl handler:

```

static int
myfirst_sysctl_reg_write(SYSCTL_HANDLER_ARGS)
{
    struct myfirst_softc *sc = arg1;
    bus_size_t offset = arg2;
    uint32_t oldval, newval;
    int error;

    if (!myfirst_is_attached(sc))
        return (ENODEV);

    MYFIRST_LOCK(sc);
    if (sc->regs_buf == NULL) {
        MYFIRST_UNLOCK(sc);
        return (ENODEV);
    }
    oldval = MYFIRST_REG_READ(sc, offset);
    MYFIRST_UNLOCK(sc);

    newval = oldval;
    error = sysctl_handle_int(oidp, &newval, 0, req);
    if (error != 0 || req->newptr == NULL)
        return (error);

    MYFIRST_LOCK(sc);
    if (sc->regs_buf == NULL) {
        MYFIRST_UNLOCK(sc);
        return (ENODEV);
    }
    MYFIRST_REG_WRITE(sc, offset, newval);
    if (offset == MYFIRST_REG_CTRL)
        myfirst_ctrl_update(sc, oldval, newval);
    MYFIRST_UNLOCK(sc);

    return (0);
}

```

The handler acquires the lock twice: once to read the current value, once to apply the new value. Between them the lock is released and `sysctl_handle_int` runs. The pattern is slightly awkward but is standard in FreeBSD: you acquire a lock for a brief operation, release it for a sleepable call, reacquire for the next brief operation, and tolerate the fact that state may have changed in between.

The `myfirst_ctrl_update` call now happens under the lock, so its `printf` still works but any future state changes it makes can rely on lock ownership.

The ticker task callback likewise acquires the lock:

```
static void
myfirst_reg_ticker_cb(void *arg, int pending)
{
    struct myfirst_softc *sc = arg;

    if (!myfirst_is_attached(sc))
        return;

    MYFIRST_LOCK(sc);
    if (sc->regs_buf != NULL) {
        uint32_t v = MYFIRST_REG_READ(sc, MYFIRST_REG_SCRATCH_A);
        MYFIRST_REG_WRITE(sc, MYFIRST_REG_SCRATCH_A, v + 1);
    }
    MYFIRST_UNLOCK(sc);
}
```

And the `myfirst_write` and `myfirst_read` paths, which already held `sc->mtx` around ring-buffer access, need to extend the hold across the register updates, or to release and reacquire briefly. The simplest change is to keep the register updates inside the existing locked region:

```
/* In myfirst_write, while still holding sc->mtx after the ring-buffer update: */
if (wrote_any) {
    MYFIRST_REG_WRITE(sc, MYFIRST_REG_DATA_IN, (uint32_t)last_byte);
    myfirst_reg_update(sc, MYFIRST_REG_STATUS, 0, MYFIRST_STATUS_DATA_AV);
}
```

Because `myfirst_reg_update` now asserts the mutex is held, and it is, the call succeeds. The lock-hold time grows slightly, but only by a handful of `bus_space_write_4` calls, which compile to single `mov` instructions; the cost is negligible.

16.7.3 Why Barriers Matter Even On x86

With locking in place, the driver is correct under concurrency on x86. On weakly-ordered platforms (arm64, RISC-V, some older MIPS), the story is not quite finished. A sequence like:

```
MYFIRST_REG_WRITE(sc, MYFIRST_REG_DATA_IN, value);
MYFIRST_REG_WRITE(sc, MYFIRST_REG_CTRL, CTRL_GO);
```

implies that the `DATA_IN` write reaches the device before the `CTRL.GO` trigger. On x86, hardware preserves program order for stores, and the compiler's reordering is limited by the `volatile` qualifier in `bus_space_write_4`. On arm64, the CPU can reorder the two stores, and the device might see `CTRL.GO` before `DATA_IN` is ready, which breaks the protocol.

The fix is a write barrier:

```
MYFIRST_REG_WRITE(sc, MYFIRST_REG_DATA_IN, value);
bus_space_barrier(sc->regs_tag, sc->regs_handle, 0, sc->regs_size,
    BUS_SPACE_BARRIER_WRITE);
MYFIRST_REG_WRITE(sc, MYFIRST_REG_CTRL, CTRL_GO);
```

On x86 this barrier is a compiler fence. On arm64 it is a DSB or DMB that forces the first store to complete before the second is issued.

For Chapter 16 the protocol we are simulating does not actually require this ordering, because our “device” is kernel memory whose observers all take the same lock and do not reorder within a critical section. But the habit is worth developing. When the code eventually talks to a real device, the barriers will be there, and the driver will be portable across architectures.

As a teaching vehicle, introduce a helper that makes barrier-annotated writes easy:

```
static __inline void
myfirst_reg_write_barrier(struct myfirst_softc *sc, bus_size_t offset,
    uint32_t value, int flags)
{
    MYFIRST_ASSERT(sc);
    MYFIRST_REG_WRITE(sc, offset, value);
    bus_space_barrier(sc->regs_tag, sc->regs_handle, 0, sc->regs_size,
        flags);
}
```

Flags are `BUS_SPACE_BARRIER_READ`, `BUS_SPACE_BARRIER_WRITE`, or the OR of the two. A driver that reads status right after writing a command uses the combined flag. One that only wants subsequent writes to see this write’s effect uses just `WRITE`.

The Stage 3 driver does not use `myfirst_reg_write_barrier` in many places; it is defined and used in a single demonstration path (inside the ticker, after the scratch increment, to illustrate the usage). Later chapters that deal with real protocols will use it more heavily.

16.7.4 Partitioning Register Access by Context

With locking uniform, the next question is: which contexts access which registers, and is that mix intentional?

A Stage 3 audit on the driver shows:

- Syscall context (write): accesses `DATA_IN`, `STATUS`.
- Syscall context (read): accesses `DATA_OUT`, `STATUS`.
- Sysctl context: accesses every register (read) and `CTRL`, `SCRATCH_A`, `SCRATCH_B`, etc. (write) through the writeable sysctl.
- Task context (ticker): accesses `SCRATCH_A`.

Every access is lock-protected. Every access touches a register the driver has explicitly allocated for that purpose. The access discipline is simple: syscalls read and write the data registers and the data-available bit; sysctls are

for inspection and one-off configuration; the ticker writes one specific register. A rule of “contexts do not overlap their register responsibilities” is easy to state and easy to hold.

This is the kind of thing `HARDWARE.md` exists to document. Update the document to include per-register ownership:

Per-Register Owners

```
CTRL:          sysctl writer, driver (via myfirst_ctrl_update)
STATUS:        driver (via write/read paths)
DATA_IN:        syscall write path
DATA_OUT:       syscall read path
INTR_MASK:     sysctl writer only (Stage 3); driver attach (Chapter 19)
INTR_STATUS:   sysctl writer only (Stage 3)
DEVICE_ID:     attach only (initialised once, never written thereafter)
FIRMWARE_REV:  attach only (initialised once, never written thereafter)
SCRATCH_A:     ticker task; sysctl writer
SCRATCH_B:     sysctl writer only
```

A future contributor can glance at this table and immediately see where a register write is expected. A future change that adds a new owner must update the table, which keeps the documentation honest.

16.7.5 Avoiding Busy-Wait Loops

One style of bug that new driver authors fall into is the busy-wait loop for register state. The canonical example:

```
/* BAD: busy-waits forever if the device never becomes ready. */
while ((MYFIRST_REG_READ(sc, MYFIRST_REG_STATUS) & MYFIRST_STATUS_READY) == 0)
    ;
```

The loop spins reading the register until the `READY` bit becomes set. On real hardware, the time between “not ready” and “ready” may be microseconds. On an overloaded system, it may be longer. During the spin, the CPU is consumed by the loop; no other thread on that CPU can run; the driver’s own other threads cannot even unlock the mutex the loop may be holding.

Several better patterns exist.

Bounded spin with `DELAY(9)` for short waits. If the expected wait is short (less than a few hundred microseconds, typically), use a loop with a bounded iteration count and a `DELAY` between iterations. `DELAY(usec)` busy-waits for at least `usec` microseconds, allowing the CPU to serve interrupts in the meantime.

```
for (i = 0; i < 100; i++) {
    if (MYFIRST_REG_READ(sc, MYFIRST_REG_STATUS) & MYFIRST_STATUS_READY)
        break;
```

```

        DELAY(10);
    }
    if ((MYFIRST_REG_READ(sc, MYFIRST_REG_STATUS) & MYFIRST_STATUS_READY) == 0) {
        device_printf(sc->dev, "timeout waiting for READY\n");
        return (ETIMEDOUT);
    }

```

The loop runs at most 100 times, waiting 10 microseconds between reads, for a total bound of 1 millisecond. On a successful completion the loop exits early. On timeout the driver gives up and returns an error.

Sleep-based wait with `msleep`. For longer expected waits (milliseconds to seconds), do not busy-wait at all. Sleep the thread until a wakeup arrives, or until a timeout fires. The example below is hypothetical (our simulated device never clears `READY` in Chapter 16), but it shows the shape you will reach for when real hardware starts changing status bits:

```

/* Hypothetical; assumes sc->status_wait is a dummy address the driver
 * uses as a sleep channel and a matching wakeup(&sc->status_wait) fires
 * when the ready bit is expected to change. */
while ((MYFIRST_REG_READ(sc, MYFIRST_REG_STATUS) & MYFIRST_STATUS_READY) == 0) {
    error = msleep(&sc->status_wait, &sc->mtx, PCATCH, "myfready", hz / 10);
    if (error == EWOULDBLOCK) {
        /* Timeout: return to caller with ETIMEDOUT. */
        return (ETIMEDOUT);
    }
    if (error != 0)
        return (error);
}

```

The thread sleeps on `&sc->status_wait`, with the driver mutex as the interlock, for up to 100ms. A wakeup from another context (typically an interrupt handler or a task that observed the register change) breaks the sleep. On arm64, where register polling would be expensive and imprecise, this pattern is strongly preferred. Chapter 17 makes this pattern concrete: a callout flips `READY` on a timer, and a syscall path sleeps on the matching channel until the callout wakes it up.

Event-driven with `cv_wait`. Same as above, but using a condition variable, which is more natural in Part 3's idiom:

```

MYFIRST_LOCK(sc);
while ((MYFIRST_REG_READ(sc, MYFIRST_REG_STATUS) & MYFIRST_STATUS_READY) == 0) {
    cv_timedwait_sig(&sc->status_cv, &sc->mtx, hz / 10);
}
MYFIRST_UNLOCK(sc);

```

With a matching `cv_signal` in whichever context sets the bit.

Chapter 16's simulated device does not need any of these patterns yet, because the `READY` bit is set at attach and never cleared. But the patterns are named here because Section 6 is the right place to introduce them, and Chapter 17 will use them when status bits start changing dynamically.

16.7.6 Interrupts and MMIO: A Forward Pointer

A brief note on a topic Chapter 19 owns. When a real driver has an interrupt handler that runs in filter or `ithread` context, register access from the handler has additional constraints. Filter handlers run in an interrupt context with very limited primitives available; they typically acknowledge the interrupt by writing a register, record the event somehow, and defer the real work to a task. The acknowledgement write is the kind of thing `bus_space_write_*` exists for, and it runs under specific locking rules that differ from the ordinary driver `mutex`.

Chapter 16's driver has no interrupt handler, so this concern does not yet apply. Chapter 19 introduces the handler and the lock-type changes it forces. For now, treat “interrupt context accessing registers” as a topic you know exists and will learn about later; the register-access mechanism (`bus_space_*`) is the same, but the locking around it changes.

16.7.7 Stage 3 Complete

With locking added, barriers introduced, context ownership documented, and busy-wait patterns discouraged, the driver is at `0.9-mmio-stage3`. The shape of the driver is still that of Stage 2, but every register access is now lock-protected, every context's access pattern is documented, and the driver is prepared to handle more sophisticated hardware protocols in later chapters.

Build, test, and stress:

```
# cd examples/part-04/ch16-accessing-hardware/stage3-synchronized
# make clean && make
# kldload ./myfirst.ko
# examples/part-04/ch16-accessing-hardware/labs/reg_stress.sh
```

The stress script spawns several concurrent writers, readers, `sysctl` readers, and ticker-toggle operations, and checks that the final register state is consistent. Under `WITNESS`, any locking violation produces an immediate warning; under `INVARIANTS`, any out-of-bounds access produces a panic. If the script completes cleanly, the driver's register discipline is sound.

16.7.8 Wrapping Up Section 6

MMIO safety rests on three disciplines: locking (every register access happens under the appropriate lock), ordering (barriers where the protocol requires them, even if the platform is x86), and context partitioning (each register has a named owner and an intended access path). The driver's `HARDWARE.md` captures the last two; the `mutex` assertions in the accessors enforce the first.

Section 7 takes the next step: making the register access observable for debugging. Logs, `sysctls`, `DTrace` probes, and the small observability layer that catches bugs before they become crashes.

16.8 Section 7: Debugging and Tracing Hardware Access

A register access is, by design, invisible. It compiles to a CPU instruction that reads or writes a handful of bytes. There is no stack frame, no call log, no return value you can `printf` without adding code. When a driver works, invisibility is a virtue. When it does not, invisibility is the problem.

Section 7 covers the tools and idioms that make register access visible enough to debug without being so noisy that the driver becomes unreadable. The goal is a small observability layer: enough instrumentation to catch bugs early, placed where a beginner can switch it on and off, and integrated with the logging the rest of the driver already does.

16.8.1 What You Want to Observe

Three things are worth seeing when a driver is misbehaving.

The value at a specific register, right now. The `sysctl` handlers from Stage 2 already give you this. A `sysctl dev.myfirst.0.reg_ctrl` returns the current value at any moment, and nothing else in the driver's behaviour changes because of the read.

The sequence of register accesses the driver made recently. When a bug involves register ordering or incorrect bit manipulation, knowing “the driver wrote 0x1 then 0x2 then 0x4 in that order” is exactly what you need. The raw sequence cannot be reconstructed from register contents alone.

The stack and context for a specific register access. When a write happens from an unexpected code path, you want to know which function did it, which thread was running, and what was on the stack. DTrace is good at this.

The rest of this section walks through each kind of observation and shows what to add to the driver to support it.

16.8.2 A Simple Access Log

The simplest observability tool is a log of the last *N* register accesses, kept in a ring buffer inside the softc. Every register read and write records an entry; the `sysctl` exposes the ring.

Define the log entry:

```
#define MYFIRST_ACCESS_LOG_SIZE 64

struct myfirst_access_log_entry {
    uint64_t    timestamp_ns;
    uint32_t    value;
    bus_size_t  offset;
    uint8_t     is_write;
    uint8_t     width;
    uint8_t     context_tag;
    uint8_t     _pad;
};
```


Each entry is 24 bytes, holding the time (nanoseconds since boot), the value, the offset, whether it was a read or a write, the access width, and a tag identifying the caller's context (syscall, task, sysctl). Padding rounds to 24; a log of 64 entries is 1.5 KiB, which is trivially small.

Add the ring to the softc:

```
struct myfirst_access_log_entry access_log[MYFIRST_ACCESS_LOG_SIZE];
unsigned int access_log_head; /* index of next write */
bool access_log_enabled;
```

Record an entry in the accessors. Stage 3's myfirst_reg_write becomes:

```
static __inline void
myfirst_reg_write(struct myfirst_softc *sc, bus_size_t offset, uint32_t value)
{
    MYFIRST_ASSERT(sc);
    KASSERT(offset + 4 <= sc->regs_size, (...));
    bus_space_write_4(sc->regs_tag, sc->regs_handle, offset, value);

    if (sc->access_log_enabled) {
        unsigned int idx = sc->access_log_head++ % MYFIRST_ACCESS_LOG_SIZE;
        sc->access_log[idx].timestamp_ns = nanouptime_ns();
        sc->access_log[idx].value = value;
        sc->access_log[idx].offset = offset;
        sc->access_log[idx].is_write = 1;
        sc->access_log[idx].width = 4;
        sc->access_log[idx].context_tag = myfirst_current_context_tag();
    }
}
```

(nanouptime_ns() is a small helper that wraps nanouptime() and returns a uint64_t. myfirst_current_context_tag() returns a small code like 'S' for syscall, 'T' for task, 'C' for sysctl; its implementation is a few-line switch on the current thread's identity.)

The read accessor records the read (value is the value read). Access recording happens under the driver mutex (Stage 3 requires the mutex for every register access), so the ring itself needs no additional locking.

Enable with a sysctl:

```
SYSCTL_ADD_BOOL(&sc->sysctl_ctx,
    SYSCTL_CHILDREN(sc->sysctl_tree), OID_AUTO, "access_log_enabled",
    CTLFLAG_RW, &sc->access_log_enabled, 0,
    "Record every register access in a ring buffer");
```

Expose the log through a special sysctl handler that dumps the ring:

```

static int
myfirst_sysctl_access_log(SYSCTL_HANDLER_ARGS)
{
    struct myfirst_softc *sc = arg1;
    struct sbuf *sb;
    int error;
    unsigned int i, start;

    sb = sbuf_new_for_sysctl(NULL, NULL, 256 * MYFIRST_ACCESS_LOG_SIZE, req);
    if (sb == NULL)
        return (ENOMEM);

    MYFIRST_LOCK(sc);
    start = sc->access_log_head;
    for (i = 0; i < MYFIRST_ACCESS_LOG_SIZE; i++) {
        unsigned int idx = (start + i) % MYFIRST_ACCESS_LOG_SIZE;
        struct myfirst_access_log_entry *e = &sc->access_log[idx];
        if (e->timestamp_ns == 0)
            continue;
        sbuf_printf(sb, "%16ju ns  %s%1d  off=%#04x  val=%#010x  ctx=%c\n",
            (uintmax_t)e->timestamp_ns,
            e->is_write ? "W" : "R", e->width,
            (unsigned)e->offset, e->value, e->context_tag);
    }
    MYFIRST_UNLOCK(sc);

    error = sbuf_finish(sb);
    sbuf_delete(sb);
    return (error);
}

```

The handler walks the ring from the oldest to the newest entry, skipping empty slots, and formats each as a line. The output looks like:

```

123456789 ns  W4  off=0x00  val=0x00000001  ctx=C
123567890 ns  R4  off=0x00  val=0x00000001  ctx=C
124001234 ns  W4  off=0x08  val=0x00000041  ctx=S
124001567 ns  W4  off=0x04  val=0x00000009  ctx=S

```

Four entries: a writeable-sysctl write to CTRL, then its immediate readback (both ctx=C), then a syscall write that set DATA_IN to 0x41 ('A') and updated STATUS to 0x9 (READY | DATA_AV). The context tag makes the source obvious.

For debugging, this log is priceless. You see exactly what the driver did, in order, with timestamps.

16.8.3 Kernel Printf: A Controlled Flood

Sometimes the log is not enough and you want a printed message per register access, perhaps during a specific failing test. The driver should support a knob for that.

Add a debug-level sysctl (if one does not already exist from Chapter 12's `debug_level`) and use it in the accessors:

```
#define MYFIRST_DBG_REGS 0x10u

static __inline void
myfirst_reg_write(struct myfirst_softc *sc, bus_size_t offset, uint32_t value)
{
    MYFIRST_ASSERT(sc);
    KASSERT(offset + 4 <= sc->regs_size, (...));

    if ((sc->debug_level & MYFIRST_DBG_REGS) != 0)
        device_printf(sc->dev, "W%d reg= %#04x val= %#010x\n",
            4, (unsigned)offset, value);

    bus_space_write_4(sc->regs_tag, sc->regs_handle, offset, value);

    /* ... access log update ... */
}
```

When `debug_level` has the `MYFIRST_DBG_REGS` bit set, every register write is printed to the console. Setting it during a test and clearing it immediately afterwards gives a focused log without flooding the system for the duration of the driver's life.

The debug-level bitfield is a common FreeBSD pattern. Many real drivers use a sysctl of debug or verbose with bits for different subsystems: `DBG_PROBE`, `DBG_ATTACH`, `DBG_INTR`, `DBG_REGS`, and so on. The user enables only the subset they need.

16.8.4 DTrace Probes

DTrace is the right tool when you want to observe register access patterns without modifying the driver. FreeBSD's `fbt` (function boundary tracing) provider automatically instruments every non-inlined function in the kernel. If `myfirst_reg_read` and `myfirst_reg_write` are compiled as out-of-line functions (not inlined), DTrace can hook them.

By default, `static __inline` functions are candidates for inlining, and inlined functions do not have `fbt` probes. To make the accessors DTrace-visible in debug builds, split the declarations:

```
#ifdef MYFIRST_DEBUG_REG_TRACE
static uint32_t myfirst_reg_read(struct myfirst_softc *sc, bus_size_t offset);
static void myfirst_reg_write(struct myfirst_softc *sc, bus_size_t offset,
```

```

        uint32_t value);
#else
static __inline uint32_t myfirst_reg_read(struct myfirst_softc *sc,
        bus_size_t offset);
static __inline void      myfirst_reg_write(struct myfirst_softc *sc,
        bus_size_t offset, uint32_t value);
#endif

```

With MYFIRST_DEBUG_REG_TRACE set at compile time, the accessors are regular functions with function-boundary probes. DTrace can then show every call:

```
# dtrace -n 'fbt::myfirst_reg_write:entry { printf("off=%#x val=%#x", arg1, arg2); }'
```

The output lists every register write with its offset and value, live, across the whole system. DTrace can aggregate, count, filter by stack, and join with process information in ways a hand-rolled log cannot match.

For a release build, leave MYFIRST_DEBUG_REG_TRACE unset; the accessors inline and pay no runtime cost. For a debugging build, set the macro and get full visibility.

16.8.5 Specialised DTrace Probes: sdt(9)

A more targeted alternative to fbt probes is to register Statically Defined Tracepoints (SDT) at specific points in the driver. FreeBSD's sdt(9) API lets you declare probes that DTrace can hook by name, without the overhead of a full function-boundary trace.

A probe for every register write:

```

#include <sys/sdt.h>

SDT_PROVIDER_DEFINE(myfirst);
SDT_PROBE_DEFINE3(myfirst, , , reg_write,
    "struct myfirst_softc *", "bus_size_t", "uint32_t");
SDT_PROBE_DEFINE2(myfirst, , , reg_read,
    "struct myfirst_softc *", "bus_size_t");

```

And in the accessor:

```
SDT_PROBE3(myfirst, , , reg_write, sc, offset, value);
```

DTrace picks up the probe by name:

```
# dtrace -n 'sdt::myfirst:::reg_write { printf("off=%#x val=%#x", arg1, arg2); }'
```

The probe is visible in DTrace regardless of inlining, because the kernel registers it statically. When DTrace is not running the probe, it is a no-op on modern x86 (a single NOP instruction in the inlined expansion).

SDT probes are appropriate for production code. They are permanent, named, documented parts of the driver interface. A driver's users might rely on specific SDT probe names for their own monitoring tools; removing them breaks those tools.

Chapter 16 introduces SDT lightly. Later chapters (especially Chapter 23 on debugging and tracing) dig in.

16.8.6 The Heartbeat Log From Stage 1 Onwards

One piece of instrumentation the driver already has is Chapter 13's heartbeat callout. With Chapter 16's register state added, the heartbeat becomes more informative if it prints a register snapshot:

```
static void
myfirst_heartbeat_cb(void *arg)
{
    struct myfirst_softc *sc = arg;

    MYFIRST_ASSERT(sc);
    if (!myfirst_is_attached(sc))
        return;

    if (sc->debug_level & MYFIRST_DBG_HEARTBEAT) {
        uint32_t ctrl, status;
        ctrl = sc->regs_buf != NULL ?
            MYFIRST_REG_READ(sc, MYFIRST_REG_CTRL) : 0;
        status = sc->regs_buf != NULL ?
            MYFIRST_REG_READ(sc, MYFIRST_REG_STATUS) : 0;
        device_printf(sc->dev,
            "heartbeat: ctrl=0x%x status=0x%x open=%d writers=%d\n",
            ctrl, status, sc->open_count,
            sema_value(&sc->writers_sema));
    }

    /* ... existing heartbeat work (stall detection, etc.) ... */

    callout_reset(&sc->heartbeat_co,
        msecs_to_ticks(sc->heartbeat_interval_ms),
        myfirst_heartbeat_cb, sc);
}
```

With the heartbeat bit set and a 1-second interval, the driver logs its register state every second. During a failing test, the heartbeat output often shows exactly when the state went off the rails.

16.8.7 Using kgdb on a Core Dump

When a driver panics, the kernel produces a core dump. kgdb can read the dump and inspect the softc. With the register block inside the softc, a single command can print the current register values:

```
(kgdb) print *((struct myfirst_softc *)0xffffffff8000a123400)
(kgdb) x/16xw ((struct myfirst_softc *)0xffffffff8000a123400)->regs_buf
```

The x/16xw command dumps 16 words in hex at the register buffer's address. The output is literally the 64 bytes of register state at the moment of panic. A developer staring at those bytes can often spot the wrong value that led to the panic.

The reason this works is the simulation: regs_buf is kernel memory, visible to kgdb. A real device's registers would not be visible in a core dump, because the core dump captures RAM only, not device state. For simulated devices and DMA descriptors, a core dump is a gold mine.

16.8.8 DDB Extensions

For live debugging without a panic, ddb can be extended with driver-specific commands. The DB_COMMAND macro registers a new command that ddb recognises at the prompt:

```
#include <ddb/ddb.h>

DB_COMMAND(myfirst_regs, myfirst_ddb_regs)
{
    struct myfirst_softc *sc;

    /* ... find the softc, e.g., via devclass ... */
    if (sc == NULL || sc->regs_buf == NULL) {
        db_printf("myfirst: no device or no regs\n");
        return;
    }

    db_printf("CTRL    %#010x\n", *(uint32_t *) (sc->regs_buf + 0x00));
    db_printf("STATUS  %#010x\n", *(uint32_t *) (sc->regs_buf + 0x04));
    db_printf("DATA_IN  %#010x\n", *(uint32_t *) (sc->regs_buf + 0x08));
    /* ... and so on ... */
}
```

At the db> prompt during a break:

```
db> myfirst_regs
CTRL    0x00000001
STATUS  0x00000009
```

```
DATA_IN 0x0000006f
...
```

ddb commands are a niche tool. Chapter 16 introduces them to show that they exist. Later chapters (especially Chapter 23) use them more. For now, the access log and DTrace cover most day-to-day debugging.

16.8.9 What to Do When a Register Read Returns Garbage

A short field guide to the most common mistakes that produce a “garbage” register value, with the diagnostic to try for each.

Wrong offset. The offset in the code does not match the offset in the datasheet. Diagnostic: cross-check the offset against the datasheet, and audit the header for transcription errors.

Wrong width. The code reads 32 bits of a 16-bit register, or reads 8 bits of a 32-bit register. Diagnostic: verify the width in the datasheet, and adjust the call.

Missing virtual mapping. The resource was allocated without `RF_ACTIVE`, or the driver is reading from a saved pointer that points to freed memory. Diagnostic: confirm `bus_alloc_resource_any` was called with `RF_ACTIVE`; assert `sc->regs_buf != NULL` before reading in the simulation path.

Race with another writer. Another context wrote a different value between the read and the expected observation. Diagnostic: enable the access log, reproduce the issue, inspect the log.

Read-to-clear side effect the code did not expect. The previous read cleared the bits you now expect to see. Diagnostic: check the datasheet for read side effects; consider whether the reading code should cache the value.

Cache attribute mismatch. On platforms where it matters, the virtual mapping was set up with wrong cache attributes. Diagnostic: not usually a problem on x86 with `bus_alloc_resource_any`; on other platforms, check `pmap_mapdev_attr` and the bus provider. Rare in practice if you are using the standard bus allocation path.

Endianness mismatch. On a big-endian device accessed from a little-endian CPU without the right tag or stream accessor, the bytes come back in the wrong order. Diagnostic: compare the value byte-swapped; if it now makes sense, you need the `_stream_` accessors or a swap-aware tag.

Each diagnosis points to a different fix. Keeping them in mind saves hours.

16.8.10 What to Do When a Register Write Has No Effect

Sister field guide for writes.

The register is read-only or write-once. The datasheet defines the register as readable only, or writable only until the first successful write. Diagnostic: check the datasheet’s direction column.

The write was masked out. The register has bits that are writable only under specific conditions (device disabled, chip in test mode, a specific field set). Diagnostic: enable `debug_level` printing; confirm the write happened; then check the device’s state.

The write was reordered. A barrier-needed sequence was issued without barriers, and the device saw the writes in a different order than the code intended. Diagnostic: add explicit `bus_space_barrier` calls and retest.

The write was lost to a concurrent read-modify-write. Another context clobbered the new value. Diagnostic: access log; locking audit.

The write went to the wrong offset. A transcription error or an arithmetic mistake aimed the write at a different register. Diagnostic: access log; comparison with the datasheet.

The overlap with the read diagnostic is high: most issues boil down to “the code is not doing what you think it is”, and the access log is the most direct way to see what the code is doing.

16.8.11 A Lab: The Access Log Does Its Job

A small exercise that shows the access log paying off. Enable it, exercise the driver, dump the log.

```
# sysctl dev.myfirst.0.access_log_enabled=1
# sysctl dev.myfirst.0.reg_ticker_enabled=1
# echo hello > /dev/myfirst0
# dd if=/dev/myfirst0 bs=1 count=6 of=/dev/null
# sysctl dev.myfirst.0.reg_ticker_enabled=0
# sysctl dev.myfirst.0.access_log_enabled=0
# sysctl dev.myfirst.0.access_log
```

The last `sysctl` emits several dozen lines: the ticker’s `SCRATCH_A` increments, the write’s `DATA_IN` update and `STATUS ORing`, the read’s `DATA_OUT` update and `STATUS AND-ing`, and every `sysctl`-driven read of a register along the way. The log reads like a transcript of the driver’s conversation with itself.

A beginner who sees this transcript for the first time usually has a moment of recognition: “oh, *that* is what the driver is doing under the covers”. The moment is the whole point of the exercise.

16.8.12 Wrapping Up Section 7

The register access path is invisible by default. A small observability layer makes it visible: a ring buffer of recent accesses, a debug bitfield that controls per-access `printf`, `DTrace` probes through `fbt` or `sdt`, an enhanced heartbeat that logs register snapshots, and `ddb` or `kgdb` access to the `softc` for post-mortem inspection. Each tool fits a different use case, and together they cover almost every register-level bug a driver can have.

Section 8 consolidates everything Chapter 16 has added into a refactored, documented, versioned driver. The final stage.

16.9 Section 8: Refactoring and Versioning Your MMIO-Ready Driver

Stage 3 produced a correct driver. Section 8 produces a maintainable one. The changes Stage 4 makes are organisational: split the hardware-access code out of `myfirst.c` into its own file, wrap the remaining register accesses in macros that mirror the `CSR_*` idiom real drivers use, update `HARDWARE.md` to its final form, bump the version to `0.9-mmio`, and run the full regression pass.

A driver that works is valuable. A driver that works *and* reads cleanly to the next person who opens it is far more valuable. Section 8 is about that second step.

16.9.1 The File Split

Through Chapter 15, the `myfirst` driver lived in one C file plus a header. Chapter 16 adds about 200 to 300 lines of hardware-access code, which is enough that a reader opening `myfirst.c` is now greeted by a mix of “driver business logic” and “hardware register mechanics” that compete for attention.

Stage 4 separates them. Create a new file `myfirst_hw.c` alongside `myfirst.c`. Move into it:

- The accessor implementations (`myfirst_reg_read`, `myfirst_reg_write`, `myfirst_reg_update`, `myfirst_reg_write_barrier`).
- The register-driven side-effect helpers (`myfirst_ctrl_update`).
- The ticker task callback (`myfirst_reg_ticker_cb`).
- The access log rotation helpers.
- The sysctl handlers for register views (`myfirst_sysctl_reg`, `myfirst_sysctl_reg_write`, `myfirst_sysctl_access_log`).

Move into `myfirst_hw.h`:

- Register offsets and bit masks (already there).
- Fixed-value constants (already there).
- Function prototypes for the hardware-access API (`myfirst_hw_attach`, `myfirst_hw_detach`, `myfirst_hw_set_ctrl`, `myfirst_hw_add_sysctls`, etc.).
- A small struct defining the hardware state (fewer softc fields, more grouping).

The remaining `myfirst.c` keeps:

- The softc declaration (including a struct `myfirst_hw` *hw pointer to a sub-struct for hardware state).
- The driver lifecycle (`attach`, `detach`, `module init`).
- The syscall handlers (`open`, `close`, `read`, `write`, `ioctl`, `poll`, `kqfilter`).
- The callout callbacks (`heartbeat`, `watchdog`, `tick_source`).
- The non-hardware tasks (`selwake`, `bulk_writer`, `reset_delayed`, `recovery`).
- The non-hardware sysctls.

This split mirrors how real drivers with multiple subsystems organise themselves. A network driver might have `foo.c` for the main lifecycle, `foo_hw.c` for hardware access, `foo_rx.c` for the receive path, `foo_tx.c` for the transmit path. The principle is that each file holds code of one kind, and cross-file calls go through a named API.

16.9.2 The Hardware State Structure

Inside `myfirst_hw.h`, group the hardware-related fields into their own structure:

```
struct myfirst_hw {  
    uint8_t                *regs_buf;
```

```

    size_t          regs_size;
    bus_space_tag_t  regs_tag;
    bus_space_handle_t regs_handle;

    struct task      reg_ticker_task;
    int              reg_ticker_enabled;

    struct myfirst_access_log_entry access_log[MYFIRST_ACCESS_LOG_SIZE];
    unsigned int     access_log_head;
    bool             access_log_enabled;
};

```

Add a pointer to it in the softc:

```

struct myfirst_softc {
    /* ... existing fields ... */
    struct myfirst_hw *hw;
};

```

Allocate the hw struct in myfirst_hw_attach:

```

int
myfirst_hw_attach(struct myfirst_softc *sc)
{
    struct myfirst_hw *hw;

    hw = malloc(sizeof(*hw), M_MYFIRST, M_WAITOK | M_ZERO);

    hw->regs_size = MYFIRST_REG_SIZE;
    hw->regs_buf = malloc(hw->regs_size, M_MYFIRST, M_WAITOK | M_ZERO);
    #if defined(__amd64__) || defined(__i386__)
        hw->regs_tag = X86_BUS_SPACE_MEM;
    #else
        #error "Chapter 16 simulation supports x86 only"
    #endif

    hw->regs_handle = (bus_space_handle_t)(uintptr_t)hw->regs_buf;

    TASK_INIT(&hw->reg_ticker_task, 0, myfirst_hw_ticker_cb, sc);

    /* Initialise fixed registers. */
    bus_space_write_4(hw->regs_tag, hw->regs_handle, MYFIRST_REG_DEVICE_ID,
        MYFIRST_DEVICE_ID_VALUE);
    bus_space_write_4(hw->regs_tag, hw->regs_handle, MYFIRST_REG_FIRMWARE_REV,

```

```

        MYFIRST_FW_REV_VALUE);
    bus_space_write_4(hw->regs_tag, hw->regs_handle, MYFIRST_REG_STATUS,
        MYFIRST_STATUS_READY);

    sc->hw = hw;
    return (0);
}

```

Free it in `myfirst_hw_detach`:

```

void
myfirst_hw_detach(struct myfirst_softc *sc)
{
    struct myfirst_hw *hw;

    if (sc->hw == NULL)
        return;
    hw = sc->hw;
    sc->hw = NULL;

    taskqueue_drain(sc->tq, &hw->reg_ticker_task);
    if (hw->regs_buf != NULL) {
        free(hw->regs_buf, M_MYFIRST);
        hw->regs_buf = NULL;
    }
    free(hw, M_MYFIRST);
}

```

The `myfirst_attach` and `myfirst_detach` functions now call `myfirst_hw_attach(sc)` and `myfirst_hw_detach(sc)` at the appropriate points in their ordering. The hardware sub-attach fits between “softc locks initialised” and “cdev registered”; the hardware sub-detach fits between “tasks drained” and “sysctl context released”.

16.9.3 The CSR Macros

Wrap the register accesses in macros that match the real-driver idiom:

```

#define CSR_READ_4(sc, off) \
    myfirst_reg_read((sc), (off))
#define CSR_WRITE_4(sc, off, val) \
    myfirst_reg_write((sc), (off), (val))
#define CSR_UPDATE_4(sc, off, clear, set) \
    myfirst_reg_update((sc), (off), (clear), (set))

```

The driver body now reads:

```
/* In myfirst_write: */
CSR_WRITE_4(sc, MYFIRST_REG_DATA_IN, (uint32_t)last_byte);
CSR_UPDATE_4(sc, MYFIRST_REG_STATUS, 0, MYFIRST_STATUS_DATA_AV);

/* In the ticker: */
uint32_t v = CSR_READ_4(sc, MYFIRST_REG_SCRATCH_A);
CSR_WRITE_4(sc, MYFIRST_REG_SCRATCH_A, v + 1);

/* In the heartbeat: */
uint32_t status = CSR_READ_4(sc, MYFIRST_REG_STATUS);
```

The call sites read like they are talking to hardware, because that is exactly what the abstraction represents. A newcomer opening the driver and reading any of these lines immediately understands what is happening: the driver is reading or writing a register named by a constant from the hardware header.

16.9.4 Moving Sysctls

The register-view sysctls move to `myfirst_hw_add_sysctls`:

```
void
myfirst_hw_add_sysctls(struct myfirst_softc *sc)
{
    /* ... SYSCTL_ADD_PROC calls for every register ... */
    /* ... SYSCTL_ADD_BOOL for ticker_enabled ... */
    /* ... SYSCTL_ADD_PROC for access_log ... */
}
```

The function is called from `myfirst_attach` at the usual point where sysctls are registered. The main file no longer needs to care about which sysctls exist for hardware; it delegates.

16.9.5 Final `HARDWARE.md`

With the file split and API stable, `HARDWARE.md` finalises:

```
# myfirst Hardware Interface

## Version

0.9-mmio. Chapter 16 complete.

## Register Block
```

- Size: 64 bytes (MYFIRST_REG_SIZE)
- Access: 32-bit reads and writes on 32-bit-aligned offsets
- Storage: malloc(9), M_WAITOK|M_ZERO, allocated in myfirst_hw_attach, freed in myfirst_hw_detach
- bus_space_tag: X86_BUS_SPACE_MEM (x86 only, simulation shortcut)
- bus_space_handle: pointer to the malloc'd block

API

All register access goes through:

- CSR_READ_4(sc, offset): read a 32-bit register
- CSR_WRITE_4(sc, offset, value): write a 32-bit register
- CSR_UPDATE_4(sc, offset, clear, set): read-modify-write

The driver's main mutex (sc->mtx) must be held for every register access. Accessor macros assert this via MYFIRST_ASSERT.

Register Map

(table as in Section 4 ...)

Per-Register Owners

(table as in Section 6 ...)

Observability

- dev.myfirst.0.reg_*: read each register (sysctl)
- dev.myfirst.0.reg_ctrl_set: write CTRL (sysctl, Stage 1 demo aid)
- dev.myfirst.0.access_log_enabled: record access ring
- dev.myfirst.0.access_log: dump recorded accesses
- Debug bit MYFIRST_DBG_REGS in debug_level: printf per access

Architecture Portability

The simulation path uses X86_BUS_SPACE_MEM as the tag and a kernel virtual address as the handle. On non-x86 platforms, bus_space_tag_t is a pointer to a structure and this shortcut does not compile; Chapter 17 introduces a portable alternative. Real-hardware Chapter 18 drivers use rman_get_bustag and rman_get_bushandle on a resource from bus_alloc_resource_any, which is portable by design.

The document is now a single source of truth for how the driver accesses hardware. A future contributor reads it once to understand the interface and never again needs to reverse-engineer it from code.

16.9.6 The Version Bump

In `myfirst.c`:

```
#define MYFIRST_VERSION "0.9-mmio"
```

The string appears in `kldstat -v` output (through `MODULE_VERSION`) and in the `device_printf` at attach time. Bumping it is a small change with a big signalling value: anyone looking at the running driver knows exactly which chapter's features it has.

Update the top-of-file comment to note the additions:

```
/*
 * myfirst: a beginner-friendly device driver tutorial vehicle.
 *
 * Version 0.9-mmio (Chapter 16): adds a simulated MMIO register
 * block with bus_space(9) access, lock-protected register updates,
 * barrier-aware writes, an access log, and a refactored layout that
 * splits hardware-access code into myfirst_hw.c and myfirst_hw.h.
 *
 * ... (previous version notes preserved) ...
 */
```

The top-of-file comment is the shortest path for a newcomer to understand the driver's history. Keeping it current is a small discipline with a big payoff.

16.9.7 The Final Regression Pass

Chapter 15 established the regression discipline: after every version bump, run the full stress suite from every previous chapter, confirm WITNESS is silent, confirm INVARIANTS is silent, confirm `kldunload` completes cleanly.

For Stage 4 that means:

- The Chapter 11 concurrency tests (multiple writers, multiple readers) pass.
- The Chapter 12 blocking tests (reader waits for data, writer waits for room) pass.
- The Chapter 13 callout tests (heartbeat, watchdog, tick source) pass.
- The Chapter 14 task tests (selwake, bulk writer, delayed reset) pass.
- The Chapter 15 coordination tests (writers sema, stats cache, interruptible waits) pass.
- The Chapter 16 register tests (see the Hands-On Labs below) pass.
- `kldunload myfirst` returns cleanly after the full suite.

No test is skipped. A regression in any previous chapter's test is a bug, not a deferred issue. The discipline is the same as it has been throughout Part 3.

16.9.8 Running the Final Stage

```
# cd examples/part-04/ch16-accessing-hardware/stage4-final
# make clean && make
# kldstat | grep myfirst
# kldload ./myfirst.ko
# kldstat -v | grep -i myfirst
# dmesg | tail -5
# sysctl dev.myfirst.0 | head -40
```

The `kldstat -v` output should show `myfirst` at version `0.9-mmio`. The `dmesg` tail should show the device probe and attach with no errors. The `sysctl` output should list every Chapter 11 through Chapter 16 `sysctl`, including the register `sysctls`.

Run the stress suite:

```
# ../labs/full_regression.sh
```

If every test passes, Chapter 16 is complete.

16.9.9 A Small Rule for Chapter 16's Refactor

A rule of thumb that Stage 4 embodies: when a module acquires a new responsibility, give it its own file before the responsibility grows big enough to require one. Chapter 16 adds register access as a new responsibility. The responsibility is currently small: 200 to 300 lines across all the code. Splitting it into `myfirst_hw.c` now, while it is still small, is cheap. Splitting it later, when Chapter 18 adds PCI attach logic and Chapter 19 adds an interrupt handler and Chapter 20 adds DMA, would require disentangling three intertwined subsystems at once, which is expensive.

The same rule applied to Chapter 10's `cbuf`: the ring buffer got its own `cbuf.c` as soon as it had any logic beyond “declare a struct”, which paid off when concurrency and state machines entered the picture. It applies to every future subsystem this driver grows.

16.9.10 What Stage 4 Accomplished

The driver is now at `0.9-mmio`. Compared to `0.9-coordination`, it has:

- A separate hardware-access layer in `myfirst_hw.c` and `myfirst_hw.h`.
- A full register map, documented in `myfirst_hw.h` and `HARDWARE.md`.
- `bus_space(9)`-based register accessors wrapped in `CSR_*` macros.
- Lock-protected, barrier-aware register access.
- An access log for post-hoc debugging.
- Per-register context ownership documented in `HARDWARE.md`.
- A ticker task that demonstrates autonomous register behaviour.

- An end-to-end path from user-space write to register update to user-space read.

The driver's code is recognisably FreeBSD. The layout is the layout real drivers use. The vocabulary is the vocabulary real drivers share. A reader opening the driver for the first time finds a familiar structure, reads the headers to understand the registers, and can navigate the code by subsystem.

16.9.11 Wrapping Up Section 8

The refactor is small in code but large in organisation. A file split, a structure grouping, a macro layer, a documented interface, a version bump, and a regression pass. Each is a few minutes of work. Together they turn a correct driver into a maintainable one.

The Chapter 16 driver is done. The chapter closes with labs, challenges, troubleshooting, and a bridge to Chapter 17, where the simulated device acquires dynamic behaviour.

16.10 Hands-On Labs

Labs in Chapter 16 focus on two things: observing register access while exercising the driver, and breaking the register contract to see how the driver reacts. Each lab takes 15 to 45 minutes.

16.10.1 Lab 1: Observe the Register Dance

Enable the access log. Exercise the driver across its full interface. Dump the log. Read the transcript.

```
# kldload ./myfirst.ko
# sysctl dev.myfirst.0.access_log_enabled=1

# echo -n "hello" > /dev/myfirst0
# dd if=/dev/myfirst0 bs=1 count=5 of=/dev/null 2>/dev/null
# sysctl dev.myfirst.0.reg_ctrl_set=1
# sysctl dev.myfirst.0.reg_ticker_enabled=1
# sleep 2
# sysctl dev.myfirst.0.reg_ticker_enabled=0

# sysctl dev.myfirst.0.access_log
```

You should see, in order:

- Five writes to `DATA_IN` (one per byte of “hello”).
- Updates to `STATUS` setting the `DATA_AV` bit.
- Five writes to `DATA_OUT` (one per byte read).
- Updates to `STATUS` clearing the `DATA_AV` bit as the buffer drains.
- The `sysctl`-driven `CTRL` write to enable, plus the read-back.
- Two increments of `SCRATCH_A` from the ticker.

Read each line. Every value should make sense. If a value does not make sense, the driver has a bug, the test has a typo, or your understanding of the register protocol has a gap.

16.10.2 Lab 2: Trigger a Lock Violation (Debug Kernel)

This lab only works on a kernel built with WITNESS enabled. If you are not running one, skip this lab.

Temporarily remove the MYFIRST_LOCK(sc) from the sysctl read handler. Rebuild and reload the driver. Run:

```
# sysctl dev.myfirst.0.reg_ctrl
```

The console should emit a WITNESS warning about an unprotected register access (via the MYFIRST_ASSERT in myfirst_reg_read). The sysctl output may still return a plausible value, because the lack of locking does not always produce incorrect results, but the assertion makes the violation visible.

Restore the lock. Rebuild. Verify the warning is gone.

This lab demonstrates the value of MYFIRST_ASSERT as a safety net. A production driver without the assertion would carry the bug silently until something went wrong.

16.10.3 Lab 3: Simulate a Concurrent Writer Race

Two processes writing simultaneously to /dev/myfirst0 exercise the register update code path twice over. Run:

```
# for i in 1 2 3 4; do
    (for j in $(seq 1 100); do echo -n "$i"; done > /dev/myfirst0) &
done
# wait

# sysctl dev.myfirst.0.reg_data_in
# sysctl dev.myfirst.0.reg_status
```

The DATA_IN register should hold the ASCII code of whichever writer ran last ('1' = 49, '2' = 50, '3' = 51, '4' = 52). The result is non-deterministic, which is the point: register state from concurrent writers reflects the last winner.

With Stage 3 locking, the driver is correct (no lost updates, no torn reads). Without Stage 3 locking (try reverting to Stage 2 and re-running), you may observe inconsistencies or WITNESS warnings.

16.10.4 Lab 4: Watch the Heartbeat Register Log

Enable the heartbeat debug bit, increase the interval, and let it run.

```
# sysctl dev.myfirst.0.debug_level=0x8      # MYFIRST_DBG_HEARTBEAT
# sysctl dev.myfirst.0.heartbeat_interval_ms=1000
# sysctl dev.myfirst.0.reg_ticker_enabled=1
# sleep 5
# dmesg | tail -10

# sysctl dev.myfirst.0.reg_ticker_enabled=0
# sysctl dev.myfirst.0.debug_level=0
# sysctl dev.myfirst.0.heartbeat_interval_ms=0
```

The dmesg tail should contain five lines, one per heartbeat, each showing the current register values. `SCRATCH_A` should climb by one per heartbeat because the ticker is firing in parallel.

This lab demonstrates how a production driver might use a debug log to observe live behaviour without interrupting the driver's normal operation.

16.10.5 Lab 5: Add a New Register

A practical exercise. Add a new register `SCRATCH_C` at offset `0x28`. Extend the header, extend the sysctl list, extend `HARDWARE.md`. Rebuild, reload, and verify the new register is readable and writable via sysctl.

This exercises the full workflow of adding a register: header change, sysctl addition, documentation update, test. A driver that makes all four steps easy is a well-organised driver.

16.10.6 Lab 6: Inject a Bogus Access (Debug Kernel)

A deliberate break-and-observe exercise.

Modify the ticker callback to read from an out-of-bounds offset: `MYFIRST_REG_READ(sc, 0x80)`. Rebuild. Enable the ticker. On a debug kernel with `INVARIANTS`, the `KASSERT` in `myfirst_reg_read` should panic the kernel within a few seconds, with the panic string naming the offset.

Restore the callback. Rebuild. Verify the driver runs cleanly again.

This lab shows the value of bounds assertions: an out-of-bounds access fires immediately instead of silently corrupting nearby memory. Production code should never remove these assertions; they pay their cost many times over the life of the driver.

16.10.7 Lab 7: Trace With DTrace

Compile the driver with `CFLAGS+=-DMYFIRST_DEBUG_REG_TRACE` to make the accessors out-of-line. Rebuild and reload.

Run DTrace:

```
# dtrace -n 'fbt::myfirst_reg_write:entry {  
    printf("off=%#x val=%#x", arg1, arg2);  
}'
```

In another terminal:

```
# echo hi > /dev/myfirst0
```

DTrace should print two lines, one per register write (DATA_IN and the STATUS update).

Try more advanced queries:

```
# dtrace -n 'fbt::myfirst_reg_write:entry /arg1 == 0/ { @ = count(); }'
```

Counts writes to CTRL (offset 0) for the duration of the run. Leave it running while triggering various operations, then Ctrl-C to see the total.

DTrace's power comes from the combination of low overhead, flexible filtering, and rich aggregation. A beginner who grows comfortable with it early will save hours on every later debugging session.

16.10.8 Lab 8: The Watchdog-Meets-Register Scenario

The Chapter 13 watchdog callout was introduced to catch stalls in the ring buffer. Chapter 16's register integration adds a second failure mode: the watchdog could detect a register in an impossible state. Extend the watchdog callback to complain if STATUS.ERROR is set:

```
if (MYFIRST_REG_READ(sc, MYFIRST_REG_STATUS) & MYFIRST_STATUS_ERROR) {  
    device_printf(sc->dev, "watchdog: STATUS.ERROR is set\n");  
}
```

Set the error bit from a sysctl handler:

```
# sysctl dev.myfirst.0.reg_ctrl_set=? # use your writeable-register sysctl
```

(You can similarly make a writeable sysctl for STATUS to trigger the watchdog check.)

On the next watchdog tick (default 5 seconds), the message should appear. Clear the bit; the message should stop.

This lab integrates the register path with the callout-driven monitoring path, showing how the two subsystems compose.

16.11 Challenge Exercises

Challenges go further than labs. Each should take one to four hours and exercises judgment, not just keystrokes.

16.11.1 Challenge 1: Per-File-Handle Register Snapshot

Each open file descriptor gets its own snapshot of the register block, captured at open time and accessible via a custom ioctl. Modify `myfirst_open` to snapshot the registers into a per-fd structure; implement an ioctl that returns the snapshot; write a user-space program that opens `/dev/myfirst0`, fetches the snapshot, and prints it.

Think about: how much memory does the snapshot cost per open? When should the snapshot be refreshed? Should a second ioctl be added to refresh?

16.11.2 Challenge 2: Register Diff Logging

Extend the access log to record only *changes* to registers (where the new value differs from the previous value at the same offset). Writes that do not change the value are not logged. This compresses the log significantly and focuses it on meaningful state transitions.

Think about: how do you track the “previous value”? Is it per-offset, or do you store it alongside each log entry?

16.11.3 Challenge 3: Loopback Mode

Add a `CTRL.LOOPBACK` bit (already defined in `myfirst_hw.h`). When the bit is set, writes to `DATA_IN` are also copied to `DATA_OUT`, making the driver “loop back” user-space writes without needing a read. Implement the logic, add a lab test, and confirm user-space reads return the bytes just written.

Think about: where in the write path does the copy belong? Is it still correct if multiple bytes are written in one call? Do you set `DATA_AV` differently in loopback mode?

16.11.4 Challenge 4: Read-to-Clear on `INTR_STATUS`

The Chapter 16 simulation has `INTR_STATUS` as a plain register. Real hardware often uses read-to-clear semantics. Implement them: make the sysctl read of `reg_intr_status` return the current value and then clear it, so the next read returns zero. Add a way to set pending bits (a writeable sysctl that ORs into the register).

Think about: is the read-to-clear behaviour safe for the debug sysctl? How do you handle the case where the sysctl is used to observe the value?

16.11.5 Challenge 5: A Barrier Correctness Stress Test

Write a stress harness that exercises a specific pattern: write to `CTRL`, issue a barrier, read `STATUS`, verify the read reflects the write. Run it thousands of times, measure how often the verification fails. On x86 with correct barriers, failures should be zero.

Then remove the barrier and run again. On x86, failures might still be zero (strong memory model). On arm64 (if you have access), removing the barrier may produce failures.

Think about: what does this exercise tell you about the cost and value of barriers on different architectures? Should a driver always include them?

16.11.6 Challenge 6: A Register-Aware Lockstat Run

Use lockstat to profile your Stage 3 driver under load. Identify the hottest locks. Is the driver mutex (sc->mtx) saturated by register access, or by the ring buffer path, or by neither? Generate a report and interpret the numbers.

Think about: does the result change if you split out a dedicated sc->reg_mtx for register access? Do WITNESS warnings appear? Is the driver faster or slower?

16.11.7 Challenge 7: Read a Real Driver's Register Interface

Pick a real driver in /usr/src/sys/dev/ and read its register header. Candidates include /usr/src/sys/dev/ale/if_alereg.h, /usr/src/sys/dev/e1000/e1000_regs.h, and /usr/src/sys/dev/uart/uart_dev_ns8250.h. Answer:

- How many registers does the driver define?
- What width are they (8, 16, 32, 64 bits)?
- Which registers have bit-field macros? Are there any bit-field macros that correspond to fields spanning multiple bytes?
- How does the driver wrap bus_read_* and bus_write_* (if at all)?
- How are the register offsets documented (comments, external spec references)?

Writing up the answers as a one-page analysis is a great way to consolidate Chapter 16's material. You will likely also find patterns you want to apply to your own driver.

16.12 Troubleshooting Reference

A quick reference for the problems Chapter 16's code is most likely to surface.

16.12.1 Driver fails to load

- **“resolve_symbol failed”**: Missing include or a typo in a function name. Check /var/log/messages for the exact symbol; add the include; retry.
- **“undefined reference to bus_space_read_4”**: Missing #include <machine/bus.h>. This pulls in the per-architecture bus header.
- **“invalid KMOD Makefile”**: A typo in the Makefile. Compare against the stage's known-good Makefile.

16.12.2 Driver loads but dmesg shows myfirst: cannot allocate register block

malloc(9) returned NULL at attach. Usually means M_WAITOK was passed but the system was under memory pressure; rare for a 64-byte allocation. Check the kernel's malloc statistics (vmstat -m) for myfirst. Try a reboot.

16.12.3 `sysctl dev.myfirst.0.reg_ctrl` returns `ENOENT`

The `sysctl` was not registered. Confirm `myfirst_hw_add_sysctls` is called in `attach`. Confirm the `sysctl` context and tree are the same as the rest of the driver's `sysctls`. Look for a typo in `OID_AUTO` or the leaf name.

16.12.4 `sysctl dev.myfirst.0.reg_ctrl` returns a plausible value, but changes never happen

The writeable `sysctl reg_ctrl_set` might be missing `CTLFLAG_RW`. Without `_RW`, the `sysctl` is read-only. Also check the handler is not short-circuiting because of an early `ENODEV` return.

16.12.5 Kernel panic on first register write: “page fault in kernel mode”

`sc->regs_buf` is `NULL` or dangling. Confirm `myfirst_hw_attach` ran successfully and set `sc->hw->regs_buf`. Confirm nothing freed the buffer prematurely (detach running in parallel, or a `free` in an error path).

16.12.6 Kernel panic: “myfirst: register read past end of register block”

The `KASSERT` fired. A bug in the driver is passing an out-of-range offset. Use the crash stack to find the call site. Common cause: an arithmetic expression for the offset that exceeds `MYFIRST_REG_SIZE`.

16.12.7 WITNESS warning: “acquiring duplicate lock”

Usually a sign that a call chain is acquiring `sc->mtx` recursively. The Chapter 11 mutex is a sleep mutex without the `MTX_RECURSE` flag, which is correct. Trace the stack; one of the callers is re-entering.

16.12.8 WITNESS warning: “lock order reversal”

The driver is holding `sc->mtx` while acquiring another lock (or vice versa) in an order that violates the documented order. Check `LOCKING.md` against the stack trace and fix the call site.

16.12.9 Ticker does not fire

The tick source callout's interval is zero (disabled). Confirm `dev.myfirst.0.tick_source_interval_ms` is positive. Confirm `reg_ticker_enabled` is 1. Look at the access log for `SCRATCH_A` writes; if there are none, the callout is the problem. If there are, the `sysctl` may be stale (re-read it).

16.12.10 Access log returns empty

Confirm `access_log_enabled` is 1. Confirm the driver mutex is being acquired in access paths (the log update happens under the lock). If the log is genuinely empty but registers should have been accessed, check the access paths for missing accessor calls.

16.12.11 dmesg shows no output from myfirst_ctrl_update

The `debug_level` is 0, or the specific bit is not set. Set `debug_level` to include the right bits and retry.

16.12.12 kldunload myfirst returns EBUSY

Open file descriptors still exist on the device. Close them (or use `fstat -f /dev/myfirst0` to find who holds them) and retry.

16.12.13 kldunload myfirst hangs

Detach is stuck draining a primitive. Use `procstat -kk <pid-of-kldunload>` to see where. Usually the stuck drain is a task or callout that is not cancelling. Check the detach ordering against `LOCKING.md`.

16.12.14 The stress test complains about WITNESS warnings

Each warning is a real bug. Fix one, retest, continue. Do not mass-disable WITNESS and call the issue resolved; the warnings are pointing at the problem.

16.13 Wrapping Up

Chapter 16 opened Part 4 by giving the `myfirst` driver its first hardware story. The driver now has a register block, even if that block is simulated. It uses `bus_space(9)` the way a real driver does. It protects register access with the Chapter 11 mutex, inserts barriers where ordering matters, and documents every register and every access path. It has an access log for post-hoc debugging, DTrace probes for live observation, and a `ddb` command for live inspection. It is organised across two files: the main driver lifecycle and the hardware-access layer.

What Chapter 16 deliberately did not do: real PCI (Chapter 18), real interrupts (Chapter 19), real DMA (Chapters 20 and 21), full hardware simulation with dynamic behaviour (Chapter 17). Each of those topics deserves its own chapter; each builds on Chapter 16's vocabulary.

The version is 0.9-mmio. The file layout is `myfirst.c` plus `myfirst_hw.c` plus `myfirst_hw.h` plus `myfirst_sync.h` plus `cbuf.c` plus `cbuf.h`. The documentation is `LOCKING.md` plus the new `HARDWARE.md`. The test suite has grown by the Chapter 16 labs. Every earlier Part 3 test still passes.

16.13.1 A Reflection Before Chapter 17

A pause before the next chapter. Chapter 16 was a careful introduction to a set of ideas that will recur for the rest of Part 4 and beyond. The register read and write, the barrier, the bus tag and handle, the locking discipline around MMIO: these are the pieces every later hardware-facing chapter uses without re-teaching them. You have met them all once, in a setting where you could observe, experiment, and make mistakes safely.

The same pattern that defined Part 3 defines Part 4: introduce a primitive, apply it to the driver in a small refactor, document it, test it, move on. The difference is that Part 4's primitives face outward. The driver is no longer a

self-contained world; it is a participant in a conversation with hardware. That conversation has rules the driver must respect, and the rules have consequences when they are broken.

Chapter 17 makes the hardware side of the conversation more interesting. The simulated device will grow a callout that changes STATUS bits over time. It will signal “data available” after a write by flipping a bit on a delay. It will fail occasionally to teach error-handling paths. The register vocabulary stays the same; the device behaviour becomes richer.

16.13.2 What to Do If You Are Stuck

If the Chapter 16 material feels overwhelming on first pass, a few suggestions.

First, re-read Section 3. The `bus_space` vocabulary is the foundation; if it is shaky, everything else is shaky.

Second, type Stage 1 by hand, end to end, and run it. Muscle memory produces understanding in a way that reading does not.

Third, open `/usr/src/sys/dev/ale/if_alevar.h` and find the `CSR_*` macros. The real driver’s idiom is the same as your Stage 4’s. Seeing the pattern in a production driver makes the abstraction feel less arbitrary.

Fourth, skip the challenges on first pass. The labs are calibrated for Chapter 16; the challenges assume the chapter’s material is already solid. Come back to them after Chapter 17 if they feel out of reach now.

Chapter 16’s goal was clarity of vocabulary. If you have that, the rest of Part 4 will feel navigable.

16.14 Bridge to Chapter 17

Chapter 17 is titled *Simulating Hardware*. Its scope is the deeper simulation that Chapter 16 deliberately stepped around: a register block whose contents change over time, whose protocol has side effects, and whose failures can be injected deliberately for testing. The driver at `0.9-mmio` has a register block that behaves statically; Chapter 17 makes it breathe.

Chapter 16 prepared the ground in four specific ways.

First, **you have a register map**. The offsets, bit masks, and register semantics are documented in `myfirst_hw.h` and `HARDWARE.md`. Chapter 17 extends the map with a few new registers and enriches the protocol of the existing ones; the structure is established.

Second, **you have lock-protected, barrier-aware accessors**. Chapter 17 introduces a callout that updates registers periodically from its own context. Without Chapter 16’s locking discipline, the callout would race with the syscall path. With it, the callout slots into the existing mutex without additional work.

Third, **you have an access log**. Chapter 17 uses more elaborate register updates (read-to-clear `INTR_STATUS`, write-triggered delayed `DATA_AV`, simulated errors). The access log is how you will see those updates in action, and Chapter 17 leans on it heavily.

Fourth, **you have a split file layout**. Chapter 17’s simulation logic lands in `myfirst_hw.c`, alongside the Chapter 16 accessors. The main driver file stays focused on the driver lifecycle. The split keeps the simulation code contained.

Specific topics Chapter 17 will cover:

- A callout that updates STATUS bits on a schedule, simulating autonomous device activity.
- A write-to-trigger-delayed-event pattern: writing CTRL.GO schedules a callout that, after a delay, flips STATUS.DATA_AV.
- Read-to-clear semantics on INTR.STATUS, with the driver's sysctl being careful not to inadvertently clear bits.
- Simulated error injection: a sysctl that causes the next operation to “fail” with a fault bit set in STATUS.
- Timeouts: the driver reacts correctly when the simulated device fails to become ready.
- A latency simulation path with DELAY(9) and callout_reset_sbt for different granularities.

You do not need to read ahead. Chapter 16 is sufficient preparation. Bring your myfirst driver at 0.9-mmio, your LOCKING.md, your HARDWARE.md, your WITNESS-enabled kernel, and your test kit. Chapter 17 starts where Chapter 16 ended.

A small closing reflection. Part 3 taught you the synchronisation vocabulary and a driver that coordinated itself. Chapter 16 added a register vocabulary and a driver that now has a hardware surface. Chapter 17 will give that surface dynamic behaviour; Chapter 18 will replace the simulation with real PCI hardware; Chapter 19 will add interrupts; Chapter 20 and Chapter 21 will add DMA. Each of those chapters is narrower than its topic suggests because Chapter 16 did the vocabulary work first.

The hardware conversation is now beginning. The vocabulary is yours. Chapter 17 opens the next round.

16.15 Reference: bus_space(9) Cheat Sheet

A one-page summary of the bus_space(9) API, for quick reference while coding.

16.15.1 Types

Type	Meaning
bus_space_tag_t	Identifies the address space (memory or I/O).
bus_space_handle_t	Identifies a specific region in the address space.
bus_size_t	Unsigned integer for offsets inside a region.

16.15.2 Reads

Function	Width	Notes
bus_space_read_1(t, h, o)	8	Returns u_int8_t
bus_space_read_2(t, h, o)	16	Returns u_int16_t
bus_space_read_4(t, h, o)	32	Returns u_int32_t
bus_space_read_8(t, h, o)	64	amd64 memory only

16.15.3 Writes

Function	Width	Notes
<code>bus_space_write_1(t, h, o, v)</code>	8	<code>v</code> is <code>u_int8_t</code>
<code>bus_space_write_2(t, h, o, v)</code>	16	<code>v</code> is <code>u_int16_t</code>
<code>bus_space_write_4(t, h, o, v)</code>	32	<code>v</code> is <code>u_int32_t</code>
<code>bus_space_write_8(t, h, o, v)</code>	64	amd64 memory only

16.15.4 Multi Accesses (same offset, different buffer positions)

Function	Purpose
<code>bus_space_read_multi_1(t, h, o, buf, count)</code>	Read <code>count</code> bytes from <code>o</code> .
<code>bus_space_read_multi_4(t, h, o, buf, count)</code>	Read <code>count</code> 32-bit values from <code>o</code> .
<code>bus_space_write_multi_4(t, h, o, buf, count)</code>	Write <code>count</code> 32-bit values to <code>o</code> .

16.15.5 Region Accesses (advancing offset and buffer)

Function	Purpose
<code>bus_space_read_region_4(t, h, o, buf, count)</code>	Read <code>count</code> 32-bit values from <code>o</code> . .
<code>bus_space_write_region_4(t, h, o, buf, count)</code>	Write <code>count</code> 32-bit values to <code>o</code> . .

16.15.6 Barrier

Function	Purpose
<code>bus_space_barrier(t, h, o, len, flags)</code>	Enforce read/write ordering over offset range.

Flags:

Flag	Meaning
<code>BUS_SPACE_BARRIER_READ</code>	Prior reads complete before later reads.
<code>BUS_SPACE_BARRIER_WRITE</code>	Prior writes complete before later writes.

16.15.7 Resource Shorthand (`/usr/src/sys/sys/bus.h`)

Function	Equivalent
<code>bus_read_4(r, o)</code>	<code>bus_space_read_4(r->r_bustag, r->r_bushandle, o)</code>
<code>bus_write_4(r, o, v)</code>	<code>bus_space_write_4(r->r_bustag, r->r_bushandle, o, v)</code>
<code>bus_barrier(r, o, l, f)</code>	<code>bus_space_barrier(r->r_bustag, r->r_bushandle, o, l, f)</code>

16.15.8 Allocation

Function	Purpose
<code>bus_alloc_resource_any(dev, type, &rid, flags)</code>	Allocate a resource.
<code>bus_release_resource(dev, type, rid, res)</code>	Release a resource.
<code>rman_get_bustag(res)</code>	Extract the tag.
<code>rman_get_bushandle(res)</code>	Extract the handle.

16.15.9 Types of Resource

Constant	Meaning
<code>SYS_RES_MEMORY</code>	Memory-mapped I/O region.
<code>SYS_RES_IOPORT</code>	I/O port range.
<code>SYS_RES_IRQ</code>	Interrupt line.

16.15.10 Flags

Constant	Meaning
<code>RF_ACTIVE</code>	Activate the resource (establish mapping).
<code>RF_SHAREABLE</code>	The resource may be shared with other drivers.

16.16 Reference: Further Reading

16.16.1 Manual Pages

- `bus_space(9)`: the full API reference.

- `bus_dma(9)`: DMA API (Chapter 20 reference).
- `bus_alloc_resource(9)`: resource allocation reference.
- `rman(9)`: the underlying resource manager.
- `pci(9)`: PCI subsystem overview (Chapter 18 preview).
- `device(9)`: the device identity API.
- `memguard(9)`: kernel memory debugging.

16.16.2 Source Files

- `/usr/src/sys/sys/bus.h`: the bus shorthand macros and the resource API.
- `/usr/src/sys/x86/include/bus.h`: the x86 `bus_space` implementation.
- `/usr/src/sys/arm64/include/bus.h`: the arm64 equivalent (for comparison).
- `/usr/src/sys/dev/ale/if_alevar.h`: a clean example of `CSR_*` macros.
- `/usr/src/sys/dev/e1000/if_em.c`: a production driver's allocation flow.
- `/usr/src/sys/dev/uart/uart_dev_ns8250.c`: a register-heavy driver for a classic device.
- `/usr/src/sys/dev/led/led.c`: a pseudo-device driver with no hardware.

16.16.3 Reading Order

If you want to go deeper before Chapter 17, read in this order:

1. `/usr/src/sys/sys/bus.h`, the `bus_read_* / bus_write_*` shorthand-macro block (search for `#define bus_read_1`).
2. `/usr/src/sys/x86/include/bus.h` in full (the implementation).
3. `/usr/src/sys/dev/ale/if_alevar.h` in full (the `softc`, macros, constants).
4. `/usr/src/sys/dev/ale/if_ale.c` attach path (search for `bus_alloc_resource`).
5. `/usr/src/sys/dev/e1000/if_em.c` attach path (same search).

Each reading is 15 to 45 minutes. The cumulative effect is a strong grasp of the idioms Chapter 16 introduced.

16.17 Reference: A Glossary of Chapter 16 Terms

16.17.1 Terms Introduced in This Chapter

access log: A ring buffer of recent register accesses, kept in the `softc` for debugging.

alignment: The requirement that a register access's offset be a multiple of the access width.

barrier: A function or instruction that enforces ordering between prior and subsequent memory accesses.

BAR (Base Address Register): On PCI, a device's register that advertises the physical address of its MMIO region. Chapter 18 treats BARs directly.

bus_space_handle_t: An opaque identifier of a specific region within a bus address space.

bus_space_tag_t: An opaque identifier of a bus address space (typically memory or I/O port).

CSR macro: A driver-specific wrapper macro (e.g., `CSR_READ_4`) that abstracts register access behind a short name.

endianness: The byte order in which a multi-byte register is laid out. Little-endian puts the low byte first; big-endian puts the high byte first.

field: A sub-bit-range of a register, with its own name and meaning.

firmware revision register: A read-only register that reports the device's firmware version.

I/O port: An x86-specific address space, accessed with `in` and `out` instructions. Contrasts with MMIO.

MMIO (memory-mapped I/O): The mechanism where device registers are exposed as a range of physical addresses, reachable through ordinary load and store instructions.

offset: The distance, in bytes, from the start of a device region to a specific register.

PIO (port-mapped I/O): On x86, the alternative to MMIO, using separate I/O port instructions.

region: A contiguous range of device address space, or the API family that walks across offsets.

register: A named, offset-located, width-specific unit of communication between the driver and a device.

register map: A table describing every register in a device's interface: offset, width, direction, meaning.

resource (FreeBSD): A named allocation from the bus subsystem, encapsulating a tag, handle, and ownership of a specific range.

sbuf: The `sbuf(9)` kernel API for building variable-length strings, used by the `access-log` `sysctl` handler.

side effect (register): A change in device state that a read or write causes as part of its semantics, beyond returning or storing the value.

simulation: In Chapter 16, the use of kernel memory allocated with `malloc(9)` to stand in for a device's MMIO region.

stream accessor: A `bus_space_*_stream_*` variant that does not apply endian swaps.

virtual mapping: The MMU's translation from a virtual address to a physical address, with specific cache and access attributes.

width: The bit count of a register or an access function's operand (8, 16, 32, 64).

16.17.2 Terms Previously Introduced (Reminders)

- **softc:** The per-instance driver state structure (Chapter 6).
- **device_t:** The kernel's identity for a device instance (Chapter 6).
- **malloc(9):** The kernel allocator (Chapter 5).
- **WITNESS:** The kernel's lock-order checker (Chapter 11).
- **INVARIANTS:** The kernel's defensive assertion framework (Chapter 11).
- **callout:** A timer primitive that invokes a callback after a delay (Chapter 13).
- **taskqueue:** A deferred-work primitive (Chapter 14).
- **cv_wait / cv_timedwait_sig:** Condition-variable waits (Chapter 12, Chapter 15).

16.18 Reference: The Chapter 16 Driver Diff Summary

A compact view of what Chapter 16 added to the `myfirst` driver, stage by stage, for readers who want to see the whole arc on one page.

16.18.1 Stage 1 (Section 4)

- New file: `myfirst_hw.h` with register offsets, masks, fixed values.
- `regs_buf` and `regs_size` in `softc`; allocated and freed in `attach/detach`.
- Accessor helpers: `myfirst_reg_read`, `myfirst_reg_write`, `myfirst_reg_update`.
- Macros: `MYFIRST_REG_READ`, `MYFIRST_REG_WRITE`.
- Sysctls: `reg_ctrl`, `reg_status`, `reg_device_id`, `reg_firmware_rev` (read), `reg_ctrl_set` (write).
- Coupling: `myfirst_ctrl_update` on `CTRL` writes.
- Version tag: `0.9-mmio-stage1`.

16.18.2 Stage 2 (Section 5)

- Added `regs_tag`, `regs_handle` in `softc`.
- Accessor bodies rewritten to use `bus_space_read_4` and `bus_space_write_4`.
- Write path updates `DATA_IN` and `STATUS.DATA_AV`.
- Read path updates `DATA_OUT` and clears `STATUS.DATA_AV` when the buffer drains.
- `reg_ticker_task` added; increments `SCRATCH_A` per tick.
- New sysctls: `reg_data_in`, `reg_data_out`, `reg_intr_mask`, `reg_intr_status`, `reg_scratch_a`, `reg_scratch_b`, `reg_ticker_enabled`.
- New document: `HARDWARE.md`.
- Version tag: `0.9-mmio-stage2`.

16.18.3 Stage 3 (Section 6)

- `MYFIRST_ASSERT` added to accessors.
- All register access paths acquire `sc->mtx`.
- Access log and its sysctls added (`access_log_enabled`, `access_log`).
- `myfirst_reg_write_barrier` helper for barrier-aware writes.
- `HARDWARE.md` extended with per-register ownership.
- Version tag: `0.9-mmio-stage3`.

16.18.4 Stage 4 (Section 8)

- File split: `myfirst_hw.c`, `myfirst_hw.h`, `myfirst.c`.
- `struct myfirst_hw` grouping of hardware state.
- `myfirst_hw_attach`, `myfirst_hw_detach`, `myfirst_hw_add_sysctls` APIs.
- `CSR_READ_4`, `CSR_WRITE_4`, `CSR_UPDATE_4` macros.
- `HARDWARE.md` finalised.
- Full regression pass.
- Version tag: `0.9-mmio`.

16.18.5 Lines of Code

- Stage 1 adds about 80 lines (header, accessor helpers, sysctls).

- Stage 2 adds about 90 lines (accessor rewrite, data-path coupling, ticker task).
- Stage 3 adds about 70 lines (locking, access log, barrier helper).
- Stage 4 is a net reorganisation: lines move between files but the total is roughly unchanged.

Total additions, Chapter 16: roughly 240 to 280 lines across four small stages.

16.19 Reference: A Comparison with Linux Device Register Access

Because many readers come to FreeBSD from Linux, a short comparison of the register-access vocabulary clarifies what translates and what does not.

16.19.1 Linux: `ioremap` + `readl` / `writel`

Linux uses a different shape. A driver obtains a virtual address through `ioremap` (for MMIO) or uses the raw I/O port number directly (for PIO). Register access is performed through `readl(addr)` and `writel(value, addr)`, with variants for different widths (`readb`, `readw`, `readl`, `readq`). The `addr` is a kernel virtual pointer cast to a specific marker type.

16.19.2 FreeBSD: `bus_alloc_resource` + `bus_read_*` / `bus_write_*`

FreeBSD uses the tag-and-handle abstraction. A driver obtains a `struct resource *` through `bus_alloc_resource_any`, then uses `bus_read_4` and `bus_write_4` on it. The tag and handle are extracted by the macro from the resource; the driver does not see them directly in most code.

16.19.3 What Translates

- The mental model: registers at fixed offsets, accessed by width, with barriers for ordering.
- The idea of defining a header of register offsets and bit masks.
- The idea of wrapping access in driver-specific macros (Linux's `read_reg32`, FreeBSD's `CSR_READ_4`).
- The discipline of lock-protected access for shared state.

16.19.4 What Differs

- FreeBSD carries an explicit tag that encodes the address-space type. Linux does not; the function variants choose the address space implicitly.
- FreeBSD's resource abstraction is more explicit about ownership and lifecycle. Linux's `ioremap` is a thinner wrapper.
- FreeBSD's barrier function takes offset and length arguments that bus bridges can use for narrow barriers. Linux's `mb`, `rmb`, `wmb`, and `mmiowb` are CPU-wide.
- FreeBSD's `bus_space` is usable for simulation (as in this chapter); Linux's equivalent path is less friendly for that use.

Porting a driver from Linux to FreeBSD or vice versa involves rewriting the register-access layer but not the register map, because the map is defined by the device, not the OS. A well-organised driver that keeps its register access behind CSR-style macros can have its CSR macros replaced with minimal other code changes.

16.20 Reference: A Worked Example: The Full `myfirst_hw.h`

For reference, the complete Stage 4 header. This is what lives at `examples/part-04/ch16-accessing-hardware/stage4-final/myfirst_hw.h`.

```

/* myfirst_hw.h -- Chapter 16 Stage 4 simulated hardware interface. */
#ifndef _MYFIRST_HW_H
#define _MYFIRST_HW_H

#include <sys/types.h>
#include <sys/bus.h>
#include <machine/bus.h>

/* Register offsets. */
#define MYFIRST_REG_CTRL      0x00
#define MYFIRST_REG_STATUS    0x04
#define MYFIRST_REG_DATA_IN    0x08
#define MYFIRST_REG_DATA_OUT  0x0c
#define MYFIRST_REG_INTR_MASK 0x10
#define MYFIRST_REG_INTR_STATUS 0x14
#define MYFIRST_REG_DEVICE_ID 0x18
#define MYFIRST_REG_FIRMWARE_REV 0x1c
#define MYFIRST_REG_SCRATCH_A 0x20
#define MYFIRST_REG_SCRATCH_B 0x24

#define MYFIRST_REG_SIZE      0x40

/* CTRL bits. */
#define MYFIRST_CTRL_ENABLE    0x00000001u
#define MYFIRST_CTRL_RESET     0x00000002u
#define MYFIRST_CTRL_MODE_MASK 0x000000f0u
#define MYFIRST_CTRL_MODE_SHIFT 4
#define MYFIRST_CTRL_LOOPBACK 0x00000100u

/* STATUS bits. */
#define MYFIRST_STATUS_READY    0x00000001u
#define MYFIRST_STATUS_BUSY     0x00000002u
#define MYFIRST_STATUS_ERROR    0x00000004u
#define MYFIRST_STATUS_DATA_AV 0x00000008u

/* INTR bits. */
#define MYFIRST_INTR_DATA_AV    0x00000001u
#define MYFIRST_INTR_ERROR      0x00000002u
#define MYFIRST_INTR_COMPLETE  0x00000004u

```



```
/* Fixed values. */
#define MYFIRST_DEVICE_ID_VALUE 0x4D594649u
#define MYFIRST_FW_REV_MAJOR 1
#define MYFIRST_FW_REV_MINOR 0
#define MYFIRST_FW_REV_VALUE ((MYFIRST_FW_REV_MAJOR << 16) | MYFIRST_FW_REV_MINOR)

/* Access log. */
#define MYFIRST_ACCESS_LOG_SIZE 64

struct myfirst_access_log_entry {
    uint64_t timestamp_ns;
    uint32_t value;
    bus_size_t offset;
    uint8_t is_write;
    uint8_t width;
    uint8_t context_tag;
    uint8_t _pad;
};

/* Hardware state, grouped. */
struct myfirst_hw {
    uint8_t *regs_buf;
    size_t regs_size;
    bus_space_tag_t regs_tag;
    bus_space_handle_t regs_handle;

    struct task reg_ticker_task;
    int reg_ticker_enabled;

    struct myfirst_access_log_entry access_log[MYFIRST_ACCESS_LOG_SIZE];
    unsigned int access_log_head;
    bool access_log_enabled;
};

/* API. */
struct myfirst_softc;

int myfirst_hw_attach(struct myfirst_softc *sc);
void myfirst_hw_detach(struct myfirst_softc *sc);
void myfirst_hw_add_sysctls(struct myfirst_softc *sc);

uint32_t myfirst_reg_read(struct myfirst_softc *sc, bus_size_t offset);
void myfirst_reg_write(struct myfirst_softc *sc, bus_size_t offset,
    uint32_t value);
```

```

void    myfirst_reg_update(struct myfirst_softc *sc, bus_size_t offset,
                          uint32_t clear_mask, uint32_t set_mask);
void    myfirst_reg_write_barrier(struct myfirst_softc *sc, bus_size_t offset,
                                  uint32_t value, int flags);

#define CSR_READ_4(sc, off)      myfirst_reg_read((sc), (off))
#define CSR_WRITE_4(sc, off, val) myfirst_reg_write((sc), (off), (val))
#define CSR_UPDATE_4(sc, off, clear, set) \
    myfirst_reg_update((sc), (off), (clear), (set))

#endif /* _MYFIRST_HW_H_ */

```

This single header is what the rest of the driver includes to gain access to the hardware interface. A newcomer reads it once and understands what registers exist, how they are accessed, and which macros the driver body uses.

16.21 Reference: A Worked Example: The `myfirst_hw.c` Accessor Functions

Complementing the header, the implementations. For reference and as a template.

```

/* myfirst_hw.c -- Chapter 16 Stage 4 hardware access layer. */
#include <sys/param.h>
#include <sys/system.h>
#include <sys/kernel.h>
#include <sys/bus.h>
#include <sys/malloc.h>
#include <sys/lock.h>
#include <sys/mutex.h>
#include <sys/taskqueue.h>
#include <sys/sysctl.h>
#include <sys/sbuf.h>
#include <machine/bus.h>

#include "myfirst.h"      /* struct myfirst_softc, MYFIRST_LOCK, ... */
#include "myfirst_hw.h"

MALLOC_DECLARE(M_MYFIRST);

uint32_t
myfirst_reg_read(struct myfirst_softc *sc, bus_size_t offset)
{
    struct myfirst_hw *hw = sc->hw;
    uint32_t value;

```

```

MYFIRST_ASSERT(sc);
KASSERT(hw != NULL, ("myfirst: hw is NULL in reg_read"));
KASSERT(offset + 4 <= hw->regs_size,
    ("myfirst: register read past end: offset=%#x size=%zu",
    (unsigned)offset, hw->regs_size));

value = bus_space_read_4(hw->regs_tag, hw->regs_handle, offset);

if (hw->access_log_enabled) {
    unsigned int idx = hw->access_log_head++ % MYFIRST_ACCESS_LOG_SIZE;
    struct myfirst_access_log_entry *e = &hw->access_log[idx];
    struct timespec ts;
    nanouptime(&ts);
    e->timestamp_ns = (uint64_t)ts.tv_sec * 1000000000ULL + ts.tv_nsec;
    e->value = value;
    e->offset = offset;
    e->is_write = 0;
    e->width = 4;
    e->context_tag = 'd';
}

return (value);
}

void
myfirst_reg_write(struct myfirst_softc *sc, bus_size_t offset, uint32_t value)
{
    struct myfirst_hw *hw = sc->hw;

    MYFIRST_ASSERT(sc);
    KASSERT(hw != NULL, ("myfirst: hw is NULL in reg_write"));
    KASSERT(offset + 4 <= hw->regs_size,
        ("myfirst: register write past end: offset=%#x size=%zu",
        (unsigned)offset, hw->regs_size));

    bus_space_write_4(hw->regs_tag, hw->regs_handle, offset, value);

    if (hw->access_log_enabled) {
        unsigned int idx = hw->access_log_head++ % MYFIRST_ACCESS_LOG_SIZE;
        struct myfirst_access_log_entry *e = &hw->access_log[idx];
        struct timespec ts;
        nanouptime(&ts);
        e->timestamp_ns = (uint64_t)ts.tv_sec * 1000000000ULL + ts.tv_nsec;
    }
}

```

```

        e->value = value;
        e->offset = offset;
        e->is_write = 1;
        e->width = 4;
        e->context_tag = 'd';
    }
}

void
myfirst_reg_update(struct myfirst_softc *sc, bus_size_t offset,
                  uint32_t clear_mask, uint32_t set_mask)
{
    uint32_t v;

    MYFIRST_ASSERT(sc);
    v = myfirst_reg_read(sc, offset);
    v &= ~clear_mask;
    v |= set_mask;
    myfirst_reg_write(sc, offset, v);
}

void
myfirst_reg_write_barrier(struct myfirst_softc *sc, bus_size_t offset,
                          uint32_t value, int flags)
{
    struct myfirst_hw *hw = sc->hw;

    MYFIRST_ASSERT(sc);
    myfirst_reg_write(sc, offset, value);
    bus_space_barrier(hw->regs_tag, hw->regs_handle, 0, hw->regs_size, flags);
}

```

This is a complete, working file. The `myfirst_hw_attach`, `myfirst_hw_detach`, and `myfirst_hw_add_sysctls` functions follow in the same file; they are longer but follow the same patterns as Section 4’s worked text.

16.22 Reference: A Minimal Stand-Alone Test Module

For readers who want to practise `bus_space(9)` in isolation from the `myfirst` driver, here is a minimal stand-alone kernel module that allocates a kernel-memory “device”, exposes it through `sysctls`, and lets the reader experiment. Save as `hwsim.c`:

```

/* hwsim.c -- Chapter 16 stand-alone bus_space(9) practice module. */
#include <sys/param.h>

```

```
#include <sys/system.h>
#include <sys/kernel.h>
#include <sys/module.h>
#include <sys/malloc.h>
#include <sys/sysctl.h>
#include <sys/bus.h>
#include <machine/bus.h>

MALLOC_DEFINE(M_HWSIM, "hwsim", "hwsim test module");

#define HWSIM_SIZE 0x40

static uint8_t      *hwsim_buf;
static bus_space_tag_t hwsim_tag;
static bus_space_handle_t hwsim_handle;

static SYSCTL_NODE(_dev, OID_AUTO, hwsim,
    CTLFLAG_RW | CTLFLAG_MPSAFE, NULL, "hwsim practice module");

static int
hwsim_sysctl_reg(SYSCTL_HANDLER_ARGS)
{
    bus_size_t offset = arg2;
    uint32_t value;
    int error;

    if (hwsim_buf == NULL)
        return (ENODEV);
    value = bus_space_read_4(hwsim_tag, hwsim_handle, offset);
    error = sysctl_handle_int(oidp, &value, 0, req);
    if (error != 0 || req->newptr == NULL)
        return (error);
    bus_space_write_4(hwsim_tag, hwsim_handle, offset, value);
    return (0);
}

static int
hwsim_modevent(module_t mod, int event, void *arg)
{
    switch (event) {
    case MOD_LOAD:
        hwsim_buf = malloc(HWSIM_SIZE, M_HWSIM, M_WAITOK | M_ZERO);
#if defined(__amd64__) || defined(__i386__)
        hwsim_tag = X86_BUS_SPACE_MEM;
#endif
    }
}
```

```

else
    free(hwsim_buf, M_HWSIM);
    hwsim_buf = NULL;
    return (EOPNOTSUPP);
#endif

hwsim_handle = (bus_space_handle_t)(uintptr_t)hwsim_buf;

SYSCTL_ADD_PROC(NULL, SYSCTL_STATIC_CHILDREN(_dev_hwsim),
    OID_AUTO, "reg0",
    CTLTYPE_UINT | CTLFLAG_RW | CTLFLAG_MPSAFE,
    NULL, 0x00, hwsim_sysctl_reg, "IU",
    "Offset 0x00");
SYSCTL_ADD_PROC(NULL, SYSCTL_STATIC_CHILDREN(_dev_hwsim),
    OID_AUTO, "reg4",
    CTLTYPE_UINT | CTLFLAG_RW | CTLFLAG_MPSAFE,
    NULL, 0x04, hwsim_sysctl_reg, "IU",
    "Offset 0x04");
    return (0);
case MOD_UNLOAD:
    if (hwsim_buf != NULL) {
        free(hwsim_buf, M_HWSIM);
        hwsim_buf = NULL;
    }
    return (0);
default:
    return (EOPNOTSUPP);
}
}

static moduledata_t hwsim_mod = {
    "hwsim",
    hwsim_modevent,
    NULL
};

DECLARE_MODULE(hwsim, hwsim_mod, SI_SUB_DRIVERS, SI_ORDER_ANY);
MODULE_VERSION(hwsim, 1);

```

A two-line Makefile:

```

KMOD=  hwsim
SRCS=  hwsim.c

.include <bsd.kmod.mk>

```

Build, load, and play:

```
# make clean && make
# kldload ./hwsim.ko
# sysctl dev.hwsim.reg0
# sysctl dev.hwsim.reg0=0xdeadbeef
# sysctl dev.hwsim.reg0
# sysctl dev.hwsim.reg4=0x12345678
# sysctl dev.hwsim.reg4
# kldunload hwsim
```

The module demonstrates `bus_space(9)` in its simplest possible form: a memory buffer, a tag, a handle, two register slots, a pair of `sysctls`. A reader who types this in and runs it has the whole vocabulary of Section 3 in about 80 lines of C.

16.23 Reference: Why `volatile` Matters in `bus_space`

A note on a subtle detail the chapter touched on but did not expand.

When `bus_space_read_4` on x86 expands to `*(volatile u_int32_t*)(handle + offset)`, the `volatile` qualifier is not decorative. It is load-bearing.

Without `volatile`, the compiler assumes that reading a memory location twice in sequence, with no intervening store to that location, must return the same value. It is free to reorder, coalesce, or elide reads based on that assumption. For ordinary memory, the assumption holds: RAM does not change underneath you. For device memory, the assumption is wrong. A read might consume an event; a write might have immediate and visible effects that a subsequent read sees.

The `volatile` qualifier tells the compiler: treat this access as having observable side effects. Do not reorder it with other volatile accesses. Do not elide it. Do not cache its result. Emit a load (or store) every time, exactly as written.

On x86, this is enough. The CPU's memory model is strong enough that once the load is emitted in program order, it executes in program order. On arm64, additional barriers are needed to enforce program order across CPU-level reordering, which is why `bus_space_barrier` on arm64 emits DMB or DSB instructions and on x86 emits only a compiler fence.

The short rule: every time you write a hand-rolled accessor for device memory, use `volatile`. Every time you use `bus_space_*` directly, the `volatile` is already there. Every time you cast a pointer to device memory through a non-volatile type, you have a bug waiting to happen.

16.24 Reference: A Short Comparison of Access Patterns in Real FreeBSD Drivers

An informal survey of patterns used in real drivers. Each example cites the file and the characteristic pattern; read the files themselves to see the pattern in context.

/usr/src/sys/dev/ale/if_ale.c: Uses `CSR_READ_4(sc, reg)` and `CSR_WRITE_4(sc, reg, val)` macros defined in `if_alevar.h` over `bus_read_4` and `bus_write_4`. Softc holds `ale_res[]`, an array of resources. The pattern is clean and scales well.

/usr/src/sys/dev/e1000/if_em.c: Uses `E1000_READ_REG(hw, reg)` and `E1000_WRITE_REG(hw, reg, val)` that wrap `bus_space_read_4` and `bus_space_write_4` on the `osdep struct`'s tag and handle. More indirection than `ale`, justified by Intel's cross-OS shared-code model.

/usr/src/sys/dev/uart/uart_bus_pci.c: A glue driver that allocates resources and hands them to the generic UART subsystem. The register access happens in subsystem code (`uart_dev_ns8250.c`), not in the PCI glue.

/usr/src/sys/dev/uart/uart_dev_ns8250.c: Direct `bus_read_1` and `bus_write_1` on a struct `uart_bas *` that holds tag and handle. Legacy 8-bit register layout.

/usr/src/sys/dev/virtio/pci/virtio_pci_modern.c: Uses `bus_read_4` and `bus_write_4` through struct `resource *` fields in the softc. Chapter 18 uses `virtio` as a test target for real-PCI exercises.

/usr/src/sys/dev/random/ivy.c (Intel Ivy Bridge RDRAND): Uses CPU instructions directly (`rdrand`) rather than `bus_space`; this is an unusual case because the “device” is the CPU itself, accessible through inline assembly.

Across all of these, the pattern is “wrap `bus_*` or `bus_space_*` in driver-specific macros, keep register offsets in a header, access registers through the macros in the body”. Chapter 16's Stage 4 matches this convention.

16.25 Reference: The Road Ahead in Part 4

A preview of how Chapter 16's material feeds into later chapters, for readers who like a single-page map.

Chapter 17 (Simulating Hardware): Extends the simulation with dynamic behaviour. Timers change STATUS bits. Writing `CTRL.GO` triggers a delayed update. Errors can be injected. The register vocabulary stays the same; the simulation becomes richer.

Chapter 18 (Writing a PCI Driver): Replaces the simulation with real PCI. `bus_alloc_resource_any` enters in earnest. Vendor and device IDs, BAR mapping, `pci_enable_busmaster`, `pciconf`. The simulation path remains available behind a compile-time flag for continued testing.

Chapter 19 (Handling Interrupts): Adds `bus_setup_intr`, filter vs `ithread`, interrupt acknowledgement, the `INTR_STATUS` register's read-to-clear semantics in a real context. The Chapter 16 access log becomes invaluable for debugging interrupt sequences.

Chapter 20 and Chapter 21 (DMA): Add `bus_dma(9)`. Register accesses become the control surface for DMA operations: set up a descriptor, write a doorbell register, wait for completion. The `bus_space_barrier` story becomes essential.

Chapter 22 (Power Management): Suspend, resume, dynamic power states. Registers that save and restore device state. Most of the Part 4 vocabulary applies; power management adds a few more idioms.

Each chapter introduces a new layer; Chapter 16's layer (the register) is the foundation for all of them. A reader who leaves Chapter 16 comfortable with `bus_space(9)` will find each subsequent chapter adds a new vocabulary on top of familiar ground.

16.26 Reference: How to Read a Datasheet

Every real driver starts with a datasheet: the document the device's manufacturer publishes describing the register interface, the programming model, and the operational behaviour. Chapter 16 works with a simulated device, so there is no datasheet to consult. Later chapters point at real devices, and a driver author who is comfortable with datasheets will learn faster.

A brief primer follows. The reader can skip this on first pass and return when Chapter 18 or a later chapter points at a real device's specification.

16.26.1 The Shape of a Datasheet

A datasheet is usually a PDF of fifty to fifteen hundred pages. It covers:

- A functional overview (what the device does, at a high level).
- Pinout or physical interface description (what signals the device has, what they mean).
- Register reference (the mapping Chapter 16 has been teaching you to read).
- Programming model (the sequence of operations a driver must perform for each high-level action).
- Electrical characteristics (voltages, timings, environmental ratings).
- Package dimensions (mechanical data for the circuit board designer).

Driver authors care primarily about the register reference and the programming model. Everything else is for hardware designers.

16.26.2 Reading the Register Reference

The register reference is usually a series of tables, one per register, with the following columns:

- Offset.
- Width.
- Reset value (the value the register has after power-on or reset).
- Access type (R, W, RW, R/W1C for read with write-one-to-clear, and so on).
- Field names and bit ranges.
- Field descriptions.

A seasoned driver author reads this table first, notes any unusual access types, and translates the register map into a C header with named offsets and bit masks. The translation is mechanical; the care is in getting every bit right.

A particular note on **reset values**. The reset value tells you what the register reads immediately after the device has been powered on or reset. If the driver writes a field and later reads it back, the read should return the value written (not the reset value). But if the driver has not written the register, the read returns the reset value. Getting this wrong produces surprising bugs: the driver “sees” a register it did not initialise and misinterprets the reset value as a state change.

16.26.3 Reading the Programming Model

The programming model section describes the sequences of register operations required to drive the device. A typical entry looks like:

Transmit one packet. 1. Confirm `STATUS.TX_READY` is set. 2. Write the packet data to `TX_BUF[0..n-1]` in order. 3. Write the packet length to `TX_LEN`. 4. Write `CTRL.TX_START`. 5. Wait for `STATUS.TX_DONE` to assert (may take up to 100us). 6. Clear `STATUS.TX_DONE` by writing 1 to the same bit.

This sequence is what a driver's transmit path implements. The order of steps is fixed; reordering can leave the device in an inconsistent state. The driver's job is to translate each step into a `bus_space_write_*` or `bus_space_read_*` call, with the proper barriers and locking.

Most datasheets have several such sequences. A network device might have receive, transmit, link initialisation, error recovery, and shutdown sequences. Each is documented independently.

16.26.4 Extracting the C Header

A skilled driver author reads the register reference once and produces a C header something like:

```
/* foo_regs.h -- derived from Foo Corp. Foo-9000 datasheet, rev 3.2. */

#define F00_REG_CTRL      0x0000
#define F00_REG_STATUS    0x0004
#define F00_REG_TX_LEN    0x0010
#define F00_REG_TX_BUF    0x0100 /* base of 4 KiB TX buffer region */

#define F00_CTRL_TX_START 0x00000001u
#define F00_CTRL_RX_ENABLE 0x00000002u

#define F00_STATUS_TX_READY 0x00000001u
#define F00_STATUS_TX_DONE  0x00000002u

/* ... and so on ... */
```

The header is where the driver's knowledge of the device's register interface lives. Keep it up to date with the datasheet; reference the datasheet revision in the header so future contributors know which version the offsets match.

16.26.5 A Pattern for Each Type of Register

Different access types imply different coding patterns.

Read-only, no side effect. Read whenever. Cache if convenient. No locking needed beyond “do not read a register from a region that has been freed”.

Read-only, with side effect (read-to-clear). Read exactly as often as the protocol requires. Do not add debug reads that clear state. Do not re-read to “confirm” a value.

Write-only. Write with whatever value the protocol requires. Do not read it back; the read returns garbage.

Read/write, no side effect. Safe read-modify-write sequences under lock.

Read/write, with side effect on write. Be careful with read-modify-write: the write of an unchanged bit may still trigger the side effect. Sometimes a datasheet documents this by saying “writes of 0 to bit X have no effect”; sometimes it does not, and the driver must be conservative.

Write-one-to-clear (W1C). Common for interrupt status registers. Writing 1 to a bit clears it; writing 0 has no effect. Use `CSR_WRITE_4(sc, REG, mask_of_bits_to_clear)`, not a read-modify-write.

16.26.6 An Exercise: Pretend the Chapter 16 Device Has a Datasheet

To close this reference, practise extracting a register header from a “datasheet” for the Chapter 16 simulated device. Write a fake datasheet in prose describing each register, its reset value, its access type, and its field layout. Then produce the corresponding `myfirst_hw.h`. Compare your version with the one Chapter 16 provided.

The exercise builds the muscle you will need for every real device later in the book.

16.27 Reference: A Case Study in Missing Barriers

A short cautionary tale to make the barrier story concrete, using only the MMIO vocabulary Chapter 16 has already introduced.

Imagine a real device whose datasheet says: “To send a command, write the 32-bit command word into `CMD_DATA`, then write the 32-bit command code into `CMD_GO`. The device picks up the command word when `CMD_GO` is written.” The driver expresses this sequence naively:

```
/* Step 1: write the command payload. */
CSR_WRITE_4(sc, MYFIRST_REG_CMD_DATA, payload);

/* Step 2: write the command code to trigger execution. */
CSR_WRITE_4(sc, MYFIRST_REG_CMD_GO, opcode);
```

On x86 this works. The x86 memory model guarantees that stores retire in program order from the CPU’s point of view, and the compiler’s `volatile`-annotated write inside `bus_space_write_4` prevents it from reordering the two statements. By the time `CMD_GO` reaches the device, `CMD_DATA` has already been written.

On arm64 the same code can fail. The CPU is free to reorder the two stores at the memory subsystem level. The device can observe `CMD_GO` first, grab whatever stale value still sits in `CMD_DATA`, and execute a command the driver did not intend. The symptom is intermittent, load-dependent, and only appears on arm64 hardware. A driver tested only on x86 would ship with this bug undetected.

The fix is a one-line change:

```
CSR_WRITE_4(sc, MYFIRST_REG_CMD_DATA, payload);

/* Ensure the payload write reaches the device before the doorbell. */
bus_space_barrier(sc->hw->regs_tag, sc->hw->regs_handle, 0, sc->hw->regs_size,
    BUS_SPACE_BARRIER_WRITE);

CSR_WRITE_4(sc, MYFIRST_REG_CMD_GO, opcode);
```

On x86, `bus_space_barrier` with `BUS_SPACE_BARRIER_WRITE` emits only a compiler fence, which is free in instruction terms and preserves the program order the x86 CPU was already going to preserve. On arm64, it emits a `dmb` or `dsb` that forces the CPU to drain its store buffer before the next store is issued. The same source code does the right thing on both.

The tale makes three points.

First, x86 gives driver authors a false sense of security. Code tested only on x86 may pass every test yet be broken on arm64 in ways that manifest only under specific load patterns.

Second, portability costs are tiny if you build them in early. Adding a `bus_space_barrier` at the right spot is a one-line change. Diagnosing the bug a year later on an arm64 deployment is a week of work.

Third, the cost of barriers on x86 is negligible for typical drivers. A compiler fence is free in instruction terms; it constrains the compiler's reordering, which for a driver's cold paths matters not at all.

A related family of ordering bugs appears when the driver writes to memory the device reads through DMA (a descriptor ring, for example) and then rings a doorbell register. That pattern needs a `bus_dmamap_sync` call, not just a `bus_space_barrier`; Chapter 20 teaches the DMA path in depth. The vocabulary is different, but the intuition (writes must drain before the doorbell) is the same.

The discipline Chapter 16 encourages, even when the immediate benefit is invisible, pays off when the code runs on hardware the author never saw.

16.28 Reference: Reading `if_ale.c` Step By Step

A guided walk through the attach path of a real driver so the vocabulary of Chapter 16 lands in production code. Open `/usr/src/sys/dev/ale/if_ale.c`, jump to `ale_attach`, and follow along.

16.28.1 Step 1: The Attach Entry Point

The `ale_attach` function begins:

```
static int
ale_attach(device_t dev)
{
    struct ale_softc *sc;
    if_t ifp;
```

```

uint16_t burst;
int error, i, msic, msixc;
uint32_t rxf_len, txf_len;

error = 0;
sc = device_get_softc(dev);
sc->ale_dev = dev;

```

`device_get_softc(dev)` is the same pattern Chapter 6 introduced. Nothing new here.

16.28.2 Step 2: Early Locking Setup

The driver initialises its data-path mutex, its callout, and its first task:

```

mtx_init(&sc->ale_mtx, device_get_nameunit(dev), MTX_NETWORK_LOCK,
        MTX_DEF);
callout_init_mtx(&sc->ale_tick_ch, &sc->ale_mtx, 0);
NET_TASK_INIT(&sc->ale_int_task, 0, ale_int_task, sc);

```

Every line here came straight out of Part 3. The mutex is the Chapter 11 primitive; the lock-aware callout is the Chapter 13 primitive; the task is the Chapter 14 primitive. A reader who has done Part 3 recognises all three immediately.

16.28.3 Step 3: PCI Bus-Mastering and Resource Allocation

```

pci_enable_busmaster(dev);
sc->ale_res_spec = ale_res_spec_mem;
sc->ale_irq_spec = ale_irq_spec_legacy;
error = bus_alloc_resources(dev, sc->ale_res_spec, sc->ale_res);
if (error != 0) {
    device_printf(dev, "cannot allocate memory resources.\n");
    goto fail;
}

```

`pci_enable_busmaster` is PCI-specific; Chapter 18 covers it. The `ale_res_spec` is a struct `resource_spec` array (defined earlier in the file) that describes which resources the driver wants. `bus_alloc_resources` (plural) takes the spec and fills in the `sc->ale_res` array with the allocated resources. This is a slight convenience wrapper over calling `bus_alloc_resource_any` in a loop; either pattern is common, and the Chapter 18 discussion of PCI resource allocation covers both.

After this call, `sc->ale_res[0]` holds a struct `resource *` for the device's MMIO region, and the `CSR_READ_*` / `CSR_WRITE_*` macros (defined in `/usr/src/sys/dev/ale/if_alevar.h`, right after the `softc` structure) can be used to access registers through it.

16.28.4 Step 4: Reading the First Register

The driver reads the PHY_STATUS register to decide which chip variant it is running on:

```
if ((CSR_READ_4(sc, ALE_PHY_STATUS) & PHY_STATUS_100M) != 0) {
    /* L1E AR8121 */
    sc->ale_flags |= ALE_FLAG_JUMBO;
} else {
    /* L2E Rev. A. AR8113 */
    sc->ale_flags |= ALE_FLAG_FASTETHER;
}
```

This is the first register access in `ale_attach`. It is a single `CSR_READ_4` call that returns a 32-bit value, masked against the `PHY_STATUS_100M` bit, and used to select a code path. The `ALE_PHY_STATUS` constant is a register offset defined in `/usr/src/sys/dev/ale/if_alereg.h`. The bit mask `PHY_STATUS_100M` is defined in the same header.

Every element of that one line is Chapter 16 vocabulary. `CSR_READ_4` expands to `bus_read_4` over the first resource; `bus_read_4` expands to `bus_space_read_4` over the tag and handle inside the resource; on x86 memory space, `bus_space_read_4` compiles to a single `mov` instruction.

16.28.5 Step 5: Reading More Registers

A few lines later the driver reads three more registers to gather chip-identification data:

```
sc->ale_chip_rev = CSR_READ_4(sc, ALE_MASTER_CFG) >>
    MASTER_CHIP_REV_SHIFT;
/* ... */
txf_len = CSR_READ_4(sc, ALE_SRAM_TX_FIFO_LEN);
rx_len = CSR_READ_4(sc, ALE_SRAM_RX_FIFO_LEN);
```

Same pattern, three more registers. Notice the uninitialised-hardware check a few lines down, guarded by `sc->ale_chip_rev == 0xFFFF`: if any of the returned values looks like `0xFFFFFFFF`, the driver assumes the hardware is not correctly initialised and bails out with `ENXIO`. This kind of sanity check is a common, quiet habit in production drivers: hardware that returns all-ones on every register typically means the mapping is wrong, the device is not responding, or the device was power-gated and never brought up.

16.28.6 Step 6: IRQ Setup

Further down:

```
error = bus_alloc_resources(dev, sc->ale_irq_spec, sc->ale_irq);
```

IRQ resources get their own allocation. Chapter 19 covers what happens next: `bus_setup_intr`, the filter-ithread split, and the interrupt handler.

16.28.7 Step 7: Reading the Whole File

After the IRQ allocation, `ale_attach` continues with DMA tag creation (Chapter 20), ifnet registration (Chapter 28 in Part 6), PHY attach, and so on. Each step uses patterns that will be introduced by later chapters in this book. What Chapter 16 gave you is the vocabulary to read every `CSR_*` macro call without stopping.

The exercise that consolidates the walk: pick three `CSR_READ_4` or `CSR_WRITE_4` calls from anywhere in `/usr/src/sys/dev/ale/if_ale.c`, look up the register offset in `if_alereg.h`, decode the bit mask in the same header, and write one sentence explaining what the driver is doing at that call site. If you can do that for three arbitrary calls, you have internalised the vocabulary this chapter taught.

16.29 Reference: An Honest Accounting of Chapter 16's Simplifications

A chapter that teaches a small slice of a large topic inevitably simplifies. For honesty with the reader, a catalogue of what Chapter 16 simplified and what the full story looks like.

16.29.1 The Simulated Tag

Chapter 16's simulation uses `X86_BUS_SPACE_MEM` as the tag and a kernel virtual address as the handle. On x86 this works because the `x86_bus_space_read_*` functions reduce to a `volatile` dereference of `handle + offset` for memory space. On other architectures the trick fails because the tag is not an integer; it is a pointer to a structure, and manufacturing one by hand requires reproducing the structure the platform's `bus_space` expects.

The full story: real drivers never manufacture a tag; they receive one from the bus subsystem through `rman_get_bustag`. The Chapter 16 simulation shortcut is pedagogical, and the chapter explicitly marks it as x86-only. Chapter 17's richer simulation introduces a portable alternative, and Chapter 18's real PCI path retires the shortcut entirely.

16.29.2 The Register Protocol

The simulated device's registers have no side effects on read or write. `STATUS` is set by the driver; it does not change autonomously. `DATA_IN` is written by the driver; it does not forward the write to an imaginary downstream consumer. `INTR_STATUS` is a plain register, not read-to-clear.

The full story: real devices have protocols. Reading a status register may consume an event. Writing a command register may trigger a multi-cycle operation inside the device. The driver's job is to follow the protocol exactly; a single missed step produces a misbehaving device. Chapter 17 introduces some of this complexity by adding a callout-driven protocol: a write triggers a delayed status change.

16.29.3 Locking Granularity

Chapter 16 uses a single driver mutex (`sc->mtx`) for all register access. In practice, real drivers sometimes split locks: a fast-path lock for per-packet register writes (in a network driver) and a slower lock for configuration changes. Splitting increases concurrency at the cost of more locking discipline.

The full story: lock splitting is a performance tuning decision that belongs to later chapters on scaling and profiling. Chapter 16 uses a single lock because it is the simplest correct design, and because the driver's throughput requirements are nowhere near where lock contention matters.

16.29.4 Endianness

Chapter 16 assumes host byte order for all register values. Real devices sometimes use a different byte order than the host CPU. The `_stream_` variants of `bus_space` handle this; Chapter 16 does not use them.

The full story: FreeBSD's `bus_space` API supports per-tag byte-swap semantics. A driver whose device is big-endian on a little-endian CPU uses either a swap-aware tag or the `_stream_` variants plus explicit `htobe32/be32toh` conversions. Chapter 16's simulation is host-endian, so the issue does not arise; real drivers for big-endian devices handle it explicitly.

16.29.5 Cache Attributes

Chapter 16's `malloc(9)` allocation produces ordinary cacheable kernel memory. Real device memory is mapped with different cache attributes (uncached, write-combining, device-strongly-ordered) depending on the platform and the device's requirements.

The full story: `bus_alloc_resource_any` with `RF_ACTIVE` on a real PCI BAR produces a mapping with the right cache attributes. The simulation does not go through this path; it uses plain cacheable memory. Under Chapter 16's patterns (serialised access, volatile accesses), the cache attribute difference does not manifest. In Chapter 18's real-PCI path, the allocation flow takes care of it.

16.29.6 Error Handling

Chapter 16's simulated device never returns an error from a register access. Real hardware sometimes does: a read may time out, a write may be rejected, a bus may hang. The driver must handle these.

The full story: FreeBSD provides `bus_peek_*` and `bus_poke_*` variants (since FreeBSD 13) that return an error if the access faults. Chapter 16 does not use them because the simulation cannot fault. Chapter 19 introduces them in the context of interrupt handlers that may touch a device in an uncertain state.

16.29.7 Interrupts

Chapter 16's driver polls registers through callouts and syscall paths. Real drivers typically use interrupts to know when to read a register.

The full story: interrupts are Chapter 19's topic. Chapter 16's polling pattern is a stepping stone; after Chapter 19 the driver will have an interrupt handler that replaces much of the polling logic.

16.29.8 DMA

Chapter 16's driver does not use DMA. Every byte flowing through the driver is copied by the CPU, register by register.

The full story: real high-throughput devices use DMA for bulk data. The driver programs the device's DMA engine through registers, then the device reads or writes system RAM directly. Chapter 20 and Chapter 21 cover the DMA API.

16.29.9 Summary

Chapter 16 is an on-ramp. Every simplification it makes is deliberate, named, and picked up by a later chapter. The vocabulary Chapter 16 teaches is the vocabulary every later chapter extends; the discipline Chapter 16 builds is the discipline every later chapter relies on. The chapter is short of the full hardware story on purpose. Subsequent chapters fill in the rest.

16.30 Reference: A Quick Reference for Common MMIO Bugs

When a driver misbehaves, the bug is often in a small set of recurring categories. A quick reference to recognise each.

16.30.1 1. Off-by-One in the Register Map

Symptom: A read returns a plausible but wrong value, or a write has no effect.

Cause: An offset in the driver's header is one or two or four bytes off from the datasheet.

Diagnosis: Cross-check the header against the datasheet, register by register.

Fix: Correct the offset.

16.30.2 2. Wrong Access Width

Symptom: A read returns a value that looks like only part of the register, or a write affects only part of the register.

Cause: The driver uses `bus_read_4` on a 16-bit register or vice versa.

Diagnosis: Check the datasheet's width column against the accessor suffix.

Fix: Use the correct width.

16.30.3 3. Missing Volatile Qualifier in a Hand-Rolled Accessor

Symptom: The compiler optimises away a register access, and the driver misses a state change.

Cause: A driver that wraps `bus_space_*` in a non-volatile intermediate loses the volatile annotation.

Diagnosis: Audit any custom accessor that is not a direct `bus_space_*` call.

Fix: Keep accessors as simple wrappers around `bus_space_*`; do not introduce intermediate variables without `volatile`.

16.30.4 4. Lost Update in Read-Modify-Write

Symptom: A bit set by the driver disappears; another bit set by a second context disappears.

Cause: Two contexts do RMW on the same register without a lock; one clobbers the other.

Diagnosis: Use the access log or DTrace to observe two writes in rapid succession.

Fix: Protect RMW with the driver mutex, or use a write-one-to-clear idiom if the hardware supports it.

16.30.5 5. Missing Barrier Before a Doorbell

Symptom: The device sometimes reads stale descriptor data or wrong command buffers.

Cause: The descriptor writes are reordered past the doorbell register write (on arm64 or other weakly-ordered platforms).

Diagnosis: The symptom is often transient and load-dependent.

Fix: Insert `bus_barrier` with `BUS_SPACE_BARRIER_WRITE` between the descriptor writes and the doorbell.

16.30.6 6. Reading a Write-Only Register

Symptom: The driver reads a register and gets zero or garbage; based on that value, it takes the wrong action.

Cause: The register is marked write-only in the datasheet; reads return a fixed value unrelated to state.

Diagnosis: Check the access type in the datasheet.

Fix: Do not read write-only registers. If you need to remember the last written value, cache it in the softc.

16.30.7 7. Unexpected Side Effect on Read

Symptom: A debug read changes driver behaviour.

Cause: The register has read-to-clear semantics and the debug read consumes an event.

Diagnosis: Disable the debug read; if the issue disappears, the read was the culprit.

Fix: Cache the value in the softc on the protocol-driven read; expose the cached value through the debug interface.

16.30.8 8. Dangling Tag or Handle

Symptom: Kernel panic on the first register access, with a fault at an address that does not look like the mapped region.

Cause: The driver stored a tag and handle before allocation completed, or kept them after release.

Diagnosis: `MYFIRST_ASSERT` firing; `regs_buf == NULL` in the panic.

Fix: Set the tag and handle only after successful allocation; clear them (or null the `sc->hw` pointer) before release.

16.30.9 9. Sysctl Handler Without Lock

Symptom: WITNESS warning about an unprotected register access, or rare incorrect values observed from user space.

Cause: A sysctl handler reads or writes a register without acquiring the driver lock.

Diagnosis: The MYFIRST_ASSERT inside the accessor produces a WITNESS entry.

Fix: Wrap the register access in MYFIRST_LOCK/MYFIRST_UNLOCK.

16.30.10 10. Detach Races

Symptom: Kernel panic during kldunload with a stack that includes a register access.

Cause: A callout or task accesses registers after the register buffer has been freed.

Diagnosis: regs_buf == NULL in the panic; the caller is a task or callout that was not drained before the free.

Fix: Review detach ordering; drain all callouts and tasks before freeing regs_buf.

Each of these bugs has a short diagnostic path and a well-defined fix. Keeping the list nearby during development catches most issues on first contact.

17 Simulating Hardware

17.1 Reader Guidance & Outcomes

Chapter 16 ended with a driver that owned a register block. The `myfirst` module at version `0.9-mmio` carries a 64-byte region of kernel memory shaped to look like a device: ten named 32-bit registers, a small set of bit masks, a device ID that reads `0x4D594649`, a firmware revision that encodes `1.0`, and a `STATUS` register whose `READY` bit is asserted at attach and cleared at detach. Every access goes through `bus_space_read_4` and `bus_space_write_4`, wrapped by the familiar `CSR_READ_4`, `CSR_WRITE_4`, and `CSR_UPDATE_4` macros. The driver's mutex (`sc->mtx`) protects each register touch. An access log remembers the last sixty-four accesses, a ticker task flips `SCRATCH_A` on demand, and a small `HARDWARE.md` documents the whole surface.

That driver can already do a great deal. It can read registers. It can write registers. It can observe its own accesses, catch out-of-bounds offsets on debug kernels, and enforce the lock discipline that every later chapter will depend on. What it cannot do yet is behave like a real device. Its `STATUS` bits never change on their own. Its `DATA_IN` register does not start a multi-cycle operation. Its `INTR_STATUS` does not accumulate events. Writing `CTRL.GO` does not schedule anything. Nothing in the simulation has a heartbeat of its own; every change to the register block is a direct consequence of the last user-space write.

Real devices are not like that. A temperature sensor refreshes its value every few milliseconds without anyone asking. A serial controller raises an interrupt when a byte lands in the receive FIFO, long after the driver's last write. A network card's descriptor ring fills with packets while the driver sleeps. The interesting parts of a device driver come from reacting to change, not from producing it. Teaching a reader to write those reactive paths requires a simulated device that produces change, and Chapter 17 is where the simulation learns to do so.

17.1.1 What Chapter 17 Is About

Chapter 17's scope is narrow but deep. It takes the static register block from Chapter 16 and gives it four new properties:

- **Autonomous behaviour.** A callout fires on a cadence the driver controls and updates `STATUS` bits, `INTR_STATUS` bits, or data registers as if a real device's internal state machine were running in silicon. The reader sees values change between two adjacent `sysctl` reads, even though no user-space write occurred between them.
- **Command-triggered protocol.** Writing a designated bit in `CTRL` schedules a delayed state change. After a configurable delay, `STATUS.DATA_AV` asserts and, optionally, `INTR_STATUS.DATA_AV` latches. This is the pattern every command-response device follows: a write starts something, a later status change signals that something finished.

- Realistic timing. Delays on the order of microseconds use `DELAY(9)` where appropriate and `pause_sbt(9)` where a sleep is safe. Delays on the order of milliseconds use `callout_reset_sbt(9)` so no thread is blocked while the simulated device works. Each choice has a place; each choice has a cost; Chapter 17 teaches both.
- Fault injection. A `sysctl` lets the reader ask the simulation to fail. Failures take several shapes: the next read returns a fixed error value, the next write never asserts `DATA_AV`, a random fraction of operations set `STATUS.ERROR`, a `STATUS.BUSY` bit stays on indefinitely. Each mode exercises a different error-handling path in the driver, and the driver learns to notice, recover, or report.

The chapter builds these properties on top of the Chapter 16 driver without replacing it. The same register map, the same accessors, the same `sc->mtx`, the same access log. What grows is the simulation backend. The `myfirst_hw.c` file from Chapter 16 gains a sibling file, `myfirst_sim.c`, that holds the simulation's callouts, the fault-injection hooks, and the helpers that make the register block breathe. By the end of the chapter the driver is at version 1.0-simulated, and the hardware-surface it shows the driver is rich enough to exercise almost every lesson of Part 3.

17.1.2 Why Autonomous Behaviour Deserves a Chapter of Its Own

A natural question at this stage is whether callout-driven behaviour and fault injection really deserve a full chapter. Chapter 16 already placed a register block in front of the driver, and Chapter 18 will replace that block with real PCI hardware. Is simulation not just a stepping stone we should step off as soon as possible?

Three answers, each worth taking seriously.

First, the simulation in Chapter 16 was static on purpose. It existed to let the reader practise register access without also managing device behaviour. Chapter 17 is where device behaviour enters the picture, and the right place to introduce it is not in the middle of the real-PCI chapter, where the reader is already juggling BAR allocation, vendor and device IDs, and the newbus glue. A chapter dedicated to simulation gives the reader room to reason about how a device behaves, how delays shape protocol, and how faults propagate, without the distractions real hardware brings.

Second, simulation is not a one-time scaffold we discard after Chapter 18. It is a permanent part of driver development. Real drivers get tested against simulated devices long after they first run on real hardware, because simulation is the only way to produce deterministic failures, reproducible timing, and fault conditions on demand. Every serious FreeBSD subsystem has some form of simulation or test harness, and every working driver author learns how to build one. Chapter 17 teaches the technique on a small scale where the reader can see every moving part.

Third, the simulation in Chapter 17 is a teaching device in its own right. By writing a fake device, the reader is forced to think about what a real device does: when it asserts a bit, when it clears one, when it fails, when it latches, when it forgets. A reader who has written the simulation understands the protocol in a way that a reader who has only used the simulation does not. The chapter is as much about the discipline of thinking like hardware as it is about the code that implements it.

17.1.3 Where Chapter 16 Left the Driver

A brief recap of where you should stand. Chapter 17 extends the driver produced at the end of Chapter 16 Stage 4, tagged as version 0.9-mmio. If any of the items below feels uncertain, return to Chapter 16 before starting this chapter.

- Your driver compiles cleanly and identifies itself as 0.9-mmio in `kldstat -v`.
- The `softc` contains a pointer `sc->hw` to a struct `myfirst_hw` that holds `regs_buf`, `regs_size`, `regs_tag`, `regs_handle`, and the Chapter 16 access log.
- Every register access in the driver goes through `CSR_READ_4(sc, off)`, `CSR_WRITE_4(sc, off, val)`, or `CSR_UPDATE_4(sc, off, clear, set)`.
- The accessors assert `sc->mtx` is held via `MYFIRST_ASSERT` on debug kernels.
- `sysctl dev.myfirst.0` lists the `reg_*` sysctls for each register, the `reg_ctrl_set` writeable sysctl, the `access_log_enabled` toggle, and the `access_log` dumper.
- `HARDWARE.md` documents the register map and the `CSR_*` API.
- `LOCKING.md` documents the detach ordering, including the new `myfirst_hw_detach` step.

That driver is what Chapter 17 extends. The additions are again moderate in line count: a new `myfirst_sim.c` file, four new callouts, two new sysctl groups, a small fault-injection state structure, and a handful of protocol helpers. The mental-model change is larger than the line count suggests.

17.1.4 What You Will Learn

Once you have finished this chapter, you will be able to:

- Explain what “simulating hardware” means in kernel space, why simulation is a permanent part of driver development rather than a temporary crutch, and what makes a good simulation versus a misleading one.
- Design a register map for a simulated device, with separate sections for control, status, data, interrupts, and metadata, and justify each choice against the protocol the device is meant to implement.
- Choose register widths, offsets, bit-field layouts, and access semantics (read-only, write-only, read-to-clear, write-one-to-clear, read/write) that mirror patterns used in real FreeBSD drivers.
- Implement a simulated hardware backend in kernel memory that behaves autonomously, with a callout that updates registers on a cadence and a task that reacts to writes the driver issues.
- Expose the fake device to the driver using the same `bus_space(9)` abstraction that real drivers use, so the driver code has no knowledge of whether it is talking to silicon or to a struct in kernel memory.
- Test device behaviour from the driver side by writing commands, polling for status, reading data, and checking invariants under a variety of loads.
- Add timing and delay to the simulation using `DELAY(9)`, `pause_sbt(9)`, and `callout_reset_sbt(9)`, and explain why each tool is appropriate for the time scale where it is used.
- Simulate errors and fault conditions safely, with a sysctl-driven fault-injection path that can reproduce timeouts, data errors, busy-forever states, and random failures, and use those injected faults to exercise the driver’s error paths.
- Refactor the simulation so it lives in its own file (`myfirst_sim.c`), document its interface in a `SIMULATION.md`, version the driver as 1.0-simulated, and run a full regression pass.
- Recognise the limits of simulation and the scenarios where only real hardware (or hypervisor-backed virtual hardware) will produce trustworthy results.

The list is long; each item is narrow. The point of the chapter is the composition.

17.1.5 What This Chapter Does Not Cover

Several adjacent topics are explicitly deferred so Chapter 17 stays focused.

- **Real PCI devices.** The `pci(4)` subsystem, vendor and device ID matching, BAR allocation through `bus_alloc_resource_any`, `pci_enable_busmaster`, and the glue that ties a driver to a real bus belong to Chapter 18. Chapter 17 stays in simulation and uses the Chapter 16 shortcut for the tag and handle.
- **Interrupts.** The simulation produces state changes that mimic what an interrupt handler would see. It does not yet register an actual interrupt handler via `bus_setup_intr(9)`, and it does not split work between a filter handler and an interrupt thread. Chapter 19 covers that. Chapter 17 polls through callouts and user-space syscalls; the callouts stand in for an interrupt source.
- **DMA.** Descriptor rings, scatter-gather lists, `bus_dma(9)` tags, bounce buffers, and cache flushing around DMA are Chapters 20 and 21. Chapter 17's data register stays a single 32-bit slot.
- **Full-system simulators.** QEMU, bhyve, virtio devices, and the kinds of simulation hypervisors offer are mentioned in passing as a bridge to Chapter 18. Chapter 17 stays in-kernel; the simulation runs inside the same kernel as the driver.
- **Protocol verification and formal methods.** Real hardware verification uses formal tools to prove a driver's protocol handling is correct. Those tools exist outside the scope of this book. Chapter 17 uses INVARIANTS, WITNESS, stress tests, and deliberate fault injection to build confidence.
- **Hypervisor-backed device emulation.** Chapter 17's simulation runs in the kernel's own address space. Virtio devices in a VM, emulated NICs in QEMU, and bhyve's device emulation all run in a hypervisor; the driver sees them through real PCI. That path is Chapter 18's concern.

Staying inside those lines keeps Chapter 17 a chapter about making a fake device behave like a real one. The vocabulary is what transfers; the specific subsystems are what Chapters 18 through 22 apply the vocabulary to.

17.1.6 Estimated Time Investment

- **Reading only:** three to four hours. The simulation concepts are small but accumulate; each section introduces a new behaviour pattern and the composition is what makes the chapter rich.
- **Reading plus typing the worked examples:** eight to ten hours over two sessions. The driver evolves in five stages; each stage is a small refactor that merges into the existing `myfirst_hw.c` or into the new `myfirst_sim.c`.
- **Reading plus all labs and challenges:** thirteen to sixteen hours over three or four sessions, including stress testing under the debug kernel, working through the fault-injection scenarios, and reading a couple of real callout-driven drivers.

Sections 3, 6, and 7 are the densest. If the interplay of a callout, a mutex, and a state change feels opaque on first pass, that is expected: three primitives from Part 3 are composing in a new way. Stop, re-read Section 6's timing table, and continue when the composition has settled.

17.1.7 Prerequisites

Before starting this chapter, confirm:

- Your driver source matches Chapter 16 Stage 4 (0.9-mmio). The starting point assumes every Chapter 16 register accessor, the split between `myfirst.c` and `myfirst_hw.c`, the `CSR_*` macros, and the access log.
- Your lab machine runs FreeBSD 14.3 with `/usr/src` on disk and matching the running kernel.
- A debug kernel with `INVARIANTS`, `WITNESS`, `WITNESS_SKIPSPIN`, `DDB`, `KDB`, and `KDB_UNATTENDED` is built, installed, and booting cleanly.
- You are comfortable with `callout_init_mtx`, `callout_reset_sbt`, and `callout_drain` from Chapter 13, and with `taskqueue` from Chapter 14.
- You understand `arc4random(9)` at a reading level. If it is new, skim the manual page before Section 7.

If any item above is shaky, fix it now rather than pushing through Chapter 17 and trying to reason from a moving foundation. Simulation code produces timing-sensitive bugs, and a debug kernel plus solid Part 3 foundations catch most of them on first contact.

17.1.8 How to Get the Most Out of This Chapter

Three habits will pay off quickly.

First, keep a second terminal open on the machine where you are testing. Simulation makes registers change between two `sysctl` reads, and the only way to see that is to read the same `sysctl` twice in a row. A shell that loops `sysctl dev.myfirst.0.reg_status` every 200 milliseconds is how the chapter's examples come alive. If you cannot run two terminals side by side, a `watch -n 0.2 sysctl ... loop` in a `tmux` pane works just as well.

Second, read the access log after every experiment. The log captures every register access with a timestamp, an offset, a value, and a context tag. For a static simulation the log is boring: user-space reads, user-space writes, nothing else. For the Chapter 17 simulation the log is dense: callouts firing, delayed events triggering, faults being injected, timeouts expiring. Every interesting behaviour leaves a trail, and the trail is where the chapter's intuition comes from.

Third, break the simulation on purpose. Set the fault-injection rate to 100%. Set the callout interval to zero. Make the simulated device always busy. See how the driver reacts. The chapter's error-handling lessons are a lot clearer when you have seen the failure modes play out. The simulation is safe: you can inject as many faults as you like without risking real hardware, and the worst case is a kernel panic that a debug kernel catches cleanly.

17.1.9 Roadmap Through the Chapter

The sections in order are:

1. **Why Simulate Hardware?** The case for simulation as a permanent part of driver development, the difference between a good simulation and a misleading one, and the goals Chapter 17's simulation is aiming at.

2. **Designing the Register Map.** How to choose registers, widths, layouts, and access semantics for a simulated device, with a worked example of the additions Chapter 17 makes on top of the Chapter 16 map.
3. **Implementing the Simulated Hardware Backend.** The first callout that updates registers autonomously, a `myfirst_sim_tick` helper that carries the simulation state machine, and the first stage of the Chapter 17 driver (`1.0-sim-stage1`).
4. **Exposing the Fake Device to the Driver.** The tag, the handle, the accessor path, and how the Chapter 16 abstraction carries forward without modification. A short section; the point is that the Chapter 16 work already did most of it.
5. **Testing Device Behavior from the Driver.** Writing a command, polling for status, reading data, and handling the race between the driver's observation and the simulation's own state updates. Stage 2 of the Chapter 17 driver.
6. **Adding Timing and Delay.** Microsecond-scale `DELAY(9)`, millisecond-scale `pause_sbt(9)`, and second-scale `callout_reset_sbt(9)`; when each is safe; when each is not; what latency the driver can expect from each. Stage 3.
7. **Simulating Errors and Fault Conditions.** A fault-injection framework built on top of the simulation, with modes that cover timeouts, data errors, stuck-busy, and random failures. Teaching the driver to notice, recover, or report. Stage 4.
8. **Refactoring and Versioning Your Simulated Hardware Driver.** The final split into `myfirst_sim.c`, a new `SIMULATION.md`, the version bump to `1.0-simulated`, and the regression pass. Stage 5.

After the eight sections come hands-on labs, challenge exercises, a troubleshooting reference, a Wrapping Up that closes Chapter 17's story and opens Chapter 18's, and a bridge to Chapter 18. The reference material at the end of the chapter is meant to be kept nearby while you work through the labs.

If this is your first pass, read linearly and do the labs in order. If you are revisiting, Sections 3, 6, and 7 stand alone reasonably well and make good single-sitting reads.

17.2 Section 1: Why Simulate Hardware?

Chapter 16 already made the case for simulation as an on-ramp. A reader who does not own the exact device the book references can still practise register access against a simulated block. That argument is valid and still holds in Chapter 17, but it is not the only reason simulation earns a chapter of its own. Simulation is part of how serious driver development is done, and understanding why is what this section is about.

17.2.1 The Working Driver Author's Toolkit

Pick any real FreeBSD driver and walk backward from the commit history. Most drivers that see active development share a common pattern: the first commits build the driver against real hardware. Later commits, often many later commits, add tests, harnesses, scaffolding, and instrumentation that let the driver be exercised without the original hardware sitting on the developer's desk. Some of this instrumentation runs in user space. Some of it is a hypervisor-backed fake device that speaks the same protocol as the real one. Some of it is a set of kernel-side stubs that pretend to be the hardware well enough to run the driver's state machines.

Why do working driver authors invest in this? Because real hardware is not reproducible. A real device's behaviour depends on its firmware revision, its temperature, its link partner, the signal integrity of its physical

connection, and the phase of the moon. A driver that passes tests against one instance of a device may fail against a slightly different instance. A driver that passes tests today may fail tomorrow because the device's firmware autoupdate went through overnight. A driver whose failure is timing-dependent may pass a hundred runs and fail the hundred-and-first. Real hardware is a moving target, and a moving target is a bad foundation for regression testing.

Simulation fixes the target. A simulated device behaves the same way every time the same inputs arrive. A simulated device's fault conditions are under the author's control. A simulated device can be run in a tight loop without wearing out. A simulated device does not require lab hardware, cabling, a power supply, or a colleague whose help you need. Every property that makes simulation convenient for learning makes it valuable for production testing.

The conclusion is that Chapter 17's simulation is not disposable scaffolding. The patterns you learn here are the same patterns FreeBSD committers use to test drivers that nobody has access to any more because the silicon is twenty years old. The same techniques catch regressions before they reach a customer's machine. The same discipline, applied to the real PCI device in Chapter 18, lets the driver be exercised on a workstation with no network at all by telling a virtio-net backend to drop packets, corrupt frames, or duplicate descriptors at a controlled rate.

17.2.2 Simulation Goals: Behaviour, Timing, Side Effects

When we say “simulate a device”, we usually mean one of three things, and it is worth being explicit about which one we intend.

The first kind of simulation is **behavioural**. The simulation exposes a register interface, and writes and reads produce the same effects a real device would produce. The state machine is correct: writing `CTRL.GO` moves the device from idle to busy to done, setting `STATUS.DATA_AV` at the right point. The data path is correct: data written to `DATA_IN` shows up in `DATA_OUT` after some processing. The simulation does not have to match the real device's timing; it just has to follow the same rules.

The second kind of simulation is **timing-aware**. The simulation enforces the delays the real device would enforce. A write that the real device needs 500 microseconds to process takes 500 microseconds in the simulation, more or less, subject to the CPU's own timing resolution. A status bit that the real device asserts 2 milliseconds after a command asserts 2 milliseconds after the command in the simulation. A driver that passes timing-aware simulation is likely to work on the real hardware; a driver that only passes behavioural simulation may fail when the real device's latency produces a race the driver has not considered.

The third kind of simulation is **side-effect-aware**. The simulation mirrors the subtle parts of the protocol: reads that clear state, writes that have no effect, registers that return all-ones when the device is in a certain mode, bits that are write-one-to-clear versus bits that are read-to-clear. A driver that passes this kind of simulation is confident in the low-level protocol details, not just the high-level state flow.

The three kinds stack. A realistic simulation is all three: behaviour, timing, and side effects. Chapter 17 builds a simulation of all three kinds, though the timing precision is limited by what the kernel's scheduling machinery can deliver. The goal is not to match silicon microsecond-for-microsecond; the goal is to make the simulation realistic enough that the driver's important protocol paths get exercised and the race conditions those paths contain get a chance to manifest.

17.2.3 What a Good Simulation Teaches and What It Does Not

A good simulation teaches protocol. It exercises every branch in the driver's state machine, every error path, every status check, every delay tolerance. A driver written against a good simulation is robust against the scenarios the simulation covers.

A good simulation does not teach silicon. The CPU's memory hierarchy, the device's internal pipelines, the signal integrity of the PCB traces, the interaction with the rest of the system's traffic, the thermal behaviour of the chip under load: none of these are in the simulation. A driver that passes simulation can still fail on real hardware if the failure is rooted in physics.

This distinction matters. Chapter 17's simulation is a teaching tool for protocol, and the driver it produces is a vehicle for learning, not a production network driver. Chapters 20 through 22 introduce DMA, interrupts, and real device performance concerns, all of which extend the simulation in directions that need real hardware to validate fully. Chapter 17 is step one of a longer progression: get the protocol right, then get the real-hardware details right.

There is a second subtlety worth naming. A simulation can teach the *wrong* protocol if the simulation is inconsistent with the real device. A reader who learns to expect the simulation's behaviour, then encounters real hardware whose behaviour diverges, will be confused. Chapter 17's simulation is deliberately designed to match the patterns common in real FreeBSD drivers: `STATUS_READY` clears while the device is busy and sets when it is ready, `STATUS_DATA_AV` latches until the driver reads the data, `INTR_STATUS` is read-to-clear, `CTRL_RESET` is write-one-to-self-clear. These patterns are not universal; different devices choose different conventions. But they are common, and a driver that handles these patterns well has learned transferable skill.

17.2.4 No Hardware Dependency: The Practical Benefit

For this book, the most immediate benefit of simulation is that the reader does not need specific hardware to follow along. A reader working through Chapter 16 on a laptop with no external devices, on a virtual machine with no PCI passthrough, on an ARM single-board computer with only the onboard peripherals, can still build the driver and watch it behave. Chapter 17 preserves that property. The simulation lives entirely in kernel memory; the driver needs nothing beyond a running FreeBSD 14.3 system.

This matters for a teaching book. A reader who has to stop, order hardware, install it, and wait for delivery before they can continue is a reader who may not continue. A reader who can build and run the driver in the same session they read the chapter is a reader who is learning. Every Chapter 17 lab and challenge is designed to run on any x86 FreeBSD system without external hardware.

A secondary benefit is that simulation lets the book run ahead of what a reader's hardware can do. The driver in Chapter 17 exercises error paths that a real device would almost never trigger under normal operation. A reader who owns a production-quality network card would have trouble forcing the card into the kinds of failure modes Chapter 7's fault injection teaches. A simulated device can be told to fail on demand, as often as needed, under any condition the reader wants to exercise. The learning surface is broader.

17.2.5 Full Control: The Pedagogical Benefit

Simulation gives the reader more control than real hardware ever does. The reader can freeze time in the simulation, step forward by a known number of ticks, and observe every intermediate state. The reader can

disable the simulation's autonomous behaviour, make a single manipulation, and re-enable it. The reader can inject a specific fault, observe how the driver reacts, and then repair the fault by toggling a sysctl. None of this is possible against real silicon.

This level of control is pedagogically valuable. A chapter that explains “when the device is busy, the driver should wait for `STATUS.READY`” can demonstrate the scenario concretely. Chapter 17 does just that: a sysctl tells the simulation to enter a fake-busy state, and the reader can type commands while the driver patiently waits for the busy bit to clear. On real hardware, producing this scenario reliably would require a specific workload that the device would reject, or an artificial test fixture, or a debugger. In simulation, it is a one-line sysctl change.

Control also means the reader can experiment without consequences. Setting a register to an illegal value, issuing a command the device does not expect, asking the driver to access a register outside the mapped range: each of these is safe in simulation. The worst case is a `KASSERT` firing and a panic that a debug kernel catches cleanly. The reader can try ten bad ideas in an hour, learn from each, and come out better for it. Experimentation with real hardware carries real risks, and readers (reasonably) exercise more caution there.

17.2.6 Safe Experiments: The Risk-Reduction Benefit

The word “safe” deserves expansion. A kernel panic from a simulated device is an embarrassment; a kernel panic from a real device can be worse. A driver that writes a wrong bit to a real device's `CTRL` register can brick the device. A driver that corrupts a real device's DMA configuration can write garbage over arbitrary RAM. A driver that mishandles a real interrupt can leave the system in a state where legitimate interrupts are missed indefinitely. Each of these failure modes has appeared in production FreeBSD over the years, and each has required a kernel update to fix.

Simulation eliminates most of this risk. The simulated device is a struct in kernel memory; a wrong bit goes into memory that is not connected to anything else. The simulated DMA does not exist yet (it arrives in Chapter 20). The simulated interrupts are callouts that the driver can stop at any time. A beginner writing their first driver can make every mistake the driver development lifecycle contains without causing a physical problem.

This is worth internalising. Chapter 17's simulation gives the reader permission to experiment aggressively, which is what a beginner needs to do to build confidence. A reader who has never broken a driver is a reader who has not yet learned how drivers break. Chapter 17 is the chapter where the reader learns both how to break a driver on purpose and how to recognise it when it breaks on its own.

17.2.7 What We Will Simulate in Chapter 17

A short list of the specific behaviours the chapter will introduce. Each is added in the section that discusses it; the list here is a roadmap.

- **Autonomous `STATUS` updates.** A callout fires every 100 milliseconds and updates fields in `STATUS` as if a background hardware monitor were sampling the device's internal state. The driver can observe the updates; the test suite can assert them.
- **Command-triggered delayed events.** Writing `CTRL.GO` schedules a callout that, after a configurable delay (default 500 milliseconds), sets `STATUS.DATA_AV` and copies `DATA_IN` to `DATA_OUT`. This is the textbook “asynchronous device” pattern.

- **Read-to-clear `INTR_STATUS`.** Reading `INTR_STATUS` returns the current pending-interrupt bits and then atomically clears them. Writing `INTR_STATUS` has no effect (real devices vary; some allow write-one-to-clear, which is a common variant we will discuss).
- **Simulated sensor data.** A `SENSOR` register holds a value that the simulation updates on a cadence, mimicking a temperature or pressure reading. The driver can poll it; a user-space tool can graph it.
- **Timeout tolerance.** The driver's command path waits up to a configurable number of milliseconds for `STATUS.DATA_AV` to assert. If the simulation is configured to delay longer than the timeout, the driver reports a timeout and recovers.
- **Random fault injection.** A `sysctl` controls the probability that any operation will inject a fault. The fault types include: `STATUS.ERROR` set after a command, the next command timing out entirely, a read returning `0xFFFFFFFF`, the device reporting fake-busy indefinitely. Each fault exercises a different driver path.

Every one of these behaviours is implemented by Section 8. The chapter builds them one at a time so the reader can internalise each before moving on.

17.2.8 The Simulation's Place in the Book

Chapter 17 sits between Chapter 16 (static register access) and Chapter 18 (real PCI hardware). It is the bridge between the abstract vocabulary and the concrete device. A reader who completes Chapter 17 has a driver that has exercised nearly every protocol pattern the book will discuss, against a simulated device that behaves like a real one. That reader walks into Chapter 18 with a driver that is already close to production-grade in structure, needing only the real-PCI glue to run on actual silicon.

The simulation is also a reference. When Chapter 18 replaces the simulated register block with a real PCI BAR, the reader can compare the simulation's behaviour against the real device's behaviour and notice the differences. The simulation is the yardstick. When Chapter 19 adds interrupts, the simulation's callout-driven state changes become the model for what an interrupt handler reacts to. When Chapter 20 adds DMA, the simulation's data registers become the model for what the DMA engine reads and writes. The simulation is the teaching vocabulary that every later chapter extends.

17.2.9 Wrapping Up Section 1

Simulation is a permanent part of driver development, not a temporary scaffold. It gives driver authors reproducible tests, controlled failure injection, and the ability to exercise error paths that real hardware rarely produces. Chapter 17 builds a simulation of three kinds (behavioural, timing-aware, side-effect-aware) on top of the Chapter 16 driver. The simulation is deliberately designed to match common patterns in real FreeBSD drivers so the reader's intuition transfers to real hardware later.

The chapter's specific goals include autonomous `STATUS` updates, command-triggered delayed events, read-to-clear semantics, simulated sensor data, timeout tolerance, and random fault injection. Each is introduced in the section that needs it; together they produce a driver rich enough to exercise almost every Part 3 lesson.

Section 2 takes the next step. Before we simulate device behaviour, we need a register map that the behaviour operates on. Chapter 16 gave us a static map; Chapter 17 extends it for dynamic behaviour.

17.3 Section 2: Designing the Register Map

Chapter 16 gave the simulated device a register map. Chapter 17's work extends that map, but the extensions are not automatic. Adding a register to a real device requires a team of hardware engineers, a silicon respin, and a new datasheet. Adding a register to a simulated device is easy in mechanical terms, but the design work (what should the register do? who writes it? who reads it? what are its side effects?) is the same, and the discipline of doing it well is the same.

This section walks through the design, not just the result. By the end of Section 2 you will have a register map that supports every behaviour Section 1 listed, documented well enough that the implementation in Sections 3 through 7 is almost mechanical. The exercise is transferable: every time you read a real datasheet, you will recognise the decisions the hardware designers made, and you will have a vocabulary for evaluating whether the decisions were wise.

17.3.1 What Is a Register Map?

A register map is the complete catalogue of a device's register interface. It lists every register, every offset, every width, every field, every bit, every access type, every reset value, every side effect. A real device's register map may run to hundreds of pages; a simple device's register map may fit on a single page. The form is similar in both cases: a table, or a set of tables, with headings that answer the questions a driver author needs to ask.

A register map is not prose. It is a reference document. The driver author consults it when writing code, and the test author consults it when writing tests. A register map that is vague, incomplete, or ambiguous produces drivers that are wrong in ways that are hard to diagnose. A register map that is precise produces drivers that match what the device expects.

For a simulated device, the register map plays one extra role. It is also the specification against which the simulation's behaviour is tested. If the simulation disagrees with the register map, the simulation is wrong and must be fixed. If the driver disagrees with the register map, the driver is wrong and must be fixed. The map is the contract. Changes to the map require changes to both the simulation and the driver. This is exactly how real device development works: the datasheet is the contract, and the hardware team, the firmware team, and the driver team all work to it.

17.3.2 The Chapter 16 Map as a Starting Point

Recall the Chapter 16 register map:

Offset	Width	Name	Access	Chapter 16 Behaviour
0x00	32 bit	CTRL	R/W	Plain read-write memory.
0x04	32 bit	STATUS	R/W	Plain read-write memory.
0x08	32 bit	DATA_IN	R/W	Plain read-write memory.
0x0c	32 bit	DATA_OUT	R/W	Plain read-write memory.
0x10	32 bit	INTR_MASK	R/W	Plain read-write memory.
0x14	32 bit	INTR_STATUS	R/W	Plain read-write memory.

Offset	Width	Name	Access	Chapter 16 Behaviour
0x18	32 bit	DEVICE_ID	R-only	Fixed 0x4D594649.
0x1c	32 bit	FIRMWARE_REV	R-only	Fixed 0x00010000.
0x20	32 bit	SCRATCH_A	R/W	Plain read-write memory.
0x24	32 bit	SCRATCH_B	R/W	Plain read-write memory.

Ten registers, 40 bytes of defined space, 64 bytes allocated to leave room for growth. Every access is direct: reads return whatever the last write stored, writes go straight into memory. No side effects on reads. No side effects on writes. No hardware state machine behind the registers.

Chapter 17 changes two things about this map. First, the existing registers grow behavioural semantics: `STATUS.DATA_AV` will be set by the simulation, not just by the driver's writes; `INTR_STATUS` becomes read-to-clear. Second, the map adds a small number of new registers that the Chapter 17 simulation needs: a `SENSOR` register, a `DELAY_MS` register, a `FAULT_MASK` register, and a few more. The 64-byte allocation from Chapter 16 already has room; no allocator change is required.

17.3.3 Register Access Semantics

Before we write any register, we should name the access semantics we care about. A register's access semantics are the rules for what happens when the driver reads or writes it. The main categories in common use are:

Read-only (RO). The register returns a value the device or the simulation produced. Writes are ignored or produce an error. `DEVICE_ID` and `FIRMWARE_REV` are examples.

Read/Write (RW). The register stores whatever the driver writes. Reads return the last-written value. `CTRL`, `SCRATCH_A`, and `SCRATCH_B` are examples.

Write-only (WO). The register accepts writes, but reads return a fixed value (often zero, sometimes garbage, occasionally the last-read value from a different source). `DATA_IN` is often write-only in real devices.

Read-to-clear (RC). Reading the register returns its current value and then clears it. Writes are typically ignored. `INTR_STATUS` is the classic example.

Write-one-to-clear (W1C). Writing a 1 to a bit clears that bit; writing a 0 has no effect. Reads return the current value. `INTR_STATUS` on some devices uses W1C instead of RC.

Write-one-to-set (W1S). Writing a 1 sets the bit; writing a 0 has no effect. Less common than W1C, but used in some hardware for control registers where the driver should not have to do a read-modify-write.

Read/write with side effect (RWSE). The register is readable and writable, but the write (or the read) triggers something beyond a plain memory update. `CTRL` often falls in this category: writing `CTRL.RESET` triggers a device reset, which is a side effect.

Sticky (latched). A bit, once set by the hardware, stays set until the driver explicitly clears it. Error bits in `STATUS` are usually sticky. `STATUS.ERROR` will be sticky in the Chapter 17 map.

Reserved. The bit has no defined meaning. Writes should either preserve the existing value or write zero; reads may return any value. A driver that ignores reserved bits is robust against future hardware revisions; a driver that depends on reserved-bit behaviour is fragile.

Each access type has implications for the driver's code. A driver reading an RC register should do so exactly when the protocol demands, because every extra read consumes an event. A driver reading a WO register is reading garbage, which is a bug waiting to appear if the value is ever acted on. A driver doing a read-modify-write on a sticky-bit register must be careful not to clear a bit it did not intend to touch. A driver writing a WIC register uses `CSR_WRITE_4(sc, REG, mask)` and not `CSR_UPDATE_4(sc, REG, mask, 0)`; the two look similar but produce different behaviour on the hardware.

Chapter 17's simulation uses RO, RW, RC, sticky, and RWSE semantics. It does not introduce WIC or WIS (though the challenges invite the reader to add them). The simulation's goal is to cover the common cases thoroughly; the less common cases become natural extensions once the common ones are understood.

17.3.4 Chapter 17 Additions

The Chapter 17 register map adds the following registers:

Offset	Width	Name	Access	Chapter 17 Behaviour
0x28	32 bit	SENSOR	RO	Simulated sensor value. Updated by a callout every 100 ms.
0x2c	32 bit	SENSOR_CONFIG	RW	Sensor update interval and amplitude settings.
0x30	32 bit	DELAY_MS	RW	Number of milliseconds the device takes per command.
0x34	32 bit	FAULT_MASK	RW	Which simulated faults are enabled. Bitfield.
0x38	32 bit	FAULT_PROB	RW	Probability of random fault per operation (0..10000).
0x3c	32 bit	OP_COUNTER	RO	Count of commands the simulated device has processed.

Six new registers, using offsets 0x28 through 0x3c, fitting entirely inside the 64-byte region Chapter 16 allocated. No softc change, no allocator change, no `bus_space_map` change.

Each register has a specific job:

- **SENSOR** is the simulated temperature, pressure, or voltage. The exact interpretation does not matter for the simulation; what matters is that it changes on its own, teaching the reader how to handle registers whose value is produced autonomously. The value oscillates around a baseline using a simple formula the callout will implement.
- **SENSOR_CONFIG** lets the driver (or the user through a `sysctl`) control the sensor's behaviour. The low 16 bits are the update interval in milliseconds; the high 16 bits are the amplitude of the oscillation. A value of `0x0064_0040` means "100 ms interval, amplitude 64". Changing the register changes the simulation's next update.
- **DELAY_MS** is how long a command takes in the simulation. Writing `CTRL.GO` schedules `STATUS.DATA_AV` to assert `DELAY_MS` milliseconds later. The default is 500. Setting it to 0 makes commands complete on the next callout tick; setting it to a large value lets the reader exercise the driver's timeout path.
- **FAULT_MASK** is a bitfield that selects which fault modes are active. Bit 0 enables the "operation times out" fault; bit 1 enables the "read returns all-ones" fault; bit 2 enables the "error bit set after every command" fault; bit 3 enables the "device always busy" fault. Multiple bits can be set simultaneously.

- **FAULT_PROB** controls the probability that any single operation will inject a fault, in tenths of a basis point. A value of 10000 is 100% (fault every time); 5000 is 50%; 0 disables random faulting entirely. This gives the reader fine-grained control over how aggressively faults are injected.
- **OP_COUNTER** counts every command the simulation has processed. It is read-only; writes are ignored. The driver can use it to verify that a command actually reached the simulation, and the user can read it from user space to check activity.

17.3.5 Header Additions for the New Registers

The header for the additions lives in `myfirst_hw.h`, extending the Chapter 16 definitions:

```
/* Chapter 17 additions to the simulated device's register map. */

#define MYFIRST_REG_SENSOR          0x28
#define MYFIRST_REG_SENSOR_CONFIG 0x2c
#define MYFIRST_REG_DELAY_MS       0x30
#define MYFIRST_REG_FAULT_MASK     0x34
#define MYFIRST_REG_FAULT_PROB     0x38
#define MYFIRST_REG_OP_COUNTER     0x3c

/* SENSOR_CONFIG register fields. */
#define MYFIRST_SCFG_INTERVAL_MASK 0x0000ffffu /* interval in ms */
#define MYFIRST_SCFG_INTERVAL_SHIFT 0
#define MYFIRST_SCFG_AMPLITUDE_MASK 0xffff0000u /* oscillation range */
#define MYFIRST_SCFG_AMPLITUDE_SHIFT 16

/* FAULT_MASK register bits. */
#define MYFIRST_FAULT_TIMEOUT      0x00000001u /* next op times out          */
#define MYFIRST_FAULT_READ_1S     0x00000002u /* reads return 0xFFFFFFFF */
#define MYFIRST_FAULT_ERROR       0x00000004u /* STATUS.ERROR after op    */
#define MYFIRST_FAULT_STUCK_BUSY  0x00000008u /* STATUS.BUSY latched on  */

/* CTRL register: GO bit (new for Chapter 17). */
#define MYFIRST_CTRL_GO           0x00000200u /* bit 9: start command    */

/* STATUS register: BUSY and DATA_AV are now dynamic (set by simulation). */
/* (The mask constants were already defined in Chapter 16.) */
```

The new CTRL bit `MYFIRST_CTRL_GO` does not overlap any existing bit. `MYFIRST_CTRL_ENABLE` is bit 0, `RESET` is bit 1, `MODE_MASK` covers bits 4 through 7, `LOOPBACK` is bit 8. The new GO bit at bit 9 fits cleanly in the gap.

The comments in the header are deliberately terse. In a real driver, comments at this level would point at the datasheet section (for example, */* See datasheet section 4.2.1 */*). For the simulation, the datasheet is this chapter; a reader looking for the definitive behaviour of `MYFIRST_FAULT_ERROR` should consult Section 7 of Chapter 17.

17.3.6 Writing a Register Map Table for Your Own Device

The exercise of writing a register map is one of the most valuable things you can do as a beginning driver author. Even if you never write the simulation or the driver, the act of sitting down and thinking “what would this device’s registers be?” forces you to think about the device as an entity with a protocol rather than as a magic box.

A useful template:

1. Decide what the device does at a high level. One sentence.
2. Identify the commands the device must accept. Each command may become a bit in a CTRL register, a value in a COMMAND register, or its own register.
3. Identify the states the device reports. Each state may become a bit in a STATUS register, a value in a STATE register, or its own read-only register.
4. Identify the data the device produces or consumes. Input data goes in a DATA_IN register or a FIFO. Output data comes from a DATA_OUT register or a FIFO.
5. Identify the interrupts the device can raise. Each interrupt source becomes a bit in INTR_STATUS (or an equivalent).
6. Identify any metadata the driver needs at startup. Device ID, firmware revision, feature flags, capability bits.
7. Leave space for configuration registers the driver may want to tune. In Chapter 17, DELAY_MS, FAULT_MASK, and FAULT_PROB are configuration registers; a real device might have timeout thresholds, error recovery options, power management settings.
8. Decide the width of each register. 32 bits is the common default on modern devices. Narrower widths exist for legacy devices or for devices where the register count is very large.
9. Decide the offset of each register. Keep related registers contiguous when possible. Align offsets to the register’s width (32-bit registers at offsets divisible by 4).
10. Decide the access semantics (RO, RW, RC, W1C, ...). Be explicit; ambiguity here produces bugs.
11. Decide the reset value of each register. What should the register hold at power-on? At reset? At module attach?
12. Document the side effects of every non-trivial register.

This list is long, but each step is small. A real device team might take weeks to produce a register map this detailed. A simulated device’s map can be produced in an afternoon. The Chapter 17 map is the result of this exercise applied to a teaching device.

17.3.7 Example Layout and Design Choices

To make the discipline concrete, walk through the design decisions behind the Chapter 17 additions.

Decision 1: Group related registers. The new registers SENSOR (0x28) and SENSOR_CONFIG (0x2c) are adjacent. A driver reading the sensor code can access both with a single region read if needed, and a reader scanning the register dump sees them together. The same logic puts FAULT_MASK (0x34) and FAULT_PROB (0x38) adjacent.

Decision 2: Put configuration after diagnostic. DEVICE_ID (0x18) and FIRMWARE_REV (0x1c) are read-only diagnostic registers; they come first in the high-offset section. SENSOR and its config come next. DELAY_MS,

FAULT_MASK, FAULT_PROB are configuration; they come after. OP_COUNTER is another diagnostic; it goes at the end.

Decision 3: Use the existing CTRL register for the new GO bit. Rather than adding a dedicated COMMAND register, the GO bit in CTRL is how the driver initiates a command. This matches the Chapter 16 pattern (the ENABLE and RESET bits are both in CTRL) and saves a register.

Decision 4: Make OP_COUNTER read-only. A writable counter would invite bugs where the driver accidentally reset the counter mid-operation. A read-only counter is always monotonic (modulo 32-bit wrap) and always reflects the simulation's view.

Decision 5: Encode two fields in SENSOR_CONFIG. Rather than two separate registers for interval and amplitude, a single register with two 16-bit fields saves one slot and matches a common hardware pattern. A real device might have an 8-bit command ID in the high byte and a 24-bit argument in the lower three bytes of a register; the principle is the same.

Decision 6: Use FAULT_PROB in tenths of a basis point (range 0 to 10000). Using 10000 as full-scale rather than 100 gives two extra decimal places of precision. Fault probabilities of 0.5%, 0.75%, 1.25% become expressible as integers without fractional arithmetic.

Each decision is small; the composition is what makes the map usable. A register map designed carelessly produces a driver full of awkward patches; a map designed carefully produces a driver that writes itself.

17.3.8 Naming Conventions for Constants

A note on the naming conventions Chapter 17 uses for bit masks, which mirror what real FreeBSD drivers do.

The prefix MYFIRST_ marks everything as belonging to the driver. Inside that, REG_ indicates a register offset, CTRL_ indicates a bit or field in CTRL, STATUS_ indicates a bit in STATUS, and so on. The pattern is <driver>_<register>_<field>. For example, MYFIRST_STATUS_READY is the READY bit in the STATUS register. The name is long, but it is unambiguous, and modern editors complete it quickly.

Masks that cover multiple bits end with _MASK; shifts to align the mask end with _SHIFT. MYFIRST_CTRL_MODE_MASK is the mask for the 4-bit MODE field; MYFIRST_CTRL_MODE_SHIFT is the number of bits to shift right to bring the field to bit zero. Extracting the field reads as `(ctrl & MYFIRST_CTRL_MODE_MASK) >> MYFIRST_CTRL_MODE_SHIFT`; setting it reads as `(mode << MYFIRST_CTRL_MODE_SHIFT) & MYFIRST_CTRL_MODE_MASK`.

Fixed values end with _VALUE. MYFIRST_DEVICE_ID_VALUE is the constant value of the DEVICE_ID register. This convention keeps values distinguishable from masks at a glance.

Real drivers sometimes deviate from this convention, often for historical reasons or because their register names are long. The `if_em` driver uses shorter prefixes (EM_ rather than E1000_ in some places) to fit within reasonable line lengths. The `if_ale` driver uses CSR_READ_4(sc, ALE_REG_NAME) with ALE_ as the prefix. The principle is the same across drivers: every constant is visibly part of one driver's namespace, and the pattern inside the namespace is consistent.

For Chapter 17, consistency matters more than brevity. The reader is still learning the patterns; a long, explicit name is more educational than a short, ambiguous one.

17.3.9 Validating the Map Before Writing Code

Before implementing the simulation, it is worth validating the map against a few sanity checks.

Check 1: Offsets fit inside the allocated region. The Chapter 16 region is 64 bytes. The highest defined offset in the Chapter 17 map is 0x3c, with width 4, so the last byte used is at offset 0x3f. 0x40 is one past the allocated region. The map fits exactly; there is no room for a further register without enlarging the allocation. That is a design constraint worth knowing.

Check 2: All offsets are 4-byte aligned. Every register is 32 bits wide, and every offset is a multiple of 4. `bus_space_read_4` and `bus_space_write_4` require 4-byte alignment on most platforms; an unaligned access would fault on some architectures and silently misbehave on others. The map follows the rule.

Check 3: No two registers overlap. Each 32-bit register occupies 4 consecutive bytes. Check that adjacent offsets differ by at least 4. They do: 0x00, 0x04, 0x08, 0x0c, 0x10, 0x14, 0x18, 0x1c, 0x20, 0x24, 0x28, 0x2c, 0x30, 0x34, 0x38, 0x3c.

Check 4: No reserved bits are used for multiple purposes. A bit defined for one field must not appear in another field. `MYFIRST_CTRL_ENABLE` is bit 0, `RESET` is bit 1, `MODE` covers bits 4 through 7, `LOOPBACK` is bit 8, `GO` is bit 9. No overlap.

Check 5: Access semantics are consistent with the simulation's intent. `DEVICE_ID` and `FIRMWARE_REV` are RO, which means the simulation will refuse to change them after attach. `INTR_STATUS` will be RC, meaning the simulation clears it when the driver reads it. `OP_COUNTER` is RO, meaning writes do nothing. Everything else is RW (possibly with side effects). This is consistent with what Section 1 described.

Check 6: Reset values are sensible. At attach, `DEVICE_ID` is 0x4D594649, `FIRMWARE_REV` is 0x00010000, `STATUS` has `READY` set, everything else is zero. `DELAY_MS` defaults to 500 (500 ms per command). `SENSOR_CONFIG` defaults to 0x0064_0040 (100 ms interval, amplitude 64). `FAULT_MASK` defaults to 0 (no faults enabled). `FAULT_PROB` defaults to 0 (no random faults). The defaults describe a simulation that behaves like a reliable device until told otherwise.

These checks are the kind of thing a disciplined driver author runs through before writing any code. They take a few minutes and catch several common bugs before they appear.

17.3.10 Documenting the Map

Chapter 16 introduced `HARDWARE.md`. Chapter 17 extends it. The section headings in `HARDWARE.md` at the end of Chapter 17 will be:

1. Version and scope
2. Register summary table (every register, every offset, every width, every access type, every default value)
3. `CTRL` register fields (table of every bit in `CTRL`)
4. `STATUS` register fields (table of every bit in `STATUS`)
5. `INTR_MASK` and `INTR_STATUS` fields (table of every interrupt source)
6. `SENSOR_CONFIG` fields (interval and amplitude)
7. `FAULT_MASK` fields (table of every fault type)
8. Register-by-register reference (one paragraph per register, describing behaviour, side effects, and usage)
9. Common sequences (write-then-read-to-verify, command-and-wait, fault-injection-for-test)
10. Observability (every `sysctl` that reads or writes a register)

11. Simulation notes (which registers are dynamic, which are static, what the callouts do)

This document is long, perhaps 100 lines. It is the single source of truth for what the driver expects from the simulation. A reader who forgets a register's behaviour opens `HARDWARE.md` and finds the answer. A contributor adding a new register updates `HARDWARE.md` first, then changes the code.

17.3.11 Common Mistakes in Register Map Design

Beginners designing their first register map often make the same few mistakes. A brief catalogue so you can avoid them.

Mistake 1: Using magic numbers in the code instead of named constants. A driver that writes `CSR_WRITE_4(sc, 0x00, 0x1)` instead of `CSR_WRITE_4(sc, MYFIRST_REG_CTRL, MYFIRST_CTRL_ENABLE)` is hard to read and impossible to refactor. The rule is: every offset is a constant; every bit is a constant; every field is a macro.

Mistake 2: Overlapping bit definitions. Two names, one bit. A reader doing a bitwise OR of `FLAG_A` and `FLAG_B` gets a value that corresponds to a third thing entirely, because `FLAG_A` and `FLAG_B` turn out to be the same bit. The defence is a comment next to each bit name listing its bit number, plus a sanity check at compile time (`_Static_assert((MYFIRST_CTRL_ENABLE & MYFIRST_CTRL_RESET) == 0, "bits overlap")` if you want to be defensive).

Mistake 3: Unaligned accesses. Defining a 32-bit register at offset `0x06` looks harmless but fails on any architecture where unaligned accesses are disallowed. Always align offsets to the register's width.

Mistake 4: Forgetting reserved bits. A 32-bit register where only bits 0 through 3 are defined may have unspecified values in bits 4 through 31. A driver that writes `CSR_WRITE_4(sc, REG, 0xF)` is fine for now, but a future revision that defines bit 5 may break because the driver is effectively writing zero to it every time. The defence is read-modify-write on registers with reserved bits: `CSR_UPDATE_4(sc, REG, mask_of_fields_we_touch, new_field_values)`.

Mistake 5: Side effects on read that the driver does not expect. An RC register that the driver reads for debugging purposes clears state the driver was about to act on. The defence is to document every side effect explicitly in the header and in `HARDWARE.md`, so a reader can see the danger before writing the bug.

Mistake 6: Assuming writes always succeed. A write to a RO register may be silently ignored, or may raise a fault, or may do something surprising on certain hardware. The defence is to use the correct access semantics and to stress-test the driver's use of each register.

Mistake 7: Inconsistent widths. A 32-bit register next to an 8-bit register where the access width is not obvious from the name. The defence is to make the width part of the accessor call (`CSR_READ_4` vs `CSR_READ_1`) and to name the register's type somewhere visible.

Mistake 8: Overly clever bit layouts. A 16-bit field that spans two non-adjacent byte lanes, requiring bit-gathering in the driver. Real devices do this sometimes (often because of historical constraints), but a simulated device should stick to contiguous fields. The simulation is a teaching device; clarity wins over cleverness.

Mistake 9: Missing reset values. A register whose reset value is unspecified forces the driver to guess what the register holds after attach. The defence is to initialise every register to a documented value in the attach path.

Mistake 10: Stale documentation. The code changes, the documentation does not. Two months later the driver works but the documentation lies. The defence is the discipline of updating `HARDWARE.md` in the same commit that changes the code.

Each mistake is small in isolation and expensive in aggregate. A driver written without these mistakes is a driver that reads well, tests well, and ages well.

17.3.12 Wrapping Up Section 2

A register map is the specification of a device's programmer-visible interface. Chapter 17's map extends Chapter 16's with six new registers: `SENSOR`, `SENSOR_CONFIG`, `DELAY_MS`, `FAULT_MASK`, `FAULT_PROB`, and `OP_COUNTER`. Each register has a purpose, a width, an access type, a reset value, and a documented behaviour. The design choices behind each register are worth understanding, because they mirror the decisions real device designers make.

The map is documented in the header (`myfirst_hw.h`) and in `HARDWARE.md`. The header is consumed by the compiler; the markdown is consumed by humans. Both are kept in sync as the code evolves.

Section 3 takes the map and implements the simulation. The first callout fires; the first autonomous register update happens; the static Chapter 16 block acquires a heartbeat.

17.4 Section 3: Implementing the Simulated Hardware Backend

Section 2 produced a register map. Section 3 makes the map behave. The simulation's first job is to update registers autonomously, without the driver asking it to. That is how real hardware behaves: a sensor refreshes its value on a clock, an interrupt controller latches events as they arrive, a network card's receive counter ticks up as frames arrive. Chapter 17's first simulation step is to produce a similar heartbeat.

The tool is one the reader already knows: a callout. Chapter 13 introduced `callout_init_mtx` and `callout_reset` for internal driver timers. Chapter 17 uses the same primitive for a different purpose: driving simulated device state. The callout fires every 100 milliseconds, acquires the driver's mutex, updates some registers, releases the mutex, and schedules its next firing. From the driver's point of view the registers just change; from the simulation's point of view a timer is running.

17.4.1 The First Simulation File

Chapter 16's file layout was `myfirst.c` for the driver lifecycle, `myfirst_hw.c` for the hardware-access layer, and `myfirst_hw.h` for the shared definitions. Chapter 17 adds a new file, `myfirst_sim.c`, for the simulation backend. The split keeps the simulation code out of the hardware-access file, so a later chapter can replace `myfirst_sim.c` with real PCI-facing code without touching the accessors.

Create `myfirst_sim.c` alongside the Chapter 16 files. The first version is small:

```
/*-
 * myfirst_sim.c -- Chapter 17 simulated hardware backend.
 */
```

```
* Adds a callout that drives autonomous register changes, a
* command-scheduling callout for command-triggered delays, and
* the fault-injection state that Section 7 will populate.
*
* This file assumes Chapter 16's register access layer
* (myfirst_hw.h, myfirst_hw.c) is present and functional.
*/

#include <sys/param.h>
#include <sys/system.h>
#include <sys/kernel.h>
#include <sys/malloc.h>
#include <sys/lock.h>
#include <sys/mutex.h>
#include <sys/callout.h>
#include <sys/sysctl.h>
#include <sys/bus.h>
#include <sys/random.h>
#include <machine/bus.h>

#include "myfirst.h"
#include "myfirst_hw.h"
#include "myfirst_sim.h"
```

The includes follow FreeBSD conventions: `sys/param.h` first, then `sys/system.h`, then the specific subsystem headers. `sys/random.h` will be needed for `arc4random` later; pulling it in now avoids a second edit. The driver's private headers come last.

The file compiles against the existing `myfirst.h` and `myfirst_hw.h`, plus a new `myfirst_sim.h` that will contain the simulation's API. A stub of that header:

```
/* myfirst_sim.h -- Chapter 17 simulation API. */
#ifndef _MYFIRST_SIM_H_
#define _MYFIRST_SIM_H_

struct myfirst_softc;

/* Attach/detach the simulation. Called from myfirst_attach/detach. */
int myfirst_sim_attach(struct myfirst_softc *sc);
void myfirst_sim_detach(struct myfirst_softc *sc);

/* Enable/disable the simulation's autonomous updates. */
void myfirst_sim_enable(struct myfirst_softc *sc);
void myfirst_sim_disable(struct myfirst_softc *sc);
```

```
/* Command scheduling: called when the driver writes CTRL.GO. */
void myfirst_sim_start_command(struct myfirst_softc *sc);

/* Sysctl registration for simulation controls. */
void myfirst_sim_add_sysctls(struct myfirst_softc *sc);

#endif /* _MYFIRST_SIM_H_ */
```

Five functions, one header file, one source file. The API is small. The simulation's internal state lives in a struct myfirst_sim, pointed at from the softc via sc->sim, in the same pattern as sc->hw.

17.4.2 The Simulation State Structure

Inside myfirst_sim.h, define the state structure:

```
struct myfirst_sim {
    /* Autonomous update callout. Fires every sensor_interval_ms. */
    struct callout      sensor_callout;

    /* Command completion callout. Fires DELAY_MS after CTRL.GO. */
    struct callout      command_callout;

    /* Last scheduled command's data. Saved so command_cb can
     * latch DATA_OUT when it fires. */
    uint32_t            pending_data;

    /* Whether a command is currently in flight. */
    bool                command_pending;

    /* Baseline sensor value; the callout oscillates around this. */
    uint32_t            sensor_baseline;

    /* Counter used by the sensor oscillation algorithm. */
    uint32_t            sensor_tick;

    /* Operation counter. Written to OP_COUNTER on every completion. */
    uint32_t            op_counter;

    /* Whether the simulation is running. Stops cleanly at detach. */
    bool                running;
};
```

Seven fields. Two are callouts (sensor updates and command completion). One is boolean state (command in flight). One is the saved pending data. Three are internal counters. One is the running flag.

The softc gains a pointer to the simulation structure:

```
struct myfirst_softc {
    /* ... all existing fields, including hw ... */
    struct myfirst_sim    *sim;
};
```

The allocation and initialisation happen in `myfirst_sim_attach`, called from `myfirst_attach` after `myfirst_hw_attach` has already set up `sc->hw`. Putting the sim attach after the hw attach is deliberate: the simulation reads and writes registers through the Chapter 16 accessors, so the accessors must be ready before the simulation starts.

17.4.3 The Sensor Callout: The First Autonomous Update

The smallest useful simulation update is the sensor callout. It fires every 100 milliseconds, updates the `SENSOR` register, and re-arms itself. The code:

```
static void
myfirst_sim_sensor_cb(void *arg)
{
    struct myfirst_softc *sc = arg;
    struct myfirst_sim *sim = sc->sim;
    uint32_t config, interval_ms, amplitude, phase, value;

    MYFIRST_LOCK_ASSERT(sc);

    if (!sim->running)
        return;

    /* Read the current SENSOR_CONFIG. */
    config = CSR_READ_4(sc, MYFIRST_REG_SENSOR_CONFIG);
    interval_ms = (config & MYFIRST_SCFG_INTERVAL_MASK) >>
        MYFIRST_SCFG_INTERVAL_SHIFT;
    amplitude = (config & MYFIRST_SCFG_AMPLITUDE_MASK) >>
        MYFIRST_SCFG_AMPLITUDE_SHIFT;

    /* Compute a simple oscillation. phase cycles 0..7 and back. */
    sim->sensor_tick++;
    phase = sim->sensor_tick & 0x7;
    value = sim->sensor_baseline +
        ((phase < 4) ? phase : (7 - phase)) *
        (amplitude / 4);

    /* Publish. */
```

```

        CSR_WRITE_4(sc, MYFIRST_REG_SENSOR, value);

        /* Re-arm at the current interval. */
        if (interval_ms == 0)
            interval_ms = 100;
        callout_reset_sbt(&sim->sensor_callout,
            interval_ms * SBT_1MS, 0,
            myfirst_sim_sensor_cb, sc, 0);
    }

```

Step by step.

First, the callout asserts the lock. A callout that was scheduled with `callout_init_mtx(&co, &mtx, 0)` automatically acquires `mtx` before calling the callback, so the assertion is redundant for a well-disciplined driver, but it is cheap and it documents the invariant. Debug kernels catch any violation.

Second, the callout checks the running flag. If the simulation is being torn down (detach path), the callout exits immediately without re-arming. The detach path then drains the callout with `callout_drain`, which waits for any in-flight callback to finish.

Third, the callback reads `SENSOR_CONFIG` to learn the interval and amplitude. Reading the config each time means the user can change it through a `sysctl` and the next callback will use the new value, without any signalling mechanism beyond the register write.

Fourth, the callback computes an oscillating value. The phase $< 4 ? \text{phase} : (7 - \text{phase})$ trick produces a triangle wave that goes 0, 1, 2, 3, 4, 3, 2, 1, 0, 1, ... over successive calls. Multiplied by `amplitude / 4`, it oscillates between baseline and baseline + amplitude. The formula is simple by design; the point is to produce visible change, not to model a real sensor.

Fifth, the callback writes the new value to `SENSOR`. The write is what the driver or a user-space observer will see.

Sixth, the callback re-arms itself. `callout_reset_sbt` takes a signed binary time, so `interval_ms * SBT_1MS` converts milliseconds into the right units. The `pr` argument (precision) is zero, meaning the kernel can defer the callback by up to 100% of the interval to batch with other timers; a lower value would force stricter timing. The `flags` argument is zero, meaning no special flags like `C_DIRECT_EXEC` or `C_HARDCLOCK`; the callback runs on a normal callout thread with the registered mutex held.

If `interval_ms` was zero, the code defaults to 100 ms. A zero interval would immediately re-arm with zero delay, which would spin the kernel in a tight loop; the guard is defensive.

17.4.4 Initialising the Sensor Callout

In `myfirst_sim_attach`:

```

int
myfirst_sim_attach(struct myfirst_softc *sc)
{

```

```

struct myfirst_sim *sim;

sim = malloc(sizeof(*sim), M_MYFIRST, M_WAITOK | M_ZERO);

/* Initialise the callouts with the main mutex. */
callout_init_mtx(&sim->sensor_callout, &sc->mtx, 0);
callout_init_mtx(&sim->command_callout, &sc->mtx, 0);

/* Pick a baseline sensor value; 0x1000 is arbitrary but visible. */
sim->sensor_baseline = 0x1000;

/* Default config: 100 ms interval, amplitude 64. */
MYFIRST_LOCK(sc);
CSR_WRITE_4(sc, MYFIRST_REG_SENSOR_CONFIG,
    (100 << MYFIRST_SCFG_INTERVAL_SHIFT) |
    (64 << MYFIRST_SCFG_AMPLITUDE_SHIFT));
CSR_WRITE_4(sc, MYFIRST_REG_DELAY_MS, 500);
CSR_WRITE_4(sc, MYFIRST_REG_FAULT_MASK, 0);
CSR_WRITE_4(sc, MYFIRST_REG_FAULT_PROB, 0);
CSR_WRITE_4(sc, MYFIRST_REG_OP_COUNTER, 0);
MYFIRST_UNLOCK(sc);

sc->sim = sim;
return (0);
}

```

The function allocates the `sim` structure, initialises its callouts with the driver's mutex (so the callbacks run with `sc->mtx` held automatically), picks a baseline sensor value, writes the default configuration to the simulation registers under the lock, and stores the pointer in the `softc`.

The callouts are not started yet. The simulation is ready but idle. A separate function, `myfirst_sim_enable`, starts the callouts; a user can toggle the simulation on and off through a `sysctl`. This separation is useful for debugging and for tests that need a known-quiet starting state.

17.4.5 Enabling and Disabling the Simulation

```

void
myfirst_sim_enable(struct myfirst_softc *sc)
{
    struct myfirst_sim *sim = sc->sim;
    uint32_t config, interval_ms;

    MYFIRST_LOCK_ASSERT(sc);

```

```
    if (sim->running)
        return;

    sim->running = true;

    config = CSR_READ_4(sc, MYFIRST_REG_SENSOR_CONFIG);
    interval_ms = (config & MYFIRST_SCFG_INTERVAL_MASK) >>
        MYFIRST_SCFG_INTERVAL_SHIFT;
    if (interval_ms == 0)
        interval_ms = 100;

    callout_reset_sbt(&sim->sensor_callout,
        interval_ms * SBT_1MS, 0,
        myfirst_sim_sensor_cb, sc, 0);
}

void
myfirst_sim_disable(struct myfirst_softc *sc)
{
    struct myfirst_sim *sim = sc->sim;

    MYFIRST_LOCK_ASSERT(sc);

    if (!sim->running)
        return;

    sim->running = false;

    callout_stop(&sim->sensor_callout);
    callout_stop(&sim->command_callout);
}
```

`myfirst_sim_enable` sets `running` to true and schedules the first sensor callout. `myfirst_sim_disable` clears the flag and stops both callouts. `callout_stop` does not wait for any in-flight callback to finish; it just cancels any pending reschedule. The in-flight callback will notice `running == false` on its next invocation and exit.

Note that neither `callout_stop` nor `callout_reset_sbt` releases the mutex. Both are safe to call with the mutex held. This is why `callout_init_mtx` was used: the mutex is threaded through the callout subsystem so the callbacks always run with it held, and the control calls never fight the lock.

Also note that `myfirst_sim_disable` does not drain the callouts. Draining must happen in detach, not in disable. The reason is that `callout_drain` sleeps, and sleeping with a mutex held is illegal. Drain happens in the detach path, after the mutex has been released.

17.4.6 Wiring the Simulation into Attach and Detach

In `myfirst_attach` (the main `myfirst.c` file), add the simulation attach call after the hardware attach:

```
int
myfirst_attach(device_t dev)
{
    struct myfirst_softc *sc;
    int error;

    sc = device_get_softc(dev);

    /* ... existing Chapter 11-15 init ... */

    /* Chapter 16: hardware layer. */
    error = myfirst_hw_attach(sc);
    if (error != 0)
        goto fail_hw;

    /* Chapter 17: simulation backend. */
    error = myfirst_sim_attach(sc);
    if (error != 0)
        goto fail_sim;

    /* ... existing sysctl and cdev setup ... */

    myfirst_hw_add_sysctls(sc);
    myfirst_sim_add_sysctls(sc);

    /* Enable the simulation by default. */
    MYFIRST_LOCK(sc);
    myfirst_sim_enable(sc);
    MYFIRST_UNLOCK(sc);

    return (0);

fail_sim:
    myfirst_hw_detach(sc);
fail_hw:
    /* ... existing unwind ... */
    return (error);
}
```

The simulation is attached after the hardware layer and before the sysctl registration. The simulation is enabled at the end of attach, once everything else is ready. If any step fails, the unwind path runs in reverse order: disable

what was enabled, detach what was attached, free what was allocated.

In `myfirst_detach`, the simulation detach happens early, before the hardware detach:

```
int
myfirst_detach(device_t dev)
{
    struct myfirst_softc *sc = device_get_softc(dev);

    /* ... existing cdev destroy, wait for outstanding fds ... */

    /* Chapter 17: stop the simulation, drain its callouts. */
    MYFIRST_LOCK(sc);
    myfirst_sim_disable(sc);
    MYFIRST_UNLOCK(sc);
    myfirst_sim_detach(sc);

    /* Chapter 16: detach the hardware layer. */
    myfirst_hw_detach(sc);

    /* ... existing synchronization primitive teardown ... */

    return (0);
}
```

Note the ordering. Disable happens with the lock held; detach happens after the lock is released. Detach is where the draining happens:

```
void
myfirst_sim_detach(struct myfirst_softc *sc)
{
    struct myfirst_sim *sim;

    if (sc->sim == NULL)
        return;

    sim = sc->sim;
    sc->sim = NULL;

    callout_drain(&sim->sensor_callout);
    callout_drain(&sim->command_callout);

    free(sim, M_MYFIRST);
}
```

`callout_drain` waits for any in-flight callback to finish. If the callback is running on another CPU when `callout_drain` is called, `callout_drain` blocks until the callback completes and returns. After `callout_drain` returns, no callback will ever run for that callout again, so freeing the simulation state is safe.

The pattern is the same one Chapter 13 taught for callouts in general. The only subtlety is that the mutex must be released before calling `callout_drain`; if the callback is waiting to acquire the mutex when we call drain, and we are holding the mutex, we deadlock. The unlock-before-drain ordering is essential.

17.4.7 A First Test: The Sensor Updates Visibly

Build and load the driver. Then:

```
# kldload ./myfirst.ko
# sysctl dev.myfirst.0.reg_sensor
dev.myfirst.0.reg_sensor: 4096
# sysctl dev.myfirst.0.reg_sensor
dev.myfirst.0.reg_sensor: 4128
# sysctl dev.myfirst.0.reg_sensor
dev.myfirst.0.reg_sensor: 4144
# sysctl dev.myfirst.0.reg_sensor
dev.myfirst.0.reg_sensor: 4160
```

The value is changing between reads. The baseline is $0 \times 1000 = 4096$, and the oscillation goes up by amplitude / 4 = 16 per tick, so the values are 4096, 4112, 4128, 4144, 4160, 4144, 4128, 4112, 4096, ... Four values up, four values down, back to the baseline.

If you read fast enough, you may see the same value twice (the callout fires every 100 ms; a `sysctl` that takes less time to dispatch than 100 ms may see the same state). Slow down and read about once per second, and you will see the full oscillation.

Try changing the config:

```
# sysctl dev.myfirst.0.reg_sensor_config=0x01000040
```

This sets interval to $0 \times 0100 = 256$ ms and amplitude to $0 \times 0040 = 64$. The sensor now updates every 256 ms. You can see the change in the access log:

```
# sysctl dev.myfirst.0.access_log_enabled=1
# sleep 2
# sysctl dev.myfirst.0.access_log_enabled=0
# sysctl dev.myfirst.0.access_log
```

The log should show roughly 8 writes to `SENSOR` over the 2-second window ($2000 \text{ ms} / 256 \text{ ms} \approx 7.8$). Before the config change, the rate would have been 20 writes in 2 seconds.

This is what the simulation is producing. The driver does nothing during this time; the simulation runs on its own, driven by the sensor callout.

17.4.8 A Second Callout: The Command Completion

The sensor callout is periodic. The command callout is one-shot: it fires once, when a command completes, and does nothing until the next command is issued.

```
static void
myfirst_sim_command_cb(void *arg)
{
    struct myfirst_softc *sc = arg;
    struct myfirst_sim *sim = sc->sim;
    uint32_t status;

    MYFIRST_LOCK_ASSERT(sc);

    if (!sim->running || !sim->command_pending)
        return;

    /* Complete the command: copy pending data to DATA_OUT,
     * set STATUS.DATA_AV, clear STATUS.BUSY, increment OP_COUNTER. */
    CSR_WRITE_4(sc, MYFIRST_REG_DATA_OUT, sim->pending_data);

    status = CSR_READ_4(sc, MYFIRST_REG_STATUS);
    status &= ~MYFIRST_STATUS_BUSY;
    status |= MYFIRST_STATUS_DATA_AV;
    CSR_WRITE_4(sc, MYFIRST_REG_STATUS, status);

    sim->op_counter++;
    CSR_WRITE_4(sc, MYFIRST_REG_OP_COUNTER, sim->op_counter);

    sim->command_pending = false;
}
```

The callback does four things. It copies the pending data to DATA_OUT (the simulation’s “result”). It updates STATUS: BUSY clears, DATA_AV sets. It increments OP_COUNTER. And it clears the command_pending flag.

The command is initiated by myfirst_sim_start_command, called from the driver when it writes CTRL.GO:

```
void
myfirst_sim_start_command(struct myfirst_softc *sc)
{
    struct myfirst_sim *sim = sc->sim;
    uint32_t data_in, delay_ms;

    MYFIRST_LOCK_ASSERT(sc);
```



```

    if (!sim->running)
        return;

    if (sim->command_pending) {
        /* Command already in flight. Real devices might reject
         * this, set an error bit, or queue the command. For the
         * simulation, the simplest behaviour is to treat it as
         * a no-op. The driver should not do this. */
        device_printf(sc->dev,
            "sim: overlapping command; ignored\n");
        return;
    }

    /* Snapshot DATA_IN now; the callout fires later. */
    data_in = CSR_READ_4(sc, MYFIRST_REG_DATA_IN);
    sim->pending_data = data_in;

    /* Set STATUS.BUSY, clear STATUS.DATA_AV. */
    {
        uint32_t status = CSR_READ_4(sc, MYFIRST_REG_STATUS);
        status |= MYFIRST_STATUS_BUSY;
        status &= ~MYFIRST_STATUS_DATA_AV;
        CSR_WRITE_4(sc, MYFIRST_REG_STATUS, status);
    }

    /* Read the configured delay. */
    delay_ms = CSR_READ_4(sc, MYFIRST_REG_DELAY_MS);
    if (delay_ms == 0) {
        /* Zero delay: complete on the next tick. */
        delay_ms = 1;
    }

    sim->command_pending = true;

    callout_reset_sbt(&sim->command_callout,
        delay_ms * SBT_1MS, 0,
        myfirst_sim_command_cb, sc, 0);
}

```

The function snapshots `DATA_IN` into the simulation state, sets `STATUS.BUSY`, clears `STATUS.DATA_AV`, reads the configured `DELAY_MS`, and schedules the command callout. The command callout fires `delay_ms` milliseconds later and finishes the command.

A subtle point: why snapshot `DATA_IN` instead of reading it in the command callout? Because the driver might write `DATA_IN` again before the callout fires. The simulation should use the value that was in `DATA_IN` when

CTRL.G0 was written, not whatever happens to be there when the callout runs. Snapshotting at start time is the correct behaviour for most real devices too: the device reads the command register when the command is issued and does not re-read it during execution.

17.4.9 Intercepting CTRL.G0

The `myfirst_sim_start_command` function is called when the driver writes CTRL.G0. The write intercept lives in the existing `myfirst_ctrl_update` helper (introduced in Chapter 16):

```
void
myfirst_ctrl_update(struct myfirst_softc *sc, uint32_t old, uint32_t new)
{
    /* Chapter 16 behaviour: log ENABLE transitions. */
    if ((old & MYFIRST_CTRL_ENABLE) != (new & MYFIRST_CTRL_ENABLE)) {
        device_printf(sc->dev, "CTRL.ENABLE now %s\n",
            (new & MYFIRST_CTRL_ENABLE) ? "on" : "off");
    }

    /* Chapter 17: if G0 was set, start a command in the simulation. */
    if (!(old & MYFIRST_CTRL_G0) && (new & MYFIRST_CTRL_G0)) {
        myfirst_sim_start_command(sc);
        /* Clear the G0 bit; it is a one-shot trigger. */
        CSR_UPDATE_4(sc, MYFIRST_REG_CTRL, MYFIRST_CTRL_G0, 0);
    }
}
```

The extension watches for the G0 bit transitioning from 0 to 1. When it does, it calls `myfirst_sim_start_command` and immediately clears the G0 bit, because G0 is a one-shot trigger: it asserts, the command starts, the bit goes back to zero. This is a common pattern for “start” bits in real hardware; the hardware clears the bit automatically once the command has been accepted.

The self-clearing behaviour means the driver does not have to remember to clear the bit itself. It writes `CSR_UPDATE_4(sc, MYFIRST_REG_CTRL, 0, MYFIRST_CTRL_G0)` to start a command, and the simulation handles the rest.

17.4.10 A Second Test: The Command Path

With the command-path code in place, the simulation now supports a command cycle. Try it:

```
# sysctl dev.myfirst.0.reg_status
dev.myfirst.0.reg_status: 1

# sysctl dev.myfirst.0.reg_ctrl_set_datain=0x12345678 # (new sysctl for DATA_IN)
# sysctl dev.myfirst.0.reg_ctrl_set=0x200             # bit 9 = CTRL.G0
```

```
# sysctl dev.myfirst.0.reg_status
dev.myfirst.0.reg_status: 3      # BUSY | READY

# sleep 1
# sysctl dev.myfirst.0.reg_status
dev.myfirst.0.reg_status: 9      # READY | DATA_AV

# sysctl dev.myfirst.0.reg_data_out
dev.myfirst.0.reg_data_out: 305419896  # 0x12345678

# sysctl dev.myfirst.0.reg_op_counter
dev.myfirst.0.reg_op_counter: 1
```

Reading the status right after `CTRL.GO` shows `BUSY|READY`. Waiting for the 500 ms delay and reading again shows `READY|DATA_AV`. Reading `DATA_OUT` returns the value that was in `DATA_IN` when `CTRL.GO` was written. `OP_COUNTER` has been incremented by one.

A new `sysctl reg_ctrl_set_datain` was implied in the example; it is a trivial addition that mirrors `reg_ctrl_set` but writes `DATA_IN` instead of `CTRL`. The pattern is identical to the Chapter 16 `reg_ctrl_set`: an `SYSCTL_ADD_PROC` with a writeable handler. Add one `sysctl` per register the user may want to poke directly.

17.4.11 Protecting Against the Overlapping Command

The check for `command_pending` in `myfirst_sim_start_command` prevents overlapping commands. If the driver writes `CTRL.GO` while a command is already in flight, the simulation ignores the second command and logs a warning. This is not realistic for every real device (some queue commands, some reject them with an error), but it is the simplest correct behaviour for a teaching simulation.

The driver, for its part, should not issue a command while one is in flight. The driver can poll `STATUS.BUSY` to see whether the device is ready to accept a new command, and wait until `BUSY` clears before writing `CTRL.GO`. Section 5 teaches this pattern in the driver.

17.4.12 Sysctls for the Simulation Controls

The simulation registers are already exposed through the Chapter 16 register sysctls (`reg_sensor`, `reg_sensor_config`, `reg_delay_ms`, `reg_fault_mask`, `reg_fault_prob`, `reg_op_counter`). Chapter 17 adds a few top-level sysctls that are not register-mapped:

```
void
myfirst_sim_add_sysctls(struct myfirst_softc *sc)
{
    struct sysctl_ctx_list *ctx = &sc->sysctl_ctx;
```

```

struct sysctl_oid *tree = sc->sysctl_tree;

SYSCTL_ADD_BOOL(ctx, SYSCTL_CHILDREN(tree), OID_AUTO,
    "sim_running", CTLFLAG_RD,
    &sc->sim->running, 0,
    "Whether the simulation callouts are active");

SYSCTL_ADD_UINT(ctx, SYSCTL_CHILDREN(tree), OID_AUTO,
    "sim_sensor_baseline", CTLFLAG_RW,
    &sc->sim->sensor_baseline, 0,
    "Baseline value around which SENSOR oscillates");

SYSCTL_ADD_UINT(ctx, SYSCTL_CHILDREN(tree), OID_AUTO,
    "sim_op_counter_mirror", CTLFLAG_RD,
    &sc->sim->op_counter, 0,
    "Mirror of the OP_COUNTER value (for observability)");

/* Add writeable sysctls for the register-mapped fields. */
SYSCTL_ADD_PROC(ctx, SYSCTL_CHILDREN(tree), OID_AUTO,
    "reg_delay_ms_set",
    CTLTYPE_UINT | CTLFLAG_RW | CTLFLAG_MPSAFE,
    sc, MYFIRST_REG_DELAY_MS, myfirst_sysctl_reg_write,
    "IU", "Command delay in milliseconds (writeable)");

SYSCTL_ADD_PROC(ctx, SYSCTL_CHILDREN(tree), OID_AUTO,
    "reg_sensor_config_set",
    CTLTYPE_UINT | CTLFLAG_RW | CTLFLAG_MPSAFE,
    sc, MYFIRST_REG_SENSOR_CONFIG, myfirst_sysctl_reg_write,
    "IU", "Sensor config: interval|amplitude (writeable)");
}

```

The top-level `sim_running`, `sim_sensor_baseline`, and `sim_op_counter_mirror` expose internal state that is not register-mapped. The rest add writeable sysctls for the Chapter 17 registers, using the existing `myfirst_sysctl_reg_write` handler from Chapter 16.

The addition keeps the user's view consistent: every interesting piece of state is visible through `sysctl dev.myfirst.0`, and the writeable sysctls follow the `_set` suffix pattern the driver already uses.

17.4.13 What Stage 1 Accomplished

At the end of Section 3, the driver has:

- A new file `myfirst_sim.c` with the simulation backend.
- A new header `myfirst_sim.h` with the simulation API.
- A new simulation state structure pointed at by `sc->sim`.

- Two callouts: one for periodic sensor updates, one for one-shot command completion.
- An extension to `myfirst_ctrl_update` that intercepts `CTRL.GO`.
- Writeable sysctls for the new registers that should be configurable.

The version tag becomes `1.0-sim-stage1`. The driver still does everything Chapter 16 taught; it now also has two callouts that produce autonomous register behaviour.

Build, load, and test:

```
# cd examples/part-04/ch17-simulating-hardware/stage1-backend
# make clean && make
# kldload ./myfirst.ko
# sysctl dev.myfirst.0.sim_running
# sysctl dev.myfirst.0.reg_sensor
# sleep 1
# sysctl dev.myfirst.0.reg_sensor
# sysctl dev.myfirst.0.reg_op_counter
# kldunload myfirst
```

You should see `sim_running` is 1, the sensor value changes, and `op_counter` stays at zero until you issue a command. If `kldunload` returns cleanly, the detach path is correct.

17.4.14 What Stage 1 Is Not

Stage 1 has the simulation running, but the driver's syscall paths do not yet know about the new behaviour. A `write(2)` to `/dev/myfirst0` still writes into the ring buffer, not into the simulated device. The bytes that go into the ring buffer never appear in `DATA_IN`, never cause a command, never appear in `DATA_OUT`. The simulation is ready; the driver is not yet using it for real.

Section 4 takes the next step: it connects the simulation to the driver's data path, so a user-space write causes a simulated command, and the command's result is what the user-space read sees. Until then, the simulation is a side show.

17.4.15 Wrapping Up Section 3

A simulated device's behaviour comes from a small set of primitives. A callout for periodic updates, a callout for one-shot timed events, and a shared state structure that both callouts consult. Chapter 17's simulation uses three of these: a sensor callout that runs always, a command callout that runs per-command, and an internal `command_pending` flag that prevents overlapping commands.

The simulation lives in its own file, `myfirst_sim.c`, which depends on the Chapter 16 `myfirst_hw.h` for register definitions. The split keeps the simulation code isolated; a later chapter that replaces the simulation with real PCI-facing code can delete `myfirst_sim.c` and replace it with `myfirst_pci.c` without touching the driver.

Section 4 begins the real integration: the driver's data path starts sending and receiving through the simulation, and the simulation starts looking like a proper device from the driver's side.

17.5 Section 4: Exposing the Fake Device to the Driver

Section 3 built a simulation backend that runs on callouts and writes to registers through the Chapter 16 accessors. Section 4 asks a different question: how does the driver see the simulation? From the driver's side, what is the access path? What does the code look like, and how does it differ from code that would talk to real hardware?

The short answer is that the access path does not change. The whole point of the Chapter 16 `bus_space(9)` abstraction is that the driver does not know, and does not care, whether the register block is real or simulated. Section 4 is deliberately a shorter section because the heavy lifting was done in Chapter 16. What remains is a careful look at how the abstraction survives the addition of dynamic behaviour, and at the few small hooks the driver needs to let the simulation drive its own behaviour.

17.5.1 The Tag and the Handle, Still

Chapter 16 set up the simulation's `bus_space_tag_t` and `bus_space_handle_t` in `myfirst_hw_attach`:

```
#if defined(__amd64__) || defined(__i386__)
    hw->regs_tag = X86_BUS_SPACE_MEM;
#else
    #error "Chapter 16 simulation supports x86 only"
#endif

hw->regs_handle = (bus_space_handle_t)(uintptr_t)hw->regs_buf;
```

That code does not change for Chapter 17. The simulation writes registers through `CSR_WRITE_4`, which expands to `myfirst_reg_write`, which expands to `bus_space_write_4(hw->regs_tag, hw->regs_handle, offset, value)`. The same tag; the same handle; the same accessor. The callouts that Section 3 introduced use the exact same API as the syscall path.

This is not an accident. The reason `bus_space` exists is precisely to hide the difference between real and simulated register access. A driver that uses `bus_space` correctly can be pointed at a real PCI BAR or at a `malloc(9)` allocation without any changes to the access code. The simulation exercises this property.

A useful exercise at this point: open `/usr/src/sys/dev/ale/if_ale.c` in one window and `myfirst_hw.c` in another. Compare the register access code. The ALE driver uses `CSR_READ_4(sc, ALE_REG_XYZ)`; your driver uses `CSR_READ_4(sc, MYFIRST_REG_XYZ)`. The expansion is identical. The only difference is the tag and handle: ALE gets them from `rman_get_bustag(sc->ale_res[0])` and `rman_get_bushandle(sc->ale_res[0])`; your driver gets them from the Chapter 16 simulation setup. Replacing the simulation with real hardware, which is what Chapter 18 will do, is a one-function change in `myfirst_hw_attach`.

17.5.2 Why the Abstraction Matters More Now

In Chapter 16, the argument for using `bus_space` was forward-looking: “when you eventually switch to real hardware, the driver code will not have to change”. In Chapter 17, the argument becomes concrete: the register block now changes on its own, and the driver has no way to know whether a change came from a simulation callout or from a real device's internal state machine. The abstraction is doing real work.

Consider a small example. A driver function that polls `STATUS.DATA_AV`:

```

static int
myfirst_wait_for_data(struct myfirst_softc *sc, int timeout_ms)
{
    int i;

    MYFIRST_LOCK_ASSERT(sc);

    for (i = 0; i < timeout_ms; i++) {
        uint32_t status = CSR_READ_4(sc, MYFIRST_REG_STATUS);
        if (status & MYFIRST_STATUS_DATA_AV)
            return (0);

        /* Release the lock, sleep briefly, reacquire. */
        MYFIRST_UNLOCK(sc);
        pause_sbt("mfwait", SBT_1MS, 0, 0);
        MYFIRST_LOCK(sc);
    }

    return (ETIMEDOUT);
}

```

This function reads STATUS, checks a bit, and either returns or sleeps for a millisecond and retries. It does not know whether the bit is being set by the Chapter 17 simulation’s command callout, by a real device’s internal state machine, or by something else. It just polls.

The `pause_sbt` call is worth examining. The function releases the driver lock before sleeping and reacquires it afterwards. Sleeping with a lock held would block every other context that needs the lock, including the simulation callouts; dropping the lock before sleeping lets the simulation run. `pause_sbt` takes a sleep identifier (“mfwait”), a signed binary time (1 ms here), a precision, and flags. The precision of zero means the kernel can coalesce this sleep with other sleeps, reducing timer interrupts.

Section 6 revisits the choice between `DELAY(9)`, `pause_sbt(9)`, and `callout_reset_sbt(9)` in more depth. The short version: polling a register a thousand times per second while sleeping between each poll is not the most efficient pattern, but it is correct, simple, and readable. Chapter 17’s driver uses it because the protocol is small and the polling overhead does not matter. Real high-performance drivers use interrupts (Chapter 19) instead.

17.5.3 The Command-Path Integration

The driver’s current data path (`myfirst_write`, from Chapter 10) writes bytes into the ring buffer. Chapter 17 wants to route those bytes through the simulation: user writes a byte, the driver writes it into `DATA_IN`, sets `CTRL.GO`, waits for `STATUS.DATA_AV`, reads `DATA_OUT`, writes the result into the ring buffer. This is the full command cycle.

For Stage 2 of the Chapter 17 driver, we add the command-cycle code path. The write function becomes:

```
static int
myfirst_write_cmd(struct myfirst_softc *sc, uint8_t byte)
{
    int error;

    MYFIRST_LOCK_ASSERT(sc);

    /* Wait for the device to be ready for a new command. */
    error = myfirst_wait_for_ready(sc, 100);
    if (error != 0)
        return (error);

    /* Write the byte to DATA_IN. */
    CSR_WRITE_4(sc, MYFIRST_REG_DATA_IN, (uint32_t)byte);

    /* Issue the command. */
    CSR_UPDATE_4(sc, MYFIRST_REG_CTRL, 0, MYFIRST_CTRL_GO);

    /* Wait for the command to complete (STATUS.DATA_AV). */
    error = myfirst_wait_for_data(sc, 2000);
    if (error != 0)
        return (error);

    /* Check for errors. */
    {
        uint32_t status = CSR_READ_4(sc, MYFIRST_REG_STATUS);
        if (status & MYFIRST_STATUS_ERROR) {
            /* Clear the error latch. */
            CSR_UPDATE_4(sc, MYFIRST_REG_STATUS,
                MYFIRST_STATUS_ERROR, 0);
            return (EIO);
        }
    }

    /* Read the result. DATA_OUT will hold the byte the simulation
     * echoed (or, if loopback is active, the same byte we wrote). */
    (void)CSR_READ_4(sc, MYFIRST_REG_DATA_OUT);

    /* Clear DATA_AV so the next command can set it. */
    CSR_UPDATE_4(sc, MYFIRST_REG_STATUS, MYFIRST_STATUS_DATA_AV, 0);

    return (0);
}
```


Six steps. Wait for ready, write DATA_IN, set CTRL.GO, wait for DATA_AV, check for errors, read DATA_OUT. Clear DATA_AV to be polite to the next iteration.

The `myfirst_wait_for_ready` helper polls `STATUS.BUSY`:

```
static int
myfirst_wait_for_ready(struct myfirst_softc *sc, int timeout_ms)
{
    int i;

    MYFIRST_LOCK_ASSERT(sc);

    for (i = 0; i < timeout_ms; i++) {
        uint32_t status = CSR_READ_4(sc, MYFIRST_REG_STATUS);
        if (!(status & MYFIRST_STATUS_BUSY))
            return (0);
        MYFIRST_UNLOCK(sc);
        pause_sbt("mfrdy", SBT_1MS, 0, 0);
        MYFIRST_LOCK(sc);
    }

    return (ETIMEDOUT);
}
```

Same structure as `myfirst_wait_for_data`; only the bit being polled differs. Both helpers are reasonable candidates for a shared implementation:

```
static int
myfirst_wait_for_bit(struct myfirst_softc *sc, uint32_t mask,
    uint32_t target, int timeout_ms)
{
    int i;

    MYFIRST_LOCK_ASSERT(sc);

    for (i = 0; i < timeout_ms; i++) {
        uint32_t status = CSR_READ_4(sc, MYFIRST_REG_STATUS);
        if ((status & mask) == target)
            return (0);
        MYFIRST_UNLOCK(sc);
        pause_sbt("mfwait", SBT_1MS, 0, 0);
        MYFIRST_LOCK(sc);
    }

    return (ETIMEDOUT);
}
```

```
}
```

Then `wait_for_ready` calls `wait_for_bit(sc, MYFIRST_STATUS_BUSY, 0, timeout_ms)` and `wait_for_data` calls `wait_for_bit(sc, MYFIRST_STATUS_DATA_AV, MYFIRST_STATUS_DATA_AV, timeout_ms)`. The abstraction is worth extracting once you have two users; Section 6 revisits this pattern.

17.5.4 Where the Command Integrates with the Write Syscall

The existing Chapter 10 `myfirst_write` pushes bytes into the ring buffer. Chapter 17 does not replace that path; it augments it. For Stage 2, we add a per-byte command-cycle hook that sends each byte through the simulation in addition to pushing it into the ring.

```
static int
myfirst_write(struct cdev *dev, struct uio *uio, int ioflag)
{
    struct myfirst_softc *sc = dev->si_drv1;
    uint8_t buf[64];
    size_t n;
    int error;

    /* ... existing validation ... */

    while (uio->uio_resid > 0) {
        n = MIN(uio->uio_resid, sizeof(buf));
        error = uiomove(buf, n, uio);
        if (error != 0)
            return (error);

        MYFIRST_LOCK(sc);
        for (size_t i = 0; i < n; i++) {
            /* Send each byte through the simulated device. */
            error = myfirst_write_cmd(sc, buf[i]);
            if (error != 0) {
                MYFIRST_UNLOCK(sc);
                return (error);
            }

            /* Existing Chapter 10 path: push into ring. */
            error = cbuf_put(sc->cbuf, buf[i]);
            if (error != 0) {
                MYFIRST_UNLOCK(sc);
                return (error);
            }
        }
    }
}
```

```
        MYFIRST_UNLOCK(sc);  
    }  
  
    return (0);  
}
```

The loop is one-byte-at-a-time for clarity. Each byte goes through the command cycle, then into the ring. A real driver would usually pipeline this (write several bytes to a FIFO, issue a single command, read several bytes back), but Chapter 17's teaching point is about single commands; pipelining is a refinement for later.

Note the locking. The outer lock covers the entire batch (up to 64 bytes). Each `myfirst_write_cmd` call releases and reacquires the lock internally when it sleeps. Other contexts that need the lock (notably the simulation callouts) get a chance to run between commands.

17.5.5 Stage 2 Test

Build, load, and test:

```
# kldload ./myfirst.ko  
# echo -n "hi" > /dev/myfirst0  
# sysctl dev.myfirst.0.reg_op_counter  
dev.myfirst.0.reg_op_counter: 2  
  
# dd if=/dev/myfirst0 bs=1 count=2 2>/dev/null  
hi
```

Two bytes written; `OP_COUNTER` incremented by two. The reads return the same two bytes because the simulation echoes `DATA_IN` to `DATA_OUT`. The test validates the end-to-end flow: user space writes, driver commands, simulation processes, driver reads, user space receives.

The whole cycle takes about 1 second (500 ms per byte, two bytes). This is slow, but it is realistic: a real device with similar latency (for example, a SPI-attached sensor that takes 500 ms to produce a reading) would feel exactly like this. Speeding it up means reducing `DELAY_MS` (try `sysctl dev.myfirst.0.reg_delay_ms_set=10` for 10 ms per command).

17.5.6 What Does Not Change

A few properties of the Chapter 16 driver that do not change in Stage 2 are worth enumerating.

The access log continues to work. Every register read and write the driver performs shows up in the log. The simulation's own register writes also show up (because they go through the same accessors). The log is dense, which is good: you can see the simulation's heartbeat alongside the driver's activity.

The `sysctl` interface continues to work. Every register is readable; every writeable register is writeable. The Chapter 16 conveniences (read-every-register, write-specific-registers) are unchanged.

The locking discipline continues to work. Every register access happens under `sc->mtx`; the simulation callouts were designed to run with `sc->mtx` held (via `callout_init_mtx`), so there are no lock-order inversions to worry about.

The detach path continues to work. The simulation detach was added to the existing detach ordering, and the draining of the simulation callouts happens before the hardware region is freed.

17.5.7 What Still Needs Attention

A few loose ends that Sections 5 and 6 will tighten.

The timeouts in `myfirst_wait_for_ready` and `myfirst_wait_for_data` are hardcoded (100 ms and 2000 ms respectively). A configurable timeout (perhaps exposed through a `sysctl`) would be better; Section 5 adds one.

The `pause_sbt` sleep of 1 ms is a reasonable default but not optimal for every situation. On a fast simulation (e.g., `DELAY_MS=0`), polling every 1 ms is wasteful; on a slow simulation (e.g., `DELAY_MS=5000`), polling every 1 ms is overkill. Section 6 discusses better strategies.

The command-cycle write path does not interact with the simulation's fault injection. If the simulation is configured to fail a command (Section 7's work), the driver needs to recognise and recover from the failure. Section 5 adds the basic error-recovery; Section 7 adds the fault-injection infrastructure.

The driver's read path is not yet integrated. Reads currently pull from the ring buffer, which contains whatever was pushed there by write commands. A more realistic pattern would have reads also cause commands (a "please give me another byte" command), but that is a Section 5 topic.

17.5.8 Wrapping Up Section 4

Exposing the simulated device to the driver is not a new mechanism in Chapter 17. The Chapter 16 `bus_space(9)` abstraction already handles this; the simulation's writes and the driver's reads go through the same tag and handle, and the driver neither knows nor cares that the register block is simulated. What Stage 2 adds is the command-cycle code path: the driver writes `DATA_IN`, sets `CTRL.GO`, waits for `DATA_AV`, reads `DATA_OUT`. Two helper functions (`myfirst_wait_for_ready` and `myfirst_wait_for_data`) make the polling explicit; one umbrella function (`myfirst_wait_for_bit`) factors the shared structure. The driver now exercises the simulation in a realistic command-and-response pattern.

Section 5 deepens the testing story. It adds error-recovery, configurable timeouts, a clearer separation between the read and the write paths, and the first load tests that hit the simulation with enough traffic to expose race conditions.

17.6 Section 5: Testing Device Behavior from the Driver

The command-cycle code from Section 4 works for a single byte. It times out correctly when the simulation is slow; it returns a data value when the simulation responds; it reads `STATUS` at the appropriate points. What it has not yet experienced is volume, diversity, or concurrency. A driver that passes a one-byte test may still fail under load, under concurrent writers, or when the simulation is configured with unusual parameters.

Section 5 is about stress-testing the driver's interaction with the simulated device. It adds configurable timeouts, a more robust error-recovery path, a read-path integration that lets the simulation drive reads as well as writes, and a set of load tests that exercise the driver across its full behavioural surface. By the end of Section 5, the driver is at Stage 2 of Chapter 17 and has survived a meaningful stress run.

17.6.1 Configurable Timeouts

Hardcoded timeouts are brittle. 100 ms is enough for the default DELAY_MS of 500, but what if the user wants to test a slow configuration? Bumping DELAY_MS to 2000 would mean every command times out with the 100 ms limit, which is a false failure mode. Make the timeouts configurable.

Add two fields to the softc (or to a suitable sub-struct):

```
struct myfirst_softc {
    /* ... existing fields ... */
    int      cmd_timeout_ms;      /* max wait for command completion */
    int      rdy_timeout_ms;      /* max wait for device ready      */
};
```

Initialise them in attach:

```
sc->cmd_timeout_ms = 2000;      /* 2 s default      */
sc->rdy_timeout_ms = 100;      /* 100 ms default    */
```

Expose them through sysctl:

```
SYCTL_ADD_INT(ctx, SYSCTL_CHILDREN(tree), OID_AUTO,
    "cmd_timeout_ms", CTLFLAG_RW, &sc->cmd_timeout_ms, 0,
    "Command completion timeout in milliseconds");

SYCTL_ADD_INT(ctx, SYSCTL_CHILDREN(tree), OID_AUTO,
    "rdy_timeout_ms", CTLFLAG_RW, &sc->rdy_timeout_ms, 0,
    "Device-ready polling timeout in milliseconds");
```

And change myfirst_write_cmd to use them:

```
error = myfirst_wait_for_ready(sc, sc->rdy_timeout_ms);
/* ... */
error = myfirst_wait_for_data(sc, sc->cmd_timeout_ms);
```

The test suite can now adjust the timeouts to match the test's expected latency. A fast-simulation test sets them low; a slow-simulation test sets them high. A pathological test sets them very low and expects timeout errors.

17.6.2 Read-Path Integration

The Chapter 10 `myfirst_read` pulls from the ring buffer. The ring buffer contains whatever previous writes pushed in. Chapter 17 can extend this pattern: when the ring is empty and the reader is willing to wait, issue a “generate a byte” command to the simulated device and push the result into the ring.

This is not how every driver works. A real device has data-production semantics that are device-specific; a network card produces bytes as packets arrive, not on demand. For a simulated sensor-like device, a “sample” command that produces one reading per invocation is a natural pattern. Our driver adopts it.

Add a `myfirst_sample_cmd` function that issues a command and pushes the result into the ring:

```
static int
myfirst_sample_cmd(struct myfirst_softc *sc)
{
    uint32_t data_out;
    int error;

    MYFIRST_LOCK_ASSERT(sc);

    error = myfirst_wait_for_ready(sc, sc->rdy_timeout_ms);
    if (error != 0)
        return (error);

    /* DATA_IN does not matter for a sample; write a marker. */
    CSR_WRITE_4(sc, MYFIRST_REG_DATA_IN, 0xCAFE);

    CSR_UPDATE_4(sc, MYFIRST_REG_CTRL, 0, MYFIRST_CTRL_GO);

    error = myfirst_wait_for_data(sc, sc->cmd_timeout_ms);
    if (error != 0)
        return (error);

    data_out = CSR_READ_4(sc, MYFIRST_REG_DATA_OUT);
    CSR_UPDATE_4(sc, MYFIRST_REG_STATUS, MYFIRST_STATUS_DATA_AV, 0);

    /* Push one byte of the sample into the ring. */
    error = cbuf_put(sc->cbuf, (uint8_t)(data_out & 0xFF));
    return (error);
}
```

The read path can now call `myfirst_sample_cmd` when the ring is empty. Wire it into `myfirst_read`:

```
static int
myfirst_read(struct cdev *dev, struct uio *uio, int ioflag)
{
```

```

    struct myfirst_softc *sc = dev->si_drv1;
    uint8_t byte;
    int error;

    MYFIRST_LOCK(sc);
    while (uio->uio_resid > 0) {
        while (cbuf_empty(sc->cbuf)) {
            if (ioflag & O_NONBLOCK) {
                MYFIRST_UNLOCK(sc);
                return (EWOULDBLOCK);
            }

            /* Chapter 17: trigger a sample when ring is empty. */
            error = myfirst_sample_cmd(sc);
            if (error != 0) {
                MYFIRST_UNLOCK(sc);
                return (error);
            }
        }
        error = cbuf_get(sc->cbuf, &byte);
        if (error != 0) {
            MYFIRST_UNLOCK(sc);
            return (error);
        }
        MYFIRST_UNLOCK(sc);
        error = uiomove(&byte, 1, uio);
        MYFIRST_LOCK(sc);
        if (error != 0) {
            MYFIRST_UNLOCK(sc);
            return (error);
        }
    }
    MYFIRST_UNLOCK(sc);
    return (0);
}

```

Now a read from `/dev/myfirst0` triggers a simulated command if the ring is empty, effectively pulling one byte per command from the simulation. Combined with the write path from Section 4 (push-style), the driver now exercises both directions.

17.6.3 A First Load Test

With both paths integrated, the driver is ready for load testing. A simple test script:

```
#!/bin/sh
# cmd_load.sh: exercise the command path under load.

set -e

# Fast simulation for load testing.
sysctl dev.myfirst.0.reg_delay_ms_set=10

# Spawn 4 writers and 4 readers in parallel.
for i in 1 2 3 4; do
    (for j in $(seq 1 100); do echo -n "X" > /dev/myfirst0; done) &
done
for i in 1 2 3 4; do
    (dd if=/dev/myfirst0 of=/dev/null bs=1 count=100 2>/dev/null) &
done
wait

# Show the resulting counter.
sysctl dev.myfirst.0.reg_op_counter

# Reset to default.
sysctl dev.myfirst.0.reg_delay_ms_set=500
```

With `DELAY_MS=10` and 4 writers each sending 100 bytes plus 4 readers each receiving 100 bytes, the test should complete in about 10 seconds (800 commands at 10 ms each, serialised through the simulation). `OP_COUNTER` should reach at least 800.

Two things are worth noticing about this test.

First, the commands are serialised. The simulation rejects overlapping commands (Section 3's `command_pending` check). The driver waits for `STATUS.BUSY` to clear before issuing the next command. With multiple user-space processes contending for the driver, the lock on `sc->mtx` serialises them. The effective rate is one command per `DELAY_MS` milliseconds, regardless of how many writers are running.

Second, the driver did not change to accommodate concurrency. The Chapter 11 mutex, the Chapter 14 tasks, the Chapter 16 accessors, and the Section 4 command-cycle code all compose naturally. The discipline Part 3 built up pays off here: no additional synchronisation is needed to add simulated hardware to a driver that already coordinates itself properly.

17.6.4 Error-Recovery Refinement

The command-cycle code in Section 4 clears `STATUS.ERROR` and returns `EIO`. Section 5 refines this to distinguish several error cases.


```

static int
myfirst_write_cmd(struct myfirst_softc *sc, uint8_t byte)
{
    uint32_t status;
    int error;

    MYFIRST_LOCK_ASSERT(sc);

    error = myfirst_wait_for_ready(sc, sc->rdy_timeout_ms);
    if (error != 0) {
        sc->stats.cmd_rdy_timeouts++;
        return (error);
    }

    CSR_WRITE_4(sc, MYFIRST_REG_DATA_IN, (uint32_t)byte);
    CSR_UPDATE_4(sc, MYFIRST_REG_CTRL, 0, MYFIRST_CTRL_GO);

    error = myfirst_wait_for_data(sc, sc->cmd_timeout_ms);
    if (error != 0) {
        sc->stats.cmd_data_timeouts++;
        /* Clear any partial state in the simulation. */
        myfirst_recover_from_stuck(sc);
        return (error);
    }

    status = CSR_READ_4(sc, MYFIRST_REG_STATUS);
    if (status & MYFIRST_STATUS_ERROR) {
        CSR_UPDATE_4(sc, MYFIRST_REG_STATUS,
            MYFIRST_STATUS_ERROR, 0);
        CSR_UPDATE_4(sc, MYFIRST_REG_STATUS,
            MYFIRST_STATUS_DATA_AV, 0);
        sc->stats.cmd_errors++;
        return (EIO);
    }

    (void)CSR_READ_4(sc, MYFIRST_REG_DATA_OUT);
    CSR_UPDATE_4(sc, MYFIRST_REG_STATUS, MYFIRST_STATUS_DATA_AV, 0);
    sc->stats.cmd_successes++;
    return (0);
}

```

Three changes from Section 4.

First, per-error-type counters. `sc->stats` is a structure of `uint64_t` counters that the driver updates on each outcome. `cmd_successes`, `cmd_rdy_timeouts`, `cmd_data_timeouts`, and `cmd_errors` each get their own

counter. A sysctl exposes them. Under load, the counters become the primary diagnostic: “the driver issued 800 commands, 5 timed out, 0 had errors, 795 succeeded” is a much more useful summary than “the driver ran”.

Second, `myfirst_recover_from_stuck`. When a command times out, the simulation may still be in the middle of processing. Clearing stale state matters:

```
static void
myfirst_recover_from_stuck(struct myfirst_softc *sc)
{
    uint32_t status;

    MYFIRST_LOCK_ASSERT(sc);

    /* Clear any pending command flag in the simulation. The next
     * CTRL.GO will be accepted regardless of any stale state. */
    if (sc->sim != NULL) {
        sc->sim->command_pending = false;
        callout_stop(&sc->sim->command_callout);
    }

    /* Force-clear BUSY and DATA_AV. */
    status = CSR_READ_4(sc, MYFIRST_REG_STATUS);
    status &= ~(MYFIRST_STATUS_BUSY | MYFIRST_STATUS_DATA_AV);
    CSR_WRITE_4(sc, MYFIRST_REG_STATUS, status);
}
```

Note that `myfirst_recover_from_stuck` reaches into `sc->sim->command_pending` directly. This is a legitimate simulation-only path: on real hardware, there would be no such variable; instead, the driver would issue a reset command to the device. The function is a simulation-specific recovery path, and a comment in the code should say so. When Chapter 18 replaces the simulation with real hardware, this function changes entirely.

Third, status clearing on error. The error path clears both `ERROR` and `DATA_AV`. Leaving either one set would cause the next command’s `wait_for_data` to return immediately (if `DATA_AV` is still set) or observe an unexpected error (if `ERROR` is still set). The defensive cleanup ensures the simulation is in a clean state for the next command.

17.6.5 Stats Infrastructure

The `sc->stats` structure is worth defining properly. Place it in `myfirst.h`:

```
struct myfirst_stats {
    uint64_t      cmd_successes;
    uint64_t      cmd_rdy_timeouts;
    uint64_t      cmd_data_timeouts;
```

```
uint64_t      cmd_errors;  
uint64_t      cmd_rejected;  
uint64_t      cmd_recoveries;  
uint64_t      samples_taken;  
uint64_t      fault_injected;  
};
```

Add it to the softc:

```
struct myfirst_softc {  
    /* ... existing fields ... */  
    struct myfirst_stats stats;  
};
```

Expose each counter through sysctl:

```
static void  
myfirst_add_stats_sysctls(struct myfirst_softc *sc)  
{  
    struct sysctl_ctx_list *ctx = &sc->sysctl_ctx;  
    struct sysctl_oid *tree = sc->sysctl_tree;  
  
    SYSCTL_ADD_U64(ctx, SYSCTL_CHILDREN(tree), OID_AUTO,  
        "cmd_successes", CTLFLAG_RD,  
        &sc->stats.cmd_successes, 0,  
        "Successfully completed commands");  
  
    SYSCTL_ADD_U64(ctx, SYSCTL_CHILDREN(tree), OID_AUTO,  
        "cmd_rdy_timeouts", CTLFLAG_RD,  
        &sc->stats.cmd_rdy_timeouts, 0,  
        "Commands that timed out waiting for READY");  
  
    SYSCTL_ADD_U64(ctx, SYSCTL_CHILDREN(tree), OID_AUTO,  
        "cmd_data_timeouts", CTLFLAG_RD,  
        &sc->stats.cmd_data_timeouts, 0,  
        "Commands that timed out waiting for DATA_AV");  
  
    SYSCTL_ADD_U64(ctx, SYSCTL_CHILDREN(tree), OID_AUTO,  
        "cmd_errors", CTLFLAG_RD,  
        &sc->stats.cmd_errors, 0,  
        "Commands that completed with STATUS.ERROR set");  
  
    SYSCTL_ADD_U64(ctx, SYSCTL_CHILDREN(tree), OID_AUTO,
```

```

        "cmd_recoveries", CTLFLAG_RD,
        &sc->stats.cmd_recoveries, 0,
        "Recovery operations triggered");

        /* ... add the remaining counters ... */
}

```

Under load, `sysctl dev.myfirst.0 | grep -E 'cmd_|samples_|fault_'` gives a one-screen summary of the driver's command-path behaviour. After a test run, diffing before and after is how you answer “did the driver behave correctly?”.

17.6.6 Observing the Driver Under Load

A useful debugging pattern: watch the stats while a load test is running. In one terminal:

```
# sysctl dev.myfirst.0 | grep cmd_ > before.txt
```

In another terminal, start the load test. Wait for it to finish. Then:

```
# sysctl dev.myfirst.0 | grep cmd_ > after.txt
# diff before.txt after.txt
```

The diff shows exactly how many of each event occurred during the test window. Ideal runs show many successes and zero timeouts or errors. Runs with faults injected (Section 7) show nonzero counts for the expected error types; unexpected counts indicate a bug.

For continuous observation, use `watch`:

```
# watch -n 1 'sysctl dev.myfirst.0 | grep cmd_'
```

Every second, the current counters print. As the load test runs, the successes counter climbs, while the timeouts and errors counters (ideally) stay flat.

17.6.7 Hands-On Behaviour Test

A concrete test for Section 5. Set up:

```
# kldload ./myfirst.ko
# sysctl dev.myfirst.0.reg_delay_ms_set=50      # fast commands
# sysctl dev.myfirst.0.access_log_enabled=1
```

Run:

```
# echo -n "Hello, World!" > /dev/myfirst0
# dd if=/dev/myfirst0 bs=1 count=13 of=/dev/null 2>/dev/null
# sysctl dev.myfirst.0 | grep -E 'cmd_|op_counter|samples_'
```

Expected output:

```
dev.myfirst.0.cmd_successes: 26
dev.myfirst.0.cmd_rdy_timeouts: 0
dev.myfirst.0.cmd_data_timeouts: 0
dev.myfirst.0.cmd_errors: 0
dev.myfirst.0.samples_taken: 13
dev.myfirst.0.reg_op_counter: 26
```

Thirteen writes (one per byte of “Hello, World!”), thirteen reads (one per byte back out). Each write and each read is a simulation command, for a total of 26. All successful. All within the latency budget.

Now inspect the access log:

```
# sysctl dev.myfirst.0.access_log | head -20
```

The log shows the command cycle in detail. For each write command:

1. Read STATUS (polling for BUSY to clear)
2. Write DATA_IN (the byte)
3. Read/Write CTRL (setting GO)
4. Read STATUS (polling for DATA_AV to set)
5. Read DATA_OUT (the result)
6. Read/Write STATUS (clearing DATA_AV)

For each sample command, the same sequence plus a `cbuf_put`. The log is dense but readable, and it is where you would look if the driver misbehaved.

17.6.8 What Stage 2 Accomplished

Stage 2 integrates the simulation with the driver’s data path. Both the write and the read paths now exercise simulated commands. Configurable timeouts let tests target specific behaviour. Per-error-type stats counters expose the driver’s interaction with the simulation. A recovery path cleans up stuck state when a command times out. Load tests exercise the driver at volume and confirm the composition of Part 3 synchronisation with Chapter 17 simulation.

The version tag becomes `1.0-sim-stage2`. The driver is now interacting with a dynamic register block in a way that approaches real-hardware behaviour.

17.6.9 Wrapping Up Section 5

Testing the driver against the simulation is not a separate activity; it is a natural extension of the driver's command path. Add configurable timeouts, integrate simulated commands into both read and write, track per-outcome statistics, and run enough load to expose the common failure modes.

The hands-on test validates the round-trip: user-space input goes through the driver, through the simulation, and back to user space intact. The counters confirm no errors. The access log shows every register touch in order.

Section 6 deepens the timing discussion. The 1 ms `pause_sbt` in `myfirst_wait_for_bit` is a reasonable default, but not the only choice. `DELAY(9)` is better for very short waits; `callout_reset_sbt` is better for fire-and-forget scheduling. Understanding when each is appropriate is the topic of Section 6.

17.7 Section 6: Adding Timing and Delay

Timing is where real hardware gets interesting. A device that processes a command in 5 microseconds needs a very different driver pattern than one that processes a command in 500 milliseconds. A driver that uses the wrong timing primitive for the wrong time scale burns CPU on short waits or accumulates latency on long ones, and in the worst case it deadlocks under contention.

FreeBSD offers three primary timing primitives for driver code: `DELAY(9)`, `pause_sbt(9)`, and `callout_reset_sbt(9)`. Each is appropriate at a specific time scale, each has a specific cost, and each has a specific correctness requirement. Section 6 teaches them in order and shows how to pick the right one for each situation in the Chapter 17 simulation.

17.7.1 The Three Timing Primitives

The primitives differ along three axes: whether they busy-wait or sleep, whether they are cancelable, and what they cost.

`DELAY(n)` busy-waits for `n` microseconds. It does not sleep, does not yield the CPU, and does not release any locks. It is safe to call from any context, including from a spin mutex, an interrupt filter, or a callout. Its cost is the CPU time it consumes: 100 microseconds of `DELAY` is 100 microseconds the CPU cannot do anything else. Its precision is usually quite good (single-digit microseconds), limited by the CPU's time-stamp counter resolution. It is the right choice for very short waits where sleeping would be impractical.

`pause_sbt(wmesg, sbt, pr, flags)` puts the calling thread to sleep for `sbt` signed-binary-time units. While the thread is asleep, the CPU is free for other work, and any locks the thread does not hold are free for other contexts. If the thread has a lock held, `pause_sbt` does not release it; it is the caller's job to drop locks that should not be held during the sleep. The precision is on the order of the kernel's timer tick (1 ms on a typical system). It is the right choice for short-to-medium waits where the thread has nothing else to do. `pause_sbt` is not cancelable through any mechanism shorter than the sleep itself; once a thread has called it, it will sleep for at least the requested time (barring exceptional events).

`callout_reset_sbt(c, sbt, pr, fn, arg, flags)` schedules a callback to run `sbt` signed-binary-time units in the future. The calling thread does not wait; it returns immediately. The callback runs on a callout thread when the time expires. The callout can be cancelled with `callout_stop` or `callout_drain`. The precision is

similar to `pause_sbt`. It is the right choice for fire-and-forget scheduling, for timeouts that may be cancelled early, and for periodic work.

The three primitives compose. A driver might use `DELAY(10)` to wait 10 microseconds for a register bit to settle, `pause_sbt(..., SBT_1MS * 50, ...)` to sleep 50 ms waiting for a command to complete, and `callout_reset_sbt(..., SBT_1S * 5, ...)` to schedule a watchdog 5 seconds in the future. Each choice reflects the time scale and the context.

17.7.2 A Time-Scale Decision Table

A quick reference for choosing among the three primitives:

Wait Duration	Context	Preferred Primitive
< 10 us	Any	<code>DELAY(9)</code>
10-100 us	Non-interrupt, non-spin	<code>DELAY(9)</code> or <code>pause_sbt(9)</code>
100 us - 10 ms	Non-interrupt, non-spin	<code>pause_sbt(9)</code>
> 10 ms	Non-interrupt, non-spin	<code>pause_sbt(9)</code> or <code>callout</code>
Fire-and-forget	Any (callout state machine)	<code>callout_reset_sbt(9)</code>
Interrupt/spin	Any duration	<code>DELAY(9)</code> only

The table is a starting point, not a rigid rule. The real question is: can the caller afford to sleep? If yes, pause or callout. If no (spin lock held, interrupt handler, filter handler, critical section), `DELAY`.

17.7.3 Where the Chapter 17 Driver Already Uses Each

Section 4's polling loop uses `pause_sbt` with 1 ms delays. That is appropriate for the time scale: the simulation's default `DELAY_MS` is 500, so we expect the driver to sleep many times before the command completes. Sleeping lets other threads use the CPU while we wait. `DELAY(1000)` would also work but would waste CPU; the driver currently sits at about 500 polls per command with no useful CPU work between them, and `DELAY` would spin all 500 times.

Section 3's simulation callouts use `callout_reset_sbt` with variable intervals. That is appropriate because the simulation needs to do work in the future without any thread actively waiting. `pause_sbt` would require a dedicated kernel thread for each callout, which would be wasteful. `DELAY` would require the thread to stay in kernel mode for the full interval, which is unacceptable.

There is no `DELAY` in the driver yet. Section 6 introduces it where appropriate.

17.7.4 A Good Use of `DELAY`

Consider the recovery path from Section 5. After a timeout, the driver calls `myfirst_recover_from_stuck`, which clears some state and returns. The code works, but there is a subtle race: the simulation's command

callout might fire between the timeout detection and the recovery call, changing STATUS in unexpected ways. A small fix: use `DELAY(10)` to wait 10 microseconds for any in-flight callout to drain before clearing the state.

Except this is wrong. `DELAY(10)` does nothing useful here. The callout cannot run while we hold `sc->mtx` (because it was registered with `callout_init_mtx`), so there is no race. Adding a `DELAY` here would be cargo-culting: adding a delay because something might go wrong, without a clear model of what the delay is preventing.

A better example. Some real hardware requires a small delay between two register writes to let the device's internal logic settle. A datasheet might say: "After writing `CTRL.RESET`, wait at least 5 microseconds before writing any other register." The Chapter 17 simulation's `CTRL.RESET` bit is not actually implemented, but we can imagine it. If it were, the reset code would use `DELAY(5)`:

```
static void
myfirst_reset_device(struct myfirst_softc *sc)
{
    MYFIRST_LOCK_ASSERT(sc);

    /* Assert reset. */
    CSR_UPDATE_4(sc, MYFIRST_REG_CTRL, 0, MYFIRST_CTRL_RESET);

    /* Datasheet: wait 5 us for internal state to clear. */
    DELAY(5);

    /* Deassert reset. */
    CSR_UPDATE_4(sc, MYFIRST_REG_CTRL, MYFIRST_CTRL_RESET, 0);
}
```

Five microseconds is too short for `pause_sbt` to be efficient (the kernel's timer resolution is typically 1 ms, so `pause_sbt` for 5 us would actually sleep for the next timer tick, wasting most of a millisecond). `DELAY(5)` is the right primitive.

17.7.5 A Good Use of `pause_sbt` (With the Lock Dropped)

The polling loop from Section 4:

```
for (i = 0; i < timeout_ms; i++) {
    uint32_t status = CSR_READ_4(sc, MYFIRST_REG_STATUS);
    if ((status & mask) == target)
        return (0);
    MYFIRST_UNLOCK(sc);
    pause_sbt("mfwait", SBT_1MS, 0, 0);
    MYFIRST_LOCK(sc);
}
```


The lock is dropped before `pause_sbt` and reacquired after. This pattern is essential and deserves careful attention.

If the lock were held across `pause_sbt`, the simulation callout (which needs the lock) would be blocked for the duration of the sleep. Since the callout is the thing the driver is waiting on, this would deadlock. Dropping the lock lets the callout run, lets the simulation update `STATUS`, and lets the next iteration of the loop observe the new value.

The correctness requirement is that nothing the driver cares about can change in an unrecoverable way while the lock is dropped. In our case, the driver's loop variables are local (`i`, `status`, `mask`, `target`); they do not live in memory that other contexts touch. The state that other contexts touch (the register block) is exactly what we want them to update. So dropping the lock is safe.

A variation on this pattern uses a sleepable condition variable instead of polling:

```
MYFIRST_LOCK_ASSERT(sc);

while ((CSR_READ_4(sc, MYFIRST_REG_STATUS) & mask) != target) {
    int error = cv_timedwait_sbt(&sc->data_cv, &sc->mtx,
                               SBT_1MS * timeout_ms, 0, 0);
    if (error == EWOULDBLOCK)
        return (ETIMEDOUT);
}
return (0);
```

The condition variable waits until either the target bit changes (signalled by some other context) or the timeout expires. `cv_timedwait_sbt` drops the mutex automatically, sleeps, and reacquires the mutex before returning. No explicit `UNLOCK/LOCK` wrapping is needed.

For this to work, the simulation callout that changes `STATUS` must `cv_signal` the condition variable after the change. The code in `myfirst_sim_command_cb` becomes:

```
/* After setting DATA_AV: */
cv_broadcast(&sc->data_cv);
```

The driver's wait loop and the simulation's update cooperate through the condition variable: the driver sleeps until something interesting happens, and the simulation wakes it when something does. This is a much more efficient pattern than polling with `pause_sbt`. The driver sleeps for the whole wait; one wakeup is all that is needed; no thrashing.

Chapter 17 uses `pause_sbt` in the main polling loop for teaching simplicity (you can see exactly when the sleep happens), but the condition-variable pattern is mentioned here because it is closer to what production drivers do, especially for long waits.

17.7.6 A Good Use of `callout_reset_sbt`

The simulation callouts from Section 3 are the canonical example. The sensor callout fires every 100 ms, updates the sensor register, and re-arms. The command callout fires once after `DELAY_MS` milliseconds to complete a

command. Both use `callout_reset_sbt` with:

- `sbt` = the interval in SBT units
- `pr` = 0 (no precision hint; kernel may coalesce)
- `flags` = 0 (no special behaviour)

For a more precise schedule, `pr` can be set to bound the coalescing:

```
callout_reset_sbt(&sim->sensor_callout,
    interval_ms * SBT_1MS,
    SBT_1MS,                      /* precision: within 1 ms */
    myfirst_sim_sensor_cb, sc, 0);
```

With `pr = SBT_1MS`, the callout fires no more than 1 ms later than requested. For our simulation, precision is not critical; the default of 0 is fine.

For a callout that must run on a specific CPU, the `C_DIRECT_EXEC` flag arranges for the callback to run directly from the hardclock interrupt rather than being dispatched to a callout thread. This is higher performance but more restrictive: the callback cannot sleep or take sleep locks. The Chapter 17 simulation does not need this; it uses plain mutex-based callouts.

For a callout aligned with the hardclock boundary, the `C_HARDCLOCK` flag is appropriate. Our simulation does not need alignment either; the default is fine.

17.7.7 A Realistic Example: Simulating a Slow Device

Put the primitives together. Suppose the simulation is configured with `DELAY_MS=2000` (a slow device, two seconds per command). The driver issues a write command. What does each primitive do?

1. The driver writes `DATA_IN`, sets `CTRL.GO`, and calls `myfirst_wait_for_data(sc, 3000)` (3 s timeout).
2. `myfirst_wait_for_data` reads `STATUS`, finds `DATA_AV` not set, drops the lock, calls `pause_sbt("mfwait", SBT_1MS, 0, 0)`, and sleeps for ~1 ms.
3. Meanwhile, the simulation's `myfirst_sim_command_cb` is scheduled to fire in ~2000 ms. It is not running yet.
4. The driver's loop wakes, reacquires the lock, reads `STATUS` again, sees `DATA_AV` still not set, drops the lock, sleeps again.
5. Steps 2-4 repeat about 2000 times over the two seconds.
6. The simulation's callout fires. It reads `STATUS`, sets `DATA_AV`, writes the command-completion values.
7. On the next iteration of the driver's loop, the driver sees `DATA_AV` set and returns.

Total CPU time burnt by the driver waiting: minimal. Each `pause_sbt` is a sleep, not a spin. The CPU is free for other work during each of the 2000 sleeps.

Now suppose the driver had used `DELAY(1000)` instead of `pause_sbt(..., SBT_1MS, ...)`. Same time scale per iteration; completely different CPU behaviour. Each `DELAY(1000)` burns 1000 microseconds of CPU time. 2000 iterations = 2 seconds of pure CPU. The driver works, but it hogs one core entirely for the duration of the command. If other processes need CPU, they wait.

The difference illustrates why `pause_sbt` is the right choice for millisecond-scale waits: `sleep` is free; `spin` costs CPU.

Now suppose the driver had used `callout_reset_sbt` instead of polling. That is harder: `callout_reset_sbt` does not make the current thread wait. The thread would return to its caller immediately. The caller would need to know to sleep until the callout fires. This is exactly what condition variables do. The code becomes:

```
MYFIRST_LOCK_ASSERT(sc);

while (!(CSR_READ_4(sc, MYFIRST_REG_STATUS) & MYFIRST_STATUS_DATA_AV)) {
    int error = cv_timedwait_sbt(&sc->data_cv, &sc->mtx,
                                SBT_1MS * timeout_ms, 0, 0);
    if (error == EWOULDBLOCK)
        return (ETIMEDOUT);
}
```

And the simulation callout calls `cv_broadcast(&sc->data_cv)` after setting `DATA_AV`. The driver thread sleeps once (not 2000 times); the CPU overhead is minimal; the response time is bounded by the simulation's own timing, not by the driver's polling rate.

For Chapter 17, the polling pattern is taught first because it is easier to read: the `for` loop is explicit, the `pause_sbt` is explicit, the `CSR_READ_4` is explicit. The condition-variable pattern is more efficient but requires understanding all three of Chapter 12's primitives. We use the polling pattern as a stepping stone and note the condition-variable pattern as the production preference.

17.7.8 Timing for the Simulation Itself

The simulation's own timing choices are worth examining.

The sensor callout fires on an interval that the `SENSOR_CONFIG` register specifies. The default is 100 ms. The user can change it. The kernel honours the requested interval within a few milliseconds of precision. This is fine for a teaching sensor; a real device might require sub-microsecond precision, which would require `C_DIRECT_EXEC` callbacks or even non-standard timing sources.

The command callout fires `DELAY_MS` milliseconds after the command is issued. Again, millisecond precision is fine. The driver's timeouts are calibrated against this: the default command timeout is 2000 ms, well above the default `DELAY_MS` of 500 ms.

The driver's polling interval is 1 ms. For a command that takes 500 ms, that means ~500 polls, which is tolerable. For a command that takes 5 ms, that means ~5 polls, which is fast enough. For a command that takes 5 seconds, that means 5000 polls, which is wasteful. The polling interval could be adaptive (longer polls for longer commands), but the implementation complexity is not worth it for a teaching driver.

17.7.9 Realistic Constraints: Ready After 500 ms

To make the timing story concrete, configure the simulation for a 500 ms per-command delay and run a stress test:

```
# sysctl dev.myfirst.0.reg_delay_ms_set=500
# time echo -n "test" > /dev/myfirst0
```

Expected output:

```
real    0m2.012s
user    0m0.001s
sys     0m0.005s
```

Four bytes, 500 ms each, 2 seconds total. Almost all of the time is kernel sleep time; CPU usage is negligible. The driver is efficient even though the command is slow.

Now stress with concurrent writers:

```
# for i in 1 2 3 4; do
    (echo -n "ABCD" > /dev/myfirst0) &
done
# time wait
```

Four processes, 4 bytes each, 500 ms per byte. Total commands: 16. At one command per 500 ms (serialised through the driver lock), total time: 8 seconds. Expected:

```
real    0m8.092s
```

The driver lock serialises the commands, and the simulation's own `command_pending` check prevents overlapping. The effective throughput is one command per `DELAY_MS` milliseconds, regardless of the number of writers.

17.7.10 Handling Timeouts Gracefully

The driver's behaviour when a command times out deserves its own walkthrough. The command path:

1. Driver writes `DATA_IN`, sets `CTRL.GO`.
2. Driver waits up to `cmd_timeout_ms` for `STATUS.DATA_AV`.
3. If the wait times out:
 - a. Increment `cmd_data_timeouts` counter.
 - b. Call `myfirst_recover_from_stuck` to clear stale simulation state.
 - c. Return `ETIMEDOUT` to the caller.
4. If the wait succeeds, continue with the normal path.

The recovery is important. A simulation that is actually slow (not stuck) will eventually set `DATA_AV`, but the driver has already given up. If the driver did not clear the simulation's pending state, a subsequent command might see the lingering `DATA_AV` from the old command and think the new command completed immediately.

The recovery function stops the pending callout, clears the `command_pending` flag, and clears the relevant STATUS bits.

On real hardware, the equivalent recovery is a device reset. The driver writes `CTRL.RESET`, waits briefly, and restarts. Some devices have a separate reset mechanism accessible through PCI config space. The pattern is the same: after a timeout, get the device back to a known state before doing anything else.

17.7.11 A Debugging Technique: Artificial Slowdown

A useful debugging technique for timing-sensitive code: artificially slow the simulation down to stretch race windows.

```
# sysctl dev.myfirst.0.reg_delay_ms_set=3000
# sysctl dev.myfirst.0.cmd_timeout_ms=1000
```

Now every command takes 3 seconds but the driver only waits 1 second. Every command times out. The driver's recovery path runs on every command. Bugs in the recovery path surface immediately:

- If the recovery path crashes, the kernel panics on the first command.
- If the recovery path leaves stale state, the second command fails in some unusual way.
- If the recovery path leaks memory, `vmstat -m | grep myfirst` climbs over time.

This is a legitimate debugging technique. Real drivers are hard to stress this way because real hardware is hard to make slow on demand. The simulation, by contrast, is a single register away from being very slow. Use this capability to exercise code paths that normal operation rarely reaches.

17.7.12 Common Timing Mistakes

A short catalogue of timing mistakes to avoid.

Mistake 1: Using `DELAY` for milliseconds. `DELAY(1000000)` busy-waits for a full second, consuming a CPU core for that time. Use `pause_sbt(..., SBT_1S, ...)` instead.

Mistake 2: Holding a lock across `pause_sbt`. Blocks every context that needs the lock. Drop the lock, sleep, reacquire.

Mistake 3: Spinning without yielding on a condition that depends on another context. If your code is waiting for a callout to set a bit, the callout cannot run while you spin. Yield with `pause_sbt` (dropping the lock) or use a condition variable.

Mistake 4: Using `callout_reset_sbt` for a fire-and-forget action that should complete immediately. If the action is quick, just do it. Callouts have setup cost and scheduling overhead; they are worth it when the delay is meaningful.

Mistake 5: Setting a timeout shorter than the expected operation time. Every command times out. The driver does no useful work. Make timeouts generous but finite.

Mistake 6: Setting a timeout much longer than the expected operation time. A legitimate slow operation hangs the user-space call for the full timeout. Users lose patience. Make timeouts reasonable for the specific operation.

Mistake 7: Not draining callouts before freeing memory the callouts touch. The callout fires after the memory is freed, and the callback dereferences freed memory. Drain before free.

Mistake 8: Re-arming a callout from within its callback without checking for cancellation. The detach path asks the callout to stop; the callback re-arms itself anyway. Infinite loop; detach never completes. Check the running flag (as Section 3's `myfirst_sim_sensor_cb` does) before re-arming.

Each mistake is common in beginning driver code. The Chapter 17 stages avoid all of them by following the patterns Chapter 13 taught for callouts, Chapter 12 taught for condition variables, and Section 6 is now making explicit.

17.7.13 Wrapping Up Section 6

Three timing primitives, three time scales, three cost profiles. `DELAY(9)` for very short waits (< 100 us). `pause_sbt(9)` for medium waits (100 us to several seconds) when the thread has nothing else to do. `callout_reset_sbt(9)` for fire-and-forget scheduling and for periodic work. Choosing the right one is a judgment call informed by the time scale, the context, and what the thread would otherwise be doing.

The Chapter 17 driver uses `pause_sbt` for its polling loop and `callout_reset_sbt` for the simulation's own timers. A reset path uses `DELAY(5)` where appropriate. The driver's timing story is coherent: it never wastes CPU, never holds locks during sleeps, and never blocks other contexts unnecessarily.

Section 7 adds the last major piece of the Chapter 17 simulation: a fault-injection framework. With timing now reliable, the driver can be exercised under a variety of failure conditions and the error-recovery paths validated.

17.8 Section 7: Simulating Errors and Fault Conditions

A driver that passes every happy-path test and never exercises its error paths is a driver that will eventually fail in production. Error paths are the code that runs when something unexpected happens, and “unexpected” is the most honest word for what real hardware does on a bad day. A network card whose PHY lost its link signal. A disk controller whose command queue overflowed. A sensor whose calibration went out of range. A device whose firmware deadlocked after a specific command sequence.

Real hardware produces these conditions rarely, and usually at the worst possible moment. A driver author who waits for real hardware to produce them is a driver author whose customers will find the bugs. A driver author who uses simulation to produce them on demand can fix the bugs before they ship.

Section 7 builds a fault-injection framework into the Chapter 17 simulation. The framework lets the reader ask the simulation to misbehave in specific, controlled ways: a timeout, a data corruption, a stuck-busy state, a random failure. Each mode exercises a different driver code path. By the end of Section 7, the driver has seen every error it is designed to handle, and the developer has confidence that the error paths work.

17.8.1 The Fault-Injection Philosophy

Before writing code, a short philosophical pause. What should a fault-injection framework look like, and what should it not look like?

A good fault-injection framework:

- Targets specific failure modes, not random chaos. A fault that says “the next read returns 0xFFFFFFFF” is useful; a fault that says “something bad might happen” is not.
- Is controllable at fine granularity. Turn faults on, off, or partial. Inject one, inject many, inject probabilistically.
- Is observable. When a fault fires, the framework logs it (or provides a counter). The tester can see what happened and correlate with driver behaviour.
- Is orthogonal to other testing. A load test should not care whether faults are enabled; a fault test should not require load to be disabled.
- Is deterministic when configured to be. A fault configured at probability 100% fires every time. A fault configured at probability 50% fires half the time, and the pseudo-random decision is reproducible with a seed.

A bad fault-injection framework is either useless (never triggers the paths you need) or dangerous (produces failures the driver cannot recover from even in principle). The goal is useful unreliability, not destructive chaos.

The Chapter 17 framework achieves these goals with a small set of flags in `FAULT_MASK`, a probability in `FAULT_PROB`, and a counter that tracks how many faults have fired. The simulation consults these whenever an operation begins and either proceeds normally or injects a fault according to the configuration.

17.8.2 The Fault Modes

The four fault modes Chapter 17 implements:

MYFIRST_FAULT_TIMEOUT (bit 0). The next command never completes. The command callout is not scheduled; `STATUS.DATA_AV` never asserts. The driver’s `myfirst_wait_for_data` times out, triggering the recovery path.

MYFIRST_FAULT_READ_1S (bit 1). The next `DATA_OUT` read returns 0xFFFFFFFF instead of the real value. This simulates a bus read that fails in a way many real hardware buses do (a read of a disconnected or power-gated device commonly returns all-ones).

MYFIRST_FAULT_ERROR (bit 2). The next command completes, but with `STATUS.ERROR` set. The driver is expected to detect the error, clear the latch, and report an error to the caller.

MYFIRST_FAULT_STUCK_BUSY (bit 3). `STATUS.BUSY` is latched on and never clears. The driver’s `myfirst_wait_for_ready` times out before any command can be issued. This simulates a device that got wedged and needs a reset.

Each fault exercises a different path. The timeout fault exercises `cmd_data_timeouts` and the recovery. The read-1s fault tests whether the driver notices corrupted data. The error fault exercises `cmd_errors` and the error-clearing logic. The stuck-busy fault exercises `cmd_rdy_timeouts`.

Multiple faults can be active simultaneously: set multiple bits in `FAULT_MASK`. Each fault is applied independently at the point in the simulation where it is relevant.

17.8.3 The Fault-Probability Field

FAULT_MASK selects which faults are candidates; FAULT_PROB controls how often a candidate fault actually fires. Probabilities are expressed as integers from 0 to 10000, where 10000 means 100% (fault every time) and 0 means never. This gives four decimal places of precision without fractional arithmetic.

The test for whether a fault fires:

```
static bool
myfirst_sim_should_fault(struct myfirst_softc *sc)
{
    uint32_t prob, r;

    MYFIRST_LOCK_ASSERT(sc);

    prob = CSR_READ_4(sc, MYFIRST_REG_FAULT_PROB);
    if (prob == 0)
        return (false);
    if (prob >= 10000)
        return (true);

    r = arc4random_uniform(10000);
    return (r < prob);
}
```

arc4random_uniform(10000) returns a pseudo-random integer in [0, 10000). If it is less than prob, the fault fires. With prob = 5000 (50%), the fault fires about half the time. With prob = 100 (1%), about one in a hundred operations. With prob = 10000, every operation.

The function is called at the start of each operation. If it returns true, the operation applies a fault from FAULT_MASK (the simulation's logic decides which fault or faults to apply, typically the first matching one).

17.8.4 Implementing the Timeout Fault

The timeout fault is the simplest. In myfirst_sim_start_command, check the fault mask before scheduling the command callout:

```
void
myfirst_sim_start_command(struct myfirst_softc *sc)
{
    struct myfirst_sim *sim = sc->sim;
    uint32_t data_in, delay_ms, fault_mask;
    bool fault;

    MYFIRST_LOCK_ASSERT(sc);
```



```
if (!sim->running)
    return;

if (sim->command_pending) {
    sc->stats.cmd_rejected++;
    device_printf(sc->dev,
        "sim: overlapping command; ignored\n");
    return;
}

data_in = CSR_READ_4(sc, MYFIRST_REG_DATA_IN);
sim->pending_data = data_in;

{
    uint32_t status = CSR_READ_4(sc, MYFIRST_REG_STATUS);
    status |= MYFIRST_STATUS_BUSY;
    status &= ~MYFIRST_STATUS_DATA_AV;
    CSR_WRITE_4(sc, MYFIRST_REG_STATUS, status);
}

delay_ms = CSR_READ_4(sc, MYFIRST_REG_DELAY_MS);
if (delay_ms == 0)
    delay_ms = 1;

sim->command_pending = true;

/* Fault-injection check. */
fault_mask = CSR_READ_4(sc, MYFIRST_REG_FAULT_MASK);
fault = (fault_mask != 0) && myfirst_sim_should_fault(sc);

if (fault && (fault_mask & MYFIRST_FAULT_TIMEOUT)) {
    /* Do not schedule the completion callout. The command
     * will never complete; the driver will time out.
     * (Simulation does not clear command_pending; the
     * driver's recovery path is responsible for that.) */
    sc->stats.fault_injected++;
    device_printf(sc->dev,
        "sim: injecting TIMEOUT fault\n");
    return;
}

/* Save the fault state for the callout to honour. */
sim->pending_fault = fault ? fault_mask : 0;
```

```

        callout_reset_sbt(&sim->command_callout,
            delay_ms * SBT_1MS, 0,
            myfirst_sim_command_cb, sc, 0);
    }

```

Four changes from the Section 3 version.

First, a statistics counter for `cmd_rejected` replaces the bare `printf`.

Second, the function reads `FAULT_MASK` and calls `myfirst_sim_should_fault` to decide whether this operation injects a fault.

Third, if the timeout fault is selected, the function returns without scheduling the completion callout. `command_pending` is set to true, so overlapping commands are still rejected, but no callout ever fires to complete the command.

Fourth, the simulation stores the fault state in `sim->pending_fault` so the callout (when it fires, for non-timeout faults) can honour the fault.

`pending_fault` is added to the simulation state structure:

```

struct myfirst_sim {
    /* ... existing fields ... */
    uint32_t pending_fault; /* FAULT_MASK bits to apply at completion */
};

```

17.8.5 Implementing the Error Fault

The error fault fires at command completion. In `myfirst_sim_command_cb`:

```

static void
myfirst_sim_command_cb(void *arg)
{
    struct myfirst_softc *sc = arg;
    struct myfirst_sim *sim = sc->sim;
    uint32_t status, fault;

    MYFIRST_LOCK_ASSERT(sc);

    if (!sim->running || !sim->command_pending)
        return;

    fault = sim->pending_fault;

    /* Always clear BUSY. */

```

```

status = CSR_READ_4(sc, MYFIRST_REG_STATUS);
status &= ~MYFIRST_STATUS_BUSY;

if (fault & MYFIRST_FAULT_ERROR) {
    /* Set ERROR, do not set DATA_AV. The driver should
     * detect the error on the next STATUS read. */
    status |= MYFIRST_STATUS_ERROR;
    sc->stats.fault_injected++;
    device_printf(sc->dev,
        "sim: injecting ERROR fault\n");
} else if (fault & MYFIRST_FAULT_READ_1S) {
    /* Normal completion, but DATA_OUT is corrupted. */
    CSR_WRITE_4(sc, MYFIRST_REG_DATA_OUT, 0xFFFFFFFF);
    status |= MYFIRST_STATUS_DATA_AV;
    sc->stats.fault_injected++;
    device_printf(sc->dev,
        "sim: injecting READ_1S fault\n");
} else {
    /* Normal completion. */
    CSR_WRITE_4(sc, MYFIRST_REG_DATA_OUT, sim->pending_data);
    status |= MYFIRST_STATUS_DATA_AV;
}

CSR_WRITE_4(sc, MYFIRST_REG_STATUS, status);

sim->op_counter++;
CSR_WRITE_4(sc, MYFIRST_REG_OP_COUNTER, sim->op_counter);

sim->command_pending = false;
sim->pending_fault = 0;
}

```

Three branches. The error fault sets `STATUS.ERROR` instead of `STATUS.DATA_AV`, leaving `DATA_OUT` untouched. The read-1s fault writes `0xFFFFFFFF` to `DATA_OUT` and sets `DATA_AV` normally. The normal branch writes the real data and sets `DATA_AV`.

The simulation increments `op_counter` and `fault_injected` appropriately. The driver will see the effects through the register values; the counters let the test validate that the faults fired.

17.8.6 Implementing the Stuck-Busy Fault

The stuck-busy fault is orthogonal to the command cycle: it keeps `STATUS.BUSY` set independently of any command. Add a callout that monitors `FAULT_MASK` and keeps `STATUS.BUSY` latched when the bit is set:

```

static void
myfirst_sim_busy_cb(void *arg)
{
    struct myfirst_softc *sc = arg;
    struct myfirst_sim *sim = sc->sim;
    uint32_t fault_mask, status;

    MYFIRST_LOCK_ASSERT(sc);

    if (!sim->running)
        return;

    fault_mask = CSR_READ_4(sc, MYFIRST_REG_FAULT_MASK);
    if (fault_mask & MYFIRST_FAULT_STUCK_BUSY) {
        status = CSR_READ_4(sc, MYFIRST_REG_STATUS);
        if (!(status & MYFIRST_STATUS_BUSY)) {
            status |= MYFIRST_STATUS_BUSY;
            CSR_WRITE_4(sc, MYFIRST_REG_STATUS, status);
        }
    }

    /* Re-arm every 50 ms. */
    callout_reset_sbt(&sim->busy_callout,
        50 * SBT_1MS, 0,
        myfirst_sim_busy_cb, sc, 0);
}

```

Initialise and start the callout alongside the sensor callout in `myfirst_sim_attach` and `myfirst_sim_enable`:

```

callout_init_mtx(&sim->busy_callout, &sc->mtx, 0);
/* ... in enable: */
callout_reset_sbt(&sim->busy_callout, 50 * SBT_1MS, 0,
    myfirst_sim_busy_cb, sc, 0);
/* ... in disable: */
callout_stop(&sim->busy_callout);
/* ... in detach: */
callout_drain(&sim->busy_callout);

```

Now, when `FAULT_STUCK_BUSY` is set, the busy callout continually re-asserts `STATUS.BUSY` every 50 ms. Any command the driver tries to issue will find `BUSY` set, wait for it to clear, and time out on `wait_for_ready`. Clearing the `FAULT_STUCK_BUSY` bit stops the re-assertion, and `BUSY` returns to its natural state (whichever the command path left it in).

17.8.7 The Driver's Error-Handling Paths

With faults wired in, the driver's error-handling paths get exercised. Review them now to confirm they work correctly.

Timeout on wait_for_ready (from Section 5):

```
error = myfirst_wait_for_ready(sc, sc->rdy_timeout_ms);
if (error != 0) {
    sc->stats.cmd_rdy_timeouts++;
    return (error);
}
```

When `FAULT_STUCK_BUSY` is set, this path triggers. The caller sees `ETIMEDOUT`. The driver does not try to issue the command because it cannot. The counter reflects the event.

Timeout on wait_for_data:

```
error = myfirst_wait_for_data(sc, sc->cmd_timeout_ms);
if (error != 0) {
    sc->stats.cmd_data_timeouts++;
    myfirst_recover_from_stuck(sc);
    return (error);
}
```

When `FAULT_TIMEOUT` is set, this path triggers. The driver calls `myfirst_recover_from_stuck`, which clears the simulation's `command_pending` flag and `STATUS.BUSY`. The next command sees a clean state. The counter reflects the event.

STATUS.ERROR detection:

```
status = CSR_READ_4(sc, MYFIRST_REG_STATUS);
if (status & MYFIRST_STATUS_ERROR) {
    CSR_UPDATE_4(sc, MYFIRST_REG_STATUS,
        MYFIRST_STATUS_ERROR, 0);
    CSR_UPDATE_4(sc, MYFIRST_REG_STATUS,
        MYFIRST_STATUS_DATA_AV, 0);
    sc->stats.cmd_errors++;
    return (EIO);
}
```

When `FAULT_ERROR` is set, this path triggers. The driver clears `STATUS.ERROR` and `STATUS.DATA_AV`, increments the counter, returns `EIO`.

Corrupted DATA_OUT handling requires an addition. When `FAULT_READ_1S` is set, `DATA_OUT` is `0xFFFFFFFF`, but the driver currently does not check for this. Whether the driver should check depends on the protocol: for

a sensor, 0xFFFFFFFF is often a legitimate “read error” marker; for other devices, it might be a plausible data value. Chapter 17’s driver treats it as a potential error for illustration:

```
if (data_out == 0xFFFFFFFF) {  
    /* Likely a bus read error. Real devices rarely produce this  
     * value as legitimate data; treat as an error. */  
    sc->stats.cmd_errors++;  
    return (EIO);  
}
```

This behaviour is a judgment call. A real driver for a real device would check the datasheet for the device’s documented bad values and respond appropriately.

17.8.8 Testing the Faults One at a Time

With the infrastructure in place, each fault can be tested in isolation:

Test 1: Timeout fault at 100%.

```
# sysctl dev.myfirst.0.reg_fault_mask_set=0x1    # MYFIRST_FAULT_TIMEOUT  
# sysctl dev.myfirst.0.reg_fault_prob_set=10000  # 100%  
# echo -n "test" > /dev/myfirst0  
write: Operation timed out  
# sysctl dev.myfirst.0.cmd_data_timeouts  
dev.myfirst.0.cmd_data_timeouts: 1  
# sysctl dev.myfirst.0.fault_injected  
dev.myfirst.0.fault_injected: 1
```

The first byte triggers a timeout. The driver recovers. Subsequent bytes would also time out until the fault is cleared. The counters reflect the event.

Turn off the fault:

```
# sysctl dev.myfirst.0.reg_fault_mask_set=0  
# sysctl dev.myfirst.0.reg_fault_prob_set=0  
# echo -n "test" > /dev/myfirst0  
# echo "write succeeded"  
write succeeded  
# sysctl dev.myfirst.0.cmd_successes  
dev.myfirst.0.cmd_successes: 4    # or more, depending on history
```

Commands succeed again.

Test 2: Error fault at 25%.

```
# sysctl dev.myfirst.0.reg_fault_mask_set=0x4    # MYFIRST_FAULT_ERROR
# sysctl dev.myfirst.0.reg_fault_prob_set=2500    # 25%
# sysctl dev.myfirst.0.cmd_errors
dev.myfirst.0.cmd_errors: 0
# for i in $(seq 1 40); do echo -n "X" > /dev/myfirst0; done
write: Input/output error      (occurs roughly 10 times)
# sysctl dev.myfirst.0.cmd_errors
dev.myfirst.0.cmd_errors: 9     # approximately 25% of 40
```

About one in four commands errors out. The driver detects the error, clears the state, and returns EIO. The counter reflects the event.

Test 3: Stuck-busy fault.

```
# sysctl dev.myfirst.0.reg_fault_mask_set=0x8    # MYFIRST_FAULT_STUCK_BUSY
# sysctl dev.myfirst.0.reg_fault_prob_set=10000   # (prob doesn't matter; latch is
↪ always on)
# echo -n "X" > /dev/myfirst0
write: Operation timed out
# sysctl dev.myfirst.0.cmd_rdy_timeouts
dev.myfirst.0.cmd_rdy_timeouts: 1
# sysctl dev.myfirst.0.reg_status
dev.myfirst.0.reg_status: 3     # READY|BUSY
```

The driver cannot issue a command because BUSY is latched on. Clear the fault:

```
# sysctl dev.myfirst.0.reg_fault_mask_set=0
# sleep 1    # wait for the busy callout to stop re-asserting
# sysctl dev.myfirst.0.reg_status
dev.myfirst.0.reg_status: 1     # READY only
# echo -n "X" > /dev/myfirst0
# echo "write succeeded"
write succeeded
```

17.8.9 Integration: Random Faults Under Load

The most realistic test configures random faults at a low probability and runs the driver under load:

```
#!/bin/sh
# fault_stress.sh: random faults under load.

# 1% probability of TIMEOUT or ERROR (bits 0 and 2 both set).
sysctl dev.myfirst.0.reg_fault_mask_set=0x5
```

```

sysctl dev.myfirst.0.reg_fault_prob_set=100

# Fast commands so the test runs in reasonable time.
sysctl dev.myfirst.0.reg_delay_ms_set=10
sysctl dev.myfirst.0.cmd_timeout_ms=50    # very short, to spot timeouts fast

# Load: 8 parallel workers, each doing 200 round trips.
for i in $(seq 1 8); do
    (for j in $(seq 1 200); do
        echo -n "X" > /dev/myfirst0 2>/dev/null
        dd if=/dev/myfirst0 bs=1 count=1 of=/dev/null 2>/dev/null
    done) &
done
wait

# Report.
sysctl dev.myfirst.0 | grep -E 'cmd_|fault_|op_counter'

# Clean up.
sysctl dev.myfirst.0.reg_fault_mask_set=0
sysctl dev.myfirst.0.reg_fault_prob_set=0
sysctl dev.myfirst.0.reg_delay_ms_set=500
sysctl dev.myfirst.0.cmd_timeout_ms=2000

```

Expected counters after the run:

- cmd_successes: around 3168 (out of ~3200 expected round trips)
- cmd_data_timeouts: around 16 (1% of half the operations)
- cmd_errors: around 16 (1% of half the operations)
- fault_injected: around 32 (sum of the two)
- cmd_recoveries: around 16 (one per timeout)

The exact numbers vary with each run (the faults are random), but the proportions should be roughly 1% for each fault type. If the proportions are wildly off, there is either a bug in the fault injection or a bug in the driver's error-counting. Either way, the test is doing its job by surfacing anomalies.

17.8.10 A Common Pitfall: Recovery That Never Clears Everything

A bug that shows up under fault testing is recovery logic that does not clear everything. Suppose `myfirst_recover_from_stuck` forgot to clear `STATUS.BUSY`:

```

/* BUGGY version: */
static void
myfirst_recover_from_stuck(struct myfirst_softc *sc)

```



```

{
    MYFIRST_LOCK_ASSERT(sc);
    if (sc->sim != NULL) {
        sc->sim->command_pending = false;
        callout_stop(&sc->sim->command_callout);
    }
    /* Missing: the STATUS cleanup. */
}

```

Running the stress test would show:

- First timeout: recovery runs, `command_pending` clears.
- Second command attempt: `STATUS.BUSY` is still set from the first attempt. The command cannot issue. `cmd_rdy_timeouts` climbs.
- Third attempt: same.
- ... forever.

The counter `cmd_successes` stops incrementing. `cmd_rdy_timeouts` climbs without bound. A careful test notices this and flags it. A careless test just sees “lots of errors” and shrugs.

The lesson is that fault injection exposes incomplete recovery. A driver that works under fault-free conditions may still have bugs that only appear when faults happen. The stress test is what surfaces them.

17.8.11 Combining Faults

The Chapter 17 framework allows multiple faults to be active simultaneously:

```

# sysctl dev.myfirst.0.reg_fault_mask_set=0xf    # all four faults
# sysctl dev.myfirst.0.reg_fault_prob_set=1000   # 10%

```

Each operation has a 10% chance of triggering a fault. If it does, the simulation picks one of the enabled faults (in practice, the code checks them in order: `TIMEOUT`, `ERROR`, `READ_1S`, `STUCK_BUSY`). The driver’s response must be robust against all of them.

An even harsher test:

```

# sysctl dev.myfirst.0.reg_fault_prob_set=10000  # 100%

```

Every operation faults. The driver never completes a command successfully. Every error path runs. Run the driver for a minute under this setting. If the driver crashes, there is a bug in an error path. If the driver leaks memory (check `vmstat -m | grep myfirst`), there is a leak in an error path. If the driver deadlocks (check with `procstat -kk`), there is a deadlock in an error path.

Running under 100% faults for a sustained period is a standard test for fault-injection frameworks. It is intentionally hostile; a driver that passes is robust to real-world failure conditions.

17.8.12 Observability for Fault Testing

The driver's existing observability (access log, stats counters) carries most of what fault testing needs. Three additions are worth making.

Fault-injected log entries. The access log records every register access; it should record that a fault was injected. Add a new access-log entry type:

```
#define MYFIRST_CTX_FAULT    0x10    /* fault injected */
```

And log it when a fault fires:

```
myfirst_access_log_push(sc, offset, 0, 4, true, MYFIRST_CTX_FAULT);
```

Now the access log shows, interleaved with normal accesses, entries that indicate “fault X was injected here”. The tester can correlate fault injection with subsequent driver behaviour.

Per-fault-type counters. Instead of a single `fault_injected` counter, track each type:

```
struct myfirst_stats {
    /* ... existing counters ... */
    uint64_t    fault_timeout;
    uint64_t    fault_read_1s;
    uint64_t    fault_error;
    uint64_t    fault_stuck_busy;
};
```

And increment the right one based on the fault:

```
if (fault & MYFIRST_FAULT_TIMEOUT)
    sc->stats.fault_timeout++;
else if (fault & MYFIRST_FAULT_ERROR)
    sc->stats.fault_error++;
else if (fault & MYFIRST_FAULT_READ_1S)
    sc->stats.fault_read_1s++;
```

The `sysctl` exposes each counter. The tester can see, at a glance, which fault types have fired and which have not.

A fault-injection summary `sysctl`. A single `sysctl` that reports “fault mask is X, probability is Y, total injected is Z”. Useful for quick checks during interactive debugging:

```
static int
myfirst_sysctl_fault_summary(SYSCTL_HANDLER_ARGS)
{
```

```

    struct myfirst_softc *sc = arg1;
    char buf[128];
    uint32_t mask, prob;
    uint64_t injected;

    MYFIRST_LOCK(sc);
    mask = CSR_READ_4(sc, MYFIRST_REG_FAULT_MASK);
    prob = CSR_READ_4(sc, MYFIRST_REG_FAULT_PROB);
    injected = sc->stats.fault_injected;
    MYFIRST_UNLOCK(sc);

    snprintf(buf, sizeof(buf),
             "mask=0x%x prob=%u/10000 injected=%ju",
             mask, prob, (uintmax_t)injected);
    return (sysctl_handle_string(oidp, buf, sizeof(buf), req));
}

```

Invocation:

```

# sysctl dev.myfirst.0.fault_summary
dev.myfirst.0.fault_summary: mask=0x5 prob=100/10000 injected=47

```

This is the kind of summary that fits on one line and tells the whole story of the current fault configuration.

17.8.13 Safe Fault Injection

The word “safe” needs qualification. Fault injection in the Chapter 17 simulation is safe because the simulation runs in the same kernel as the driver, and the faults are local: a corrupted DATA_OUT is a corrupted four-byte value in malloc'd memory, not a corrupted DMA descriptor that scribbles over random RAM. A stuck BUSY bit is a stuck bit in a simulated register, not a hardware queue that backs up until the system hangs.

Real hardware fault injection is a different story. A deliberately timed-out real device may hold a bus lock indefinitely, preventing other devices from making progress. A deliberately corrupted DMA transaction may write to arbitrary physical addresses. Real fault injection on production systems requires careful setup, dedicated test hardware, and usually a hypervisor or virtual device emulation.

Chapter 17's simulation sidesteps all of this by running in memory. The worst case of a misbehaved fault is a kernel panic (if a KASSERT fires) or a deadlock (if recovery code is incorrect). Both are survivable: a debug kernel catches panics cleanly, and a deadlocked driver can be killed by rebooting. No physical hardware is at risk.

This safety is why fault injection belongs in simulation, not just in production testing. Production testing catches the faults real hardware actually produces; simulation testing catches the faults real hardware sometimes produces and always hurts from.

17.8.14 Hands-On: The Fault-Injection Checklist

A procedure for using fault injection effectively during development:

1. Write the driver's error path first. Before enabling the fault, read your code and identify what the recovery looks like.
2. Enable the fault at 100% probability briefly. Verify the driver's error path runs. Check the counters.
3. Enable the fault at low probability (1-10%). Run a stress test. Verify the driver still makes progress overall.
4. Check `vmstat -m | grep myfirst` before and after the stress test. Memory usage should not grow.
5. Check `WITNESS` output for lock-order warnings. A fault that causes an unusual path may expose a latent lock-order bug.
6. Check `INVARIANTS` output for assertion failures. A fault that corrupts state may trip an invariant.
7. Turn the fault off. Run the happy path. Verify everything returns to normal.
8. Document the fault in `HARDWARE.md` or `SIMULATION.md`.

This procedure turns fault injection into a disciplined activity, not a chaotic experiment. Each step has a clear goal, and each step validates a specific property of the driver.

17.8.15 A Reflection: What Fault Injection Taught Us

Fault injection is a microcosm of the larger driver-development discipline. The driver must handle errors it cannot prevent, from sources it does not control, at times it cannot predict. Fault injection gives the developer a tool to exercise all three properties on demand.

More than that, fault injection reveals the structure of the driver's error-handling philosophy. A driver that treats every error as equally recoverable handles some real-world failures poorly. A driver that distinguishes "transient" (try again) from "catastrophic" (reset the device) has a clearer model of what the device is doing. A driver that logs every error with enough context to diagnose later is a driver whose bugs can be found by log analysis.

The Chapter 17 driver does not go all the way to this level of sophistication; it has a single error-handling path that treats all errors as transient and retries only if the caller does. But the foundation is in place. A challenge at the end of the chapter invites the reader to extend the driver with per-error-type handling, and the simulation has the hooks to exercise each type.

17.8.16 Wrapping Up Section 7

Fault injection is how driver authors exercise error-handling code before real failures arrive. Chapter 17's framework provides four fault modes (timeout, read-1s, error, stuck-busy), a probability field, and enough observability to correlate injected faults with driver behaviour.

The driver's error paths are validated by enabling each fault in isolation and confirming the expected counter increments and the expected recovery. Combining faults and running under load at low probabilities exercises the composition of error paths. Running at 100% probability stresses every error path simultaneously.

The version tag becomes `1.0-sim-stage4`. The driver has now seen every error it is designed to handle, and the developer has observed each error path running correctly. The remaining work is organisational: refactor, version, document, and bridge to Chapter 18.

17.9 Section 8: Refactoring and Versioning Your Simulated Hardware Driver

Stage 4 produced a driver that works correctly under both happy-path and fault-injection conditions. Stage 5 (Section 8) makes the driver maintainable. The changes are organisational: clean file boundaries, updated documentation, a version bump, and the regression pass that verifies nothing broke along the way.

A driver that works is valuable. A driver that works and reads cleanly to the next contributor is far more valuable. Section 8 is about that second step.

17.9.1 The Final File Layout

Through Chapter 16, the driver was `myfirst.c`, `myfirst_hw.c`, `myfirst_hw.h`, `myfirst_sync.h`, `cbuf.c`, and `cbuf.h`. Chapter 17 added `myfirst_sim.c` and `myfirst_sim.h`. The final file tree:

- `myfirst.c`: the driver lifecycle, the syscall handlers, the Chapter 11-15 synchronisation primitives.
- `myfirst_hw.c`: the Chapter 16 hardware-access layer. `CSR_*` macros, register accessors, access log, register-view sysctls.
- `myfirst_hw.h`: the register map, bit masks, fixed values, and API prototypes for the hardware layer.
- `myfirst_sim.c`: the Chapter 17 simulation backend. Sensor callout, command callout, busy callout, fault-injection logic, simulation API.
- `myfirst_sim.h`: the simulation's API prototypes and state structure.
- `myfirst_sync.h`: the Chapter 15 synchronisation-primitive header.
- `cbuf.c`, `cbuf.h`: the Chapter 10 ring buffer.

Seven source files, three headers, a Makefile, and three documentation files (`HARDWARE.md`, `LOCKING.md`, and the new `SIMULATION.md`). The split mirrors how production FreeBSD drivers organise themselves: one file per responsibility, with named interfaces between them.

17.9.2 The Simulation-Specific Fields Move into the Sim Struct

Chapter 17 introduced the simulation state gradually. By the end of Section 7, some simulation-related fields were in the `softc` rather than in the struct `myfirst_sim`. Stage 5 tidies this up.

Fields that should live in struct `myfirst_sim`:

- The three callouts (`sensor_callout`, `command_callout`, `busy_callout`).
- The simulation-internal state (`pending_data`, `pending_fault`, `command_pending`, `running`, `sensor_baseline`, `sensor_tick`, `op_counter`).

Fields that should stay in the `softc` or in struct `myfirst_hw`:

- The register block and bus_space state (`regs_buf`, `regs_tag`, `regs_handle`): `myfirst_hw`.
- The driver's own timeouts (`cmd_timeout_ms`, `rdy_timeout_ms`): the `softc`.
- The statistics (`sc->stats`): the `softc` (shared by hw and sim paths).

The reasoning: hardware state belongs to the hardware layer; simulation state belongs to the simulation layer; the driver-visible behaviour belongs to the driver. Chapter 18 will replace the simulation with a real PCI back-end, and the cleanest replacement is one where `myfirst_sim.c` and `myfirst_sim.h` disappear entirely, to be

replaced by `myfirst_pci.c` and (optionally) `myfirst_pci.h`. The `sim` struct's fields do not need to survive that transition; the driver's fields do.

17.9.3 The `SIMULATION.md` Document

A new markdown file, `SIMULATION.md`, captures the simulation's interface. The sections:

1. **Version and scope.** “SIMULATION.md version 1.0. Chapter 17 complete.”
2. **What the simulation is.** One paragraph: “A kernel-memory-backed simulated device that mimics a minimal command-response hardware protocol, used to teach driver development without requiring real hardware.”
3. **What the simulation does.** Enumerated behaviours: sensor autonomy, command cycle, fault injection, configurable delays.
4. **What the simulation does not do.** Enumerated limitations: no DMA, no interrupts, no PCI, no real timing precision.
5. **Callout map.** Table of each callout, its purpose, its default interval, its relationship to the driver lock.
6. **Fault modes.** Table of each fault type, its trigger, its effect, its recovery.
7. **Sysctl reference.** Table of each sysctl exposed by the simulation layer, its type, its default, its purpose.
8. **Development guidance.** How to add a new behaviour to the simulation (where to put the code, what locking discipline applies, how to document).
9. **Relationship to real hardware.** How the patterns in the simulation map to patterns in real FreeBSD drivers.

The document is perhaps 150 to 200 lines of markdown. It is the single source of truth for what the simulation promises. A developer reading it should be able to reason about the simulation without reading the code.

An example entry from the fault-modes table:

Fault	Bit	Trigger	Effect
↪ Recovery			
----- ----- ----- -----			
↪ ----- ----- -----			
MYFIRST_FAULT_TIMEOUT	0	should_fault() returns true	Command callout not
↪ scheduled driver timeout path			
MYFIRST_FAULT_READ_1S	1	should_fault() returns true	DATA_OUT = 0xFFFFFFFF
↪ driver error check			
MYFIRST_FAULT_ERROR	2	should_fault() returns true	STATUS.ERROR set
↪ instead of DATA_AV driver error path			
MYFIRST_FAULT_STUCK_BUSY	3	FAULT_MASK bit set	STATUS.BUSY
↪ continuously latched clear FAULT_MASK			

Such tables make reading the code much easier: a reader sees `MYFIRST_FAULT_READ_1S` in the code and can look up the full story in one place.

17.9.4 The Updated `HARDWARE.md`

Chapter 16's `HARDWARE.md` described a static register block. Chapter 17 extends it with the dynamic registers and the behaviours that drive them. The sections that change:

Register Map. The table now includes the Chapter 17 registers: `SENSOR` (0x28), `SENSOR_CONFIG` (0x2c), `DELAY_MS` (0x30), `FAULT_MASK` (0x34), `FAULT_PROB` (0x38), `OP_COUNTER` (0x3c). Each with its access type, default value, and one-line description.

CTRL register fields. A new bit: `MYFIRST_CTRL_GO` (bit 9). The table adds a row: “bit 9: GO, write 1 to trigger a command; hardware self-clears”.

STATUS register fields. The table gains notes on which bits are dynamic in Chapter 17: `READY` is set at attach and stays on; `BUSY` is set by `start_command` and cleared by `command_cb`; `DATA_AV` is set by `command_cb` and cleared by the driver after reading `DATA_OUT`; `ERROR` is set by fault injection and cleared by the driver.

SENSOR_CONFIG fields. A new subsection explaining the two-field layout (low 16 bits interval, high 16 bits amplitude).

FAULT_MASK fields. A new subsection listing each fault bit and its effect.

Dynamic behaviour summary. A new subsection explaining what changes autonomously: `SENSOR` (every 100 ms by default), `STATUS.BUSY` (set/clear by command cycle or by `FAULT_STUCK_BUSY` callout), `STATUS.DATA_AV` (set by `command_cb`), `OP_COUNTER` (incremented on each command).

Locking discipline for dynamic behaviour. A sentence: “All dynamic updates happen under `sc->mtx`, which callouts acquire automatically via `callout_init_mtx`. The driver's command path acquires the mutex and drops it only during `pause_sbt` calls, where the simulation callouts may run.”

The document grows from perhaps 80 lines (Chapter 16 version) to perhaps 140 lines. It remains scannable. The new sections are organised so a reader looking for Chapter 17 information finds it quickly.

17.9.5 The Updated `LOCKING.md`

Chapter 15's `LOCKING.md` described a lock order of `sc->mtx -> sc->cfg_sx -> sc->stats_cache_sx` and a detach ordering that drained several primitives. Chapter 16 added the hardware layer's detach to the list. Chapter 17 adds the simulation layer's detach.

The updated detach ordering, from outermost to innermost:

1. Destroy `sc->cdev` and wait for any open file descriptors to close.
2. Stop and drain all driver-level callouts (heartbeat, watchdog, tick_source).
3. Disable and drain the simulation's callouts (sensor, command, busy).
4. Free the simulation state.
5. Detach the hardware layer (free `regs_buf`, free `hw`).
6. Destroy the driver's synchronisation primitives (mutex, sx, cv, sema).

Step 3 is new in Chapter 17. The simulation is stopped before the hardware layer is torn down, because the simulation's callouts touch the register block through the hardware layer's accessors. Freeing the register block while a callout is still in flight would produce a use-after-free.

The ordering of step 3 and step 2 matters. Driver-level callouts (heartbeat) may read registers, so they must drain before the simulation drains. A deliberate ordering: driver callouts first (they are the outermost consumers), then simulation callouts (the innermost), then the registers themselves.

17.9.6 The Version Bump

In `myfirst.c`:

```
#define MYFIRST_VERSION "1.0-simulated"
```

The version goes to 1.0 because Chapter 17 marks a real milestone: the driver now behaves, end to end, like a driver against a functional device. Chapter 16's 0.9-mmio was a register-access chapter; Chapter 17's 1.0-simulated is a fully-functional-driver chapter. The jump from 0.9 to 1.0 reflects this.

The top-of-file comment is updated:

```
/*
 * myfirst: a beginner-friendly device driver tutorial vehicle.
 *
 * Version 1.0-simulated (Chapter 17): adds dynamic simulation of the
 * Chapter 16 register block. Includes autonomous sensor updates,
 * command-triggered delayed events, read-to-clear semantics, and a
 * fault-injection framework. Simulation code lives in myfirst_sim.c;
 * the driver sees it only through register accesses.
 *
 * ... (previous version notes preserved) ...
 */
```

The Chapter 17 comment is two sentences. It points at the new file and names the new capabilities. A future contributor reading it has enough context to understand what the version represents.

17.9.7 The Regression Pass

Chapter 15 established the regression discipline: after every version bump, run the full stress suite from every previous chapter, confirm WITNESS is silent, confirm INVARIANTS is silent, confirm `kldunload` completes cleanly.

For Stage 5 that means:

- The Chapter 11 concurrency tests (multiple writers, multiple readers) pass.
- The Chapter 12 blocking tests (reader waits for data, writer waits for room) pass.
- The Chapter 13 callout tests pass.
- The Chapter 14 task tests pass.
- The Chapter 15 coordination tests pass.
- The Chapter 16 register-access tests pass.

- The Chapter 17 simulation tests (fault injection, timing, behaviour) pass.
- `kldunload myfirst` returns cleanly after the full suite.

No test is skipped. A regression in any previous chapter’s test is a bug, not a deferred issue. The discipline is the same as it has been throughout Part 3.

The Chapter 17 test additions include:

- `sim_sensor_oscillates.sh`: confirms `SENSOR` changes over time.
- `sim_command_cycle.sh`: runs a series of write commands and verifies `OP_COUNTER` increments.
- `sim_timeout_fault.sh`: enables the timeout fault and verifies the driver recovers.
- `sim_error_fault.sh`: enables the error fault and verifies the driver reports `EIO`.
- `sim_stuck_busy_fault.sh`: enables the stuck-busy fault and verifies the driver times out on ready.
- `sim_mixed_faults_under_load.sh`: 10% fault probability, 8 parallel workers, 30 seconds.

Each script is a few dozen lines. Together they add about 300 lines of test infrastructure, which is small compared to the driver itself but significant in the confidence it provides.

17.9.8 Running the Final Stage

```
# cd examples/part-04/ch17-simulating-hardware/stage5-final
# make clean && make
# kldstat | grep myfirst
# kldload ./myfirst.ko
# kldstat -v | grep -i myfirst

myfirst: version 1.0-simulated

# dmesg | tail -5
# sysctl dev.myfirst.0 | head -40
```

The `kldstat -v` output shows `myfirst` at version `1.0-simulated`. The `dmesg` tail shows the device probe and attach with no errors. The `sysctl` output lists every Chapter 11 through Chapter 17 `sysctl`, including the simulation controls.

Run the stress suite:

```
# ../labs/full_regression.sh
```

If every test passes, Chapter 17 is complete.

17.9.9 A Small Rule for Chapter 17’s Refactor

Chapter 16’s refactor was about separating “driver business logic” from “hardware register mechanics”. Chapter 17’s refactor is about separating “simulation” from “hardware access”. The rule generalises: when a subsystem acquires a new responsibility, give it its own file; when a file acquires multiple responsibilities, split them.

A rule of thumb: a file more than about 800 to 1000 lines often has more than one responsibility. A header that exports more than about ten functions often merits a split. A file that imports a header from a subsystem unrelated to its main purpose often signals a responsibility leak.

Chapter 17's `myfirst_sim.c` is about 300 lines at Stage 5. `myfirst_hw.c` is about 400 lines. `myfirst.c` is about 800 lines. Each file holds one responsibility. No file grew out of control. The split scales: Chapter 18 will add `myfirst_pci.c` (about 200 to 300 lines), Chapter 19 will add `myfirst_intr.c`, Chapter 20 will add `myfirst_dma.c`. Each subsystem lives in its own file. The main `myfirst.c` stays roughly constant in size; the subsystems grow around it.

17.9.10 What Stage 5 Accomplished

The driver is now at 1.0-simulated. Compared to 0.9-mmio, it has:

- A simulation backend in `myfirst_sim.c` and `myfirst_sim.h`.
- A dynamic register block with autonomous sensor updates, command-triggered delayed events, and fault injection.
- Configurable timeouts and per-error-type counters.
- A `SIMULATION.md` document describing the simulation's interface and limits.
- An updated `HARDWARE.md` reflecting the new registers and dynamic behaviour.
- An updated `LOCKING.md` reflecting the new detach ordering.
- Regression tests that exercise every Chapter 17 behaviour.

The driver's code is recognisably FreeBSD. The layout is the layout real drivers use when they have distinct simulation, hardware, and driver responsibilities. The vocabulary is the vocabulary real drivers share. A contributor opening the driver for the first time finds a familiar structure, reads the documentation, and can navigate the code by subsystem.

17.9.11 Real FreeBSD Drivers That Use the Same Patterns

The patterns this chapter exercises are not confined to simulated hardware. Three places in the FreeBSD tree are especially worth opening alongside Chapter 17, because each one uses a similar shape in a real subsystem and reading them turns the techniques from “how I wrote the simulation” into “how the kernel actually operates.”

The first is the **watchdog(4)** subsystem in `/usr/src/sys/dev/watchdog/watchdog.c`. The core routine `wdog_kern_pat()` at the top of that file is a small state machine driven by a periodic “pat” from user space or from another kernel subsystem; if the pat does not arrive within the configured timeout, the subsystem fires a pretimeout handler and, ultimately, a system reset. The parallel with the Chapter 17 simulation is direct: a timeout value in ticks, a callout that advances state in the background, an ioctl surface (`WDIOC_SETTIMEOUT`) that changes the interval from user space, and a sysctl surface that exposes the last configured timeout for observation. It is also short enough to read end-to-end, which is rare for a production subsystem.

The second is **random_harvestq**, the entropy-collection path, in `/usr/src/sys/dev/random/random_harvestq.c`. The function `random_harvestq_fast_process_event()` and its surrounding queue discipline are the kernel's version of the “accept events from many sources and process them in the background” pattern this chapter exercised with a simulated sensor. The harvest queue uses a ring buffer, a worker thread, and explicit backpressure when consumers fall behind, and it is one of the cleaner examples in the tree

of a driver-like subsystem that must never block the code paths that feed it. Reading it after Chapter 17 shows what the autonomous-update pattern looks like when the “sensor” is every entropy source in the system at once.

The third, worth mentioning briefly, is the **pseudo-random device** in `/usr/src/sys/dev/random/randomdev.c`. It uses a cdev surface, configurable sysctls, and careful separation between the harvest side and the output side. That separation is the same split Chapter 17 introduced between `myfirst_sim.c` and `myfirst.c`, and looking at how `randomdev.c` arranges its files is a useful second example of the discipline this section just applied to the simulated driver.

None of these are “toy” subsystems. They are live kernel code that has shipped for years. The point of the pointers is not to ask you to master them now, but to mark where the techniques you just practised in simulation live in production code, so that when you open `/usr/src/sys` later and a file looks familiar, you have already seen the pattern.

17.9.12 Wrapping Up Section 8

The refactor is, again, small in code but significant in organisation. A new file split, a new documentation file, updates to two existing documentation files, a version bump, and a regression pass. Each step is cheap; together they turn a working driver into a maintainable one.

The Chapter 17 driver is done. The chapter closes with labs, challenges, troubleshooting, and a bridge to Chapter 18, where the simulated register block is replaced with a real PCI BAR.

17.10 Hands-On Labs

Labs in Chapter 17 focus on exercising the simulation from multiple angles: watching autonomous behaviour, running commands under load, injecting faults, and observing driver reactions. Each lab takes 20 to 60 minutes.

17.10.1 Lab 1: Watch the Sensor Breathe

Load the driver and watch the sensor value change without any driver activity.

```
# kldload ./myfirst.ko
# sysctl dev.myfirst.0.sim_running
dev.myfirst.0.sim_running: 1

# while true; do
    sysctl dev.myfirst.0.reg_sensor
    sleep 0.5
done
```

You should see the value oscillating. Default baseline is $0 \times 1000 = 4096$, amplitude is 64, so the value ranges from about 4096 to about 4160 and back, over several seconds.

Change the config to speed up the oscillation:

```
# sysctl dev.myfirst.0.reg_sensor_config_set=0x01000020
```

This sets interval to $0x0100 = 256$ ms and amplitude to $0x0020 = 32$. The sensor updates every 256 ms with a smaller range. Watch the output change.

Change it again to slow it down dramatically:

```
# sysctl dev.myfirst.0.reg_sensor_config_set=0x03e80100
```

This sets interval to 1000 ms and amplitude to 256. The sensor changes once per second, with a larger range.

Stop the simulation:

```
# sysctl dev.myfirst.0.sim_running=0    # (requires a writeable sysctl, added in the lab  
↪ examples)
```

The sensor value freezes. Re-enable:

```
# sysctl dev.myfirst.0.sim_running=1
```

It resumes. This lab exercises the autonomous-update path end to end.

17.10.2 Lab 2: Run a Single Command

Issue a command by hand and observe every register change.

```
# sysctl dev.myfirst.0.access_log_enabled=1  
# sysctl dev.myfirst.0.reg_delay_ms_set=200  
  
# echo -n "A" > /dev/myfirst0  
# sysctl dev.myfirst.0.access_log | head -20  
# sysctl dev.myfirst.0.access_log_enabled=0
```

The log should show:

1. Reads of STATUS (polling for BUSY to clear at command start).
2. A write to DATA_IN (the byte 'A' = 0x41).
3. A read of CTRL followed by a write of CTRL (setting the GO bit).
4. A write of CTRL (clearing the GO bit; the self-clearing logic).
5. Reads of STATUS (polling for DATA_AV).
6. An eventual transition where DATA_AV is set (from the simulation's callout).
7. A read of DATA_OUT (returning 0x41).
8. A write of STATUS (clearing DATA_AV).

Try to match each entry to the source code in `myfirst_sim.c` and `myfirst_write_cmd`. If any entry does not make sense, you have a gap to fill.

17.10.3 Lab 3: Stress the Command Path

Run many concurrent commands and verify correctness.

```
# sysctl dev.myfirst.0.reg_delay_ms_set=20
# sysctl dev.myfirst.0.cmd_successes      # note value

# for i in 1 2 3 4 5 6 7 8; do
    (for j in $(seq 1 50); do
        echo -n "X" > /dev/myfirst0
    done) &
done
# wait

# sysctl dev.myfirst.0.cmd_successes      # should have grown by 400
# sysctl dev.myfirst.0.cmd_errors         # should still be 0
# sysctl dev.myfirst.0.reg_op_counter     # should match the growth in cmd_successes
```

Eight writers, 50 commands each, 400 total. With 20 ms per command, the test runs in about 8 seconds (serialised at the driver lock). `cmd_successes` should grow by 400. `cmd_errors` should remain at zero (no faults are enabled). `reg_op_counter` should match the increment in `cmd_successes`.

17.10.4 Lab 4: Inject a Timeout Fault

Enable the timeout fault and observe the driver's recovery.

```
# sysctl dev.myfirst.0.cmd_data_timeouts      # note value
# sysctl dev.myfirst.0.cmd_recoveries          # note value

# sysctl dev.myfirst.0.reg_fault_mask_set=0x1  # TIMEOUT bit
# sysctl dev.myfirst.0.reg_fault_prob_set=10000 # 100%

# echo -n "X" > /dev/myfirst0
write: Operation timed out

# sysctl dev.myfirst.0.cmd_data_timeouts      # should have grown by 1
# sysctl dev.myfirst.0.cmd_recoveries          # should have grown by 1

# sysctl dev.myfirst.0.reg_status              # should be 1 (READY, BUSY cleared by
↪ recovery)

# sysctl dev.myfirst.0.reg_fault_mask_set=0
# sysctl dev.myfirst.0.reg_fault_prob_set=0
# echo -n "X" > /dev/myfirst0
```

```
# echo "write succeeded"
write succeeded
```

The first write times out; the driver recovers; a subsequent write succeeds. The counters confirm the sequence.

17.10.5 Lab 5: Inject an Error Fault

Run a small batch with an error fault at 25% probability.

```
# sysctl dev.myfirst.0.cmd_errors          # note value

# sysctl dev.myfirst.0.reg_fault_mask_set=0x4  # ERROR bit
# sysctl dev.myfirst.0.reg_fault_prob_set=2500  # 25%

# for i in $(seq 1 40); do
#     echo -n "X" > /dev/myfirst0 || echo "error on iteration $i"
# done

# sysctl dev.myfirst.0.cmd_errors          # should have grown by ~10

# sysctl dev.myfirst.0.reg_fault_mask_set=0
# sysctl dev.myfirst.0.reg_fault_prob_set=0
```

About 10 out of 40 iterations should report an error. The driver's error-counter should reflect the count exactly. The driver should remain usable after the test (subsequent commands succeed).

17.10.6 Lab 6: Inject Stuck-Busy and Watch the Driver Wait

Enable the stuck-busy fault, try a command, watch it time out, clear the fault, confirm recovery.

```
# sysctl dev.myfirst.0.reg_fault_mask_set=0x8  # STUCK_BUSY bit
# sleep 0.1                                     # let the busy callout assert

# sysctl dev.myfirst.0.reg_status
dev.myfirst.0.reg_status: 3                     # READY|BUSY

# sysctl dev.myfirst.0.cmd_rdy_timeouts         # note value

# echo -n "X" > /dev/myfirst0
write: Operation timed out                      # after sc->rdy_timeout_ms

# sysctl dev.myfirst.0.cmd_rdy_timeouts         # should have grown by 1
```

```
# sysctl dev.myfirst.0.reg_fault_mask_set=0
# sleep 0.2                                # let the busy callout stop

# sysctl dev.myfirst.0.reg_status
dev.myfirst.0.reg_status: 1                # READY only

# echo -n "X" > /dev/myfirst0
# echo "write succeeded"
write succeeded
```

The fault latches BUSY; the driver cannot issue commands; the driver times out on `wait_for_ready`. Clearing the fault and waiting briefly lets BUSY clear (via the command path, if any commands slipped through, or via the natural state of no outstanding command). The driver recovers.

17.10.7 Lab 7: Mixed Faults Under Load

Enable multiple faults at low probabilities, run a long stress test, analyse the results.

```
# sysctl dev.myfirst.0.reg_fault_mask_set=0x5    # TIMEOUT | ERROR
# sysctl dev.myfirst.0.reg_fault_prob_set=200    # 2%
# sysctl dev.myfirst.0.reg_delay_ms_set=10
# sysctl dev.myfirst.0.cmd_timeout_ms=50

# # Record starting counters.
# sysctl dev.myfirst.0 | grep -E 'cmd_|fault_' > /tmp/before.txt

# # 8 workers, 100 commands each.
# for i in 1 2 3 4 5 6 7 8; do
#     (for j in $(seq 1 100); do
#         echo -n "X" > /dev/myfirst0 2>/dev/null
#     done) &
# done
# wait

# # Record ending counters.
# sysctl dev.myfirst.0 | grep -E 'cmd_|fault_' > /tmp/after.txt
# diff /tmp/before.txt /tmp/after.txt

# # Clean up.
# sysctl dev.myfirst.0.reg_fault_mask_set=0
# sysctl dev.myfirst.0.reg_fault_prob_set=0
# sysctl dev.myfirst.0.reg_delay_ms_set=500
# sysctl dev.myfirst.0.cmd_timeout_ms=2000
```

The diff should show `cmd_successes` growing by about 784 (800 commands minus about 16 failures). `fault_injected` should grow by about 16. `cmd_data_timeouts` and `cmd_errors` should each grow by about 8. The exact numbers vary with each run (because of `arc4random`), but the proportions are stable.

Also check: `vmstat -m | grep myfirst` should show no growth in memory usage between before and after; the test was not supposed to leak.

17.10.8 Lab 8: Observe Sensor Updates During Heavy Command Load

The sensor callout and the command callout share the same mutex. A long-running load test might starve the sensor of updates if the lock is held constantly. Check this.

```
# sysctl dev.myfirst.0.reg_delay_ms_set=5          # very fast commands
# # Start a long load test in the background.
# (for i in $(seq 1 5000); do
#     echo -n "X" > /dev/myfirst0 2>/dev/null
# done) &

# # Meanwhile, poll the sensor.
# while kill -0 $! 2>/dev/null; do
#     sysctl dev.myfirst.0.reg_sensor
#     sleep 0.2
# done
```

Each polled sensor value should differ from the previous (the sensor callout is running). If the values freeze during the load (all identical for many reads), the lock is being held too long and the sensor is starving. A correct driver sees smoothly updating sensor values even under heavy command load, because the command path drops the lock during `pause_sbt`.

17.10.9 Lab 9: Build and Run the `hwsim2` Module

A stand-alone version of the Chapter 17 simulation, `hwsim2`, lives in the companion sources. Build and load it.

```
# cd examples/part-04/ch17-simulating-hardware/hwsim2-standalone
# make clean && make
# kldload ./hwsim2.ko

# # The hwsim2 module exposes a single register block with sensor
# # updates and command cycle; no cdev, just sysctls.
# sysctl dev.hwsim2.sensor
# sleep 1
# sysctl dev.hwsim2.sensor          # different value
# sysctl dev.hwsim2.do_command=0x12345678
```



```
# sleep 0.6
# sysctl dev.hwsim2.result           # 0x12345678

# kldunload hwsim2
```

The hwsim2 module is about 150 lines of C. Reading it in one sitting is a useful consolidation of Chapter 17's material.

17.10.10 Lab 10: Inject a Memory-Corruption Attack (Debug Kernel)

A deliberate break-and-observe exercise. Modify the simulation's sensor callback to write one byte past the end of the register block. This should fire the KASSERT on a debug kernel.

In `myfirst_sim.c`, temporarily add to `myfirst_sim_sensor_cb`:

```
/* DELIBERATE BUG for Lab 10. Remove after testing. */
CSR_WRITE_4(sc, 0x80, 0xdead); /* 0x80 is past the 64-byte region */
```

Rebuild and load. The kernel should panic within 100 ms (the sensor callback fires, the KASSERT catches the out-of-bounds offset, the panic string names offset 0x80).

Remove the bug. Rebuild. Verify the driver runs cleanly again.

This lab shows the value of bounds assertions in the hardware accessors: an out-of-bounds access fires immediately instead of silently corrupting unrelated memory. Production code should never remove these assertions.

17.11 Challenge Exercises

Challenges extend the Chapter 17 material with optional depth work. Each takes one to four hours and exercises judgment, not just keystrokes.

17.11.1 Challenge 1: Read-to-Clear INTR_STATUS

Chapter 17's `INTR_STATUS` is still RW, not RC. Implement RC semantics: when the driver reads `INTR_STATUS`, return the current value and then clear it. Add a hook in the accessor path that recognises RC registers and treats them specially. Update `HARDWARE.md`.

Think about: how does the driver ensure the read happens under the lock? What happens if a `sysctl` reads `INTR_STATUS` (inadvertently clearing state the driver needed)?

17.11.2 Challenge 2: Add a Queue of Commands

The current simulation rejects overlapping commands. Real devices often queue them. Implement a small command queue: if a command arrives while another is pending, add it to the queue; when the current command completes, start the next. Limit the queue to 4 entries.

Think about: how does the driver wait for queued commands? Should the driver have its own view of the queue, or rely on the simulation's view?

17.11.3 Challenge 3: Simulate a Sampling Rate Drift

The sensor callout fires at a fixed interval. Real sensors often drift: the interval shifts slightly over time because of temperature or voltage changes. Modify the sensor callback to add a small random perturbation to each interval (say, +/- 10%). Observe how this affects the driver.

Think about: does the drift matter for a driver that reads the sensor value? Does it matter for a driver that cares about sample timing (the first driver does not; a later driver might)?

17.11.4 Challenge 4: A Write-One-to-Clear `INTR_STATUS`

W1C semantics are a common variant of RC. Implement W1C for `INTR_STATUS`: writing a 1 to a bit clears that bit; writing a 0 has no effect; reads return the current value without side effects. Contrast with the RC implementation in Challenge 1.

Think about: which is more convenient for the driver (reading-to-clear or writing-to-clear)? Which is more defensive against accidental debug reads?

17.11.5 Challenge 5: Error Recovery Without Reset

The Chapter 17 driver's recovery path clears simulation-specific state directly. Real hardware cannot do this; the driver must use only register-level operations. Rewrite `myfirst_recover_from_stuck` to use only register writes (for example, a `CTRL.RESET` bit). Adjust the simulation to honour `CTRL.RESET`.

Think about: how does the driver wait for the reset to complete? Does the reset affect other registers (for example, does it clear the fault mask)?

17.11.6 Challenge 6: A User-Space Debugger

Write a small user-space program, `myfirstctl`, that opens `/dev/myfirst0`, issues a series of `ioctl`s to control the simulation, and displays the register state. Make it interactive: the user can type commands like `status`, `go X`, `delay 100`, `fault timeout 50%`.

Think about: how does the user-space program communicate with the driver (`ioctl`, `sysctl`, `read/write`)? What is a sensible command syntax?

17.11.7 Challenge 7: A DTrace-Driven Fault Injection

DTrace can provide dynamic probes. Add a DTrace probe at the top of `myfirst_sim_start_command` that takes a pointer to `sc` as its argument. Write a D script that, based on some user-specified condition (for example, "every 100th call"), sets `sc->sim->pending_fault = MYFIRST_FAULT_ERROR`. The script injects faults from user space without the kernel's fault-injection framework.

Think about: can DTrace safely modify kernel state like this? What are the limits of what DTrace can do?

17.11.8 Challenge 8: Simulate Two Devices

The current driver has a single device instance. Modify it so that two `/dev/myfirstN` files exist, each with its own simulation. Verify that operations on one device do not affect the other.

Think about: what changes in `attach`? How does the per-device `sysctl` tree work? Does the simulation need per-device random state?

17.12 Troubleshooting Reference

A quick reference for the problems Chapter 17's simulation is most likely to surface.

17.12.1 The sensor does not change

- `sim_running` is 0. Toggle it with the writeable `sysctl`.
- `SENSOR_CONFIG`'s `interval` field is zero. Set it to a positive value.
- The sensor callout is not scheduled. Confirm `myfirst_sim_enable` was called. Check `dmesg` for error messages.
- `sc->sim` is `NULL`. Confirm `myfirst_sim_attach` ran successfully.

17.12.2 Commands always time out

- `DELAY_MS` is set very high. Reduce it or increase the command timeout.
- The `FAULT_TIMEOUT` bit is set in `FAULT_MASK`. Clear it.
- The `FAULT_STUCK_BUSY` bit is set. Clear it and wait ~100 ms for the busy callout to stop re-asserting.
- The simulation is disabled (`sim_running=0`). Enable it.
- A previous command got stuck and was not recovered. Check `command_pending` (through the `sim sysctl`) and manually reset if needed.

17.12.3 WITNESS warns about lock order

- The detach path is acquiring locks in the wrong order. Compare the stack trace against `LOCKING.md`.
- A new code path is taking `sc->mtx` after a sleep lock. The ordering from Chapter 15 must be preserved.

17.12.4 Kernel panic on `kldunload`

- The simulation callouts were not drained before `free`. Check that `callout_drain` runs before `free(sim)`.
- The simulation state was freed while a callout was in flight. The drain is the fix.
- The hardware layer was freed before the simulation layer. The Chapter 17 ordering (`sim detach`, then `hw detach`) must be followed.

17.12.5 Memory usage grows over time

- `vmstat -m | grep myfirst` shows the driver's allocation size climbing. This is a leak. Check the fault-injection error paths; they often neglect to free something.
- The access log is allocated dynamically in some configurations. Confirm it is freed on detach.

17.12.6 The access log shows unexpected entries

- A callout is running on its own cadence; all Chapter 17 callouts are visible in the log. If the log shows callouts firing faster than expected, check the interval.
- A user-space tool (`watch -n 0.1 sysctl ...`) is polling very fast. Each `sysctl` access goes through the accessors.

17.12.7 Fault injection probability does not match observations

- `FAULT_PROB` is scaled as 0 to 10000. A value of 50 is 0.5%, not 50%. For 50%, use 5000.
- The fault only applies to operations that trigger `myfirst_sim_should_fault`. Not every operation is a candidate; sensor updates, for example, do not go through that path.

17.12.8 `sim_running` toggles ignored

- The `sysctl` handler may not call `myfirst_sim_enable` or `myfirst_sim_disable` under the lock. Confirm the handler acquires `sc->mtx` before delegating.

17.12.9 Commands succeed but take much longer than expected

- `cmd_timeout_ms` is set very high; any command that stalls makes the timeout budget huge.
- The fault mask is set with `STUCK_BUSY` and a successful command unexpectedly becomes a `wait_for_ready` stall until the next fault-clear.

17.12.10 Load tests are slower than expected

- Commands are serialised through the driver mutex. N writers at M commands each take NM `DELAY_MS` total time.
- Reducing `DELAY_MS` speeds up the simulation; reducing `cmd_timeout_ms` does not speed anything up unless timeouts are actually occurring.

17.13 Wrapping Up

Chapter 17 opened with a static register block from Chapter 16 and closes with a driver that behaves, end to end, like a driver talking to a functional device. The simulation is dynamic: a sensor register updates on its own, commands schedule delayed completions, `STATUS` bits change over time, and fault injection produces controlled failures that exercise the driver's error paths. The driver handles all of this through the register

vocabulary Chapter 16 taught, with the synchronisation discipline Part 3 built, and with the timing primitives Section 6 introduced.

What Chapter 17 deliberately did not do: real PCI (Chapter 18), real interrupts (Chapter 19), real DMA (Chapters 20 and 21). Each of those topics deserves its own chapter; each will extend the driver in specific ways while reusing the simulation framework where appropriate.

The version is 1.0-simulated. The file layout has grown: `myfirst.c`, `myfirst_hw.c`, `myfirst_hw.h`, `myfirst_sim.c`, `myfirst_sim.h`, `myfirst_sync.h`, `cbuf.c`, `cbuf.h`. The documentation has grown: `LOCKING.md`, `HARDWARE.md`, and the new `SIMULATION.md`. The test suite now exercises every simulation behaviour, every fault mode, and every error path.

17.13.1 A Reflection Before Chapter 18

A pause before the next chapter. Chapter 17 taught simulation as a technique that extends beyond learning. The patterns you practised here (callout-driven state changes, command-response protocols, fault injection, recovery paths) are patterns you will use throughout your driver-writing life. They apply as much to a production storage driver's test harness as to this tutorial driver. The simulation skill is permanent.

Chapter 17 also taught the discipline of thinking like hardware. Designing the register map, choosing access semantics, timing the responses, deciding what to latch and what to clear: these are decisions the hardware team makes for real devices, and a driver author who understands them reads datasheets differently. The next time you encounter a real device's datasheet, you will notice the decisions the designers made, and you will have a framework for evaluating whether the decisions were wise.

Chapter 18 shifts the scene entirely. The simulation goes away (temporarily); a real PCI device takes its place. The register block moves from `malloc(9)` memory to a PCI BAR. The accessors switch from `X86_BUS_SPACE_MEM` to `rman_get_bustag` and `rman_get_bushandle`. The driver's high-level code does not change at all, because the Chapter 16 abstraction is portable across the change. What does change is the device's behaviour: a real virtio device in a VM has its own protocol, its own timing, its own quirks. The patterns Chapter 17 taught are what let you handle all of that.

17.13.2 What to Do If You Are Stuck

A few suggestions if the Chapter 17 material feels dense.

First, re-read Section 3. The callout pattern is the foundation of every dynamic behaviour in the chapter. If the interaction between `callout_init_mtx`, `callout_reset_sbt`, `callout_stop`, and `callout_drain` feels opaque, everything downstream is opaque. Chapter 13 is the other good reference.

Second, run Lab 1 and Lab 2 by hand, one step at a time. Watching the sensor breathe and tracing a single command through the access log is where the simulation's behaviour becomes concrete.

Third, skip the challenges on first pass. The labs are calibrated for Chapter 17; the challenges assume the chapter's material is already solid. Come back to them after Chapter 18 if they feel out of reach now.

Fourth, open `myfirst_sim.c` and read the three callouts (sensor, command, busy) in order. Each is a self-contained piece of code that illustrates one aspect of the simulation's design. If you can explain each one to a colleague (or to a rubber duck), you have internalised Chapter 17's core.

Chapter 17's goal was to make the register block come alive. If it has, the rest of Part 4 will feel like a natural progression: Chapter 18 trades the simulation for real hardware, Chapter 19 adds real interrupts, Chapters 20 and 21 add DMA. Each chapter builds on what Chapter 17 established.

17.14 Bridge to Chapter 18

Chapter 18 is titled *Writing a PCI Driver*. Its scope is the real-hardware path that Chapter 17 deliberately did not take: a driver that probes a PCI bus, matches a real device by vendor and device ID, claims the device's BAR as a memory resource, maps it through `bus_alloc_resource_any`, and talks to the mapped region through the same `CSR_*` macros the Chapter 16 and Chapter 17 drivers already use.

Chapter 17 prepared the ground in four specific ways.

First, **you have a complete driver**. The Chapter 17 driver at `1.0-simulated` exercises every common protocol pattern: command-response, delayed completion, status polling, error recovery, timeout handling. Chapter 18 will swap the register backend without changing the driver's high-level logic. The swap is a one-function change in `myfirst_hw_attach`; everything else stays put.

Second, **you have a fault model**. The Chapter 17 fault-injection framework taught the driver to handle errors. Real PCI devices produce real errors: bus timeouts, uncorrectable ECC errors, power-state transitions, link drops. The discipline of handling simulated errors is what will let the Chapter 18 driver handle real ones.

Third, **you have a timing model**. Chapter 17 taught the reader when to use `DELAY(9)`, `pause_sbt(9)`, and `callout_reset_sbt(9)`. Real PCI devices have their own timing requirements, often documented in their datasheets. The discipline Chapter 17 built is what will let the driver respect those timings in Chapter 18.

Fourth, **you have a documentation habit**. `HARDWARE.md`, `LOCKING.md`, and `SIMULATION.md` are three living documents that the driver's contributors maintain. Chapter 18 will add a fourth: `PCI.md` or a similar document describing the specific PCI device the driver targets, its vendor and device ID, its BAR layout, and any quirks. The habit of documenting is established; Chapter 18 extends it.

Specific topics Chapter 18 will cover:

- The PCI subsystem in FreeBSD: `pci(4)`, the bus-device-function tuple, `pciconf -lv`.
- The probe and attach lifecycle: `DRIVER_MODULE`, vendor and device ID matching, the probe and attach methods.
- Resource allocation: `bus_alloc_resource_any`, the resource specification, `SYS_RES_MEMORY`, `RF_ACTIVE`.
- BAR mapping: how the PCI BAR becomes a `bus_space` region; `rman_get_bustag` and `rman_get_bushandle`.
- Enabling bus mastering: `pci_enable_busmaster` and when it is necessary.
- Attach-time initialisation and detach cleanup.
- Testing against a virtio device in a VM: the FreeBSD guest on `qemu` or `bhyve`, how to pass through a synthetic device, how to verify the driver attaches and behaves.

You do not need to read ahead. Chapter 17 is sufficient preparation. Bring your `myfirst` driver at `1.0-simulated`, your `LOCKING.md`, your `HARDWARE.md`, your `SIMULATION.md`, your WITNESS-enabled kernel, and your test kit. Chapter 18 starts where Chapter 17 ended.

A small closing reflection. Part 3 taught the synchronisation vocabulary and produced a driver that coordinated itself. Chapter 16 gave the driver a register vocabulary. Chapter 17 gave the register block dynamic behaviour

and the driver a fault model. Chapter 18 will retire the simulation and connect the driver to real silicon. Each step has built on the last; the driver that walks into Chapter 18 is a driver that is almost production-grade, needing only the real-bus glue that Chapter 18 provides.

The hardware conversation is deepening. The vocabulary is yours; the protocol is yours; the discipline is yours. Chapter 18 adds the last missing piece.

17.15 Reference: The Chapter 17 Simulation Cheat Sheet

A one-page summary of the Chapter 17 simulation API and the sysctls it exposes, for quick reference while coding.

17.15.1 Simulation API (in `myfirst_sim.h`)

Function	Purpose
<code>myfirst_sim_attach(sc)</code>	Allocate sim state, register callouts (not started).
<code>myfirst_sim_detach(sc)</code>	Drain callouts, free sim state.
<code>myfirst_sim_enable(sc)</code>	Start the callouts. Requires <code>sc->mtx</code> held.
<code>myfirst_sim_disable(sc)</code>	Stop the callouts (without drain).
<code>myfirst_sim_start_command(sc)</code>	Triggered by <code>CTRL.GO</code> write; schedules completion.
<code>myfirst_sim_add_sysctls(sc)</code>	Register simulation-specific sysctls.

17.15.2 Register Additions

Offset	Register	Access	Purpose
0x28	SENSOR	RO	Simulated sensor value, updated by callout.
0x2c	SENSOR_CONFIG	RW	Interval (low 16) and amplitude (high 16).
0x30	DELAY_MS	RW	Command processing delay in milliseconds.
0x34	FAULT_MASK	RW	Bitmask of enabled fault types.
0x38	FAULT_PROB	RW	Fault probability, 0 to 10000 (10000 = 100%).
0x3c	OP_COUNTER	RO	Count of commands processed.

17.15.3 CTRL Additions

Bit	Name	Purpose
9	MYFIRST_CTRL_GO	Start a command. Self-clearing.

17.15.4 Fault Modes

Bit	Name	Effect
0	MYFIRST_FAULT_TIMEOUT	Command never completes.
1	MYFIRST_FAULT_READ_1S	DATA_OUT returns 0xFFFFFFFF.
2	MYFIRST_FAULT_ERROR	STATUS.ERROR set instead of DATA_AV.
3	MYFIRST_FAULT_STUCK_BUSY	STATUS.BUSY latched on continuously.

17.15.5 New Sysctls

Sysctl	Type	Purpose
dev.myfirst.0.sim_running	RW	Enable/disable the simulation.
dev.myfirst.0.sim_sensor_baseline	RW	Baseline sensor value.
dev.myfirst.0.sim_op_counter_mirror	RO	Mirror of OP_COUNTER.
dev.myfirst.0.reg_delay_ms_set	RW	Writable DELAY_MS.
dev.myfirst.0.reg_sensor_config_set	RW	Writable SENSOR_CONFIG.
dev.myfirst.0.reg_fault_mask_set	RW	Writable FAULT_MASK.
dev.myfirst.0.reg_fault_prob_set	RW	Writable FAULT_PROB.
dev.myfirst.0.cmd_timeout_ms	RW	Command completion timeout.
dev.myfirst.0.rdy_timeout_ms	RW	Device-ready polling timeout.
dev.myfirst.0.cmd_successes	RO	Successful commands.
dev.myfirst.0.cmd_rdy_timeouts	RO	Ready-wait timeouts.
dev.myfirst.0.cmd_data_timeouts	RO	Data-wait timeouts.
dev.myfirst.0.cmd_errors	RO	Commands that reported an error.
dev.myfirst.0.cmd_recoveries	RO	Recovery invocations.
dev.myfirst.0.fault_injected	RO	Total faults injected.

17.16 Reference: Timing Primitive Cheat Sheet

Primitive	Cost	Cancelable	Appropriate Range
DELAY(us)	Busy-wait, full CPU	No	< 100 us
pause_sbt(..., sbt, ...)	Sleep, yields CPU	No (not interruptible in this form)	100 us - seconds
callout_reset_sbt(...)	Scheduled callback	Yes	fire-and-forget, periodic
cv_timedwait_sbt(...)	Sleep on condition	Yes (via cv_signal)	waits that can be shortened

Contexts where each is legal:

- DELAY: any (including filter interrupts, spin mutexes).
- pause_sbt: process context, spin locks not held.
- callout_reset_sbt: any (the callback runs in callout context).
- cv_timedwait_sbt: process context, specific mutex held.

17.17 Reference: Anatomy of a Simulation Callout

A template for adding a new simulation behaviour, modelled on the Chapter 17 sensor callout.

17.17.1 Step 1: Declare the callout in the sim state

```
struct myfirst_sim {
    /* ... existing fields ... */
    struct callout    my_new_callout;
    int               my_new_interval_ms;
};
```

17.17.2 Step 2: Initialise the callout at attach

```
/* In myfirst_sim_attach: */
callout_init_mtx(&sim->my_new_callout, &sc->mtx, 0);
sim->my_new_interval_ms = 200; /* default */
```

17.17.3 Step 3: Write the callback

```
static void
myfirst_sim_my_new_cb(void *arg)
{
    struct myfirst_softc *sc = arg;
    struct myfirst_sim *sim = sc->sim;

    MYFIRST_LOCK_ASSERT(sc);

    if (!sim->running)
        return;

    /* Do the work: update a register, signal a condition, ... */
    CSR_UPDATE_4(sc, MYFIRST_REG_SENSOR, 0, 0x100);

    /* Re-arm. */
    callout_reset_sbt(&sim->my_new_callout,
        sim->my_new_interval_ms * SBT_1MS, 0,
        myfirst_sim_my_new_cb, sc, 0);
}
```

17.17.4 Step 4: Start the callout at enable

```
/* In myfirst_sim_enable: */
callout_reset_sbt(&sim->my_new_callout,
    sim->my_new_interval_ms * SBT_1MS, 0,
    myfirst_sim_my_new_cb, sc, 0);
```

17.17.5 Step 5: Stop it at disable, drain at detach

```
/* In myfirst_sim_disable: */
callout_stop(&sim->my_new_callout);

/* In myfirst_sim_detach, after releasing the lock: */
callout_drain(&sim->my_new_callout);
```

17.17.6 Step 6: Document in SIMULATION.md

Add an entry to the callout-map table:

Callout	Interval	Purpose
-----	-----	-----
my_new_callout	my_new_interval_ms	... explain ...

Six steps. Each is mechanical. The pattern is the same for every simulation behaviour; once you have internalised the template, adding new behaviours is quick and reliable.

17.18 Reference: Test Scripts Inventory

A short catalogue of the test scripts that ship under `examples/part-04/ch17-simulating-hardware/labs/`.

Script	Purpose	Typical Runtime
<code>sim_sensor_oscillates.sh</code>	Confirm SENSOR value changes over time.	~10 s
<code>sim_command_cycle.sh</code>	Run 50 commands, verify OP_COUNTER matches.	~30 s
<code>sim_timeout_fault.sh</code>	Enable timeout fault; verify recovery.	~5 s
<code>sim_error_fault.sh</code>	Enable error fault; verify cmd_errors grows.	~10 s
<code>sim_stuck_busy_fault.sh</code>	Enable stuck-busy fault; verify timeout on ready.	~10 s
<code>sim_mixed_faults_under_load.sh</code>	50% random faults, 8 workers, 100 commands each.	~30 s
<code>sim_sensor_during_load.sh</code>	Verify sensor updates during heavy command load.	~30 s
<code>full_regression_ch17.sh</code>	All of the above plus Chapter 11-16 regression.	~3 minutes

Each script exits with a non-zero status if something unexpected happened. A successful `full_regression_ch17.sh` means every Chapter 17 behaviour has been validated and no regressions have been introduced.

17.19 Reference: An Honest Accounting of Chapter 17's Simplifications

A chapter that teaches a slice of a large topic inevitably simplifies. For honesty with the reader, a catalogue of what Chapter 17 simplified and what the full story looks like.

17.19.1 The Callout's Mutex Interaction

Chapter 17 uses `callout_init_mtx` so every simulation callback runs with `sc->mtx` held. This is the simplest and safest pattern but not the only one. Real drivers sometimes use `callout_init` (no associated mutex) and acquire the lock inside the callback; this gives finer control over locking but is more error-prone. Some real drivers use `callout_init_rm` (for reader-writer locks) or even `callout_init_mtx` with a spin mutex for cases where the callback must run in hardirq context.

The full story: FreeBSD's callout API supports several lock styles, each suited to a different concurrency model. The `callout_init_mtx` choice in Chapter 17 is pedagogically clean; production drivers pick the style that

matches their specific locking graph. `/usr/src/sys/dev/e1000/if_em.c` uses `callout_init_mtx` for its tick callout and `callout_init` for its spinlock-guarded DMA-polling path, illustrating the difference.

17.19.2 The Fault-Injection Framework

Chapter 17's fault injection is limited to four modes, a probability, and a simple `should_fault` check at the start of each operation. Real fault-injection frameworks (FreeBSD's own `fail(9)`, for example) support a richer vocabulary: callsite-specific injection, probability curves that change over time, deterministic injection for the Nth call, and injection paths that can be triggered from user space by name.

The full story: the `fail(9)` framework in `/usr/src/sys/kern/kern_fail.c` implements a production-grade fault-injection system. Chapter 17's framework is a simplified version that focuses on the simulation's specific needs. A challenge at the end of the chapter invites the reader to replace the chapter's ad-hoc framework with a `fail(9)`-based one.

17.19.3 The Random Number Source

Chapter 17 uses `arc4random_uniform(10000)` as the fault-probability decision source. This is non-deterministic: two runs of the same test will produce different fault patterns. For reproducible testing, a driver author would use a deterministic source (a simple LCG seeded by a fixed value, or a counter) and expose the seed through a `sysctl`.

The full story: reproducible fault injection is important for regression testing. A driver that passes all faults at seed `0x12345` one day and fails the same seed next week has a bug that was not there before. Chapter 17's non-determinism is acceptable for teaching but insufficient for serious regression suites.

17.19.4 The Command Queue

Chapter 17 allows only one command in flight at a time. Real devices often queue commands; a network driver might have thousands of in-flight descriptors. The single-command restriction simplifies the simulation's state machine and the driver's locking story, but it limits the throughput of the tutorial driver.

The full story: a queued command model requires per-command state (status, arrival time, result), a queue data structure with its own synchronisation, and a driver pattern that handles command coalescing, batch completion, and out-of-order completion. Chapter 20's DMA descriptor rings introduce the pattern for DMA-based devices; real command-queue drivers extend it to non-DMA contexts too.

17.19.5 The Sensor Model

Chapter 17's sensor produces a triangle-wave value using a simple arithmetic formula. Real sensors produce values influenced by physical noise, temperature drift, sampling jitter, and device-specific non-linearities. A more realistic simulation would include Gaussian noise, a slow drift term, and occasional outliers.

The full story: sensor modelling is a discipline of its own, and real driver testing for sensor devices often uses captured real-world data played back through a simulation layer. Chapter 17's arithmetic sensor is pedagogically sufficient; a production simulation would be substantially richer.

17.19.6 The Error Taxonomy

Chapter 17 distinguishes four fault types: timeout, read-1s, error, stuck-busy. Real hardware produces a much wider range of failure modes, including partial writes, stuck interrupt bits, inverted fields, off-by-one register values, and transient electrical noise. The four chapter faults are a starting point.

The full story: FreeBSD driver authors who work with specific device families develop much more elaborate fault taxonomies over time. The `e1000` driver's error-handling code is a few hundred lines of its own, handling dozens of distinct failure modes. Chapter 17 teaches the pattern; applying it to real hardware produces a richer taxonomy by necessity.

17.19.7 The Lack of Interrupts

Chapter 17's driver polls `STATUS` by reading the register in a loop (with `pause_sbt` between polls). Real drivers use interrupts: the device raises an interrupt when `STATUS` changes, the driver's interrupt handler wakes the waiting thread, the thread reads the new state. Polling is wasteful compared to interrupts.

The full story: Chapter 19 introduces interrupts. The Chapter 17 driver will get an `INTR_STATUS`-driven wake-up path in Chapter 19, and the polling loops will become much shorter (wake-once rather than wake-every-millisecond). Chapter 17's polling is a pedagogical stepping stone.

17.19.8 The Timing Precision

Chapter 17's callouts run at approximately the configured interval, with precision on the order of 1 ms (the kernel's default timer resolution). Real devices sometimes require sub-microsecond timing. A driver that depends on such timing must use `DELAY(9)` for short waits and cannot rely on callout precision.

The full story: high-precision timing in drivers requires platform-specific mechanisms (the TSC, the ACPI-PM timer, the HPET, or hardware-specific timers on embedded platforms). Chapter 17's millisecond-scale simulation does not exercise sub-millisecond paths; a driver for a real device with such requirements would use `DELAY` or a hardware-assisted mechanism.

17.19.9 Summary

Chapter 17 is a teaching simulation. Every simplification it makes is deliberate, named, and picked up by a later chapter or by a real-driver extension. The patterns Chapter 17 teaches are the patterns every later chapter extends; the discipline Chapter 17 builds is the discipline every later chapter relies on. The chapter is short of the full simulation story on purpose. Subsequent chapters and real-world practice fill in the rest.

17.20 Reference: A Glossary of Chapter 17 Terms

A short glossary of terms introduced or used extensively in Chapter 17.

Autonomous update. A change to a register that happens without the driver initiating it. In Chapter 17, the sensor callout produces autonomous updates.

Command cycle. The full sequence of events from “driver decides to issue a command” to “command completes and the driver observes the result”. In Chapter 17: write `DATA_IN`, set `CTRL.GO`, wait for `STATUS.DATA_AV`, read `DATA_OUT`.

Callout. A FreeBSD primitive that schedules a callback to run at a future time. The callback runs on a callout thread. Created with `callout_init_mtx`, armed with `callout_reset_sbt`, cancelled with `callout_stop`, drained with `callout_drain`.

Fault injection. The deliberate introduction of a failure condition into the system for testing purposes. Chapter 17’s framework supports four fault modes configurable through `FAULT_MASK` and `FAULT_PROB`.

Latched. A register bit that, once set, remains set until explicitly cleared. Error bits in real devices are typically latched. Chapter 17’s `STATUS.ERROR` is latched.

Polling. Repeatedly reading a register to detect a change. Chapter 17’s `myfirst_wait_for_bit` polls `STATUS` with 1 ms sleep between reads.

Pending command. A command that has been started but not yet completed. Chapter 17’s simulation tracks this with `sim->command_pending`.

Protocol. The rules the driver must follow when talking to the device. In Chapter 17: write `DATA_IN` before `CTRL.GO`; wait for `STATUS.DATA_AV` before reading `DATA_OUT`; clear `DATA_AV` after reading.

Read-to-clear (RC). A register semantic where reading the register returns its current value and then clears it. `INTR_STATUS` on many real devices is RC.

Simulation backend. The kernel code that provides the simulated device’s behaviour. In Chapter 17, this is `myfirst_sim.c`.

Side effect. Behaviour that a register access triggers beyond the obvious read or write. `CTRL.RESET` writes have a side effect (they reset the device). `INTR_STATUS` reads have a side effect (they clear pending bits, under RC semantics).

Sticky. Similar to latched. A bit that persists until deliberately cleared.

Timeout. A bounded wait that returns an error if the expected event does not occur in time. Chapter 17’s `wait_for_bit` timeouts after the configured number of milliseconds.

17.21 Reference: The Chapter 17 Driver Diff Summary

A compact summary of what Chapter 17 changes in the driver, for readers who want to see at a glance how the driver evolved from 0.9-mmio to 1.0-simulated.

17.21.1 New files

- `myfirst_sim.c` (about 300 lines).
- `myfirst_sim.h` (about 50 lines).
- `SIMULATION.md` (about 200 lines).

17.21.2 Modified files

- `myfirst.c`: added `myfirst_sim_attach` and `myfirst_sim_detach` calls; added `myfirst_write_cmd`, `myfirst_sample_cmd`, `myfirst_wait_for_bit` helpers; added timeout fields to `softc`; added stats counters.
- `myfirst_hw.h`: added new register offsets (0x28 through 0x3c) and bit masks; added `MYFIRST_CTRL_GO`; added fault constants.
- `myfirst_hw.c`: no change to the accessor code; `myfirst_ctrl_update` extended to intercept `GO`.
- `HARDWARE.md`: added the new registers, the new `CTRL.GO` bit, the dynamic-behaviour notes.
- `LOCKING.md`: added the simulation detach step to the ordered detach sequence.
- `Makefile`: added `myfirst_sim.c` to `SRCS`.

17.21.3 Line count evolution

The file sizes at each stage (approximate):

Stage	<code>myfirst.c</code>	<code>myfirst_hw.c</code>	<code>myfirst_sim.c</code>	Total
Chapter 16 Stage 4 (start)	650	380	(not yet)	1030
Chapter 17 Stage 1	680	400	120	1200
Chapter 17 Stage 2	800	400	150	1350
Chapter 17 Stage 3	820	410	180	1410
Chapter 17 Stage 4	840	410	250	1500
Chapter 17 Stage 5 (final)	800	400	300	1500

The driver grew from roughly 1030 lines to 1500 lines over the chapter. About 470 lines of new code, split roughly 150 in `myfirst.c`, 20 in `myfirst_hw.c`, and 300 in the new `myfirst_sim.c`. The Stage 5 refactor slightly reduced `myfirst.c` by moving some helpers into `myfirst_sim.c`.

17.21.4 Behavioural additions

- Sensor autonomously updates every 100 ms.
- `CTRL.GO` triggers a command that completes after `DELAY_MS` milliseconds.
- `OP_COUNTER` increments on each command.
- `FAULT_MASK` and `FAULT_PROB` control injected failures.
- Four fault modes (timeout, read-1s, error, stuck-busy).
- Per-outcome statistics counters.
- Configurable command and ready timeouts.
- Command cycle integrated into the write and read paths.

17.21.5 Test additions

- Eight new test scripts under `labs/`.

- An updated `full_regression.sh` that includes the new scripts.
- Expected runtime of the full regression: about 3 minutes.

17.22 Reference: Reading a Callout-Driven Driver

A guided walk through a real FreeBSD driver whose design is similar to Chapter 17's simulation: a driver whose periodic work is scheduled through callouts, and whose main job is to respond to state changes. The driver is `/usr/src/sys/dev/led/led.c`, the LED pseudo-device driver.

Open it in one terminal. Follow along.

17.22.1 The LED Driver's Structure

`led.c` is about 400 lines. Its job is to expose an interface for other drivers to announce an LED device through `/dev/led/NAME`, and for user space to blink the LED by writing patterns. The actual hardware work is delegated to whichever driver registered the LED; `led.c` does the scheduling and the pattern matching.

The key data structures:

- `struct led_softc`: per-LED state, including a `struct callout led_ch`, a `struct sbuf *spec` holding the blink pattern, and a function pointer to the driver's "set state" callback.
- `struct mtx led_mtx`: a global mutex protecting the list of all LEDs.
- A linked list of all LEDs.

17.22.2 The Callout Callback

The blink-pattern interpreter runs in the callout callback:

```
static void
led_timeout(void *p)
{
    struct ledsc *sc = p;
    char c;
    int count;

    if (sc->spec == NULL || sc->ptr == NULL || sc->count == 0) {
        /* no pattern; stop blinking */
        sc->func(sc->private, sc->on);
        return;
    }

    c = *(sc->ptr)++;
    if (c == '.') {
        /* Pattern complete, restart. */
        sc->ptr = sbuf_data(sc->spec);
    }
}
```



```

        c = *(sc->ptr)++;
    }

    if (c >= 'a' && c <= 'j') {
        sc->func(sc->private, 0);    /* LED off */
        count = (c - 'a') + 1;
    } else if (c >= 'A' && c <= 'J') {
        sc->func(sc->private, 1);    /* LED on */
        count = (c - 'A') + 1;
    } else {
        count = 1;
    }

    callout_reset(&sc->led_ch, count * hz / 10, led_timeout, sc);
}

```

(The code is lightly abridged from the real source for presentation.)

17.22.3 The Pattern

Look at the shape of the callback.

First, it checks whether there is anything to do. If `spec == NULL`, the pattern has been cleared; the function turns the LED off (to a known state) and returns without re-arming. This is exactly the `if (!sim->running) return; pattern` in Chapter 17’s callouts.

Second, it advances through the pattern buffer one character at a time. The pattern language uses lowercase letters ‘a’ through ‘j’ to mean “LED off for 100 ms through 1000 ms” and uppercase ‘A’ through ‘J’ for “LED on”. This is a small state machine driven by the callout.

Third, it calls `sc->func(sc->private, state)` to actually toggle the LED. This is the hardware work, delegated to whichever driver registered the LED. The `led_timeout` callback does not know whether the LED is a GPIO pin, an I2C-attached LED controller, or a message to a remote device; it just calls the function pointer.

Fourth, it re-arms the callout with an interval determined by the current pattern character. This is Chapter 17’s `callout_reset_sbt pattern`, though `led.c` uses the older `callout_reset` with tick-based intervals.

17.22.4 Lessons for Chapter 17

The `led.c` driver illustrates several Chapter 17 patterns:

- A callout carries a state machine forward one step per invocation.
- The callout re-arms itself with a variable interval based on the current state.
- A “no work to do” branch returns without re-arming; the detach path relies on this to eventually drain.
- The callback uses a function pointer for delegated work, keeping the callout logic separate from the hardware-specific work.

Chapter 17's simulation uses the same patterns in slightly different clothes. The sensor callback updates a register; the command callback triggers a completion; the busy callback re-asserts a status bit. Each is a small state machine advanced by the callout.

17.22.5 An Exercise

Read `led.c` start to finish. It is one of the shortest drivers in `/usr/src/sys/dev/` and illustrates a clean callout-driven design. After reading, write a one-paragraph summary explaining how `led_timeout` interacts with `led_drvinit`, `led_destroy`, and `led_write`. If you can articulate the relationship, you have internalised the callout-driven-driver pattern.

17.23 Reference: The Difference Between “In Simulation” and “In Reality”

Chapter 17 simulates a device. A natural question is: how different is the simulation from a real device, and what are the specific ways the driver would need to change when the two are swapped?

A short comparison.

17.23.1 What the simulation and reality have in common

- The register map: the same offsets, the same widths, the same access types.
- The CSR macros: `CSR_READ_4`, `CSR_WRITE_4`, `CSR_UPDATE_4` expand to the same `bus_space_*` calls.
- The driver's command-cycle logic: write `DATA_IN`, set `CTRL.GO`, wait for `DATA_AV`, read `DATA_OUT`.
- The locking discipline: `sc->mtx` protects register access in both cases.
- The statistics: `cmd_successes`, `cmd_errors`, and so on track real events either way.

17.23.2 What differs

- The tag and handle: simulation uses `X86_BUS_SPACE_MEM` and a `malloc(9)` address; reality uses the tag and handle that `rman_get_bustag` and `rman_get_bushandle` return for an allocated PCI BAR.
- The register-block lifetime: simulation's lives from `myfirst_sim_attach` to `myfirst_sim_detach`; reality's lives from `bus_alloc_resource_any` to `bus_release_resource`.
- The autonomous behaviour: simulation produces it through callouts; reality produces it through device internal logic. The driver does not see the difference.
- The fault modes: simulation's faults are deliberate and controlled; reality's faults happen when they happen.
- The timing: simulation's `DELAY_MS` is a register; reality's timing is determined by the device's design and cannot be changed.
- The `CTRL.GO` self-clear: simulation's is implemented by the write intercept; reality's is implemented in silicon. The driver expects the same behaviour.
- The error recovery: simulation's `myfirst_recover_from_stuck` directly manipulates the simulation state; reality's recovery path uses only register operations (typically a reset).

The list is short. That is the point of the abstraction: most of the driver is identical, and the parts that are not are well-localised.

17.23.3 The Chapter 18 Change

When Chapter 18 replaces the simulation with real PCI, the changes to the driver are:

1. `myfirst_hw_attach` changes: instead of `malloc`-ing the register block, it calls `bus_alloc_resource_any` to allocate the PCI BAR.
2. `myfirst_hw_detach` changes: instead of `free`-ing the register block, it calls `bus_release_resource` to release the BAR.
3. The simulation files (`myfirst_sim.c`, `myfirst_sim.h`) are no longer built; the Makefile drops them.
4. The driver's probe logic gains a probe method that matches on the real device's vendor and device ID.
5. The `DRIVER_MODULE` registration changes from a pseudo-device style to a PCI attachment style.

Everything else (the command cycle, the statistics, the locking, the documentation) stays the same. The discipline Chapter 17 built is what makes this possible.

17.24 Reference: When Simulation Is Not Enough

A balanced view of simulation's limits. Chapter 17 teaches simulation as a primary technique, but it is not a substitute for every kind of testing. Several scenarios require real hardware or a hypervisor-backed virtual device.

17.24.1 When the bug is in the real-silicon timing

A race condition that manifests only at specific bus clock ratios, specific memory traffic patterns, or specific device firmware revisions cannot be reproduced in simulation. The simulation runs at kernel-thread speed, not at hardware speed, and its timing is dictated by the kernel's scheduler, not by the bus fabric.

17.24.2 When the bug is in the platform-specific code

A driver that needs to handle IOMMU remapping, specific cache attributes, or architecture-specific memory barriers cannot be fully validated in simulation. The simulation's kernel memory has different cache behaviour than real device memory, and the architecture-specific paths are exercised only when real device memory is involved.

17.24.3 When the bug depends on actual hardware resources

A driver that needs to configure a PCIe link width, manage power states, or interact with the device's firmware cannot be validated in simulation. The simulation has no PCIe link, no power states, and no firmware.

17.24.4 When the bug is in the test itself

A test that exercises the simulation in a way that does not correspond to how the real device is used produces false confidence. The simulation might pass the test; the real device might still fail under realistic use. Testing against simulation must be paired with testing against reality whenever reality is available.

17.24.5 What to do about it

The right approach is layered testing. Simulation catches protocol bugs, lock-order bugs, error-handling bugs, and the majority of logic errors. Hypervisor-backed virtual devices (virtio in bhyve or qemu) catch PCI-specific bugs, bus-mastering issues, and some timing issues. Real hardware on a lab system catches the last-mile bugs that nothing else can reproduce.

For Chapter 17 the focus is simulation. For Chapter 18 the focus is hypervisor-backed PCI. Later chapters on DMA and interrupts will introduce additional testing surfaces. Real-hardware testing on a dedicated lab system is a discipline the working driver author develops over time; this book can give you the vocabulary but not the hardware.

17.25 Reference: Further Reading

If you want to go deeper into the topics Chapter 17 touched, the following are good next steps.

17.25.1 FreeBSD manual pages

- `callout(9)`: the callout API in full.
- `pause(9)` and `pause_sbt(9)`: the sleep primitives in detail.
- `arc4random(9)`: the pseudo-random number generator.
- `bus_space(9)`: revisit with the simulation lens.
- `fail(9)`: the production fault-injection framework.

17.25.2 Source files worth reading

- `/usr/src/sys/dev/led/led.c`: a callout-driven pseudo-device driver. Short, readable, and illustrative.
- `/usr/src/sys/dev/random/random_harvestq.c`: a callout-driven harvest queue. More complex than `led.c` but with a similar pattern.
- `/usr/src/sys/kern/kern_fail.c`: the `fail(9)` framework. About 1500 lines but highly modular.
- `/usr/src/sys/kern/kern_timeout.c`: the callout subsystem implementation. Read when you want to understand how `callout_reset_sbt` actually works.

17.25.3 Real drivers with callouts similar to Chapter 17's

- `/usr/src/sys/dev/ale/if_ale.c`: `ale_tick` is a callout that periodically checks link state. Similar pattern to the Chapter 17 sensor callout.
- `/usr/src/sys/dev/e1000/if_em.c`: `em_local_timer` is a callout that updates statistics and handles watchdog events. Slightly more elaborate than the sensor callout.
- `/usr/src/sys/dev/iwm/if_iwm.c`: uses multiple callouts for different protocol state machines. An advanced but educational example.

17.25.4 Related reading on hardware simulation

- The FreeBSD Handbook chapter on emulation (for understanding how virtio and bhyve relate to in-kernel simulation).
- The bhyve(8) manual page, for insight into hypervisor-backed virtual devices that will matter in Chapter 18.

17.26 Reference: A Worked Example: The Full myfirst_sim.h

The complete myfirst_sim.h as it stands at the end of Chapter 17, for quick reference. The source is also in `examples/part-04/ch17-simulating-hardware/stage5-final/myfirst_sim.h`.

```
/*-
 * SPDX-License-Identifier: BSD-2-Clause
 *
 * Copyright (c) 2026 Edson Brandi
 *
 * myfirst_sim.h -- Chapter 17 simulation API.
 *
 * The simulation layer turns the Chapter 16 static register block into
 * a dynamic device: autonomous sensor updates, command-triggered
 * delayed completions, read-to-clear semantics for INTR_STATUS, and a
 * fault-injection framework. The API is small; most of the simulation
 * is in myfirst_sim.c.
 */

#ifndef _MYFIRST_SIM_H_
#define _MYFIRST_SIM_H_

#include <sys/callout.h>
#include <sys/stdbool.h>

struct myfirst_softc;

/*
 * Simulation state. One per driver instance. Allocated in
 * myfirst_sim_attach, freed in myfirst_sim_detach.
 */
struct myfirst_sim {
    /* The three simulation callouts. */
    struct callout    sensor_callout;
    struct callout    command_callout;
    struct callout    busy_callout;
}
```

```
/* Last scheduled command's data. Saved so command_cb can
 * latch DATA_OUT when it fires. */
uint32_t          pending_data;

/* Saved fault state for this command. Set in start_command,
 * consumed by command_cb. */
uint32_t          pending_fault;

/* Whether a command is currently in flight. */
bool              command_pending;

/* Baseline sensor value; the sensor callout oscillates
 * around this. */
uint32_t          sensor_baseline;

/* Counter used by the sensor oscillation algorithm. */
uint32_t          sensor_tick;

/* Local operation counter; mirrors OP_COUNTER register. */
uint32_t          op_counter;

/* Whether the simulation callouts are running. Checked by
 * every callout before doing work. */
bool              running;
};

/*
 * API. All functions assume sc->sim is valid (that is,
 * myfirst_sim_attach has been called successfully) unless noted
 * otherwise.
 */

/*
 * Allocate and initialise the simulation state. Registers callouts
 * with sc->mtx. Does not start the callouts; call _enable for that.
 * Returns 0 on success, an errno on failure.
 */
int myfirst_sim_attach(struct myfirst_softc *sc);

/*
 * Drain all simulation callouts, free the simulation state. Safe to
 * call with sc->sim == NULL. The caller must not hold sc->mtx (this
 * function sleeps in callout_drain).
 */
```

```

void myfirst_sim_detach(struct myfirst_softc *sc);

/*
 * Start the simulation callouts. Requires sc->mtx held.
 */
void myfirst_sim_enable(struct myfirst_softc *sc);

/*
 * Stop the simulation callouts. Does not drain (that is _detach's
 * job). Requires sc->mtx held.
 */
void myfirst_sim_disable(struct myfirst_softc *sc);

/*
 * Start a command. Called from the driver when CTRL.GO is written.
 * Reads DATA_IN and DELAY_MS, schedules the command completion
 * callout. Rejects overlapping commands. Requires sc->mtx held.
 */
void myfirst_sim_start_command(struct myfirst_softc *sc);

/*
 * Register simulation-specific sysctls on the driver's sysctl tree.
 * Called from myfirst_attach after sc->sysctl_tree is established.
 */
void myfirst_sim_add_sysctls(struct myfirst_softc *sc);

#endif /* _MYFIRST_SIM_H */

```

17.27 Reference: A Comparison with Chapter 16 Patterns

A side-by-side comparison of where Chapter 17 extends Chapter 16 and where it introduces genuinely new material.

Pattern	Chapter 16	Chapter 17
Register access	CSR_READ_4, etc.	Same API, unchanged
Access log	Introduced	Reused, extended with fault-injection entries
Lock discipline	sc->mtx around each access	Same, plus callouts via callout_init_mtx
File layout	myfirst_hw.c added	myfirst_sim.c added
Register map	10 registers, 40 bytes	16 registers, 60 bytes (all in same 64-byte allocation)

Pattern	Chapter 16	Chapter 17
CTRL bits	ENABLE, RESET, MODE, LOOPBACK	Same, plus GO (bit 9)
STATUS bits	READY, BUSY, ERROR, DATA_AV	Same, but dynamically changed by callouts
Callouts	One (reg_ticker_task as a task)	Three (sensor, command, busy)
Timeouts	Not applicable	Introduced (cmd_timeout_ms, rdy_timeout_ms)
Error recovery	Minimal	Full recovery path
Statistics	None	Per-outcome counters
Fault injection	None	Four fault modes
HARDWARE.md	Introduced	Extended
LOCKING.md	Extended from Ch15	Extended
SIMULATION.md	Not present	Introduced

Chapter 17 builds on Chapter 16 without breaking anything. Every Chapter 16 capability is preserved; every new capability is added. The driver at 1.0-simulated is a strict superset of the driver at 0.9-mmio.

17.28 Reference: A Closing Note on Simulation Philosophy

A paragraph to close the chapter with, which is worth returning to after you have worked through the labs.

Simulation is, at its heart, an act of modelling. You take a system you do not fully control (a real device) and build a smaller system you do control (a simulation) that behaves similarly. The simulation is never perfect. Its value comes not from matching the real system in every detail, but from preserving the properties that matter for the question you are asking.

For Chapter 17 the property that matters is protocol correctness: does the driver handle the command cycle, the timing, the error cases, the recovery? A simulation that preserves protocol correctness is a simulation that earns its keep, even if it gets every timing detail slightly wrong.

For later chapters and for your own future work, the properties that matter may be different. A performance test needs a simulation that preserves throughput behaviour. A power test needs a simulation that preserves idle and active states. A security test needs a simulation that can produce adversarial inputs.

The skill Chapter 17 teaches is not “how to simulate this particular device”. It is “how to identify what a simulation must preserve, and how to build one that preserves it”. That skill is transferable, and it is the skill that will serve you across every driver you write.

18 Writing a PCI Driver

18.1 Reader Guidance & Outcomes

Chapter 17 ended with a driver that looked like a real device from the outside and behaved like one from the inside. The `myfirst` module at version `1.0-simulated` carries a register block, a `bus_space(9)`-based accessor layer, a simulation backend with callouts that produce autonomous state changes, a fault-injection framework, a command-response protocol, statistics counters, and three living documentation files (`HARDWARE.md`, `LOCKING.md`, `SIMULATION.md`). Every register access in the driver still goes through `CSR_READ_4(sc, off)`, `CSR_WRITE_4(sc, off, val)`, and `CSR_UPDATE_4(sc, off, clear, set)`. The hardware layer (`myfirst_hw.c` and `myfirst_hw.h`) is a thin wrapper that produces a tag and a handle, and the simulation layer (`myfirst_sim.c` and `myfirst_sim.h`) is what makes those registers come alive. The driver itself does not know whether the register block is real silicon or a `malloc(9)` allocation.

That ambiguity is the gift Chapter 16 and Chapter 17 gave us, and Chapter 18 is where we finally cash it in. The driver will now meet real PCI hardware. The register block will not come from the kernel's heap any more; it will come from a device's Base Address Register, assigned at boot by the firmware, mapped by the kernel into a virtual address range with device-memory attributes, and handed to the driver as a `struct resource *`. The accessor layer does not change. A compile-time switch keeps the Chapter 17 simulation available as a separate build for readers without a PCI test environment; on the PCI build the simulation callouts do not run, so they cannot accidentally write into the real device's registers. What changes is the point at which the tag and handle originate: instead of being produced by `malloc(9)`, they will be produced by `bus_alloc_resource_any(9)` against a PCI child in the newbus tree.

Chapter 18's scope is precisely this transition. It teaches what PCI is, how FreeBSD represents a PCI device in the newbus tree, how a driver matches a device by vendor and device ID, how BARs appear in configuration space and become `bus_space` resources, how attach-time initialisation proceeds against a real BAR without disturbing the device, and how detach unwinds everything in reverse order. It covers the configuration-space accessors `pci_read_config(9)` and `pci_write_config(9)`, the capability walkers `pci_find_cap(9)` and `pci_find_extcap(9)`, and a brief introduction to PCIe Advanced Error Reporting so the reader knows where it lives without being asked to handle it yet. It ends with a small but significant refactor that splits the new PCI-specific code into its own file, versions the driver as `1.1-pci`, and runs the full regression pass against both the simulation build and the real PCI build.

Chapter 18 deliberately narrows itself to the probe-attach dance and what feeds it. Real interrupt handlers through `bus_setup_intr(9)`, filter plus `ithread` composition, and the rules for what a handler may and may not do are Chapter 19. MSI and MSI-X, along with the richer PCIe capabilities they expose, are Chapter 20. Descriptor rings, scatter-gather DMA, cache coherence around device writes, and the full `bus_dma(9)` story are Chapters 20 and 21. Configuration-space quirks on specific chipsets, power-management state machines during suspend and resume, and SR-IOV are later chapters. The chapter stays inside the ground it can cover well and hands off explicitly when a topic deserves its own chapter.

The layers of Part 4 stack. Chapter 16 taught the vocabulary of register access; Chapter 17 taught how to think like a device; Chapter 18 teaches how to meet a real one. Chapter 19 will teach you how to react to what the device says, and Chapters 20 and 21 will teach you how to let the device reach directly into RAM. Each layer depends on the one before it. Chapter 18 is your first encounter with the newbus tree as something more than an abstract diagram, and the disciplines Part 3 built are what keep the encounter honest.

18.1.1 Why the PCI Subsystem Earns a Chapter of Its Own

At this point you may be asking why the PCI subsystem needs a chapter of its own. The simulation already gave us registers; the real hardware will give us the same registers. Why not simply say “call `bus_alloc_resource_any`, pass the returned handle to `bus_read_4`, and proceed”?

Two reasons.

The first is that the PCI subsystem is the most widely used bus in modern FreeBSD, and the newbus conventions around it are the conventions every other bus driver imitates. A reader who understands the PCI probe-attach dance can read the ACPI attach dance, the USB attach dance, the SD card attach dance, and the virtio attach dance without retraining. The patterns differ in detail, but the shape is PCI’s. Spending a whole chapter on the canonical bus is spending a chapter on the pattern every bus borrows from.

The second is that PCI introduces concepts no earlier chapter prepared you for. Configuration space is a second address space per device, separate from the BARs themselves, where the device advertises what it is and what it needs. Vendor and device IDs are a sixteen-bit plus sixteen-bit tuple that the driver matches against a table of supported devices. Subvendor and subsystem IDs are a second-level tuple that disambiguates cards built around a common chipset by different vendors. Class codes let a driver match broad categories (any USB host controller, any UART) when a device-specific table would be too narrow. BARs exist in configuration space as thirty-two-bit or sixty-four-bit addresses the driver never directly dereferences. PCI capabilities are a linked list of extra metadata the driver reads at attach time. Each of these is new vocabulary; each of them is the reason Chapter 18 is not a single section bolted onto Chapter 17.

The chapter also earns its place by being the chapter where the `myfirst` driver gains its first real bus child. Until now, the driver has lived as a kernel module with a single implicit instance, attached by hand through `kldload` and detached by `kldunload`. After Chapter 18, the driver will be a proper PCI bus child, enumerated by the kernel’s newbus code, attached automatically when a matching device is present, detached automatically when the device goes away, and visible in `devinfo -v` as a device with a parent (`pci0`), a unit (`myfirst0`, `myfirst1`), and a set of claimed resources. That change is the change from “a module that happens to exist” to “a driver for a device that the kernel knows about”. Every later Part 4 chapter assumes you have made it.

18.1.2 Where Chapter 17 Left the Driver

A few prerequisites to verify before starting. Chapter 18 extends the driver produced at the end of Chapter 17 Stage 5, tagged as version `1.0-simulated`. If any of the items below feels uncertain, return to Chapter 17 before starting this chapter.

- Your driver compiles cleanly and identifies itself as `1.0-simulated` in `kldstat -v`.
- The softc carries `sc->hw` (a `struct myfirst_hw *` from Chapter 16) and `sc->sim` (a `struct myfirst_sim *` from Chapter 17). Every register access goes through `sc->hw`; every simulated behaviour lives under `sc->sim`.

- The register map of sixteen 32-bit registers spans offsets 0x00 through 0x3c, with the Chapter 17 additions (SENSOR, SENSOR_CONFIG, DELAY_MS, FAULT_MASK, FAULT_PROB, OP_COUNTER) in place.
- CSR_READ_4, CSR_WRITE_4, and CSR_UPDATE_4 wrap bus_space_read_4, bus_space_write_4, and a read-modify-write helper. Every access asserts sc->mtx is held on debug kernels.
- The sensor callout runs once per second on a ten-second cadence and oscillates SENSOR. The command callout fires per command with a configurable delay. The fault-injection framework is live.
- The module depends on nothing outside the base kernel; it is a kldload-able standalone driver.
- HARDWARE.md, LOCKING.md, and SIMULATION.md are current.
- INVARIANTS, WITNESS, WITNESS_SKIPSPIN, DDB, KDB, and KDB_UNATTENDED are enabled in your test kernel.

That driver is what Chapter 18 extends. The additions are again modest in volume: one new file (myfirst_pci.c), one new header (myfirst_pci.h), one new set of probe and attach routines, a small change to myfirst_hw_attach to accept a resource rather than allocate a buffer, an expanded detach ordering, a bump to 1.1-pci, a new PCI.md document, and an updated regression script. The mental-model change is larger than the line count suggests.

18.1.3 What You Will Learn

When you close this chapter you should be able to:

- Describe what PCI and PCIe are in one paragraph, in a way a beginner could follow, with the key vocabulary of bus, device, function, BAR, configuration space, vendor ID, and device ID clearly placed.
- Read the output of `pciconf -lv` and `devinfo -v` and locate the information a driver author cares about: the B:D:F tuple, the vendor and device IDs, the class, the subclass, the interface, the claimed resources, and the parent bus.
- Write a minimal PCI driver that registers a probe routine against the `pci` bus, matches a specific vendor and device ID, returns a meaningful `BUS_PROBE_*` priority, prints the matched device through `device_printf`, and unloads cleanly.
- Attach and detach a PCI driver, matching the expected lifecycle of `newbus`: probe runs first (sometimes twice), attach runs once per matched device, detach runs once when the kernel removes the device or the driver, and the softc is freed in the correct order.
- Use `DRIVER_MODULE(9)` and `MODULE_DEPEND(9)` correctly, naming the bus (`pci`) and the module dependency (`pci`) so the kernel's module loader and `newbus` enumerator understand the relationship.
- Explain what BARs are, how the firmware assigns them at boot, how the kernel discovers them, and why the driver does not choose the address.
- Claim a PCI memory BAR with `bus_alloc_resource_any(9)`, extract its `bus_space` tag and handle with `rman_get_bustag(9)` and `rman_get_bushandle(9)`, and hand them to the Chapter 16 accessor layer without any change to the CSR macros.
- Recognise when a BAR is a 64-bit BAR that spans two configuration-space slots, how `PCIR_BAR(index)` works, and why counting BARs by simple integer increments is not always safe on 64-bit BARs.
- Use `pci_read_config(9)` and `pci_write_config(9)` to read device-specific configuration-space fields that the generic accessors do not cover, and understand the width argument (1, 2, or 4) and the side-effect contract.
- Walk a device's PCI capability list with `pci_find_cap(9)` to locate standard capabilities (Power Management, MSI, MSI-X, PCI Express), and walk the PCIe extended capability list with `pci_find_extcap(9)`

to reach modern capabilities such as Advanced Error Reporting.

- Call `pci_enable_busmaster(9)` when the device will initiate DMA later, recognise why the `MEMEN` and `PORTEN` bits of the command register are usually already set by the bus driver by attach time, and know when a quirky device needs them asserted manually.
- Write an attach-time initialisation sequence that keeps the Chapter 17 simulation backend inactive on the PCI path while preserving a simulation-only build (via a compile-time switch) for readers without a PCI test environment.
- Write a detach path that releases resources in strictly the reverse order of attach, never leaking a resource or double-freeing one, even in the presence of partial attach failure.
- Exercise the driver against a real PCI device in a bhyve or QEMU guest, using a vendor and device ID that does not collide with a base-system driver, and observe the full attach, operate, detach, and unload cycle.
- Split the PCI-specific code into its own file, update the module's `SRCS` line, tag the driver as `1.1-pci`, and produce a short `PCI.md` that documents the vendor and device IDs the driver supports.
- Describe at a high level where MSI, MSI-X, and PCIe AER fit into the PCI picture, and know which later chapter picks each topic up.

The list is long; each item is narrow. The point of the chapter is the composition.

18.1.4 What This Chapter Does Not Cover

Several adjacent topics are explicitly deferred so Chapter 18 stays focused.

- **Real interrupt handlers.** `bus_alloc_resource_any(9)` for `SYS_RES_IRQ`, `bus_setup_intr(9)`, the split between a filter handler and an ithread handler, `INTR_TYPE_*` flags, `INTR_MPSAFE`, and the rules for what may and may not happen inside a handler belong to Chapter 19. Chapter 18's driver still polls through user-space writes and the Chapter 17 callouts; it never takes a real interrupt.
- **MSI and MSI-X.** `pci_alloc_msi(9)`, `pci_alloc_msix(9)`, vector allocation, per-queue interrupt routing, and the MSI-X table layout are Chapter 20. Chapter 18 only mentions these as future work when it lists PCI capabilities.
- **DMA.** `bus_dma(9)` tags, `bus_dmamap_create(9)`, `bus_dmamap_load(9)`, scatter-gather lists, bounce buffers, and cache-coherent descriptor rings are Chapters 20 and 21. Chapter 18 treats the BAR as a set of memory-mapped registers and nothing else.
- **PCIe AER handling.** The existence of Advanced Error Reporting is introduced because the reader should know the topic exists. Implementing a fault handler that subscribes to AER events, decodes the uncorrectable error register, and participates in system-wide recovery is a later-chapter topic.
- **Hot plug, device removal, and live runtime suspend.** A PCI device arriving or departing at runtime triggers a specific newbus sequence that a driver must respect; most drivers do respect it simply by having a correct detach path. Chapter 18 demonstrates the correct detach path and leaves runtime power management for Chapter 22 and hotplug for Part 7 (Chapter 32 on embedded platforms and Chapter 35 on asynchronous I/O and event handling).
- **Passthrough to virtual machines.** `bhyve(8)` and `vmm(4)` can pass a real PCI device through to a guest, and this is a useful technique for testing. Chapter 18 mentions it in passing. A deeper treatment belongs to the chapter where it serves the topic.
- **SR-IOV and virtual functions.** A PCIe capability where a single device advertises multiple virtual functions, each with its own configuration space, is out of scope for a beginner-level chapter.

- **Specific chipset quirks.** Real drivers often carry a long list of errata and workarounds for specific revisions of specific silicon. Chapter 18 aims at the common case; the book's later troubleshooting chapters cover how to reason about quirks when you meet them.

Staying inside those lines keeps Chapter 18 a chapter about the PCI subsystem and the driver's place in it. The vocabulary is what transfers; the specific chapters that follow apply the vocabulary to interrupts, DMA, and power.

18.1.5 Estimated Time Investment

- **Reading only:** four to five hours. The PCI topology and the newbus sequence are small in concept but dense in detail, and each piece rewards a slow read.
- **Reading plus typing the worked examples:** ten to twelve hours over two or three sessions. The driver evolves in four stages; each stage is a small but real refactor on top of the Chapter 17 codebase.
- **Reading plus all labs and challenges:** sixteen to twenty hours over four or five sessions, including standing up a bhyve or QEMU lab, reading `uart_bus_pci.c` and `virtio_pci_modern.c` in the real FreeBSD tree, and running the regression pass against both simulation and real PCI.

Sections 2, 3, and 5 are the densest. If the probe-attach sequence or the BAR allocation path feels unfamiliar on first pass, that is normal. Stop, re-read Section 3's diagram of how a BAR becomes a tag and handle, and continue when the picture has settled.

18.1.6 Prerequisites

Before starting this chapter, confirm:

- Your driver source matches Chapter 17 Stage 5 (1.0-simulated). The starting point assumes the Chapter 16 hardware layer, the Chapter 17 simulation backend, the complete `CSR_*` accessor family, the sync header, and every primitive introduced in Part 3.
- Your lab machine runs FreeBSD 14.3 with `/usr/src` on disk and matching the running kernel.
- A debug kernel with `INVARIANTS`, `WITNESS`, `WITNESS_SKIPSPIN`, `DDB`, `KDB`, and `KDB_UNATTENDED` is built, installed, and booting cleanly.
- `bhyve(8)` or `qemu-system-x86_64` is available on your lab host, and you can start a FreeBSD guest that boots your debug kernel. A bhyve guest is the canonical choice in this book; QEMU will work equivalently for every lab in this chapter.
- The `devinfo(8)` and `pciconf(8)` tools are in your path. Both live in the base system.

If any item above is shaky, fix it now rather than pushing through Chapter 18 and trying to reason from a moving foundation. PCI code is less forgiving than simulation code because a mismatch between your driver's expectations and the real bus's behaviour will usually surface as a failed probe, a failed attach, or a kernel page fault.

18.1.7 How to Get the Most Out of This Chapter

Four habits will pay off quickly.

First, keep `/usr/src/sys/dev/pci/pci.h` and `/usr/src/sys/dev/pci/pciivar.h` bookmarked. The first file is the authoritative register map of PCI and PCIe configuration space; every macro starting with `PCIR_`, `PCIM_`, or `PCIZ_` is defined there. The second file is the authoritative list of PCI accessor functions (`pci_get_vendor`, `pci_read_config`, `pci_find_cap`, and so on), along with their documentation comments. Reading both files once takes about an hour and removes most of the guesswork the rest of the chapter might otherwise require.

A note on line numbers. The declarations we will lean on later, such as `pci_read_config`, `pci_find_cap`, and the `PCIR_*` register-offset macros, sit in `pciivar.h` and `pci.h` under stable names. Whenever the chapter hands you a landmark in those files, the landmark is the symbol. Line numbers drift release to release; the names do not. Grep for the symbol and trust what your editor reports.

Second, run `pciconf -lv` on your lab host and your guest, and keep the output open in a terminal as you read. Every vocabulary item in the chapter (vendor, device, class, subclass, capabilities, resources) appears verbatim in that output. A reader who has read `pciconf -lv` for their own hardware finds the PCI subsystem much less abstract than one who has not.

Third, type the changes by hand and run each stage. PCI code is where small typos become silent mismatches. Mistyping `0x1af4` as `0x1af5` does not produce a compile error; it produces a driver that compiles cleanly and never attaches. Typing the values character by character, checking them against your test target, and confirming that `kldstat -v` shows the driver claiming the expected device are the habits that prevent a day of confused debugging.

Fourth, read `/usr/src/sys/dev/uart/uart_bus_pci.c` after Section 2 and `/usr/src/sys/dev/virtio/pci/virtio_pci_modern.c` after Section 5. The first file is a straightforward example of the pattern Chapter 18 teaches, written at a level a beginner can follow. The second file is a slightly richer example that shows how a real modern driver composes the pattern with additional machinery. Neither needs to be understood line by line; both repay a careful first reading.

18.1.8 Roadmap Through the Chapter

The sections in order are:

1. **What Is PCI and Why It Matters.** The bus, the topology, the B:D:F tuple, how FreeBSD represents PCI through the `pci(4)` subsystem, and how a driver author perceives all of this through `pciconf -lv` and `devinfo -v`. The conceptual foundation.
2. **Probing and Attaching a PCI Device.** The newbus dance from the driver's side: `device_method_t`, `probe`, `attach`, `detach`, `resume`, `suspend`, `DRIVER_MODULE(9)`, `MODULE_DEPEND(9)`, vendor and device ID matching, and the first stage of the Chapter 18 driver (`1.1-pci-stage1`).
3. **Understanding and Claiming PCI Resources.** What a BAR is, how the firmware assigns it, how `bus_alloc_resource_any(9)` claims it, and how the returned struct `resource *` becomes a `bus_space_tag_t` and `bus_space_handle_t`. The second stage (`1.1-pci-stage2`).
4. **Accessing Device Registers via bus_space(9).** How the Chapter 16 accessor layer snaps onto the new resource without modification, how the CSR macros carry through, and how the driver's first real PCI read happens.
5. **Driver Attach-Time Initialisation.** `pci_enable_busmaster(9)`, `pci_read_config(9)`, `pci_find_cap(9)`, `pci_find_extcap(9)`, and the small set of configuration-space operations the driver performs at attach. Keeping the Chapter 17 simulation inactive on the PCI path. The third stage (`1.1-pci-stage3`).

6. **Supporting Detach and Resource Cleanup.** Releasing resources in reverse order, handling partial attach failure, `device_delete_child`, and the detach regression script.
7. **Testing PCI Driver Behavior.** Standing up a bhyve or QEMU guest that exposes a device the driver recognises, observing attach, exercising the driver against the real BAR, reading and writing configuration space from user space with `pciconf -r` and `pciconf -w`, and using `devinfo -v` and `dmesg` to trace the driver's view of the world.
8. **Refactoring and Versioning Your PCI Driver.** The final split into `myfirst_pci.c`, the new `PCI.md`, the version bump to 1.1-pci, and the regression pass.

After the eight sections come hands-on labs, challenge exercises, a troubleshooting reference, a Wrapping Up that closes Chapter 18's story and opens Chapter 19's, and a bridge to Chapter 19 with a forward pointer to Chapter 20. The reference-and-cheat-sheet material at the end of the chapter is meant to be re-read as you work through later Part 4 chapters; Chapter 18's vocabulary is the vocabulary every later PCI-family chapter reuses.

If this is your first pass, read linearly and do the labs in order. If you are revisiting, Sections 3 and 5 stand alone and make good single-sitting reads.

18.2 Section 1: What Is PCI and Why It Matters

A reader who has come this far in the book has built a full driver around a simulated register block. The accessor layer, the command-response protocol, the locking discipline, the fault-injection framework, and the detach ordering are all in place. The one concession the driver makes to unreality is the origin of its registers: they come from a `malloc(9)` allocation in kernel memory, not from a piece of silicon on the far side of a bus. Section 1 introduces the subsystem that will change that. PCI is the most widespread peripheral bus in modern computing. It is also the canonical newbus child for a FreeBSD driver. Understanding what PCI is, how it got here, and how FreeBSD represents it is the foundation every remaining section in this chapter rests on.

18.2.1 A Short History, and Why It Matters

PCI stands for Peripheral Component Interconnect. It was introduced by Intel in the early 1990s as a replacement for the previous generation of PC expansion buses (ISA, EISA, VESA local bus, and others), none of which scaled to the speeds and widths that modern peripherals would soon demand. The original PCI specification described a parallel, shared, clocked bus that carried thirty-two bits of data, was clocked at 33 MHz, and let a single device request and hold the bus for a transaction. A handful of revisions increased the width to sixty-four bits, raised the clock to 66 MHz, and introduced signalling variants for server platforms (PCI-X), but the basic shape remained that of a shared parallel bus.

PCI Express, called PCIe, is the modern successor. It keeps the software-visible model of PCI almost unchanged, but it replaces the physical bus with a collection of point-to-point serial links. Where PCI had many devices sharing a single set of wires, PCIe has each device connected to the chipset's root complex through its own lane (or set of lanes, up to sixteen in common use and thirty-two in some high-end cards). The bandwidth per lane has climbed through successive generations, from 2.5 Gb/s in PCIe Gen 1 to 32 Gb/s in PCIe Gen 5 and beyond.

Why does this history matter to a driver author? Because the software model did not change across the transition. From the driver's point of view, a PCIe device still has a configuration space, still has BARs, still has a vendor

and device ID, still has capabilities, and still follows the probe-attach-detach lifecycle that PCI established. The physical layer changed; the software vocabulary did not. This is one of the few places in computing where a thirty-year-old interface is still the thing you read about in the FreeBSD source, and the continuity of the software model is what makes that possible. Code written for PCI in 1995 can, with minor updates for new capabilities, drive a PCIe Gen 5 device in 2026.

There is one important practical consequence of this continuity. When the book refers to “PCI”, it almost always means “PCI or PCIe”. The kernel’s `pci(4)` subsystem handles both. When a distinction matters, such as when a PCIe-only feature like MSI-X or AER appears, the book will call that out. Everywhere else, “PCI” and “PCIe” are the same thing at the driver level.

18.2.2 Where PCI Devices Live on a Modern Machine

Open almost any laptop, desktop, or server manufactured in the last twenty years, and you will find PCIe devices. The obvious ones are the add-in cards: a network adapter in a slot, a graphics card in another slot, an NVMe drive on the motherboard’s M.2 socket, a Wi-Fi module on a Mini-PCIe daughtercard. The less obvious ones are integrated: the storage controller that talks to the SATA ports is a PCI device; the USB host controllers are PCI devices; the onboard Ethernet is a PCI device; the audio codec is a PCI device; the platform’s integrated graphics is a PCI device. Everything that appears on the chipset-to-device interconnect between the CPU and the outside world is almost certainly a PCI device.

The kernel enumerates these devices at boot. Firmware (the system BIOS or UEFI) walks the bus, reads each device’s configuration space, assigns BARs, and hands control to the OS. The OS re-walks the bus, builds its own representation, and attaches drivers. FreeBSD’s `pci(4)` driver is what performs this walk. By the time the system is multi-user, every PCI device in the machine has been enumerated, every BAR has been assigned a kernel virtual address, and every device that matched a driver has been attached.

A practical demonstration: run `pciconf -lv` on any FreeBSD system. Each entry shows a device with its B:D:F (bus, device, function) address, its vendor and device IDs, its subvendor and subsystem IDs, its class and subclass, its current driver binding (if any), and a human-readable description of what it is. The entries are what the kernel saw; the descriptions are what `pciconf` looked up from its internal database. Running this command on your lab host is the best quick introduction to what your machine’s PCI topology looks like.

18.2.3 The Bus-Device-Function Tuple

The address of a PCI device has three components. Together they are called the **bus-device-function tuple**, or B:D:F, or sometimes just “PCI address”.

The **bus number** is which physical or logical PCI bus the device is on. A machine usually has one primary bus (bus 0), plus additional buses behind PCI-to-PCI bridges. A laptop might have buses 0, 2, 3, and 4; a server might have dozens. Each bus is eight bits wide, so the system supports up to 256 buses in the original PCI spec. PCIe extended this to 16 bits (65,536 buses) through the Enhanced Configuration Access Mechanism, ECAM.

The **device number** is the slot on the bus. Each bus can hold up to 32 devices. On PCIe, the point-to-point nature of the physical link means each bridge has one device on each of its downstream buses; the device number is essentially always 0 in that case. On legacy PCI, several devices share a bus and each gets its own device number.

The **function number** is which function of a multi-function device is being addressed. A single physical device can expose up to 8 functions, each with its own configuration space, each presenting itself as an independent PCI device. Multi-function devices are common: a typical x86 chipset presents its USB host controllers as multiple functions of a single physical unit; a storage controller might present SATA, IDE, and AHCI on separate functions. Single-function devices (the common case) use function 0.

The combined tuple is written in FreeBSD's `pciconf` output as `pciN:D:F`, where `N` is the domain-plus-bus value. On the author's test machine, `pci0:0:2:0` refers to domain 0, bus 0, device 2, function 0, which on an Intel platform is typically the integrated graphics. This notation is stable across FreeBSD versions; you will see it in the kernel's boot messages, in `dmesg`, in `devinfo -v`, and in the bus documentation.

A driver rarely cares about the `B:D:F` value directly. The newbus subsystem hides it behind a `device_t` handle. But a driver author cares, because two things use `B:D:F`: system administrators (who match a `B:D:F` against a slot or a physical device when installing or troubleshooting), and kernel messages (which print it in `dmesg` when a device attaches, detaches, or misbehaves). When you see `pci0:3:0:0: <Intel Corporation Ethernet Controller ...>` in your boot log, you are reading a `B:D:F`.

18.2.4 Configuration Space and What Lives There

PCI distinguishes two address spaces per device. The first is the set of BARs, which map the device's registers into the host memory (or I/O port) space; this is what Chapter 16 called "MMIO" and what Chapter 18's Section 3 and Section 4 will explore. The second is **configuration space**, which is a small, structured block of memory per device that describes the device itself.

Configuration space is where the vendor ID, device ID, class code, revision, BAR addresses, capability list pointer, and a number of other metadata fields live. It is 256 bytes on legacy PCI, expanded to 4096 bytes on PCIe. The layout of the first sixty-four bytes is standardised across every PCI device; the remaining space is used for capabilities and extended capabilities.

The driver reaches configuration space through the `pci_read_config(9)` and `pci_write_config(9)` interfaces. These two functions take a device handle, a byte offset into configuration space, and a width (1, 2, or 4 bytes), and return or accept a `uint32_t` value. The width argument lets the driver read or write a byte, a sixteen-bit field, or a thirty-two-bit field; the kernel translates this into the right underlying access primitive for the platform.

Most of what a driver needs to know about configuration space is already extracted by the newbus layer and cached in the device's ivars. That is why the driver can call `pci_get_vendor(dev)`, `pci_get_device(dev)`, `pci_get_class(dev)`, and `pci_get_subclass(dev)` without reading configuration space by hand. These accessors are defined in `/usr/src/sys/dev/pci/pcivar.h` and expanded through the `PCI_ACCESSOR` macro to inline functions that read a cached value. The values are read once, at enumeration time, and kept in the device's ivars thereafter.

For everything the common accessors do not cover, `pci_read_config(9)` and `pci_write_config(9)` are the fallback. An example: if a device's datasheet says "the firmware revision is at offset 0x48 in configuration space, as a 32-bit little-endian integer", the driver reads that value by calling `pci_read_config(dev, 0x48, 4)`. The kernel arranges the access so that the return value is the little-endian value the datasheet specified, on every supported architecture.

18.2.5 Vendor ID and Device ID: How Matches Happen

The core of PCI device identification is the pair of sixteen-bit values called vendor ID and device ID.

The **vendor ID** is assigned by the PCI Special Interest Group (PCI-SIG) to a company that manufactures PCI devices. Intel has 0x8086. Broadcom has several (originally 0x14e4 and others by acquisition). Red Hat and the Linux community's virtualisation project share 0x1af4 for virtio devices. Every PCI device carries its vendor's ID in the `VENDOR` field of configuration space.

The **device ID** is assigned by the vendor to each specific product. Intel's 0x10D3 is the 82574L gigabit Ethernet controller. Broadcom's 0x165F is a specific NetXtreme BCM5719 variant. Red Hat's 0x1001 in the virtio range is virtio-block. Vendors maintain their own device ID allocations.

A **subvendor ID** and **subsystem ID** together form a second-level tuple. They identify the board that a chipset is built into, as opposed to the chipset itself. The same Intel 82574L Ethernet chip might appear on a Dell server with subvendor 0x1028, on an HP server with subvendor 0x103c, and on a generic OEM board with subvendor 0x8086. Drivers can use the subvendor or subsystem to apply specific quirks, to print a more useful identification string, or to select between slightly different board-level behaviour.

A driver's probe routine matches against these IDs. In the simplest case, the driver has a static table listing every supported vendor-and-device pair; the probe walks the table and returns `BUS_PROBE_DEFAULT` on a match or `ENXIO` on no match. In more complex cases, the driver also checks subvendor and subsystem, or walks a broader class-based match, or uses both. The `uart_bus_pci.c` file in `/usr/src/sys/dev/uart/` shows this pattern at a readable scale.

Chapter 18's driver will use the simple tabular form. The table will hold one or two entries. The vendor and device IDs we target are the ones a bhyve or QEMU guest will expose for a synthetic test device, and the teaching path will unload the base-system driver that would otherwise claim the same ID before loading `myfirst`.

18.2.6 How FreeBSD Enumerates PCI Devices

The steps from "the firmware has set up the bus" to "a driver's attach routine runs" are worth understanding in outline, because understanding them makes the probe-and-attach sequence feel inevitable rather than mysterious.

First, the platform's bus-enumeration code runs. On x86 this lives under `/usr/src/sys/dev/pci/` and is driven by platform-specific attach code (x86 uses ACPI bridges and legacy host bridges; arm64 uses device-tree-based host bridges). The enumeration walks the bus, reads each device's vendor and device IDs, reads each device's BARs, and records what it finds.

Second, the kernel's newbus layer creates a `device_t` for each discovered device and adds it as a child of the PCI bus device (`pci0`, `pci1`, and so on). Each child has a device method table placeholder; the newbus code does not yet know which driver will bind. The child has ivars: the vendor, device, subvendor, subsystem, class, subclass, interface, revision, B:D:F, and resource descriptors are all cached in the ivars for later access.

Third, the kernel invites every registered driver to probe each device. Each driver's probe method is called in priority order. A driver inspects the device's vendor, device, and whatever else it needs, and returns one of a small set of values:

- A negative number: “I match this device and want it”. Values closer to zero mean higher priority. The standard tier for a vendor-and-device-ID match is `BUS_PROBE_DEFAULT`, which is `-20`. `BUS_PROBE_VENDOR` is `-10` and wins over it; `BUS_PROBE_GENERIC` is `-100` and loses to it. Section 2 lists the full tier set.
- 0: “I match this device with absolute priority”. The `BUS_PROBE_SPECIFIC` tier. No other driver can outbid it.
- A positive errno (commonly `ENXIO`): “I do not match this device”.

The kernel picks the driver that returned the numerically smallest value and attaches it. If two drivers return the same value, the one that registered first wins. The tiered priority lets a generic driver coexist with a device-specific driver: the generic driver returns `BUS_PROBE_GENERIC`, the specific driver returns `BUS_PROBE_DEFAULT`, and the specific driver wins because `-20` is closer to zero than `-100`.

Fourth, the kernel calls the winning driver’s `attach` method. The driver allocates its softc (usually pre-allocated by `newbus`), claims resources with `bus_alloc_resource_any(9)`, sets up interrupts, and registers a character device or network interface or whatever the device exposes to user space. If `attach` returns 0, the device is live. If `attach` returns an errno, the kernel detaches the driver (calling `detach` is not strictly required on `attach` failure in modern `newbus`; the driver is expected to unwind cleanly before returning the error).

Fifth, the kernel moves to the next device. The process repeats until every PCI device has been probed and every device that found a match has been attached.

Detach is the inverse: the kernel calls each driver’s `detach` method when the device is removed (through `devctl detach` or at module unload), and the driver releases everything it claimed in `attach`, in reverse order.

This is the `newbus` dance Chapter 18 teaches the driver how to follow. Section 2 writes the first version of it; Sections 3 through 6 add each additional capability; Section 8 consolidates it into a clean module.

18.2.7 The pci(4) Subsystem from the Driver’s Perspective

The driver does not see bus enumeration. It sees a device handle (`device_t dev`) and a set of accessor calls. The `/usr/src/sys/dev/pci/pcivar.h` header defines the accessors. The core ones are:

- `pci_get_vendor(dev)` returns the vendor ID as `uint16_t`.
- `pci_get_device(dev)` returns the device ID as `uint16_t`.
- `pci_get_subvendor(dev)` and `pci_get_subdevice(dev)` return the subvendor and subsystem.
- `pci_get_class(dev)`, `pci_get_subclass(dev)`, and `pci_get_progif(dev)` return the class code fields.
- `pci_get_revid(dev)` returns the revision.
- `pci_read_config(dev, reg, width)` reads from configuration space.
- `pci_write_config(dev, reg, val, width)` writes to configuration space.
- `pci_find_cap(dev, cap, &capreg)` finds a standard PCI capability; returns 0 on success, `ENOENT` on absence.
- `pci_find_extcap(dev, cap, &capreg)` finds a PCIe extended capability; same return convention.
- `pci_enable_busmaster(dev)` sets the Bus Master Enable bit in the command register.
- `pci_disable_busmaster(dev)` clears it.

This is the vocabulary of Chapter 18. Every section in the chapter uses one or more of these accessors. A reader who is comfortable with the shape of the list is ready to start writing PCI code.

18.2.8 Common PCI Devices in the Real World

Before moving on, a brief tour of the devices PCI presents.

Network interface controllers. NICs are nearly all PCI devices. The Intel `em(4)` driver for the 8254x family, the Intel `ix(4)` driver for the 82599 family, the Intel `ixl(4)` driver for the X710 family, and the Broadcom `bge(4)` / `bnxt(4)` drivers for the NetXtreme family all live under `/usr/src/sys/dev/e1000/` or `/usr/src/sys/dev/ixl/` or `/usr/src/sys/dev/bge/`. They are large, production-grade drivers that exercise essentially every topic in Part 4.

Storage controllers. AHCI SATA controllers, NVMe drives, SAS HBAs, and RAID controllers are all PCI. `ahci(4)`, `nvme(4)`, `mpr(4)`, `mpi3mr(4)`, and others live under `/usr/src/sys/dev/`. These are among the best-maintained drivers in the tree.

USB host controllers. xHCI, EHCI, and OHCI controllers are PCI. The generic host-controller driver attaches to each, and the USB subsystem handles everything above it. `xhci(4)` is the canonical one for modern systems.

Graphics cards and integrated graphics. GPU drivers in FreeBSD are mostly maintained out of tree (DRM drivers from the `drm-kmod` ports), but the bus attach for them is standard PCI.

Audio controllers. HDA codecs, older AC'97 bridges, and various USB-attached audio devices all reach the system through PCI in some way. `snd_hda(4)` is the usual attach point.

Virtio devices in virtual machines. When a FreeBSD guest runs under bhyve, KVM, VMware, or Hyper-V, the paravirtualised devices appear as PCI. `virtio-network`, `virtio-block`, `virtio-entropy`, and `virtio-console` all look like PCI devices to the guest. The `virtio_pci(4)` driver attaches first and publishes child nodes for each of the transport-specific virtio drivers.

The machine's own chipset components. The platform's LPC bridge, SMBus controller, thermal sensor interface, and various miscellaneous control functions are PCI.

If you have ever wondered why a FreeBSD source tree is as large as it is, the PCI device ecosystem is most of the answer. Every device in the list above needs a driver. The driver you are building in Chapter 18 is small and does very little; the drivers in the examples are large because they implement real protocols over the PCI bus. But the shape of every one of them is what Chapter 18 teaches.

18.2.9 Simulated PCI Devices: bhyve and QEMU

A reader who owns a complete set of test hardware can skip this subsection. Everyone else depends on virtualisation to provide the PCI devices they will drive.

FreeBSD's bhyve(8) hypervisor, built into the base system, can present a guest with a set of emulated PCI devices. The common ones are `virtio-net`, `virtio-blk`, `virtio-rnd`, `virtio-console`, `ahci-hd`, `ahci-cd`, `e1000`, `xhci`, and a framebuffer device. Each has a well-known vendor and device ID; the guest's PCI enumerator sees them as real PCI devices; the guest's drivers attach to them as they would to real hardware. Running a FreeBSD guest under bhyve is the canonical way, for this book, to have a PCI device the reader's driver can attach to.

QEMU with KVM (on Linux hosts) or with the HVF accelerator (on macOS hosts) provides a superset of bhyve's emulated devices, plus a few specifically designed for testing. The `pci-testdev` device (vendor 0x1b36, device

0x0005) is a deliberately minimal PCI device intended for kernel test code; it has two BARs (one memory, one I/O), and writes to specific offsets trigger specific behaviours. Chapter 18's Section 7 can use either a virtio-rnd device under bhyve or a pci-testdev under QEMU as the target.

For the teaching path, the book targets a virtio-rnd device under bhyve. The reason is that bhyve ships with every FreeBSD installation, whereas QEMU requires additional packages. The cost is small: the virtio-rnd device has a real driver in the base system (`virtio_random(4)`), and the chapter will show how to prevent that driver from claiming the device so that `myfirst` can claim it instead.

An important note about the teaching path's choice. The `myfirst` driver is not a real virtio-rnd driver. It has no idea how to speak the virtio-rnd protocol; it treats the BAR as a set of opaque registers and reads and writes them for demonstration. This is fine for the chapter's purpose (proving that the driver can attach, read, write, and detach), but it is not fine for production use. Chapter 18 is a hands-on introduction to the PCI attach sequence, not a tutorial on how to write a virtio driver. When you finish the chapter, the driver you have is still the `myfirst` educational driver, now capable of attaching to a PCI bus rather than only to a `kldload` path.

18.2.10 Placing Chapter 18 in the Driver's Evolution

A quick map of where the `myfirst` driver has been and where it is going.

- **Version 0.1 through 0.8** (Part 1 through Part 3): the driver learned the driver lifecycle, the `cdev` machinery, concurrency primitives, and coordination.
- **Version 0.9-coordination** (end of Chapter 15): full lock discipline, condition variables, `sx` locks, callouts, taskqueue, counting semaphore.
- **Version 0.9-mmio** (end of Chapter 16): `bus_space(9)`-backed register block, CSR macros, access log, hardware layer in `myfirst_hw.c`.
- **Version 1.0-simulated** (end of Chapter 17): dynamic register behaviour, callouts that change state, command-response protocol, fault injection, simulation layer in `myfirst_sim.c`.
- **Version 1.1-pci** (end of Chapter 18, our target): the simulation is switchable, and when the driver attaches to a real PCI device, the BAR becomes the register block, `myfirst_hw_attach` uses `bus_alloc_resource_any` instead of `malloc`, and the Chapter 16 accessor layer points at real silicon.
- **Version 1.2-intr** (Chapter 19): a real interrupt handler registered through `bus_setup_intr(9)`, so the driver can react to the device's own state changes rather than polling.
- **Version 1.3-msi** (Chapter 20): MSI and MSI-X, giving the driver a richer interrupt-routing story.
- **Version 1.4-dma** (Chapters 20 and 21): a `bus_dma(9)` tag, descriptor rings, and the first real DMA transfers.

Each version is a layer on the one before it. Chapter 18 is one layer, small enough to teach clearly, large enough to matter.

18.2.11 Exercise: Read Your Own PCI Topology

Before Section 2, a short exercise to make the vocabulary concrete.

On your lab host, run:

```
sudo pciconf -lv
```

You will see a list of every PCI device the kernel enumerated. Each entry looks roughly like:

```
em0@pci0:0:25:0:      class=0x020000 rev=0x03 hdr=0x00 vendor=0x8086 device=0x15ba
↪  subvendor=0x8086 subdevice=0x2000
    vendor      = 'Intel Corporation'
    device      = 'Ethernet Connection (2) I219-LM'
    class       = network
    subclass    = ethernet
```

Pick three devices from the list. For each, identify:

- The device's logical name in FreeBSD (the leading name@pciN:B:D:F string).
- The vendor and device IDs.
- The class and subclass (the meaningful English categories, not just the hex codes).
- Whether the device has a driver bound to it (em0, for example, is bound to em(4); an entry with just none0@... has no driver).

Keep this output in a terminal while you read the rest of the chapter. Every vocabulary item introduced in Sections 2 through 5 refers to fields you can find here. The exercise is to anchor the abstract vocabulary to a concrete set of devices on your machine.

If you are reading the book without a FreeBSD machine available, the following snippet is the output of pciconf -lv on the author's lab host, truncated to the first three devices:

```
hostb0@pci0:0:0:0:      class=0x060000 rev=0x00 hdr=0x00 vendor=0x8086 device=0x3e31
    vendor      = 'Intel Corporation'
    device      = '8th Gen Core Processor Host Bridge/DRAM Registers'
    class       = bridge
    subclass    = HOST-PCI
pcib0@pci0:0:1:0:      class=0x060400 rev=0x00 hdr=0x01 vendor=0x8086 device=0x1901
    vendor      = 'Intel Corporation'
    device      = '6th-10th Gen Core Processor PCIe Controller (x16)'
    class       = bridge
    subclass    = PCI-PCI
vgapci0@pci0:0:2:0:     class=0x030000 rev=0x00 hdr=0x00 vendor=0x8086 device=0x3e9b
    vendor      = 'Intel Corporation'
    device      = 'CoffeeLake-H GT2 [UHD Graphics 630]'
    class       = display
    subclass    = VGA
```

Three devices, three drivers, three class codes. The host bridge (hostb0) is the PCI-to-memory-bus bridge; the PCI bridge (pcib0) is a PCI-to-PCI bridge leading to the GPU slot; the VGA-class device (vgapci0) is the integrated graphics on a Coffee Lake chipset. Every one of them follows the probe-attach-detach dance Chapter 18 teaches. The driver is what changes. The bus dance does not.

18.2.12 Wrapping Up Section 1

PCI is the canonical peripheral bus of modern systems and the canonical newbus child of FreeBSD. It is shared by PCI and PCIe, which differ at the physical layer but present the same software-visible model. Every PCI device has a B:D:F address, a configuration space, a set of BARs, a vendor ID, a device ID, and a place in the kernel's newbus tree. A driver's job is to match one or more devices by their IDs, claim their BARs, and expose their behaviour through some user-space interface. FreeBSD's `pci(4)` subsystem does the enumeration; the driver does the attachment.

The vocabulary of Section 1 is the vocabulary the rest of the chapter uses. B:D:F, configuration space, BARs, vendor and device IDs, class codes, capabilities, and the newbus probe-attach-detach sequence. If any one of these feels unfamiliar, re-read the relevant subsection before continuing. Section 2 takes the vocabulary and builds the first version of the driver.

18.3 Section 2: Probing and Attaching a PCI Device

Section 1 established what PCI is and how FreeBSD represents it. Section 2 is where the driver finally uses that vocabulary. The goal here is to stand up the minimum viable PCI driver: a driver that registers itself as a candidate for the PCI bus, matches a specific vendor and device ID, prints a banner in `dmesg` when the match succeeds, and unloads cleanly. No BAR claim yet. No register access yet. Just the skeleton.

The skeleton is important. It introduces the probe-attach-detach sequence in isolation, before BARs and resources and configuration-space walks crowd the picture. A reader who writes this skeleton once, by hand, and then types `kldload ./myfirst.ko`, sees `dmesg` report the driver probing and attaching, and types `kldunload myfirst` to see the detach side fire cleanly, has built the right mental model for everything that follows. Every later Part 4 chapter assumes this mental model.

18.3.1 The Probe-Attach-Detach Contract

Every newbus driver has three methods at the heart of its lifecycle. `probe` asks “is this device something I know how to drive?” `attach` says “yes I want it, and here is how I claim it”. `detach` says “release this device, I am going away”.

Probe. Called by the kernel once per device that the bus has enumerated, for each driver that has registered an interest in that bus. The driver reads the device's vendor and device IDs (and whatever else it needs to decide), returns a priority value if it wants the device, and returns `ENXIO` if it does not. The priority system is what lets a specific driver win over a generic one: a driver that returns `BUS_PROBE_DEFAULT` wins over one that returns `BUS_PROBE_GENERIC` when both want the same device. If no driver returns a match, the device stays unclaimed (you will see this as `nonea@pci0:...` entries in `devinfo -v`).

A subtle point: **probe may be called more than once for a given device**. The newbus reprobe machinery exists to handle devices that appear at runtime (hotplug) or that come back from suspend. A good probe is idempotent: it reads the same state, makes the same decision, returns the same value. Probe must not allocate resources, set up timers, register interrupts, or do anything that would need to be undone. It only inspects and decides.

A second subtle point: **probe runs before attach, but after the kernel has assigned the device's resources**. The BARs, the IRQ, and the configuration space are all accessible from probe. This means probe can read

device-specific registers through `pci_read_config` to distinguish variants of a chipset by revision or silicon ID if needed. Real drivers occasionally do this. Chapter 18's driver does not need to; vendor and device IDs are enough.

Attach. Called once per device, after probe has selected a winner. The driver's attach routine is where real work happens: softc initialisation, resource allocation, register mapping, character-device creation, and any configuration the device needs at bring-up. If attach returns 0, the device is live; the kernel considers the driver bound to the device and moves on. If attach returns a non-zero value, the kernel treats the attach as having failed. The driver must clean up anything it allocated before returning the error; modern newbus does not call detach in that case (older convention did, so older drivers still structure their error paths to handle it).

Detach. Called when the driver is being unbound from the device. The call is the mirror of attach: everything attach allocated, detach releases. Everything attach set up, detach tears down. Everything attach registered, detach unregisters. The ordering is strict: detach must undo in the reverse of the order attach did. A mistake here produces kernel panics at unload time, leaked resources at the best, or subtle use-after-free bugs at the worst.

Resume and **suspend** are optional methods. They are called when the system suspends and resumes, and they give the driver a chance to save and restore device state across the power event. Chapter 18's driver does not implement either method in the first stage; we will add resume in a later chapter when the topic serves the material.

There are other methods (shutdown, quiesce, identify) that less commonly matter for a basic PCI driver. The Chapter 18 skeleton registers only the three core methods plus `DEVMETHOD_END`.

18.3.2 The Device Method Table

FreeBSD's newbus machinery reaches driver methods through a table. The table is an array of `device_method_t` entries, each entry mapping a method name to the C function that implements it. The table ends with `DEVMETHOD_END`, which is simply a zeroed entry that tells newbus "no more methods here".

The table is declared at file scope, like this, in the driver's source:

```
static device_method_t myfirst_pci_methods[] = {
    DEVMETHOD(device_probe,      myfirst_pci_probe),
    DEVMETHOD(device_attach,     myfirst_pci_attach),
    DEVMETHOD(device_detach,     myfirst_pci_detach),
    DEVMETHOD_END
};
```

Each `DEVMETHOD(name, func)` expands to a `{ name, func }` initialiser. The newbus layer reaches a driver's method by looking up the name in this table. If a method is not registered (for example, `device_resume` is not in this table), the newbus layer uses a default implementation; for resume the default is a no-op, for probe the default is `ENXIO`.

The method names are defined in `/usr/src/sys/sys/bus.h` and expanded by the newbus build system. Each one corresponds to a function prototype the driver must match. For example, the `device_probe` method's prototype is:


```
int probe(device_t dev);
```

A driver implementation must have that exact signature. Type mismatches produce compile errors, not runtime mysteries; if your probe's signature is wrong, the build will fail.

18.3.3 The Driver Structure

Alongside the method table, the driver declares a `driver_t`. This structure ties together the method table, the softc size, and a short name:

```
static driver_t myfirst_pci_driver = {  
    "myfirst",  
    myfirst_pci_methods,  
    sizeof(struct myfirst_softc),  
};
```

The name ("myfirst") is what newbus will use when numbering unit instances. The first attached device becomes `myfirst0`, the second `myfirst1`, and so on. This name is what `devinfo -v` shows and what user-space tools (like `/dev/myfirst0` if the driver creates a cdev with that name) expose.

The softc size tells newbus how many bytes to allocate for each device's softc. The allocation is automatic: by the time `attach` runs, `device_get_softc(dev)` returns a pointer to a zeroed block of the requested size. The driver does not call `malloc` for the softc itself; it uses what newbus gave it. This is a convenience that the `myfirst` driver has already been using since Chapter 10; it becomes more important with PCI because every unit has its own softc and newbus manages the lifetime.

18.3.4 DRIVER_MODULE and MODULE_DEPEND

The driver is glued to the PCI bus through two macros. The first is `DRIVER_MODULE(9)`:

```
DRIVER_MODULE(myfirst, pci, myfirst_pci_driver, NULL, NULL);
```

The expansion of this macro performs several things. It registers the driver as a child candidate of the `pci` bus, wrapping the `driver_t` in a kernel module descriptor. It schedules the driver to participate in probe for every device the `pci` bus enumerates. It provides hooks for optional module event handlers (the two `NULL`s are for module init and cleanup, respectively; we leave them empty for now).

The first argument is the module's name, which must match the name in the `driver_t`. The second argument is the bus's name; `pci` is the newbus name of the PCI bus driver. The third argument is the driver itself. The remaining arguments are for optional callbacks.

The macro has a subtle consequence: the driver will participate in probe on every PCI bus in the system. If you have multiple PCI domains, the driver will be offered every device on every domain. Probe's job is to say yes only to the devices the driver actually supports; the kernel's job is to ask.

The second macro is `MODULE_DEPEND(9)`:

```
MODULE_DEPEND(myfirst, pci, 1, 1, 1);
```

This tells the module loader that `myfirst.ko` depends on the `pci` kernel module. The three numbers are minimum, preferred, and maximum version. A zero-to-one dependency on version 1 is the common case. The loader uses this information to refuse to load `myfirst.ko` if the kernel's PCI subsystem is not present (which is essentially never false on a real system, but the check is good hygiene).

Without `MODULE_DEPEND`, the loader might load `myfirst.ko` before the PCI subsystem is available at early boot, leading to a panic when `DRIVER_MODULE` tries to register against a bus that does not exist yet. With it, the loader serialises the load correctly.

18.3.5 Matching by Vendor and Device ID

The probe routine is where the vendor-and-device match happens. The pattern is a static table and a loop. Consider a minimal version:

```
static const struct myfirst_pci_id {
    uint16_t    vendor;
    uint16_t    device;
    const char  *desc;
} myfirst_pci_ids[] = {
    { 0x1af4, 0x1005, "Red Hat / Virtio entropy source (demo target)" },
    { 0, 0, NULL }
};

static int
myfirst_pci_probe(device_t dev)
{
    uint16_t vendor = pci_get_vendor(dev);
    uint16_t device = pci_get_device(dev);
    const struct myfirst_pci_id *id;

    for (id = myfirst_pci_ids; id->desc != NULL; id++) {
        if (id->vendor == vendor && id->device == device) {
            device_set_desc(dev, id->desc);
            return (BUS_PROBE_DEFAULT);
        }
    }
    return (ENXIO);
}
```

A few things to notice. The table is small and static, one entry per supported device. `pci_get_vendor` and `pci_get_device` read the cached ivars, so the calls are cheap. The comparison is a plain loop; the table is short enough not to warrant a hash. `device_set_desc` installs a human-readable description that `pciconf -lv` and

dmesg will print when the device attaches. `BUS_PROBE_DEFAULT` is the standard priority for a vendor-specific match; it wins over generic class-based drivers but loses to any driver that explicitly returns a more negative value.

A subtle but important point: this probe routine targets the `virtio-rnd` (entropy) device that the base-system `virtio_random(4)` driver normally claims. If both drivers are loaded, the system's priority rules decide the winner. `virtio_random` registers `BUS_PROBE_DEFAULT`, as does `myfirst`. The tiebreaker is registration order, which varies. The reliable way to guarantee that `myfirst` attaches is to unload `virtio_random` before loading `myfirst`. Section 7 will show how.

A second note: the vendor and device IDs in the example above target a `virtio` device. Real PCI drivers for real hardware would target chips whose IDs are not already claimed by base-system drivers. For a production driver, the list would include every supported variant of the target chipset, often with descriptive strings that identify the silicon revision. `uart_bus_pci.c` has sixty-plus entries; `ix(4)` has over a hundred.

18.3.6 The Probe Priority Tiers

FreeBSD defines several probe priority tiers, defined in `/usr/src/sys/sys/bus.h`:

- `BUS_PROBE_SPECIFIC = 0`. The driver matches the device precisely. No other driver can outbid this.
- `BUS_PROBE_VENDOR = -10`. The driver is vendor-provided and should win over anything generic.
- `BUS_PROBE_DEFAULT = -20`. The standard tier for a vendor-and-device-ID match.
- `BUS_PROBE_LOW_PRIORITY = -40`. A lower-priority match, often for drivers that want to be the default only if nothing else claims the device.
- `BUS_PROBE_GENERIC = -100`. A generic driver that attaches to a class of devices if nothing more specific exists.
- `BUS_PROBE_HOOVER = -1000000`. Absolute last resort; a driver that wants devices no other driver claimed.
- `BUS_PROBE_NOWILDCARD = -2000000000`. Special-case marker used by the newbus identify machinery.

Most drivers you will write or read use `BUS_PROBE_DEFAULT`. Some use `BUS_PROBE_VENDOR` if they expect to coexist with generic drivers. A handful use `BUS_PROBE_GENERIC` or lower for their fallback mode. Chapter 18's driver uses `BUS_PROBE_DEFAULT` throughout.

The priority values are negative by convention so that the numerically lowest value wins. A more specific driver has a more negative value. This is counter-intuitive on first reading; a mental model is “distance from perfection, measured downward”. `BUS_PROBE_SPECIFIC` is zero distance. `BUS_PROBE_GENERIC` is a hundred units worse.

18.3.7 Writing a Minimal PCI Driver

Putting it all together, here is the Chapter 18 Stage 1 driver, presented as a single self-contained file that grows out of the Chapter 17 skeleton. The file name is `myfirst_pci.c`; it is new in Chapter 18 and lives alongside the existing `myfirst.c`, `myfirst_hw.c`, and `myfirst_sim.c`.

```
/*-
 * SPDX-License-Identifier: BSD-2-Clause
 *
```

```
* Copyright (c) 2026 Edson Brandi
*
* myfirst_pci.c -- Chapter 18 Stage 1 PCI probe/attach skeleton.
*
* At this stage the driver only probes, attaches, and detaches.
* It does not yet claim BARs or touch device registers. Section 3
* adds resource allocation. Section 5 wires the accessor layer to
* the claimed BAR.
*/

#include <sys/param.h>
#include <sys/system.h>
#include <sys/kernel.h>
#include <sys/module.h>
#include <sys/bus.h>

#include <dev/pci/pci.h>
#include <dev/pci/pcivar.h>

#include "myfirst.h"
#include "myfirst_pci.h"

static const struct myfirst_pci_id myfirst_pci_ids[] = {
    { MYFIRST_VENDOR_REDHAT, MYFIRST_DEVICE_VIRTIO_RNG,
      "Red Hat Virtio entropy source (myfirst demo target)" },
    { 0, 0, NULL }
};

static int
myfirst_pci_probe(device_t dev)
{
    uint16_t vendor = pci_get_vendor(dev);
    uint16_t device = pci_get_device(dev);
    const struct myfirst_pci_id *id;

    for (id = myfirst_pci_ids; id->desc != NULL; id++) {
        if (id->vendor == vendor && id->device == device) {
            device_set_desc(dev, id->desc);
            return (BUS_PROBE_DEFAULT);
        }
    }
    return (ENXIO);
}
```

```

static int
myfirst_pci_attach(device_t dev)
{
    struct myfirst_softc *sc = device_get_softc(dev);

    sc->dev = dev;
    device_printf(dev,
        "attaching: vendor=0x%04x device=0x%04x revid=0x%02x\n",
        pci_get_vendor(dev), pci_get_device(dev), pci_get_revid(dev));
    device_printf(dev,
        "        subvendor=0x%04x subdevice=0x%04x class=0x%02x\n",
        pci_get_subvendor(dev), pci_get_subdevice(dev),
        pci_get_class(dev));

    /*
     * Stage 1 has no resources to claim and nothing to initialise
     * beyond the softc pointer. Stage 2 will add the BAR allocation.
     */
    return (0);
}

static int
myfirst_pci_detach(device_t dev)
{
    device_printf(dev, "detaching\n");
    return (0);
}

static device_method_t myfirst_pci_methods[] = {
    DEVMETHOD(device_probe,     myfirst_pci_probe),
    DEVMETHOD(device_attach,   myfirst_pci_attach),
    DEVMETHOD(device_detach,   myfirst_pci_detach),
    DEVMETHOD_END
};

static driver_t myfirst_pci_driver = {
    "myfirst",
    myfirst_pci_methods,
    sizeof(struct myfirst_softc),
};

DRIVER_MODULE(myfirst, pci, myfirst_pci_driver, NULL, NULL);
MODULE_DEPEND(myfirst, pci, 1, 1, 1);
MODULE_VERSION(myfirst, 1);

```

The companion header, `myfirst_pci.h`:

```
/*-
 * SPDX-License-Identifier: BSD-2-Clause
 *
 * Copyright (c) 2026 Edson Brandi
 *
 * myfirst_pci.h -- Chapter 18 PCI interface for the myfirst driver.
 */

#ifndef _MYFIRST_PCI_H_
#define _MYFIRST_PCI_H_

#include <sys/types.h>

/* Target vendor and device IDs for the Chapter 18 demo. */
#define MYFIRST_VENDOR_REDHAT      0x1af4
#define MYFIRST_DEVICE_VIRTIO_RNG  0x1005

/* A single entry in the supported-device table. */
struct myfirst_pci_id {
    uint16_t    vendor;
    uint16_t    device;
    const char  *desc;
};

#endif /* _MYFIRST_PCI_H_ */
```

And the Makefile needs a small update:

```
# Makefile for the Chapter 18 Stage 1 myfirst driver.

KMOD=  myfirst
SRCS=  myfirst.c myfirst_hw.c myfirst_sim.c myfirst_pci.c cbuf.c

CFLAGS+= -DMYFIRST_VERSION_STRING=\"1.1-pci-stage1\"

.include <bsd.kmod.mk>
```

Three things have changed from Chapter 17 Stage 5. `myfirst_pci.c` is added to `SRCS`. The version string is bumped to `1.1-pci-stage1`. Nothing else needs to change.

18.3.8 What Happens When the Driver Loads

Walking through the loading sequence makes the skeleton concrete.

The reader invokes `kldload ./myfirst.ko`. The kernel's module loader reads the module's metadata. It sees the `MODULE_DEPEND(myfirst, pci, ...)` declaration and verifies that the `pci` module is loaded. (It always is on a running kernel, so this check passes.) It sees the `DRIVER_MODULE(myfirst, pci, ...)` declaration and registers the driver as a probe candidate for the PCI bus.

The kernel then iterates over every PCI device in the system and calls `myfirst_pci_probe` for each. Most probes return `ENXIO` because the vendor and device IDs do not match. One probe, against the `virtio-rnd` device in the guest, returns `BUS_PROBE_DEFAULT`. The kernel picks `myfirst` as the driver for that device.

If the `virtio-rnd` device is already attached to `virtio_random`, the new driver's probe result competes with the existing binding. The kernel does not re-bind a device automatically just because a new driver appeared; instead, `myfirst` will not attach. To force the rebind, the reader must first detach the existing driver: `devctl detach virtio_random0`, or `kldunload virtio_random`. Section 7 walks through this.

Once the kernel decides `myfirst` wins, it allocates a new `softc` (the `sizeof(struct myfirst_softc)` block requested in the `driver_t`), clears it to zero, and calls `myfirst_pci_attach`. The attach routine runs. It prints a short banner. It returns 0. The kernel marks the device as attached.

`dmesg` shows the sequence:

```
myfirst0: <Red Hat Virtio entropy source (myfirst demo target)> port 0x6040-0x605f mem
↳ 0xc1000000-0xc100001f irq 19 at device 5.0 on pci0
myfirst0: attaching: vendor=0x1af4 device=0x1005 revid=0x00
myfirst0:          subvendor=0x1af4 subdevice=0x0004 class=0xff
```

`devinfo -v` shows the device with its parent, its resources, and its driver binding. `pciconf -lv` shows it with `myfirst0` as its bound name.

On unload, the inverse happens. `kldunload myfirst` calls `myfirst_pci_detach` on every attached device. The detach prints its own banner, returns 0, and the kernel releases the `softc`. `DRIVER_MODULE` unregisters the driver from the PCI bus. The module loader removes the `myfirst.ko` image from memory.

18.3.9 device_printf and Why It Matters

One small detail worth emphasising. The skeleton uses `device_printf(dev, ...)` instead of `printf(...)`. The difference is small but important.

`printf` prints to the kernel log without any prefix. A line that says “attaching” is hard to associate with a specific device; the log is full of messages from every driver in the system. `device_printf(dev, ...)` prefixes the message with the driver's name and unit number: “`myfirst0: attaching`”. The prefix makes the log readable even when multiple instances of the driver are attached at once (`myfirst0`, `myfirst1`, and so on).

The convention is strict in the FreeBSD source tree: every driver uses `device_printf` in code paths that have a `device_t` handy, and falls back to `printf` only in very early module init or module unload, where the handle is not available. A reader who habitually uses `device_printf` produces logs that are easy to read and diagnose; a reader who uses `printf` everywhere produces logs that other contributors will ask to have corrected.

18.3.10 The Softc and device_get_softc

The Chapter 17 driver already had a softc structure. Chapter 18's attach routine simply reuses it, with one addition: the PCI layer stores the device_t in sc->dev so later code (including the Chapter 16 accessors and the Chapter 17 simulation) can reach it.

A reminder: device_get_softc(dev) returns a pointer to the softc that newbus pre-allocated. The softc is zeroed before attach runs, so every field starts at zero or NULL or false. The softc is freed automatically by newbus after detach returns; the driver does not call free on it.

This is worth calling out because it differs from the malloc-based softc pattern in older FreeBSD drivers and in some Linux drivers. In newbus, the bus owns the softc lifetime. In older patterns, the driver owned it and had to remember to allocate and free. Forgetting to allocate in the old model causes a null dereference in attach; forgetting to free in the old model causes a memory leak in detach. Neither failure mode exists in modern newbus because the bus handles both operations.

18.3.11 Probe-Attach-Detach Order in the Face of Unload

An important detail for a driver author is what happens when kldunload myfirst runs while one or more myfirst devices are attached.

The module loader's unload path first tries to detach every device bound to the driver. For each device, it calls the driver's detach method. If detach returns 0, the device is considered unbound; the softc is freed. If detach returns non-zero (usually EBUSY), the module loader aborts the unload: the module stays loaded, the device stays attached, and the unload returns an error. This is how a driver refuses to go away while it has work in progress.

The myfirst driver's detach should normally succeed, because the driver's user-facing state is idle by the time the user asks to unload it. But a driver that is actively serving requests (for example, a disk driver with open file descriptors on its cdev) returns EBUSY from detach and forces the user to close the descriptors first.

For Chapter 18 Stage 1, detach is a one-liner: print a banner and return 0. In later stages, detach will acquire the lock, cancel callouts, release resources, and finally return 0 once everything is torn down.

18.3.12 Stage 1 Output: What Success Looks Like

The reader loads the driver. On a bhyve guest with a virtio-rnd device attached, and with virtio_random unloaded first, dmesg should print something like:

```
myfirst0: <Red Hat Virtio entropy source (myfirst demo target)> ... on pci0
myfirst0: attaching: vendor=0x1af4 device=0x1005 revid=0x00
myfirst0:          subvendor=0x1af4 subdevice=0x0004 class=0xff
```

kldstat -v | grep myfirst shows the driver loaded. devinfo -v | grep myfirst shows the device attached to pci0. pciconf -lv | grep myfirst confirms the match.

On unload:


```
myfirst0: detaching
```

And the device returns to unclaimed status. (Or to `virtio_random` if that module is reloaded.)

If the `virtio-rnd` device is absent, no attach happens; the driver loads, but no `myfirst0` appears in `devinfo`. If the driver is loaded on a host without the device, the same happens: probe runs for every PCI device in the system, probe returns `ENXIO` for each, and no attach occurs. This is the correct and expected behaviour; the driver is patient.

18.3.13 Common Mistakes at This Stage

A short list of traps the author has seen beginners fall into.

Forgetting `MODULE_DEPEND`. The driver loads, but early in boot it panics because the PCI module has not been initialised yet. Adding the declaration fixes it. The symptom is easy to recognise once you know to look for it.

Wrong name in `DRIVER_MODULE`. The name must match the "name" string in the `driver_t`. Mismatches produce subtle errors where the driver loads but never probes a device. The fix is to make the two match; the convention is that both use the driver's short name.

Returning the wrong value from probe. A beginner sometimes returns 0 from probe thinking "zero means success". Zero is `BUS_PROBE_SPECIFIC`, which is the strongest possible match; the driver will win over every other driver that wants the same device. This is almost never what you meant. Return `BUS_PROBE_DEFAULT` for the standard match.

Returning a positive error code. The newbus convention is that probe returns a negative priority value or a positive errno. Returning the wrong sign is a common typo. `ENXIO` is the right "I do not match" return.

Leaving resources allocated in probe. Probe must be side-effect-free. If probe allocates a resource, it must release it before returning. The cleanest approach is to never allocate from probe; do everything in attach.

Confusing `pci_get_vendor` with `pci_read_config`. The two are different. `pci_get_vendor` reads a cached ivar. `pci_read_config(dev, PCIR_VENDOR, 2)` reads the live configuration space. Both produce the same value for this field, but one is a cheap inline function and the other is a bus transaction. Use the accessor.

Forgetting to include the right headers. `dev/pci/pci.h` defines the `PCIR_*` constants. `dev/pci/pci-var.h` defines `pci_get_vendor` and friends. Both need to be included. The compiler error is usually "undefined identifier" for `pci_get_vendor`; the fix is the missing include.

Name collision with `MODULE_VERSION`. The first argument must match the driver name. `MODULE_VERSION(myfirst, 1)` is fine. `MODULE_VERSION(myfirst_pci, 1)` is not, because `myfirst_pci` is a file name, not a module name. The module loader looks up modules by the name registered in `DRIVER_MODULE`.

Each of these is recoverable. The debug kernel catches some of them (the loaded-before-pci case produces a panic the debug kernel pretty-prints). The others produce subtle wrong behaviour that is most easily caught by carefully testing the load-attach-detach-unload cycle after every change.

18.3.14 Checkpoint: Stage 1 Working

Before moving to Section 3, confirm that the Stage 1 driver works end-to-end.

On the bhyve or QEMU guest:

- `kldload virtio_pci` (if not already loaded).
- `kldunload virtio_random` (if loaded; fails gracefully if it isn't).
- `kldload ./myfirst.ko`.
- `kldstat -v | grep myfirst` should show the module loaded.
- `devinfo -v | grep myfirst` should show `myfirst0` attached under `pci0`.
- `dmesg` should show the attach banner.
- `kldunload myfirst`.
- `dmesg` should show the detach banner.
- `devinfo -v | grep myfirst` should show nothing.

If all of these pass, you have a working Stage 1 driver. The next step is to claim the BAR.

18.3.15 Wrapping Up Section 2

The probe-attach-detach sequence is the skeleton of every PCI driver. Section 2 built it in its smallest possible form: a probe that matches one vendor-and-device pair, an attach that prints a banner, a detach that prints another banner, and enough glue (`DRIVER_MODULE`, `MODULE_DEPEND`, `MODULE_VERSION`) to make the kernel's module loader and newbus enumerator accept it.

What the Stage 1 skeleton does not yet do: claim a BAR, read a register, enable bus mastering, walk the capability list, create a cdev, or coordinate the PCI build with the Chapter 17 simulation build. Every one of these is a topic of a later section in this chapter. The skeleton is important because every later topic plugs into it without reshaping it. Attach grows from a two-line function to a twenty-line function as the chapter proceeds; probe stays exactly as it is.

Section 3 introduces BARs. It explains what they are, how they are assigned, and how a driver claims the memory range a BAR describes. By the end of Section 3, the driver will hold a `struct resource *` for its BAR and a tag-and-handle pair ready to hand to the Chapter 16 accessor layer.

18.4 Section 3: Understanding and Claiming PCI Resources

With a probe and a trivial attach in place, the driver knows when it has found the device it wants to drive. What it does not yet know is how to reach that device's registers. Section 3 closes the gap. It starts with what a BAR is in the PCI specification, walks through how the firmware and the kernel set one up, and finishes with the driver code that claims a BAR and turns it into a `bus_space` tag and handle that the Chapter 16 accessors can use unchanged.

The point of this section is to make the word "BAR" concrete. A reader who finishes Section 3 should be able to answer, in a sentence: a BAR is a configuration-space field where a device says "here is how much memory (or how many I/O ports) I need, and here is how you can reach it once the firmware has mapped it into the host's address space". Everything else in the section builds on that sentence.

18.4.1 What a BAR Is, Precisely

Every PCI device advertises the resources it needs through Base Address Registers. A standard PCI device header (the non-bridge type) has six BARs, each four bytes wide, at configuration-space offsets 0x10, 0x14, 0x18, 0x1c, 0x20, and 0x24. In FreeBSD's `/usr/src/sys/dev/pci/pci.h`, these offsets are produced by the macro `PCIR_BAR(n)`, where `n` ranges from 0 to 5.

Each BAR describes one range of addresses. A BAR's low bit tells the software whether the range is in memory space or in I/O port space. If the low bit is zero, the range is memory-mapped; if it is one, the range is in the I/O port address space. Everything above the low few bits is an address; the exact field layout depends on the BAR type.

For a memory-mapped BAR, the layout is:

- Bit 0: 0 for memory.
- Bits 2-1: type. 0b00 for 32-bit, 0b10 for 64-bit, 0b01 reserved (formerly “below 1 MB”).
- Bit 3: prefetchable. 1 if the device promises that reads have no side effects, so the CPU may prefetch and merge accesses.
- Bits 31-4 (or 63-4 for 64-bit): the address.

A 64-bit BAR occupies two consecutive BAR slots. The low slot holds the lower 32 bits of the address (with the type bits); the high slot holds the upper 32 bits. A driver walking the BAR list must recognise when it has encountered a 64-bit BAR and skip the consumed upper slot.

For an I/O port BAR:

- Bit 0: 1 for I/O.
- Bit 1: reserved.
- Bits 31-2: the port address.

I/O port BARs are less common on modern devices. Most modern PCIe devices use memory-mapped BARs exclusively. Chapter 18 focuses on memory-mapped BARs.

18.4.2 How a BAR Gets an Address

A BAR is written in two passes. The first pass is what the silicon designer specified: a read from a BAR returns the device's requirements. The low bit type field is read-only. The address field is read-write, but with a catch: writing all ones to the address field and reading it back tells the firmware how large the range is. The device returns a value where the low bits (those below the size) are zero and the high bits (those that the device does not implement) return what was written. The firmware interprets the readback as a size mask.

The second pass assigns the actual address. The firmware (BIOS or UEFI) walks every BAR on every PCI device, notes the size each requires, partitions the host's address space to satisfy all of them, and writes the assigned address back into each BAR. By the time the OS boots, every BAR has a real address that the OS can use to reach the device.

The OS can optionally re-do the assignment if it wants to (for hot-plug support or if the firmware did a poor job). FreeBSD mostly accepts the firmware's assignment; the `hw.pci.realloc_bars` sysctl and the `bus_generic_probe` logic handle the uncommon case where reassignment is needed.

From the driver's point of view, all of this is done by the time attach runs. The BAR has an address, the address is mapped into the kernel's virtual space, and the driver needs only to ask for the resource by number.

18.4.3 The `rid` Argument and `PCIR_BAR`

The driver claims a BAR by calling `bus_alloc_resource_any(9)` with a resource ID (usually called `rid`) that identifies which BAR to allocate. For a memory-mapped BAR, the `rid` is the configuration-space offset of that BAR, produced by the macro `PCIR_BAR(n)`:

- `PCIR_BAR(0) = 0x10` (BAR 0)
- `PCIR_BAR(1) = 0x14` (BAR 1)
- ...
- `PCIR_BAR(5) = 0x24` (BAR 5)

Passing `PCIR_BAR(0)` to `bus_alloc_resource_any` requests BAR 0. Passing `PCIR_BAR(1)` requests BAR 1. The macro is one line in `pcireg.h`:

```
#define PCIR_BAR(x) (PCIR_BARS + (x) * 4)
```

where `PCIR_BARS` is `0x10`.

Beginners sometimes pass 0 or 1 as the `rid` and are surprised when the allocation fails. The `rid` is not a BAR index; it is the offset. Use `PCIR_BAR(index)` unless you have a specific reason to pass a raw offset.

18.4.4 The Resource Type: `SYS_RES_MEMORY` vs `SYS_RES_IOPORT`

`bus_alloc_resource_any` takes a type argument that tells the kernel what kind of resource the driver wants. For a memory BAR, the type is `SYS_RES_MEMORY`. For an I/O port BAR, the type is `SYS_RES_IOPORT`. For an interrupt, it is `SYS_RES_IRQ`. The small set of resource types is defined in `/usr/src/sys/arm64/include/resource.h` (and the per-architecture equivalents); memory, I/O port, and IRQ are the three a PCI driver normally uses.

PCI configuration space itself is not claimed through `bus_alloc_resource_any`. The driver reaches it through `pci_read_config(9)` and `pci_write_config(9)`, which route the access through the PCI bus driver without needing a resource handle.

A driver that does not know whether its BAR is memory or I/O can inspect the low bit of the BAR in configuration space to find out. A driver that knows (because the datasheet says so, or because the device has always been MMIO in this chapter) simply passes the right type.

Most PCIe devices have their main interface in memory space and an optional compatibility window in I/O port space. A driver usually asks for the memory BAR first and, if that fails, falls back to the I/O port BAR. Chapter 18's driver asks for memory only; the `virtio-rnd` device it targets exposes its registers in a memory BAR.

18.4.5 The `RF_ACTIVE` Flag

`bus_alloc_resource_any` also takes a flags argument. The two most commonly set flags are:

- `RF_ACTIVE`: activate the resource as part of allocation. Without this, the allocation reserves the resource but does not map it; the driver must call `bus_activate_resource(9)` separately. With it, the resource is allocated and activated in one step.
- `RF_SHAREABLE`: the resource can be shared with other drivers. This matters for interrupts (IRQs shared between multiple devices); it matters less for memory BARs.

For a memory BAR, the common case is `RF_ACTIVE` alone. For an IRQ that might be shared on a legacy system, it is `RF_ACTIVE | RF_SHAREABLE`. Chapter 18 uses `RF_ACTIVE` only.

18.4.6 `bus_alloc_resource_any` in Detail

The function signature is:

```
struct resource *bus_alloc_resource_any(device_t dev, int type,
    int *rid, u_int flags);
```

Three arguments, plus a return value.

`dev` is the device handle. `type` is `SYS_RES_MEMORY`, `SYS_RES_IOPORT`, or `SYS_RES_IRQ`. `rid` is a pointer to an integer holding the resource ID; the kernel may update it (for example, to tell the driver which slot was actually used when the driver passed a wildcard). `flags` is the bit mask described above.

The return value is a `struct resource *`. Non-NULL on success; NULL on failure. The resource handle is what every subsequent operation (reads, writes, release) uses.

A typical call looks like:

```
int rid = PCIR_BAR(0);
struct resource *bar;

bar = bus_alloc_resource_any(dev, SYS_RES_MEMORY, &rid, RF_ACTIVE);
if (bar == NULL) {
    device_printf(dev, "cannot allocate BAR0\n");
    return (ENXIO);
}
```

After the call, `bar` points to an allocated and activated resource; `rid` may have been updated by the kernel if it chose a different slot than the driver asked for (for wildcard allocations, this is where the chosen slot becomes visible).

18.4.7 From Resource to Tag and Handle

The resource handle is the driver's connection to the BAR, but the Chapter 16 accessor layer expects a `bus_space_tag_t` and a `bus_space_handle_t`, not a `struct resource *`. Two helpers turn one into the other:

- `rman_get_bustag(res)` returns the `bus_space_tag_t`.

- `rman_get_bushandle(res)` returns the `bus_space_handle_t`.

Both are inline accessor functions defined in `/usr/src/sys/sys/rman.h`. The resource stores the tag and handle internally; the accessors return the stored values. The driver then stores the tag and handle in its own state (in Chapter 18, in the hardware layer's `struct myfirst_hw`) so that the Chapter 16 accessors can use them.

The pattern is short:

```
sc->hw->regs_tag = rman_get_bustag(bar);
sc->hw->regs_handle = rman_get_bushandle(bar);
```

After these two lines, `CSR_READ_4(sc, off)` and `CSR_WRITE_4(sc, off, val)` work against the real BAR. No other code in the driver needs to know that the backend changed.

18.4.8 `rman_get_size` and `rman_get_start`

Two additional helpers extract the range of addresses the resource covers:

- `rman_get_size(res)` returns the number of bytes.
- `rman_get_start(res)` returns the physical or bus start address.

The driver uses `rman_get_size` to sanity-check that the BAR is large enough for the registers the driver expects. A device whose BAR is smaller than the chapter expects is either mis-identified (wrong device behind the ID pair) or a variant the driver does not support. Either way, a failed sanity check in `attach` is better than a corrupted access at run time.

`rman_get_start` is useful mostly for diagnostic logging. The physical address of the BAR is not something the driver dereferences directly (the kernel's mapping is what the tag and handle wrap), but printing it helps when debugging because it connects the `pciconf -lv` output to the driver's view.

18.4.9 Releasing the BAR

The counterpart to `bus_alloc_resource_any` is `bus_release_resource(9)`. The signature is:

```
int bus_release_resource(device_t dev, int type, int rid, struct resource *res);
```

`dev`, `type`, and `rid` match the allocation call; `res` is the handle returned by the allocation. On success, the function returns 0; on failure, it returns an `errno`. Failures are rare because the resource was just allocated by this driver, but defensive drivers check the return value and log on failure.

The driver should always release every resource it allocated, in the reverse order of allocation. Chapter 18's driver at Stage 2 allocates one BAR; it will release that one BAR in `detach`. Later stages, after interrupts and DMA enter the picture in Chapters 19 through 21, will allocate more.

18.4.10 Partial Failure in Attach

A subtle point about attach. If the driver claims the BAR successfully but then fails at a later step (for example, the device's expected `DEVICE_ID` register does not match), the driver must release the BAR before returning the error. Forgetting to release is a resource leak: the kernel's resource manager still thinks the BAR is allocated by this driver, even though the driver has returned. The next attempt to attach will fail.

The idiom is the familiar goto-based cleanup pattern:

```
static int
myfirst_pci_attach(device_t dev)
{
    struct myfirst_softc *sc = device_get_softc(dev);
    int rid, error;

    sc->dev = dev;
    sc->bar_rid = PCIR_BAR(0);
    sc->bar_res = bus_alloc_resource_any(dev, SYS_RES_MEMORY,
        &sc->bar_rid, RF_ACTIVE);
    if (sc->bar_res == NULL) {
        device_printf(dev, "cannot allocate BAR0\n");
        error = ENXIO;
        goto fail;
    }

    error = myfirst_hw_attach_pci(sc);
    if (error != 0)
        goto fail_release;

    /* ... */
    return (0);

fail_release:
    bus_release_resource(dev, SYS_RES_MEMORY, sc->bar_rid, sc->bar_res);
    sc->bar_res = NULL;
fail:
    return (error);
}
```

The goto cascade is an idiom, not a smell. It keeps cleanup code in one place and makes the allocation-and-release pairing symmetric. The pattern was introduced in Chapter 15 for mutex-and-callout cleanup; here it extends to resource cleanup. Chapter 18's final attach uses a longer version of this cascade to handle the softc init, BAR allocation, hardware-layer attach, and cdev creation as four staged allocations.

18.4.11 What Lives in the Softc

Chapter 18 adds a few fields to the softc. They are declared in `myfirst.h` (the main driver header, not `my-first_pci.h`, because the softc is shared across all the layers).

```
struct myfirst_softc {
    device_t dev;
    /* ... Chapter 10 through 17 fields ... */

    /* Chapter 18 PCI fields. */
    struct resource *bar_res;
    int bar_rid;
    bool pci_attached;
};
```

`bar_res` is the handle to the claimed BAR. `bar_rid` is the resource ID used to allocate it (stored so detach can pass the right value to `bus_release_resource`). `pci_attached` is a flag that later code uses to distinguish the real-PCI attach path from the simulated attach path.

A single BAR is enough for the Chapter 18 driver. Drivers for more complex devices would have `bar0_res`, `bar0_rid`, `bar1_res`, `bar1_rid`, and so on, each pair matching one BAR. The `virtio-rnd` device has only one BAR, so the driver has only one pair.

18.4.12 The Stage 2 Attach

Putting the allocation into Stage 2's attach routine:

```
static int
myfirst_pci_attach(device_t dev)
{
    struct myfirst_softc *sc = device_get_softc(dev);
    int error = 0;

    sc->dev = dev;

    /* Allocate BAR0 as a memory resource. */
    sc->bar_rid = PCIR_BAR(0);
    sc->bar_res = bus_alloc_resource_any(dev, SYS_RES_MEMORY,
        &sc->bar_rid, RF_ACTIVE);
    if (sc->bar_res == NULL) {
        device_printf(dev, "cannot allocate BAR0\n");
        return (ENXIO);
    }

    device_printf(dev, "BAR0 allocated: %#jx bytes at %#jx\n",
```



```

        (uintmax_t)rman_get_size(sc->bar_res),
        (uintmax_t)rman_get_start(sc->bar_res));

    sc->pci_attached = true;
    return (error);
}

```

A successful Stage 2 attach prints a line like:

```
myfirst0: BAR0 allocated: 0x20 bytes at 0xc1000000
```

The size and address depend on the guest's layout. The important parts are that the allocation succeeded, the size is what the driver expected (the virtio-rnd device exposes at least 32 bytes of registers), and the detach path releases the resource.

18.4.13 The Stage 2 Detach

The Stage 2 detach needs to release what attach allocated:

```

static int
myfirst_pci_detach(device_t dev)
{
    struct myfirst_softc *sc = device_get_softc(dev);

    if (sc->bar_res != NULL) {
        bus_release_resource(dev, SYS_RES_MEMORY, sc->bar_rid,
            sc->bar_res);
        sc->bar_res = NULL;
    }
    sc->pci_attached = false;
    device_printf(dev, "detaching\n");
    return (0);
}

```

The `if` guard is defensive: in principle `sc->bar_res` is non-NULL whenever detach is called after a successful attach, but adding the check costs nothing and makes the detach robust against partial-failure cases that might arise in future refactors. Setting `bar_res` to NULL after release prevents a double-free if something later calls detach again.

18.4.14 What Stage 2 Does Not Yet Do

At the end of Stage 2, the driver allocates a BAR but does nothing with it. The tag and handle are available but not yet wired to the Chapter 16 accessors. The simulation from Chapter 17 still runs, but it runs against the `malloc(9)`-allocated register block, not against the real BAR.

Section 4 bridges the gap. It takes the tag and handle from Stage 2 and hands them to `myfirst_hw_attach` so that `CSR_READ_4` and `CSR_WRITE_4` operate on real silicon. After Section 4, the Chapter 17 simulation becomes a runtime option rather than the only backend.

18.4.15 Verifying Stage 2

Before moving to Section 4, confirm that Stage 2 works end-to-end.

```
# In the bhyve guest:
sudo kldunload virtio_random # may not be loaded
sudo kldload ./myfirst.ko
sudo dmesg | grep myfirst | tail -5
```

The output should look like:

```
myfirst0: <Red Hat Virtio entropy source (myfirst demo target)> ... on pci0
myfirst0: attaching: vendor=0x1af4 device=0x1005 revid=0x00
myfirst0:          subvendor=0x1af4 subdevice=0x0004 class=0xff
myfirst0: BAR0 allocated: 0x20 bytes at 0xc1000000
```

`devinfo -v | grep -A 2 myfirst0` should show the resource claim:

```
myfirst0
  pnpinfo vendor=0x1af4 device=0x1005 ...
  resources:
    memory: 0xc1000000-0xc100001f
```

The memory range printed by `devinfo -v` matches the range printed by the driver. This confirms that the allocation succeeded and that the kernel sees the BAR as claimed by `myfirst`.

Unload and verify the cleanup:

```
sudo kldunload myfirst
sudo devinfo -v | grep myfirst # should return nothing
```

No lingering devices, no leaked resources. Stage 2 is complete.

18.4.16 Common Mistakes in BAR Allocation

A short list of the typical traps.

Passing 0 as the rid for BAR 0. The rid is `PCIR_BAR(0) = 0x10`, not 0. Passing 0 asks for a resource at offset 0, which is the `PCIR_VENDOR` field; the allocation fails or produces surprising results. Always use `PCIR_BAR(index)`.

Forgetting `RF_ACTIVE`. Without this flag, `bus_alloc_resource_any` allocates but does not activate. Reading from the tag and handle at that point is undefined behaviour. The symptom is usually a page fault or garbage values. The fix is to pass `RF_ACTIVE`.

Using the wrong resource type. Passing `SYS_RES_IOPORT` for a memory BAR produces an immediate allocation failure. Passing `SYS_RES_MEMORY` for an I/O port BAR does the same. The type must match the BAR's actual type. If the driver does not know in advance (a generic driver that supports both memory and I/O variants), it reads `PCIR_BAR(index)` from configuration space and checks the low bit.

Not releasing on partial failure. A common beginner mistake: `attach` claims the BAR, a subsequent step fails, the function returns the error, and the BAR is never released. The resource leaks. The next `attach` attempt fails because the BAR is still claimed.

Releasing the BAR before the accessor layer is done with it. The reverse mistake: `detach` releases the BAR too early, before draining callouts or tasks that might still be reading from it. The symptom is a page fault inside a callout shortly after `kldunload`. The fix is to drain everything that might access the BAR, then release.

Confusing `rman_get_size` with `rman_get_end`. `rman_get_size(res)` returns the number of bytes. `rman_get_end(res)` returns the last byte's address (start plus size minus one). Use `rman_get_size` for sanity checks on BAR size; use `rman_get_start` and `rman_get_end` for diagnostic printing.

Assuming BARs are in any particular order. The driver must name the BAR it wants explicitly (by passing `PCIR_BAR(n)`). Some devices put their main BAR at index 0; some put it at index 2. The datasheet (or the `pciconf -lv` output for the specific device) says where it is. Assuming BAR 0 without verifying is a common mistake.

18.4.17 A Note on 64-bit BARs

The `virtio-rnd` device used in Chapter 18 has a 32-bit BAR, so the allocation shown here works without special handling. For devices with 64-bit BARs, there are two important details:

First, the BAR occupies two consecutive slots in the configuration-space BAR table. BAR 0 (at offset `0x10`) holds the lower 32 bits; BAR 1 (at offset `0x14`) holds the upper 32 bits. A driver walking the BAR table by simple integer increments would mistakenly treat BAR 1 as a separate BAR. The right walk reads each BAR's type bits and skips the next slot if the current slot is a 64-bit BAR.

Second, the `rid` passed to `bus_alloc_resource_any` is the lower slot's offset. The kernel recognises the 64-bit type and treats the pair as a single resource. The driver does not need to allocate two resources for a 64-bit BAR; one allocation with `rid = PCIR_BAR(0)` handles both slots.

For Chapter 18's driver, this is academic; the target device has 32-bit BARs. But a reader who later works with a device that has a 64-bit BAR will need the details. `/usr/src/sys/dev/pci/pci.h` defines `PCIM_BAR_MEM_TYPE`, `PCIM_BAR_MEM_32`, and `PCIM_BAR_MEM_64` to help with BAR inspection.

18.4.18 Prefetchable vs Non-Prefetchable BARs

A related detail. A BAR is prefetchable if its bit 3 is set. Prefetchable means "reads from this range have no side effects, so the CPU is allowed to cache, prefetch, and merge accesses as it would for normal RAM". Non-prefetchable means "reads have side effects, so every access must hit the device; the CPU must not cache, prefetch, or merge".

Device registers are almost always non-prefetchable. A read from a status register may clear the flags; a prefetched read would be a catastrophic bug. Device memory (a frame buffer on a graphics card, or a ring buffer on a NIC) is usually prefetchable.

The driver does not directly control the prefetch attribute; the BAR declares what it is, and the kernel sets up the mapping accordingly. The driver's job is to use `bus_space_read_*` and `bus_space_write_*` correctly. The `bus_space` layer handles the ordering and caching details. A driver that tries to be clever by bypassing `bus_space` and directly dereferencing a pointer might accidentally get a cached mapping on a non-prefetchable BAR and produce a driver that works in ideal conditions and fails mysteriously under load.

Chapter 16 made the case for `bus_space` in general; Section 3 of Chapter 18 confirms that the case extends to real PCI devices. There is no shortcut.

18.4.19 Wrapping Up Section 3

A BAR is a range of addresses where the device exposes its registers. The firmware assigns BAR addresses at boot; the kernel reads them during PCI enumeration; the driver claims them in `attach` through `bus_alloc_resource_any(9)` with the right type, `rid`, and flags. The returned `struct resource *` carries a `bus_space_tag_t` and `bus_space_handle_t` that `rman_get_bustag(9)` and `rman_get_bushandle(9)` extract. `Detach` must release every allocated resource in reverse order.

The Stage 2 driver allocates BAR 0 but does not yet use it. Section 4 wires the tag and handle to the Chapter 16 accessor layer, so `CSR_READ_4` and `CSR_WRITE_4` finally operate on the real BAR rather than on a `malloc(9)` block.

18.5 Section 4: Accessing Device Registers via `bus_space(9)`

Section 3 ended with a tag and a handle in the driver's hand. The tag and handle point at the real BAR; the Chapter 16 accessors expect exactly that pair. Section 4 makes the connection. It teaches how to hand the PCI-allocated tag and handle to the hardware layer, confirms that the Chapter 16 `CSR_*` macros work unchanged against a real PCI BAR, reads the driver's first real register through `bus_space_read_4(9)`, and discusses the access patterns (`bus_space_read_multi`, `bus_space_read_region`, barriers) that Chapter 16 introduced and that the PCI path will reuse.

The theme of Section 4 is continuity. The reader has been writing register-access code for two chapters. Chapter 18 does not change that code. What changes is where the tag and handle come from. The accessors are exactly the accessors; the wrapping macros are exactly the wrapping macros; the lock discipline is exactly the lock discipline. This is the payoff for the `bus_space(9)` abstraction. The driver's upper layers do not know, and should not know, that the register block's origin has changed.

18.5.1 Revisiting the Chapter 16 Accessors

A short reminder. `myfirst_hw.c` defines three public functions that the rest of the driver calls:

- `myfirst_reg_read(sc, off)` returns a 32-bit value from the register at the given offset.
- `myfirst_reg_write(sc, off, val)` writes a 32-bit value to the register at the given offset.

- `myfirst_reg_update(sc, off, clear, set)` performs a read-modify-write: read, clear the given bits, set the given bits, write.

All three are wrapped by the `CSR_*` macros defined in `myfirst_hw.h`:

- `CSR_READ_4(sc, off)` expands to `myfirst_reg_read(sc, off)`.
- `CSR_WRITE_4(sc, off, val)` expands to `myfirst_reg_write(sc, off, val)`.
- `CSR_UPDATE_4(sc, off, clear, set)` expands to `myfirst_reg_update(sc, off, clear, set)`.

The accessors reach `bus_space` through two fields in struct `myfirst_hw`:

- `hw->regs_tag` of type `bus_space_tag_t`
- `hw->regs_handle` of type `bus_space_handle_t`

The actual call inside `myfirst_reg_read` is:

```
value = bus_space_read_4(hw->regs_tag, hw->regs_handle, offset);
```

and inside `myfirst_reg_write`:

```
bus_space_write_4(hw->regs_tag, hw->regs_handle, offset, value);
```

These lines do not know about PCI. They do not know about `malloc`. They do not know whether `hw->regs_tag` came from a simulated pmap setup in Chapter 16 or from a `rman_get_bustag(9)` call in Chapter 18. Their contract is unchanged.

18.5.2 The Two Origins of a Tag and Handle

Chapter 16 used a trick to produce a tag and handle from a `malloc(9)` allocation on x86. The trick was simple: the x86 `bus_space` implementation uses `x86_bus_space_mem` as the tag for memory-mapped accesses, and a handle is just a virtual address. A `malloc`-allocated buffer has a virtual address, so casting the buffer pointer to `bus_space_handle_t` produces a usable handle. The trick is x86-specific; on other architectures, a simulated block would need a different approach.

Chapter 18 uses the proper route: `bus_alloc_resource_any(9)` allocates a BAR as a resource, and `rman_get_bustag(9)` and `rman_get_bushandle(9)` extract the tag and handle that the kernel set up. The driver does not see the physical address; it does not see the virtual mapping; it sees an opaque tag and handle that the kernel's platform code set up correctly. The accessors use them, and the register read hits the real device.

This is the fundamental shape of the PCI integration. Two different origins of the tag and handle. One set of accessors that uses them. The driver picks which origin is live at attach time, and the accessors do not need to know which one was picked.

18.5.3 Extending myfirst_hw_attach

Chapter 16's `myfirst_hw_attach` allocates a `malloc(9)` buffer and synthesises a tag and handle. Chapter 18 needs a second code path that takes an existing tag and handle (from the PCI BAR) and stores them directly. The simplest way is to rename the Chapter 16 version and introduce a new version for the PCI path.

The new header, adjusted for Chapter 18:

```
/* Chapter 16 behaviour: allocate a malloc-backed register block. */
int myfirst_hw_attach_sim(struct myfirst_softc *sc);

/* Chapter 18 behaviour: use an already-allocated resource. */
int myfirst_hw_attach_pci(struct myfirst_softc *sc,
    struct resource *bar, bus_size_t bar_size);

/* Shared teardown; safe with either backend. */
void myfirst_hw_detach(struct myfirst_softc *sc);
```

The PCI-path attach stores the tag and handle directly:

```
int
myfirst_hw_attach_pci(struct myfirst_softc *sc, struct resource *bar,
    bus_size_t bar_size)
{
    struct myfirst_hw *hw;

    if (bar_size < MYFIRST_REG_SIZE) {
        device_printf(sc->dev,
            "BAR is too small: %ju bytes, need at least %u\n",
            (uintmax_t)bar_size, (unsigned)MYFIRST_REG_SIZE);
        return (ENXIO);
    }

    hw = malloc(sizeof(*hw), M_MYFIRST, M_WAITOK | M_ZERO);

    hw->regs_buf = NULL;           /* no malloc block */
    hw->regs_size = (size_t)bar_size;
    hw->regs_tag = rman_get_bustag(bar);
    hw->regs_handle = rman_get_bushandle(bar);
    hw->access_log_enabled = true;
    hw->access_log_head = 0;

    sc->hw = hw;

    device_printf(sc->dev,
```

```

    "hardware layer attached to BAR: %zu bytes "
    "(tag=%p handle=%p)\n",
    hw->regs_size, (void *)hw->regs_tag,
    (void *)hw->regs_handle);
    return (0);
}

```

A few things to notice. `hw->regs_buf` is `NULL` because there is no `malloc` allocation backing the registers this time; the BAR's kernel mapping is what the tag and handle point at. `hw->regs_size` is the BAR's size, sanity-checked against the minimum size the driver expects. The tag and handle come from the struct `resource *` that the PCI attach path allocated. Everything else in `myfirst_hw` is unchanged.

The shared detach is where the two backends converge:

```

void
myfirst_hw_detach(struct myfirst_softc *sc)
{
    struct myfirst_hw *hw;

    if (sc->hw == NULL)
        return;

    hw = sc->hw;
    sc->hw = NULL;

    /*
     * Free the simulated backing buffer only if the simulation
     * attach produced one. The PCI path sets regs_buf to NULL and
     * leaves regs_size as the BAR size; the BAR itself is released
     * by the PCI layer (see myfirst_pci_detach).
     */
    if (hw->regs_buf != NULL) {
        free(hw->regs_buf, M_MYFIRST);
        hw->regs_buf = NULL;
    }
    free(hw, M_MYFIRST);
}

```

The split is clean. The hardware layer knows how to tear down the Chapter 16 backing buffer or do nothing, depending on how it was attached. The BAR itself is not the hardware layer's responsibility; the PCI layer owns it. This separation is what lets Chapter 18 reuse the Chapter 16 code without rewriting the Chapter 16 teardown.

18.5.4 The First Real Register Read

With the hardware layer wired up, the driver's first real read becomes possible. At Chapter 17 the first read was of the fixed `DEVICE_ID` register, which the simulation pre-populated with `0x4D594649` ("MYFI" in ASCII, for "MY First"). The `virtio-rnd` device does not expose a `DEVICE_ID` register at that offset; its configuration space at the BAR's offset 0 is a `virtio`-specific layout that begins with a device features register.

For the Chapter 18 teaching path, we do not need to speak the `virtio-rnd` protocol. The driver reads the first 32-bit word of the BAR and logs the value. The value is whatever the `virtio-rnd` device's first register is (the first 32 bits of the `virtio` legacy device configuration, which for a running `virtio-rnd` device has no particular meaning to our driver). The point of the read is to prove that the BAR access works.

The code that does this (in `myfirst_pci_attach`, after the BAR allocation and the hardware layer attach):

```
uint32_t first_word;

MYFIRST_LOCK(sc);
first_word = CSR_READ_4(sc, 0x00);
MYFIRST_UNLOCK(sc);

device_printf(dev, "first register read: 0x%08x\n", first_word);
```

The lock-and-unlock wrapping is the Chapter 16 discipline. The read goes through `bus_space_read_4` under the hood. The output line appears in `dmesg` at attach time:

```
myfirst0: first register read: 0x10010000
```

The exact value depends on the `virtio-rnd` device's current state. A reader who sees any value (rather than a page fault or a garbage read that crashes the guest) has confirmed that the BAR allocation worked, the tag and handle are correct, and the accessor layer is operating on real silicon.

18.5.5 The Full Accessor Family

The Chapter 16 driver used `bus_space_read_4` and `bus_space_write_4` exclusively because the register map is all 32-bit registers. Real PCI devices sometimes need 8-bit, 16-bit, or 64-bit reads, and sometimes need block operations that read or write many contiguous registers at once. The `bus_space` family covers all of these:

- `bus_space_read_1, _2, _4, _8`: single-byte, 16-bit, 32-bit, or 64-bit read.
- `bus_space_write_1, _2, _4, _8`: single-byte, 16-bit, 32-bit, or 64-bit write.
- `bus_space_read_multi_*`: read multiple values from the same register offset (useful for FIFO reads).
- `bus_space_write_multi_*`: write multiple values to the same register offset.
- `bus_space_read_region_*`: read a range of registers into a memory buffer.
- `bus_space_write_region_*`: write a memory buffer to a range of registers.
- `bus_space_set_multi_*`: write the same value to the same register many times.
- `bus_space_set_region_*`: write the same value to a range of registers.
- `bus_space_barrier`: enforce ordering between accesses.

Each variant has the width suffix as a separate entry. The family is symmetrical and predictable once you have seen it.

For Chapter 18's driver, only `_4` is needed. The register map is 32-bit throughout. If a later driver uses a device with 16-bit registers, the reader simply changes `_4` to `_2`. The `CSR_*` macros can be extended to cover multiple widths if needed:

```
#define CSR_READ_1(sc, off)      myfirst_reg_read_1((sc), (off))
#define CSR_READ_2(sc, off)      myfirst_reg_read_2((sc), (off))
#define CSR_WRITE_1(sc, off, val) myfirst_reg_write_1((sc), (off), (val))
#define CSR_WRITE_2(sc, off, val) myfirst_reg_write_2((sc), (off), (val))
```

with corresponding accessor functions in `myfirst_hw.c`. Chapter 18 does not need these, but a driver author should know they exist.

18.5.6 `bus_space_read_multi` vs `bus_space_read_region`

Two of the block operations deserve a second look because the naming is easy to confuse.

`bus_space_read_multi_4(tag, handle, offset, buf, count)` reads `count` 32-bit values, all from the same offset in the BAR, into `buf`. This is the correct operation for a FIFO: the register at a fixed offset is the FIFO's read port, and each read consumes one entry. Writing a similar loop by hand with `bus_space_read_4` would work, but the block version is often faster and is clearer in intent.

`bus_space_read_region_4(tag, handle, offset, buf, count)` reads `count` 32-bit values from consecutive offsets starting at `offset`, into `buf`. This is the correct operation for a block of registers: the driver wants to snapshot a range of the register map into a local buffer. Writing a loop with `bus_space_read_4` and incrementing the offset would work equivalently; the block version expresses the intent more clearly.

The difference is whether the offset in the BAR advances. `_multi` keeps the offset fixed. `_region` advances it. Writing `_multi` when you meant `_region` reads the same register four times, not four different registers. This is a classic confusion, and the way to avoid it is to read the variant name carefully and to remember “multi = one port, many accesses” vs “region = a range of ports, one access each”.

18.5.7 When Barriers Matter

Chapter 16 introduced `bus_space_barrier(9)` as a guard against CPU reordering and compiler reordering around register accesses. The rule is: when the driver has an ordering requirement between two accesses (a write that must precede a read, for example, or a write that must precede another write), insert a barrier.

For Chapter 18's driver, the accessor layer already wraps a barrier around writes that have side effects (inside `myfirst_reg_write_barrier`, defined in Chapter 16). The simulation backend in Chapter 17 does not require additional barriers because the accesses are to RAM. The PCI backend may require barriers in more places than the simulation did, because real device memory has weaker ordering semantics than RAM on some architectures.

The common case on x86: `bus_space_write_4` to a memory-mapped BAR is strongly ordered with respect to other writes to the same BAR; no explicit barrier is needed. On arm64 with device-memory attributes, writes

to the same BAR are also ordered. On other architectures with weaker memory models, explicit barriers may be required. The `bus_space(9)` manual page specifies the default ordering guarantees per architecture; drivers that care about portability include barriers even where x86 would not require them.

Chapter 18's driver lives on x86 for teaching purposes and uses barriers the same way Chapter 16 did: after a CTRL write that has side effects (starts a command, triggers a state change, clears an interrupt). The Chapter 17 `myfirst_reg_write_barrier` helper is still the right entry point.

18.5.8 The Access Log on a Real BAR

Chapter 16's access log is a ring buffer that records every register access with a timestamp, an offset, a value, and a context tag. On the simulation backend the log shows patterns like “user-space write to CTRL, followed by a callout read of STATUS”. On the real PCI backend the log shows the same shape: any access the driver makes to the BAR passes through the accessors, and every accessor writes an entry.

This continuity is a quiet but important feature. A developer who debugs a simulation issue can consult the access log; a developer who debugs a real-hardware issue can consult the same access log. The technique transfers. The code does not change. Section 7's testing discipline depends on this continuity.

One note about the access log and real BARs: if the device sometimes takes side effects on reads (clearing a latched status bit, advancing a FIFO pointer, triggering a posted write completion), the log will record the read's value and the driver's subsequent actions. Reading the log can reveal timing problems that are not otherwise visible. A bug where the driver reads STATUS twice in rapid succession and the second read sees different bits because the first read's side effect interfered will show up clearly in the log. For Chapter 18 this does not matter yet; for Chapter 19 and later chapters it matters a great deal.

18.5.9 A Small Subtlety: the CSR Macros Do Not Know About PCI

Worth calling out. The `CSR_*` macros do not take a tag or a handle. They take only the `softc` and the offset. Everything else is inside the accessor functions.

This means: when the driver changes from the Chapter 17 simulation to the Chapter 18 real BAR, not one call site in the driver changes. `CSR_READ_4(sc, MYFIRST_REG_STATUS)` does the right thing before and after the transition. The same is true for `CSR_WRITE_4` and `CSR_UPDATE_4`.

The payoff is concrete. Chapter 17's driver has probably thirty or forty call sites that read or write registers through the CSR macros. If those macros took a tag and a handle, Chapter 18 would need to update every one. Because they take only the `softc`, Chapter 18 needs to change only the hardware layer's attach routine. The discipline of hiding the low-level details inside the accessors, introduced in Chapter 16 and maintained in Chapter 17, pays its largest dividend here.

This is a pattern worth remembering. When you write a driver, define a small set of accessor functions that hide everything above the register level: the tag, the handle, the lock, the log, the barrier. Expose to the rest of the driver only the `softc` and the offset. The code that uses the accessors then does not care whether the registers are simulated, real PCI, real USB, real I2C, or real anything else. The abstraction holds across a wide range of transports. Part 7 of the book will return to this theme when it discusses refactoring drivers for portability; Chapter 18 is where the reader first sees the dividend paid.

18.5.10 Stage 2 to Stage 3: Wiring It Together

The Stage 2 attach allocated the BAR but did not hand it to the hardware layer. Stage 3's attach does both. The relevant code is the full attach:

```
static int
myfirst_pci_attach(device_t dev)
{
    struct myfirst_softc *sc = device_get_softc(dev);
    int error;

    sc->dev = dev;
    error = myfirst_init_softc(sc); /* Ch10-15: locks, softc fields */
    if (error != 0)
        return (error);

    /* Allocate BAR0. */
    sc->bar_rid = PCIR_BAR(0);
    sc->bar_res = bus_alloc_resource_any(dev, SYS_RES_MEMORY,
        &sc->bar_rid, RF_ACTIVE);
    if (sc->bar_res == NULL) {
        device_printf(dev, "cannot allocate BAR0\n");
        error = ENXIO;
        goto fail_softc;
    }

    /* Hand the BAR to the hardware layer. */
    error = myfirst_hw_attach_pci(sc, sc->bar_res,
        rman_get_size(sc->bar_res));
    if (error != 0)
        goto fail_release;

    /* Read a diagnostic word from the BAR. */
    MYFIRST_LOCK(sc);
    sc->bar_first_word = CSR_READ_4(sc, 0x00);
    MYFIRST_UNLOCK(sc);
    device_printf(dev, "BAR[0x00] = 0x%08x\n", sc->bar_first_word);

    sc->pci_attached = true;
    return (0);

fail_release:
    bus_release_resource(dev, SYS_RES_MEMORY, sc->bar_rid, sc->bar_res);
    sc->bar_res = NULL;
fail_softc:
```

```
    myfirst_deinit_softc(sc);  
    return (error);  
}
```

And the matching detach:

```
static int  
myfirst_pci_detach(device_t dev)  
{  
    struct myfirst_softc *sc = device_get_softc(dev);  
  
    sc->pci_attached = false;  
    myfirst_hw_detach(sc);  
    if (sc->bar_res != NULL) {  
        bus_release_resource(dev, SYS_RES_MEMORY, sc->bar_rid,  
                             sc->bar_res);  
        sc->bar_res = NULL;  
    }  
    myfirst_deinit_softc(sc);  
    device_printf(dev, "detaching\n");  
    return (0);  
}
```

The attach sequence is strict: init the softc (locks, fields), allocate the BAR, attach the hardware layer against the BAR, perform any register reads that attach needs, mark the driver as attached. The detach undoes each step in reverse: mark as unattached, detach the hardware layer (which frees its wrapper struct), release the BAR, deinit the softc.

Section 5 will extend the attach with additional PCI-specific steps: enabling bus mastering, walking the capability list, reading a subvendor-specific field from configuration space. The shape of the attach stays the same; the middle grows.

18.5.11 Common Mistakes in the bus_space-on-PCI Transition

A short list of traps.

Casting the resource pointer instead of using `rman_get_bustag` / `rman_get_bushandle`. A beginner sometimes writes `hw->regs_tag = (bus_space_tag_t)bar`. This does not compile on most architectures and compiles to nonsense on the rest. Use the accessors.

Confusing the resource handle with the tag. The tag is the bus identity (memory or I/O); the handle is the address. `rman_get_bustag` returns the tag; `rman_get_bushandle` returns the handle. Swapping them produces immediate crashes or silent wrong reads. Read the function names carefully.

Not zeroing the hardware state on PCI attach. `malloc(9)` with `M_ZERO` clears the structure. Without `M_ZERO`, fields like `access_log_head` start with garbage. The ring buffer wraps to an arbitrary index and the log is unreadable.

Not releasing the hardware state in detach. A symmetry mistake: the PCI detach releases the BAR but forgets to call `myfirst_hw_detach`. The hardware wrapper struct leaks. `vmstat -m` shows the leak over time.

Reading the BAR before the lock is held. The Chapter 16 discipline is: every CSR access is under `sc->mtx`. Reading at attach time without the lock violates the invariant that every later access assumes. Even if it happens to work on a single CPU, WITNESS on a debug kernel will complain. Take the lock even for the attach-time reads.

Accidentally writing to a read-only register. On the simulation backend, writes to a read-only register just update the `malloc`-allocated buffer (the simulation's read side ignores the write and returns the fixed value). On real PCI, writes to a read-only register are either silently ignored or cause some device-specific side effect. Neither case is what the driver author expects. Read the datasheet and write to writeable registers only.

Calling `bus_space_read_multi_4` when the driver meant `_region_4`. The two functions have identical signatures and very different semantics. Reading a range of registers with `_multi` fills the buffer with the same value (the fixed offset's current value) repeated count times. Reading a range with `_region` fills the buffer with the consecutive register values. The bug is silent until the values are inspected.

18.5.12 Wrapping Up Section 4

The Chapter 16 accessor layer is unchanged by the transition from simulated to real PCI registers. The only change is in `myfirst_hw_attach_pci`, which replaces the `malloc(9)` backing buffer with a tag and handle produced by `rman_get_bustag(9)` and `rman_get_bushandle(9)` on the PCI-allocated resource. The `CSR_*` macros, the access log, the lock discipline, the ticker task, and every other part of the Chapter 16 and Chapter 17 code base continue to work without modification.

The driver's first real PCI register read happens at attach time. The value read is not meaningful in the virtio-rnd protocol sense; it is proof that the BAR mapping is live and the accessors are reading real silicon. Section 5 takes the attach sequence further: it introduces `pci_enable_busmaster(9)` (for future DMA use), walks the PCI capability list with `pci_find_cap(9)` and `pci_find_extcap(9)`, explains when a driver reads configuration-space fields directly, and shows how the Chapter 17 simulation is kept inactive on the PCI path so its callouts do not write arbitrary values into the real device's registers.

18.6 Section 5: Driver Attach-Time Initialisation

Sections 2 through 4 built the attach routine from nothing to a fully wired PCI attach that claims a BAR, hands it to the hardware layer, and performs its first register access. Section 5 finishes the attach story. It introduces the handful of configuration-space operations a PCI driver typically performs at attach, explains when and why each is needed, walks the PCI capability list to discover the device's optional features, shows how the Chapter 17 simulation is kept inactive on the PCI path (so its callouts do not write into the real device), and creates the `cdev` that the Chapter 10 driver already exposed.

By the end of Section 5, the driver is complete as a PCI driver. It attaches to the real device, brings the device to a state where the driver can use it, exposes the same user-space interface the Chapter 10 through 17 iterations exposed, and is ready to be extended in Chapter 19 with a real interrupt handler.

18.6.1 The Attach-Time Checklist

A working PCI driver's attach routine typically performs, in roughly this order:

1. **Initialise the softc.** Set `sc->dev`, initialise locks, initialise conditions and callouts, clear counters.
2. **Allocate resources.** Claim the BAR (or BARs), claim the IRQ resource (in Chapter 19), and any other bus resources.
3. **Activate device features.** Enable bus mastering if the driver will use DMA. Set up configuration-space bits the device needs.
4. **Walk capabilities.** Find PCI capabilities the driver supports and record their register offsets.
5. **Attach the hardware layer.** Hand the BAR to the accessor layer.
6. **Initialise the device.** Perform the device-specific bring-up sequence: reset, feature negotiation, queue setup. This is whatever the device's datasheet says is required to make the device usable.
7. **Register user-space interfaces.** Create `cdevs`, network interfaces, or whatever the driver exposes.
8. **Enable interrupts.** Register the interrupt handler (Chapter 19) and unmask interrupts in the device's `INTR_MASK` register.
9. **Mark the driver as attached.** Set a flag that other code can check.

Not every driver does every step. A driver for a passive device (no DMA, no interrupts, just reads and writes) skips bus mastering and interrupt setup. A driver for a device that does not need a user-space interface skips the `cdev` creation. But the order is stable: resources first, device features second, hardware layer third, device bring-up fourth, user-space fifth, interrupts last. Doing this out of order produces race conditions where an interrupt arrives before the driver is ready to handle it, or where a user-space access reaches a partially-initialised driver.

Chapter 18's driver performs steps 1 through 7. Step 8 is Chapter 19. Step 9 is a detail the driver already handled in Chapter 10.

18.6.2 `pci_enable_busmaster` and the Command Register

The PCI command register lives at configuration-space offset `PCIR_COMMAND` (`0x04`), as a 16-bit field. Three bits in that register matter for most drivers:

- `PCIM_CMD_MEMEN` (`0x0002`): enable the device's memory BARs. Must be set before the driver can read or write any memory BAR.
- `PCIM_CMD_PORTEN` (`0x0001`): enable the device's I/O port BARs. Must be set before the driver can read or write any I/O port BAR.
- `PCIM_CMD_BUSMASTEREN` (`0x0004`): enable the device to initiate DMA as a bus master. Must be set before the device can read or write RAM on its own.

The PCI bus driver sets `MEMEN` and `PORTEN` automatically when it activates a BAR. A driver that has successfully called `bus_alloc_resource_any` with `RF_ACTIVE` and has received a non-NULL result does not need to set these bits by hand; the bus driver has already done so.

`BUSMASTEREN` is different. The bus driver does not set it automatically, because not every driver needs DMA. A driver that will program its device to read or write system RAM (a NIC, a storage controller, a GPU) must set `BUSMASTEREN` explicitly. A driver that only reads and writes the device's own BARs (no DMA) does not need to set it.

The helper `pci_enable_busmaster(dev)` sets the bit. Its inverse, `pci_disable_busmaster(dev)`, clears it. Chapter 18's driver does not use DMA and does not call `pci_enable_busmaster`. Chapter 20 and Chapter 21 will.

A note about reading the command register directly. The driver can always read the command register with `pci_read_config(dev, PCIR_COMMAND, 2)` and inspect individual bits. For most drivers this is unnecessary; the kernel has already configured the relevant bits. For diagnostic purposes (a driver that wants to log the device's command register state at attach), it is fine.

18.6.3 Reading Configuration-Space Fields

Most drivers need to read at least a handful of configuration-space fields that the generic accessors do not cover. Examples include:

- Specific firmware revision numbers at vendor-specific offsets.
- PCIe link status fields inside the PCIe capability structure.
- Vendor-specific capability data.
- Subsystem-specific identification fields for multi-function devices.

The primitive is `pci_read_config(dev, offset, width)`. Offset is a byte offset into configuration space. Width is 1, 2, or 4 bytes. The return value is a `uint32_t` (narrower widths are right-aligned).

A concrete example. The PCI class code occupies bytes 0x09 through 0x0b in configuration space:

- Byte 0x09: programming interface (progIF).
- Byte 0x0a: subclass.
- Byte 0x0b: class.

Reading all three at once as a 32-bit value gives the class, subclass, and progIF in the top three bytes (the low byte is the revision ID). The cached accessors `pci_get_class`, `pci_get_subclass`, `pci_get_progif`, and `pci_get_revid` extract each field individually; the driver rarely needs to do this by hand.

For vendor-specific fields the driver must read by hand. The pattern is:

```
uint32_t fw_rev = pci_read_config(dev, 0x48, 4);
device_printf(dev, "firmware revision 0x%08x\n", fw_rev);
```

The offset 0x48 is a placeholder; the real offset is whatever the device's datasheet specifies. Reading from an offset the device does not implement returns 0xffffffff or a device-specific default; 0xffffffff is the classic "no device" value on PCI.

18.6.4 pci_write_config and the Side-Effect Contract

The counterpart is `pci_write_config(dev, offset, value, width)`. It writes `value` to the configuration-space field at `offset`, truncated to `width` bytes.

A critical point about configuration-space writes: some fields are read-only. Writing to a read-only field is either silently ignored (the common case) or causes a device-specific error. The driver must know, from the PCI specification or from the device's datasheet, which fields are writeable before issuing a write.

A second critical point: some fields have side effects on read or write. The command register, for example, has side effects: setting `MEMEN` enables memory BARs; clearing it disables them. Reading the command register has no side effects. The driver must understand the semantics of each field it touches.

The helper `pci_enable_busmaster` uses `pci_write_config` under the hood to set one bit. The driver can always use `pci_read_config` and `pci_write_config` directly to manipulate a field when a specific helper does not exist.

18.6.5 `pci_find_cap`: Walking the Capability List

PCI devices advertise optional features through a linked list of capabilities. Each capability is a small block in configuration space, beginning with a one-byte capability ID and a one-byte “next pointer”. The list starts at the offset stored in the device’s `PCIR_CAP_PTR` field (configuration-space offset 0x34) and follows the next pointers until a 0 terminates the chain.

Standard capabilities a driver might find include:

- `PCIY_PMG` (0x01): Power Management.
- `PCIY_MSI` (0x05): Message Signaled Interrupts.
- `PCIY_EXPRESS` (0x10): PCI Express. Any PCIe device has this.
- `PCIY_MSIX` (0x11): MSI-X. A richer interrupt routing mechanism than MSI.
- `PCIY_VENDOR` (0x09): vendor-specific capability.

The driver walks the list through `pci_find_cap(9)`:

```
int capreg;

if (pci_find_cap(dev, PCIY_EXPRESS, &capreg) == 0) {
    device_printf(dev, "PCIe capability at offset 0x%x\n", capreg);
}
if (pci_find_cap(dev, PCIY_MSI, &capreg) == 0) {
    device_printf(dev, "MSI capability at offset 0x%x\n", capreg);
}
if (pci_find_cap(dev, PCIY_MSIX, &capreg) == 0) {
    device_printf(dev, "MSI-X capability at offset 0x%x\n", capreg);
}
```

The function returns 0 on success and stores the capability’s offset in `*capreg`. On failure (the capability is not present) it returns `ENOENT` and does not modify `*capreg`.

The offset returned is the byte offset into configuration space where the capability’s first register lives. That register is usually the capability ID itself; the driver can confirm by reading it back and checking against the expected ID. Subsequent bytes in the capability define the feature-specific fields.

Chapter 18’s driver walks the capability list at attach time and logs which capabilities are present. The list gives the driver author a sense of what the device offers. MSI and MSI-X are relevant for Chapter 20; Power Management is relevant for Chapter 22. At Chapter 18, the driver simply records the presence and the offsets.

18.6.6 pci_find_extcap: PCIe Extended Capabilities

PCIe introduces a second list, called extended capabilities, that lives above offset 0x100 in configuration space. This is where modern features like Advanced Error Reporting, Virtual Channel, Access Control Services, and SR-IOV live. The list is structurally similar to the legacy capability list but uses 16-bit IDs and 4-byte offsets.

The walker is `pci_find_extcap(9)`. The signature is identical to `pci_find_cap`:

```
int capreg;

if (pci_find_extcap(dev, PCIZ_AER, &capreg) == 0) {
    device_printf(dev, "AER capability at offset 0x%x\n", capreg);
}
```

Extended capability IDs are defined in `/usr/src/sys/dev/pci/pci.h` under names that start with `PCIZ_` (as opposed to `PCIY_` for standard capabilities). The prefix is a mnemonic: `PCIY` for “PCI capabilitY” (the older list), `PCIZ` for “PCI eXtended” (Z comes after Y).

Chapter 18’s driver does not subscribe to AER or any other extended capability. It walks the extended list at attach time and logs what it finds, the same way it walks the standard list. This serves two purposes: it gives the reader a view of what PCIe capabilities look like in the wild, and it exercises `pci_find_extcap` so the reader has seen both walkers.

18.6.7 PCIe AER: An Introduction

Advanced Error Reporting (AER) is a PCIe extended capability that lets the system detect and report certain classes of PCI-level errors: uncorrectable transaction errors, correctable errors, malformed TLPs, completion timeouts, and others. The capability is optional; not every PCIe device implements it.

On FreeBSD the PCI bus driver (`pci(4)`, implemented in `/usr/src/sys/dev/pci/pci.c`) walks each device’s extended capability list during probe, locates the AER capability when present, and uses it for system-level error logging. A driver does not normally register its own AER callback; the bus handles AER centrally and logs correctable and uncorrectable errors into the kernel message buffer. A driver that wants custom handling reads the AER status registers through `pci_read_config(9)` at the offset returned by `pci_find_extcap(dev, PCIZ_AER, &offset)` and decodes them against the bit layouts in `/usr/src/sys/dev/pci/pci.h`.

For Chapter 18’s driver, AER is mentioned to complete the PCIe capability picture. The driver does not subscribe to AER events. Chapter 20 picks the topic up again, in its “PCIe AER recovery through MSI-X vectors” discussion, to explain where a driver-owned AER handler would hook into the MSI-X plumbing the interrupt chapters build. A full, end-to-end AER recovery implementation is beyond the scope of this book; readers who want to follow the bus-centric side all the way through can study `pci_add_child_clear_aer` and `pcie_apei_error` in `/usr/src/sys/dev/pci/pci.c`, together with the `PCIR_AER_*` and `PCIM_AER_*` bit layouts in `/usr/src/sys/dev/pci/pci.h`.

A short aside on the naming: “AER” is pronounced letter by letter (“ay-ee-ar”) in most FreeBSD conversations. The capability ID in the `pci.h` header is `PCIZ_AER = 0x0001`.

18.6.8 Composing the Simulation with the Real PCI Backend

The driver at Chapter 17 Stage 5 attached to the kernel as a standalone module (`kldload myfirst` triggered the attach). Chapter 18's driver attaches to a PCI device. Both attach paths need to set up the same upper-layer state (`softc`, `cdev`, some per-instance fields). The question is how to compose them.

Chapter 18's driver solves this with a single compile-time switch that selects which attach paths are active, and by having the Chapter 17 simulation's callouts **not** run when the driver is bound to a real PCI device. The logic is straightforward:

- If `MYFIRST_SIMULATION_ONLY` is defined at build time, the driver omits `DRIVER_MODULE` entirely. There is no PCI attach; the module behaves exactly like the Chapter 17 driver, and `kldload` spawns one simulated instance through the Chapter 17 module-event handler.
- If `MYFIRST_SIMULATION_ONLY` is **not** defined (the default for Chapter 18), the driver declares `DRIVER_MODULE(myfirst, pci, ...)`. The module is loadable. When a matching PCI device exists, `myfirst_pci_attach` runs. The Chapter 17 simulation callouts are not started on the PCI path; the accessor layer points at the real BAR, and the simulation backend stays idle. A reader who wants simulation re-activates it explicitly through a `sysctl` or by building with `MYFIRST_SIMULATION_ONLY`.

The compile-time guard in `myfirst_pci.c` is short:

```
#ifndef MYFIRST_SIMULATION_ONLY
DRIVER_MODULE(myfirst, pci, myfirst_pci_driver, NULL, NULL);
MODULE_DEPEND(myfirst, pci, 1, 1, 1);
#endif
```

And `myfirst_pci_attach` deliberately skips `myfirst_sim_enable(sc)`. The Chapter 17 sensor callout, command callout, and fault-injection machinery stay asleep. They are present in the code but never scheduled when the backend is a real PCI BAR; that keeps writes from the simulated `CTRL.GO` bit out of the real device's registers.

A reader on a host without a matching PCI device still has the option of running the Chapter 17 simulation directly: build with `MYFIRST_SIMULATION_ONLY=1`, `kldload`, and the driver behaves exactly as it did at the end of Chapter 17. The two builds share every file; the selection happens at compile time.

An alternative the reader may choose: split the driver into two modules. `myfirst_core.ko` holds the hardware layer, the simulation, the `cdev`, and the locks. `myfirst_pci.ko` holds the PCI attach. `myfirst_core.ko` is always loadable and provides the simulation. `myfirst_pci.ko` depends on `myfirst_core.ko` and adds PCI support on top.

This is the approach real FreeBSD drivers use when a chipset has multiple transport variants. The `uart(4)` driver has `uart.ko` as the core and `uart_bus_pci.ko` as the PCI attach; `virtio(4)` has `virtio.ko` as the core and `virtio_pci.ko` as the PCI transport. The book's later chapter on multi-transport drivers returns to this pattern.

For Chapter 18 the simpler approach (one module with a compile-time switch) is enough. A reader who wants to practise the split can try it as a challenge at the end of the chapter.

18.6.9 Why the Simulation Callouts Stay Silent on Real PCI

A note worth making explicit. When the Chapter 18 driver attaches to a real virtio-rnd device, the BAR does not hold the Chapter 17 register map. Offset 0x00 is the virtio legacy device-features register, not CTRL. Offset 0x12 is the virtio device_status register, not the Chapter 17 INTR_STATUS. Letting the Chapter 17 sensor callout write to SENSOR_CONFIG (at Chapter 17 offset 0x2c) or letting the command callout write to CTRL at 0x00 would poke arbitrary bytes into the virtio device's registers.

On a bhyve guest this is not catastrophic (the guest is disposable), but it is poor discipline. The correct behaviour is: the simulation callouts run only when the accessor layer is backed by the simulated buffer. When the accessor layer is backed by a real BAR, the simulation stays off. Chapter 18's `myfirst_pci_attach` enforces this by never calling `myfirst_sim_enable`. The cdev still works, `CSR_READ_4` still reads the real BAR, and the rest of the driver operates normally. The callouts simply do not fire.

This is a small design decision with a real consequence: the driver is safe to attach against a real PCI device without corrupting the device's state. A reader who later adapts the driver to a different device (whose register map the Chapter 17 simulation does match) can re-enable the callouts with a `sysctl` and watch them drive the real silicon. For the virtio-rnd teaching target, the callouts stay asleep.

18.6.10 Creating the cdev in a PCI Driver

The Chapter 10 driver created a cdev with `make_dev(9)` at module-load time. In the Chapter 18 PCI driver, `make_dev(9)` runs at attach time, once per PCI device. The name of the cdev incorporates the unit number: `/dev/myfirst0`, `/dev/myfirst1`, and so on.

The code is familiar:

```
sc->cdev = make_dev(&myfirst_cdevsw, device_get_unit(dev), UID_ROOT,
    GID_WHEEL, 0600, "myfirst%d", device_get_unit(dev));
if (sc->cdev == NULL) {
    error = ENXIO;
    goto fail_hw;
}
sc->cdev->si_drv1 = sc;
```

`device_get_unit(dev)` returns the unit number newbus assigned. "myfirst%d" with that unit number as the argument produces the per-instance device name. The `si_drv1` assignment lets the cdev's open, close, read, write, and `ioctl` entry points recover the_softc from the cdev.

The detach path destroys the cdev with `destroy_dev(9)`:

```
if (sc->cdev != NULL) {
    destroy_dev(sc->cdev);
    sc->cdev = NULL;
}
```

This code is entirely the Chapter 10 pattern; nothing about it is new. The point of including it here is that it fits naturally into the PCI attach ordering: softc, BAR, hardware layer, cdev, and (later in Chapter 19) interrupts. Reverse the order in detach. Done.

18.6.11 A Full Stage 3 Attach

Combining every piece of Section 5, the Stage 3 attach:

```
static int
myfirst_pci_attach(device_t dev)
{
    struct myfirst_softc *sc = device_get_softc(dev);
    int error, capreg;

    sc->dev = dev;
    sc->unit = device_get_unit(dev);
    error = myfirst_init_softc(sc);
    if (error != 0)
        return (error);

    /* Step 1: allocate BAR0. */
    sc->bar_rid = PCIR_BAR(0);
    sc->bar_res = bus_alloc_resource_any(dev, SYS_RES_MEMORY,
        &sc->bar_rid, RF_ACTIVE);
    if (sc->bar_res == NULL) {
        device_printf(dev, "cannot allocate BAR0\n");
        error = ENXIO;
        goto fail_softc;
    }

    /* Step 2: walk PCI capabilities (informational). */
    if (pci_find_cap(dev, PCIY_EXPRESS, &capreg) == 0)
        device_printf(dev, "PCIe capability at 0x%x\n", capreg);
    if (pci_find_cap(dev, PCIY_MSI, &capreg) == 0)
        device_printf(dev, "MSI capability at 0x%x\n", capreg);
    if (pci_find_cap(dev, PCIY_MSIX, &capreg) == 0)
        device_printf(dev, "MSI-X capability at 0x%x\n", capreg);
    if (pci_find_cap(dev, PCIY_PMG, &capreg) == 0)
        device_printf(dev, "Power Management capability at 0x%x\n",
            capreg);
    if (pci_find_extcap(dev, PCIZ_AER, &capreg) == 0)
        device_printf(dev, "PCIe AER extended capability at 0x%x\n",
            capreg);

    /* Step 3: attach the hardware layer against the BAR. */
}
```

```

    error = myfirst_hw_attach_pci(sc, sc->bar_res,
        rman_get_size(sc->bar_res));
    if (error != 0)
        goto fail_release;

    /* Step 4: create the cdev. */
    sc->cdev = make_dev(&myfirst_cdevsw, sc->unit, UID_ROOT,
        GID_WHEEL, 0600, "myfirst%d", sc->unit);
    if (sc->cdev == NULL) {
        error = ENXIO;
        goto fail_hw;
    }
    sc->cdev->si_drv1 = sc;

    /* Step 5: read a diagnostic word. */
    MYFIRST_LOCK(sc);
    sc->bar_first_word = CSR_READ_4(sc, 0x00);
    MYFIRST_UNLOCK(sc);
    device_printf(dev, "BAR[0x00] = 0x%08x\n", sc->bar_first_word);

    sc->pci_attached = true;
    return (0);

fail_hw:
    myfirst_hw_detach(sc);
fail_release:
    bus_release_resource(dev, SYS_RES_MEMORY, sc->bar_rid, sc->bar_res);
    sc->bar_res = NULL;
fail_softc:
    myfirst_deinit_softc(sc);
    return (error);
}

```

The structure is exactly the attach-time checklist from the start of this section, with labels (Step 1, Step 2, etc.) making the order explicit. The goto cascade handles partial failure cleanly. Each fail label undoes the step that succeeded most recently, chaining down to the one before it.

A reader who has seen this pattern before (in Chapter 15's complex attach with multiple primitives, or in any FreeBSD driver that allocates more than one resource) will recognise it immediately. A reader who is seeing it for the first time may benefit from tracing through a hypothetical failure at each step and verifying that the right amount of cleanup happens.

18.6.12 Verifying Stage 3

The expected dmesg output at Stage 3 attach:

```
myfirst0: <Red Hat Virtio entropy source (myfirst demo target)> ... on pci0
myfirst0: attaching: vendor=0x1af4 device=0x1005 revid=0x00
myfirst0:          subvendor=0x1af4 subdevice=0x0004 class=0xff
myfirst0: PCIe capability at 0x98
myfirst0: MSI-X capability at 0xa0
myfirst0: hardware layer attached to BAR: 32 bytes (tag=0x... handle=0x...)
myfirst0: BAR[0x00] = 0x10010000
```

The exact capability offsets depend on the guest's virtio implementation; the values shown are indicative. A reader who sees all four lines (attach, capabilities walked, hardware attached, BAR read) has confirmed that Stage 3 is complete.

`ls /dev/myfirst*` should show `/dev/myfirst0`. A user-space program that opens that device, writes a byte, and reads a byte should see the Chapter 17 simulation path in action (the command-response protocol still runs under the surface, even though the BAR is now real; Chapter 17 and Chapter 18 do not interact at the data-path level yet, they only share the accessor layer).

Detach verifies the reverse:

```
myfirst0: detaching
```

And `/dev/myfirst0` disappears. The BAR is released. The softc is freed. No leaks, no warnings, no stuck state.

18.6.13 Wrapping Up Section 5

Attach-time initialisation is the composition of many small steps. Each step allocates or configures one thing. The steps go in a strict order that builds the driver's state from the device outward: resources first, then features, then device-specific bring-up, then user-space interfaces, then (in Chapter 19) interrupts. The detach path undoes each step in reverse.

The PCI-specific pieces Chapter 18 adds to this pattern are `pci_enable_busmaster` (not needed for our driver, reserved for Chapters 20 and 21), the capability walkers `pci_find_cap(9)` and `pci_find_extcap(9)`, configuration-space reads and writes via `pci_read_config(9)` and `pci_write_config(9)`, and a brief introduction to PCIe AER that the reader will return to in later chapters.

Section 6 covers the detach side in depth. The broad strokes are familiar, but the details (handling partial attach failure, the order of teardown, interactions with callouts and tasks that might still be running) deserve their own section.

18.7 Section 6: Supporting Detach and Resource Cleanup

Attach brings the driver up. Detach takes it down. The two paths are mirrors, but not perfectly symmetric mirrors. Detach has a few concerns attach does not: the possibility that other code is still running (callouts, tasks, file descriptors, interrupt handlers), the need to refuse detach when the driver has work the caller has not

quiesced, and the care to avoid use-after-free between the last live access and the freeing of the softc. Section 6 is about getting those concerns right.

The goal of Section 6 is a detach routine that is strict, complete, and easy to audit. It releases every resource attach claimed. It drains every callout and task that might still be running. It destroys the cdev before freeing the hardware layer. It releases the BAR after the hardware layer no longer needs it. And it does all of this in a way that the book's readers can read and understand one step at a time.

18.7.1 The Core Rule: Reverse Order

The single most important discipline for detach is reverse order. Every step attach took, detach undoes in the inverse sequence. If attach allocated A, then B, then C, then detach releases C, then B, then A.

This rule sounds trivial. In practice, forgetting it or getting the order slightly wrong is one of the most common causes of kernel panics in new drivers. The typical symptom: a callout fires during detach, reads from a softc field, and the field is already freed. Or: the cdev still exists when the BAR is released, and a user-space process that has the cdev open triggers a read that dereferences an unmapped address.

The Chapter 15 detach pattern is the right model for Chapter 18. Attach built state outward from the device; detach tears it down inward toward the device. Nothing the detach frees can still be in use by anything else.

18.7.2 The Chapter 18 Detach Ordering

The detach order, matching Stage 3's attach:

1. Mark the driver as no longer attached (`sc->pci_attached = false`).
2. Cancel any user-space access paths: destroy the cdev so no new open or ioctl can start, accept no new requests.
3. Drain callouts and tasks that might be running (`myfirst_quiesce`).
4. Detach the simulation backend if it was attached (frees `sc->sim`). On the PCI path this is a no-op because the simulation was not attached.
5. Detach the hardware layer (frees `sc->hw`; does not release the BAR).
6. Release the BAR resource through `bus_release_resource`.
7. Tear down the softc state: destroy locks, destroy condition variables, free any allocated memory.
8. (Chapter 19's addition, mentioned for completeness) Release the IRQ resource.

For Chapter 18 the detach is:

```
static int
myfirst_pci_detach(device_t dev)
{
    struct myfirst_softc *sc = device_get_softc(dev);

    /* Refuse detach if something is still using the device. */
    if (myfirst_is_busy(sc))
        return (EBUSY);
}
```

```

sc->pci_attached = false;

/* Tear down the cdev so no new user-space accesses start. */
if (sc->cdev != NULL) {
    destroy_dev(sc->cdev);
    sc->cdev = NULL;
}

/* Drain callouts and tasks. Safe whether or not the simulation
 * was ever enabled on this instance. */
myfirst_quiesce(sc);

/* Release the simulation backend if it was attached. The PCI
 * path leaves sc->sim == NULL, so this is a no-op. */
if (sc->sim != NULL)
    myfirst_sim_detach(sc);

/* Detach the hardware layer (frees the wrapper struct). */
myfirst_hw_detach(sc);

/* Release the BAR. */
if (sc->bar_res != NULL) {
    bus_release_resource(dev, SYS_RES_MEMORY, sc->bar_rid,
        sc->bar_res);
    sc->bar_res = NULL;
}

/* Tear down the softc state. */
myfirst_deinit_softc(sc);

device_printf(dev, "detached\n");
return (0);
}

```

The code is longer than the Stage 2 detach because each step is its own concern. The structure is easy to read: each line or block releases one thing, in reverse order of attach. A reader auditing the detach can check each step against the attach and confirm the symmetry.

18.7.3 myfirst_is_busy: When to Refuse Detach

A driver that has an open cdev, an in-flight command, or any other in-progress work cannot safely detach. Returning EBUSY from detach tells the kernel's module loader to leave the driver alone.

The Chapter 10 through 15 driver had a simple busy check: are there any open file descriptors on the cdev? Chapter 17 extended it to include in-flight simulated commands. Chapter 18 reuses the same check:


```
static bool
myfirst_is_busy(struct myfirst_softc *sc)
{
    bool busy;

    MYFIRST_LOCK(sc);
    busy = (sc->open_count > 0) || sc->command_in_flight;
    MYFIRST_UNLOCK(sc);
    return (busy);
}
```

The check is under the lock because `open_count` and `command_in_flight` can be modified by other code paths (the `cdev`'s `open` and `close` entry points, the Chapter 17 command callout). Without the lock the check might see an inconsistent view, and the decision to refuse or allow detach would race against an in-progress open or close. The exact field names come from the Chapter 10 `softc` (`open_count`) and the Chapter 17 additions (`command_in_flight`); a reader whose `softc` uses different names substitutes the local names here.

Returning `EBUSY` from `detach` produces a visible error on `kldunload`:

```
# kldunload myfirst
kldunload: can't unload file: Device busy
```

The user then closes the open file descriptor, cancels the in-flight command, or does whatever else is needed to drain the busy state, and retries. This is the expected behaviour; a driver that never refuses detach is a driver that can be torn out from under its users.

18.7.4 Quiescing Callouts and Tasks

The Chapter 17 simulation runs a sensor callout every second; a command callout fires per command; a busy-recovery callout fires occasionally. The Chapter 16 hardware layer runs a ticker task through a taskqueue. On the PCI backend the simulation callouts are not enabled (as Section 5 explained), so their `callout_drain` is a safe no-op if they never ran. The hardware-layer ticker task is still active and must be drained.

The right primitive for callouts is `callout_drain(9)`. It waits until the callout is not running and prevents any future firings. The right primitive for tasks is `taskqueue_drain(9)`. It waits until the task has finished running and prevents any further enqueues.

The Chapter 17 API exposes two functions that wrap the callout lifecycle for the simulation: `myfirst_sim_disable(sc)` stops scheduling new firings (it requires `sc->mtx` held), and `myfirst_sim_detach(sc)` drains every callout and frees the simulation state (it must not hold `sc->mtx`). A single `myfirst_quiesce` helper in the PCI driver composes them safely:

```
static void
myfirst_quiesce(struct myfirst_softc *sc)
{
```

```
if (sc->sim != NULL) {
    MYFIRST_LOCK(sc);
    myfirst_sim_disable(sc);
    MYFIRST_UNLOCK(sc);
}

if (sc->tq != NULL && sc->hw != NULL)
    taskqueue_drain(sc->tq, &sc->hw->reg_ticker_task);
}
```

On the PCI path `sc->sim` is `NULL` (the simulation backend is not attached), so the first block is skipped entirely. On a simulation-only build where the simulation is attached, `myfirst_sim_disable` stops the callouts under the lock, and the subsequent `myfirst_sim_detach` (called later in the detach sequence) drains them without the lock.

The split matters because `callout_drain` must be called **without** `sc->mtx` held: the callout body itself may try to acquire the mutex, and holding it would deadlock. Chapter 13 taught this discipline; Chapter 18 respects it by routing the drain through `myfirst_sim_detach`, which acquires no locks.

After `myfirst_quiesce` returns, nothing else is running against the softc except the detach path itself. The subsequent teardown steps can touch `sc->hw` and the BAR without fear.

18.7.5 Releasing the BAR After the Hardware Layer

The order matters. `myfirst_hw_detach` is called before `bus_release_resource` because `myfirst_hw_detach` still needs the tag and handle to be valid (for example, if there were any last-chance reads during the hardware teardown; the Chapter 18 version does not do such reads, but defensive code keeps the order in case a later extension adds them).

After `myfirst_hw_detach` returns, `sc->hw` is `NULL`. The tag and handle stored in the (now-freed) `myfirst_hw` struct are gone. No code in the driver can read or write the BAR from this point onward. The BAR can then be released safely.

If the order were reversed (release BAR first, then `myfirst_hw_detach`), the hardware teardown code would hold a stale tag and handle; any access would be a use-after-free. On x86 the bug might be silent; on architectures with stricter memory permissions the access would page-fault.

18.7.6 Failures During Detach

Unlike attach, detach is generally expected to succeed. The kernel's unload path calls detach; if detach returns non-zero, the unload aborts, but detach itself should not leave resources in an inconsistent state. The convention is that detach returns 0 (success) or `EBUSY` (refuse the unload because the driver is in use). Returning any other error is unusual and typically indicates a driver bug.

If a resource release fails (for example, `bus_release_resource` returns an error), the driver should log the failure but continue the detach. Leaving a partially-freed state is worse than logging and continuing; the kernel will complain about the leaked resource at shutdown, but the driver will not have crashed. Chapter 18's driver

does not check the return value of `bus_release_resource` for this reason; the release either succeeds or leaves an unrecoverable kernel state, neither of which the driver can do anything about.

18.7.7 Detach vs Module Unload vs Device Removal

Three different events can trigger detach.

Module unload (`kldunload myfirst`): the user asks to remove the module. The kernel's unload path calls detach on every device bound to the module, one at a time. If every detach returns 0, the module is unloaded. If any detach returns non-zero, the module stays loaded and the unload returns an error.

Device removal by user (`devctl detach myfirst0`): the user asks to detach a specific device from its driver, without unloading the module. The driver's detach runs for that one device; the module stays loaded and can still attach to other devices.

Device removal by hardware (hotplug, such as removing a PCIe card from a hot-plug-capable slot, or the hypervisor removing a virtual device): the PCI bus detects the change and calls detach on the device. The driver's detach runs. If the device is later re-inserted, the driver's probe and attach run again.

All three paths run the same `myfirst_pci_detach` function. The driver does not need to distinguish them. The code is the same because the obligations are the same: release everything attach allocated.

18.7.8 Partial-Attach Failure and the Detach Path

A subtle case worth explaining. If attach fails partway through and returns an error, the kernel does not (in modern newbus) call detach on the partially-attached driver. The driver's own goto cascade handles the cleanup.

The attach code from Section 5 has a goto cascade that undoes exactly the steps that succeeded. If the cdev creation fails after the hardware layer attach, the cascade releases the hardware layer and the BAR before returning. If the hardware layer attach fails after the BAR allocation, the cascade releases the BAR before returning. Each fail label undoes one step.

A common beginner mistake is to write a goto cascade that skips steps. For example:

```
fail_hw:
    bus_release_resource(dev, SYS_RES_MEMORY, sc->bar_rid, sc->bar_res);
fail_softc:
    myfirst_deinit_softc(sc);
    return (error);
```

This skips the `myfirst_hw_detach` step. If the hardware layer attach succeeded but the cdev creation failed, the cascade frees the BAR without tearing down the hardware layer, leaking the hardware layer's wrapper struct. The right cascade calls every unwind step the successful attach would have needed to unwind.

A technique some drivers use: arrange the attach as a sequence of `myfirst_init_*` helpers and the detach as a matching sequence of `myfirst_uninit_*` helpers, and have a single `myfirst_fail` function that walks the unwind list based on how far attach got. This is cleaner for very complex drivers; for Chapter 18's driver the goto cascade is simpler and more readable.

18.7.9 A Concrete Walk Through the Cascade

Let's trace what happens if the cdev creation fails at Stage 3. The attach has:

1. Initialised the softc (success).
2. Allocated BAR0 (success).
3. Walked capabilities (always succeeds; it's just reads).
4. Attached the hardware layer (success).
5. Tried to create the cdev: fails because of some error (disk full? unlikely; in tests the reader can simulate this by returning NULL from a mocked `make_dev`).

The cascade runs `fail_hw`, which calls `myfirst_hw_detach` (undoes step 4), then `fail_release`, which releases the BAR (undoes step 2), then `fail_softc`, which deinit the softc (undoes step 1). Step 3's "undo" is the empty action (capability walks allocate nothing). The attach returns the error.

If the reader traces this by hand, the cleanup is clearly complete: the softc is deinit, the BAR is released, the hardware layer is detached. No leaks. No partial state. The test is the same as the test for a full attach followed by a full detach: `vmstat -m | grep myfirst` should show zero allocations after the failed attach returns.

18.7.10 Detach vs Resume: A Preview

For completeness: the suspend-and-resume paths, which Chapter 18 does not implement, look similar to detach-and-attach but preserve more state. Suspend quiesces the driver (drains callouts, stops user-space access), records the device's state in the softc, and lets the system power down. Resume re-initialises the device from the saved state, restarts callouts, re-enables user-space access, and returns.

A driver that implements only attach and detach cannot suspend cleanly; the kernel will refuse to suspend a system with a non-suspend-aware driver attached. The `myfirst` driver is small enough that Chapter 18 does not worry about this; Chapter 22 on power management revisits the topic.

18.7.11 Wrapping Up Section 6

Detach is attach, played in reverse, with a check for `EBUSY` at the top and a quiesce step before any teardown. The rule is simple; the discipline is in doing it consistently. Every resource attach allocates, detach releases. Every state attach sets up, detach tears down. Every callout attach starts, detach drains. The order is the reverse of attach.

For Chapter 18's driver, the detach at Stage 3 is six steps: refuse if busy, destroy the cdev, quiesce the simulation and hardware callouts, detach the hardware layer, release the BAR, deinit the softc. Each step is its own concern. Each step is auditable against the attach. Each step is testable in isolation.

Section 7 is the testing section. It stands up the bhyve or QEMU lab, walks through the attach-detach cycle on real PCI silicon-that-is-emulated, and teaches the reader to verify the driver's behaviour with `pciconf`, `devinfo`, `dmesg`, and a couple of small user-space programs.

18.8 Section 7: Testing PCI Driver Behaviour

A driver exists to talk to a device. Chapter 18's driver is written and compiled, but it has not yet run against any device. Section 7 closes the loop. It walks the reader through standing up a FreeBSD guest in bhyve or QEMU, exposing a virtio-rnd PCI device to the guest, loading the driver, observing the full attach-operate-detach-unload cycle, exercising the cdev from user space, reading and writing configuration space with `pciconf -r` and `pciconf -w`, and confirming through `devinfo -v` and `dmesg` that the driver's view of the world matches the kernel's.

Testing is where all of Chapters 2 through 17 finally pay off. Every habit the book has built (typing code by hand, reading the FreeBSD source, running regressions after every stage, keeping a lab logbook) serves Chapter 18's testing discipline. The section is long because real PCI testing has real moving parts; each part is worth walking through carefully.

18.8.1 The Test Environment

The canonical test environment for Chapter 18 is a FreeBSD 14.3 guest running under bhyve(8) on a FreeBSD 14.3 host. The guest receives an emulated virtio-rnd device through bhyve's `virtio-rnd` passthrough. The guest runs the reader's debug kernel. The `myfirst` driver is compiled and loaded inside the guest; it attaches to the virtio-rnd device, and the reader exercises it from inside the guest.

An equivalent environment uses `qemu-system-x86_64` on Linux or macOS as the host, with a FreeBSD 14.3 guest running a debug kernel. QEMU's `-device virtio-rng-pci` does the same job as bhyve's `virtio-rnd`. Everything else is identical.

The rest of this section assumes bhyve unless noted otherwise. Readers on QEMU substitute the equivalent commands; the concepts transfer directly.

18.8.2 Preparing the bhyve Guest

The author's lab script looks roughly like this, edited for clarity:

```
#!/bin/sh
set -eu

# Load bhyve's kernel modules.
kldload -n vmm nmdm if_bridge if_tap

# Prepare a network bridge.
# ifconfig bridge0 create 2>/dev/null || true
# ifconfig bridge0 addm em0 addm tap0
# ifconfig tap0 up
# ifconfig bridge0 up

# Launch the guest.
bhyve -c 2 -m 2048 -H -w \
```

```
-s 0:0,hostbridge \
-s 1:0,lpc \
-s 2:0,virtio-net,tap0 \
-s 3:0,virtio-blk,/dev/zvol/zroot/vm/freebsd143/disk0 \
-s 4:0,virtio-rnd \
-l com1,/dev/nmdm0A \
-l bootrom,/usr/local/share/uefi-firmware/BHYVE_UEFI.fd \
vm:fbbsd-14.3-lab
```

The key line is `-s 4:0,virtio-rnd`. It attaches a `virtio-rnd` device on PCI slot 4, function 0. The guest's PCI enumerator will see a device at `pci0:0:4:0` with vendor `0x1af4` and device `0x1005`, which is exactly the ID pair the Chapter 18 driver's probe table matches.

Other slots carry the hostbridge, LPC, network (tap-bridged), and storage (zvol-backed block). The overall guest has everything it needs to boot and run multiuser, plus one PCI device for our driver.

A shorter form for readers who prefer `vm(8)` (the FreeBSD utility from the `vm-bhyve` port):

```
vm create -t freebsd-14.3 fbbsd-lab
vm configure fbbsd-lab # edit vm.conf and add:
#   passthru0="0/0/0"      # if using passthrough, not needed here
#   virtio_rnd="1"         # add a virtio-rnd device
vm start fbbsd-lab
```

`vm-bhyve` hides the detail of the `bhyve` command line. Either form produces an equivalent lab environment.

Readers on QEMU use:

```
qemu-system-x86_64 -cpu host -m 2048 -smp 2 \
  -drive file=freebsd-14.3-lab.img,if=virtio \
  -netdev tap,id=net0,ifname=tap0 -device virtio-net,netdev=net0 \
  -device virtio-rng-pci \
  -bios /usr/share/qemu/OVMF_CODE.fd \
  -serial stdio
```

The `-device virtio-rng-pci` line does the `bhyve`-equivalent job.

18.8.3 Verifying the Guest Sees the Device

Inside the guest, after the first boot, the `virtio-rnd` device should be visible:

```
pciconf -lv
```

Look for an entry like:

```
virtio_random0@pci0:0:4:0: class=0x00ff00 rev=0x00 hdr=0x00 vendor=0x1af4
↳ device=0x1005 subvendor=0x1af4 subdevice=0x0004
    vendor      = 'Red Hat, Inc.'
    device      = 'Virtio entropy'
    class       = old
```

The entry tells you three things. First, the guest's PCI enumerator found the device. Second, the base-system `virtio_random(4)` driver has claimed it (the leading name `virtio_random0` is the clue). Third, the B:D:F is 0:0:4:0, matching the `bhyve -s 4:0,virtio-rnd` configuration.

If the entry is missing, either the `bhyve` command line did not include `virtio-rnd` or the guest booted without loading `virtio_pci.ko`. Both are fixable: review the `bhyve` command, reboot the guest, or `kldload virtio_pci` by hand.

18.8.4 Preparing the Guest for myfirst

On a stock FreeBSD 14.3 GENERIC kernel, `virtio_random` is not compiled in; it ships as a loadable module (`virtio_random.ko`). Whether it has claimed the device by the time you want to load `myfirst` depends on the platform. On a modern system, `devmatch(8)` may auto-load `virtio_random.ko` shortly after boot when it sees a matching PCI device. On a freshly booted guest where `devmatch` has not fired yet, the `virtio-rnd` device may still be unclaimed.

Check first:

```
kldstat | grep virtio_random
pciconf -lv | grep -B 1 virtio_random
```

If neither command shows `virtio_random`, the device is unclaimed and you can skip the next step.

If `virtio_random` has claimed the device, unload it:

```
sudo kldunload virtio_random
```

If the module is unloadable (not pinned, not in use), this succeeds and the `virtio-rnd` device becomes unclaimed. `devinfo -v` now shows it under `pci0` with no driver binding.

If you want a stable setup that never auto-loads `virtio_random` across reboots, add to `/boot/loader.conf`:

```
hint.virtio_random.0.disabled="1"
```

This prevents the binding at boot time without removing the module image from the system. Alternatively, adding an entry under `/etc/devd.conf` or `devmatch.blocklist` (or using the `dev.virtio_random.0.%driver sysctl` at runtime) blocks the driver from attaching. For Chapter 18's teaching path, a simple `kldunload` once per test session is enough.

Chapter 18's testing uses the first approach during development (quick iteration) and the second approach when the reader wants a stable setup for repeated tests.

A third approach worth mentioning: the kernel's `dev.NAME.UNIT.%parent` and `dev.NAME.UNIT.%driver` sysctls describe bindings but do not change them. To force a rebind, use `devctl detach` and `devctl set driver`:

```
sudo devctl detach virtio_random0
sudo devctl set driver -f pci0:0:4:0 myfirst
```

The `-f` flag forces the set even if another driver has claimed the device. This is the precise command to use in a scripted test where the reader wants to switch drivers without reloading modules.

18.8.5 Loading myfirst and Watching the Attach

With `virtio_random` out of the way, load `myfirst`:

```
sudo kldload ./myfirst.ko
```

Watch `dmesg` for the attach:

```
sudo dmesg | tail -20
```

The expected output (for Stage 3):

```
myfirst0: <Red Hat Virtio entropy source (myfirst demo target)> mem
↳ 0xc1000000-0xc100001f at device 4.0 on pci0
myfirst0: attaching: vendor=0x1af4 device=0x1005 revid=0x00
myfirst0:          subvendor=0x1af4 subdevice=0x0004 class=0xff
myfirst0: PCIe capability at 0x0
myfirst0: MSI-X capability at 0x0
myfirst0: hardware layer attached to BAR: 32 bytes (tag=... handle=...)
myfirst0: BAR[0x00] = 0x10010000
```

(The capability offsets for the virtio legacy device are 0 because the bhyve emulation does not expose a PCIe capability; a reader who tests with QEMU's `virtio-rng-pci` may see non-zero offsets.)

The `cdev /dev/myfirst0` exists:

```
ls -l /dev/myfirst*
```

and `devinfo -v` shows the device:


```
devinfo -v | grep -B 1 -A 4 myfirst
```

```
pci0
  myfirst0
    pnpinfo vendor=0x1af4 device=0x1005 ...
    resources:
      memory: 0xc1000000-0xc100001f
```

This is the driver attached to the device, visible to user space, ready to be exercised.

18.8.6 Exercising the cdev

The myfirst driver's cdev path is the Chapter 10 through 17 interface. It accepts open, close, read, and write system calls. In the Chapter 17 simulation-only build, reads pulled from the command-response ring buffer that the simulation's callouts populated. In the Chapter 18 PCI build the simulation is not attached; the cdev still responds to open, read, write, and close, but the data path has no active callouts feeding it. Reads return what the underlying Chapter 10 circular buffer contains (typically empty at start); writes queue data into the same buffer.

This is expected behaviour for Chapter 18. The point of the chapter's tests is to prove that the driver attaches to a real PCI device, the BAR is live, the cdev is reachable from user space, and detach cleans up properly. Chapter 19 adds the interrupt path that will make the cdev's data path meaningful against a real device.

A small user-space program to exercise the cdev:

```
/* Minimal read-write test. */
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>

int main(void) {
    int fd = open("/dev/myfirst0", O_RDWR);
    if (fd < 0) { perror("open"); return 1; }

    char buf[16];
    ssize_t n = read(fd, buf, sizeof(buf));
    printf("read returned %zd\n", n);

    close(fd);
    return 0;
}
```

Compile and run:

```
cc -o myfirst_test myfirst_test.c
./myfirst_test
```

The output may be a short read, a zero read, or EAGAIN (depending on whether the Chapter 10 buffer has any data ready). What matters is that the read path does not crash the kernel, does not produce an error in dmesg, and returns to user space with a defined result.

A write test is equally straightforward:

```
char cmd[16] = "hello\n";
write(fd, cmd, 6);
```

Writes push data into the Chapter 10 circular buffer. On the PCI backend, the data sits in the buffer until a reader pulls it out; no simulation callout is running to process it. The test is that the cycle runs without crashing and that dmesg stays quiet.

18.8.7 Reading and Writing Configuration Space from User Space

pciconf(8) has two flags that let user-space programs inspect and modify PCI configuration space directly:

- `pciconf -r <selector> <offset>:<length>` reads configuration-space bytes and prints them in hex.
- `pciconf -w <selector> <offset> <value>` writes a value to a specific offset.

The selector identifies the device. It can be the device's driver name (`myfirst0`) or its B:D:F (`pci0:0:4:0`).

A read example:

```
sudo pciconf -r myfirst0 0x00:8
```

Output:

```
00: 1a f4 05 10 07 05 10 00
```

The bytes are the first eight bytes of configuration space, in order: vendor ID (1af4, little-endian), device ID (1005), command register (0507, with MEMEN and BUSMASTER set), status register (0010).

A write example (dangerous, do not do this casually):

```
sudo pciconf -w myfirst0 0x04 0x0503
```

This clears BUSMASTER and leaves MEMEN set. The effect on the device depends on the device; on a running device it may cause DMA operations to fail. For a device the driver does not use DMA against (Chapter 18's case), the change is essentially harmless but also essentially meaningless.

A reader should use `pciconf -w` only in deliberate diagnostic scenarios, with full knowledge of the consequences. Writing a garbage value to the wrong field can deadlock the device, the bus, or the kernel.

18.8.8 devinfo -v and What It Tells You

`devinfo -v` is the newbus tree inspector. It walks every device in the system and prints each one with its resources, its parent, its unit, and its children. For a driver author, it is the canonical reference for “what does the kernel think the driver owns”.

A fragment of output, for the `myfirst0` device:

```
nexus0
  acpi0
    pcib0
      pci0
        myfirst0
          pnpinfo vendor=0x1af4 device=0x1005 subvendor=0x1af4 subdevice=0x0004
↪ class=0x00ff00
    resources:
      memory: 0xc1000000-0xc100001f
    virtio_pci0
      pnpinfo vendor=0x1af4 device=0x1000 ...
      resources: ...
    ... (other pci children)
```

The tree shows the path from the root (`nexus`) down through the platform (ACPI on x86), the PCI bridge (`pcib`), the PCI bus (`pci0`), and finally the devices on that bus. `myfirst0` is a child of `pci0`. Its resource list shows the claimed memory BAR.

Using `devinfo -v | grep -B 1 -A 5 myfirst` to extract just the relevant block is the standard technique when the tree is large.

18.8.9 dmesg as a Diagnostic Tool

`dmesg` is the kernel’s message buffer. Every `device_printf`, `printf`, and `KASSERT` failure in the kernel appears in `dmesg`. For a driver author, it is the primary debugging surface.

Tailing `dmesg` while loading, operating, and unloading the driver is how you catch subtle problems early. A typical session:

```
# Start a dmesg tail in a second terminal.
dmesg -w
```

Then, in the primary terminal:

```
sudo kldload ./myfirst.ko
```

The tailing terminal shows the attach messages as they happen. Run your tests:

```
./myfirst_test
```

The tailing terminal shows any messages the driver emits during the test. Unload:

```
sudo kldunload myfirst
```

The tailing terminal shows the detach messages.

If any step produces an unexpected warning or error, you see it in real time. Without a tailing `dmesg`, you might miss a single warning that indicates a latent problem.

18.8.10 Using devctl to Simulate Hotplug

`devctl(8)` lets a user-space program simulate the newbus events that a real hotplug or device-removal would generate. The common invocations:

```
# Force a device to detach (calls the driver's detach method).
```

```
sudo devctl detach myfirst0
```

```
# Re-attach the device (calls the driver's probe and attach).
```

```
sudo devctl attach myfirst0
```

```
# Disable a device (prevent future probes from binding).
```

```
sudo devctl disable myfirst0
```

```
# Re-enable a disabled device.
```

```
sudo devctl enable myfirst0
```

```
# Rescan a bus (equivalent to a hotplug notification).
```

```
sudo devctl rescan pci0
```

For testing Chapter 18's detach path, `devctl detach myfirst0` is the primary tool. It exercises the detach code without unloading the module. The driver's detach runs; the `cdev` disappears; the BAR is released; the device goes back to unclaimed.

A subsequent `devctl attach` re-triggers probe and attach. If the probe succeeds (the vendor and device IDs still match) and attach succeeds, the device is bound again. This is the cycle a reader uses to test that the driver can attach, detach, and reattach without leaking resources.

Running this cycle in a loop is the standard regression pattern:

```
for i in 1 2 3 4 5; do
    sudo devctl detach myfirst0
    sudo devctl attach myfirst0
done
sudo vmstat -m | grep myfirst
```

If `vmstat -m` shows the `myfirst` malloc type with zero current allocations after the loop, the driver is clean: every attach allocated, every detach freed, and the totals balance.

18.8.11 A Simple Regression Script

Putting it all together, a script that verifies Stage 3's PCI path:

```
#!/bin/sh
#
# Chapter 18 Stage 3 regression test.
# Run inside a bhyve guest that exposes a virtio-rnd device.

set -eu

echo "=== Unloading virtio_random if present ==="
kldstat | grep -q virtio_random && kldunload virtio_random || true

echo "=== Loading myfirst ==="
kldload ./myfirst.ko
sleep 1

echo "=== Checking attach ==="
devinfo -v | grep -q 'myfirst0' || { echo FAIL: no attach; exit 1; }

echo "=== Checking BAR claim ==="
devinfo -v | grep -A 3 'myfirst0' | grep -q 'memory:' || \
    { echo FAIL: no BAR; exit 1; }

echo "=== Exercising cdev ==="
./myfirst_test
sleep 1

echo "=== Detach-attach cycle ==="
for i in 1 2 3; do
    devctl detach myfirst0
    sleep 0.5
    devctl attach pci0:0:4:0
    sleep 0.5
done

echo "=== Unloading myfirst ==="
kldunload myfirst
sleep 1

echo "=== Checking for leaks ==="
```

```
vmstat -m | grep -q myfirst && echo WARN: myfirst malloc type still present

echo "=== Success ==="
```

The script follows a repeatable pattern: set up, load, check attach, check resources, exercise, cycle, unload, check for leaks. Running this after every change to the driver is how you catch regressions early.

18.8.12 Reading Configuration Space for Diagnostic Purposes

A small worked example of using `pciconf -r` to verify that the driver's view of configuration space matches user space's view.

Inside the driver, the attach path reads the vendor ID via `pci_get_vendor` and the device ID via `pci_get_device`. User space reads the same bytes via `pciconf -r myfirst0 0x00:4`.

Expected output:

```
00: f4 1a 05 10
```

The bytes are the vendor ID (0x1af4) and the device ID (0x1005), in little-endian order. Reversing the bytes gives 0x1af4 for vendor and 0x1005 for device, matching the driver's probe table.

Doing this check is not something you would do in production; the PCI subsystem is well-tested and the values are reliable. It is useful as a learning exercise: it proves that the driver's view of configuration space matches what user space sees, and it cements the reader's understanding of how `pci_get_vendor` relates to the underlying bytes.

18.8.13 What Section 7 Does Not Test

Section 7 verifies that the Chapter 18 driver attaches to a real PCI device, claims the BAR, exposes the `cdev`, reads the BAR through the Chapter 16 accessor layer, detaches cleanly, releases the BAR, and unloads without leaks. It does not test:

- Interrupt handling. The driver does not register an interrupt; Chapter 19 does.
- MSI or MSI-X. Chapter 20 does.
- DMA. Chapter 21 does.
- Device-specific protocol. The Chapter 17 simulation's command-response protocol is not the `virtio-rnd` protocol, so the results of a write are not meaningful. Chapter 18's driver is not a `virtio-rnd` driver.

A reader who wants a driver that actually implements the `virtio-rnd` protocol should read `/usr/src/sys/dev/virtio/random/virtio_random.c`. It is a clean, focused driver that a reader who has finished Chapter 18 should be able to follow.

18.8.14 Wrapping Up Section 7

Testing a PCI driver means standing up an environment where the driver can meet a device. For Chapter 18 that environment is a bhyve or QEMU guest with a virtio-rnd device exposed to the guest. The tools are `pciconf -lv` (to see the device), `kldload` and `kldunload` (to load and unload the driver), `devinfo -v` (to see the newbus tree and the driver's resources), `devctl` (to simulate hotplug), `dmesg` (to watch diagnostic messages), and `vmstat -m` (to check for leaks). The discipline is to run a repeatable script after every change, inspect its output, and fix any warnings or failures before moving on.

The regression script at the end of this section is the template every reader should adapt to their own driver and their own lab. Running it ten times in a row and seeing identical output each time is the proof that the driver is solid. Running it once and seeing a crash is a sign the driver has a bug that Chapter 18's discipline (attach order, detach order, resource pairing) should have prevented; the fix is usually small.

Section 8 is the final section of the instructional body. It refactors the Chapter 18 code into its final shape, bumps the version to 1.1-pci, writes the new `PCI.md`, and prepares the ground for Chapter 19.

18.9 Section 8: Refactoring and Versioning Your PCI Driver

The PCI driver is now working. Section 8 is the housekeeping section. It consolidates the Chapter 18 code into a clean, maintainable structure, updates the driver's `Makefile` and module metadata, writes the `PCI.md` document that will live alongside `LOCKING.md`, `HARDWARE.md`, and `SIMULATION.md`, bumps the version to 1.1-pci, and runs the full regression pass against both the simulation and the real PCI backend.

A reader who has made it this far may be tempted to skip Section 8. It is boring compared to the earlier sections. It does not introduce any new PCI concepts. The temptation is real, and it is a mistake. The refactor is what turns a working driver into a maintainable driver. A driver that works today but is organised badly will be painful to extend in Chapter 19 (when interrupts arrive), Chapter 20 (when MSI and MSI-X arrive), Chapters 20 and 21 (when DMA arrives), and every later chapter. The few lines of housekeeping Section 8 does pay dividends across the rest of Part 4 and beyond.

18.9.1 The Final File Layout

At the end of Chapter 18, the `myfirst` driver consists of these files:

<code>myfirst.c</code>	- Main driver: <code>softc</code> , <code>cdev</code> , module events, data path.
<code>myfirst.h</code>	- Shared declarations: <code>softc</code> , lock macros, prototypes.
<code>myfirst_hw.c</code>	- Chapter 16 hardware access layer: <code>CSR_*</code> accessors, access log, <code>sysctl</code> handlers.
<code>myfirst_hw.h</code>	- Chapter 16 register map and accessor declarations, extended in Chapter 17.
<code>myfirst_sim.c</code>	- Chapter 17 simulation backend: callouts, fault injection, command-response.
<code>myfirst_sim.h</code>	- Chapter 17 simulation interface.
<code>myfirst_pci.c</code>	- Chapter 18 PCI attach: probe, attach, detach, <code>DRIVER_MODULE</code> , <code>MODULE_DEPEND</code> .

```

myfirst_pci.h    - Chapter 18 PCI declarations: ID table entry struct,
                  vendor and device ID constants.
myfirst_sync.h  - Part 3 synchronisation primitives.
cbuf.c / cbuf.h - The Chapter 10 circular buffer, still in use.
Makefile        - kmod build: KMOD, SRCS, CFLAGS.
HARDWARE.md     - Chapter 16/17 documentation of the register map.
LOCKING.md      - Chapter 15 onward documentation of lock discipline.
SIMULATION.md   - Chapter 17 documentation of the simulation backend.
PCI.md          - Chapter 18 documentation of PCI support.

```

The split is the same split Chapter 17 anticipated. `myfirst_pci.c` and `myfirst_pci.h` are new. Every other file existed before Chapter 18 and has been either extended (`myfirst_hw.c` gained `myfirst_hw_attach_pci`) or left unchanged. The driver's main file (`myfirst.c`) grew by a few lines to add the PCI-related softc fields and a call to the PCI-specific detach helper; it did not grow substantially.

A rule of thumb worth stating: each file should have one responsibility. `myfirst.c` is the driver's integration point; it ties every piece together. `myfirst_hw.c` is about hardware access. `myfirst_sim.c` is about simulating the hardware. `myfirst_pci.c` is about attaching to real PCI hardware. When a reader opens a file, they should be able to predict, from the file name, what is in it. When Chapter 19 adds `myfirst_intr.c`, the prediction will hold: that file is about interrupts.

18.9.2 The Final Makefile

```

# Makefile for the Chapter 18 myfirst driver.
#
# Combines the Chapter 10-15 driver, the Chapter 16 hardware layer,
# the Chapter 17 simulation backend, and the Chapter 18 PCI attach.
# The driver is loadable as a standalone kernel module via
# kldload(8); when loaded, it attaches automatically to any PCI
# device whose vendor/device ID matches an entry in
# myfirst_pci_ids[] (see myfirst_pci.c).

KMOD=  myfirst
SRCS=  myfirst.c myfirst_hw.c myfirst_sim.c myfirst_pci.c cbuf.c

# Version string. Update this line alongside any user-visible change.
CFLAGS+= -DMYFIRST_VERSION_STRING=\"1.1-pci\"

# Optional: build without PCI support (simulation only).
# CFLAGS+= -DMYFIRST_SIMULATION_ONLY

# Optional: build without simulation fallback (PCI only).
# CFLAGS+= -DMYFIRST_PCI_ONLY

```



```
.include <bsd.kmod.mk>
```

Four SRCS, one version string, two commented-out compile options. The build is one command:

```
make
```

The output is `myfirst.ko`, loadable into any FreeBSD 14.3 kernel with `kldload`.

18.9.3 The Version String

The version string moves from `1.0-simulated` to `1.1-pci`. The bump reflects that the driver has acquired a new capability (real PCI support) without changing any user-visible behaviour (the `cdev` still does what it did). A minor-version bump is appropriate; a major-version bump would imply an incompatible change.

Later chapters will continue the numbering: `1.2-intr` after Chapter 19, `1.3-msi` after Chapter 20, `1.4-dma` after Chapters 20 and 21, and so on. By the end of Part 4 the driver will be at `1.4-dma` or so, with each minor version reflecting one significant capability addition.

The version string is visible in two places: `kldstat -v` shows it, and the driver's `dmesg` banner on load prints it. A user or system administrator who wants to know which version of the driver they have running can `grep dmesg` for the banner.

18.9.4 The PCI.md Document

A new document joins the driver's corpus. `PCI.md` is short; its job is to describe the PCI support the driver provides, in a form a future reader can consult without reading the source.

```
# PCI Support in the myfirst Driver
```

```
## Supported Devices
```

As of version `1.1-pci`, `myfirst` attaches to PCI devices matching the following vendor/device ID pairs:

Vendor	Device	Description
-----	-----	-----
0x1af4	0x1005	Red Hat/virtio-rnd (demo target; see README)

This list is maintained in ``myfirst_pci.c`` in the static array ``myfirst_pci_ids[]``. Adding a new supported device requires:

1. Adding an entry to ``myfirst_pci_ids[]`` with the vendor and device IDs and a human-readable description.

2. Verifying that the driver's BAR layout and register map are compatible with the new device.
3. Testing the driver against the new device.
4. Updating this document.

Attach Behaviour

The driver's probe routine returns `BUS_PROBE_DEFAULT` on a match and `ENXIO` otherwise. Attach allocates BAR0 as a memory resource, walks the PCI capability list (Power Management, MSI, MSI-X, PCIe, PCIe AER if present), attaches the Chapter 16 hardware layer against the BAR, and creates `/dev/myfirstN`. The Chapter 17 simulation backend is NOT attached on the PCI path; the driver's accessors read and write the real BAR without the simulation callouts running.

Detach Behaviour

Detach refuses to proceed if the driver has open file descriptors or in-flight commands (returns `EBUSY`). Otherwise it destroys the cdev, drains any active callouts and tasks, detaches the hardware layer, releases the BAR, and deinit the softc.

Module Dependencies

The driver's `MODULE_DEPEND` declarations:

- `pci`, version 1: the kernel's PCI subsystem.

No other module dependencies are declared.

Known Limitations

- The driver does not currently handle interrupts. See Chapter 19 for the interrupt-handling extension.
- The driver does not currently support DMA. See Chapters 20 and 21 for the DMA extension.
- The Chapter 17 simulation backend is not attached on the PCI path. The simulation's callouts and command protocol remain available in a simulation-only build (`-DMYFIRST_SIMULATION_ONLY`) for readers without matching PCI hardware.

See Also

- ``HARDWARE.md`` for the register map.
- ``SIMULATION.md`` for the simulation backend.
- ``LOCKING.md`` for the lock discipline.
- ``README.md`` for how to set up the bhyve test environment.

This document lives next to the driver source. A future reader (the author themselves, three months from now, or a contributor, or a port maintainer) can read it in five minutes and understand the driver's PCI story without opening the code.

18.9.5 Updating LOCKING.md

LOCKING.md already documents the Chapter 11 through 17 lock discipline. Chapter 18 adds two small items:

1. The detach order: new steps for `destroy_dev`, quiescing callouts, `myfirst_hw_detach`, `bus_release_resource`, and `myfirst_deinit_softc`, in that order.
2. The attach failure cascade: the goto labels (`fail_hw`, `fail_release`, `fail_softc`) and what each one undoes.

The update is a handful of lines in the existing document. No new lock is introduced in Chapter 18; the Chapter 15 lock hierarchy is unchanged.

18.9.6 Updating HARDWARE.md

HARDWARE.md already documents the Chapter 16 and Chapter 17 register map. Chapter 18 adds one small item:

- The BAR the driver attaches to is BAR 0, requested with `rid = PCIR_BAR(0)`, allocated as `SYS_RES_MEMORY` with `RF_ACTIVE`. The tag and handle are extracted with `rman_get_bustag(9)` and `rman_get_bushandle(9)`.

That is the entire addition. The register map itself does not change in Chapter 18; the same offsets, the same widths, the same bit definitions.

18.9.7 The Regression Pass

With the refactor complete, the full regression pass for Chapter 18 is:

1. **Compile cleanly.** `make` produces `myfirst.ko` without warnings. The CFLAGS already include `-Wall -Werror` from Chapter 4 onward; the build fails if any warning appears.
2. **Load without errors.** `kldload ./myfirst.ko` succeeds and `dmesg` shows the module-level banner.
3. **Attach to a real PCI device.** In a bhyve guest with a virtio-rnd device, the driver attaches and `dmesg` shows the full Chapter 18 attach sequence.
4. **Create and exercise the cdev.** `/dev/myfirst0` exists, `open / read / write / close` work, no kernel messages indicate errors.
5. **Walk capabilities.** `dmesg` shows the capability offsets for any capabilities the guest's virtio-rnd exposes.

6. **Read configuration space from user space.** `pciconf -r myfirst0 0x00:8` produces the expected bytes.
7. **Detach cleanly.** `devctl detach myfirst0` produces the detach banner in `dmesg`; the `cdev` disappears; `vmstat -m | grep myfirst` shows zero live allocations.
8. **Reattach cleanly.** `devctl attach pci0:0:4:0` re-triggers probe and attach; the full cycle runs again.
9. **Unload cleanly.** `kldunload myfirst` succeeds; `kldstat -v | grep myfirst` returns nothing.
10. **No leaks.** `vmstat -m | grep myfirst` returns nothing.

The regression script from Section 7 runs steps 1 through 10 in sequence and reports success or the first failure. Running it after every change is the discipline that catches regressions early.

18.9.8 What the Refactor Accomplished

At the start of Chapter 18 the `myfirst` driver was a simulation. It had a `malloc(9)`-backed register block, a simulation backend, and an elaborate testing harness. It did not attach to real hardware; it was a module loaded by hand.

At the end of Chapter 18 the driver is a PCI driver. It attaches to a real PCI device when one is present. It claims the device's BAR through the standard FreeBSD bus-allocation API. It uses the Chapter 16 accessor layer to read and write the device's registers through `bus_space(9)`. The Chapter 17 simulation remains available through a compile-time switch (`-DMYFIRST_SIMULATION_ONLY`) for readers without matching PCI hardware, but the default build targets the PCI path and leaves the simulation callouts idle. The attach and detach paths follow the `newbus` conventions every other FreeBSD driver uses.

The code is recognisably FreeBSD. The layout is the layout real drivers use when they have distinct simulation, hardware, and bus responsibilities. The vocabulary is the vocabulary real drivers share. A contributor opening the driver for the first time finds a familiar structure, reads the documentation, and can navigate the code by subsystem.

18.9.9 A Short Note on Symbol Visibility

A reader who diffs the Chapter 17 driver against the Chapter 18 driver will notice that several functions have changed visibility. Some that were `static` in Chapter 17 are now exported (non-`static`) because `myfirst_pci.c` needs them. Examples include `myfirst_init_softc`, `myfirst_deinit_softc`, and `myfirst_quiesce`.

The convention is: a function that is called only from within its own file is `static`. A function that is called across files (but only within this driver) is non-`static` with a declaration in `myfirst.h` or another project-local header. A function that is callable from other modules (rare, and typically only through a KPI) is explicitly exported via a kernel-style symbol table; this is not relevant for Chapter 18.

The refactor does not export any new symbol outside the driver; it only promotes a few functions from file-local to driver-local. A reader who is bothered by the promotion has two options: leave the functions in `myfirst.c` and call them through a small helper that `myfirst_pci.c` calls (one more layer of indirection), or accept the promotion and document it in the source comments. The book chooses the latter; the driver is small enough that the occasional driver-local export is easy to audit.

18.9.10 Wrapping Up Section 8

The refactor is, again, small in code but significant in organisation. A new file split, a new documentation file, updates to existing documentation files, a version bump, and a regression pass. Each step is cheap; together they turn a working driver into a maintainable one.

The Chapter 18 driver is done. The chapter closes with labs, challenges, troubleshooting, and a bridge to Chapter 19, where the PCI-attached driver gains a real interrupt handler. Chapter 20 then adds MSI and MSI-X; Chapters 20 and 21 add DMA. Each of those chapters will add a file (`myfirst_intr.c`, `myfirst_dma.c`) and extend the attach and detach paths. The shape Chapter 18 established will hold.

18.10 Reading a Real Driver Together: `uart_bus_pci.c`

The preceding eight sections built Chapter 18's driver one step at a time. Before the labs, it is worth spending time with a real FreeBSD driver that follows the same pattern. `/usr/src/sys/dev/uart/uart_bus_pci.c` is a clean example. It is the PCI attach for the `uart(4)` driver, which handles PCI-attached serial ports: modem cards, chipset integrated UARTs, hypervisor serial emulation, and the console redirection chips that enterprise servers use.

Reading this file after writing Chapter 18's driver is a short exercise in pattern recognition. Nothing in the file is new. Every line maps to a concept Chapter 18 taught. The file is 366 lines; this section walks through the important parts, flagging where each piece corresponds to a Chapter 18 concept.

18.10.1 The Top of the File

```
/* -
 * SPDX-License-Identifier: BSD-2-Clause
 *
 * Copyright (c) 2006 Marcel Moolenaar All rights reserved.
 * Copyright (c) 2001 M. Warner Losh <imp@FreeBSD.org>
 * ...
 */

#include <sys/cdefs.h>
#include <sys/param.h>
#include <sys/systm.h>
#include <sys/bus.h>
#include <sys/conf.h>
#include <sys/kernel.h>
#include <sys/module.h>
#include <machine/bus.h>
#include <sys/rman.h>
#include <machine/resource.h>
```

```
#include <dev/pci/pcivar.h>
#include <dev/pci/pcireg.h>

#include <dev/uart/uart.h>
#include <dev/uart/uart_bus.h>
#include <dev/uart/uart_cpu.h>
```

The SPDX license tag is BSD-2-Clause, the standard FreeBSD license. The include list is almost identical to Chapter 18's `myfirst_pci.c`. The `dev/pci/pcivar.h` and `dev/pci/pcireg.h` includes are the PCI subsystem interface; the `dev/uart/uart.h` and friends are the driver's internal headers that Chapter 18's driver does not have equivalents of.

18.10.2 Method Table and Driver Structure

```
static device_method_t uart_pci_methods[] = {
    DEVMETHOD(device_probe,      uart_pci_probe),
    DEVMETHOD(device_attach,     uart_pci_attach),
    DEVMETHOD(device_detach,     uart_pci_detach),
    DEVMETHOD(device_resume,     uart_bus_resume),
    DEVMETHOD_END
};

static driver_t uart_pci_driver = {
    uart_driver_name,
    uart_pci_methods,
    sizeof(struct uart_softc),
};
```

Four method entries, not three: `uart(4)` also implements `device_resume` for system suspend and resume support. The resume function is `uart_bus_resume`, which lives in the core `uart(4)` driver and is reused across every UART attach variant. Chapter 18's driver skipped resume; a production-quality driver usually implements it.

The `driver_t`'s name is `uart_driver_name`, defined elsewhere in the core UART driver as `"uart"`. The `softc` size is `sizeof(struct uart_softc)`, a structure defined in `uart_bus.h`.

18.10.3 The ID Table

```
struct pci_id {
    uint16_t    vendor;
    uint16_t    device;
```

```

uint16_t    subven;
uint16_t    subdev;
const char  *desc;
int         rid;
int         rclk;
int         regshft;
};

```

The table entry is richer than Chapter 18's. The `subven` and `subdev` fields let the match discriminate among cards from different vendors that share a common chipset. The `rid` field carries the BAR's configuration-space offset (different boards use different BARs). The `rclk` carries the reference clock frequency in Hz, which varies among manufacturers. The `regshft` carries a register shift (some boards put their UART registers on 4-byte boundaries, some on 8-byte boundaries).

```

static const struct pci_id pci_ns8250_ids[] = {
    { 0x1028, 0x0008, 0xffff, 0, "Dell Remote Access Card III", 0x14,
      128 * DEFAULT_RCLK },
    { 0x1028, 0x0012, 0xffff, 0, "Dell RAC 4 Daughter Card Virtual UART",
      0x14, 128 * DEFAULT_RCLK },
    /* ... many more entries ... */
    { 0xffff, 0, 0xffff, 0, NULL, 0, 0 }
};

```

The table has dozens of entries. Each one is a board the `uart(4)` driver supports. A subvendor value of `0xffff` means “match any subvendor”. The last entry is the sentinel.

Chapter 18's driver has one entry because it targets one demo device. `uart_bus_pci.c` has dozens because the UART hardware ecosystem is large and drivers must enumerate every supported variant.

18.10.4 The Probe Routine

```

static int
uart_pci_probe(device_t dev)
{
    struct uart_softc *sc;
    const struct pci_id *id;
    struct pci_id cid = {
        .regshft = 0,
        .rclk = 0,
        .rid = 0x10 | PCI_NO_MSI,
        .desc = "Generic SimpleComm PCI device",
    };
    int result;

```

```

    sc = device_get_softc(dev);

    id = uart_pci_match(dev, pci_ns8250_ids);
    if (id != NULL) {
        sc->sc_class = &uart_ns8250_class;
        goto match;
    }
    if (pci_get_class(dev) == PCIC_SIMPLECOMM &&
        pci_get_subclass(dev) == PCIS_SIMPLECOMM_UART &&
        pci_get_progif(dev) < PCIP_SIMPLECOMM_UART_16550A) {
        id = &cid;
        sc->sc_class = &uart_ns8250_class;
        goto match;
    }
    return (ENXIO);

match:
    result = uart_bus_probe(dev, id->regshft, 0, id->rclk,
        id->rid & PCI_RID_MASK, 0, 0);
    if (result > 0)
        return (result);
    if (sc->sc_sysdev == NULL)
        uart_pci_unique_console_match(dev);
    if (id->desc)
        device_set_desc(dev, id->desc);
    return (result);
}

```

The probe is more complex than Chapter 18's. It first searches the vendor/device ID table. If that fails, it falls back to a class-based match: any device whose class is `PCIC_SIMPLECOMM` (Simple Communications) and subclass is `PCIS_SIMPLECOMM_UART` (UART controller) and interface is older than `PCIP_SIMPLECOMM_UART_16550A` (older than 16550A means "a classic UART without enhanced features"). This is the fallback probe that lets the driver handle generic UART controllers even when their vendor and device ID are not in the table.

The `match:` label is reached from either path. It calls `uart_bus_probe` (the core UART driver's probe helper) with the entry's register shift, reference clock, and BAR offset. The return value is either a `BUS_PROBE_*` priority or a positive error code. Chapter 18's driver returns `BUS_PROBE_DEFAULT` directly; `uart(4)` delegates to `uart_bus_probe` because the core driver has additional checks.

The `pci_get_class`, `pci_get_subclass`, and `pci_get_progif` accessors return the class-code fields Chapter 18 described. Their use here is a concrete example of class-based matching.

18.10.5 The Attach Routine

```
static int
uart_pci_attach(device_t dev)
{
    struct uart_softc *sc;
    const struct pci_id *id;
    int count;

    sc = device_get_softc(dev);

    id = uart_pci_match(dev, pci_ns8250_ids);
    if ((id == NULL || (id->rid & PCI_NO_MSI) == 0) &&
        pci_msi_count(dev) == 1) {
        count = 1;
        if (pci_alloc_msi(dev, &count) == 0) {
            sc->sc_irid = 1;
            device_printf(dev, "Using %d MSI message\n", count);
        }
    }

    return (uart_bus_attach(dev));
}
```

The attach is short. It re-matches the device (because the probe's match state is not preserved across probe/attach calls), checks whether the device supports MSI (single-vector), allocates one MSI vector if available, and then delegates to `uart_bus_attach` for the actual attach.

This is a pattern Chapter 18 did not use. `uart(4)` takes advantage of MSI when available, falling back to legacy IRQs otherwise. Chapter 20 in this book will introduce MSI and MSI-X; `uart(4)`'s attach is a preview.

The `PCI_NO_MSI` flag in some table entries marks boards where MSI is known to be buggy or unreliable; for those boards the attach skips MSI and relies on legacy IRQs.

18.10.6 The Detach Routine

```
static int
uart_pci_detach(device_t dev)
{
    struct uart_softc *sc;

    sc = device_get_softc(dev);
```

```
    if (sc->sc_iris != 0)
        pci_release_msi(dev);

    return (uart_bus_detach(dev));
}
```

Eight lines, every one meaningful. Release MSI if it was allocated. Delegate to `uart_bus_detach` for the rest of the teardown.

Chapter 18's detach is longer because the `myfirst` driver does not delegate to a core driver; everything is in the PCI file. `uart(4)` factors the common teardown into `uart_bus_detach`, called from every attach variant's detach.

18.10.7 The DRIVER_MODULE Line

```
DRIVER_MODULE(uart, pci, uart_pci_driver, NULL, NULL);
```

One line. The module name is `uart` (matching the `driver_t`). The bus is `pci`. The two `NULL`s are for module init and cleanup handlers that `uart(4)` does not need.

Chapter 18's driver has the same line with `myfirst` in place of `uart`.

18.10.8 What This Walkthrough Teaches

`uart_bus_pci.c` is 366 lines. About 60 lines are code; the rest is the ID table (250+ entries, many on multiple lines each) and helper functions specific to UART handling.

The code is almost indistinguishable from Chapter 18's driver in shape. A `pci_id` struct. An ID table. A probe that matches the table. An attach that claims the BAR (through `uart_bus_attach`). A detach that releases everything. `DRIVER_MODULE`. `MODULE_DEPEND`. The differences are all about UART-specific features: the subvendor match, the class-based fallback, the MSI allocation, the register-shift and reference-clock fields.

A reader who finds `uart_bus_pci.c` readable after Chapter 18 has made the chapter's point. The Chapter 18 driver is a real FreeBSD PCI driver, not a toy. It is missing a few features (MSI, resume, DMA) that later chapters will add, but its bones are the bones of every real driver in the tree.

Worth reading after `uart_bus_pci.c`, for comparison: `/usr/src/sys/dev/virtio/pci/virtio_pci_modern.c`, which is the modern (non-legacy) virtio PCI attach. It is richer than `uart_bus_pci.c` because it handles virtio's layered transport, but the shape is the same.

18.11 A Deeper Look at the PCI Capability List

Section 5 introduced `pci_find_cap(9)` and `pci_find_extcap(9)` as tools for discovering a device's optional features. This subsection goes a level deeper, showing how the capability list is structured in configuration space and how a driver might walk the whole list rather than looking up a specific capability.

18.11.1 Structure of the Legacy Capability List

The legacy capability list lives in the first 256 bytes of configuration space. It starts at an offset held in the device's `PCIR_CAP_PTR` byte (at offset `0x34`). The byte at that offset is the capability ID; the byte immediately after is the offset of the next capability (or zero if this is the last one); the remaining bytes of the capability are feature-specific.

The minimum capability header is two bytes:

```
offset 0: capability ID (one byte, values like PCIY_MSI = 0x05)
offset 1: next pointer (one byte, offset of next capability, 0 means end)
```

A driver walking the list reads the cap pointer from `PCIR_CAP_PTR`, then follows the chain by reading each capability's next byte until a zero is reached.

A concrete walk, in code:

```
static void
myfirst_dump_caps(device_t dev)
{
    uint8_t ptr, id;
    int safety = 64; /* protects against malformed lists */

    ptr = pci_read_config(dev, PCIR_CAP_PTR, 1);
    while (ptr != 0 && safety-- > 0) {
        id = pci_read_config(dev, ptr, 1);
        device_printf(dev,
            "legacy capability ID 0x%02x at offset 0x%02x\n", id, ptr);
        ptr = pci_read_config(dev, ptr + 1, 1);
    }
}
```

The safety counter guards against a malformed configuration space where the next pointer forms a cycle. A well-behaved device never produces this, but defensive code treats the configuration space as potentially adversarial.

The walk prints each capability's ID and offset. The driver can then match the IDs against the `PCIY_*` constants and handle those it supports.

18.11.2 Structure of the Extended Capability List

The PCIe extended capability list starts at offset `PCIR_EXTCAP` (`0x100`) and uses 4-byte headers. The layout, as encoded in `/usr/src/sys/dev/pci/pci.h`, is:

bits 15:0	capability ID	(PCIM_EXTCAP_ID,	mask 0x0000ffff)
bits 19:16	capability version	(PCIM_EXTCAP_VER,	mask 0x000f0000)
bits 31:20	next pointer	(PCIM_EXTCAP_NEXTPTR,	mask 0xffff00000)

FreeBSD exposes three helper macros on top of the raw masks:

- `PCI_EXTCAP_ID(header)` returns the capability ID.
- `PCI_EXTCAP_VER(header)` returns the version.
- `PCI_EXTCAP_NEXTPTR(header)` returns the next pointer (already shifted into its natural range).

The 12-bit next pointer is always 4-byte aligned; a next pointer of zero terminates the list.

A walk using the helpers:

```
static void
myfirst_dump_extcaps(device_t dev)
{
    uint32_t header;
    int off = PCIR_EXTCAP;
    int safety = 64;

    while (off != 0 && safety-- > 0) {
        header = pci_read_config(dev, off, 4);
        if (header == 0 || header == 0xffffffff)
            break;
        device_printf(dev,
            "extended capability ID 0x%04x ver %u at offset 0x%03x\n",
            PCI_EXTCAP_ID(header), PCI_EXTCAP_VER(header), off);
        off = PCI_EXTCAP_NEXTPTR(header);
    }
}
```

The walker reads the 4-byte header and unpacks it with the helpers. A zero or all-ones header means there are no extended capabilities (the latter is what a non-PCIe device returns for any extended-capability read).

18.11.3 Why the Walk Matters

A driver rarely needs the full walk. `pci_find_cap` and `pci_find_extcap` are the common interface: the driver asks for a specific capability and either gets the offset or gets `ENOENT`. A driver that wants to dump the full capability list for diagnostic purposes uses the walkers shown above.

The value of understanding the structure is in reading datasheets. A datasheet that says “the device implements the MSI capability starting at offset 0xa0” is saying: the byte at configuration-space offset 0xa0 is the capability ID (will equal 0x05 for MSI), the byte at 0xa1 is the next pointer, and the bytes from 0xa2 onward are the MSI capability structure. `pci_find_cap(dev, PCIY_MSI, &capreg)` returns `capreg = 0xa0` because that is where the capability lives.

A driver accessing the capability structure reads from `capreg + offset`, where `offset` is defined in the MSI capability's own structure. Specific fields have specific offsets; the `pcireg.h` header defines the offsets as `PCIR_MSI_*`.

18.11.4 Walking a Specific Capability's Fields

An example. The MSI capability has several fields the driver cares about, at specific offsets relative to the capability header:

```
PCIR_MSI_CTRL (0x02): message control (16 bits, enables, vector count)
PCIR_MSI_ADDR (0x04): message address low (32 bits)
PCIR_MSI_ADDR_HIGH (0x08): message address high (32 bits, 64-bit only)
PCIR_MSI_DATA (0x08 or 0x0c): message data (16 bits)
```

A driver that has `capreg` from `pci_find_cap(dev, PCIY_MSI, &capreg)` reads the message control register with:

```
uint16_t msi_ctrl = pci_read_config(dev, capreg + PCIR_MSI_CTRL, 2);
```

The macro `PCIR_MSI_CTRL` is `0x02`; the full offset is `capreg + 0x02`. Similar patterns apply to every capability.

For Chapter 18 this level of detail is not needed because the driver does not use MSI. Chapter 20 does, and uses helper functions (`pci_alloc_msi`, `pci_alloc_msix`, `pci_enable_msi`, `pci_enable_msix`) that hide the raw field access. The walker shown here is useful mostly for diagnostics and for reading datasheets.

18.12 A Deeper Look at Configuration Space

Section 1 and Section 5 introduced configuration space; this subsection adds a few practical details a driver author should know.

18.12.1 Configuration Space Layout

The first 64 bytes of every PCI configuration space are standardised. The layout is:

Offset	Width	Field
0x00	2	Vendor ID
0x02	2	Device ID
0x04	2	Command register
0x06	2	Status register
0x08	1	Revision ID

Offset	Width	Field
0x09	3	Class code (progIF, subclass, class)
0x0c	1	Cache line size
0x0d	1	Latency timer
0x0e	1	Header type
0x0f	1	BIST (built-in self test)
0x10	4	BAR 0
0x14	4	BAR 1
0x18	4	BAR 2
0x1c	4	BAR 3
0x20	4	BAR 4
0x24	4	BAR 5
0x28	4	CardBus CIS pointer
0x2c	2	Subsystem vendor ID
0x2e	2	Subsystem device ID
0x30	4	Expansion ROM base address
0x34	1	Capability list pointer
0x35	7	Reserved
0x3c	1	Interrupt line
0x3d	1	Interrupt pin
0x3e	1	Min grant
0x3f	1	Max latency

The bytes from 0x40 to 0xff are reserved for device-specific use and for the legacy capability list (starting at the offset stored in `PCIR_CAP_PTR`).

PCIe extends configuration space to 4096 bytes. The bytes from 0x100 to 0xffff hold the extended capability list, which starts at offset 0x100 and follows its own chain of 4-byte-aligned capabilities.

18.12.2 Header Type

The byte at `PCIR_HDRTYPE` (0x0e) distinguishes among three kinds of PCI configuration headers:

- 0x00: standard device (what Chapter 18 assumes).
- 0x01: PCI-to-PCI bridge (a bridge that connects a secondary bus to the primary bus).
- 0x02: CardBus bridge (a PC card bridge; increasingly obsolete).

The layout beyond offset 0x10 differs among header types. A driver for a standard device uses offsets 0x10

through 0x24 as BARs; a driver for a bridge uses the same offsets for secondary bus number, subordinate bus number, and bridge-specific registers.

The high bit of PCIR_HDRTYPE indicates a multi-function device: if set, the device has functions beyond function 0. The kernel's PCI enumerator uses this bit to decide whether to probe functions 1 through 7.

18.12.3 Commands and Status

The command register (PCIR_COMMAND, offset 0x04) holds enable bits that control the device's PCI-level behaviour:

- PCIM_CMD_PORTEN (0x0001): enable I/O BARs.
- PCIM_CMD_MEMEN (0x0002): enable memory BARs.
- PCIM_CMD_BUSMASTEREN (0x0004): allow the device to initiate DMA.
- PCIM_CMD_SERRESPEN (0x0100): report system errors.
- PCIM_CMD_INTxDIS (0x0400): disable legacy INTx assertion (used when the driver uses MSI or MSI-X instead).

The kernel sets MEMEN and PORTEN automatically during resource activation. The driver sets BUSMASTEREN through `pci_enable_busmaster` if it uses DMA. The driver sets INTxDIS when it has successfully allocated MSI or MSI-X vectors and wants to prevent the device from also raising legacy interrupts.

The status register (PCIR_STATUS, offset 0x06) holds sticky bits that the driver reads to find out about PCI-level events: the device has received a master abort, a target abort, a parity error, or a signalled system error. A driver that cares about PCI error recovery reads the status register periodically or in its error handler; a driver that does not care (most drivers, at the Chapter 18 level) ignores it.

18.12.4 Reading Wider Than Available Width

`pci_read_config(dev, offset, width)` takes a width of 1, 2, or 4. It never takes a width of 8, even though some 64-bit fields (64-bit BARs) exist in configuration space. A driver reading a 64-bit BAR does it as two 32-bit reads:

```
uint32_t bar_lo = pci_read_config(dev, PCIR_BAR(0), 4);
uint32_t bar_hi = pci_read_config(dev, PCIR_BAR(1), 4);
uint64_t bar_64 = ((uint64_t)bar_hi << 32) | bar_lo;
```

Note that this reads the *configuration-space* BAR, which the driver rarely needs after the kernel has allocated the resource. The kernel's allocation returns the same information as a `struct resource *` whose start address is available through `rman_get_start`.

18.12.5 Alignment in Configuration-Space Reads

Configuration-space accesses are aligned by design. A read of width 1 may start at any offset; a read of width 2 must start at an even offset; a read of width 4 must start at an offset divisible by 4. Unaligned accesses (for example, a width-4 read at offset 0x03) are not supported by the PCI bus's configuration transaction and will

return undefined values or an error on some implementations. Every standard field in the first 64 bytes of configuration space is laid out so that its natural width is naturally aligned, so a driver that reads each field at its documented offset and width never runs into alignment trouble.

A driver reading a vendor-specific field whose layout is unclear should read it in the width the datasheet specifies. Do not assume that a 32-bit-wide read of a 16-bit field returns well-defined values in the high bits. The PCI specification requires unused byte lanes to return zeros, but a cautious driver reads only the width it needs.

18.12.6 Writing Configuration Space: Caveats

Three caveats for configuration-space writes.

First, some fields are sticky: once set, they do not clear. The command register's `INTxDIS` bit is an example. Writing zero to the bit does not re-enable legacy interrupts in all cases; the device may latch the disabled state. A driver that needs to toggle such a bit must write the full register (read-modify-write) and may need to tolerate the device ignoring the clearing write.

Second, some fields are RW1C ("read-write-one-to-clear"). Writing a 1 to the bit clears it; writing 0 is a no-op. The status register's error bits are all RW1C. A driver that wants to clear a sticky error bit writes a 1 to that bit's position.

Third, some writes have timing requirements. The power-management capability's control register, for example, requires 10 milliseconds of settle time after a state transition. A driver that writes such a field must respect the timing, usually with a `DELAY(9)` or `pause_sbt(9)` call.

For Chapter 18's driver, only the probe's ID reads and the capability walker's reads touch configuration space. No writes are done. Chapter 19 onward will add writes (enabling interrupts, clearing status bits); each write will have the relevant caveats called out at the point it is introduced.

18.13 A Deeper Look at the `bus_space` Abstraction

Section 4 used the Chapter 16 accessor layer unchanged against a real BAR. This subsection describes, in more depth, what the `bus_space` layer does under the hood and why it matters.

18.13.1 What `bus_space_tag_t` Is

On x86, `bus_space_tag_t` is an integer that selects between two address spaces: memory (`X86_BUS_SPACE_MEM`) and I/O port (`X86_BUS_SPACE_IO`). The tag tells the accessor which CPU instructions to emit: memory accesses use normal load and store instructions; I/O port accesses use `in` and `out`.

On arm64, `bus_space_tag_t` is a pointer to a structure of function pointers (a `struct bus_space`). The tag encodes not only memory vs I/O but also properties like endianness and access granularity.

On every platform, the tag is opaque to the driver. The driver stores it, passes it to `bus_space_read_*` and `bus_space_write_*`, and never inspects its contents. The inclusion of `machine/bus.h` brings in the platform-specific definition.

18.13.2 What `bus_space_handle_t` Is

On x86 for memory space, `bus_space_handle_t` is a kernel virtual address. The accessor dereferences it as a `volatile` pointer of the appropriate width.

On x86 for I/O port space, `bus_space_handle_t` is an I/O port number (0 to 65535). The accessor uses the `in` or `out` instruction with the port number.

On arm64, `bus_space_handle_t` is a kernel virtual address, similar to x86 memory space. The platform's MMU is configured to map the physical BAR into the virtual range with device-memory attributes.

The handle is also opaque to the driver. Together with the tag, it uniquely identifies a range of addresses where a specific resource lives.

18.13.3 What Happens Inside `bus_space_read_4`

On x86 for memory space, `bus_space_read_4(tag, handle, offset)` expands to roughly:

```
static inline uint32_t
bus_space_read_4(bus_space_tag_t tag, bus_space_handle_t handle,
                 bus_size_t offset)
{
    return (*(volatile uint32_t *) (handle + offset));
}
```

A volatile pointer dereference. The `volatile` keyword prevents the compiler from caching the value or reordering the access past other volatile accesses.

On x86 for I/O port space, the implementation uses the `inl` instruction:

```
static inline uint32_t
bus_space_read_4(bus_space_tag_t tag, bus_space_handle_t handle,
                 bus_size_t offset)
{
    uint32_t value;
    __asm volatile ("inl %w1, %0" : "=a"(value) : "Nd"(handle + offset));
    return (value);
}
```

The tag selects between the two implementations. On arm64 and other platforms, the tag is richer and the implementation dispatches through a function-pointer table.

18.13.4 Why the Abstraction Matters

A driver that uses `bus_space_read_4` and `bus_space_write_4` compiles to the right CPU instructions on every supported platform. The driver author does not need to know whether the BAR is memory or I/O; does

not need to write platform-specific code; does not need to annotate pointers with the correct access attributes. The `bus_space` layer handles all of this.

A driver that bypasses `bus_space` and dereferences a raw pointer may work on x86 by accident (because the kernel's `pmap` layer happens to set up the mapping in a way that pointer accesses work). On arm64 it will fail: the device memory is mapped with attributes that prevent normal memory access patterns from working correctly.

The lesson is: always use `bus_space` or the Chapter 16 accessors that wrap it. Never dereference a raw pointer into device memory, even if you know the virtual address.

18.13.5 The `bus_read` vs `bus_space_read` Naming

FreeBSD has two families of accessor functions that do essentially the same thing. The older `bus_space_read_*` family takes a tag, a handle, and an offset. The newer `bus_read_*` family takes a `struct resource *` and an offset, and pulls the tag and handle out of the resource internally.

The newer family is more convenient; the driver stores only the resource and does not need to store the tag and handle separately. The older family is more flexible; the driver can build a tag and handle from scratch (which Chapter 16's simulation used).

Chapter 18's driver uses the older family because it inherits from Chapter 16. A rewrite could use the newer family with no semantic change. Both families produce the same output. The book's choice is to teach the tag-and-handle story because it makes the abstraction explicit; the newer family hides the abstraction, which is nicer for writing drivers but less pedagogical.

For reference, the newer family members are named like `bus_read_4(res, offset)` and `bus_write_4(res, offset, value)`. They are defined in `/usr/src/sys/sys/bus.h` as inline functions that extract the tag and handle and delegate to `bus_space_read_*` and `bus_space_write_*`.

18.14 Hands-On Labs

The labs in this chapter are structured as graduated checkpoints. Each lab builds on the previous one. A reader who works through all five labs will have a complete PCI driver, a bhyve test environment, a regression script, and a small library of diagnostic tools. Labs 1 and 2 can be done on any FreeBSD machine without a guest; labs 3, 4, and 5 require a bhyve or QEMU guest with a `virtio-rnd` device.

The time budget for each lab assumes the reader has already read the relevant sections and understood the concepts. A reader who is still learning should budget more time.

18.14.1 Lab 1: Explore Your PCI Topology

Time: thirty minutes to an hour, depending on how much time you want to spend on comprehension.

Objective: Build an intuition for PCI on your own system.

Steps:

1. Run `pciconf -lv` on your lab host and redirect the output to a file: `pciconf -lv > ~/pci-inventory.txt`.
2. Count the devices: `wc -l ~/pci-inventory.txt`. Divide by an estimate (typically 5 lines per device in the output) to get the approximate device count.
3. Identify the following classes of devices from the inventory:
 - Host bridges (class = bridge, subclass = HOST-PCI)
 - PCI-PCI bridges (class = bridge, subclass = PCI-PCI)
 - Network controllers (class = network)
 - Storage controllers (class = mass storage)
 - USB host controllers (class = serial bus, subclass = USB)
 - Graphics (class = display)
4. For each of the above, note:
 - The device's name@pciN:B:D:F string.
 - The vendor and device IDs.
 - The driver binding (look at the leading name, before the @).
5. Pick one PCI device (any unclaimed device, visible as none@. . ., works best). Note its B:D:F.
6. Run `devinfo -v | grep -B 1 -A 5 <B:D:F>` and note the resources.
7. Compare the resource list with the BAR information in the `pciconf -lv` entry.

Expected observations:

- Most devices on a modern system are on `pci0` (the primary bus) or on a bus behind a PCIe bridge. Your machine probably has three to ten visible buses.
- Every device has at least a vendor and device ID. Many have subvendor and subsystem IDs.
- Most devices are bound to a driver. Some (especially on laptops, where a manufacturer ships hardware that FreeBSD does not yet support) are unclaimed.
- The resource lists in `devinfo -v` match the BAR information you can see in `pciconf -lv`. The addresses are the ones the firmware assigned.

This lab is about building vocabulary. No code. No driver. Just reading.

18.14.2 Lab 2: Write the Probe Skeleton on Paper

Time: one to two hours.

Objective: Internalise the probe-attach-detach sequence by writing it out longhand.

Steps:

1. Open a blank file, `myfirst_pci_sketch.c`, in your editor.
2. Without looking at Section 2's finished code, write:
 - A `myfirst_pci_id` struct.
 - A `myfirst_pci_ids[]` table with one entry for a hypothetical vendor `0x1234` and device `0x5678`.
 - A `myfirst_pci_probe` function that matches against the table.
 - A `myfirst_pci_attach` function that prints `device_printf(dev, "attached\n")`.

- A `myfirst_pci_detach` function that prints `device_printf(dev, "detached\n")`.
- A `device_method_t` table with `probe`, `attach`, `detach`.
- A `driver_t` with the driver's name.
- A `DRIVER_MODULE` line.
- A `MODULE_DEPEND` line.
- A `MODULE_VERSION` line.

3. Compare your sketch to the Section 2 code. Note every difference.
4. For each difference, ask: is mine wrong, or does it work differently for a reason?
5. Update your sketch to match the Section 2 code where yours was wrong.

Expected outcomes:

- You will probably forget `MODULE_DEPEND` and `MODULE_VERSION` on first pass.
- You may use `0` instead of `BUS_PROBE_DEFAULT` in `probe` (a common beginner mistake).
- You may forget the `device_set_desc` call in `probe`.
- You may use `printf` instead of `device_printf` in `attach` and `detach`.
- You may forget `DEVMETHOD_END` at the end of the method table.

Each of these is a real mistake that produces a real bug. Finding them in your own sketch, rather than in a compiled driver at two in the morning, is the point of the lab.

18.14.3 Lab 3: Load the Stage 1 Driver in a bhyve Guest

Time: two to three hours, including setting up the guest if you do not already have one.

Objective: Observe the probe-attach-detach sequence in action.

Steps:

1. If you do not already have a bhyve guest running FreeBSD 14.3, set one up. The canonical recipe is in `/usr/share/examples/bhyve/` or in the FreeBSD Handbook. Include a `virtio-rnd` device in the guest's bhyve command line: `-s 4:0,virtio-rnd`.
2. Inside the guest, list the PCI devices: `pciconf -lv | grep -B 1 -A 2 0x1005`. Note whether the `virtio-rnd` entry is bound to `virtio_random` (a leading `virtio_random@...`), to `none` (unclaimed), or missing entirely (check your bhyve command line).
3. Copy the Chapter 18 Stage 1 source into the guest (`scp`, shared filesystem, or any method you prefer).
4. Inside the guest, in the driver's source directory: `make`. Verify that `myfirst.ko` is produced.
5. If `virtio_random` claimed the device at step 2, unload it: `sudo kldunload virtio_random`. If the device was already unclaimed (`none`), skip this step.
6. Load `myfirst`: `sudo kldload ./myfirst.ko`.
7. Check the attach: `dmesg | tail -10`. You should see the Stage 1 attach banner.
8. Check the device: `devinfo -v | grep -B 1 -A 3 myfirst`. You should see `myfirst0` as a child of `pci0`.
9. Check the binding: `pciconf -lv | grep myfirst`. You should see the entry with `myfirst0` as the device name.
10. Unload the driver: `sudo kldunload myfirst`.
11. Check the detach: `dmesg | tail -5`. You should see the Stage 1 detach banner.

12. If you unloaded `virtio_random` at step 5 and want to restore it: `sudo kldload virtio_random`.

Expected outcomes:

- Every step produces the expected output.
- If step 7's `dmesg` does not show the attach banner, the driver did not probe the device. Check that you unloaded any other driver that might have claimed it.
- If step 7 shows the attach banner but step 8 does not show `myfirst0`, there is an accounting bug in `newbus`; unlikely, but worth reporting if you see it.
- If step 10 fails with `Device busy`, the driver's detach returned `EBUSY`. At Stage 1 there is no open `cdev`; the failure is unexpected. Check the detach code.

This lab is the first time the reader's driver meets a real device. The emotional payoff is real: `myfirst0`: attaching in `dmesg` is the proof that the driver works.

18.14.4 Lab 4: Claim the BAR and Read a Register

Time: two to three hours.

Objective: Extend the Stage 1 driver to Stage 2 (BAR allocation) and Stage 3 (first real register read).

Steps:

1. Starting from the Stage 1 driver from Lab 3, edit `myfirst_pci.c` to add Stage 2's BAR allocation. Compile. Load. Verify the BAR allocation banner in `dmesg`.
2. Verify the resource is visible: `devinfo -v | grep -A 3 myfirst0` should show a memory resource.
3. Unload. Verify the detach cleanly releases the BAR.
4. Edit `myfirst_pci.c` again to add Stage 3's capability walk and first register read. Compile. Load. Verify the capability output in `dmesg`.
5. Verify that `CSR_READ_4` operates on the real BAR by reading the first four bytes of the BAR and comparing to the first four bytes of `pciconf -r myfirst0 0x00:4`. (These are different; one is configuration space, the other is the BAR. The point of comparison is that both produce plausible values without crashing.)
6. Run the full regression script from Section 7. Verify that it completes without errors.

Expected outcomes:

- The BAR allocation succeeds and the resource is visible in `devinfo -v`.
- The capability walk may show zero offsets for the `virtio-rnd` device (the legacy layout does not have PCI capabilities in the way modern devices do); this is normal.
- The first register read returns a non-zero value; the exact value depends on the device's current state.

If any step produces a crash or a page fault, consult Section 7's common mistakes and re-check each step against Section 3's allocation discipline and Section 4's tag-and-handle code.

18.14.5 Lab 5: Exercise the cdev and Verify Detach Cleanup

Time: two to three hours.

Objective: Prove that the full Chapter 18 driver works end-to-end.

Steps:

1. Starting from the Stage 3 driver from Lab 4, write a small user-space program (`myfirst_test.c`) that opens `/dev/myfirst0`, reads up to 64 bytes, writes 16 bytes, and closes the device.
2. Compile and run the program. Observe the output. Ensure that no kernel message reports an error.
3. In a second terminal, tail `dmesg` with `dmesg -w`.
4. Run the program several times, watching for any warnings or errors.
5. Run the detach-attach cycle ten times with `devctl detach myfirst0; devctl attach pci0:0:4:0`. Verify that `dmesg` shows clean attach and detach banners each cycle.
6. After the cycle, run `vmstat -m | grep myfirst` and verify that the `myfirst` malloc type has zero live allocations.
7. Unload the driver. Verify `kldstat -v | grep myfirst` returns nothing.
8. Reload the driver. Verify the attach fires again.

Expected outcomes:

- Every step succeeds.
- The `vmstat -m` check at step 6 is the most important. If it shows live allocations after the detach cycle, there is a leak that needs fixing.
- The attach-detach-reattach cycle is stable. The driver can be bound, unbound, rebound indefinitely.

This lab is the regression proof. A driver that passes Lab 5 ten times in a row without issue is a driver that Chapter 19 can extend safely.

18.14.6 Lab Summary

The five labs together take ten to fifteen hours. They produce a complete PCI driver, a working test environment, a regression script, and a small toolbox of diagnostic commands the reader can reuse in later chapters. A reader who has done all five labs has done the hands-on equivalent of reading the chapter twice: the concepts are grounded in code that ran, failures that were fixed, and outputs that were observed.

If any lab resists (the BAR allocation fails, the capability walk produces an error, the detach leaks a resource), stop and diagnose. The troubleshooting section at the end of this chapter covers the common failure modes. The labs are calibrated to work; if a lab does not work, either the lab has a subtle error (rare) or the reader's environment has a detail that differs from the author's (much more common). Either way, the diagnosis is where the real learning happens.

18.15 Challenge Exercises

The challenges build on the labs. Each challenge is optional: the chapter is complete without them. But a reader who works through them will consolidate what they learned and extend the driver in ways the chapter did not.

18.15.1 Challenge 1: Support a Second Vendor and Device ID

Extend `myfirst_pci_ids[]` with a second entry. Target a different bhyve-emulated device: `virtio-blk` (vendor `0x1af4`, device `0x1001`) or `virtio-net` (`0x1af4`, `0x1000`). Unload the corresponding base-system driver (`virtio_blk` or `virtio_net`), load `myfirst`, and verify that the attach picks up the new device.

This exercise is trivial in code (one table entry) but exercises the reader's understanding of how the probe decision is made. After the change, both virtio devices will be eligible for `myfirst` if their drivers are unloaded.

18.15.2 Challenge 2: Print the Full Capability Chain

Extend the capability-walking code in `myfirst_pci_attach` to print every capability in the list, not only the ones the driver knows about. Walk the legacy capability list starting at `PCIR_CAP_PTR` and following the next pointers; for each capability, print the ID and the offset. Do the same for the extended capability list starting at offset `0x100`.

This exercise goes beyond the chapter's treatment of `pci_find_cap`. It requires reading `/usr/src/sys/dev/pci/pcireg.h` to find the layout of capability and extended-capability headers. The output on a typical virtio-rnd device may be sparse; on a real-hardware PCIe device it is richer.

18.15.3 Challenge 3: Implement a Simple ioctl for Configuration-Space Access

Extend the `cdev`'s `ioctl` entry point to accept a request for reading configuration space. Define a new `ioctl` command `MYFIRST_IOCTL_PCI_READ_CFG` that takes a `{ offset, width }` input and returns a `uint32_t` value. Have the implementation call `pci_read_config` under `sc->mtx`.

Write a user-space program that uses the new `ioctl` to read the first 16 bytes of configuration space, byte by byte, and prints them.

This exercise introduces the reader to custom ioctls, which are a common pattern for exposing driver-specific behaviour to user space without adding new system calls.

18.15.4 Challenge 4: Refuse to Attach if the BAR Is Too Small

The Chapter 18 driver assumes BAR 0 is at least `MYFIRST_REG_SIZE` (64) bytes. A different device with the same vendor and device IDs might expose a smaller BAR. Extend the attach path to read `rman_get_size(sc->bar_res)`, compare it to `MYFIRST_REG_SIZE`, and refuse to attach (return `ENXIO` after cleanup) if the BAR is too small.

Verify the behaviour by artificially setting `MYFIRST_REG_SIZE` to a value larger than the actual BAR size. The driver should refuse to attach and `dmesg` should print an informative message.

18.15.5 Challenge 5: Split the Driver into Two Modules

Using the technique sketched in Section 5, split the driver into `myfirst_core.ko` (hardware layer, simulation, `cdev`, locks) and `myfirst_pci.ko` (PCI attach). Add a `MODULE_DEPEND(myfirst_pci, myfirst_core, 1, 1, 1)` declaration. Verify that `kldload myfirst_pci` automatically loads `myfirst_core` as a dependency.

This exercise is a moderate refactor. It introduces the reader to cross-module symbol visibility (which functions need to be exported from `myfirst_core` to `myfirst_pci`) and to the module loader's dependency resolution. The result is a clean separation between the driver's generic machinery and its PCI-specific attach.

18.15.6 Challenge 6: Reimplement probe Using Class and Subclass Matching

Instead of matching by vendor and device ID, extend the probe routine to also match by class and subclass. For example, match any device in class `PCIC_BASEPERIPH` (base peripheral) with subclass matching a chosen value. Return `BUS_PROBE_GENERIC` (a lower-priority match) when the class-based match succeeds but no vendor/device-specific entry matched.

This exercise teaches the reader how drivers coexist. The vendor-specific match wins over the class match (by returning `BUS_PROBE_DEFAULT` vs `BUS_PROBE_GENERIC`). A fallback driver can claim devices that no specific driver recognises.

18.15.7 Challenge 7: Add a Read-Only sysctl That Reports the Driver's PCI State

Add a sysctl `dev.myfirst.N.pci_info` that returns a short string describing the driver's PCI attachment: the vendor and device IDs, the subvendor and subsystem, the B:D:F, and the BAR size and address. Use `sbuf_printf` to format the string.

The result is a user-space-readable dump of the driver's view of its device. This is useful for diagnostics and becomes a pattern that drivers for more complex devices reuse.

18.15.8 Challenge 8: Simulate a Failed Attach

Introduce a sysctl `hw.myfirst.fail_attach` that, when set to 1, causes attach to fail after claiming the BAR. Verify that the goto cascade cleans up correctly and `vmstat -m | grep myfirst` shows zero leaks after the failed attach.

This exercise exercises the partial-failure path that Section 6 described but the lab sequence did not test explicitly. It is the best way to confirm that the unwind cascade is correct.

18.15.9 Challenge Summary

Eight challenges, covering a range of difficulty. A reader who completes four or five of them has deepened their understanding meaningfully. A reader who completes all eight has essentially written a second Chapter 18.

Save your solutions. Several of them (Challenge 1, Challenge 3, Challenge 7) are natural starting points for Chapter 19 extensions.

18.16 Troubleshooting and Common Mistakes

This section consolidates the common failure modes a reader may encounter in Chapter 18's labs. Each entry names the symptom, the likely cause, and the fix.

18.16.1 "Driver does not attach; no dmesg banner"

Symptom: `kldload ./myfirst.ko` returns success. `dmesg | tail` shows nothing from `myfirst`. `devinfo -v` does not list `myfirst0`.

Likely causes:

1. Another driver has claimed the target device. Check `pciconf -lv` for the device and see which driver (if any) is bound. If `virtio_random0` owns the `virtio-rnd` device, the probe priority tie goes to `virtio_random` and `myfirst` never attaches. Fix: `kldunload virtio_random` first.
2. The vendor or device ID in `myfirst_pci_ids[]` is wrong. Check against the guest's actual device. Fix: correct the IDs.
3. The probe routine has a bug that always returns `ENXIO`. Check that the comparison compares vendor and device against the table entries, not against themselves. Fix: re-read the probe code carefully.
4. The `DRIVER_MODULE` declaration is missing or wrong. Check that the third argument is the `driver_t` and the second is `"pci"`. Fix: correct the declaration.

18.16.2 “kldload panics the kernel”

Symptom: `kldload ./myfirst.ko` crashes the kernel before it returns.

Likely causes:

1. Missing `MODULE_DEPEND(myfirst, pci, ...)`. The driver tries to register against a bus that is not yet initialised. Fix: add the declaration.
2. The driver's initialisation calls a function that does not exist at module-load time. Rare, but possible if the driver defines a `MOD_LOAD` handler that accesses `device_*` functions before the bus is ready.
3. The `softc` size declared in the `driver_t` is wrong. If the attach code expects fields that are not in the declared structure, the kernel writes past the allocated block and crashes. Fix: ensure `sizeof(struct myfirst_softc)` matches the structure definition.

The debug kernel is good at catching all three; the backtrace in `ddb` will name the function where the crash occurred.

18.16.3 “BAR allocation fails with NULL”

Symptom: `bus_alloc_resource_any` returns `NULL`. `dmesg` says “cannot allocate BAR0”.

Likely causes:

1. Wrong `rid`. Use `PCIR_BAR(0)` for BAR 0, not 0. Fix: use the macro.
2. Wrong type. If the device's BAR 0 is an I/O port (bit 0 of the BAR is set in configuration space), passing `SYS_RES_MEMORY` fails. Read the BAR value with `pci_read_config(dev, PCIR_BAR(0), 4)` and check the low bit. Fix: use the correct type.
3. The BAR is already allocated by another driver or by the BIOS. Unlikely on a bhyve guest; possible on real hardware with a misconfigured BIOS. Fix: check `devinfo -v` for the claimed resources.
4. The `RF_ACTIVE` flag is missing. The resource is allocated but not activated. The handle is not usable for `bus_space` accesses. Fix: add `RF_ACTIVE`.

18.16.4 “CSR_READ_4 returns 0xffffffff”

Symptom: register reads return all ones. The reader expects non-zero values.

Likely causes:

1. The BAR is not activated. Check `RF_ACTIVE` in the `bus_alloc_resource_any` call.
2. The tag and handle are swapped. Read `rman_get_bustag` returns the tag; `rman_get_bushandle` returns the handle. Passing them to `bus_space_read_4` in the wrong order produces undefined behaviour.
3. The offset is wrong. The BAR is 32 bytes; reading at offset 64 reads past the end. The debug kernel's `KASSERT` in `myfirst_reg_read` catches this.
4. The device has been reset or powered off. Some devices return all ones when off. Read the command register with `pci_read_config(dev, PCIR_COMMAND, 2)`; if it returns `0xffff`, the device is unresponsive.

18.16.5 “kldunload returns Device busy”

Symptom: `kldunload myfirst` fails with `Device busy`.

Likely causes:

1. A user-space process has `/dev/myfirst0` open. Close the process. Check with `fstat /dev/myfirst0`.
2. The driver has an in-flight command (simulation callout, taskqueue work). Wait a few seconds and retry.
3. The detach function incorrectly returns `EBUSY` unconditionally. Check the detach code.
4. The driver's busy check has a stale reference to an uninitialised field. Check that `sc->open_count` is zero when no descriptors are open.

18.16.6 “dmesg says ‘cleanup failed in detach’”

Symptom: `dmesg` shows a warning from the detach path.

Likely causes:

1. A callout was still scheduled when detach ran. Check that `callout_drain` was called before the driver's softc cleanup.
2. A taskqueue work item was still pending. Check that `taskqueue_drain` was called.
3. The `cdev` was open at detach. The `destroy_dev` call should block until closed, but if the driver releases other resources first, the close will find stale state. Fix the order: destroy the `cdev` before releasing dependent resources.

18.16.7 “ioctl or read returns an unexpected error”

Symptom: a user-space system call returns an error that the reader did not expect (`EINVAL`, `ENODEV`, `ENXIO`, etc.).

Likely causes:

1. The cdev's entry point checks for state the driver did not set. Example: the Chapter 10 driver checks `sc->is_attached`; the Chapter 18 driver might have forgotten to set it.
2. The `ioctl` command number in user space does not match the one in the driver. Check the `_IOR/_IOW/_IOWR` macros and confirm that the types are the same.
3. The lock order is wrong. The cdev entry point takes a lock in an order that conflicts with some other code. `WITNESS` on a debug kernel reports this.

18.16.8 “vmstat -m shows leaked allocations”

Symptom: after a load-unload cycle, `vmstat -m | grep myfirst` shows non-zero “Allocations” or “InUse”.

Likely causes:

1. A `malloc` in `attach` that is not freed in `detach`. Usually the hardware-layer wrapper struct or a `sysctl` buffer.
2. A callout that was not drained. The callout allocates a small structure; if it runs after `detach`, the structure leaks.
3. The `M_MYFIRST` `malloc` type is used for the `softc`. Newbus frees the `softc` automatically; the driver should not `malloc(M_MYFIRST, sizeof(softc))` at `attach`. The `softc` is allocated by newbus.

18.16.9 “pci_find_cap returns ENOENT for a capability I know the device has”

Symptom: `pci_find_cap(dev, PCIY_EXPRESS, &capreg)` returns `ENOENT`, but the device is a PCIe device and should have the PCI Express capability.

Likely causes:

1. The device is a legacy PCI device in a PCIe slot (it works because PCIe is backward-compatible with PCI). Legacy devices do not have the PCI Express capability. Check by reading `pci_get_class(dev)` and comparing to what you expected.
2. The capability list is corrupt or empty. Read `PCIR_CAP_PTR` directly with `pci_read_config(dev, PCIR_CAP_PTR, 1)`; if it returns zero, the device does not implement capabilities.
3. Wrong capability ID. `PCIY_EXPRESS` is `0x10`, not `0x1f`. Check `pci.h` for the correct constant.
4. The status register's `PCIM_STATUS_CAPPRESENT` bit is zero. This bit tells the PCI subsystem that the device implements a capability list. Without it, the list is not present. The bit is in `PCIR_STATUS`.

18.16.10 “Module unloads, but dmesg shows a page fault during unload”

Symptom: `kldunload myfirst` appears to succeed, but `dmesg` shows a page fault that occurred during the unload.

Likely causes:

1. A callout fired after `myfirst_hw_detach` but before the driver returned. The callout accessed `sc->hw`, which had been set to `NULL`. Fix: ensure `callout_drain` is called before `myfirst_hw_detach`.

2. A taskqueue work item ran after resources were freed. Fix: ensure `taskqueue_drain` is called before freeing anything the task touches.
3. A user-space process still has `/dev/myfirst0` open. The `destroy_dev` call completes quickly, but any outstanding I/O against the `cdev` continues until the process closes the descriptor or exits. Fix: ensure all user-space consumers close the `cdev` before detach; in emergency situations, `devctl detach` followed by killing the process works.

18.16.11 “devinfo -v shows the driver attached but the cdev does not appear”

Symptom: `devinfo -v | grep myfirst` shows `myfirst0`, but `ls /dev/myfirst*` returns nothing.

Likely causes:

1. The `make_dev` call failed and the attach did not check the return value. Check `sc->cdev` after `make_dev`; if it is `NULL`, the call failed.
2. The `cdev` name is not `myfirst%d`. Check the `make_dev` call's format string. The device node path uses the exact string passed to `make_dev`.
3. The `cdevsw` structure has not been registered or has wrong methods. Check that `myfirst_cdevsw` is initialised correctly.
4. A stale `/dev` entry is hiding the new one. Try `sudo devfs rule -s 0 apply` or reboot. Unlikely on modern FreeBSD but possible in edge cases.

18.16.12 “Attach takes a long time to return”

Symptom: `kldload ./myfirst.ko` hangs for seconds or minutes.

Likely causes:

1. A `DELAY` or `pause_sbt` call in `attach` is too long. Check for hidden delays in capability walks or device bring-up.
2. A `bus_alloc_resource_any` call is blocked on a resource that another driver has allocated. Rare on PCI; more common on platforms with limited I/O port space.
3. An infinite loop in the capability walker. A malformed device could produce a loop; the safety counter in the walker protects against this.
4. A `callout_init_mtx` call is waiting on a lock that another code path holds. Deadlock; check `WITNESS` output in `dmesg`.

18.16.13 “Driver attaches on boot but never produces output for the first few seconds”

Symptom: after a guest reboot with `myfirst` loaded at boot time, the driver attaches but takes seconds to produce any log output.

Likely causes:

1. The module was loaded early in boot, before the console was fully initialised. The messages are in the kernel buffer but not yet written to the console. Check `dmesg` for the messages; they should be present.
2. A callout was scheduled but has not yet fired. Chapter 17's sensor callout runs every second; the first tick is one second after attach.
3. The driver is waiting for a condition that takes time. Not a Chapter 18 issue, but possible in drivers that wait for the device to complete a reset.

18.16.14 “A second attempt to attach after a failed first attempt succeeds”

Symptom: `kldload` on a misconfigured kernel fails; a second `kldload` after fixing the configuration succeeds. This is actually expected behaviour.

Likely cause: the kernel's module loader is stateless between load attempts. A failed load removes any partial state. A subsequent load tries again with the fresh state. The symptom is not a bug.

18.16.15 “`vmstat -m InUse` grows after each load-unload cycle”

Symptom: the `myfirst` malloc type shows a few bytes of `InUse` memory increasing every cycle.

Likely causes:

1. A leak in attach or detach that is too small to notice in a single cycle but accumulates. Run 100 cycles and watch the growth.
2. The `myfirst_hw` or `myfirst_sim` wrapper struct is allocated but not freed. Check that the detach path calls `myfirst_hw_detach` and `myfirst_sim_detach` (if the simulation is loaded).
3. A string or similar small allocation in a `sysctl` handler is leaking. Check the `sysctl` handlers for `sbuf` that are created but not deleted.

The `vmstat -m` output has columns for `Requests`, `InUse`, `MemUse`. `Requests` is the total number of allocations ever made. `InUse` is the number currently allocated. `MemUse` is the total bytes. A healthy driver's `InUse` returns to zero after detach and unload.

18.16.16 Troubleshooting Summary

Every one of these failures is recoverable. The debug kernel (with `INVARIANTS`, `WITNESS`, and `KDB`) catches most of them with a useful message. A reader who runs a debug kernel and reads the message carefully will fix most Chapter 18 bugs in under an hour.

If a bug resists, the next step is to read the relevant section of this chapter again. The troubleshooting list above is short because the chapter's teaching is deliberately designed to prevent these failures. When a failure occurs, the question is usually “which section's discipline did I break?” and the answer is usually obvious on a second reading.

18.16.17 A Debug Kernel Checklist

If you are serious about driver development and debugging, build a debug kernel. The configuration options that catch PCI-driver bugs reliably are:

```
options INVARIANTS
options INVARIANT_SUPPORT
options WITNESS
options WITNESS_SKIPSPIN
options DEBUG_VFS_LOCKS
options DEBUG_MEMGUARD
options DIAGNOSTIC
options DDB
options KDB
options KDB_UNATTENDED
options MALLOC_DEBUG_MAXZONES=8
```

A driver that passes its regression tests under a kernel with all of these enabled is a driver that will rarely produce production bugs. The runtime cost is significant (the kernel is slower, and `WITNESS` in particular adds measurable overhead to every lock operation), but the debugging value is enormous.

Build the debug kernel with:

```
cd /usr/src
sudo make buildkernel KERNCONF=GENERIC-DEBUG
sudo make installkernel KERNCONF=GENERIC-DEBUG
sudo shutdown -r now
```

Use the debug kernel for all driver development; switch back to `GENERIC` only for performance benchmarking.

18.17 Wrapping Up

Chapter 18 turned the simulated driver into a PCI driver. The starting point was `1.0-simulated`, a module with a `malloc(9)`-backed register block and a Chapter 17 simulation that made the registers breathe. The ending point is `1.1-pci`, the same module with one new file (`myfirst_pci.c`), one new header (`myfirst_pci.h`), and a handful of small extensions to the existing files. The accessor layer did not change. The command-response protocol did not change. The lock discipline did not change. What changed is the origin of the tag and handle that the accessors use.

The transition walked through eight sections. Section 1 introduced PCI as a concept, covering the topology, the B:D:F tuple, configuration space, BARs, vendor and device IDs, and the `pci(4)` subsystem. Section 2 wrote the probe-attach-detach skeleton, tied to the PCI bus with `DRIVER_MODULE(9)` and `MODULE_DEPEND(9)`. Section 3 explained what BARs are and claimed one through `bus_alloc_resource_any(9)`. Section 4 wired the claimed BAR to the Chapter 16 accessor layer, completing the transition from simulated to real register access. Section 5 added the attach-time plumbing: `pci_find_cap(9)` and `pci_find_extcap(9)` for capability

discovery, the cdev creation, and the discipline that keeps the Chapter 17 simulation inactive on the PCI path. Section 6 consolidated the detach path with strict reverse ordering, a busy check, callout and task draining, and partial-failure recovery. Section 7 tested the driver in a bhyve or QEMU guest, exercising every path the driver exposes. Section 8 refactored the code into its final shape and documented the result.

What Chapter 18 did not do is interrupt handling. The `virtio-rnd` device under bhyve has an interrupt line; our driver does not register a handler for it; the device's internal state changes do not reach the driver. The cdev is still reachable, but the data path has no active producer on the PCI build (the Chapter 17 simulation callouts are not running). Chapter 19 introduces the real handler that will give the data path a producer.

What Chapter 18 did accomplish is the crossing of a threshold. Up to the end of Chapter 17, the `myfirst` driver was a teaching module: it existed because we loaded it, not because any device required it. From Chapter 18 onward, the driver is a PCI driver: it exists because the kernel enumerated a device and our probe said yes. The newbus machinery carries the driver now. Every later Part 4 chapter extends it without changing this fundamental relationship.

The file layout has grown: `myfirst.c`, `myfirst_hw.c`, `myfirst_hw.h`, `myfirst_sim.c`, `myfirst_sim.h`, `myfirst_pci.c`, `myfirst_pci.h`, `myfirst_sync.h`, `cbuf.c`, `cbuf.h`, `myfirst.h`. The documentation has grown: `HARDWARE.md`, `LOCKING.md`, `SIMULATION.md`, `PCI.md`. The test suite has grown: the bhyve or QEMU setup scripts, the regression script, the small user-space test programs. Each of these is a layer; each of them was introduced in a specific chapter and is now a permanent part of the driver's story.

18.17.1 A Reflection Before Chapter 19

A pause before the next chapter. Chapter 18 taught the PCI subsystem and the newbus attach dance. The patterns you practised here (probe-attach-detach, resource claim-release, tag-and-handle extraction, capability discovery) are patterns you will use throughout your driver-writing life. They apply as much to the USB attach dance in Chapter 21 as to the PCI dance you just wrote, as much to the NIC driver you might write for a real card as to the demo driver you just extended. The PCI skill is permanent.

Chapter 18 also taught the discipline of strict reverse ordering in detach. The goto cascade in attach, the mirror detach, the busy check, the quiesce step: these are the patterns that keep a driver from leaking resources across its lifecycle. They apply to every kind of driver, not only PCI. A reader who has internalised Chapter 18's detach discipline will write cleaner Chapter 19, Chapter 20, and Chapter 21 code.

One more observation. The payoff of Chapter 16's accessor layer is now visible. A reader who wrote the Chapter 16 accessors and wondered "is this worth the effort?" can look at Chapter 18's Stage 3 attach and see the answer. The upper-layer code of the driver (every call site that uses `CSR_READ_4`, `CSR_WRITE_4`, or `CSR_UPDATE_4`) did not change at all when the backend switched from simulated to real PCI. That is what a good abstraction buys: a major change in the lower layer costs zero changes in the upper layer. The Chapter 16 accessors were the abstraction. Chapter 18 was the proof.

18.17.2 What to Do If You Are Stuck

Two suggestions.

First, focus on the regression script from Section 7. If the script runs end-to-end without error, the driver is working; every confusion about internal details is decorative. If the script fails, the first failing step is the starting point for debugging.

Second, open `/usr/src/sys/dev/uart/uart_bus_pci.c` and read it slowly. The file is 366 lines. Every line is a pattern that Chapter 18 taught or referenced. Reading it after Chapter 18 should feel familiar: probe, attach, detach, ID table, `DRIVER_MODULE`, `MODULE_DEPEND`. A reader who finds the file readable after Chapter 18 has made the chapter's real progress.

Third, skip the challenges on first pass. The labs are calibrated for Chapter 18; the challenges assume the chapter's material is already solid. Come back to them after Chapter 19 if they feel out of reach now.

Chapter 18's goal was to let the driver meet real hardware. If it has, the rest of Part 4 will feel like a natural progression: Chapter 19 adds interrupts, Chapter 20 adds MSI and MSI-X, Chapters 20 and 21 add DMA. Each chapter extends what Chapter 18 established.

18.18 Bridge to Chapter 19

Chapter 19 is titled *Handling Interrupts*. Its scope is the topic Chapter 18 deliberately did not take: the path that lets a device tell the driver, asynchronously, that something happened. Chapter 17's simulation used callouts to produce autonomous state changes. Chapter 18's real-PCI driver ignores the device's interrupt line entirely. Chapter 19 registers a handler through `bus_setup_intr(9)`, attaches it to an IRQ resource allocated through `bus_alloc_resource_any(9)` with `SYS_RES_IRQ`, and teaches the driver to react to the device's own signals.

Chapter 18 prepared the ground in four specific ways.

First, **you have a PCI-attached driver**. The Chapter 18 driver at `1.1-pci` allocates a BAR, claims a memory resource, and has every newbus hook in place. Chapter 19 adds one more resource (an IRQ) and one more pair of calls (`bus_setup_intr` and `bus_teardown_intr`). The rest of the attach and detach flow stays put.

Second, **you have an accessor layer that can be called from an interrupt context**. The Chapter 16 accessors take `sc->mtx`; an interrupt handler that needs to read or write a register acquires `sc->mtx` and calls `CSR_READ_4` or `CSR_WRITE_4`. The Chapter 19 handler will compose with the accessors without any new plumbing.

Third, **you have a detach order that accommodates IRQ teardown**. Chapter 18's detach releases the BAR at a specific point in the sequence; Chapter 19's detach will release the IRQ resource before releasing the BAR. The goto cascade expands by one label; the pattern does not change.

Fourth, **you have a test environment that produces interrupts**. The bhyve or QEMU guest with a `virtio-rnd` device is the same environment Chapter 19 uses; the `virtio-rnd` device's interrupt line is what Chapter 19's handler receives. No new lab setup is required.

Specific topics Chapter 19 will cover:

- What an interrupt is, in contrast to a polled callout.
- The two-stage model of FreeBSD interrupt handlers: filter (fast, in interrupt context) and `ithread` (slow, in a kernel thread context).
- `bus_alloc_resource_any(9)` with `SYS_RES_IRQ`.
- `bus_setup_intr(9)` and `bus_teardown_intr(9)`.
- `INTR_TYPE_*` and `INTR_MPSAFE` flags.
- What an interrupt handler may and may not do (no sleeping, no blocking locks, no `malloc(M_WAITOK)`).
- Reading a status register at interrupt time to decide what happened.

- Clearing interrupt flags to prevent re-entry.
- Logging interrupts safely.
- Interaction between interrupts and the Chapter 16 access log.
- A minimal interrupt handler that increments a counter and logs.

You do not need to read ahead. Chapter 18 is sufficient preparation. Bring your `myfirst` driver at `1.1-pci`, your `LOCKING.md`, your `HARDWARE.md`, your `SIMULATION.md`, your new `PCI.md`, your WITNESS-enabled kernel, and your regression script. Chapter 19 starts where Chapter 18 ended.

Chapter 20 is two chapters further along; it is worth a brief forward pointer. MSI and MSI-X will replace the single legacy interrupt line with a richer routing mechanism: separate vectors for separate tasks, interrupt coalescing, per-queue affinity. The `pci_alloc_msi(9)` and `pci_alloc_msix(9)` functions are part of the PCI subsystem Chapter 18 introduced; we saved them for Chapter 20 because MSI-X in particular requires a deeper understanding of interrupt handling than Chapter 18 was ready to introduce. If the reader has glanced at the `PCIY_MSI` and `PCIY_MSIX` offsets in the capability walk and wondered what they were for, Chapter 20 is the answer.

The hardware conversation is deepening. The vocabulary is yours; the protocol is yours; the discipline is yours. Chapter 19 adds the next missing piece.

18.19 Reference: PCI Header Offsets the Chapter Used

A compact reference of the configuration-space offsets Chapter 18 references, sourced from `/usr/src/sys/dev/pci/pci.h`. Keep this at hand while writing PCI code.

Offset	Macro	Width	Meaning
0x00	PCIR_VENDOR	2	Vendor ID
0x02	PCIR_DEVICE	2	Device ID
0x04	PCIR_COMMAND	2	Command register
0x06	PCIR_STATUS	2	Status register
0x08	PCIR_REVID	1	Revision ID
0x09	PCIR_PROGIF	1	Programming interface
0x0a	PCIR_SUBCLASS	1	Subclass
0x0b	PCIR_CLASS	1	Class
0x0c	PCIR_CACHELNSZ	1	Cache line size
0x0d	PCIR_LATTIMER	1	Latency timer
0x0e	PCIR_HDRTYPE	1	Header type
0x0f	PCIR_BIST	1	Built-in self test
0x10	PCIR_BAR(0)	4	BAR 0
0x14	PCIR_BAR(1)	4	BAR 1

Offset	Macro	Width	Meaning
0x18	PCIR_BAR(2)	4	BAR 2
0x1c	PCIR_BAR(3)	4	BAR 3
0x20	PCIR_BAR(4)	4	BAR 4
0x24	PCIR_BAR(5)	4	BAR 5
0x2c	PCIR_SUBVEND_0	2	Subsystem vendor
0x2e	PCIR_SUBDEV_0	2	Subsystem device
0x34	PCIR_CAP_PTR	1	Capability list start
0x3c	PCIR_INTLINE	1	Interrupt line
0x3d	PCIR_INTPIN	1	Interrupt pin

18.19.1 Command Register Bits

Bit	Macro	Meaning
0x0001	PCIM_CMD_PORTEN	Enable I/O space
0x0002	PCIM_CMD_MEMEN	Enable memory space
0x0004	PCIM_CMD_BUSMASTEREN	Enable bus master
0x0008	PCIM_CMD_SPECIALEN	Enable special cycles
0x0010	PCIM_CMD_MWRICEN	Memory write and invalidate
0x0020	PCIM_CMD_PERRESPEN	Parity error response
0x0040	PCIM_CMD_SERRESPEN	SERR# enable
0x0400	PCIM_CMD_INTxDIS	Disable INTx generation

18.19.2 Capability IDs (legacy)

Value	Macro	Meaning
0x01	PCIY_PMG	Power Management
0x05	PCIY_MSI	Message Signaled Interrupts
0x09	PCIY_VENDOR	Vendor-specific
0x10	PCIY_EXPRESS	PCI Express
0x11	PCIY_MSIX	MSI-X

18.19.3 Extended Capability IDs (PCIe)

Value	Macro	Meaning
0x0001	PCIZ_AER	Advanced Error Reporting
0x0002	PCIZ_VC	Virtual Channel
0x0003	PCIZ_SERNUM	Device Serial Number
0x0004	PCIZ_PWRBDGT	Power Budgeting
0x000d	PCIZ_ACS	Access Control Services
0x0010	PCIZ_SRIOV	Single Root I/O Virtualisation

A reader who needs other PCI constants should open `/usr/src/sys/dev/pci/pci.h` directly. The file is well commented; finding a specific offset or bit takes less than a minute.

18.20 Reference: A Comparison with Chapter 16 and Chapter 17 Patterns

A side-by-side comparison of where Chapter 18 extends Chapters 16 and 17 and where it introduces genuinely new material.

Pattern	Chapter 16	Chapter 17	Chapter 18
Register access	CSR_READ_4, etc.	Same API, unchanged	Same API, unchanged
Access log	Introduced	Extended with fault-injection entries	Unchanged
Lock discipline	sc->mtx around each access	Same, plus callouts	Same
File layout	myfirst_hw.c added	myfirst_sim.c added	myfirst_pci.c added
Register map	10 registers, 40 bytes	16 registers, 60 bytes	Same
Attach routine	Simple (malloc block)	Simple (malloc block plus sim setup)	Real PCI BAR claim
Detach routine	Simple	Same plus callout drain	Same plus BAR release
Module loading	kldload triggers load	Same	kldload plus PCI probe
Device instance	Global (implicit)	Global	Per-PCI-device, numbered
BAR	N/A	N/A	BAR 0, SYS_RES_MEMORY, RF_ACTIVE
Capability walk	N/A	N/A	pci_find_cap / pci_find_extcap
cdev	Created at module load	Same	Created per attach

Pattern	Chapter 16	Chapter 17	Chapter 18
Version	0.9-mmio	1.0-simulated	1.1-pci
Documentation	HARDWARE.md introduced	SIMULATION.md introduced	PCI.md introduced

Chapter 18 builds on Chapters 16 and 17 without breaking anything. Every earlier-chapter capability is preserved; the real-PCI attach is added as a new backend that composes with the existing structure. The driver at 1.1-pci is a strict superset of the driver at 1.0-simulated.

18.21 Reference: Patterns From Real FreeBSD PCI Drivers

A short tour of patterns that appear repeatedly in the `/usr/src/sys/dev/` tree. Each pattern is a concrete snippet from a real driver, rewritten slightly for readability, with a pointer to the file and a short note on why the pattern matters. Reading these patterns after Chapter 18 consolidates the vocabulary.

18.21.1 Pattern: Walking BARs by Type

From `/usr/src/sys/dev/e1000/if_em.c`:

```
for (rid = PCIR_BAR(0); rid < PCIR_CIS;) {
    val = pci_read_config(dev, rid, 4);
    if (EM_BAR_TYPE(val) == EM_BAR_TYPE_IO) {
        break;
    }
    rid += 4;
    if (EM_BAR_MEM_TYPE(val) == EM_BAR_MEM_TYPE_64BIT)
        rid += 4;
}
```

This loop walks the BAR table looking for the I/O port BAR. It reads each BAR's configuration-space value, checks its type bit, and advances by 4 bytes (one BAR slot) or 8 bytes (two slots, for a 64-bit memory BAR). The loop terminates either at `PCIR_CIS` (the CardBus pointer, which is just beyond the BAR table) or when it finds an I/O BAR.

Why it matters: on drivers that support a mix of memory and I/O BARs across a range of hardware revisions, the BAR layout is not fixed. Walking dynamically is the right approach. Chapter 18's driver targets one device with a known BAR layout and does not need this walker; a driver like `em(4)` that covers a family of chips does.

18.21.2 Pattern: Matching on Class, Subclass, and progIF

From `/usr/src/sys/dev/uart/uart_bus_pci.c`:

```

if (pci_get_class(dev) == PCIC_SIMPLECOMM &&
    pci_get_subclass(dev) == PCIS_SIMPLECOMM_UART &&
    pci_get_progif(dev) < PCIP_SIMPLECOMM_UART_16550A) {
    id = &cid;
    sc->sc_class = &uart_ns8250_class;
    goto match;
}

```

This snippet is a class-based fallback. If the vendor-and-device match failed, the probe falls back to matching any device advertising “simple communications / UART / pre-16550A” in its class code. The progIF field distinguishes 16450, 16550A, and later variants; the snippet specifically targets the older ones.

Why it matters: class codes let a driver attach to families of devices the specific match table does not enumerate. A UART chip from a vendor not in the `uart(4)` table is still handled as long as the class code is standard. The pattern works well for standardised device types (AHCI, xHCI, UART, NVMe, HD Audio) where the programming interface is class-defined.

18.21.3 Pattern: Conditional MSI Allocation

From `/usr/src/sys/dev/uart/uart_bus_pci.c`:

```

id = uart_pci_match(dev, pci_ns8250_ids);
if ((id == NULL || (id->rid & PCI_NO_MSI) == 0) &&
    pci_msi_count(dev) == 1) {
    count = 1;
    if (pci_alloc_msi(dev, &count) == 0) {
        sc->sc_irid = 1;
        device_printf(dev, "Using %d MSI message\n", count);
    }
}

```

This snippet allocates MSI if the device supports it and the driver has not marked the entry with `PCI_NO_MSI`. The `pci_msi_count(dev)` call returns the number of MSI vectors the device advertises; `pci_alloc_msi` allocates them. The `sc->sc_irid = 1` line reflects the rid assigned to the MSI resource (MSI resources start at rid 1; legacy IRQs use rid 0).

Why it matters: MSI is preferred over legacy IRQs on modern systems because it avoids the IRQ-sharing problems of the `INTx` pin. A driver that supports MSI and falls back to legacy IRQs when MSI is unavailable is the right pattern. Chapter 20 returns to MSI in detail; the snippet here is a preview.

18.21.4 Pattern: Detach-Time IRQ Release

From `/usr/src/sys/dev/uart/uart_bus_pci.c`:

```
static int
uart_pci_detach(device_t dev)
{
    struct uart_softc *sc;

    sc = device_get_softc(dev);

    if (sc->sc_irid != 0)
        pci_release_msi(dev);

    return (uart_bus_detach(dev));
}
```

Detach releases MSI (if allocated) and delegates the rest to `uart_bus_detach`. The `sc->sc_irid != 0` check guards against calling `pci_release_msi` on a driver that used legacy IRQs; releasing MSI when it was not allocated is an error.

Why it matters: every resource allocated at attach must be released at detach. The driver tracks what it allocated through state (here, `sc_irid != 0` means MSI was used) and releases accordingly. Chapter 19 and Chapter 20 will extend Chapter 18's detach in a similar pattern.

18.21.5 Pattern: Reading Vendor-Specific Configuration Fields

From `/usr/src/sys/dev/virtio/pci/virtio_pci_modern.c` (simplified):

```
cap_offset = 0;
while (pci_find_next_cap(dev, PCIY_VENDOR, cap_offset, &cap_offset) == 0) {
    uint8_t cap_type = pci_read_config(dev,
        cap_offset + VIRTIO_PCI_CAP_TYPE, 1);
    if (cap_type == VIRTIO_PCI_CAP_COMMON_CFG) {
        /* This is the capability we're looking for. */
        break;
    }
}
```

This walks every vendor-specific capability in the list (`ID = PCIY_VENDOR = 0x09`), checking each one's vendor-defined type byte until it finds the one the driver wants. The `pci_find_next_cap` function is the iterating version of `pci_find_cap`, picking up where the last call left off.

Why it matters: when multiple capabilities share the same ID (as with virtio's vendor-specific capabilities), the driver must walk and disambiguate by reading the capability's own type field. The `pci_find_next_cap` function exists specifically for this case.

18.21.6 Pattern: A Power-Aware Resume Handler

From various drivers:

```
static int
myfirst_pci_resume(device_t dev)
{
    struct myfirst_softc *sc = device_get_softc(dev);

    /* Restore the device to its pre-suspend state. */
    MYFIRST_LOCK(sc);
    CSR_WRITE_4(sc, MYFIRST_REG_CTRL, sc->saved_ctrl);
    CSR_WRITE_4(sc, MYFIRST_REG_INTR_MASK, sc->saved_intr_mask);
    MYFIRST_UNLOCK(sc);

    /* Re-enable the user-space interface. */
    return (0);
}
```

A suspend handler saves the device state; the resume handler restores it. The pattern matters for systems that support suspend-to-RAM (S3) or suspend-to-disk (S4); a driver that does not implement suspend and resume prevents the system from entering those states.

Chapter 18's driver does not implement suspend and resume. Chapter 22 adds them.

18.21.7 Pattern: Responding to a Device-Specific Error State

From `/usr/src/sys/dev/e1000/if_em.c`:

```
if (reg_icr & E1000_ICR_RX0)
    sc->rx_overruns++;
if (reg_icr & E1000_ICR_LSC)
    em_handle_link(ctx);
if (reg_icr & E1000_ICR_INT_ASSERTED) {
    /* ... */
}
```

After an interrupt, the driver reads the interrupt cause register (`reg_icr`) and dispatches based on which bits are set. Each bit corresponds to a different event: receive overrun, link state change, general interrupt. The driver takes a different action for each.

Why it matters: a real driver handles many event types. The dispatching pattern is familiar from Chapter 17's fault injection, where the simulation could inject different fault types. Chapter 19 will introduce the interrupt-handling version of this pattern.

18.21.8 Pattern: Using sysctl to Expose Driver Configuration

From any number of drivers:

```
SYSCALL_ADD_U32(&sc->sysctl_ctx,  
    SYSCALL_CHILDREN(sc->sysctl_tree), OID_AUTO,  
    "max_retries", CTLFLAG_RW,  
    &sc->max_retries, 0,  
    "Maximum retry attempts");
```

Drivers expose tunable parameters through sysctl. The parameter can be read or written from user space with `sysctl dev.myfirst.0.max_retries`. A driver that exposes a handful of such tunables gives its operators a way to adjust behaviour without rebuilding the driver.

Why it matters: sysctl is the right place for per-driver tunables. Kernel command-line options (tunables set at boot time) are for early-boot parameters only; runtime adjustment goes through sysctl.

18.21.9 Pattern: Recording Supported Features in a Capabilities Structure

From `/usr/src/sys/dev/virtio/pci/virtio_pci_modern.c`:

```
sc->vtpci_modern_res.vtprm_common_cfg_cap_off = common_cfg_off;  
sc->vtpci_modern_res.vtprm_notify_cap_off = notify_off;  
sc->vtpci_modern_res.vtprm_isr_cfg_cap_off = isr_cfg_off;  
sc->vtpci_modern_res.vtprm_device_cfg_cap_off = device_cfg_off;
```

The driver stores the offsets of each capability in a per-device state structure. Later code that needs to access a capability's registers reaches it through the stored offset.

Why it matters: after attach, the driver should not need to re-walk the capability list. Storing the offsets at attach time saves a walk on every access. Chapter 18's driver walks the capabilities for informational purposes but does not store the offsets because it does not use them. A real driver that cares about a capability stores its offset.

18.21.10 Pattern Summary

The patterns above are the common currency of FreeBSD PCI drivers. A reader who recognises them in unfamiliar code is a reader who can learn from any driver in the tree. Chapter 18 taught the base patterns; the real drivers layer specific variations on top. The specific variations are always small (a class-based match here, an MSI allocation there); the base patterns are what recur.

After finishing Chapter 18 and the labs, pick a driver in `/usr/src/sys/dev/` you find interesting (perhaps for a device you own, or perhaps just one whose name you recognise) and read its PCI attach. Use this section as a checklist: which patterns does the driver use? Which ones does it skip? Why? A driver author who has done this exercise three or four times across different drivers has built an enormous fund of pattern recognition.

18.22 Reference: A Closing Note on PCI Driver Philosophy

A paragraph to close the chapter with, worth returning to after the labs.

A PCI driver's job is not to understand the device. A PCI driver's job is to present the device to the kernel in a form the kernel can use. The understanding of the device (what its registers mean, what protocol it speaks, what invariants it maintains) belongs to the driver's upper layers: the hardware abstraction, the protocol implementation, the user-space interface. The PCI layer is a narrow thing. It matches a vendor and device ID. It claims a BAR. It hands the BAR to the upper layers. It registers an interrupt handler. It hands control to the upper layers. It exists to connect two halves of the driver's identity: the device half, which belongs to the hardware, and the software half, which belongs to the kernel.

A reader who has written the Chapter 18 driver has written one PCI layer. It is small. The rest of the driver is what makes it useful. In Chapter 19 the driver's PCI layer will gain one more responsibility (interrupt registration). In Chapter 20 it will gain MSI and MSI-X. In Chapters 20 and 21 it will manage DMA tags. Each of these is a narrow extension of the PCI layer's existing role. None of them changes the PCI layer's fundamental character.

For this reader and for this book's future readers, the Chapter 18 PCI layer is a permanent piece of the `myfirst` driver's architecture. Every later chapter assumes it. Every later chapter extends it. The driver's overall complexity will grow, but the PCI layer will remain what Chapter 18 made it: a connector between the device and the rest of the driver, small and predictable.

The skill Chapter 18 teaches is not "how to write a driver for `virtio-rnd`". It is "how to connect a driver to a PCI device, regardless of what the device is". That skill is transferable, and it is the skill that will serve you across every PCI driver you write.

18.23 Reference: Chapter 18 Quick-Reference Card

A compact summary of the vocabulary, APIs, macros, and procedures Chapter 18 introduced. Useful as a single-page refresher while working through Chapter 19 and later chapters.

18.23.1 Vocabulary

- **PCI**: Peripheral Component Interconnect, the shared parallel bus introduced by Intel in the early 1990s.
- **PCIe**: PCI Express, the modern serial successor. Same software-visible model as PCI.
- **B:D:F**: Bus, Device, Function. The address of a PCI device. Written `pciN:B:D:F` in FreeBSD output.
- **Configuration space**: the small metadata area each PCI device exposes. 256 bytes on PCI, 4096 bytes on PCIe.
- **BAR**: Base Address Register. A configuration-space field where a device advertises a range of addresses it needs.
- **Vendor ID**: 16-bit identifier assigned by the PCI-SIG to a manufacturer.
- **Device ID**: 16-bit identifier assigned by the vendor to a specific product.
- **Subvendor/subsystem ID**: secondary 16+16-bit tuple identifying the board.
- **Capability list**: a linked list of optional feature blocks in configuration space.
- **Extended capability list**: the PCIe-specific list, starting at offset `0x100`.

18.23.2 Essential APIs

- `pci_get_vendor(dev)` / `pci_get_device(dev)`: read cached ID fields.
- `pci_get_class(dev)` / `pci_get_subclass(dev)` / `pci_get_progif(dev)` / `pci_get_revid(dev)`: read cached classification fields.
- `pci_get_subvendor(dev)` / `pci_get_subdevice(dev)`: read cached subsystem identification.
- `pci_read_config(dev, offset, width)` / `pci_write_config(dev, offset, val, width)`: raw configuration-space access (width 1, 2, or 4).
- `pci_find_cap(dev, cap, &offset)` / `pci_find_next_cap(dev, cap, start, &offset)`: walk the legacy capability list.
- `pci_find_extcap(dev, cap, &offset)` / `pci_find_next_extcap(dev, cap, start, &offset)`: walk the PCIe extended capability list.
- `pci_enable_busmaster(dev)` / `pci_disable_busmaster(dev)`: toggle the bus-master enable bit.
- `pci_msi_count(dev)` / `pci_msix_count(dev)`: report MSI and MSI-X vector counts.
- `pci_alloc_msi(dev, &count)` / `pci_alloc_msix(dev, &count)`: allocate MSI or MSI-X vectors (Chapter 20).
- `pci_release_msi(dev)`: release MSI or MSI-X.
- `bus_alloc_resource_any(dev, type, &rid, flags)`: claim a resource (BAR, IRQ, etc.).
- `bus_release_resource(dev, type, rid, res)`: release a claimed resource.
- `rman_get_bustag(res)` / `rman_get_bushandle(res)`: extract the bus_space tag and handle.
- `rman_get_start(res)` / `rman_get_size(res)` / `rman_get_end(res)`: inspect a resource's range.
- `bus_space_read_4(tag, handle, off)` / `bus_space_write_4(tag, handle, off, val)`: the low-level accessors.
- `bus_read_4(res, off)` / `bus_write_4(res, off, val)`: resource-based shorthand.

18.23.3 Essential Macros

- `DEVMETHOD(device_probe, probe_fn)` and similar: populate a method table.
- `DEVMETHOD_END`: terminate a method table.
- `DRIVER_MODULE(name, bus, driver, modev_fn, modev_arg)`: register a driver against a bus.
- `MODULE_DEPEND(name, dep, minver, prefver, maxver)`: declare a module dependency.
- `MODULE_VERSION(name, version)`: declare the driver's version.
- `PCIR_BAR(n)`: compute the configuration-space offset of BAR n.
- `BUS_PROBE_DEFAULT`, `BUS_PROBE_GENERIC`, `BUS_PROBE_VENDOR`, `BUS_PROBE_SPECIFIC`: probe priority values.
- `SYS_RES_MEMORY`, `SYS_RES_IOPORT`, `SYS_RES_IRQ`: resource types.
- `RF_ACTIVE`, `RF_SHAREABLE`: resource allocation flags.

18.23.4 Common Procedures

Attach a PCI driver to a specific device ID:

1. Write a probe that reads `pci_get_vendor(dev)` and `pci_get_device(dev)`, compares against a table, returns `BUS_PROBE_DEFAULT` on match and `ENXIO` otherwise.

2. Write an attach that calls `bus_alloc_resource_any(dev, SYS_RES_MEMORY, &rid, RF_ACTIVE)` with `rid = PCIR_BAR(0)`.
3. Extract the tag and handle with `rman_get_bustag` and `rman_get_bushandle`.
4. Store them where the accessor layer can reach them.

Release a PCI resource at detach:

1. Drain any callouts or tasks that might access the resource.
2. Release the resource with `bus_release_resource(dev, type, rid, res)`.
3. Set the stored resource pointer to NULL.

Unload a conflicting base-system driver before loading your own:

```
sudo kldunload virtio_random    # or whatever driver owns the device
sudo kldload ./myfirst.ko
```

Force a device to rebind from one driver to another:

```
sudo devctl detach <driver0_name>
sudo devctl set driver -f <pci_selector> <new_driver_name>
```

18.23.5 Useful Commands

- `pciconf -lv`: list every PCI device with its IDs, class, and driver binding.
- `pciconf -r <selector> <offset>:<length>`: dump configuration-space bytes.
- `pciconf -w <selector> <offset> <value>`: write a configuration-space value.
- `devinfo -v`: dump the newbus tree with resources and bindings.
- `devctl detach, attach, disable, enable, rescan`: control bus bindings at runtime.
- `dmesg, dmesg -w`: view (and follow) the kernel message buffer.
- `kldstat -v`: list loaded modules with verbose info.
- `kldload, kldunload`: load and unload kernel modules.
- `vmstat -m`: report memory allocations by malloc type.

18.23.6 Files to Keep Bookmarked

- `/usr/src/sys/dev/pci/pci.h`: PCI register definitions (`PCIR_*`, `PCIM_*`, `PCIY_*`, `PCIZ_*`).
- `/usr/src/sys/dev/pci/pciivar.h`: PCI accessor function declarations.
- `/usr/src/sys/sys/bus.h`: newbus method and resource macros.
- `/usr/src/sys/sys/rman.h`: resource manager accessors.
- `/usr/src/sys/sys/module.h`: module registration macros.
- `/usr/src/sys/dev/uart/uart_bus_pci.c`: a clean, readable example PCI driver.
- `/usr/src/sys/dev/virtio/pci/virtio_pci_modern.c`: a modern transport example.

18.24 Reference: Glossary of Chapter 18 Terms

A short glossary for readers who want a compact reminder of Chapter 18's vocabulary.

AER (Advanced Error Reporting): a PCIe extended capability that reports transaction-layer errors to the OS.

Attach: the newbus method a driver implements to take ownership of a specific device instance. Called once per device, after probe succeeds.

Bar (Base Address Register): a configuration-space field where a device advertises one range of addresses it needs mapped.

Bus Master: a device that initiates its own transactions on the PCI bus. Necessary for DMA. Enabled through the BUSMASTEREN bit of the command register.

Capability: an optional feature block in configuration space. Discovered by walking the capability list.

Class code: a three-byte classification (class, subclass, programming interface) that categorises the device's function.

cdev: a character device node in /dev/, created by `make_dev(9)`.

Configuration space: the per-device metadata area. 256 bytes on PCI, 4096 bytes on PCIe.

Detach: the newbus method that undoes everything attach did. Called once per device, when the driver is being unbound.

device_t: the opaque handle the newbus layer passes to driver methods.

DRIVER_MODULE: a macro that registers a driver against a bus and wraps it as a kernel module.

ENXIO: the errno a probe returns to say "I do not match this device".

EBUSY: the errno a detach returns to say "I refuse to detach; the driver is in use".

IRQ: an interrupt request. In PCI, the PCIR_INTLINE configuration-space field holds the legacy IRQ number.

Legacy interrupt (INTx): the pin-based interrupt mechanism inherited from PCI. Superseded by MSI and MSI-X on modern systems.

MMIO (Memory-Mapped I/O): the pattern of accessing device registers through memory-like load and store instructions.

MSI / MSI-X: Message Signaled Interrupts; interrupt mechanisms that use writes to specific memory addresses rather than pin assertions. Chapter 20.

Newbus: FreeBSD's device tree abstraction. Every device has a parent bus and a driver.

PCI: the older parallel bus standard.

PCIe: the modern serial successor to PCI. Software-compatible with PCI.

PIO (Port-Mapped I/O): the pattern of accessing device registers through x86 in and out instructions. Largely obsolete.

Probe: the newbus method that tests whether the driver can handle a specific device. Must be idempotent.

Resource: a generic name for a kernel-managed device resource (memory range, I/O port range, IRQ). Allocated through `bus_alloc_resource_any(9)`.

Softc (software context): the per-device state structure a driver maintains. Sized through the `driver_t` and allocated by `newbus`.

Subclass: the middle byte of the class code; refines the class.

Subvendor / subsystem ID: a second-level identification tuple that refines the primary vendor/device pair for distinct board designs.

Vendor ID: the 16-bit manufacturer identifier assigned by the PCI-SIG.

19 Handling Interrupts

19.1 Reader Guidance & Outcomes

Chapter 18 ended with a driver that had finally met real PCI hardware. The `myfirst` module at version `1.1-pci` probes a PCI device by vendor and device ID, attaches as a proper newbus child of `pci0`, claims the device's BAR through `bus_alloc_resource_any(9)` with `SYS_RES_MEMORY` and `RF_ACTIVE`, hands the BAR to the Chapter 16 accessor layer so `CSR_READ_4` and `CSR_WRITE_4` read and write real silicon, creates a per-instance `cdev`, and tears everything down in strict reverse order on detach. The Chapter 17 simulation is still in the tree but does not run on the PCI path; its callouts stay silent so they cannot write into the real device's registers.

What the driver does not yet do is react to the device. Every register access so far has been initiated by the driver: a user-space read or write hits the `cdev`, the `cdev` handler acquires `sc->mtx`, the accessor reads or writes the BAR, and control returns to user space. If the device itself has something to say, such as “my receive queue has a packet” or “a command completed” or “a temperature threshold was crossed”, the driver has no way of hearing it. The driver polls; it does not listen.

That is what Chapter 19 fixes. Real devices talk by **interrupting** the CPU. The bus fabric carries a signal from the device to the interrupt controller, the interrupt controller dispatches the signal to a CPU, the CPU takes a short detour from whatever it was doing, and a handler the driver registered runs for a few microseconds. The handler's job is small: figure out what the device wants, acknowledge the interrupt at the device, do the little bit of work that is safe to do here, and hand the rest off to a thread that has the freedom to block, sleep, or take slow locks. The “hand off” is the second half of modern interrupt discipline; the first half is getting into the handler at all.

Chapter 19's scope is precisely the core interrupt path: what interrupts are at the hardware level, how FreeBSD models them in the kernel, how a driver allocates an IRQ resource and registers a handler through `bus_setup_intr(9)`, how the split between a fast filter handler and a deferred ithread handler works, what the `INTR_MPSAFE` flag commits the driver to, how to simulate interrupts for testing when real events are not easy to produce, how to behave correctly on a shared IRQ line that other drivers also listen to, and how to tear every one of these things down in detach without leaking resources or executing stale handlers. The chapter stops short of MSI and MSI-X, which belong to Chapter 20; those mechanisms build on the core handler the reader writes here, and teaching both at once would dilute both.

Chapter 19 holds back on territory that interrupt work naturally touches. MSI and MSI-X, per-vector handlers, interrupt coalescing, and per-queue interrupt routing are Chapter 20. DMA and the interaction between interrupts and DMA descriptor rings are Chapters 20 and 21. Advanced interrupt affinity strategies on NUMA platforms are discussed briefly but the deep treatment belongs to Chapter 20 and beyond. Platform-specific interrupt routing (GICv3 on arm64, APIC on x86, NVIC on embedded targets) is mentioned only for vocabulary; the book's focus is the driver-visible API that hides those differences. Chapter 19 stays inside the ground it can cover well and hands off explicitly where a topic deserves its own chapter.

The filter-plus-ithread model Chapter 19 teaches does not stand alone. Chapter 16 gave the driver a vocabulary of register access. Chapter 17 taught it to think like a device. Chapter 18 introduced it to a real PCI device. Chapter 19 gives it ears. Chapters 20 and 21 will give it legs: direct memory access so the device can reach into RAM without the driver in the loop. Each chapter adds one layer. Each layer depends on the ones before it. Chapter 19 is where the driver stops polling and starts listening, and the disciplines Part 3 built are what keep the listening honest.

19.1.1 Why Interrupt Handling Earns a Chapter of Its Own

One concern that surfaces here is whether `bus_setup_intr(9)` and the filter-plus-ithread model really justify a full chapter. The Chapter 17 simulation used callouts to produce autonomous state changes; the Chapter 18 driver runs on real PCI but ignores the interrupt line entirely. Could we not just keep polling through callouts and avoid the topic?

Two reasons.

The first is performance. A callout that polls the device ten times a second wastes CPU time when there is nothing to do and misses events that happen between polls. A real device may produce many events per millisecond; a poll interval of 100 milliseconds misses nearly all of them. Interrupts invert the cost: no CPU is spent when nothing is happening, and the handler runs within microseconds of the event. Every serious driver in FreeBSD uses interrupts for the same reason; a driver that polls is a driver with a special excuse.

The second is correctness. Some devices require the driver to respond within a tight time window. A network card's receive FIFO fills in a handful of microseconds; if the driver does not drain it, the card drops packets. A serial port's transmit FIFO empties at the wire's rate; if the driver does not refill it, the transmitter starves. Polling at any interval long enough to be cheap is an interval short enough to miss deadlines. Interrupts are the only mechanism that lets a driver meet real-time device requirements without burning a CPU full time.

The chapter also earns its place by teaching a discipline that applies far beyond PCI. The FreeBSD interrupt model (filter plus ithread, `INTR_MPSAFE`, `bus_setup_intr(9)`, clean teardown in detach) is the same model USB drivers use, the same model SDIO drivers use, the same model virtio drivers use, and the same model arm64 SoC drivers use. A reader who understands Chapter 19's model can read any FreeBSD driver's interrupt handler with comprehension. That generality is what makes the chapter worth reading carefully even for readers who will not work on PCI.

19.1.2 Where Chapter 18 Left the Driver

A short checkpoint before we go further. Chapter 19 extends the driver produced at the end of Chapter 18 Stage 4, tagged as version `1.1-pci`. If any of the items below feels uncertain, return to Chapter 18 before starting this chapter.

- Your driver compiles cleanly and identifies itself as `1.1-pci` in `kldstat -v`.
- On a bhyve or QEMU guest that exposes a virtio-rnd device (vendor `0x1af4`, device `0x1005`), the driver attaches through `myfirst_pci_probe` and `myfirst_pci_attach`, prints its banner, claims BAR 0 as `SYS_RES_MEMORY` with `RF_ACTIVE`, walks the PCI capability list, and creates `/dev/myfirst0`.
- The softc holds the BAR resource pointer (`sc->bar_res`), the resource ID (`sc->bar_rid`), and the `pci_attached` flag.

- The detach path destroys the cdev, quiesces any active callouts and tasks, detaches the hardware layer, releases the BAR, and deinit the softc.
- The full regression script from Chapter 18 passes: attach, exercise cdev, detach, unload, no leaks.
- `HARDWARE.md`, `LOCKING.md`, `SIMULATION.md`, and `PCI.md` are current.
- `INVARIANTS`, `WITNESS`, `WITNESS_SKIPSPIN`, `DDB`, `KDB`, and `KDB_UNATTENDED` are enabled in your test kernel.

That driver is what Chapter 19 extends. The additions are again modest in volume: one new file (`myfirst_intr.c`), one new header (`myfirst_intr.h`), a small set of new softc fields (`irq_res`, `irq_rid`, `intr_cookie`, a counter or two), three new functions in the interrupt file (`setup`, `teardown`, the filter handler), one `sysctl` for simulated interrupts, a version bump to `1.2-intr`, and a short `INTERRUPTS.md` document. The mental-model change is again larger than the line count suggests: the driver finally has two threads of control instead of one, and the discipline that keeps them from stepping on each other is new.

19.1.3 What You Will Learn

By the end of this chapter you will be able to:

- Explain what an interrupt is at the hardware level, the difference between edge-triggered and level-triggered signalling, and how a CPU's interrupt-handling flow gets from the device to a driver's handler.
- Describe how FreeBSD represents interrupt events: what an interrupt event (`intr_event`) is, what an interrupt thread (`ithread`) is, what a filter handler is, and why the split between filters and ithreads matters.
- Read the output of `vmstat -i` and `devinfo -v` and locate the interrupts your system is handling, their counts, and the drivers bound to each.
- Allocate an IRQ resource through `bus_alloc_resource_any(9)` with `SYS_RES_IRQ`, using `rid = 0` on a legacy PCI line and (in Chapter 20) non-zero RIDs for MSI and MSI-X vectors.
- Register an interrupt handler through `bus_setup_intr(9)`, choosing between a filter handler (`driver_filter_t`), an ithread handler (`driver_intr_t`), or the filter-plus-ithread combination, and picking the right `INTR_TYPE_*` flag for the device's class of work.
- Write a minimal filter handler that reads the device's status register, acknowledges the interrupt at the device, returns `FILTER_HANDLED` or `FILTER_STRAY` appropriately, and cooperates with the kernel's interrupt machinery.
- Know what is safe inside a filter (spin locks only, no `malloc`, no sleeping, no blocking locks) and what the ithread relaxes (sleep mutexes, condition variables, `malloc(M_WAITOK)`), and why the constraints exist.
- Set `INTR_MPSAFE` only when you mean it, and understand what the flag commits the driver to (own synchronisation, no implicit Giant acquisition, the right to run on any CPU concurrently).
- Hand deferred work from a filter handler to a taskqueue task or to the ithread, and preserve the discipline that small urgent work happens in the filter while bulk work happens in thread context.
- Simulate interrupts through a `sysctl` that invokes the handler directly under the normal locking rules, so the reader can exercise the handler's state machine without needing a real IRQ to fire.
- Handle shared interrupt lines correctly: read the device's `INTR_STATUS` register first, decide whether this interrupt belongs to our device, return `FILTER_STRAY` if it does not, and avoid stealing another driver's work.
- Tear down the interrupt handler in detach with `bus_teardown_intr(9)` before releasing the IRQ with `bus_release_resource(9)`, and structure the detach path so no interrupt can fire against freed state.

- Recognise what an interrupt storm is, know how FreeBSD's `hw.intr_storm_threshold` machinery detects one, and understand the common device-side causes (failing to clear `INTR_STATUS`, edge-triggered lines mis-configured as level).
- Bind an interrupt to a specific CPU through `bus_bind_intr(9)` when affinity matters, and describe the interrupt to `devinfo -v` through `bus_describe_intr(9)` so operators can see which handler is on which CPU.
- Split the interrupt-related code into its own file, update the module's `SRCS` line, tag the driver as `1.2-intr`, and produce a short `INTERRUPTS.md` document that describes the handler's behaviour and the deferred-work discipline.

The list is long; each item is narrow. The point of the chapter is the composition.

19.1.4 What This Chapter Does Not Cover

Several adjacent topics are explicitly deferred so Chapter 19 stays focused.

- **MSI and MSI-X.** `pci_alloc_msi(9)`, `pci_alloc_msix(9)`, vector allocation, per-vector handlers, and the MSI-X table layout are Chapter 20. Chapter 19 targets the legacy PCI INTx line allocated with `rid = 0`; the vocabulary transfers, but the per-vector mechanics do not.
- **DMA.** `bus_dma(9)` tags, scatter-gather lists, bounce buffers, cache coherence around DMA descriptors, and the way interrupts signal the completion of descriptor-ring transfers are Chapters 20 and 21. Chapter 19's handler reads a BAR register and decides what to do; it does not touch DMA.
- **Per-queue multi-queue networking.** Modern NICs have separate receive and transmit queues with separate MSI-X vectors and separate interrupt handlers. The `iflib(9)` framework builds on this; `em(4)`, `ix(4)`, and `ixl(4)` use it. Chapter 19's driver has one interrupt; Chapter 20 onwards develops the multi-queue story.
- **Deep interrupt affinity on NUMA hardware.** `bus_bind_intr` is introduced; elaborate strategies for pinning interrupts to CPUs near the device's PCIe root port are left for later chapters on scalability.
- **Driver suspend and resume around interrupts.** `bus_suspend_intr(9)` and `bus_resume_intr(9)` exist; they are mentioned for completeness but are not exercised in Chapter 19's driver.
- **Real-time interrupt-priority manipulation.** FreeBSD's `intr_priority(9)` and the `INTR_TYPE_*` flags influence ithread priority, but the book treats the priority system as a black box outside advanced-topics chapters.
- **Software-only interrupts (SWI).** `swi_add(9)` creates a pure software interrupt that a driver can schedule from arbitrary context. The chapter mentions SWIs when discussing deferred work, but the preferred modern pattern (a taskqueue) covers the same use cases with fewer footguns.

Staying inside those lines keeps Chapter 19 a chapter about core interrupt handling. The vocabulary is what transfers; the specific chapters that follow apply the vocabulary to MSI/MSI-X, DMA, and multi-queue designs.

19.1.5 Estimated Time Investment

- **Reading only:** four to five hours. The interrupt model is conceptually small but requires careful reading, particularly around filter vs ithread and the safety rules inside a filter.

- **Reading plus typing the worked examples:** ten to twelve hours over two or three sessions. The driver evolves in four stages; each stage is a small but real extension on top of the Chapter 18 codebase.
- **Reading plus all labs and challenges:** sixteen to twenty hours over four or five sessions, including standing up the bhyve lab (if Chapter 18's setup is not already in place), reading `if_em.c`'s interrupt path and `if_mgb.c`'s filter handler, and running the Chapter 19 regression against both the simulated-interrupt path and (where possible) a real-interrupt path.

Sections 3, 4, and 6 are the densest. If the filter-versus-ithread split feels unfamiliar on first pass, that is normal. Stop, re-read Section 3's decision tree, and continue when the shape has settled.

19.1.6 Prerequisites

Before starting this chapter, confirm:

- Your driver source matches Chapter 18 Stage 4 (1.1-pci). The starting point assumes the Chapter 16 hardware layer, the Chapter 17 simulation backend, the Chapter 18 PCI attach, the complete `CSR_*` accessor family, the sync header, and every primitive introduced in Part 3.
- Your lab machine runs FreeBSD 14.3 with `/usr/src` on disk and matching the running kernel.
- A debug kernel with `INVARIANTS`, `WITNESS`, `WITNESS_SKIPSPIN`, `DDB`, `KDB`, and `KDB_UNATTENDED` is built, installed, and booting cleanly.
- `bhyve(8)` or `qemu-system-x86_64` is available, and Chapter 18's lab environment (a FreeBSD guest with a `virtio-rnd` device at `-s 4:0,virtio-rnd`) is reproducible on demand.
- The `devinfo(8)`, `vmstat(8)`, and `pciconf(8)` tools are in your path. All three live in the base system.

If any item above is shaky, fix it now rather than pushing through Chapter 19 and trying to reason from a moving foundation. Interrupt bugs often manifest as kernel panics or subtle corruption under load; the debug kernel's `WITNESS` in particular catches the common classes of lock mistakes early.

19.1.7 How to Get the Most Out of This Chapter

Four habits will pay off quickly.

First, keep `/usr/src/sys/sys/bus.h` and `/usr/src/sys/kern/kern_intr.c` bookmarked. The first file defines `driver_filter_t`, `driver_intr_t`, `INTR_TYPE_*`, `INTR_MPSAFE`, and the `FILTER_*` return values you will use in every handler. The second file is the kernel's interrupt-event machinery: the code that receives the low-level IRQ, dispatches to filters, wakes ithreads, and detects interrupt storms. You do not need to read `kern_intr.c` in depth, but skimming the top thousand lines once gives you a picture of what happens between “the device asserts IRQ 19” and “your filter gets called”.

Second, run `vmstat -i` on your lab host and your guest, and keep the output open in a terminal as you read. Every concept Section 2 and Section 3 introduce (per-handler counts, per-CPU affinity, interrupt naming conventions) is visible in that output. A reader who has stared at `vmstat -i` for their own machine finds interrupt routing much less abstract.

Third, type the changes by hand and run each stage. Interrupt code is where small mistakes become silent bugs. Forgetting `FILTER_HANDLED` makes your handler technically illegal; forgetting `INTR_MPSAFE` silently acquires Giant around your handler; forgetting to clear `INTR_STATUS` produces an interrupt storm five milliseconds

later. Typing each line, checking the `dmesg` output after each `kldload`, and watching `vmstat -i` between iterations is how you catch these mistakes at the point where they are cheap.

Fourth, read `/usr/src/sys/dev/mgb/if_mgb.c` (look for `mgb_legacy_intr` and `mgb_admin_intr`) after Section 4. `mgb(4)` is the driver for Microchip's LAN743x gigabit Ethernet controllers. Its interrupt path is a clean, readable example of filter-plus-ithread design, and it sits at about the level of complexity Chapter 19 teaches. Seven hundred lines of careful reading pays off across the rest of Part 4.

19.1.8 Roadmap Through the Chapter

The sections in order are:

1. **What Are Interrupts?** The hardware picture: what an interrupt is, edge versus level triggering, the CPU's dispatch flow, and the minimum a driver must do when one arrives. The conceptual foundation.
2. **Interrupts in FreeBSD.** How the kernel represents interrupt events, what an ithread is, how interrupts are counted and displayed through `vmstat -i` and `devinfo -v`, and what happens from the IRQ line to the driver's handler.
3. **Registering an Interrupt Handler.** The code the driver writes: `bus_alloc_resource_any(9)` with `SYS_RES_IRQ`, `bus_setup_intr(9)`, `INTR_TYPE_*` flags, `INTR_MPSAFE`, `bus_describe_intr(9)`. The first stage of the Chapter 19 driver (`1.2-intr-stage1`).
4. **Writing a Real Interrupt Handler.** The filter handler's shape: read `INTR_STATUS`, decide ownership, acknowledge the device, return the right `FILTER_*` value. The ithread handler's shape: take sleep locks, defer to a taskqueue, do the slow work. Stage 2 (`1.2-intr-stage2`).
5. **Using Simulated Interrupts for Testing.** A `sysctl` that invokes the handler synchronously with the real lock discipline, so the reader can exercise the handler without a real IRQ. Stage 3 (`1.2-intr-stage3`).
6. **Handling Shared Interrupts.** Why `RF_SHAREABLE` matters on a legacy PCI line, how a filter handler must decide ownership against other handlers on the same IRQ, and how to avoid starvation. No stage bump; this is a discipline, not a new code artefact.
7. **Cleaning Up Interrupt Resources.** `bus_teardown_intr(9)` first, then `bus_release_resource(9)`. The detach ordering now has two more steps, and the partial-failure cascade gets one more label.
8. **Refactoring and Versioning Your Interrupt-Ready Driver.** The final split into `myfirst_intr.c`, a new `INTERRUPTS.md`, the version bump to `1.2-intr`, and the regression pass. Stage 4.

After the eight sections come hands-on labs, challenge exercises, a troubleshooting reference, a Wrapping Up that closes Chapter 19's story and opens Chapter 20's, and a bridge to Chapter 20. The reference-and-cheat-sheet material at the end of the chapter is meant to be re-read as you work through Chapter 20 and 21; Chapter 19's vocabulary is the foundation both of them build on.

If this is your first pass, read linearly and do the labs in order. If you are revisiting, Sections 3 and 4 stand alone and make good single-sitting reads.

19.2 Section 1: What Are Interrupts?

Before the driver code, the hardware picture. Section 1 teaches what an interrupt is at the level of the CPU and the bus, without any FreeBSD-specific vocabulary. A reader who understands Section 1 can read the rest of the

chapter with the kernel's interrupt path as a concrete object rather than a vague abstraction. The payoff is that every later section is easier.

The one-sentence summary, which you can keep with you through the rest of the chapter: an interrupt is a way for a device to interrupt the CPU's current work, run a driver's handler for a short time, and then let the CPU go back to what it was doing. Everything else is mechanism around that sentence.

19.2.1 The Problem Interrupts Solve

A CPU runs a stream of instructions in order. Each instruction completes, the program counter advances, the next instruction runs, and so on. Left alone, the CPU would execute one program until that program finished, then another, and so on, never paying attention to anything happening outside its own instruction stream.

That is not how computers work. A keyboard pressed for half a second generates four or five separate events; a network packet arrives within a few microseconds of the previous one; a disk finishes a read, the fan controller crosses a temperature threshold, a sensor value updates, a timer expires. Each of these events happens outside the CPU's direct control, at a time the CPU did not choose. The CPU has to notice.

One way to notice is to poll. The CPU can look at the device's status register periodically. If the status register says "I have data", the CPU reads the data. If the status register says "I have nothing", the CPU moves on. Polling works for devices whose events are rare, predictable, and not time-sensitive. It works badly for everything else. A keyboard polled every hundred milliseconds feels sluggish. A network card polled every millisecond still misses most of its packets. And polling consumes CPU time proportional to the poll rate, even when nothing is happening.

The other way to notice is to let the device tell the CPU. That is an interrupt. The device raises a signal on a wire or sends a message over the bus. The CPU interrupts its current work, remembers where it was, runs a small piece of code that asks the device what happened, responds appropriately, and then resumes the work it was doing. The discipline for writing that "small piece of code" is what the rest of Chapter 19 teaches.

19.2.2 What a Hardware Interrupt Actually Is

Physically, a hardware interrupt starts as a signal on a wire (or, more commonly on modern systems, as a message on a bus). A device asserts the signal when something has happened that the OS needs to know about. Examples include:

- A network card asserts its IRQ line when a packet has arrived and is sitting in its receive FIFO.
- A serial UART asserts its IRQ line when a byte has arrived at the receiver, or when the transmit FIFO has fallen below a threshold.
- A SATA controller asserts its IRQ line when a command queue entry has completed.
- A timer chip asserts its IRQ line when a programmed interval has elapsed.
- A temperature sensor asserts its IRQ line when a programmed threshold is crossed.

The assertion is the device's way of saying "something I need you to know about". The CPU and the OS must be prepared to respond. The path from "signal asserted" to "handler called" passes through the interrupt controller, the CPU's interrupt dispatch mechanism, and the kernel's interrupt-event machinery. Section 2 walks the whole path; this subsection stays at the hardware level.

Several useful facts about the signalling itself.

First, **interrupt lines are usually shared**. A CPU has a small number of interrupt inputs, often sixteen to twenty-four on legacy PCs and more on modern platforms through APICs and GICs. A system typically has more devices than interrupt inputs, so multiple devices share a single line. When an interrupt fires on a shared line, each driver whose device might be the source has to check: is this my interrupt? If no, return a “stray” indication; if yes, handle it. Section 6 covers the shared-interrupt protocol.

Second, **interrupt signalling comes in two flavours**. Edge-triggered signalling means the interrupt is signalled by a transition on the wire (low to high, or high to low). Level-triggered signalling means the interrupt is signalled by holding the wire at a specific level (high or low) for as long as the interrupt is pending. The two flavours have different operational consequences, which the next subsection explores.

Third, **the interrupt is asynchronous with respect to the CPU**. The CPU does not know when the device will raise the signal. The driver’s handler must tolerate being called at any point in the driver’s own work, and must synchronise with its own non-interrupt code appropriately. Chapter 11’s locking discipline is what the driver uses to do that.

Fourth, **the interrupt carries essentially no information on its own**. The wire says “something happened”; it does not say what. The driver discovers what happened by reading the device’s status register. A single IRQ line can report many different events (receive data ready, transmit FIFO empty, link state change, error, and so on), and it is the driver’s job to decode the status bits and decide what to do.

19.2.3 Edge-Triggered vs Level-Triggered

The difference is worth understanding because it explains why certain bugs produce interrupt storms, certain bugs produce silently-dropped interrupts, and certain bugs produce stuck systems.

An **edge-triggered** interrupt fires once when the signal transitions. The device pulls the wire low (for an active-low line); the interrupt controller notices the transition; an interrupt is queued for the CPU. If the device continues to hold the wire low, no additional interrupts fire, because the signal is not transitioning, only continuing to be asserted. For a new interrupt to fire, the device must release the wire and then assert it again.

Edge-triggered interrupts are efficient. The interrupt controller only needs to track transitions, not ongoing signals. The downside is fragility: if an interrupt fires while the controller is not watching (because another interrupt is being handled, for example), the transition may be missed. Modern interrupt controllers queue edge-triggered interrupts to avoid most of this, but the risk is real, and some drivers (or some device bugs) produce edge-triggered setups that occasionally drop an event.

A **level-triggered** interrupt fires continuously while the signal is asserted. As long as the device holds the wire at the asserted level, the interrupt controller reports an interrupt. When the device releases the wire, the interrupt controller stops reporting. The CPU sees an interrupt, the driver’s handler runs, the handler reads the device’s status and clears the pending condition, the device stops asserting the signal, and the interrupt controller stops reporting. If the handler fails to clear the pending condition, the signal stays asserted, the interrupt controller keeps reporting, and the driver’s handler gets called again immediately, in a loop that consumes the CPU. This is the classic **interrupt storm**.

Level-triggered interrupts are robust. As long as the device has something to report, the OS will know; there is no window where an event can be missed. The cost is that a buggy driver can produce a storm. FreeBSD has storm detection to mitigate this (the appendix *A Deeper Look at Interrupt Storm Detection* later in the chapter

covers it); other operating systems have similar protections. The common rule of thumb: level-triggered is the safer default, and PCI's legacy INTx lines are level-triggered for that reason.

The distinction matters for driver authors in a few specific places:

- A driver that fails to clear the device's INTR_STATUS register before returning from the handler will produce an interrupt storm on a level-triggered line. On an edge-triggered line the same bug produces a lost interrupt instead.
- A driver that reads and writes INTR_STATUS correctly works on both types without special knowledge.
- A driver that manipulates the interrupt controller's trigger mode directly (rare; mostly legacy) must understand the distinction.

For Chapter 19's PCI driver the signalling is level-triggered INTx on the legacy path. On MSI and MSI-X (Chapter 20) the signalling is message-based and does not correspond directly to either edge or level, but the driver pattern is the same: read status, acknowledge the device, return.

19.2.4 The CPU's Interrupt Handling Flow, Simplified

What happens, step by step, when a device's IRQ line is asserted? A simplified trace on a modern x86 system:

1. The device asserts its IRQ on the bus (or sends an MSI packet, for PCIe with MSI enabled).
2. The system's interrupt controller (APIC on x86, GIC on arm64) receives the signal and determines which CPU should handle it, based on the configured affinity. On multi-CPU systems this is a steerable decision.
3. The chosen CPU's interrupt hardware detects the pending interrupt. Before completing the current instruction, the CPU saves enough state (the program counter, the flags register, and a few other fields) to return to the interrupted work later.
4. The CPU jumps to a vector in its interrupt descriptor table. The entry for this vector is a small piece of kernel code called a **trap stub** that transitions to supervisor mode, saves the interrupted thread's register set, and calls into the kernel's interrupt dispatch code.
5. The kernel's interrupt dispatch code finds the `intr_event` associated with the IRQ (this is the FreeBSD structure Section 2 covers) and calls the driver handlers attached to it.
6. The driver's filter handler runs. It reads the device's status register, decides what kind of event has occurred, writes the device's INTR_STATUS register to acknowledge the event (so the device stops asserting the line, for level-triggered), and returns a value that tells the kernel what to do next.
7. If the filter returned `FILTER_SCHEDULE_THREAD`, the kernel schedules the `ithread` associated with this interrupt. The `ithread` is a kernel thread that wakes up, runs the driver's secondary handler, and goes back to sleep.
8. After all handlers have run, the kernel sends an End-of-Interrupt (EOI) signal to the interrupt controller, which re-arms the IRQ line.
9. The CPU returns from the interrupt. The interrupted thread's register set is restored and the thread resumes at the instruction that was about to execute when the interrupt arrived.

Steps 3 through 9 take a few microseconds on modern hardware for a simple handler. The whole flow is invisible to the interrupted thread: code that was not designed to be interrupt-safe (say, a floating-point computation in user space) runs correctly on either side of the interrupt, because the CPU saves and restores its state around the whole sequence.

From the driver author's perspective, steps 1 through 5 are the kernel's concern; steps 6 and 7 are where the driver code runs. The driver's handler must be fast (step 8's EOI waits for it), must not sleep (the interrupted thread holds CPU resources), and must not take locks that could block indirectly on the interrupted thread. Section 2 will make these constraints precise in FreeBSD terms; Section 1 has now set up the mental model.

19.2.5 What a Driver Must Do When an Interrupt Arrives

The driver's obligations on an interrupt are small in number but not small in detail:

1. **Identify the cause.** Read the device's interrupt status register. If no bit is set (the device has no pending interrupt), this is a shared-IRQ spurious call; return `FILTER_STRAY` and let the kernel try the next handler on the line.
2. **Acknowledge the interrupt at the device.** Write the status bits back (typically by writing a 1 to each bit, since most `INTR_STATUS` registers are RW1C) so the device deasserts the line and the interrupt controller can re-enable the IRQ. Not every device requires the acknowledgment to be inside the filter, but doing it here is the safe default; the level-triggered storm story depends on timely acknowledgment.
3. **Decide what work to do.** Read enough of the device to decide. Was this a receive event? A transmit completion? An error? A link change? The status bits tell you.
4. **Do the urgent small work.** Update a counter. Copy a byte from a FIFO into a queue. Toggle a control bit. Whatever can be done in microseconds without taking a sleep lock is fair game here.
5. **Defer the bulk work.** If the event triggers a long operation (processing a received packet, decoding a data stream, sending a command to user space), schedule an ithread or a taskqueue task and return. The deferred work runs in thread context, where it can take sleep locks, allocate memory, and take its time.
6. **Return the appropriate `FILTER_*` value.** `FILTER_HANDLED` means the interrupt is entirely handled; no ithread is needed. `FILTER_SCHEDULE_THREAD` means the ithread should run. `FILTER_STRAY` means the interrupt was not for this driver. These three values are the vocabulary the kernel uses to dispatch further work.

A driver that does these six things correctly on every interrupt has the shape the rest of Chapter 19 teaches. A driver that skips any one of them has a bug.

19.2.6 Real-World Examples

A short tour of the events Chapter 19's vocabulary will cover.

Keypresses. A PS/2 keyboard controller fires an interrupt when a scancode arrives. The driver reads the scan-code, passes it to the keyboard subsystem, and acknowledges. The whole handler runs in a few microseconds; a taskqueue is usually unnecessary.

Network packets. A NIC fires an interrupt when packets accumulate in the receive queue. The driver's filter reads a status register to confirm receive events, schedules the ithread, and returns. The ithread walks the descriptor ring, constructs mbuf packets, and passes them up the network stack. The split between filter and ithread matters here because stack processing is slow enough that running it in the filter would extend the interrupt window too far.

Sensor readings. An I2C-attached temperature sensor fires an interrupt when a new measurement is ready. The driver reads the value, updates a sysctl cache, optionally wakes up any pending user-space readers, and acknowledges. Simple and fast.

Serial port. A UART fires an interrupt on receive or transmit-FIFO-empty conditions. The driver drains or refills the FIFO, updates a circular buffer, and acknowledges. At high baud rates this can happen tens of thousands of times per second, so the handler must be tight.

Disk completion. A SATA or NVMe controller fires an interrupt when a queued command completes. The driver walks the completion queue, matches each completion to a pending I/O request, wakes the waiting thread, and acknowledges. The matching and waking is sometimes split between filter and itthread.

Each of these devices reaches Chapter 19's vocabulary in the same way: the filter reads status, the filter decides what happened, the filter acknowledges, and the filter either handles or defers. The specific register layout differs; the pattern does not.

19.2.7 A Quick Exercise: Find Interrupt-Driven Devices on Your Lab Host

Before moving to Section 2, a short exercise to make the hardware picture concrete.

On your lab host, run:

```
vmstat -i
```

The output is a list of interrupt sources with their counts since boot. Each line looks roughly like this:

interrupt	total	rate
cpu0:timer	1234567	123
cpu1:timer	1234568	123
irq9: acpi0	42	0
irq19: uhci0+	12345	12
irq21: ahci0	98765	99
irq23: em0	123456	123

Pick three lines from your own output. For each, identify:

- The interrupt name (a mix of the IRQ number and the driver-set description).
- The total count (how many times the interrupt has fired since boot).
- The rate (interrupts per second; a high rate means the device is busy).

Run `vmstat -i` a second time ten seconds later. Compare the counts. Which interrupts are actively counting? Which ones are essentially idle?

Now match interrupts to devices with `devinfo -v`:

```
devinfo -v | grep -B 2 irq
```

Each match shows a device that claims an IRQ. Cross-check against `vmstat -i`'s output to see which driver is served by each line.

Keep this output open while you read Section 2. The `em0` entry in the example is the Intel Ethernet controller; if you are using an Intel-based system with FreeBSD, `em0` or `igc0` or `ix0` is probably running a version of the same pattern Chapter 19 teaches. A modern NUC running FreeBSD 14.3 shows one or two dozen interrupt sources; a server shows many more. The system you actually have is more interesting to stare at than any diagram.

19.2.8 A Brief History of Interrupts

Interrupts are one of the oldest ideas in computer architecture. The original PDP-1 supported them in 1961 as a way for I/O devices to signal the CPU without the CPU polling. The IBM 704 had them at around the same time. Early timesharing systems used interrupts for the clock tick that drove scheduling, and for every I/O completion.

Through the 1970s and 1980s, personal computers inherited the pattern. The original IBM PC used the 8259 Programmable Interrupt Controller (PIC), which supported eight IRQ lines; the PC/AT extended this to fifteen usable lines by cascading two PICs. The x86 instruction set added specific interrupt-handling instructions (`CLI`, `STI`, `INT`, `IRET`) that persist today in extended forms.

PCI introduced the concept of a device advertising its interrupt through configuration space (the `INTLINE` and `INTPIN` fields Chapter 18 discussed). PCIe added MSI and MSI-X, which replace the physical IRQ line with a memory-write message. All three coexist in modern systems; Chapter 19's legacy `INTx` is the oldest of the three and the only one that shares lines.

Operating systems have evolved alongside. Early Unix was monolithic and single-threaded in the kernel; interrupts preempted whatever was running. Modern kernels (FreeBSD included) have fine-grained locking, per-CPU data structures, and ithread-based deferred dispatch. The handler discipline Chapter 19 teaches is the distillation of that evolution: fast in the filter, slow in the task, MP-safe by default, shareable, debuggable.

Knowing the history is not required for writing a driver. But the vocabulary (IRQ, PIC, EOI, `INTx`) comes from specific points in the history, and a driver author who knows where the words come from finds less of the field mysterious.

19.2.9 Wrapping Up Section 1

An interrupt is a way for a device to interrupt the CPU's current work, run a small piece of driver code, and let the CPU resume. The mechanism runs through the interrupt controller, the CPU's interrupt dispatch, the kernel's interrupt-event machinery, and finally the driver's handler. Edge-triggered and level-triggered signalling have different operational consequences, most visibly in the form of interrupt storms when a level-triggered line is not properly acknowledged.

A driver's handler has six obligations: identify the cause, acknowledge the device, decide the work, do the urgent part, defer the bulk part, and return the right `FILTER_*` value. Each of these is small in isolation and demanding in aggregate; the rest of Chapter 19 is about doing each one correctly in FreeBSD terms.

Section 2 now walks the FreeBSD kernel's interrupt model: what an `intr_event` is, what an ithread is, how `vmstat -i` and `devinfo -v` expose the kernel's view of interrupts, and what constraints the model places on driver handlers.

19.3 Section 2: Interrupts in FreeBSD

Section 1 established the hardware model. Section 2 introduces the software model. The kernel's interrupt machinery is the layer between the interrupt controller and the driver's handler; understanding it clearly is what turns the hardware model into a driver-writable concept. A reader who finishes Section 2 should be able to

answer three questions in plain English: what runs when an interrupt fires, what runs later as deferred work, and what the driver has to promise so both can happen safely.

19.3.1 The FreeBSD Interrupt Model in One Picture

A driver's handler does not run in isolation. It runs inside a small ecosystem of kernel objects that together make interrupt handling orderly and debuggable. The ecosystem has three pieces worth naming up front.

The first is the **interrupt event**, represented by struct `intr_event` in `/usr/src/sys/sys/interrupt.h` (with the dispatch code in `/usr/src/sys/kern/kern_intr.c`). One `intr_event` exists per IRQ line (or per MSI vector, in Chapter 20's world). It is the central coordinator: it holds a list of handlers (the driver's filter functions and `ithread` functions), a human-readable name, flags, a loop counter used for storm detection (`ie_count`), a rate-limiter for warning messages (`ie_warntm`), and a CPU binding. When the interrupt controller reports an IRQ to the kernel, the kernel looks up the corresponding `intr_event` and walks its list of handlers. Stray interrupts are counted globally, not per-event; they surface through `vmstat -i`'s separate accounting and through kernel log messages rather than through a field on the event.

The second is the **interrupt handler**, represented by struct `intr_handler`. One `intr_handler` exists per registered handler on an `intr_event`. A single IRQ line can have many handlers (one per driver that shares the line). The handler carries the driver-provided filter function (if any), the driver-provided `ithread` function (if any), the `INTR_*` flags (most importantly `INTR_MPSAFE`), and a cookie pointer the kernel keeps for the driver's reference.

The third is the **interrupt thread**, usually called an **ithread**, represented by struct `intr_thread`. Unlike the event and the handler, the `ithread` is a real kernel thread with its own stack, its own scheduling priority, and its own `proc` structure. When a filter returns `FILTER_SCHEDULE_THREAD` (or when a driver registered an `ithread`-only handler with no filter), the `ithread` is scheduled to run. The `ithread` then calls the driver's handler function in thread context, where regular sleep mutexes and sleeping are permitted.

The three together produce the classic two-phase interrupt pattern FreeBSD has used for more than a decade: a fast filter runs in primary interrupt context to do the urgent work, and a thread-context handler runs later to do the slow work. Chapter 19's driver will use both phases.

19.3.2 IRQ Assignment and Routing

A modern x86 system has more than one interrupt controller. The legacy 8259 PIC has been superseded by the Local APIC (one per CPU, handling per-CPU interrupts like the local timer) and the I/O APIC (a shared unit that receives IRQs from the I/O fabric and routes them to CPUs). On arm64, the equivalent is the Generic Interrupt Controller (GIC), with per-CPU redistributors and a shared distributor. On embedded targets there are a handful of other controllers. FreeBSD abstracts these behind the `intr_pic(9)` interface; a driver author rarely interacts with the interrupt controller directly.

What the driver sees is an IRQ number (on legacy PCI paths, the number a BIOS assigned in configuration space) or a vector index (on MSI and MSI-X paths). The driver asks for an IRQ resource by that number, the kernel allocates a struct `resource *`, and the driver hands the resource to `bus_setup_intr(9)` to attach a handler. The kernel does the work of wiring the handler to the right `intr_event`, configuring the interrupt controller to route the IRQ to a CPU, and arming the line.

From a driver author's perspective, IRQ routing is usually a black box. The kernel handles it; the driver sees a handle and a handler. One exception: on platforms with many CPUs and multiple devices, **affinity** matters. An interrupt that fires on a CPU far from the device produces cache misses and cross-socket traffic; an interrupt that fires on a CPU close to the device is cheaper. `bus_bind_intr(9)` lets the driver ask for a specific CPU; operators use `cpuset -x <irq> -l <cpu>` to override the affinity at runtime, and `cpuset -g -x <irq>` to query it. The appendix *A Deeper Look at CPU Affinity for Interrupts* later in the chapter covers both paths in more detail.

19.3.3 SYS_RES_IRQ: The Interrupt Resource

Chapter 18 introduced three resource types: `SYS_RES_MEMORY` for memory-mapped BARs, `SYS_RES_IOPORT` for I/O port BARs, and (mentioned in passing) `SYS_RES_IRQ` for interrupts. Section 3 will use the third one for the first time. The vocabulary is the same as for BARs:

```
int rid = 0;                /* legacy PCI INTx */
struct resource *irq_res;

irq_res = bus_alloc_resource_any(dev, SYS_RES_IRQ, &rid,
    RF_SHAREABLE | RF_ACTIVE);
```

Three things are worth calling out about this allocation.

First, `rid = 0` is the convention for a legacy PCI INTx line. The PCI bus driver treats the zeroth IRQ resource as the device's legacy interrupt, set up from the `PCIR_INTLINE` field in configuration space. For MSI and MSI-X (Chapter 20), the rid is 1, 2, 3, and so on, corresponding to allocated vectors.

Second, `RF_SHAREABLE` asks the kernel to allow the IRQ line to be shared with other drivers. On legacy PCI this is the common case: one physical line can serve multiple devices. Without `RF_SHAREABLE`, the allocation fails if another driver already holds a handler on the same line. Passing `RF_SHAREABLE` does not mean your driver must handle stray interrupts; it means it must tolerate them. Section 6 is about exactly that tolerance.

Third, `RF_ACTIVE` activates the resource in one step, as with BAR allocation. Without it, the driver would need to call `bus_activate_resource(9)` separately. Chapter 19 always uses `RF_ACTIVE`.

On success, the returned `struct resource *` is a handle to the IRQ. It is not the IRQ number; the kernel does not expose that. The driver passes the handle to `bus_setup_intr(9)` and `bus_tearardown_intr(9)` and `bus_release_resource(9)`.

19.3.4 Filter Handlers vs ithread Handlers

This is the conceptual heart of Section 2 and of the chapter. A reader who internalises the filter-versus-ithread distinction reads every FreeBSD driver's interrupt code with comprehension.

A **filter handler** is a C function registered by the driver that runs in primary interrupt context. Primary interrupt context means: the CPU jumped directly from whatever it was doing, minimal state was saved, and the filter is running with the interrupted thread's context still partially in place. Specifically:

- The filter cannot sleep. There is no thread to block; the kernel is in the middle of dispatching an interrupt.

- The filter cannot acquire a sleep mutex (`mtx(9)` by default is a sleep-ish adaptive mutex, which can spin briefly but will eventually sleep). Spin mutexes (`mtx_init` with `MTX_SPIN`) are safe.
- The filter cannot allocate memory with `M_WAITOK`; it can use `M_NOWAIT`, which may fail.
- The filter cannot call into code that uses any of the above.

The filter is supposed to be fast (microseconds), do the urgent work (read status, acknowledge, update a counter), and return. The kernel's return value convention is:

- `FILTER_HANDLED`: the interrupt is mine, it is fully handled, no `ithread` is needed.
- `FILTER_SCHEDULE_THREAD`: the interrupt is mine, some of the work is done, schedule the `ithread` for the rest.
- `FILTER_STRAY`: the interrupt is not mine; try the next handler on this line.

A driver can also specify `FILTER_HANDLED | FILTER_SCHEDULE_THREAD` to say both “I handled part of it” and “schedule the thread for more”.

An **ithread handler** is a different C function that runs in thread context. The thread is scheduled by the kernel after a filter returns `FILTER_SCHEDULE_THREAD`, or if the handler is registered as `ithread-only` (with no filter), the `ithread` is scheduled automatically when the kernel dispatches the interrupt.

In `ithread` context, the constraints loosen considerably:

- The `ithread` can sleep briefly on a mutex or condition variable.
- The `ithread` can use `malloc(M_WAITOK)`.
- The `ithread` can call into most kernel APIs that use sleepable locks.
- The `ithread` cannot still sleep arbitrarily long (it is a real-time-ish thread), but it can do normal driver work.

The split lets a driver separate urgent, short work (in the filter) from slower, bulk work (in the `ithread`). A network driver's filter might read the status register and acknowledge; its `ithread` walks the receive descriptor ring and passes packets up the stack. A disk driver's filter might record which completions happened and acknowledge; its `ithread` matches completions to pending requests and wakes waiting threads.

19.3.5 When to Use Filter Only

A driver uses filter-only when every piece of work it does on an interrupt can run in primary interrupt context. Examples:

- A **minimal test driver** that increments a counter on each interrupt and nothing more.
- A **simple sensor driver** that reads one register, caches the value, and wakes a `sysctl` reader. (If `se_lwakeup` or a condition-variable broadcast requires a sleep mutex, this moves to filter-plus-`ithread`.)
- A **timer driver** whose job is to tick some kernel-internal counter.

The Chapter 19 driver at Stage 1 is filter-only: it reads `INTR_STATUS`, acknowledges, updates a counter, and returns `FILTER_HANDLED`. That is enough to prove the handler is wired up.

19.3.6 When to Use Filter Plus `ithread`

A driver uses filter-plus-`ithread` when the interrupt requires urgent small work followed by slower bulk work. Examples:

- **A NIC driver.** The filter acknowledges and marks which queues have events. The ithread walks the descriptor rings, builds mbufs, and passes packets up.
- **A disk controller.** The filter reads the completion status and acknowledges. The ithread matches completions to I/O requests and wakes waiters.
- **A USB host controller.** The filter reads the status and acknowledges. The ithread walks the transfer descriptor list and completes any pending URBs.

The Chapter 19 driver at Stage 2 moves to filter-plus-ithread when simulated “work-requested” events are added; the filter records the event, and a deferred worker (via a taskqueue; Chapter 14’s primitive) handles the work.

19.3.7 When to Use ithread Only

A driver uses ithread-only when every piece of work must run in thread context. This is less common; the usual reason is that the driver needs to take a sleep mutex on every interrupt and cannot usefully do anything in primary context.

Registering an ithread-only handler is simple: pass `NULL` for the filter argument of `bus_setup_intr(9)`. The kernel schedules the ithread whenever the interrupt fires.

The Chapter 19 driver does not use ithread-only; the filter is always cheap to run.

19.3.8 INTR_MPSAFE: What the Flag Promises

`INTR_MPSAFE` is a bit in the flags argument of `bus_setup_intr(9)`. Setting it promises the kernel two things:

1. Your handler does its own synchronisation. The kernel will not acquire the Giant lock around it.
2. Your handler is safe to run concurrently on multiple CPUs (for a handler shared by multiple CPUs, which happens in MSI-X scenarios and in some PIC configurations).

If you do **not** set `INTR_MPSAFE`, the kernel acquires Giant before calling your handler. This is the old BSD default, preserved for backward compatibility with pre-SMP drivers that relied on Giant’s implicit protection. Modern drivers always set `INTR_MPSAFE`.

Failing to set `INTR_MPSAFE` has a visible symptom: the `dmesg` banner on `kldload` includes a line like `myfirst0: [GIANT-LOCKED]`. This is the kernel telling you that Giant is being acquired around your handler. On production systems it serialises every interrupt through a single lock, which is disastrous for scalability. The line is a deliberate nag from `bus_setup_intr` to help you notice.

Setting `INTR_MPSAFE` when you actually still rely on Giant is also a bug, but a quieter one. The kernel will not acquire Giant, so any code path that used to be serialised by Giant will no longer be. Race conditions appear where none existed before. The fix is not to remove `INTR_MPSAFE` (that masks the bug); the fix is to add the right locking to the handler and the code it touches.

Chapter 19’s driver always sets `INTR_MPSAFE` and relies on the existing `sc->mtx` for synchronisation. The Chapter 11 discipline carries through.

19.3.9 INTR_TYPE_* Flags

In addition to `INTR_MPSAFE`, `bus_setup_intr` takes a category flag that hints at the interrupt’s class:

- `INTR_TYPE_TTY`: tty and serial devices.
- `INTR_TYPE_BIO`: block I/O (disk, CD-ROM).
- `INTR_TYPE_NET`: network.
- `INTR_TYPE_CAM`: SCSI (CAM framework).
- `INTR_TYPE_MISC`: miscellaneous.
- `INTR_TYPE_CLK`: clock and timer interrupts.
- `INTR_TYPE_AV`: audio and video.

The category influences the ithread's scheduling priority. Historically, each category had a distinct priority; in modern FreeBSD, only `INTR_TYPE_CLK` gets an elevated priority, and the rest are approximately equal. The category is still worth setting correctly because it flows through `devinfo -v` and `vmstat -i` output, making the interrupt self-identifying.

For the Chapter 19 driver, `INTR_TYPE_MISC` is appropriate because the demo target does not fit any of the more specific categories. Chapter 20 will use `INTR_TYPE_NET` once the driver starts targeting NICs in labs.

19.3.10 Shared vs Exclusive Interrupts

On legacy PCI, multiple devices can share a single INTx line. The kernel tracks this with two resource flags:

- `RF_SHAREABLE`: this driver is willing to share the line with other drivers.
- `RF_SHAREABLE` absent: this driver wants the line for itself; allocation fails if another driver already holds it.

A driver that wants shareable interrupts uses `RF_SHAREABLE | RF_ACTIVE` in its `bus_alloc_resource_any` call. A driver that wants exclusive access (perhaps for latency reasons) uses `RF_ACTIVE` alone, but the request may fail on crowded systems.

The kernel never stops the driver from sharing; it stops another driver from joining if this one asks for exclusivity. On modern PCIe with MSI-X, sharing is much less common because each device has its own message-signalled vector.

Chapter 19's driver sets `RF_SHAREABLE` because `virtio-rnd` in `bhyve` may or may not share its line with other `bhyve`-emulated devices, depending on the slot topology. Being shareable is the safe default.

The `INTR_EXCL` flag passed through `bus_setup_intr`'s `flags` field (not to be confused with the resource-allocation flag) is a related but distinct concept: it asks the bus to give the handler exclusive access at the interrupt-event level. Legacy PCI drivers rarely need it. Some bus drivers use it internally. For Chapter 19's driver, we do not set `INTR_EXCL`.

19.3.11 What `vmstat -i` Shows

`vmstat -i` prints the kernel's interrupt counters. Each line corresponds to one `intr_event`. The columns are:

- **interrupt**: a human-readable identifier. For hardware interrupts, the name is derived from the IRQ number and the driver description. `devinfo -v`-style names (such as `em0:rx 0`) appear when MSI-X vectors are in use.
- **total**: the number of times this interrupt has fired since boot.

- **rate**: the interrupt rate in interrupts per second, averaged over a recent window.

A few interpretation notes. A total column that grows quickly for an idle device is a red flag (interrupt storm). A rate column at zero for a device that should be handling traffic suggests the handler is not wired up correctly. When several devices share a legacy INTx line, `vmstat -i` shows one line per `intr_event` (per IRQ source), and the driver name on that line is the description of the first registered handler; the other drivers sharing the line do not get their own lines. When a device has its own MSI or MSI-X vectors, each vector is its own `intr_event`, and each gets its own line. Per-CPU interrupts such as the local timer appear as distinct per-CPU lines (`cpu0:timer`, `cpu1:timer`) because the kernel creates one event per CPU for them.

The kernel exposes the same counters through `sysctl hw.intrcnt` and `sysctl hw.intrnames`, which are the raw data `vmstat -i` formats. A driver author rarely reads these directly; `vmstat -i` is the friendly view.

19.3.12 What devinfo -v Shows About Interrupts

`devinfo -v` walks the newbus tree and prints each device with its resources. For a PCI driver with an interrupt, the resources list includes an `irq:` entry next to the `memory:` entry:

```
myfirst0
  pnpinfo vendor=0x1af4 device=0x1005 ...
  resources:
    memory: 0xc1000000-0xc100001f
    irq: 19
```

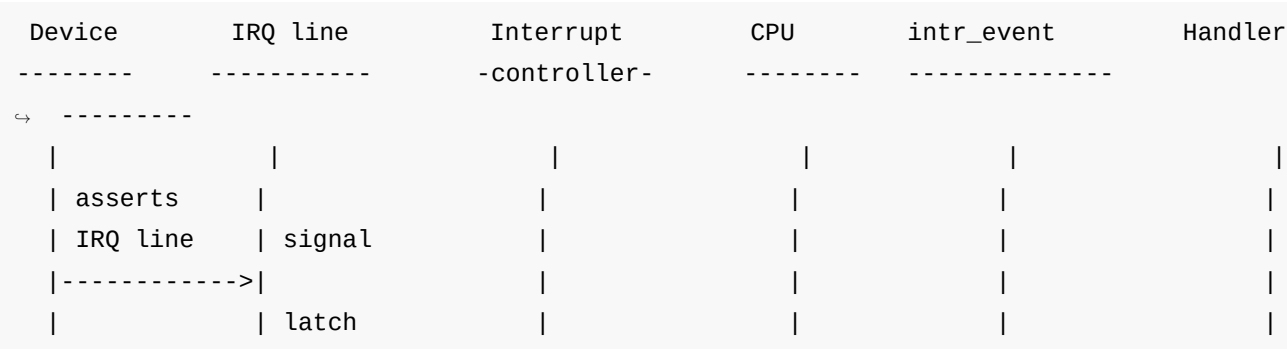
The number after `irq:` is the kernel’s IRQ identifier. On x86 it is often a pin number from the I/O APIC; on arm64 it is a GIC vector; the exact meaning is platform-specific but the number is stable across reboots of the same system.

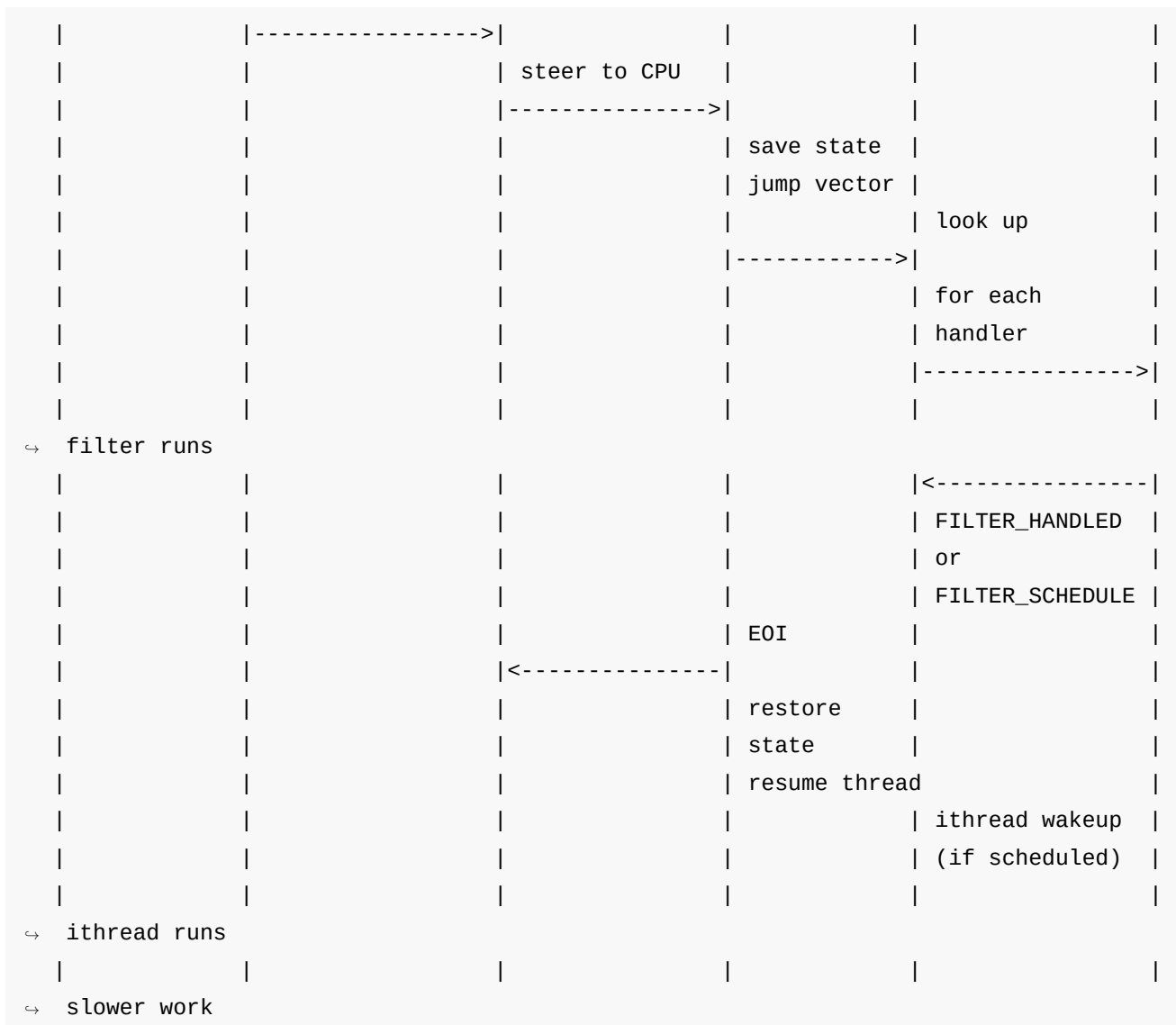
Matching the `irq: 19` to `vmstat -i`’s `irq19:` entry confirms the driver is attached to the expected interrupt line.

For MSI-X interrupts (Chapter 20), each vector has its own `irq:` entry, and `devinfo -v` lists them individually.

19.3.13 A Simple Interrupt Path Diagram

Putting it all together, here is what happens from device to driver:





The diagram elides several details (interrupt coalescing, the swapping of the interrupted thread's stack for the ithread's stack, the ithread's own scheduling), but it captures the shape. A filter runs in interrupt context, an ithread (if scheduled) runs later in thread context, the EOI happens after filters are done, and the interrupted thread resumes once the CPU is free.

19.3.14 Constraints on What Handlers May Do

A short consolidated list of what a filter handler may and may not do. This is the most-referenced list in Chapter 19; mark it for return visits.

Filter handlers MAY:

- Read and write device registers through the accessor layer.
- Acquire spin mutexes (`struct mtx` initialised with `MTX_SPIN`).
- Read softc fields that are protected only by spin locks.
- Call the kernel's atomic operations (`atomic_add_int`, etc.).
- Call `taskqueue_enqueue(9)` to schedule thread-context work.

- Call `wakeup_one(9)` to wake a thread sleeping on a channel, if the context permits (most do).
- Return `FILTER_HANDLED`, `FILTER_SCHEDULE_THREAD`, `FILTER_STRAY`, or combinations.

Filter handlers MAY NOT:

- Acquire a sleep mutex (`struct mtx` initialised default, `struct sx`, `struct rwlock`).
- Call any function that may sleep: `malloc(M_WAITOK)`, `tsleep`, `pause`, `cv_wait`, and so on.
- Acquire Giant.
- Call code that may indirectly do any of the above.
- Take a long time (microseconds are fine; milliseconds are a bug).

Ithread handlers MAY:

- Everything a filter handler may do, plus:
- Acquire sleep mutexes, `sx` locks, `rwlocks`.
- Call `malloc(M_WAITOK)`.
- Call `cv_wait`, `tsleep`, `pause`, and other blocking primitives.
- Take longer (tens or hundreds of microseconds are normal).
- Do bounded work with unpredictable completion time.

Ithread handlers SHOULD NOT:

- Sleep for arbitrarily long times. The `ithread` has a scheduling priority that assumes responsiveness; a handler that sleeps for seconds starves other work on the same `ithread`.
- Block the `ithread` waiting for external events that have no bound.

Chapter 19's filter respects the first list rigorously; any violation is a bug the debug kernel will often catch.

19.3.15 A Note on Per-CPU vs Shared `ithreads`

For legacy PCI INTx lines, the kernel usually assigns one `ithread` per `intr_event`, shared among any handlers on that event. For MSI-X (Chapter 20), each vector has its own `ithread`. The difference matters when multiple handlers need to run concurrently on the same IRQ: on the shared `ithread`, they serialise; on separate MSI-X vectors, they can run in parallel.

Chapter 19's driver uses legacy PCI. One IRQ, one `ithread` (if any `ithread` at all), one queue of deferred work. The serialisation is usually what you want for a single-device driver.

19.3.16 Wrapping Up Section 2

FreeBSD's interrupt model centres on three objects: the `intr_event` (one per IRQ line or MSI vector), the `intr_handler` (one per registered driver on that event), and the `ithread` (one per event, shared among handlers). A driver registers a filter function, an `ithread` function, or both, through `bus_setup_intr(9)`, and promises `INTR_MPSAFE` compliance via the `flags` argument. The kernel dispatches filter handlers in primary interrupt context and schedules `ithreads` after the filter returns.

The constraints on filter handlers are strict (no sleeping, no sleep locks, no slow calls); the constraints on `ithread` handlers are loose by comparison. Shared PCI INTx lines allow many drivers on one IRQ, so filters must

identify whether the interrupt belongs to their device and return `FILTER_STRAY` when it does not. `vmstat -i` and `devinfo -v` expose the kernel's view so operators and driver authors can see what is happening.

Section 3 is where the driver finally writes code against this model. It allocates an IRQ resource with `SYS_RES_IRQ`, registers a filter handler through `bus_setup_intr(9)`, sets `INTR_MPSAFE`, and logs a short message on every call. Stage 1 is the first time the driver gets interrupted in anger.

19.4 Section 3: Registering an Interrupt Handler

Sections 1 and 2 established the hardware and kernel models. Section 3 puts the driver to work. The task is narrow: extend Chapter 18's attach path so that after allocating the BAR and walking the capability list, it also allocates an IRQ resource, registers a filter handler, and sets `INTR_MPSAFE`. The detach path grows in the mirror direction: tear down the handler first, then release the IRQ. At the end of Section 3 the driver is at version `1.2-intr-stage1` and fires a small counter-bumping filter handler every time its IRQ line is asserted.

19.4.1 What Stage 1 Produces

Stage 1's handler is deliberately minimal. The driver needs a filter that is correct in every formal sense (returns the right `FILTER_*` value, respects the “no sleeping” rule, is `INTR_MPSAFE`) but does not yet do any real work. The goal is to prove the handler is wired correctly before introducing the complications of status decoding and deferred work.

The handler's behaviour at Stage 1:

1. Acquire a spin-safe counter lock (in our case a simple atomic operation).
2. Increment a counter in the softc.
3. Return `FILTER_HANDLED`.

That is it. No status register read, no acknowledgment, no deferred work. The counter lets the reader observe whether the handler fires, and if so how often. The `dmesg` output is silent by default; the count is visible through a `sysctl` the stage exposes.

Section 4 adds the real work (status decoding, acknowledgment, `ithread` scheduling). Section 5 adds simulated interrupts for testing. Section 6 extends the filter to be shared-IRQ-safe by checking whether the interrupt is actually for our device. But the scaffolding is this stage's contribution, and it needs to be right before anything else lands on top.

19.4.2 The IRQ Resource Allocation

The first new line of attach, right after the BAR allocation:

```
sc->irq_rid = 0; /* legacy PCI INTx */
sc->irq_res = bus_alloc_resource_any(dev, SYS_RES_IRQ, &sc->irq_rid,
    RF_SHAREABLE | RF_ACTIVE);
if (sc->irq_res == NULL) {
    device_printf(dev, "cannot allocate IRQ\n");
```

```

    error = ENXIO;
    goto fail_hw;
}

```

A few things worth noticing.

First, `rid = 0` is the legacy PCI INTx convention. Every PCI device has a single legacy IRQ line advertised through the `PCIR_INTLINE` and `PCIR_INTPIN` fields of configuration space; the PCI bus driver exposes this as resource `rid 0`. Chapter 20 will use non-zero `rids` for MSI and MSI-X vectors, but Chapter 19's driver uses the legacy path.

Second, the `rid` variable is updated by `bus_alloc_resource_any` if the kernel chose a different `rid` than requested. For `rid = 0` the kernel always returns `rid = 0`, so the update is a no-op, but the pattern is consistent with BAR allocation from Chapter 18.

Third, `RF_SHAREABLE | RF_ACTIVE` is the standard flag set. `RF_SHAREABLE` allows the kernel to put our handler on a shared `intr_event` with other drivers. `RF_ACTIVE` activates the resource in one step.

Fourth, the allocation can fail. The most common reason on a real system is that the device's PCI configuration-space interrupt fields are zero (the firmware did not route an interrupt to the device). On bhyve with a `virtio-rnd` device, the allocation normally succeeds; on some older QEMU configurations with `intx=off`, it can fail. If the allocation fails, the attach path unwinds through the `goto` cascade.

19.4.3 Storing the Resource in the Softc

The `softc` gains three new fields:

```

struct myfirst_softc {
    /* ... existing fields ... */

    /* Chapter 19 interrupt fields. */
    struct resource *irq_res;
    int          irq_rid;
    void         *intr_cookie;    /* for bus_teardown_intr */
    uint64_t     intr_count;      /* handler invocation count */
};

```

`irq_res` is the handle to the claimed IRQ resource. `irq_rid` is the resource ID (for the matching release call). `intr_cookie` is the opaque cookie that `bus_setup_intr(9)` returns and `bus_teardown_intr(9)` consumes; it identifies the specific handler so the kernel can remove it cleanly later. `intr_count` is a diagnostic counter the Stage 1 handler increments on every call.

The three fields parallel the three BAR fields (`bar_res`, `bar_rid`, `pci_attached`) added in Chapter 18. The parallelism is not accidental: every resource class gets a handle, an ID, and whatever bookkeeping the driver needs.

19.4.4 The Filter Handler's Signature

The driver's filter is a function with signature:

```
static int myfirst_intr_filter(void *arg);
```

The argument is whatever pointer the driver passes to `bus_setup_intr`'s `arg` parameter; by convention, a pointer to the driver's `softc`. The return value is a bitwise OR of `FILTER_STRAY`, `FILTER_HANDLED`, and `FILTER_SCHEDULE_THREAD`, as Section 2 described.

The Stage 1 implementation:

```
static int
myfirst_intr_filter(void *arg)
{
    struct myfirst_softc *sc = arg;

    atomic_add_64(&sc->intr_count, 1);
    return (FILTER_HANDLED);
}
```

One line of real work. The counter is incremented atomically because the handler may run concurrently on multiple CPUs (MSI-X scenarios) or in parallel with non-interrupt code that reads the counter through the `sysctl`. A sleep lock would be wrong in a filter; an atomic operation is the lightweight primitive that is safe here.

The return value is `FILTER_HANDLED` because we have no `ithread` work to do and no reason to return `FILTER_STRAY` (Section 6 adds the stray check; Stage 1 assumes the IRQ is ours).

19.4.5 Registering the Handler With `bus_setup_intr`

After the IRQ allocation, the driver calls `bus_setup_intr(9)`:

```
error = bus_setup_intr(dev, sc->irq_res,
    INTR_TYPE_MISC | INTR_MPSAFE,
    myfirst_intr_filter, NULL, sc,
    &sc->intr_cookie);
if (error != 0) {
    device_printf(dev, "bus_setup_intr failed (%d)\n", error);
    goto fail_release_irq;
}
```

The seven arguments:

1. **dev**: the device handle.
2. **sc->irq_res**: the IRQ resource we just allocated.

3. **INTR_TYPE_MISC | INTR_MPSAFE**: the flags. `INTR_TYPE_MISC` categorises the interrupt (Section 2). `INTR_MPSAFE` promises the handler does its own synchronisation.
4. **myfirst_intr_filter**: our filter handler. Non-NULL.
5. **NULL**: the ithread handler. NULL because Stage 1 is filter-only.
6. **sc**: the argument passed to both handlers.
7. **&sc->intr_cookie**: out-parameter where the kernel stores the cookie for later teardown.

The return value is 0 on success, or an `errno` on failure. A failure at this point is rare; the most common cause is an interrupt controller or platform-specific restriction.

A successful `bus_setup_intr` combined with the `device_printf` below produces a short banner in `dmesg` when the driver is loaded:

```
myfirst0: attached filter handler on IRQ resource
```

The IRQ number itself is not in this line; `devinfo -v` and `vmstat -i` show it (the IRQ number depends on the guest's configuration). If you see an additional `myfirst0: [GIANT-LOCKED]` line, your flags argument is missing `INTR_MPSAFE` and the kernel is warning that Giant is being acquired around the handler; fix it.

19.4.6 Describing the Handler to devinfo

An optional but recommended step. `bus_describe_intr(9)` lets the driver attach a human-readable name to the handler, which `devinfo -v` and the kernel's diagnostics will use:

```
bus_describe_intr(dev, sc->irq_res, sc->intr_cookie, "legacy");
```

After this call, `vmstat -i` shows the handler's line as `irq19: myfirst0:legacy` rather than plain `irq19: myfirst0`. The suffix is the name the driver provided. For Chapter 19's single-interrupt driver the suffix is mostly decorative; for Chapter 20's MSI-X driver with multiple vectors it becomes essential for distinguishing `rx0`, `rx1`, `tx0`, `admin`, and so on.

19.4.7 The Extended Attach Cascade

Putting the new pieces into the Stage 3 attach from Chapter 18:

```
static int
myfirst_pci_attach(device_t dev)
{
    struct myfirst_softc *sc = device_get_softc(dev);
    int error, capreg;

    sc->dev = dev;
    sc->unit = device_get_unit(dev);
    error = myfirst_init_softc(sc);
}
```

```
if (error != 0)
    return (error);

/* Step 1: allocate BAR0. */
sc->bar_rid = PCIR_BAR(0);
sc->bar_res = bus_alloc_resource_any(dev, SYS_RES_MEMORY,
    &sc->bar_rid, RF_ACTIVE);
if (sc->bar_res == NULL) {
    device_printf(dev, "cannot allocate BAR0\n");
    error = ENXIO;
    goto fail_softc;
}

/* Step 2: walk PCI capabilities (informational). */
if (pci_find_cap(dev, PCIY_EXPRESS, &capreg) == 0)
    device_printf(dev, "PCIe capability at 0x%x\n", capreg);
if (pci_find_cap(dev, PCIY_MSI, &capreg) == 0)
    device_printf(dev, "MSI capability at 0x%x\n", capreg);
if (pci_find_cap(dev, PCIY_MSIX, &capreg) == 0)
    device_printf(dev, "MSI-X capability at 0x%x\n", capreg);

/* Step 3: attach the hardware layer against the BAR. */
error = myfirst_hw_attach_pci(sc, sc->bar_res,
    rman_get_size(sc->bar_res));
if (error != 0)
    goto fail_release_bar;

/* Step 4: allocate the IRQ. */
sc->irq_rid = 0;
sc->irq_res = bus_alloc_resource_any(dev, SYS_RES_IRQ,
    &sc->irq_rid, RF_SHAREABLE | RF_ACTIVE);
if (sc->irq_res == NULL) {
    device_printf(dev, "cannot allocate IRQ\n");
    error = ENXIO;
    goto fail_hw;
}

/* Step 5: register the filter handler. */
error = bus_setup_intr(dev, sc->irq_res,
    INTR_TYPE_MISC | INTR_MPSAFE,
    myfirst_intr_filter, NULL, sc,
    &sc->intr_cookie);
if (error != 0) {
    device_printf(dev, "bus_setup_intr failed (%d)\n", error);
}
```

```

        goto fail_release_irq;
    }
    bus_describe_intr(dev, sc->irq_res, sc->intr_cookie, "legacy");
    device_printf(dev, "attached filter handler on IRQ resource\n");

    /* Step 6: create the cdev. */
    sc->cdev = make_dev(&myfirst_cdevsw, sc->unit, UID_ROOT,
        GID_WHEEL, 0600, "myfirst%d", sc->unit);
    if (sc->cdev == NULL) {
        error = ENXIO;
        goto fail_teardown_intr;
    }
    sc->cdev->si_drv1 = sc;

    /* Step 7: read a diagnostic word from the BAR. */
    MYFIRST_LOCK(sc);
    sc->bar_first_word = CSR_READ_4(sc, 0x00);
    MYFIRST_UNLOCK(sc);
    device_printf(dev, "BAR[0x00] = 0x%08x\n", sc->bar_first_word);

    sc->pci_attached = true;
    return (0);

fail_teardown_intr:
    bus_teardown_intr(dev, sc->irq_res, sc->intr_cookie);
    sc->intr_cookie = NULL;
fail_release_irq:
    bus_release_resource(dev, SYS_RES_IRQ, sc->irq_rid, sc->irq_res);
    sc->irq_res = NULL;
fail_hw:
    myfirst_hw_detach(sc);
fail_release_bar:
    bus_release_resource(dev, SYS_RES_MEMORY, sc->bar_rid, sc->bar_res);
    sc->bar_res = NULL;
fail_softc:
    myfirst_deinit_softc(sc);
    return (error);
}

```

The attach sequence now has seven steps instead of five. Two new goto labels (`fail_teardown_intr`, `fail_release_irq`) extend the cascade. The pattern is the same as Chapter 18's: each step undoes the one before it, chaining down to the softc init.

19.4.8 The Extended Detach

The detach path mirrors the attach, with the interrupt teardown fitting between the cdev teardown and the hardware-layer detach:

```
static int
myfirst_pci_detach(device_t dev)
{
    struct myfirst_softc *sc = device_get_softc(dev);

    if (myfirst_is_busy(sc))
        return (EBUSY);

    sc->pci_attached = false;

    /* Destroy the cdev so no new user-space access starts. */
    if (sc->cdev != NULL) {
        destroy_dev(sc->cdev);
        sc->cdev = NULL;
    }

    /* Tear down the interrupt handler before anything it depends on. */
    if (sc->intr_cookie != NULL) {
        bus_teardown_intr(dev, sc->irq_res, sc->intr_cookie);
        sc->intr_cookie = NULL;
    }

    /* Quiesce callouts and tasks (includes Chapter 17 simulation if
     * attached; includes any deferred taskqueue work). */
    myfirst_quiesce(sc);

    /* Release the Chapter 17 simulation if attached. */
    if (sc->sim != NULL)
        myfirst_sim_detach(sc);

    /* Detach the hardware layer. */
    myfirst_hw_detach(sc);

    /* Release the IRQ resource. */
    if (sc->irq_res != NULL) {
        bus_release_resource(dev, SYS_RES_IRQ, sc->irq_rid,
            sc->irq_res);
        sc->irq_res = NULL;
    }
}
```



```

/* Release the BAR. */
if (sc->bar_res != NULL) {
    bus_release_resource(dev, SYS_RES_MEMORY, sc->bar_rid,
        sc->bar_res);
    sc->bar_res = NULL;
}

myfirst_deinit_softc(sc);

device_printf(dev, "detached\n");
return (0);
}

```

Two changes from Chapter 18: the `bus_tearardown_intr` call, and the `bus_release_resource(..., SYS_RES_IRQ, ...)` call. The ordering is important. `bus_tearardown_intr` must happen before anything the handler reads or writes is freed; in particular, before `myfirst_hw_detach` (which frees `sc->hw`). After `bus_tearardown_intr` returns, the kernel guarantees the handler is not running and will not be called again; the driver can then free anything the handler touched.

Releasing the IRQ resource happens after the teardown and after the hardware layer is detached. The exact position between the hardware detach and the BAR release is a judgment call: the BAR and the IRQ do not depend on each other, so either order works. The Chapter 19 driver releases the IRQ first because that is the reverse of the attach order (attach allocated the IRQ after the BAR; detach releases it before the BAR).

Section 7 has more to say about ordering.

19.4.9 The sysctl for the Interrupt Counter

A small diagnostic: a sysctl exposing the `intr_count` field so the reader can watch the counter grow:

```

SYSCTL_ADD_U64(&sc->sysctl_ctx,
    SYSCTL_CHILDREN(sc->sysctl_tree), OID_AUTO, "intr_count",
    CTLFLAG_RD, &sc->intr_count, 0,
    "Number of times the interrupt filter has run");

```

After load, `sysctl dev.myfirst.0.intr_count` returns the current count. For the `virtio-rnd` device with no interrupts firing (the device has nothing to signal yet), the count stays at zero. Section 5's simulated interrupts will drive the count up without needing real IRQ events.

The `sysctl` is readable by any user (the `CTLFLAG_RD` makes it world-readable at the sysctl level; file permissions on the sysctl MIB are set elsewhere). Access is through:

```
sysctl dev.myfirst.0.intr_count
```

19.4.10 What Stage 1 Proves

Loading the Stage 1 driver on a bhyve guest with a virtio-rnd device produces:

```
myfirst0: <Red Hat Virtio entropy source (myfirst demo target)> ... on pci0
myfirst0: attaching: vendor=0x1af4 device=0x1005 revid=0x00
myfirst0: BAR0 allocated: 0x20 bytes at 0xc1000000
myfirst0: hardware layer attached to BAR: 32 bytes
myfirst0: attached filter handler on IRQ resource
myfirst0: BAR[0x00] = 0x10010000
```

`vmstat -i | grep myfirst` shows the interrupt event the kernel created:

```
irq19: myfirst0:legacy          0          0
```

(The IRQ number and the rate depend on the environment.)

The initial count is zero because the virtio-rnd device is not generating interrupts yet (we have not programmed it to). The driver is correctly wired up, the handler is registered, and the filter is ready to fire. Stage 1's job is done.

19.4.11 What Stage 1 Does Not Do

Several things are deliberately absent from Stage 1 and appear in later stages:

- **Status register reading.** The filter does not read the device's `INTR_STATUS`; it just increments a counter. Section 4 adds the status read.
- **Acknowledgment.** The filter does not write `INTR_STATUS` to acknowledge. On a level-triggered line with a device that is actually firing, this is a bug. On our virtio-rnd target in bhyve, the device is not firing, so the absence is invisible. Section 4 adds the acknowledgment and explains why it matters.
- **Ithread handler.** No deferred work yet. Section 4 introduces a taskqueue-based deferred path and wires the filter to schedule it.
- **Simulated interrupts.** There is no way to make the handler fire without a real IRQ from the device. Section 5 adds a `sysctl` that invokes the filter directly under the driver's normal lock rules.
- **Shared-IRQ discipline.** The filter assumes every interrupt belongs to our device. Section 6 adds the `FILTER_STRAY` check for devices that share a line.

These are the topics of Sections 4, 5, and 6. Stage 1 is deliberately incomplete; each later section adds a specific thing that Stage 1 left out.

19.4.12 Common Mistakes at This Stage

A short list of traps beginners hit at Stage 1.

Forgetting `INTR_MPSAFE`. The handler gets wrapped in Giant. Scalability disappears. `dmesg` prints `[GIANT-LOCKED]`. Fix: add `INTR_MPSAFE` to the flags argument.

Passing the wrong argument to the filter. A C function pointer is fussy; passing `&sc` instead of `sc` produces a double pointer that the filter then dereferences incorrectly. The result is usually a kernel panic. Fix: the arg in `bus_setup_intr` is `sc`; the filter receives the same value as a void `*`.

Returning 0 from the filter. The return value is a bitwise OR of `FILTER_*` values. Zero is “no flags”, which is illegal (the kernel requires at least one of `FILTER_STRAY`, `FILTER_HANDLED`, or `FILTER_SCHEDULE_THREAD`). The debug kernel asserts on this. Fix: return `FILTER_HANDLED`.

Using a sleep lock in the filter. The filter takes `sc->mtx` (a regular sleep mutex). `WITNESS` complains; the debug kernel panics. Fix: use atomic operations, or move the work to an `ithread`.

Tearing down the IRQ before the cookie is consumed. Calling `bus_release_resource` on the IRQ before `bus_teardown_intr` is a bug: the resource is gone, but the handler is still registered against it. The next interrupt fires and the kernel dereferences freed state. Fix: always `bus_teardown_intr` first.

Mismatched rid. The rid passed to `bus_release_resource` must match the rid that `bus_alloc_resource_any` returned (or the rid passed in initially, for `rid = 0`). Mismatch usually surfaces as “Resource not found” or a kernel message. Fix: store the rid in the `softc` alongside the resource handle.

Forgetting to drain pending deferred work before teardown. This applies more in Stage 2, but worth flagging here: if the filter has scheduled a taskqueue item, the item must complete before the `softc` goes away. A teardown that releases the IRQ but leaves a taskqueue item pending produces a use-after-free when the item runs.

19.4.13 Checkpoint: Stage 1 Working

Before Section 4, confirm Stage 1 is in place:

- `kldstat -v | grep myfirst` shows the driver at version `1.2-intr-stage1`.
- `dmesg | grep myfirst` shows the attach banner including attached filter handler on IRQ resource.
- No `[GIANT-LOCKED]` warning.
- `devinfo -v | grep -A 5 myfirst` shows both the BAR and the IRQ resource.
- `vmstat -i | grep myfirst` shows the handler’s row.
- `sysctl dev.myfirst.0.intr_count` returns 0 (or a small number, depending on whether the device happens to interrupt).
- `kldunload myfirst` runs cleanly; no panic, no warning.

If any step fails, return to the relevant subsection. The failures are diagnosed the same way Chapter 18’s failures were: check `dmesg` for banners, check `devinfo -v` for resources, check `WITNESS` output for lock-order issues.

19.4.14 Wrapping Up Section 3

Registering an interrupt handler is three new calls (`bus_alloc_resource_any` with `SYS_RES_IRQ`, `bus_setup_intr`, `bus_describe_intr`), three new `softc` fields (`irq_res`, `irq_rid`, `intr_cookie`), and one new counter (`intr_count`). The attach cascade grows by two labels; the detach path grows by the `bus_teardown_intr` call. The handler itself is a one-line atomic increment returning `FILTER_HANDLED`.

The point of Stage 1 is not the work the handler does. The point is that the handler is correctly registered, `INTR_MPSAFE` is set, the counter increments when an interrupt fires, and the teardown runs cleanly on unload.

Every later stage builds on top of this scaffolding; getting it right now is the investment that pays off through the rest of the chapter.

Section 4 makes the handler do real work: read `INTR_STATUS`, decide what to do, acknowledge the device, and defer bulk work to a taskqueue. This is the heart of a real interrupt handler, and the content that matters most for the rest of Part 4.

19.5 Section 4: Writing a Real Interrupt Handler

Stage 1 proved the handler wiring is correct. Stage 2 makes the handler do the work a real driver's filter does. The structure of Section 4 is a close walk through the Chapter 17 simulation's hardware model, with the filter now reading and acknowledging the device's `INTR_STATUS` register, making decisions based on which bits are set, handling the urgent small work inline, and deferring the bulk work to a taskqueue. By the end of Section 4 the driver is at version `1.2-intr-stage2` and has a filter-plus-task pipeline that behaves like a small real driver.

19.5.1 The Register Picture

A quick recap of the Chapter 17 simulation's interrupt register layout (see `HARDWARE.md` for the full details). Offset `0x14` holds `INTR_STATUS`, a 32-bit register with these bits defined:

- `MYFIRST_INTR_DATA_AV` (`0x00000001`): a data-available event has occurred.
- `MYFIRST_INTR_ERROR` (`0x00000002`): an error condition has been detected.
- `MYFIRST_INTR_COMPLETE` (`0x00000004`): a command has completed.

The register is “write-one-to-clear” (RW1C) semantics: writing a 1 to a bit clears that bit; writing a 0 leaves it unchanged. This is the standard PCI interrupt-status convention and is what Chapter 19's handler expects.

Offset `0x10` holds `INTR_MASK`, a parallel register that controls which bits of `INTR_STATUS` actually assert the IRQ line. Setting a bit in `INTR_MASK` enables that interrupt class; clearing it disables it. The driver sets `INTR_MASK` at attach time to enable the interrupts it wants to receive.

Chapter 17's simulation can drive these bits autonomously. Chapter 18's PCI driver runs against a real `virtio-rnd` BAR, where the offsets mean something different (`virtio` legacy configuration, not the Chapter 17 register map). Section 4 writes the handler against the Chapter 17 semantics; Section 5 shows how to exercise the handler without real IRQ events; the `virtio-rnd` device does not implement this register layout, so in the bhyve lab the handler is primarily exercised through the simulated-interrupt path.

This is an honest limitation of the teaching target. A reader adapting the driver to a real device that implements Chapter 17-style registers would see the handler fire on real interrupts directly. For the bhyve `virtio-rnd` target, Section 5's `sysctl`-triggered handler is the way to exercise the Stage 2 filter in practice.

19.5.2 The Filter Handler at Stage 2

The Stage 2 filter reads `INTR_STATUS`, decides what happened, acknowledges the bits it is handling, and either does the urgent work inline or schedules a task for bulk work.

```
int
myfirst_intr_filter(void *arg)
{
    struct myfirst_softc *sc = arg;
    uint32_t status;
    int rv = 0;

    /*
     * Read the raw status. The filter runs in primary interrupt
     * context and cannot take sc->mtx (a sleep mutex), so the access
     * goes through the specialised accessor that asserts the correct
     * context. We use a local, spin-safe helper for the BAR access;
     * Stage 2 uses a small inline instead of the lock-asserting
     * CSR_READ_4 macro.
     */
    status = bus_read_4(sc->bar_res, MYFIRST_REG_INTR_STATUS);
    if (status == 0)
        return (FILTER_STRAY);

    atomic_add_64(&sc->intr_count, 1);

    /* Handle the DATA_AV bit: small urgent work only. */
    if (status & MYFIRST_INTR_DATA_AV) {
        atomic_add_64(&sc->intr_data_av_count, 1);
        bus_write_4(sc->bar_res, MYFIRST_REG_INTR_STATUS,
            MYFIRST_INTR_DATA_AV);
        taskqueue_enqueue(sc->intr_tq, &sc->intr_data_task);
        rv |= FILTER_HANDLED;
    }

    /* Handle the ERROR bit: log and acknowledge. */
    if (status & MYFIRST_INTR_ERROR) {
        atomic_add_64(&sc->intr_error_count, 1);
        bus_write_4(sc->bar_res, MYFIRST_REG_INTR_STATUS,
            MYFIRST_INTR_ERROR);
        rv |= FILTER_HANDLED;
    }

    /* Handle the COMPLETE bit: wake any pending waiters. */
    if (status & MYFIRST_INTR_COMPLETE) {
        atomic_add_64(&sc->intr_complete_count, 1);
        bus_write_4(sc->bar_res, MYFIRST_REG_INTR_STATUS,
            MYFIRST_INTR_COMPLETE);
        rv |= FILTER_HANDLED;
    }
}
```

```

    }

    /* If we didn't recognise any bit, this wasn't our interrupt. */
    if (rv == 0)
        return (FILTER_STRAY);

    return (rv);
}

```

Several things worth reading carefully.

The raw access. The filter uses `bus_read_4` and `bus_write_4` (the newer resource-based accessors) directly, not the `CSR_READ_4` and `CSR_WRITE_4` macros from Chapter 16. The reason is subtle. The Chapter 16 macros take `sc->mtx` via `MYFIRST_ASSERT`, which is a sleep mutex. A filter must not take a sleep mutex. The right approach is either to use the raw `bus_space` accessors directly (as shown) or to introduce a parallel family of CSR macros that assert no lock requirement. Section 8's refactor introduces `ICSR_READ_4` and `ICSR_WRITE_4` ("I" for interrupt-context) to make the distinction explicit; Stage 2 uses the raw accessors.

The early stray check. A status of zero means no bit is set; this is a shared-IRQ call from another driver. Returning `FILTER_STRAY` lets the kernel try the next handler. The check is also a defence against a real hardware race: if the interrupt controller asserts the line but the device has already cleared the status (by the time we read it), we should not claim the interrupt.

The per-bit handling. Each bit of interest is checked, counted, and acknowledged. The order does not matter (the bits are independent), but the structure is conventional: one `if` per bit.

The acknowledgment. Writing the bit back to `INTR_STATUS` clears it (RW1C). This is what makes the interrupt line deassert. Failing to acknowledge on a level-triggered line produces an interrupt storm.

The taskqueue enqueue. The `DATA_AV` bit triggers deferred work. The filter enqueues a task; the taskqueue's worker thread runs the task later in thread context, where it can take sleep locks and do slow work. The enqueue is safe to call from a filter (taskqueues use spin locks internally for this path).

The final return value. A bitwise OR of `FILTER_HANDLED` for each bit we recognised, or `FILTER_STRAY` if nothing matched. If we had work for an ithread, we would OR in `FILTER_SCHEDULE_THREAD`; but Stage 2 uses a taskqueue rather than the ithread, so the return value is just `FILTER_HANDLED`.

19.5.3 Why a Taskqueue, Not an ithread?

FreeBSD lets a driver register an ithread handler through the fifth argument of `bus_setup_intr(9)`. Why does Stage 2 use a taskqueue instead?

Two reasons.

First, the taskqueue is more flexible. An ithread is tied to the specific `intr_event`; it runs the driver's ithread function after the filter. A taskqueue lets the driver schedule a task from any context (filter, ithread, other tasks, user-space ioctl paths) and have it run on a shared worker thread. For the Chapter 19 driver, which exercises the handler through simulated interrupts as well as real ones, the taskqueue is a more uniform deferred-work primitive.

Second, the taskqueue separates priority from the interrupt type. The ithread's priority is derived from `INTR_TYPE_*`; the taskqueue's priority is controlled by `taskqueue_start_threads(9)`. For drivers that want their deferred work at a different priority than the interrupt category implies, the taskqueue gives that control.

Real FreeBSD drivers use both patterns. Simple drivers with fire-and-forget interrupts often use the ithread (less code). Drivers with richer deferred-work patterns use taskqueues. The `iflib(9)` framework uses a kind of hybrid.

Chapter 19 teaches the taskqueue pattern because it composes better with the rest of the book. Chapter 17 already has a taskqueue; Chapter 14 introduced the pattern; the deferred-work discipline is a book-wide theme.

19.5.4 The Deferred-Work Task

The filter enqueued `sc->intr_data_task` when it saw `DATA_AV`. That task is:

```
static void
myfirst_intr_data_task_fn(void *arg, int npending)
{
    struct myfirst_softc *sc = arg;

    MYFIRST_LOCK(sc);

    /*
     * The data-available event has fired. Read the device's data
     * register through the Chapter 16 accessor (which takes sc->mtx
     * implicitly), update the driver's state, and wake any waiting
     * readers.
     */
    uint32_t data = CSR_READ_4(sc, MYFIRST_REG_DATA_OUT);
    sc->intr_last_data = data;
    sc->intr_task_invocations++;

    /* Wake any thread sleeping on the data-ready condition. */
    cv_broadcast(&sc->data_cv);

    MYFIRST_UNLOCK(sc);
}
```

Several notable properties.

The task runs in thread context. It can take `sc->mtx`, use `cv_broadcast`, call `malloc(M_WAITOK)`, and do slow work.

The task respects the Chapter 11 locking discipline. The mutex is acquired; the CSR access uses the standard Chapter 16 macro; the condition-variable broadcast uses the Chapter 12 primitive.

The task's argument is the softc. Same as the filter. One subtle implication: the task cannot assume the driver has not been detached. If detach fires after the filter has enqueued the task but before the task runs, the task could execute against a freed softc. Section 7 covers the discipline that prevents this (drain before release).

The npending argument is the number of times the task was enqueued since its last run. For most drivers this is useful as a coalescing hint: if npending is 5, the device signalled five data-ready events that all coalesce into one run. Stage 2's task ignores it; larger drivers use it to size batch operations.

19.5.5 Declaring and Initialising the Task

The softc gains task-related fields:

```
struct myfirst_softc {
    /* ... existing fields ... */

    /* Chapter 19 interrupt-related fields. */
    struct resource      *irq_res;
    int                  irq_rid;
    void                 *intr_cookie;
    uint64_t             intr_count;
    uint64_t             intr_data_av_count;
    uint64_t             intr_error_count;
    uint64_t             intr_complete_count;
    uint64_t             intr_task_invocations;
    uint32_t             intr_last_data;

    struct taskqueue     *intr_tq;
    struct task          intr_data_task;
};
```

In myfirst_init_softc (or the init path):

```
TASK_INIT(&sc->intr_data_task, 0, myfirst_intr_data_task_fn, sc);
sc->intr_tq = taskqueue_create("myfirst_intr", M_WAITOK,
    taskqueue_thread_enqueue, &sc->intr_tq);
taskqueue_start_threads(&sc->intr_tq, 1, PI_NET,
    "myfirst intr taskq");
```

The taskqueue is created with one worker thread at priority PI_NET (an interrupt priority; see /usr/src/sys/sys/priority.h). The name "myfirst intr taskq" appears in top -H for diagnosis. The M_WAITOK during create is fine because myfirst_init_softc runs in attach context, before any interrupts fire.

19.5.6 Enabling the Interrupts at the Device

A detail often forgotten: the device itself must be told to deliver interrupts. For the Chapter 17 simulation's register layout, this is done by setting bits in the INTR_MASK register:

```
/* After attaching the hardware layer, enable the interrupts we care
   * about. */
MYFIRST_LOCK(sc);
CSR_WRITE_4(sc, MYFIRST_REG_INTR_MASK,
    MYFIRST_INTR_DATA_AV | MYFIRST_INTR_ERROR |
    MYFIRST_INTR_COMPLETE);
MYFIRST_UNLOCK(sc);
```

The INTR_MASK register controls which bits of INTR_STATUS actually assert the IRQ line. Without it the device may set INTR_STATUS bits internally but never raise the line, so the handler never fires. Setting all three bits enables all three interrupt classes.

This is another honest limitation of the teaching target. Offset 0x10 on the virtio-rnd legacy BAR is not an interrupt-mask register at all. In the legacy virtio layout (see /usr/src/sys/dev/virtio/pci/virtio_pci_legacy_var.h), the dword starting at 0x10 is shared by three small fields: queue_notify at offset 0x10 (16-bit), device_status at offset 0x12 (8-bit), and isr_status at offset 0x13 (8-bit). A 32-bit write of our DATA_AV | ERROR | COMPLETE pattern (0x00000007) at that offset writes 0x0007 to queue_notify (notifying a virtqueue index the device does not have) and 0x00 to device_status (which the virtio specification defines as a **device reset**). Writing zero to device_status is how the virtio driver is supposed to reset the device before reinitialisation.

For that reason, the CSR_WRITE_4(sc, MYFIRST_REG_INTR_MASK, ...) call as written is **safe but pointless** on the bhyve virtio-rnd target: it resets the device's virtio state machine (which our driver was not using anyway), and it never enables any real interrupt because the Chapter 17 INTR_MASK register does not exist on that device. If you plan to drive a reader through this on bhyve, keep the write in the code for continuity with a real Chapter-17-compatible device, and rely on Section 5's simulated-interrupt sysctl for testing instead of expecting real IRQ events. A reader who adapts the driver to a real device that matches the Chapter 17 register map would see the mask write do its job.

19.5.7 Disabling the Interrupts at Detach

The symmetric step at detach:

```
/* Disable all interrupts at the device before tearing down. */
MYFIRST_LOCK(sc);
if (sc->hw != NULL)
    CSR_WRITE_4(sc, MYFIRST_REG_INTR_MASK, 0);
MYFIRST_UNLOCK(sc);
```

This write happens before bus_tearardown_intr so the device stops asserting the line before the handler is removed. The guard against sc->hw == NULL protects against partial-attach cases where the hardware layer failed; the disable is skipped if the hardware is not attached.

19.5.8 A Worked Flow

A concrete trace of what happens when a `DATA_AV` event fires (on a device that actually implements the Chapter 17 semantics):

1. The device sets `INTR_STATUS.DATA_AV`. Because `INTR_MASK.DATA_AV` is set, the device asserts its IRQ line.
2. The interrupt controller routes the IRQ to a CPU.
3. The CPU takes the interrupt and jumps to the kernel's dispatch code.
4. The kernel finds the `intr_event` for our IRQ and calls `myfirst_intr_filter`.
5. The filter reads `INTR_STATUS`, sees `DATA_AV`, increments counters, writes `DATA_AV` back to `INTR_STATUS` (clearing it), enqueues `intr_data_task`, returns `FILTER_HANDLED`.
6. The device deasserts its IRQ line (because `INTR_STATUS.DATA_AV` is now clear).
7. The kernel sends EOI, returns to the interrupted thread.
8. Some milliseconds later, the taskqueue's worker thread wakes up, runs `myfirst_intr_data_task_fn`, reads `DATA_OUT`, updates the softc, broadcasts the condition variable.
9. Any thread that was waiting on the condition variable wakes up and proceeds.

Steps 1 through 7 take a few microseconds. Step 8 can take hundreds of microseconds or more, which is why it is in thread context. The split is what lets the interrupt path stay fast.

For the bhyve virtio-rnd target, steps 1 through 6 do not happen (the device does not match the Chapter 17 register layout). Steps 4 through 9 can still be exercised through the Section 5 simulated-interrupt path.

19.5.9 Urgent Work Inline vs Deferred

A useful way to decide what goes in the filter versus in the task: the filter handles what must be done **per interrupt**, the task handles what must be done **per event**.

Per-interrupt (filter): - Read `INTR_STATUS` to identify the event. - Acknowledge the event at the device (write back to `INTR_STATUS`). - Update counters. - Make a single scheduling decision (enqueue the task).

Per-event (task): - Read data from device registers or DMA buffers. - Update the driver's internal state machine. - Wake waiting threads. - Pass data up to the network stack, the storage stack, or the cdev queue. - Handle errors that require slow recovery.

The rule of thumb: if the filter takes more than a hundred CPU cycles of real work (not counting the register accesses, which are cheap in themselves), it is probably doing too much.

19.5.10 FILTER_SCHEDULE_THREAD vs Taskqueue

A reader might ask: when would I use `FILTER_SCHEDULE_THREAD` instead of a taskqueue?

Use `FILTER_SCHEDULE_THREAD` when: - You want the kernel's per-event ithread (one per `intr_event`) to run the slow work. - You do not need to schedule the work from anywhere except the filter. - You want the scheduling priority to follow the interrupt's `INTR_TYPE_*`.

Use a taskqueue when: - You want to schedule the same work from multiple paths (filter, `ioctl`, `sysctl`, sleep-based timeout). - You want to share the worker thread across multiple devices. - You want explicit control over priority via `taskqueue_start_threads`.

For the Chapter 19 driver the taskqueue is the cleaner choice because Section 5 will schedule the same task from a simulated-interrupt path. An ithread would not be reachable from there.

19.5.11 When the Taskqueue Itself Is the Wrong Answer

A caveat. The taskqueue is great for short deferred work. It is not great for long-running operations. If the driver needs to run a state machine for seconds, or block waiting on a USB transfer, or process a large buffer chain, a dedicated worker thread is better. The taskqueue's worker thread is shared across tasks; a single task that blocks for a long time delays every other task behind it.

Chapter 19's task runs for microseconds. The taskqueue is fine. Chapter 20's MSI-X driver with per-queue receive processing may want per-queue worker threads. Chapter 21's DMA-driven bulk transfer may want a dedicated thread. Each chapter picks the right primitive for its workload; Chapter 19 uses the simplest one that fits.

19.5.12 Common Mistakes at This Stage

A short list.

Reading INTR_STATUS without acknowledging. The handler reads, decides, and returns without writing back. On a level-triggered line, the device keeps asserting; the handler fires again immediately; storm. Fix: acknowledge every bit you handled.

Acknowledging too many bits. A sloppy handler writes 0xffffffff to INTR_STATUS on every call to “clear all bits”. This also clears events the handler has not processed, dropping data or confusing the state machine. Fix: acknowledge only the bits you actually handled.

Taking sleep locks in the filter. MYFIRST_LOCK(sc) takes sc->mtx, which is a sleep mutex. In the filter this is a bug; WITNESS panics. Fix: use atomic operations in the filter, and take the sleep mutex only in the task (which runs in thread context).

Scheduling a task after the softc is torn down. If the task is scheduled from the filter but the filter runs after detach has partially torn down, the task runs against stale state. Fix: Section 7 covers the ordering. Briefly: bus_tearardown_intr must happen before the hardware layer is freed, and taskqueue_drain must happen before the taskqueue is freed.

Using CSR_READ_4/CSR_WRITE_4 directly in the filter. If the Chapter 16 accessor asserts sc->mtx held (which it does on debug kernels), the filter panics. Fix: use raw bus_read_4/bus_write_4 or introduce a parallel set of interrupt-safe CSR macros. Section 8 handles this with ICSR_READ_4.

Enqueueing the task without TASK_INIT. A task enqueued before TASK_INIT has a corrupt function pointer. The first run of the task jumps to garbage. Fix: initialise the task in the attach path before enabling interrupts.

Forgetting to enable interrupts at the device. The handler is registered and bus_setup_intr succeeded; vmstat -i still shows zero firings. The problem is the device's INTR_MASK register is still zero (or whatever post-reset value it has), so the device never asserts the line. Fix: write the INTR_MASK during attach.

Forgetting to disable interrupts at detach. The handler has been torn down but the device is still asserting the line. The kernel will eventually complain about a stray interrupt, or (worse) another driver that shares the line will see mysterious activity. Fix: clear INTR_MASK before bus_tearardown_intr.

19.5.13 Stage 2 Output: What Success Looks Like

After loading Stage 2 on a real device that produces interrupts, `dmesg` shows:

```
myfirst0: <Red Hat Virtio entropy source (myfirst demo target)> ... on pci0
myfirst0: attaching: vendor=0x1af4 device=0x1005 revid=0x00
myfirst0: BAR0 allocated: 0x20 bytes at 0xc1000000
myfirst0: hardware layer attached to BAR: 32 bytes
myfirst0: attached filter handler on IRQ resource
myfirst0: interrupts enabled (mask=0x7)
myfirst0: BAR[0x00] = 0x10010000
```

The `interrupts enabled` line is new. It confirms the driver has written to `INTR_MASK`.

On a real device generating interrupts, `sysctl dev.myfirst.0.intr_count` will tick up. On the bhyve `virtio-rnd` target, the count stays at zero because the device does not fire our expected interrupts. Section 5's simulated-interrupt path is the way to exercise the handler from there.

19.5.14 Wrapping Up Section 4

A real interrupt handler reads `INTR_STATUS` to identify the cause, handles each bit of interest, acknowledges the bits it handled by writing back to `INTR_STATUS`, and returns the right combination of `FILTER_*` values. Urgent work (register access, counter updates, acknowledgments) happens in the filter. Slow work (data reads through Chapter 16 accessors that take `sc->mtx`, condition-variable broadcasts, user-space notifications) happens in a taskqueue task that the filter enqueues.

The filter is short (twenty to forty lines of real code for a typical device). The task is short too (ten to thirty lines). The composition is what makes the driver functional: the filter handles interrupts at interrupt rate; the task handles events at thread-rate; the split keeps the interrupt window tight and the deferred work free to block.

Section 5 is the section that lets a reader exercise this machinery on a bhyve target where the real IRQ path does not match the Chapter 17 register semantics. It adds a `sysctl` that invokes the filter under the driver's normal lock rules, lets the reader trigger simulated interrupts at will, and confirms that the counters, the task, and the condition-variable broadcast all behave as designed.

19.6 Section 5: Using Simulated Interrupts for Testing

Section 4's filter and task are real driver code. They are ready to handle real interrupts on a device that matches the Chapter 17 register layout. The problem Chapter 19's lab target presents is that the device we have (`virtio-rnd` under bhyve) does not match that layout; writing the Chapter 17 interrupt-mask bits to the `virtio-rnd` BAR has defined but unrelated effects, and reading Chapter 17 interrupt-status bits from the `virtio-rnd` BAR returns `virtio`-specific values that have nothing to do with our simulated semantics. On this target the filter, if it fires at all, will see garbage.

Section 5 solves this by teaching the reader to simulate interrupts. The core idea is simple: expose a `sysctl` that, when written, invokes the filter handler directly under the driver's normal lock rules, exactly as the kernel would

invoke it from a real interrupt. The filter reads the `INTR_STATUS` register (which the reader has also written to through another `sysctl`, or through the Chapter 17 simulation backend on a simulation-only build), makes the same decisions it would on a real interrupt, and drives the full pipeline end-to-end.

19.6.1 Why Simulation Deserves a Section

A reader who finished Chapter 17 might reasonably ask: was the whole Chapter 17 simulation not already a way to simulate interrupts? Yes and no.

Chapter 17 simulated an **autonomous device**. Its callouts changed register values on their own schedule, its command callout fired when the driver wrote `CTRL.GO`, its fault-injection framework made the simulated device misbehave. The driver at Chapter 17 was a simulation-only driver; there was no `bus_setup_intr` because there was no real bus.

Chapter 19 is different. The driver now has a real `bus_setup_intr` handler registered on a real IRQ line. Chapter 17's callouts are not involved; on the PCI build the Chapter 17 simulation does not run. What we want is a way to trigger the **filter handler** directly, with the exact lock semantics a real interrupt would produce, so we can validate Section 4's filter and task pipeline without depending on a device that actually produces the right interrupts.

The cleanest way to do this, and the way many FreeBSD drivers have for similar purposes, is a `sysctl` write that invokes the filter function directly. The filter runs in the caller's context (thread context, where the `sysctl` write originates), but the filter's code does not care about the outer context as long as the lock discipline inside is right. An atomic increment, a BAR read, a BAR write, a `taskqueue_enqueue`: all of these work from thread context too. The simulated call exercises the same code paths the kernel would exercise on a real interrupt.

There is one subtle distinction. On a real interrupt, the kernel arranges for filters on the same `intr_event` to run serially on one CPU. A `sysctl`-triggered simulated call does not have that guarantee; another thread could be invoking the filter at the same moment. For the Chapter 19 driver this is fine because the filter's state is protected by atomic operations (not by the kernel's single-CPU-per-IRQ guarantee). For a driver that relies on implicit single-CPU serialisation, simulation through a `sysctl` would not be a faithful test. The lesson is: `INTR_MPSAFE` drivers that use atomics and spin locks translate cleanly to simulation.

19.6.2 The Simulated-Interrupt `sysctl`

The mechanism is a write-only `sysctl` that invokes the filter:

```
static int
myfirst_intr_simulate_sysctl(SYSCTL_HANDLER_ARGS)
{
    struct myfirst_softc *sc = arg1;
    uint32_t mask;
    int error;

    mask = 0;
    error = sysctl_handle_int(oidp, &mask, 0, req);
    if (error != 0 || req->newptr == NULL)
```

```

        return (error);

    /*
     * "mask" is the INTR_STATUS bits the caller wants to pretend
     * the device has set. Set them in the real register, then call
     * the filter directly.
     */
    MYFIRST_LOCK(sc);
    if (sc->hw == NULL) {
        MYFIRST_UNLOCK(sc);
        return (ENODEV);
    }
    bus_write_4(sc->bar_res, MYFIRST_REG_INTR_STATUS, mask);
    MYFIRST_UNLOCK(sc);

    /* Invoke the filter directly. */
    (void)myfirst_intr_filter(sc);

    return (0);
}

```

And the sysctl declaration in `myfirst_intr_add_sysctls`:

```

SYSCTL_ADD_PROC(&sc->sysctl_ctx,
    SYSCTL_CHILDREN(sc->sysctl_tree), OID_AUTO, "intr_simulate",
    CTLTYPE_UINT | CTLFLAG_WR | CTLFLAG_MPSAFE,
    sc, 0, myfirst_intr_simulate_sysctl, "IU",
    "Simulate an interrupt by setting INTR_STATUS bits and "
    "invoking the filter");

```

Writing to `dev.myfirst.0.intr_simulate` causes the handler to run with the specified `INTR_STATUS` bits set.

19.6.3 Exercising the Simulation

Once the sysctl is in place, the reader can drive the full pipeline from user space:

```

# Simulate a DATA_AV event.
sudo sysctl dev.myfirst.0.intr_simulate=1

# Check counters.
sysctl dev.myfirst.0.intr_count
sysctl dev.myfirst.0.intr_data_av_count

```

```
sysctl dev.myfirst.0.intr_task_invocations

# Simulate an ERROR event.
sudo sysctl dev.myfirst.0.intr_simulate=2

# Simulate a COMPLETE event.
sudo sysctl dev.myfirst.0.intr_simulate=4

# Simulate all three at once.
sudo sysctl dev.myfirst.0.intr_simulate=7
```

The first call increments `intr_count` (filter fired), `intr_data_av_count` (DATA_AV bit recognised), and eventually `intr_task_invocations` (the taskqueue task ran). The second increments `intr_count` and `intr_error_count`. The third increments `intr_count` and `intr_complete_count`. The fourth hits all three.

A reader can verify the full pipeline:

```
# Watch counters in a loop.
while true; do
    sudo sysctl dev.myfirst.0.intr_simulate=1
    sleep 0.5
    sysctl dev.myfirst.0 | grep intr_
done
```

The counters tick forward at the expected rate. The driver behaves as if real interrupts were arriving.

19.6.4 Why This Is Not a Toy

One might think this simulated path is a teaching artifact. It is not. Many real drivers keep a similar path for diagnostic purposes. The reasons are worth naming:

Regression testing. A simulated-interrupt path lets a CI pipeline exercise the handler without needing real hardware. Chapter 17 made the same argument for simulating device behaviour; Section 5 makes it for simulating the interrupt path.

Fault injection. A simulated-interrupt `sysctl` lets a test inject specific `INTR_STATUS` patterns to exercise error-handling code. The driver's response to `INTR_STATUS = ERROR | COMPLETE` (both bits set simultaneously) is hard to trigger through real hardware; a `sysctl` that sets both bits and calls the handler makes it easy.

Developer productivity. When a driver author is debugging the handler logic, having a `sysctl` that triggers the handler on demand is enormously useful. `dtrace -n 'fbt::myfirst_intr_filter:entry'` combined with `sudo sysctl dev.myfirst.0.intr_simulate=1` gives a single-step view of the handler on demand.

Bringing up new hardware. A driver author often has a prototype device that does not yet produce interrupts correctly. A simulated-interrupt path lets the driver's upper layers be tested before the hardware works, which means the driver and the hardware can be brought up in parallel rather than serially.

Teaching. For this book’s purpose, the simulated path makes the filter and task observable on a lab target that does not naturally produce the expected interrupts. The reader can see the pipeline work even though the hardware does not cooperate.

19.6.5 Locking in the Simulated Path

A detail worth thinking through. The sysctl writes `INTR_STATUS` while holding `sc->mtx`. The filter handler, when invoked through the kernel’s real interrupt path, runs without `sc->mtx` held (the filter uses `bus_read_4` / `bus_write_4` directly, not the lock-asserting CSR macros). When invoked through the sysctl, what is the calling context?

The sysctl handler runs in thread context. The `MYFIRST_LOCK(sc)` acquires the sleep mutex. Between the lock acquisition and the release, the thread holds the mutex. Then the lock is released, and `myfirst_intr_filter(sc)` is called. The filter takes no locks, uses only atomics and `bus_read_4/bus_write_4`, enqueues a task, and returns. The whole sequence is safe.

Would it be safe to call the filter with `sc->mtx` held? Yes, actually: the filter does not try to acquire the same mutex, and the filter runs in a context where holding the lock is not itself illegal (thread context). But the filter was designed to be context-agnostic; calling it with a sleep lock held would obscure that contract. The sysctl releases the lock before invoking the filter for clarity.

19.6.6 Using the Chapter 17 Simulation to Produce Interrupts

A complementary technique worth mentioning. Chapter 17’s simulation backend, if attached, produces autonomous state changes on its own schedule. In particular, its `DATA_AV`-producing callout sets `INTR_STATUS.DATA_AV`. On a simulation-only build (`MYFIRST_SIMULATION_ONLY` defined at compile time), the simulation is live, the callout fires, and the Chapter 17 driver can even call the filter from the callout itself.

Chapter 19 does not change Chapter 17’s behaviour on simulation-only builds. A reader who wants to see the filter driven by the Chapter 17 simulation can build with `-DMYFIRST_SIMULATION_ONLY`, load the module, and watch the callouts set `INTR_STATUS` bits. The sysctl-triggered path from Section 5 remains available on both builds.

On the PCI build, the Chapter 17 simulation is not attached (Chapter 18’s discipline), so the Chapter 17 callouts do not run. The simulated-interrupt path is the only way to drive the filter on the PCI build.

19.6.7 Extending the sysctl to Schedule at a Rate

A useful extension for load testing: a sysctl that schedules simulated interrupts periodically through a callout. The callout fires every `N` milliseconds, sets a bit in `INTR_STATUS`, and invokes the filter. A reader can tune the rate and observe the pipeline under load.

```
static void
myfirst_intr_sim_callout_fn(void *arg)
{
```



```
struct myfirst_softc *sc = arg;

MYFIRST_LOCK(sc);
if (sc->intr_sim_period_ms > 0 && sc->hw != NULL) {
    bus_write_4(sc->bar_res, MYFIRST_REG_INTR_STATUS,
        MYFIRST_INTR_DATA_AV);
    MYFIRST_UNLOCK(sc);
    (void)myfirst_intr_filter(sc);
    MYFIRST_LOCK(sc);
    callout_reset_sbt(&sc->intr_sim_callout,
        SBT_1MS * sc->intr_sim_period_ms, 0,
        myfirst_intr_sim_callout_fn, sc, 0);
}
MYFIRST_UNLOCK(sc);
}
```

The callout reschedules itself as long as `intr_sim_period_ms` is non-zero. A `sysctl` exposes the period:

```
# Fire a simulated interrupt every 100 ms.
sudo sysctl hw.myfirst.intr_sim_period_ms=100

# Stop simulating.
sudo sysctl hw.myfirst.intr_sim_period_ms=0
```

Watch the counters grow at the expected rate:

```
sleep 10
sysctl dev.myfirst.0.intr_count
```

After ten seconds of 100 ms period, the counter should read about 100. If it reads much less, the filter or the task is the bottleneck (unlikely at this scale; more of a concern for high-rate tests). If it reads much more, something is firing the filter from elsewhere.

19.6.8 What the Simulation Does Not Capture

Honest limits of the technique.

Concurrent firings. The `sysctl` serialises simulated interrupts at one per write. A real interrupt path can see two interrupts back-to-back on different CPUs, which a `sysctl` test would not produce. For stress-testing concurrency, a separate test that spawns multiple threads each writing the `sysctl` is more effective.

Interrupt-controller behaviour. The simulation bypasses the interrupt controller entirely. Tests that depend on EOI timing, masking, or storm detection cannot be driven this way.

CPU affinity. The simulated filter runs on whichever CPU the sysctl-writing thread is on. A real interrupt fires on whichever CPU the affinity configuration selects. Tests of per-CPU behaviour need real interrupts or another mechanism.

Contention with the real interrupt path. If real interrupts are also firing (perhaps because the device actually does generate some), the simulated path can race with the real path. The atomic counters handle it correctly; more complex shared state might not.

These are limits, not deal-breakers. For most Chapter 19 testing, the simulated path is sufficient. For advanced stress testing, additional techniques (rt-threads, multi-CPU invocations, real hardware) apply.

19.6.9 Observing the Task Running

A diagnostic worth exposing. The task’s `intr_task_invocations` counter ticks every time the task runs. A reader can compare it to `intr_data_av_count` to check whether the taskqueue is keeping up:

```
sudo sysctl dev.myfirst.0.intr_simulate=1    # fire DATA_AV
sleep 0.1
sysctl dev.myfirst.0.intr_data_av_count      # should be 1
sysctl dev.myfirst.0.intr_task_invocations   # should also be 1
```

If the task counter lags the `DATA_AV` counter, the taskqueue worker is backlogged. At this scale it should not be; at higher rates (thousands per second) it might be.

A more sensitive probe: add a `cv_signal` path that a user-space program waits on. The sysctl fires the simulated interrupt; the filter enqueues the task; the task updates `sc->intr_last_data` and broadcasts; the user-space thread that was waiting on the condition variable (via the `cdev`’s `read`) wakes up. Round-trip latency from sysctl write to wake-up is the driver’s interrupt-to-user-space latency, roughly, a useful number to know.

19.6.10 Integration With Chapter 17’s Fault Framework

A thought worth noting. Chapter 17’s fault-injection framework (the `FAULT_MASK` and `FAULT_PROB` registers) applies to commands, not to interrupts. Chapter 19 can extend the framework by adding a “fault on next interrupt” option: a sysctl that makes the next filter call skip the acknowledgment, causing a storm on a level-triggered line.

This is an optional extension. The challenge exercises mention it; the chapter’s main body does not require it.

19.6.11 Wrapping Up Section 5

Simulating interrupts is a simple but effective technique. A sysctl writes `INTR_STATUS`, invokes the filter directly, and the filter drives the full pipeline: counter updates, acknowledgment, task enqueue, task execution. The technique lets a driver be exercised end-to-end on a lab target that does not naturally produce the expected interrupts, and it is cheap to keep in production drivers for regression testing and diagnostic access.

Section 6 is the last conceptual piece of core interrupt handling. It covers shared interrupts: what happens when multiple drivers listen on the same IRQ line, how a filter handler must identify whether the interrupt belongs to its device, and what `FILTER_STRAY` means in practice.

19.7 Section 6: Handling Shared Interrupts

Section 4's Stage 2 filter already has the right return-value shape for shared IRQs: it returns `FILTER_STRAY` when `INTR_STATUS` is zero. Section 6 explores why that check is the entire discipline, what goes wrong when a handler gets it wrong, and when the `RF_SHAREABLE` flag is worth setting.

19.7.1 Why Share an IRQ at All?

Two reasons.

First, **hardware constraints**. The classic PC architecture had 16 hardware IRQ lines; the I/O APIC expanded this to 24 on many chipsets. A system with 30 devices necessarily has some of them sharing lines. On modern x86 systems the shortage is less acute (hundreds of vectors), but on legacy PCI and on many arm64 SoCs sharing is normal.

Second, **driver portability**. A driver that handles shared interrupts correctly also handles exclusive interrupts correctly (the shared path is a superset). A driver that assumes exclusive interrupts breaks when the hardware changes or when another driver arrives on the same line. Writing for the shared case costs essentially nothing and future-proofs the driver.

On PCIe with MSI or MSI-X enabled (Chapter 20), each device has its own vectors and sharing is rarely needed. But even there, a driver that correctly handles a stray interrupt (by returning `FILTER_STRAY`) is a better driver than one that does not. The discipline transfers.

19.7.2 The Flow on a Shared IRQ

When a shared IRQ fires, the kernel walks the list of filter handlers attached to the `intr_event` in registration order. Each filter runs, checks whether the interrupt belongs to its device, and returns accordingly:

- If the filter claims the interrupt (returns `FILTER_HANDLED` or `FILTER_SCHEDULE_THREAD`), the kernel continues with the next filter if any and aggregates the results. On modern kernels a filter that returns `FILTER_HANDLED` does not stop later filters from running; the kernel always walks the whole list.
- If the filter returns `FILTER_STRAY`, the kernel tries the next filter.

After all filters have run, if any filter claimed the interrupt, the kernel acknowledges at the interrupt controller and returns. If all filters returned `FILTER_STRAY`, the kernel increments a stray-interrupt counter; if the stray count exceeds a threshold, the kernel disables the IRQ (the drastic last resort).

A filter that returns `FILTER_STRAY` when the interrupt actually was for its device is a bug: the line stays asserted (level-triggered), the storm machinery kicks in, and the device does not get serviced. A filter that returns `FILTER_HANDLED` when the interrupt was not for its device is also a bug: another driver's interrupt is marked as serviced, its handler never runs, its data sits in its FIFO, and the user's network or disk stops working.

The discipline is to decide ownership precisely, based on the device's state, and return the right value.

19.7.3 The `INTR_STATUS` Test

The standard way to decide ownership is to read a device register that tells you whether the interrupt is pending. On a device with a per-device `INTR_STATUS` register, the question is "is any bit in `INTR_STATUS` set?" If

yes, the interrupt is mine. If no, it is not.

The Chapter 17 register layout makes this easy:

```
status = bus_read_4(sc->bar_res, MYFIRST_REG_INTR_STATUS);  
if (status == 0)  
    return (FILTER_STRAY);
```

This is exactly what the Stage 2 filter already does. The pattern is robust: if the status register reads as zero, there is no pending event from this device, so the interrupt is not ours.

A subtle detail: the read of `INTR_STATUS` must happen before any state changes that could mask or reset the bits. Reading `INTR_STATUS` with the device in a mid-state is fine (the register reflects the device's current view); writing to other registers first and then reading `INTR_STATUS` could miss bits that the writes inadvertently cleared.

19.7.4 What “Is It Mine?” Looks Like on Real Hardware

The `INTR_STATUS` test is textbook because the Chapter 17 register layout is textbook. Real devices come in flavours.

Devices with a clean `INTR_STATUS`. Most modern devices have a register that, when read as zero, definitively says “not mine”. The Chapter 19 driver's filter shape applies directly.

Devices with bits that are always set. Some devices have pending-interrupt bits that stay set across interrupts (waiting for the driver to reset them). The filter must mask these out or check against a per-interrupt-class mask. The Chapter 17 register layout avoids this complication; real drivers occasionally face it.

Devices with no `INTR_STATUS` at all. A few older devices require the driver to read a separate register sequence (or to infer from state registers) whether the interrupt is pending. These drivers are more complex; the filter may need to acquire a spin lock and read several registers. The FreeBSD source has examples in a few embedded drivers.

Devices with a global `INTR_STATUS` and per-source registers. A common pattern on NICs: a top-level register reports which queue has a pending event, and per-queue registers contain the event details. The filter reads the top-level register to decide ownership; the ithread or task reads the per-queue registers to process events.

Chapter 19's driver uses the first flavour. The discipline for the other flavours is the same: read a register, decide.

19.7.5 Returning `FILTER_STRAY` Correctly

The rule is simple: if the filter does not recognise any bit as belonging to a class it handles, return `FILTER_STRAY`.

```
if (rv == 0)  
    return (FILTER_STRAY);
```

The variable `rv` accumulates `FILTER_HANDLED` from each recognised bit. If no bit was recognised, `rv` is zero, and the filter has nothing to return but `FILTER_STRAY`.

A subtle corollary: a filter that recognises some bits but not others returns `FILTER_HANDLED` for the bits it recognised, and does not return `FILTER_STRAY` for the bits it did not. Setting a bit in `INTR_MASK` that the driver will not handle is a driver bug; the kernel cannot help.

An interesting edge case: a bit is set in `INTR_STATUS` but the driver does not recognise it (perhaps a new device revision added a bit the driver's code predates). The driver has two options:

1. Ignore the bit. Do not acknowledge it. Let it stay set. On a level-triggered line this produces a storm because the bit asserts the line forever. Bad.
2. Acknowledge the bit without doing any work. Write it back to `INTR_STATUS`. The device stops asserting for that bit, no storm, but the event is lost. On an essential event this is a functional bug; on a diagnostic event it may be acceptable.

The recommended pattern is option 2 with a log message: acknowledge unrecognised bits, log them at a reduced rate (to avoid log flooding if the bit is asserted continuously), and move on. This makes the driver robust against new hardware revisions at the cost of potentially losing information about unknown events.

```
uint32_t unknown = status & ~(MYFIRST_INTR_DATA_AV |
    MYFIRST_INTR_ERROR | MYFIRST_INTR_COMPLETE);
if (unknown != 0) {
    atomic_add_64(&sc->intr_unknown_count, 1);
    bus_write_4(sc->bar_res, MYFIRST_REG_INTR_STATUS, unknown);
    rv |= FILTER_HANDLED;
}
```

This snippet is not in Stage 2's filter; it is a useful extension for Stage 3 or beyond.

19.7.6 What Happens If Multiple Drivers Share an IRQ

A concrete scenario. Suppose the `virtio-rnd` device in a bhyve guest shares IRQ 19 with the AHCI controller. Both drivers have registered handlers. An interrupt arrives on IRQ 19.

The kernel walks the handler list in registration order. Suppose AHCI registered first, so its filter runs first:

1. AHCI filter: reads its `INTR_STATUS`, sees bits set (AHCI has pending I/O), acknowledges, returns `FILTER_HANDLED`.
2. `myfirst` filter: reads its `INTR_STATUS`, reads zero, returns `FILTER_STRAY`.

The kernel sees “at least one `FILTER_HANDLED`” and does not mark the interrupt as stray.

Now the reverse case. The `virtio-rnd` device has an event:

1. AHCI filter: reads its `INTR_STATUS`, sees zero, returns `FILTER_STRAY`.
2. `myfirst` filter: reads its `INTR_STATUS`, sees `DATA_AV`, acknowledges, returns `FILTER_HANDLED`.

The kernel sees one `FILTER_HANDLED` and is happy.

The key property is that each filter checks only its own device. No filter assumes the interrupt is its own; each decides from its device's state.

19.7.7 What Happens If a Driver Gets It Wrong

A broken AHCI filter that returns `FILTER_HANDLED` whenever it fires (without checking status) would claim our `myfirst` interrupt. The `myfirst` filter would never run, `DATA_AV` would never be acknowledged, and the line would storm.

The fix is not on the `myfirst` side; it is on the AHCI side. In reality, all major FreeBSD drivers get the check right because the code has been audited and tested for years. The lesson is that the shared-IRQ protocol requires cooperation: every driver on the line must check its own state correctly.

The protection against a single broken driver is `hw.intr_storm_threshold`. When the kernel detects a run of interrupts all marked as stray (or all returning `FILTER_HANDLED` without any device actually having work), it will eventually mask the line. The machinery is detection, not prevention.

19.7.8 Coexisting With Non-Sharing Drivers

A driver that allocates its IRQ with `RF_SHAREABLE` can coexist with drivers that allocate without sharing, provided the kernel can satisfy both requests. If our `myfirst` driver allocates first with `RF_SHAREABLE`, then AHCI tries to allocate exclusively, AHCI's allocation will fail (the line is already held by a driver that might not be exclusive). If AHCI allocates first without sharing, our `myfirst` allocation (with `RF_SHAREABLE`) will fail.

In practice, modern drivers almost always use `RF_SHAREABLE`. Legacy drivers occasionally omit it; if a reader's driver cannot be loaded because of an interrupt-allocation conflict, the fix is often to add `RF_SHAREABLE` to the allocation.

Exclusive allocation is appropriate for:

- Drivers with strict latency requirements that cannot tolerate other handlers on the line.
- Drivers that use `INTR_EXCL` for a specific kernel reason.
- Some legacy drivers that were written before shared-IRQ support matured.

For Chapter 19's driver, `RF_SHAREABLE` is the default and never wrong.

19.7.9 The bhyve Virtio IRQ Topology

A practical detail about the Chapter 19 lab environment. The bhyve emulator maps each emulated PCI device to an IRQ line based on the slot's `INTx` pin. Multiple devices on the same slot's different functions share a line; different slots usually have different lines. The `virtio-rnd` device at slot 4 function 0 has its own pin.

In practice, on a bhyve guest with just a handful of emulated devices, each device often has its own IRQ line (no sharing). The `myfirst` driver with `RF_SHAREABLE` allocated on a non-shared line behaves identically to a non-shareable allocation; the flag is harmless.

To deliberately test shared-IRQ behaviour in bhyve, a reader can stack multiple virtio devices onto the same slot (different functions), forcing them to share a line. This is advanced and not necessary for basic Chapter 19 labs.

19.7.10 The Starvation Concern

A shared-IRQ line has a potential starvation problem: a single driver that takes too long in its filter can delay every other driver on the line. Every filter sees its device's state as "unchanging" for the duration of the slow filter, and events can accumulate undetected.

The discipline is the same one Section 4 covered: filters must be fast. Tens or hundreds of microseconds of real work is usually the maximum a well-behaved filter does; anything slower moves to the task. A filter that does long work starves not only its own driver's upper layers but also every other driver on the line.

On MSI-X (Chapter 20), each vector has its own `intr_event`, so the starvation concern disappears for the specific driver pairs that use MSI-X. But the discipline still applies: a filter that takes a millisecond is hurting latency for every subsequent interrupt.

19.7.11 False Positives and Defensive Handling

A useful property of the status-register check is that it is naturally tolerant of false positives from the kernel side. Occasionally, interrupt controllers report a spurious interrupt when no device is actually asserting (noise on the line, a race between edge-trigger and masking, a platform-specific quirk). The kernel dispatches, the filter reads `INTR_STATUS`, it is zero, the filter returns `FILTER_STRAY`, the kernel moves on.

This is a no-op for the driver. The count of stray interrupts goes up; nothing else changes.

Some drivers add a rate-limited log message to make spurious interrupts visible. A reasonable default is to log only if the rate exceeds a threshold:

```
static struct timeval last_stray_log;
static int stray_rate_limit = 5; /* messages per second */
if (rv == 0) {
    if (ppsratecheck(&last_stray_log, &stray_rate_limit, 1))
        device_printf(sc->dev, "spurious interrupt\n");
    return (FILTER_STRAY);
}
```

The `ppsratecheck(9)` utility limits the message rate. Without it, a line that is storming would flood `dmesg` with identical messages.

Chapter 19's driver does not include the rate-limited log in its Stage 2 filter; it is added in a challenge exercise.

19.7.12 When the Filter Should Handle and the Task Should Not Run

A thought experiment. Imagine the filter recognises `ERROR` but not `DATA_AV`. The filter handles `ERROR` (acknowledges, increments counter) and returns `FILTER_HANDLED`. No task is queued. The device is satisfied; the line deasserts.

But `INTR_STATUS.DATA_AV` might still be set, because the filter did not acknowledge it (the filter did not recognise the bit as belonging to a class the driver handles). On a level-triggered line, the device keeps asserting for `DATA_AV`, a new interrupt fires, and the loop repeats.

This is a version of the “unknown bit storm” problem. The fix is to acknowledge every bit the driver is willing to see, even if the driver does not do anything with some of them. Setting `INTR_MASK` to only the bits the driver handles is the preventive measure; acknowledging unrecognised bits in the filter is the defensive measure.

19.7.13 Wrapping Up Section 6

Shared interrupts are the common case on legacy PCI and still the right assumption to write for on modern hardware. A filter on a shared line must check whether the interrupt belongs to its device (usually by reading the device’s `INTR_STATUS` register), handle the bits it recognises, acknowledge those bits, and return `FILTER_STRAY` if it recognised nothing. The discipline is small in code and large in reliability: a driver that gets it right coexists with every other well-behaved driver on its line, and `RF_SHAREABLE` in the allocation is the only additional line of code it needs.

Section 7 is the teardown section. It is short: `bus_tearardown_intr` before `bus_release_resource`, drain the taskqueue before anything the task touches is freed, clear `INTR_MASK` so the device stops asserting, and verify the counters make sense. But the ordering is strict, and Chapter 19’s detach path extends by exactly these steps.

19.8 Section 7: Cleaning Up Interrupt Resources

The attach path gained three new operations (allocate IRQ, register handler, enable interrupts at the device); the detach path must undo each of them in strict reverse order. Section 7 is short because the pattern is now familiar; but the order matters in specific ways the previous sections did not touch on, and a mistake here produces kernel panics that the debug kernel is very good at catching and very good at making confusing to diagnose.

19.8.1 The Required Order

From most-specific-to-most-general, the detach sequence at Chapter 19 Stage 2 is:

1. **Refuse if busy.** `myfirst_is_busy(sc)` returns true if the cdev is open or a command is in flight.
2. **Mark as no longer attached** so user-space paths refuse to start.
3. **Destroy the cdev** so no new user-space access begins.
4. **Disable interrupts at the device.** Clear `INTR_MASK` so the device stops asserting.
5. **Tear down the interrupt handler.** `bus_tearardown_intr(9)` on `irq_res` with the saved cookie. After this returns, the kernel guarantees the filter will not run again.
6. **Drain the taskqueue.** `taskqueue_drain(9)` waits for any pending task to complete and prevents new ones from starting.
7. **Destroy the taskqueue.** `taskqueue_free(9)` shuts down the worker threads.
8. **Quiesce Chapter 17 simulation callouts** if `sc->sim` is non-NULL.
9. **Detach the Chapter 17 simulation** if attached.
10. **Detach the hardware layer** so `sc->hw` is freed.
11. **Release the IRQ resource** with `bus_release_resource(9)`.

12. **Release the BAR** with `bus_release_resource(9)`.

13. **Deinit the softc.**

Thirteen steps. Each does one thing. The hazards are in the ordering.

19.8.2 Why Disable at the Device Before `bus_tearardown_intr`

Clearing `INTR_MASK` before tearing down the handler is defensive. If we tore down the handler first, a pending interrupt at the device might fire and have no handler; the kernel would mark it as stray and eventually disable the line. Clearing `INTR_MASK` first stops the device from asserting, then the teardown removes the handler, and no interrupts can fire in between.

For MSI-X (Chapter 20), the logic is slightly different because each vector is independent. But the principle transfers: stop the source before removing the handler.

On real hardware this window is microseconds; a stray interrupt during it is rare. On bhyve, where the event rate is low, it essentially never happens. But careful drivers close the window anyway because careful drivers are the ones you want to read in production.

19.8.3 Why `bus_tearardown_intr` Before Releasing the Resource

`bus_tearardown_intr` removes the driver's handler from the `intr_event`. After it returns, the kernel guarantees the filter will not run again. But the IRQ resource (the `struct resource *`) is still valid; the kernel has not freed it. `bus_release_resource` is what frees it.

If we released the resource first, the kernel's internal bookkeeping around the `intr_event` would see a handler registered against a resource that no longer exists. Depending on timing, this produces either an immediate failure during `bus_release_resource` (the kernel detects the handler is still attached) or a delayed problem later when the line tries to fire.

The safe ordering is always `bus_tearardown_intr` first. The man page for `bus_setup_intr(9)` makes this explicit.

19.8.4 Why Drain the Taskqueue Before Freeing the Softc

The filter may have enqueued a task that has not yet run. The task's function pointer is stored in the `struct task`, and the argument pointer is the `softc`. If we freed the `softc` before the task ran, the task would dereference a freed pointer and panic.

`taskqueue_drain(9)` on a specific task waits for that task to complete and prevents future enqueues of that task from running. Calling `taskqueue_drain` on `&sc->intr_data_task` is the exact right thing: it waits for the data-available task to finish.

After `taskqueue_drain` returns, no task run is in progress. The `softc` can be freed safely.

A common mistake: draining a single task with `taskqueue_drain(tq, &task)` is different from draining the whole taskqueue with `taskqueue_drain_all(tq)`. For a driver with multiple tasks on the same taskqueue, each task needs its own drain, or `taskqueue_drain_all` handles them as a group.

For Chapter 19's driver, there is one task, so a single `taskqueue_drain` suffices.

19.8.5 Why bus_tearardown_intr Before taskqueue_drain

The filter may still enqueue a task between the time INTR_MASK was cleared and the time bus_tearardown_intr returns. If we drained the taskqueue before tearing down the handler, a still-running filter could enqueue a task after the drain, and the drain's guarantee would be violated.

The correct order is: clear INTR_MASK (stops new interrupts), tear down the handler (stops the filter from running again), drain the taskqueue (stops any previously enqueued task from running). Each step narrows the set of code paths that can touch the state.

19.8.6 The Cleanup Code

Putting the order into the Chapter 19 Stage 2 detach:

```
static int
myfirst_pci_detach(device_t dev)
{
    struct myfirst_softc *sc = device_get_softc(dev);

    if (myfirst_is_busy(sc))
        return (EBUSY);

    sc->pci_attached = false;

    /* Destroy the cdev so no new user-space access starts. */
    if (sc->cdev != NULL) {
        destroy_dev(sc->cdev);
        sc->cdev = NULL;
    }

    /* Disable interrupts at the device. */
    MYFIRST_LOCK(sc);
    if (sc->hw != NULL && sc->bar_res != NULL)
        bus_write_4(sc->bar_res, MYFIRST_REG_INTR_MASK, 0);
    MYFIRST_UNLOCK(sc);

    /* Tear down the interrupt handler. */
    if (sc->intr_cookie != NULL) {
        bus_tearardown_intr(dev, sc->irq_res, sc->intr_cookie);
        sc->intr_cookie = NULL;
    }

    /* Drain and destroy the interrupt taskqueue. */
    if (sc->intr_tq != NULL) {
        taskqueue_drain(sc->intr_tq, &sc->intr_data_task);
    }
}
```

```
    taskqueue_free(sc->intr_tq);
    sc->intr_tq = NULL;
}

/* Quiesce Chapter 17 callouts (if sim attached). */
myfirst_quiesce(sc);

/* Detach Chapter 17 simulation if attached. */
if (sc->sim != NULL)
    myfirst_sim_detach(sc);

/* Detach the hardware layer. */
myfirst_hw_detach(sc);

/* Release the IRQ resource. */
if (sc->irq_res != NULL) {
    bus_release_resource(dev, SYS_RES_IRQ, sc->irq_rid,
        sc->irq_res);
    sc->irq_res = NULL;
}

/* Release the BAR. */
if (sc->bar_res != NULL) {
    bus_release_resource(dev, SYS_RES_MEMORY, sc->bar_rid,
        sc->bar_res);
    sc->bar_res = NULL;
}

myfirst_deinit_softc(sc);

device_printf(dev, "detached\n");
return (0);
}
```

Thirteen distinct actions, each simple. The code is longer than earlier stages only because each new capability adds its own teardown step.

19.8.7 Handling Partial-Attach Failure

The attach path's goto cascade in Section 3 had labels for each allocation step. With the interrupt handler registered at Stage 2, the cascade grows one more:

```
fail_teardown_intr:
    MYFIRST_LOCK(sc);
    if (sc->hw != NULL && sc->bar_res != NULL)
        bus_write_4(sc->bar_res, MYFIRST_REG_INTR_MASK, 0);
    MYFIRST_UNLOCK(sc);
    bus_teardown_intr(dev, sc->irq_res, sc->intr_cookie);
    sc->intr_cookie = NULL;
fail_release_irq:
    bus_release_resource(dev, SYS_RES_IRQ, sc->irq_rid, sc->irq_res);
    sc->irq_res = NULL;
fail_hw:
    myfirst_hw_detach(sc);
fail_release_bar:
    bus_release_resource(dev, SYS_RES_MEMORY, sc->bar_rid, sc->bar_res);
    sc->bar_res = NULL;
fail_softc:
    myfirst_deinit_softc(sc);
    return (error);
```

Each cascade label undoes the step that succeeded before it. A failed `bus_setup_intr` jumps to `fail_release_irq` (skips teardown because the handler was not registered). A failed `make_dev` (cdev creation) jumps to `fail_teardown_intr` (tears down the handler before releasing the IRQ).

The taskqueue is initialised in `myfirst_init_softc` and destroyed in `myfirst_deinit_softc`, so the cascade does not need to handle it explicitly; whichever label reaches `fail_softc` gets the taskqueue cleaned up via `deinit`.

19.8.8 Verifying the Teardown

After `kldunload myfirst` the kernel should be in a clean state. Specific checks:

- `kldstat -v | grep myfirst` returns nothing (module unloaded).
- `devinfo -v | grep myfirst` returns nothing (device detached).
- `vmstat -i | grep myfirst` returns nothing (interrupt event cleaned up).
- `vmstat -m | grep myfirst` returns nothing or shows zero InUse (malloc type drained).
- `dmesg | tail` shows the detach banner and no warnings or panics.

Failing any of these is a bug. The most common failure is `vmstat -i` showing a stale entry; this usually means `bus_teardown_intr` was not called. The second most common is `vmstat -m` showing live allocations; this usually means a task was enqueued and not drained, or the simulation was attached and not detached.

19.8.9 Handling the “Handler Fired During Detach” Case

A subtle case worth thinking through. Suppose a real interrupt fires on a shared IRQ line between the cdev destruction and the `INTR_MASK` write. Another driver’s device is asserting, our filter runs (because the line is

shared), our filter reads `INTR_STATUS` (which is zero on our device), and returns `FILTER_STRAY`. No state is touched, no task is enqueued.

Suppose instead that the interrupt is from our device. Our `INTR_STATUS` has a bit set. The filter recognises it, acknowledges, enqueues a task, returns. The task enqueue happens against a taskqueue that has not yet been drained. The task runs later, acquires `sc->mtx`, reads `DATA_OUT` through the hardware layer (which is still attached because we haven't called `myfirst_hw_detach` yet). All safe.

Suppose the interrupt arrives after `INTR_MASK = 0` but before `bus_tear_down_intr`. The device has stopped asserting for the bits we cleared, but an already-in-flight interrupt (queued in the interrupt controller) can still run the filter. The filter reads `INTR_STATUS`, sees zero (because the mask write was ahead of the device's internal state), returns `FILTER_STRAY`. The interrupt is counted as stray; the kernel ignores it.

Suppose the interrupt arrives after `bus_tear_down_intr`. The handler is gone. The kernel's stray-interrupt accounting notices. After enough strays, the kernel disables the line. This is the scenario the `INTR_MASK = 0` step is designed to prevent; if the mask is cleared first, no strays can accumulate.

The code paths are all defensive. The debug kernel's assertions catch the common mistakes. A driver that follows the Chapter 19 order reliably tears down cleanly.

19.8.10 What Goes Wrong When Teardown Is Skipped

A few scenarios that produce concrete symptoms.

Handler not torn down, resource released. `kldunload` calls `bus_release_resource` on the IRQ without `bus_tear_down_intr`. The kernel detects an active handler on a resource being released and panics with a message like “releasing allocated IRQ with active handler”. The debug kernel is reliable here.

Handler torn down, taskqueue not drained. The task is enqueued in the filter, the filter's last call happens just before teardown, the task has not yet run. The driver frees `sc` (via `softc_deinit`) and unloads. The task's worker thread wakes up, runs the task function, dereferences the freed `softc`, panics with a null-pointer or use-after-free fault. The debug kernel's `WITNESS` or `MEMGUARD` may catch it; if not, the crash is at the task function's first memory access.

Taskqueue drained, not freed. `taskqueue_drain` succeeds, but `taskqueue_free` is skipped. The taskqueue's worker thread keeps running (idle). A `vmstat -m` shows the allocation. Not a functional bug, but a leak that accumulates across load-unload cycles.

Simulation callouts not quiesced. If the Chapter 17 simulation is attached (on a simulation-only build), its callouts are running. Without quiescing, they fire after detach has freed the register block, and access garbage. The `WITNESS` or `MEMGUARD` catch varies by hit; sometimes a plain null dereference is the symptom.

INTR_MASK not cleared. Real interrupts fire after detach starts. The filter (briefly, until teardown) handles them; after teardown, they are strays that the kernel eventually disables the line for. The line's disabled state is visible in `vmstat -i` (growing stray count) and in `dmesg` (kernel warnings).

Each of these is recoverable by fixing the teardown order. The Chapter 19 code is set up correctly; the hazards are for a reader who modifies the order.

19.8.11 Sanity Testing the Teardown

A simple sanity test the reader can run after writing the detach code:

```
# Load.
sudo kldload ./myfirst.ko

# Fire a few simulated interrupts, make sure tasks run.
for i in 1 2 3 4 5; do
    sudo sysctl dev.myfirst.0.intr_simulate=1
done
sleep 1
sysctl dev.myfirst.0.intr_task_invocations # should be 5

# Unload.
sudo kldunload myfirst

# Check nothing leaked.
vmstat -m | grep myfirst # should be empty
devinfo -v | grep myfirst # should be empty
vmstat -i | grep myfirst # should be empty
```

Running this sequence in a loop (twenty iterations in a shell loop) is a reasonable regression test: any leak accumulates, any crash manifests, any failure pattern becomes visible.

19.8.12 Wrapping Up Section 7

Cleaning up interrupt resources is six small operations in the detach path: disable `INTR_MASK`, tear down the handler, drain and free the taskqueue, detach the hardware layer, release the IRQ, release the BAR. Each operation undoes exactly one attach-path operation. The order is the inverse of attach. The taskqueue drain is an important new concern specific to filter-plus-task drivers; a driver that skips it has a use-after-free bug waiting for the next load-unload cycle.

Section 8 is the housekeeping section: split the interrupt code into its own file, bump the version to `1.2-intr`, write `INTERRUPTS.md`, and run the regression pass. The driver is functionally complete after Section 7; Section 8 makes it maintainable.

19.9 Section 8: Refactoring and Versioning Your Interrupt-Ready Driver

The interrupt handler is working. Section 8 is the housekeeping section. It splits the interrupt code into its own file, updates the module metadata, adds a new `INTERRUPTS.md` document, introduces a small set of interrupt-context CSR macros so the filter can access registers without the lock-asserting macros, bumps the version to `1.2-intr`, and runs the regression pass.

A reader who has made it this far may again be tempted to skip this section. It is the same temptation Chapter 18's Section 8 warned against, and the same refusal: a driver whose interrupt code is mixed into the PCI file, whose filter uses `raw_bus_read_4` ad hoc, whose taskqueue setup is split across three files, becomes painful to extend. Chapter 20 adds MSI and MSI-X; Chapter 21 adds DMA. Both of them build on Chapter 19's interrupt code. A clean structure now saves effort across both.

19.9.1 The Final File Layout

At the end of Chapter 19, the driver consists of these files:

<code>myfirst.c</code>	- Main driver: <code>softc</code> , <code>cdev</code> , module events, data path.
<code>myfirst.h</code>	- Shared declarations: <code>softc</code> , lock macros, prototypes.
<code>myfirst_hw.c</code>	- Ch16 hardware access layer: <code>CSR_*</code> accessors, access log, <code>sysctl</code> handlers.
<code>myfirst_hw_pci.c</code>	- Ch18 hardware layer extension: <code>myfirst_hw_attach_pci</code> .
<code>myfirst_hw.h</code>	- Register map and accessor declarations.
<code>myfirst_sim.c</code>	- Ch17 simulation backend.
<code>myfirst_sim.h</code>	- Ch17 simulation interface.
<code>myfirst_pci.c</code>	- Ch18 PCI attach: <code>probe</code> , <code>attach</code> , <code>detach</code> , <code>DRIVER_MODULE</code> , <code>MODULE_DEPEND</code> , ID table.
<code>myfirst_pci.h</code>	- Ch18 PCI declarations.
<code>myfirst_intr.c</code>	- Ch19 interrupt handler: <code>filter</code> , <code>task</code> , <code>setup</code> , <code>teardown</code> .
<code>myfirst_intr.h</code>	- Ch19 interrupt interface.
<code>myfirst_sync.h</code>	- Part 3 synchronisation primitives.
<code>cbuf.c</code> / <code>cbuf.h</code>	- Ch10 circular buffer.
<code>Makefile</code>	- <code>kmod</code> build.
<code>HARDWARE.md</code>	- Ch16/17 register map.
<code>LOCKING.md</code>	- Ch15 onward lock discipline.
<code>SIMULATION.md</code>	- Ch17 simulation.
<code>PCI.md</code>	- Ch18 PCI support.
<code>INTERRUPTS.md</code>	- Ch19 interrupt handling.

`myfirst_intr.c` and `myfirst_intr.h` are new. `INTERRUPTS.md` is new. Every other file either existed before or was extended slightly (the `softc` gained fields; the PCI attach calls into `myfirst_intr.c`).

The rule of thumb remains: each file has one responsibility. `myfirst_intr.c` owns the interrupt handler, the deferred task, and the simulated-interrupt `sysctl`. `myfirst_pci.c` owns the PCI attach but delegates interrupt setup and teardown to functions exported by `myfirst_intr.c`.

19.9.2 The Final Makefile

```
# Makefile for the Chapter 19 myfirst driver.
```

```

KMOD=  myfirst
SRCS=  myfirst.c \
      myfirst_hw.c myfirst_hw_pci.c \
      myfirst_sim.c \
      myfirst_pci.c \
      myfirst_intr.c \
      cbuf.c

CFLAGS+= -DMYFIRST_VERSION_STRING=\"1.2-intr\"

# CFLAGS+= -DMYFIRST_SIMULATION_ONLY
# CFLAGS+= -DMYFIRST_PCI_ONLY

#include <bsd.kmod.mk>

```

One additional source file in the SRCS list; the version string bumped; the rest unchanged.

19.9.3 The Version String

1.1-pci to 1.2-intr. The bump reflects that the driver has acquired a significant new capability (interrupt handling) without changing any user-visible interface (the cdev still does what it did). A minor-version bump is appropriate.

Later chapters continue: 1.3-msi after Chapter 20's MSI and MSI-X work; 1.4-dma after Chapters 20 and 21 add DMA. Each minor version reflects one significant capability addition.

19.9.4 The myfirst_intr.h Header

The header exports the interrupt layer's public interface to the rest of the driver:

```

#ifndef _MYFIRST_INTR_H_
#define _MYFIRST_INTR_H_

#include <sys/types.h>
#include <sys/taskqueue.h>

struct myfirst_softc;

/* Interrupt setup and teardown, called from the PCI attach path. */
int myfirst_intr_setup(struct myfirst_softc *sc);
void myfirst_intr_teardown(struct myfirst_softc *sc);

/* Register sysctl nodes specific to the interrupt layer. */
void myfirst_intr_add_sysctls(struct myfirst_softc *sc);

```



```

/* Interrupt-context accessor macros. These do not acquire sc->mtx
 * and therefore are safe in the filter. They are NOT a replacement
 * for CSR_READ_4 / CSR_WRITE_4 in other contexts. */
#define ICSR_READ_4(sc, off) \
    bus_read_4((sc)->bar_res, (off))
#define ICSR_WRITE_4(sc, off, val) \
    bus_write_4((sc)->bar_res, (off), (val))

#endif /* _MYFIRST_INTR_H_ */

```

The public API is three functions (`myfirst_intr_setup`, `myfirst_intr_teardown`, `myfirst_intr_add_sysctls`) and two accessor macros (`ICSR_READ_4`, `ICSR_WRITE_4`). The “I” prefix is for “interrupt-context”; these macros do not take `sc->mtx`, so they are safe in the filter.

19.9.5 The `myfirst_intr.c` File

The full file is in the companion examples tree; here is the core structure:

```

#include <sys/param.h>
#include <sys/system.h>
#include <sys/kernel.h>
#include <sys/bus.h>
#include <sys/lock.h>
#include <sys/mutex.h>
#include <sys/sysctl.h>
#include <sys/taskqueue.h>
#include <sys/rman.h>

#include <machine/bus.h>
#include <machine/resource.h>

#include "myfirst.h"
#include "myfirst_hw.h"
#include "myfirst_intr.h"

/* Deferred task for data-available events. */
static void myfirst_intr_data_task_fn(void *arg, int npending);

/* The filter handler. Exported so the simulated-interrupt sysctl can
 * call it directly. */
int myfirst_intr_filter(void *arg);

int

```

```

myfirst_intr_setup(struct myfirst_softc *sc)
{
    int error;

    TASK_INIT(&sc->intr_data_task, 0, myfirst_intr_data_task_fn, sc);
    sc->intr_tq = taskqueue_create("myfirst_intr", M_WAITOK,
        taskqueue_thread_enqueue, &sc->intr_tq);
    taskqueue_start_threads(&sc->intr_tq, 1, PI_NET,
        "myfirst intr taskq");

    sc->irq_rid = 0;
    sc->irq_res = bus_alloc_resource_any(sc->dev, SYS_RES_IRQ,
        &sc->irq_rid, RF_SHAREABLE | RF_ACTIVE);
    if (sc->irq_res == NULL)
        return (ENXIO);

    error = bus_setup_intr(sc->dev, sc->irq_res,
        INTR_TYPE_MISC | INTR_MPSAFE,
        myfirst_intr_filter, NULL, sc,
        &sc->intr_cookie);
    if (error != 0) {
        bus_release_resource(sc->dev, SYS_RES_IRQ, sc->irq_rid,
            sc->irq_res);
        sc->irq_res = NULL;
        return (error);
    }

    bus_describe_intr(sc->dev, sc->irq_res, sc->intr_cookie, "legacy");

    /* Enable the interrupts we care about at the device. */
    MYFIRST_LOCK(sc);
    if (sc->hw != NULL)
        CSR_WRITE_4(sc, MYFIRST_REG_INTR_MASK,
            MYFIRST_INTR_DATA_AV | MYFIRST_INTR_ERROR |
            MYFIRST_INTR_COMPLETE);
    MYFIRST_UNLOCK(sc);

    return (0);
}

void
myfirst_intr_teardown(struct myfirst_softc *sc)
{
    /* Disable interrupts at the device. */

```

```
MYFIRST_LOCK(sc);
if (sc->hw != NULL && sc->bar_res != NULL)
    CSR_WRITE_4(sc, MYFIRST_REG_INTR_MASK, 0);
MYFIRST_UNLOCK(sc);

/* Tear down the handler. */
if (sc->intr_cookie != NULL) {
    bus_teardown_intr(sc->dev, sc->irq_res, sc->intr_cookie);
    sc->intr_cookie = NULL;
}

/* Drain and destroy the taskqueue. */
if (sc->intr_tq != NULL) {
    taskqueue_drain(sc->intr_tq, &sc->intr_data_task);
    taskqueue_free(sc->intr_tq);
    sc->intr_tq = NULL;
}

/* Release the IRQ resource. */
if (sc->irq_res != NULL) {
    bus_release_resource(sc->dev, SYS_RES_IRQ, sc->irq_rid,
        sc->irq_res);
    sc->irq_res = NULL;
}
}

int
myfirst_intr_filter(void *arg)
{
    /* ... as in Section 4 ... */
}

static void
myfirst_intr_data_task_fn(void *arg, int npending)
{
    /* ... as in Section 4 ... */
}

void
myfirst_intr_add_sysctls(struct myfirst_softc *sc)
{
    /* ... counters and intr_simulate sysctl ... */
}
```

The file is about 250 lines at Stage 4. `myfirst_pci.c` shrinks correspondingly: the interrupt allocation and setup move out.

19.9.6 The Refactored PCI Attach

After moving interrupt code into `myfirst_intr.c`, `myfirst_pci_attach` becomes:

```
static int
myfirst_pci_attach(device_t dev)
{
    struct myfirst_softc *sc = device_get_softc(dev);
    int error;

    sc->dev = dev;
    sc->unit = device_get_unit(dev);
    error = myfirst_init_softc(sc);
    if (error != 0)
        return (error);

    /* Step 1: allocate BAR 0. */
    sc->bar_rid = PCIR_BAR(0);
    sc->bar_res = bus_alloc_resource_any(dev, SYS_RES_MEMORY,
        &sc->bar_rid, RF_ACTIVE);
    if (sc->bar_res == NULL) {
        device_printf(dev, "cannot allocate BAR0\n");
        error = ENXIO;
        goto fail_softc;
    }

    /* Step 2: attach the hardware layer. */
    error = myfirst_hw_attach_pci(sc, sc->bar_res,
        rman_get_size(sc->bar_res));
    if (error != 0)
        goto fail_release_bar;

    /* Step 3: set up interrupts. */
    error = myfirst_intr_setup(sc);
    if (error != 0) {
        device_printf(dev, "interrupt setup failed (%d)\n", error);
        goto fail_hw;
    }

    /* Step 4: create cdev. */
    sc->cdev = make_dev(&myfirst_cdevsw, sc->unit, UID_ROOT,
        GID_WHEEL, 0600, "myfirst%d", sc->unit);
}
```

```

    if (sc->cdev == NULL) {
        error = ENXIO;
        goto fail_intr;
    }
    sc->cdev->si_drv1 = sc;

    /* Step 5: register sysctls. */
    myfirst_intr_add_sysctls(sc);

    sc->pci_attached = true;
    return (0);

fail_intr:
    myfirst_intr_tear_down(sc);
fail_hw:
    myfirst_hw_detach(sc);
fail_release_bar:
    bus_release_resource(dev, SYS_RES_MEMORY, sc->bar_rid, sc->bar_res);
    sc->bar_res = NULL;
fail_softc:
    myfirst_deinit_softc(sc);
    return (error);
}

```

The PCI attach is shorter; the interrupt details are hidden behind `myfirst_intr_setup`. The goto cascade is four labels instead of six (the interrupt-specific labels moved into `myfirst_intr.c`).

19.9.7 The Refactored Detach

```

static int
myfirst_pci_detach(device_t dev)
{
    struct myfirst_softc *sc = device_get_softc(dev);

    if (myfirst_is_busy(sc))
        return (EBUSY);

    sc->pci_attached = false;

    if (sc->cdev != NULL) {
        destroy_dev(sc->cdev);
        sc->cdev = NULL;
    }
}

```

```

myfirst_intr_teardown(sc);

if (sc->sim != NULL)
    myfirst_sim_detach(sc);

myfirst_hw_detach(sc);

if (sc->bar_res != NULL) {
    bus_release_resource(dev, SYS_RES_MEMORY, sc->bar_rid,
        sc->bar_res);
    sc->bar_res = NULL;
}

myfirst_deinit_softc(sc);

device_printf(dev, "detached\n");
return (0);
}

```

The interrupt-specific teardown is one call to `myfirst_intr_teardown`, which encapsulates the mask-clear, teardown, drain, and resource-release steps.

19.9.8 The INTERRUPTS.md Document

The new document lives next to the driver source. Its role is to describe the driver's interrupt handling to a future reader without requiring them to read `myfirst_intr.c`:

Interrupt Handling in the myfirst Driver

Allocation and Setup

The driver allocates a single legacy PCI IRQ through ``bus_alloc_resource_any(9)`` with ``SYS_RES_IRQ``, ``rid = 0``, ``RF_SHAREABLE | RF_ACTIVE``. The filter handler is registered through ``bus_setup_intr(9)`` with ``INTR_TYPE_MISC | INTR_MPSAFE``. A taskqueue named "myfirst_intr" is created with one worker thread at ``PI_NET`` priority.

On successful setup, ``INTR_MASK`` is written with ``DATA_AV | ERROR | COMPLETE`` so the device will assert the line for those three event classes.

Filter Handler

``myfirst_intr_filter(sc)`` reads ``INTR_STATUS``. If zero, it returns ``FILTER_STRAY`` (shared-IRQ defence). Otherwise it inspects each of the three recognised bits, increments a per-bit counter atomically, writes the bit back to ``INTR_STATUS`` to acknowledge the device, and (for ``DATA_AV``) enqueues ``intr_data_task`` on the taskqueue.

The filter returns ``FILTER_HANDLED`` if any bit was recognised, or ``FILTER_STRAY`` otherwise.

Deferred Task

``myfirst_intr_data_task_fn(sc, npending)`` runs in thread context on the taskqueue's worker thread. It acquires ``sc->mtx``, reads ``DATA_OUT``, stores the value in ``sc->intr_last_data``, broadcasts ``sc->data_cv`` to wake pending readers, and releases the lock.

Simulated Interrupt sysctl

``dev.myfirst.N.intr_simulate`` is write-only; writing a bitmask to it sets the corresponding bits in ``INTR_STATUS`` and invokes ``myfirst_intr_filter`` directly. This exercises the full pipeline without needing real IRQ events.

Teardown

``myfirst_intr_teardown(sc)`` runs during detach. It clears ``INTR_MASK``, calls ``bus_teardown_intr``, drains and destroys the taskqueue, and releases the IRQ resource. The order is strict: mask-clear before teardown (so strays do not accumulate), teardown before drain (so no new task enqueues happen), drain before free (so no task runs against freed state).

Interrupt-Context Accessor Macros

Since the filter runs in primary interrupt context, it cannot take ``sc->mtx``. Two macros in ``myfirst_intr.h`` hide the raw ``bus_read_4``/``bus_write_4`` calls without asserting any lock: ``ICSR_READ_4`` and ``ICSR_WRITE_4``. Use them only in contexts where a sleep lock would be illegal.

Known Limitations

- Only the legacy PCI INTx line is handled. MSI and MSI-X are

Chapter 20.

- The filter coalesces per-bit counters via atomic ops; the task runs at a single priority. Per-queue or per-priority designs are later-chapter topics.
- Interrupt storm detection is managed by the kernel (``hw.intr_storm_threshold``); the driver does not implement its own storm mitigation.
- Chapter 17 simulation callouts are not active on the PCI build; the simulated-interrupt sysctl is the way to drive the pipeline on a bhyve lab target.

Five minutes to read; a clear picture of the interrupt layer's shape.

19.9.9 The Regression Pass

The Chapter 19 regression is a superset of Chapter 18's:

1. Compile cleanly. `make` succeeds; no warnings.
2. Load. `kldload ./myfirst.ko` succeeds; `dmesg` shows the attach sequence.
3. Attach to a real PCI device. `devinfo -v` shows the BAR and the IRQ.
4. No `[GIANT-LOCKED]` warning.
5. `vmstat -i | grep myfirst` shows the `intr_event`.
6. `sysctl dev.myfirst.0.intr_count` starts at zero.
7. Simulated interrupt. `sudo sysctl dev.myfirst.0.intr_simulate=1`; counters increment; task runs.
8. Rate test. Set `intr_sim_period_ms` to 100; check counters after 10 seconds.
9. Detach. `devctl detach myfirst0`; `dmesg` shows clean detach.
10. Reattach. `devctl attach pci0:0:4:0`; full attach cycle runs.
11. Unload. `kldunload myfirst`; `vmstat -m | grep myfirst` shows zero live allocations; `vmstat -i | grep myfirst` returns nothing.

Running the full regression takes a minute or two per iteration. A CI job that runs it twenty times in a loop is the kind of guard that catches regressions introduced by Chapter 20's and Chapter 21's extensions.

19.9.10 What the Refactor Accomplished

The Chapter 19 code is one file smaller than it would have been without the refactor; one new document exists; the version number has moved forward by one. The driver is recognisably FreeBSD, structurally parallel to a production driver in `/usr/src/sys/dev/`, and ready to accept Chapter 20's MSI-X machinery and Chapter 21's DMA machinery without another reorganisation.

19.9.11 Wrapping Up Section 8

The refactor follows the same shape Chapters 16 through 18 established. A new file owns the new responsibility. A new header exports the public interface. A new document explains the behaviour. The version bumps; the

regression passes; the driver stays maintainable. Nothing dramatic; a little housekeeping; a clean codebase to build on.

The instructional body of Chapter 19 is complete. Labs, challenges, troubleshooting, a wrap-up, and the bridge to Chapter 20 follow.

19.10 Reading a Real Driver Together: mgb(4)'s Interrupt Path

Before the labs, a short walk through a real driver that uses the same filter-plus-task pattern Chapter 19 teaches. `/usr/src/sys/dev/mgb/if_mgb.c` is the FreeBSD driver for Microchip's LAN743x gigabit Ethernet controllers. It is readable, it is production-quality, and its interrupt handling is about the level of complexity Chapter 19's vocabulary covers.

This section walks through the interrupt-relevant parts of `mgb_legacy_intr` and the setup code, and flags where each piece corresponds to a Chapter 19 concept.

19.10.1 The Legacy Filter

The filter handler for `mgb(4)`'s legacy-IRQ path:

```
int
mgb_legacy_intr(void *xsc)
{
    struct mgb_softc *sc;
    if_softc_ctx_t scctx;
    uint32_t intr_sts, intr_en;
    int qidx;

    sc = xsc;
    scctx = iflib_get_softc_ctx(sc->ctx);

    intr_sts = CSR_READ_REG(sc, MGB_INTR_STS);
    intr_en = CSR_READ_REG(sc, MGB_INTR_ENBL_SET);
    intr_sts &= intr_en;

    /* TODO: shouldn't continue if suspended */
    if ((intr_sts & MGB_INTR_STS_ANY) == 0)
        return (FILTER_STRAY);

    if ((intr_sts & MGB_INTR_STS_TEST) != 0) {
        sc->isr_test_flag = true;
        CSR_WRITE_REG(sc, MGB_INTR_STS, MGB_INTR_STS_TEST);
        return (FILTER_HANDLED);
    }
    if ((intr_sts & MGB_INTR_STS_RX_ANY) != 0) {
```

```

    for (qidx = 0; qidx < scctx->isc_nrxqsets; qidx++) {
        if ((intr_sts & MGB_INTR_STS_RX(qidx))) {
            iflib_rx_intr_deferred(sc->ctx, qidx);
        }
    }
    return (FILTER_HANDLED);
}

if ((intr_sts & MGB_INTR_STS_TX_ANY) != 0) {
    for (qidx = 0; qidx < scctx->isc_ntxqsets; qidx++) {
        if ((intr_sts & MGB_INTR_STS_RX(qidx))) {
            CSR_WRITE_REG(sc, MGB_INTR_ENBL_CLR,
                MGB_INTR_STS_TX(qidx));
            CSR_WRITE_REG(sc, MGB_INTR_STS,
                MGB_INTR_STS_TX(qidx));
            iflib_tx_intr_deferred(sc->ctx, qidx);
        }
    }
    return (FILTER_HANDLED);
}

return (FILTER_SCHEDULE_THREAD);
}

```

Walk through it. The filter reads two registers (INTR_STS and INTR_ENBL_SET), ANDs them to get the subset of pending interrupts that are enabled, and checks whether any bit is set. If none, it returns FILTER_STRAY, the Chapter 19 discipline for shared IRQs.

For each class of interrupt (test, receive, transmit), the filter acknowledges the relevant bits in INTR_STS (by writing them back) and schedules deferred processing. `iflib_rx_intr_deferred` is the `iflib` framework's way of scheduling receive-queue work; conceptually it is the same as Chapter 19's `taskqueue_enqueue`.

A line worth noting: the test-interrupt handler writes INTR_STS but also sets a flag (`sc->isr_test_flag = true`). This is the driver's way of signalling user-space code (via a `sysctl` or `ioctl`) that a test interrupt fired. Chapter 19's equivalent is the `intr_count` counter.

The last return is `FILTER_SCHEDULE_THREAD`. This fires if none of the specific bit classes matched but `MGB_INTR_STS_ANY` did. The `ithread` handles the residual case. Chapter 19's driver does not have this particular fall-through because it does not have a registered `ithread`; `mgb(4)` does.

19.10.2 What the `mgb` Filter Teaches

Three lessons transfer directly to Chapter 19's filter:

1. **Read-AND-mask.** `intr_sts & intr_en` ensures the filter only reports interrupts that were actually enabled. A device may internally report events the driver has masked out; the AND filters them.
2. **Acknowledge per-bit.** Each bit class is acknowledged individually (by writing the specific bits back). The filter does not write `0xffffffff`; it writes only the bits it handled.

3. **Defer per-queue work.** Each receive queue and transmit queue has its own deferred path. Chapter 19's simpler driver has one task; `mgb(4)`'s multi-queue driver has many.

19.10.3 Interrupt Setup in `mgb`

Searching for `bus_setup_intr` in `if_mgb.c` shows several call sites, one for the legacy IRQ path and one for each MSI-X vector:

```
if (bus_setup_intr(sc->dev, sc->irq[0], INTR_TYPE_NET | INTR_MPSAFE,
    mgb_legacy_intr, NULL, sc, &sc->irq_tag[0]) != 0) {
    /* ... */
}
```

The pattern is exactly Chapter 19's: filter handler, no `ithread`, `INTR_MPSAFE`, `softc` as argument, cookie returned. The only difference is `INTR_TYPE_NET` instead of `INTR_TYPE_MISC` (the driver targets networking).

19.10.4 Interrupt Teardown in `mgb`

The teardown pattern is distributed across `iflib(9)`'s helpers, which handle the drain and release for the driver. A bespoke driver outside the `iflib` framework does the teardown explicitly; Chapter 19's driver does it explicitly.

19.10.5 What the Walkthrough Teaches

`mgb(4)`'s interrupt path is not a toy. It is a production-quality implementation of the same pattern Chapter 19's driver follows. A reader who can read `mgb_legacy_intr` with comprehension has internalised the Chapter 19 vocabulary. The file is freely available; reading the surrounding code (the attach path, the `ithread`, the `iflib` integration) deepens the understanding further.

Worth reading after `mgb(4)`: `/usr/src/sys/dev/e1000/em_txrx.c` for MSI-X multi-vector patterns (Chapter 20 material), `/usr/src/sys/dev/usb/controller/xhci_pci.c` for a USB host controller's interrupt path (Chapter 21+), and `/usr/src/sys/dev/ahci/ahci_pci.c` for a storage controller's interrupt path.

19.11 A Deeper Look at Interrupt Context

Section 2 listed what may and may not happen in a filter. This section goes one level deeper and explains why each rule exists. A reader who understands the why can reason about unfamiliar constraints (new locks, new kernel APIs) and predict whether they are filter-safe.

19.11.1 The Stack Situation

When the CPU takes an interrupt, it saves the interrupted thread's registers and switches to a kernel interrupt stack. The interrupt stack is small (a few KB, platform-dependent), is per-CPU, and is shared among all interrupts on that CPU. The filter runs on this stack.

Two implications:

First, the filter has limited stack space. A filter that allocates a large array on the stack (hundreds of bytes or more) can overflow the interrupt stack. The symptom is usually a panic, sometimes a silent corruption of whatever sits next to the stack in memory. The rule is: filters have small stack budgets. Big arrays belong to the task.

Second, the stack is shared among interrupts. A filter that sleeps (hypothetically, though it cannot) would leave the stack occupied; other interrupts on the same CPU could not reuse it. Even if sleep were allowed, it would not be free. The small-stack constraint is one reason the filter must be short.

19.11.2 Why No Sleep Locks

A sleep mutex (default `mtx(9)`) may block: if another thread holds the mutex, `mtx_lock` puts the calling thread to sleep on the mutex. In the filter's context:

- There is no calling "thread" in the usual sense. The interrupted thread was suspended mid-instruction; the filter is a kernel-side excursion on the CPU.
- Sleeping from the interrupt would stall the CPU: the interrupt stack is occupied, no other interrupts can run on this CPU, and the scheduler cannot easily schedule a different thread without a working kernel state.

The kernel could, in principle, handle this (some kernels do). FreeBSD's design is to prohibit it. The prohibition is enforced by `WITNESS` on debug kernels: any attempt to acquire a sleepable lock in interrupt context produces an immediate panic.

Spin mutexes (`MTX_SPIN` mutexes) are safe because they do not sleep; they just spin. A filter acquiring a spin mutex is fine.

19.11.3 Why No Sleeping malloc

`malloc(M_WAITOK)` calls the VM page allocator and may sleep if the system is low on memory. Same problem as with locks: the caller cannot be suspended. `malloc(M_NOWAIT)` is the alternative; it may fail, but it never sleeps.

In the filter, the only safe options are `M_NOWAIT`, UMA zones (which have their own bounded allocators), or pre-allocated buffers. Chapter 19's driver does not allocate in the filter at all; all the memory the filter needs is in the `softc`, pre-allocated during attach.

19.11.4 Why No Condition Variables

`cv_wait` and `cv_timedwait` sleep. The filter cannot sleep. `cv_signal` and `cv_broadcast` do not sleep, but they acquire a sleep mutex internally in most usages; a filter using them must be careful. The Chapter 19 task handles the `cv_broadcast`; the filter only enqueues the task.

19.11.5 Why the Filter Cannot Re-enter Itself

The kernel's interrupt dispatch disables further interrupts on the CPU while the filter is running (on most platforms; some architectures use priority levels instead). This means the filter cannot recursively fire itself, even if its device asserts the line during execution. Any such assertion is queued and fires after the filter returns.

A consequence: the filter does not need internal re-entry protection. A single `sc->intr_count++` is safe from the filter's perspective against simultaneous filter calls on the same CPU. It may still race with other code (the task, user-space reads), which is why the atomic operation is used, but the filter does not race with itself.

19.11.6 Why Atomic Operations Are Safe

Atomic operations in FreeBSD are implemented as CPU instructions that are, by definition, atomic. They do not take a lock; they do not sleep; they do not block. They are safe in every context, including the filter.

Chapter 19 uses `atomic_add_64` extensively: for the interrupt counter, for each per-bit counter, and for task-invocation counts. The operations are cheap (a few cycles) and predictable (no scheduling involved).

19.11.7 Why the ithread Gets More Rope

The ithread runs in thread context. It has:

- Its own stack (normal kernel thread stack, much larger than the interrupt stack).
- Its own scheduling priority (elevated, but still a normal thread).
- The ability to sleep if the scheduler decides to.

Its constraints are the usual thread-context rules: hold sleepable locks in order (to prevent deadlocks), avoid holding a lock while calling unbounded code, use `M_WAITOK` if the allocation might fail otherwise, and so on. The Chapter 13 and 15 disciplines apply.

The ithread's elevated priority means it should not block for arbitrary lengths of time. Blocking for microseconds (a short mutex contention) is fine; blocking for seconds starves every other ithread on the system.

19.11.8 Why taskqueue Threads Have Still More Rope

A taskqueue's worker thread is a regular kernel thread, usually at normal priority (or at the priority the driver specified in `taskqueue_start_threads`). It can sleep, it can block on any sleepable lock, it can allocate arbitrarily. It is the most flexible of the three contexts.

The trade-off is that taskqueue work is less timely than ithread work. The taskqueue's worker may not run immediately; the scheduler decides. For latency-critical work, the ithread is better; for bulk work, the taskqueue is simpler.

Chapter 19's driver uses the taskqueue because the bulk work it does (reading `DATA_OUT`, updating the `softc`, broadcasting a condition variable) is not latency-critical. Chapter 20's and Chapter 21's drivers may choose ithreads or taskqueues differently depending on their workload.

19.11.9 How Interrupt Context Interacts With Locking

The locking discipline introduced in Part 3 still applies, with one addition: know which locks you can take in each context.

Filter context. Only spin locks, atomics, lock-free algorithms. No sleepable mutexes, sx locks, rwlocks, or mtx initialised without MTX_SPIN.

ithread context. All lock types. Obeys the project's lock ordering as defined in `LOCKING.md`.

Taskqueue worker context. All lock types. Obeys the project's lock ordering. May sleep arbitrarily if needed (though the driver author should not).

Thread context in general (cdev open/read/write/ioctl, sysctl handlers). All lock types. Obeys the project's lock ordering.

For Chapter 19's driver, the filter takes no locks (uses atomics), the task takes `sc->mtx` (a sleep mutex) via `MYFIRST_LOCK`, and the sysctl handlers take `sc->mtx` too. The discipline is preserved.

19.11.10 The “Giant” Footnote

Older BSDs used a single global lock called Giant to serialise the whole kernel. When FreeBSD introduced SMPng (fine-grained locking) in the late 1990s and early 2000s, most of the kernel was converted, but some legacy paths still hold Giant. Drivers that do not set `INTR_MPSAFE` are automatically wrapped with Giant acquisition around their handlers; `WITNESS` may complain about lock-ordering issues involving Giant.

Chapter 19's driver sets `INTR_MPSAFE` and does not touch Giant. Modern FreeBSD driver conventions deprecate Giant in driver code. The footnote is here because a reader who searches for “Giant” in `kern_intr.c` will find references; they are backward-compatibility artefacts.

19.12 A Deeper Look at CPU Affinity for Interrupts

A short appendix on interrupt affinity, which Section 2 introduced briefly. The deep treatment belongs to Chapter 20 (when multiple MSI-X vectors are available) and to later scalability chapters; Chapter 19's coverage is a starting point.

19.12.1 What Affinity Means

Interrupt affinity is the set of CPUs on which an interrupt is allowed to fire. For a single-CPU system, affinity is trivial (one CPU). For multi-CPU systems, affinity becomes interesting: routing an interrupt to a specific CPU (rather than letting the interrupt controller pick) can improve cache locality, reduce cross-socket traffic, and align interrupt handling with thread placement.

On x86, the I/O APIC has a programmable destination field per IRQ; the kernel uses this to route IRQs. On arm64, the GIC has similar facilities. FreeBSD's `bus_bind_intr(9)` is the portable API that configures the affinity for a specific IRQ resource.

19.12.2 Default Behaviour

Without explicit binding, FreeBSD spreads interrupts across CPUs using a round-robin or platform-specific algorithm. For a single-interrupt driver like Chapter 19's, this usually means the interrupt fires on whichever CPU the kernel decided at boot. The current affinity is visible through `cpuset -g -x <irq>`; the per-CPU breakdown of firings for a given IRQ is not part of `vmstat -i`'s default output (which aggregates all firings for the `intr_event` into one count) but can be reconstructed from kernel tooling when the platform supports it.

For many drivers the default is fine. The interrupt rate is low enough that affinity does not matter, or the work is brief enough that cross-CPU costs are negligible. Chapter 19's driver is in this category.

19.12.3 When Affinity Matters

Three scenarios where a driver author wants explicit affinity:

1. **High interrupt rate.** A NIC handling ten gigabits of traffic fires tens of thousands of interrupts per second. The overhead of moving interrupt work between CPUs becomes a real cost. Binding each receive queue's MSI-X vector to a specific CPU keeps its cache lines warm.
2. **NUMA locality.** On multi-socket systems, the device's PCIe root complex is physically attached to one socket. Interrupts from that device are cheaper to handle on CPUs in the same NUMA node as the root complex. Placement matters for both latency and throughput.
3. **Real-time constraints.** A system that needs low-latency response on specific CPUs (for real-time applications) may pin housekeeping interrupts away from those CPUs. `bus_bind_intr` lets the driver participate in this partitioning.

19.12.4 The `bus_bind_intr` API

The function signature:

```
int bus_bind_intr(device_t dev, struct resource *r, int cpu);
```

`cpu` is an integer CPU ID in the range 0 to `mp_ncpus - 1`. On success, the interrupt is routed to that CPU. On failure, the function returns an `errno` (most commonly `EINVAL` if the platform does not support rebinding or the CPU is invalid).

The call goes after `bus_setup_intr`:

```
error = bus_setup_intr(dev, irq_res, flags, filter, ihand, arg,
    &cookie);
if (error == 0)
    bus_bind_intr(dev, irq_res, preferred_cpu);
```

Chapter 19's driver does not bind its interrupt. A challenge exercise adds a `sysctl` that lets the operator set the preferred CPU.

19.12.5 The Kernel's CPU-Set Abstraction

A more sophisticated API: `bus_get_cpus(9)` lets the driver query which CPUs are considered “local” for a device, useful for multi-queue drivers that want to spread interrupts across a NUMA-local subset of CPUs. The `LOCAL_CPUS` and `INTR_CPUS` cpusets from `/usr/src/sys/sys/bus.h` expose this information.

Chapter 20's MSI-X work will use `bus_get_cpus(9)` to place per-queue interrupts on different CPUs in the device's local NUMA node. Chapter 19's single-interrupt driver does not need the complexity.

19.12.6 Observing Affinity

The `cpuset -g -x <irq>` command shows the current CPU mask for an IRQ. For the `myfirst` driver on a multi-CPU system, obtain the IRQ number from `devinfo -v | grep -A 5 myfirst0`, bind the interrupt to (say) CPU 1 with `cpuset -l 1 -x <irq>`, and confirm with `cpuset -g -x <irq>`.

The details are platform-specific. On x86 the I/O APIC (or MSI routing) implements the request; on arm64 the GIC redistributor does. Some architectures refuse rebinding and return an error; a cooperative driver's `bus_bind_intr` call treats that as a non-fatal hint.

19.13 A Deeper Look at Interrupt Storm Detection

The FreeBSD kernel has built-in protection against a specific failure mode: a level-triggered IRQ that fires continuously because the driver fails to acknowledge. The protection is called interrupt storm detection, is implemented in `/usr/src/sys/kern/kern_intr.c`, and is controlled by a single sysctl.

19.13.1 The `hw.intr_storm_threshold` Sysctl

```
static int intr_storm_threshold = 0;
SYSCTL_INT(_hw, OID_AUTO, intr_storm_threshold, CTLFLAG_RWTUN,
    &intr_storm_threshold, 0,
    "Number of consecutive interrupts before storm protection is enabled");
```

The default is zero (storm detection disabled). Setting the sysctl to a positive value enables detection: if an `intr_event` delivers more than N interrupts in a row without any other interrupt happening on the same CPU, the kernel assumes a storm and throttles the event.

Throttling means the kernel pauses (via `pause("istorm", 1)`) before running the handlers again. The pause is a single clock tick, which on most systems is a millisecond or so. The effect is to cap the rate at which a storming source can consume CPU.

19.13.2 When to Enable Detection

Default off is the production setting. Enabling storm detection means the kernel pauses interrupts when it thinks a storm is happening; if the detection is wrong (a high-rate legitimate interrupt, say a 10-gigabit NIC), the pause is a performance bug.

For driver development, enabling storm detection is useful: a forgotten acknowledgment in the filter produces an interrupt storm, which the kernel detects and throttles (and logs to `dmesg`). Without detection, the storm consumes a CPU forever; with detection, the storm is visible and throttled.

A reasonable development-time setting is `hw.intr_storm_threshold=1000`. A thousand consecutive interrupts on the same event without interleaving is unusual for legitimate traffic and reliably flags a storm.

19.13.3 What a Storm Looks Like

In `dmesg`:

```
interrupt storm detected on "irq19: myfirst0:legacy"; throttling interrupt source
```

Repeated at rate-limited intervals (once per second by default, governed by the `ppsratecheck` inside the kernel's storm code). The interrupt source is named; the driver can be identified from the name.

The kernel does not disable the line permanently; it paces the handler. After the storm ends (perhaps because the driver was unloaded, or the device stopped asserting), the handler resumes at full rate.

19.13.4 Driver-Side Storm Mitigation

A driver can implement its own storm mitigation. The classic technique is:

1. Count interrupts in a rolling window.
2. If the rate exceeds a threshold, mask the device's interrupts (via `INTR_MASK`) and schedule a task to re-enable them later.
3. In the task, inspect the device, clear whatever is causing the storm, and re-enable.

This is more invasive than the kernel's default. Most drivers do not implement it. The Chapter 19 driver does not; the kernel's threshold is sufficient for the scenarios the chapter exercises.

19.13.5 The Relationship to Shared IRQs

On a shared IRQ line, one driver's storm can interfere with another driver's legitimate interrupts. The kernel's storm detection is per-event, not per-handler, so if one driver's handler is slow or incorrect, the whole event throttles. This is a strong argument for writing correct filters: the storm impact is not confined to the buggy driver.

19.14 A Mental Model for Filter vs `ithread` Selection

Beginners often struggle with the decision between filter-only, filter-plus-`ithread`, filter-plus-taskqueue, and `ithread`-only. This section provides a decision framework that helps in most situations, grounded in questions a driver author can answer about their specific device.

19.14.1 Four Questions

Ask these about the interrupt's work:

1. **Can every piece of the work be done in primary context?** If yes (all state access through spin locks or atomics; all acknowledgments through BAR writes; no sleeping), filter-only is the cleanest choice.
2. **Does any piece of the work require a sleep lock or a condition-variable broadcast?** If yes, the bulk work must go in thread context. The choice is between ithread and taskqueue.
3. **Is the deferred work scheduled from anywhere besides the interrupt?** If yes (sysctl handlers, ioctl, timer callouts, other tasks), a taskqueue is better. The same work can be scheduled from any context.
4. **Is the deferred work sensitive to the interrupt's priority class?** If yes (you want `INTR_TYPE_NET` ithread priority for network work), register an ithread handler. The ithread inherits the interrupt's priority; a taskqueue runs at whatever priority its worker thread got at creation.

19.14.2 Applying the Framework

Filter-only fits: - A counter-incrementing demo driver. - A driver that only reads a device register and passes the value through an atomic. - A very simple sensor whose data is produced rarely and read directly.

Filter-plus-ithread fits: - A simple driver where the deferred work only matters on interrupt. - A driver that benefits from the interrupt's priority class. - A driver that wants the kernel-managed ithread without the taskqueue's extra machinery.

Filter-plus-taskqueue fits: - A driver where the same deferred work can be triggered by multiple sources (interrupt, sysctl, ioctl). - A driver that needs to coalesce interrupts (the taskqueue's npending count tells you how many enqueues happened since the last run). - A driver that wants a specific worker-thread count or priority independent of the interrupt category. - Chapter 19's target case: the `myfirst` driver schedules the same task from the filter and from the simulated-interrupt sysctl.

ithread-only fits: - A driver where there is no urgent work and every action needs a sleep lock. - A driver where the filter would be trivial (just "schedule the thread"); registering no filter and letting the kernel schedule the ithread saves a function call.

19.14.3 Worked Example: A Hypothetical Storage Driver

Suppose you are writing a driver for a small storage controller. The device has one IRQ line. When an I/O completes, it sets `INTR_STATUS.COMPLETION` and lists the completed command IDs in a completion queue register.

The decisions:

- **Can every piece be done in primary context?** No. Waking up the thread that issued the I/O requires a condition-variable broadcast, which requires the thread's lock. The filter cannot take that lock.
- **Which deferred mechanism?** The completion-handling work is scheduled only by the interrupt, so filter-plus-ithread is clean. The priority class is `INTR_TYPE_BIO`, which the ithread inherits.
- **Final design.** Filter reads `INTR_STATUS`, extracts the completed command IDs into a per-interrupt-context queue, acknowledges, returns `FILTER_SCHEDULE_THREAD`. ithread walks the per-context queue, matches command IDs to pending requests, wakes each request's thread.

19.14.4 Worked Example: A Hypothetical Network Driver

A NIC with four MSI-X vectors (two receive queues, two transmit queues). Each vector has its own filter.

The decisions:

- **Filter work?** Per-queue: acknowledge, note that the queue has events.
- **Deferred work?** Per-queue: walk the descriptor ring, build mbufs, pass to the stack.
- **Multiple sources?** Just the interrupt for normal operation; polling mode (for high-load offload) is a second source. Taskqueue is better: both the filter and the poll-mode timer can enqueue.
- **Priority?** `INTR_TYPE_NET`, which the taskqueue worker's `PI_NET` priority matches.
- **Final design.** Per-vector filter returns `FILTER_HANDLED` after enqueueing the per-queue task. Taskqueue per receive queue, one worker each. Taskqueues configured at priority `PI_NET`.

19.14.5 Worked Example: The Chapter 19 Driver

One IRQ line, simple event types, taskqueue-based deferred work.

- **Filter work:** Read `INTR_STATUS`, per-bit counters, acknowledge, enqueue task for `DATA_AV`.
- **Deferred work:** Read `DATA_OUT`, update softc, broadcast `data_cv`.
- **Multiple sources?** Filter and simulated-interrupt `sysctl` both need the task. Taskqueue is right.
- **Priority?** `PI_NET` is a sensible default even though the driver isn't a NIC; the simulation framework expects responsiveness.

19.14.6 When to Revisit the Decision

The decision is not permanent. A driver that starts as filter-only may grow a task when it gains a new capability; a driver that starts as taskqueue may move to `ithread` when the extra flexibility of taskqueue is not needed. The refactor is usually small (half an hour of code movement).

The framework helps you avoid an obviously-wrong initial choice. The details are a judgment call that the driver author makes based on the specific device.

19.15 Lock Ordering and the Interrupt Path

Part 3's lock discipline introduced the idea that the driver has a fixed lock order: `sc->mtx` -> `sc->cfg_sx` -> `sc->stats_cache_sx`. Chapter 19 does not add new locks but does add new contexts where existing locks are touched. This subsection examines whether the Chapter 19 additions respect the existing order.

19.15.1 The Filter Takes No Locks

The filter reads `INTR_STATUS`, updates atomic counters, and enqueues a task. No sleep lock is acquired. The filter's access to `INTR_STATUS` uses `ICSR_READ_4` and `ICSR_WRITE_4`, which do not assert any lock. Therefore the filter does not participate in the lock order; it is lock-free.

This is the simplest possible choice. A more sophisticated filter might use a spin lock (to protect a small shared data structure); Chapter 19's filter is simpler than that.

19.15.2 The Task Acquires `sc->mtx`

The task's function `myfirst_intr_data_task_fn` acquires `sc->mtx` (via `MYFIRST_LOCK`), does its work, and releases. It does not acquire any other lock. Therefore the task respects the existing lock order by not introducing any new lock acquisition pattern.

19.15.3 The Simulated-Interrupt `sysctl` Acquires and Releases `sc->mtx`

The `sysctl` handler acquires `sc->mtx` to set `INTR_STATUS`, releases the lock, and then invokes the filter. This is not a lock-order violation because the filter takes no locks; no new edge is added to the lock graph.

19.15.4 The Attach and Detach Paths

The attach path acquires `sc->mtx` briefly to set `INTR_MASK` and to perform the initial diagnostic read. It does not hold the lock across `bus_setup_intr` (which could, in principle, call into other parts of the kernel that take their own locks; `bus_setup_intr` is documented as lockable, meaning the caller can hold nothing of its own). The detach path similarly brief-holds `sc->mtx` around the `INTR_MASK` clear, then releases before calling `bus_teardown_intr`.

19.15.5 A Subtle Ordering Concern: `bus_teardown_intr` Can Block

A detail worth calling out. `bus_teardown_intr` waits for any in-progress filter or `ithread` invocation to complete before returning. If the driver holds a lock that the filter needs (say, a spin lock that the filter acquires briefly), `bus_teardown_intr` may block forever because the filter cannot run to completion.

Chapter 19's filter takes no spin locks, so this concern is academic. But a driver that uses spin locks in the filter must be careful: never hold the filter's spin lock while calling `bus_teardown_intr`.

19.15.6 WITNESS and the Interrupt Path

The debug kernel's `WITNESS` tracks lock ordering across every context, including the filter. A filter that takes a spin lock creates an ordering edge in `WITNESS`'s graph. If any thread-context code takes the same spin lock while holding a different spin lock, `WITNESS` flags a potential deadlock.

For the Chapter 19 driver, no edges are added. `WITNESS` is silent.

19.15.7 What to Document in `LOCKING.md`

A good driver's `LOCKING.md` documents the lock order clearly. Chapter 19's additions are minor:

- The filter takes no locks (atomic operations only).
- The task takes `sc->mtx` (leaf of the existing order).
- The simulated-interrupt `sysctl` takes `sc->mtx` briefly to set state, releases, then invokes the filter (outside any lock).

A short paragraph in `LOCKING.md` notes these facts. The order itself does not change.

19.16 Observability: What Chapter 19 Exposes to Operators

A chapter about interrupts is also, indirectly, about observability. The user of a driver (a system operator or a driver author debugging an issue) wants to see what the driver is doing. Chapter 19 exposes a modest amount of observability through counters and the simulated-interrupt `sysctl`; this subsection consolidates what is visible and how.

19.16.1 The Counter Suite

After Stage 4, the driver exposes these read-only `sysctls`:

- `dev.myfirst.N.intr_count`: total filter invocations.
- `dev.myfirst.N.intr_data_av_count`: `DATA_AV` events.
- `dev.myfirst.N.intr_error_count`: `ERROR` events.
- `dev.myfirst.N.intr_complete_count`: `COMPLETE` events.
- `dev.myfirst.N.intr_task_invocations`: task runs.
- `dev.myfirst.N.intr_last_data`: most recent `DATA_OUT` read by the task.

The counters give a concise view of the interrupt activity. Watching them over time (through `watch sysctl dev.myfirst.0` or a shell loop) shows the driver's activity live.

19.16.2 The Writable `Sysctls`

- `dev.myfirst.N.intr_simulate`: write a bitmask to simulate an interrupt.

(Chapter 19's driver only exposes this one writable `sysctl` for interrupts. Challenge exercises add `intr_sim_period_ms` for rate-based simulation and `intr_cpu` for affinity.)

19.16.3 The Kernel-Level View

`vmstat -i` and `devinfo -v` already show the kernel's view:

- `vmstat -i` shows the `intr_event`'s total count and rate.
- `devinfo -v` shows the device's IRQ resource.

These are not specific to `myfirst`; they are available for every driver. Learning to read them is part of general FreeBSD operator skill.

19.16.4 Correlating the Views

An operator trying to diagnose an issue might cross-check counters:

```
# The kernel's count of interrupts delivered to our handler.  
vmstat -i | grep myfirst
```

```
# The driver's count of times the filter was invoked.
sysctl dev.myfirst.0.intr_count
```

If these numbers match, the kernel's path and the driver's filter are in agreement. If the kernel count exceeds the driver count, some interrupts are being handled but not recognised (perhaps by another handler on a shared line). If the driver count exceeds the kernel count, something is wrong (the driver is counting invocations the kernel did not deliver; the simulated-interrupt sysctl is the most likely culprit if it was fired recently).

A disparity of one or two across a load-unload cycle is normal (timing around the unload). A consistent growing disparity indicates a bug.

19.16.5 DTrace

The kernel's fbt provider lets you trace the entry and exit of any kernel function, including `myfirst_intr_filter`:

```
sudo dtrace -n 'fbt::myfirst_intr_filter:entry { @[probefunc] = count(); }'
```

This prints the count of filter invocations seen by DTrace. Cross-check against `intr_count`.

More interestingly, a DTrace script can aggregate per-call timing:

```
sudo dtrace -n '
fbt::myfirst_intr_filter:entry { self->t = timestamp; }
fbt::myfirst_intr_filter:return /self->t/ {
    @["filter_ns"] = quantize(timestamp - self->t);
    self->t = 0;
}'
```

The output is a histogram of filter execution times in nanoseconds. A healthy filter spends between a few hundred nanoseconds and a few microseconds; anything higher is a bug or an extremely slow device.

19.16.6 ktrace and kgdb

For deep debugging, `ktrace` can trace system-call activity; `kgdb` can inspect a panic's kernel core dump. Chapter 19 does not use these directly, but a reader whose driver panics in the interrupt path would need them.

Each lab builds on the previous one and corresponds to one of the chapter's stages. A reader who works through all five has a complete interrupt-aware driver, a simulated-interrupt pipeline, and a regression script that validates the whole thing.

Time budgets assume the reader has already read the relevant sections.

19.16.7 Lab 1: Explore Interrupt Sources on Your System

Time: thirty minutes.

Objective: Build an intuition for what interrupts your system is handling and at what rate.

Steps:

1. Run `vmstat -i > /tmp/intr_before.txt`.
2. Do something that exercises the system for thirty seconds: run `dd if=/dev/urandom of=/dev/null bs=1m count=1000`, or open a browser page (on a system with a graphical session), or scp a file from another host.
3. Run `vmstat -i > /tmp/intr_after.txt`.
4. Compute the difference with `diff`:

```
paste /tmp/intr_before.txt /tmp/intr_after.txt
```

5. For each source that changed, note:
 - The interrupt name.
 - The count before and after.
 - The inferred rate during the thirty seconds.
6. Pick one source and identify its driver with `devinfo -v` or `pciconf -lv`.

Expected observations:

- The timer interrupts (`cpu0:timer` and so on) are high and steady, one per CPU.
- Network interrupts (`em0`, `igc0`, etc.) are high during `dd` or `scp` activity, near-zero otherwise.
- Storage interrupts (`ahci0`, `nvme0`, etc.) are high during disk activity, low otherwise.
- Some interrupts never change; those are devices that are quiet during your test.

This lab is about reading reality. No code. The payoff is that every later lab's `vmstat -i` output is familiar territory.

19.16.8 Lab 2: Stage 1, Register and Fire the Handler

Time: two to three hours.

Objective: Add the interrupt allocation, filter registration, and cleanup to Chapter 18's driver. Version target: `1.2-intr-stage1`.

Steps:

1. Starting from Chapter 18 Stage 4, copy the driver source to a new working directory.
2. Edit `myfirst.h` and add the four softc fields (`irq_res`, `irq_rid`, `intr_cookie`, `intr_count`).
3. In `myfirst_pci.c`, add the minimal filter handler (`atomic_add_64`; return `FILTER_HANDLED`).
4. Extend the attach path with the IRQ allocation, `bus_setup_intr`, and `bus_describe_intr` calls. Add the corresponding goto labels.
5. Extend the detach path with `bus_teardown_intr` and `bus_release_resource` for the IRQ.

6. Add a read-only sysctl `dev.myfirst.N.intr_count`.
7. Bump the version string to `1.2-intr-stage1`.
8. Compile: `make clean && make`.
9. Load on a bhyve guest. Check:
 - No `[GIANT-LOCKED]` warning in `dmesg`.
 - `devinfo -v | grep -A 5 myfirst0` shows both `memory:` and `irq:`.
 - `vmstat -i | grep myfirst` shows the handler.
 - `sysctl dev.myfirst.0.intr_count` returns a sensible value (zero if the device is quiet).
10. Unload. Check `vmstat -m | grep myfirst` shows zero live allocations.

Common failures:

- Missing `INTR_MPSAFE`: check for `[GIANT-LOCKED]` in `dmesg`.
- Wrong `rid` value: `bus_alloc_resource_any` returns `NULL`. Confirm `sc->irq_rid = 0`.
- Sleep lock in the filter: `WITNESS` panics.
- Missing teardown: `kldunload` panics or the debug kernel complains about active handlers.

19.16.9 Lab 3: Stage 2, Real Filter and Deferred Task

Time: three to four hours.

Objective: Extend the filter to read `INTR_STATUS`, acknowledge, and enqueue a deferred task. Version target: `1.2-intr-stage2`.

Steps:

1. Starting from Lab 2, add the per-bit counters (`intr_data_av_count`, `intr_error_count`, `intr_complete_count`, `intr_task_invocations`, `intr_last_data`) to the softc.
2. Add the taskqueue (`intr_tq`) and task (`intr_data_task`) fields.
3. In `myfirst_init_softc`, initialise the task and create the taskqueue.
4. In `myfirst_deinit_softc`, drain the task, free the taskqueue.
5. Rewrite the filter to read `INTR_STATUS`, check each bit, acknowledge, enqueue the task for `DATA_AV`, and return the right `FILTER_*` value.
6. Write the task function (`myfirst_intr_data_task_fn`) that reads `DATA_OUT`, updates the softc, and broadcasts the condition variable.
7. In the attach path, after the filter registration, enable `INTR_MASK` at the device.
8. In the detach path, disable `INTR_MASK` before `bus_teardown_intr`.
9. Add read-only sysctls for the new counters.
10. Bump the version to `1.2-intr-stage2`.
11. Compile, load, verify basic wiring (same as Lab 2).

For observation, wait a few seconds after load: if the device is producing real interrupts that match our bit layout, counters tick up. On the bhyve virtio-rnd target, no real interrupts of the right kind arrive; verify counters by proceeding to Lab 4.

19.16.10 Lab 4: Stage 3, Simulated Interrupts via sysctl

Time: two to three hours.

Objective: Add the `intr_simulate` sysctl, use it to drive the pipeline. Version target: 1.2-intr-stage3.

Steps:

1. Starting from Lab 3, add the `intr_simulate` sysctl handler (the one in Section 5).
2. Register it in `myfirst_init_softc` or the sysctl setup.
3. Compile, load.
4. Simulate a single `DATA_AV` event:

```
sudo sysctl dev.myfirst.0.intr_simulate=1
sleep 0.1
sysctl dev.myfirst.0.intr_count
sysctl dev.myfirst.0.intr_data_av_count
sysctl dev.myfirst.0.intr_task_invocations
```

All three counters should show 1.

5. Simulate ten `DATA_AV` events in a loop:

```
for i in 1 2 3 4 5 6 7 8 9 10; do
    sudo sysctl dev.myfirst.0.intr_simulate=1
done
sleep 0.5
sysctl dev.myfirst.0.intr_task_invocations
```

The task count should be close to 10 (may be less if the taskqueue coalesced multiple enqueues into a single run; each run records only one invocation but `npending` would be larger).

6. Simulate all three bits together:

```
sudo sysctl dev.myfirst.0.intr_simulate=7
```

All three per-bit counters increment.

7. Check the `intr_error_count` and `intr_complete_count` increment correctly:

```
sudo sysctl dev.myfirst.0.intr_simulate=2 # ERROR
sudo sysctl dev.myfirst.0.intr_simulate=4 # COMPLETE
sysctl dev.myfirst.0 | grep intr_
```

8. Implement the optional rate-based callout (`intr_sim_period_ms`), verify rate:

```
sudo sysctl hw.myfirst.intr_sim_period_ms=100
sleep 10
sysctl dev.myfirst.0.intr_count # around 100
sudo sysctl hw.myfirst.intr_sim_period_ms=0
```

19.16.11 Lab 5: Stage 4, Refactor, Regression, Version

Time: three to four hours.

Objective: Move interrupt code into `myfirst_intr.c/.h`, introduce `ICSR_*` macros, write `INTERRUPTS.md`, run the regression. Version target: `1.2-intr`.

Steps:

1. Starting from Lab 4, create `myfirst_intr.c` and `myfirst_intr.h`.
2. Move the filter, the task, the setup, the teardown, and the sysctl registration into `myfirst_intr.c`.
3. Add `ICSR_READ_4` and `ICSR_WRITE_4` macros to `myfirst_intr.h`.
4. Update the filter to use `ICSR_READ_4/ICSR_WRITE_4` instead of `raw bus_read_4/bus_write_4`.
5. In `myfirst_pci.c`, replace the inline interrupt code with calls to `myfirst_intr_setup` and `myfirst_intr_teardown`.
6. Update the Makefile to add `myfirst_intr.c` to SRCS. Bump the version to `1.2-intr`.
7. Write `INTERRUPTS.md` documenting the interrupt handler's design.
8. Compile.
9. Run the full regression script (ten cycles of attach/detach/unload with counter checks; see the companion example).
10. Confirm: no warnings, no leaks, counters match expectations.

Expected outcomes:

- The driver at `1.2-intr` has the same behaviour as Stage 3 but a cleaner file structure.
- `myfirst_pci.c` is shorter by 50-80 lines.
- `myfirst_intr.c` is roughly 200-300 lines.
- The regression script passes ten times in a row.

19.17 Challenge Exercises

The challenges are optional. Each builds on one of the labs and extends the driver in a direction the chapter did not take. They consolidate the chapter's material and are good preparation for Chapter 20.

19.17.1 Challenge 1: Add a Filter-Plus-ithread Handler

Rewrite Stage 2's filter so that it returns `FILTER_SCHEDULE_THREAD` instead of enqueueing a taskqueue task. Register an ithread handler through the fifth argument of `bus_setup_intr(9)` that does the work the task did. Compare the two approaches.

This exercise is the way to internalise the difference between ithread-based deferred work and taskqueue-based deferred work. After completing it, the reader should be able to say when each is appropriate.

19.17.2 Challenge 2: Implement Driver-Side Storm Mitigation

Add a counter that tracks the number of interrupts handled in the current millisecond. If the count exceeds a threshold (say, 10000), mask the device's interrupts and schedule a task to re-enable them 10 ms later.

This exercise demonstrates that driver-side mitigation is possible, and it shows why the kernel's default (do-nothing) is usually fine.

19.17.3 Challenge 3: Bind the Interrupt to a Specific CPU

Add a `sysctl dev.myfirst.N.intr_cpu` that accepts a CPU ID. When written, call `bus_bind_intr(9)` to route the interrupt to that CPU. Verify with `cpuset -g` or the per-CPU counts in `vmstat -i`.

This exercise introduces the CPU-affinity API and shows how the choice is visible in system-level tools.

19.17.4 Challenge 4: Extend the Simulated Interrupts With Per-Type Rates

Modify the `intr_sim_period_ms` callout to accept a bitmask of which event classes to simulate, not just `DATA_AV`. A reader should be able to simulate alternating `ERROR` and `COMPLETE` events at different rates.

The exercise exercises the reader's understanding of the Stage 2 filter's per-bit handling.

19.17.5 Challenge 5: Add a Rate-Limited Stray Log

Implement the `ppsratecheck(9)`-based stray-interrupt log mentioned in Section 6. Verify that the log appears at the expected rate when the driver receives strays (you can induce strays by disabling `INTR_MASK` while the device is generating events, or by manually calling the filter with a zero status).

19.17.6 Challenge 6: Implement MSI Allocation (Preview of Chapter 20)

Add code to the attach path that attempts `pci_alloc_msi(9)` first and falls back to legacy IRX if MSI is not available. The filter remains the same. This is a preview of Chapter 20; doing it now gets the reader comfortable with the MSI allocation API.

Note that on the bhyve virtio-rnd target, MSI is typically not available (bhyve's legacy virtio transport uses INTx). QEMU's `virtio-rng-pci` exposes MSI-X; you may want to switch labs to QEMU for this challenge.

19.17.7 Challenge 7: Write a Latency Test

Use the simulated-interrupt path to measure the driver's interrupt-to-user-space latency. A user-space program opens `/dev/myfirst0`, issues a `read(2)` that sleeps waiting on the condition variable; a second program writes the `intr_simulate` sysctl, starting a wall-clock timer; the first program's read returns, stopping the timer. Plot the distribution over many iterations.

This exercise exposes the reader to performance measurement on the driver's deferred path. Typical latencies are tens of microseconds on a well-tuned system.

19.17.8 Challenge 8: Share the IRQ Deliberately

If you have a bhyve guest configured with multiple devices on the same slot's functions, deliberately force them to share an IRQ. Load both drivers (the base-system driver for the other device; our `myfirst` for `virtio-rnd`). Verify with `vmstat -i` that they share the line. Observe the behaviour when either fires.

This exercise is the clearest demonstration of shared-IRQ correctness. A driver that gets Section 6's discipline wrong will misbehave here.

19.18 Troubleshooting and Common Mistakes

A consolidated list of interrupt-specific failure modes, symptoms, and fixes. Kept as a reference you can return to.

19.18.1 “Driver loads but no interrupts are counted”

Symptom: `kldload` succeeds, `dmesg` shows the attach banner, but `sysctl dev.myfirst.0.intr_count` stays at zero indefinitely.

Likely causes:

1. The device is not producing interrupts. On the bhyve `virtio-rnd` target this is normal because the device does not generate Chapter 17-style events. Use the simulated-interrupt `sysctl` to drive the pipeline.
2. `INTR_MASK` was not set. The handler is registered, but the device is not asserting the line because the mask is zero. Check the attach path for the `CSR_WRITE_4(sc, MYFIRST_REG_INTR_MASK, ...)` call.
3. The device is masked by other means. Check the command register for `PCIM_CMD_INTxDIS` (interrupt disable bit); if set, clear it.
4. The wrong IRQ was allocated. `rid = 0` should produce the device's legacy `INTx`. Check `devinfo -v | grep -A 5 myfirst0` shows an `irq:` entry.

19.18.2 “GIANT-LOCKED warning in dmesg”

Symptom: `dmesg` after `kldload` shows `myfirst0: [GIANT-LOCKED]`.

Cause: `INTR_MPSAFE` was not passed to `bus_setup_intr`'s flags argument.

Fix: Add `INTR_MPSAFE` to the flags. Verify the filter uses spin-safe operations only (atomics, spin mutexes). Verify the lock discipline in the softc allows MP-safe operation.

19.18.3 “Kernel panic in the filter”

Symptom: A kernel panic whose backtrace shows `myfirst_intr_filter`.

Likely causes:

1. `sc` is `NULL` or stale. Check the argument to `bus_setup_intr`; it should be `sc`, not `&sc`.
2. A sleep lock is being acquired. `WITNESS` panics on this. The fix is to remove the sleep lock or move the work to a taskqueue.

3. The filter is called against a freed softc. This usually means detach did not tear down the handler before freeing state. Check the detach order.
4. `sc->bar_res` is NULL. A race between attach partial-failure unwinding and the filter running. Guard the filter's first access with a check.

19.18.4 “Task runs but accesses freed state”

Symptom: Kernel panic in the task function, backtrace shows `myfirst_intr_data_task_fn`.

Cause: The task was enqueued during or just before detach, and detach freed the softc before the task ran.

Fix: Add `taskqueue_drain` to the detach path, before freeing anything the task touches. See Section 7.

19.18.5 “INTR_STATUS bits keep firing; storm detected”

Symptom: `dmesg` shows `interrupt storm detected`.

Cause: The filter is not acknowledging `INTR_STATUS` correctly. Possibilities:

1. Filter does not write `INTR_STATUS` at all. Add the write.
2. Filter writes the wrong value. Write the specific bits you handled, not 0 and not `0xffffffff`.
3. Filter handles only some bits; unrecognised bits stay set and keep asserting. Either recognise all bits, or acknowledge unrecognised ones explicitly, or unset them in `INTR_MASK`.

19.18.6 “Simulated interrupt does not run the task”

Symptom: `sudo sysctl dev.myfirst.0.intr_simulate=1` increments `intr_count` but not `intr_task_invocations`.

Likely causes:

1. The simulated bit does not match what the filter looks for. The Stage 2 filter enqueues the task for `DATA_AV` (bit 0x1). Writing 2 or 4 sets `ERROR` or `COMPLETE`; those do not enqueue. Write 1 or 7.
2. The task function is not registered. Check `TASK_INIT` is called in `myfirst_init_softc`.
3. The taskqueue is not created. Check `taskqueue_create` and `taskqueue_start_threads` in `myfirst_init_softc`.

19.18.7 “kldunload fails with Device busy”

Symptom: `kldunload myfirst` fails with `Device busy`.

Causes: Same as Chapter 18. A user-space process has the `cdev` open; an in-flight command is not complete; the driver's busy check has a bug. Add `fstat /dev/myfirst0` to see who has it open.

19.18.8 “vmstat -m shows live allocations after unload”

Symptom: `vmstat -m | grep myfirst` returns non-zero `InUse` after `kldunload`.

Likely causes:

1. The taskqueue was not drained. Check `taskqueue_drain` in the detach path.
2. The simulation backend was attached (simulation-only build) and not detached. Check for `my-first_sim_detach` in the detach path.
3. A leak in `myfirst_init_softc / myfirst_deinit_softc`. Check that every allocation has a matching free.

19.18.9 “Handler fires on the wrong CPU”

Symptom: `cpuset -g` shows the interrupt fired on CPU X; the reader wanted it on CPU Y.

Cause: `bus_bind_intr` was not called, or was called with the wrong CPU argument.

Fix: Add a `sysctl` that lets the operator set the desired CPU and call `bus_bind_intr`. See Challenge 3.

19.18.10 “INTR_MASK write has unintended side effects”

Symptom: On the bhyve virtio-rnd target, writing to offset 0x10 (the Chapter 17 `INTR_MASK` offset) changes the device’s state in unexpected ways.

Cause: The Chapter 17 register layout does not match virtio-rnd’s. The offset 0x10 on virtio-rnd is `queue_notify`, not `INTR_MASK`.

Fix: This is a target mismatch, not a driver bug. The chapter acknowledges the issue. For correctness on a real device with the Chapter 17 layout, the write is right. For the bhyve teaching target, the write is harmless (it notifies an idle virtqueue) but not meaningful.

19.18.11 “Stray interrupt messages in dmesg”

Symptom: `dmesg` periodically shows messages about stray interrupts on the IRQ line.

Likely causes:

1. The handler is not masking `INTR_MASK` at the device properly during detach (level-triggered stray).
2. The device is producing interrupts the driver has not enabled. Check `INTR_MASK` setting.
3. Another driver sharing the line is returning the wrong `FILTER_*` value. This is that driver’s bug, not ours.

19.18.12 “Handler is called concurrently on multiple CPUs”

Symptom: Atomic counters increment non-monotonically, suggesting concurrent filter invocations.

Cause: On MSI-X (Chapter 20) the same handler can run concurrently on different CPUs. This is by design. For legacy IRQs this is rare but possible in some configurations.

Fix: Ensure all filter state access is atomic or spin-lock-protected. The Chapter 19 driver uses `atomic_add_64` throughout; no changes needed.

19.18.13 “bus_setup_intr returns EINVAL”

Symptom: The return value of `bus_setup_intr` is `EINVAL` and the driver fails to load.

Likely causes:

1. Both `filter` and `ihand` arguments are `NULL`. At least one must be non-`NULL`; the kernel has nothing to call otherwise.
2. The `INTR_TYPE_*` flag was omitted from the `flags` argument. Exactly one category must be set.
3. The IRQ resource was not allocated with `RF_ACTIVE`. An unactivated resource cannot have a handler attached.
4. The `flags` argument contains mutually-exclusive bits (rare; a driver author would have to invent this).

Fix: read the `bus_setup_intr(9)` manual page; the common case is that the `filter` or `ithread` argument is missing or the category flag is missing.

19.18.14 “bus_setup_intr returns EEXIST”

Symptom: `bus_setup_intr` returns `EEXIST` on a subsequent load.

Cause: The IRQ line already has an exclusive handler attached. Either this driver was loaded before and did not tear down properly, or another driver has claimed the line exclusively.

Fix: First, try unloading any previous instance (`kldunload myfirst`). If the problem persists, check `devinfo -v` for any driver currently using the IRQ.

19.18.15 “Debug kernel panics on taskqueue_drain”

Symptom: `taskqueue_drain` panics in the debug kernel.

Likely causes:

1. The taskqueue was never created. `sc->intr_tq` is `NULL`. Check `myfirst_init_softc`.
2. The taskqueue was already freed. Check for double-free in the teardown path.
3. `TASK_INIT` was never called. The task function pointer is garbage.

Fix: Ensure `TASK_INIT` runs before `taskqueue_enqueue` ever runs; ensure `taskqueue_free` runs at most once.

19.18.16 “Filter is called but INTR_STATUS reads as 0xffffffff”

Symptom: The filter runs, reads `INTR_STATUS`, sees `0xffffffff`.

Likely causes:

1. The device is unresponsive (perhaps the bhyve guest died or the device was hot-unplugged).
2. The BAR mapping is wrong. Check the attach path.
3. A PCI error has put the device into a limp state.

Fix: If the device is alive, the read returns real status bits. If `0xffffffff`, something else is wrong. The filter should still return `FILTER_STRAY` (because `0xffffffff` is unlikely to be a legitimate status value; cross-check with the device's datasheet for valid bit combinations).

19.18.17 “Interrupts are counted but the device makes no progress”

Symptom: `intr_count` ticks up, but the device's operation (data transfer, task completion, etc.) does not advance.

Likely causes:

1. The filter acknowledges every bit but the task does not run. Check `intr_task_invocations`; if zero, the `taskqueue_enqueue` path is broken.
2. The task runs but does not wake waiters. Check `cv_broadcast` in the task.
3. The device is signalling an unusual condition. Check `INTR_STATUS` contents (read via the `sysctl` path, or in DDB).

Fix: Add logging in the task (via `device_printf`); check whether the task's logic matches the device's actual behaviour.

19.18.18 “kldunload hangs”

Symptom: `kldunload myfirst` does not return. No panic, no output.

Likely causes:

1. `bus_tear_down_intr` is blocked waiting for an in-progress handler (filter or `ithread`). The handler is stuck.
2. `taskqueue_drain` is blocked waiting for a task that is stuck.
3. The detach function is waiting on a condition variable that is never broadcast.

Fix: If the system is responsive otherwise, switch to DDB (press the NMI key or type `sysctl debug.kdb.enter=1`) and `ps` to find the stuck threads. The backtrace usually pinpoints the stuck function.

19.18.19 “Unaligned memory access in the filter”

Symptom: A kernel panic on an architecture sensitive to alignment (arm64, MIPS, SPARC), with a backtrace pointing to the filter.

Cause: The filter is reading or writing a register at an unaligned offset. PCI BAR reads and writes require natural alignment (4-byte for 32-bit reads, 2-byte for 16-bit reads).

Fix: Use `bus_read_4` / `bus_write_4` at 4-byte-aligned offsets. Chapter 17's register map is 4-byte aligned throughout.

19.18.20 “device_printf from the filter slows the system”

Symptom: Adding `device_printf` calls to the filter makes the system noticeably laggy at high interrupt rates.

Cause: `device_printf` acquires a lock and does a formatted print. At ten thousand interrupts per second, the overhead is measurable.

Fix: Remove debug prints from the filter before production testing. Use counters and DTrace for observability instead.

19.18.21 “Driver passes all tests but misbehaves under load”

Symptom: The single-threaded tests pass, but load testing with many concurrent processes triggers occasional errors or state corruption.

Likely causes:

1. A race condition between the filter and the task. The filter sets a flag the task reads; the task updates state the filter reads. Without proper synchronisation, one can miss the other’s update.
2. A race between the task and another thread-context path (cdev handler, `sysctl`). The task takes `sc->mtx`; the other path should too.
3. An atomic variable used in a compound operation without a lock. `atomic_add_64` on its own is atomic; `atomic_load_64` followed by a computation followed by `atomic_store_64` is not atomic as a sequence.

Fix: Review the locking discipline. WITNESS does not catch pure atomic-variable races; careful code review does. Run under heavy load with INVARIANTS enabled and watch for assertion failures.

19.18.22 “vmstat -i shows many strays on a line I don’t own”

Symptom: A driver on a shared line sees the line’s stray counter growing steadily.

Likely causes:

1. Another driver on the line returns `FILTER_STRAY` incorrectly (the interrupt is for it but it claims otherwise).
2. A device on the line is signalling events the drivers do not acknowledge, producing phantom strays.
3. Hardware noise or a misconfigured trigger mode.

Fix: The fix is usually in whichever driver is returning `FILTER_STRAY` wrongly. Your own driver’s behaviour is correct as long as its status-register check is right.

19.19 Advanced Observability: Integrating With DTrace

FreeBSD’s DTrace can observe the interrupt path at many levels. This subsection shows some useful DTrace one-liners and scripts a driver author can use during development.

19.19.1 Counting Filter Invocations

```
sudo dtrace -n '  
fbt::myfirst_intr_filter:entry { @invocations = count(); }'
```

Shows the total number of times the filter has been called since DTrace started. Compare with `sysctl dev.myfirst.0.intr_count`; they should agree.

19.19.2 Measuring Filter Latency

```
sudo dtrace -n '  
fbt::myfirst_intr_filter:entry { self->ts = vtimestamp; }  
fbt::myfirst_intr_filter:return /self->ts/ {  
    @["filter_ns"] = quantize(vtimestamp - self->ts);  
    self->ts = 0;  
}'
```

`vtimestamp` measures CPU-time (not wall-clock), so the histogram is truly the filter's CPU time. A healthy filter is in the hundreds of nanoseconds to single-digit microseconds range.

19.19.3 Observing the Task Queue

```
sudo dtrace -n '  
fbt::myfirst_intr_data_task_fn:entry {  
    @["task_runs"] = count();  
    self->ts = vtimestamp;  
}  
fbt::myfirst_intr_data_task_fn:return /self->ts/ {  
    @["task_ns"] = quantize(vtimestamp - self->ts);  
    self->ts = 0;  
}'
```

Shows the task's invocation count and per-invocation execution time. The task is typically an order of magnitude slower than the filter (because it takes a sleep lock and does more work).

19.19.4 Correlating the Filter and Task

```
sudo dtrace -n '  
fbt::myfirst_intr_filter:entry /!self->in_filter/ {  
    self->in_filter = 1;  
    self->filter_start = vtimestamp;  
    @["filter_enters"] = count();  
}  
fbt::myfirst_intr_filter:return /self->in_filter/ {  
    self->in_filter = 0;  
}  
fbt::myfirst_intr_data_task_fn:entry {  
    @["task_starts"] = count();  
}'
```

If `filter_enters` is 100 and `task_starts` is 80, some filter invocations did not schedule a task (because the event was `ERROR` or `COMPLETE`, not `DATA_AV`).

19.19.5 Tracing Taskqueue Scheduling Decisions

The taskqueue infrastructure has DTrace probes too; one can observe how the task is enqueued and when the worker thread runs:

```
sudo dtrace -n '  
fbt::taskqueue_enqueue:entry /arg0 == ${tq_addr}/ {  
    @["enqueues"] = count();  
}'
```

where `${tq_addr}` is the numeric address of `sc->intr_tq`, obtainable through `kldstat` / `kgdb` combinations. This level of detail is usually overkill for Chapter 19's driver.

19.19.6 DTrace and the Simulated-Interrupt Path

Simulated interrupts are distinguishable from real interrupts because the simulated path goes through the `sysctl` handler:

```
sudo dtrace -n '  
fbt::myfirst_intr_simulate_sysctl:entry { @["simulate"] = count(); }  
fbt::myfirst_intr_filter:entry { @["filter"] = count(); }'
```

The difference between the two counts is the number of real interrupts (filter calls not preceded by a `sysctl` call).

19.20 Detailed Walkthrough: Stage 2 End to End

To make the Chapter 19 driver concrete, here is a complete walkthrough of what happens when a `DATA_AV` event is simulated through the `sysctl`, traced step by step.

19.20.1 The Sequence

1. User runs `sudo sysctl dev.myfirst.0.intr_simulate=1`.
2. The kernel's `sysctl` machinery routes the write to `myfirst_intr_simulate_sysctl`.
3. The handler parses the value (1), acquires `sc->mtx` via `MYFIRST_LOCK`.
4. The handler writes 1 to `INTR_STATUS` in the BAR.
5. The handler releases `sc->mtx`.
6. The handler calls `myfirst_intr_filter(sc)` directly.
7. The filter reads `INTR_STATUS` through `ICSR_READ_4`. Value is 1 (`DATA_AV`).
8. The filter increments `intr_count` atomically.
9. The filter sees `DATA_AV` bit set, increments `intr_data_av_count`.
10. The filter acknowledges by writing 1 back to `INTR_STATUS` via `ICSR_WRITE_4`.
11. The filter enqueues `intr_data_task` on `intr_tq` via `taskqueue_enqueue`.
12. The filter returns `FILTER_HANDLED`.
13. The `sysctl` handler returns 0 to the kernel's `sysctl` layer.
14. The user's `sysctl` command returns success.

Meanwhile, in the taskqueue:

15. The taskqueue's worker thread (woken by `taskqueue_enqueue`) schedules.
16. The worker thread calls `myfirst_intr_data_task_fn(sc, 1)`.
17. The task acquires `sc->mtx`.
18. The task reads `DATA_OUT` through `CSR_READ_4`.
19. The task stores the value in `sc->intr_last_data`.
20. The task increments `intr_task_invocations`.
21. The task broadcasts `sc->data_cv` (no waiters in this example).
22. The task releases `sc->mtx`.
23. The worker thread goes back to waiting for more work.

Steps 1-14 take microseconds; steps 15-23 take tens to hundreds of microseconds depending on scheduling.

19.20.2 What the Counters Show

After one simulated interrupt:

```
dev.myfirst.0.intr_count: 1
dev.myfirst.0.intr_data_av_count: 1
dev.myfirst.0.intr_error_count: 0
dev.myfirst.0.intr_complete_count: 0
dev.myfirst.0.intr_task_invocations: 1
```

If `intr_task_invocations` is still 0, the task has not run yet (usually because the `sysctl` returned before the worker thread was scheduled). A short `sleep 0.01` is enough.

19.20.3 What `dmesg` Shows

By default, nothing. The Stage 4 driver is not verbose. A reader who wants to see the filter fire can add `device_printf` calls for debugging, but production-quality drivers typically do not print on every interrupt.

19.20.4 What `vmstat -i` Shows

`vmstat -i | grep myfirst` shows the `intr_event`'s total count. This counts only real interrupts delivered by the kernel to our filter. Simulated interrupts invoked through the `sysctl` do not pass through the kernel's interrupt dispatcher, so they do not appear in the `vmstat -i` count.

This is a useful distinction: the `sysctl`-delivered simulation is a complementary mechanism, not a replacement. Real interrupts still count; simulated ones do not.

19.20.5 Tracing With Print Statements

For quick debugging, adding `device_printf` calls to the filter and task gives a live picture:

```
/* In the filter, temporarily for debugging: */
device_printf(sc->dev, "filter: status=0x%x\n", status);

/* In the task: */
device_printf(sc->dev, "task: data=0x%x npending=%d\n",
    data, npending);
```

This produces `dmesg` output like:

```
myfirst0: filter: status=0x1
myfirst0: task: data=0xdeadbeef npending=1
```

Remove these prints before production; the chatter at high interrupt rates is expensive.

19.21 Patterns From Real FreeBSD Drivers

A compact tour of interrupt patterns that appear repeatedly in `/usr/src/sys/dev/`. Each pattern is a concrete snippet from a real driver (rewritten slightly for readability) with a note on why it matters. Reading these after Chapter 19 consolidates the vocabulary.

19.21.1 Pattern: Fast-Filter-With-Slow-Task

From `/usr/src/sys/dev/mgb/if_mgb.c`:

```
int
mgb_legacy_intr(void *xsc)
{
    struct mgb_softc *sc = xsc;
    uint32_t intr_sts = CSR_READ_REG(sc, MGB_INTR_STS);
    uint32_t intr_en = CSR_READ_REG(sc, MGB_INTR_ENBL_SET);

    intr_sts &= intr_en;
    if ((intr_sts & MGB_INTR_STS_ANY) == 0)
        return (FILTER_STRAY);

    /* Acknowledge and defer per-queue work. */
    if ((intr_sts & MGB_INTR_STS_RX_ANY) != 0) {
        for (int qidx = 0; qidx < scctx->isc_nrxqsets; qidx++) {
            if (intr_sts & MGB_INTR_STS_RX(qidx))
                iflib_rx_intr_deferred(sc->ctx, qidx);
        }
        return (FILTER_HANDLED);
    }
    return (FILTER_SCHEDULE_THREAD);
}
```

Why it matters: the filter is short, the deferred work is per-queue, the shared-IRQ discipline is maintained. Chapter 19's filter follows the same shape.

19.21.2 Pattern: ithread-Only Handler

From `/usr/src/sys/dev/ath/if_ath_pci.c`:

```
bus_setup_intr(dev, psc->sc_irq,
    INTR_TYPE_NET | INTR_MPSAFE,
    NULL, ath_intr, sc, &psc->sc_ih);
```

The filter argument is `NULL`; `ath_intr` is the ithread handler. The kernel schedules `ath_intr` on every interrupt without a filter in the middle.

Why it matters: sometimes all the work needs thread context. Registering `NULL` for the filter is simpler than writing a trivial filter that just returns `FILTER_SCHEDULE_THREAD`.

19.21.3 Pattern: INTR_EXCL for Exclusive Access

Some drivers need exclusive access to an interrupt line:

```
bus_setup_intr(dev, irq,
    INTR_TYPE_BIO | INTR_MPSAFE | INTR_EXCL,
    NULL, driver_intr, sc, &cookie);
```

Why it matters: in rare cases, a driver needs the line to itself (the handler's assumption of being the only listener is baked in). INTR_EXCL asks the kernel to refuse other drivers on the same event.

19.21.4 Pattern: Short Debug Log

Some drivers have an optional verbose mode that logs every filter call:

```
if (sc->sc_debug > 0)
    device_printf(sc->sc_dev, "interrupt: status=0x%x\n", status);
```

Why it matters: a driver under development benefits from logging; a production driver wants the log suppressed. A sysctl (dev.driver.N.debug) toggles the mode.

19.21.5 Pattern: Bind to a Specific CPU

Drivers that know their topology bind the interrupt to a local CPU:

```
/* After bus_setup_intr: */
error = bus_bind_intr(dev, irq, local_cpu);
if (error != 0)
    device_printf(dev, "bus_bind_intr: %d\n", error);
/* Non-fatal: some platforms do not support binding. */
```

Why it matters: NUMA-local handlers are faster. A driver that bothers to bind produces a better scalability story on multi-socket systems.

19.21.6 Pattern: Describe the Handler for Diagnostics

Every driver should call bus_describe_intr:

```
bus_describe_intr(dev, irq, cookie, "rx-%d", queue_id);
```

Why it matters: `vmstat -i` and `devinfo -v` use the description to distinguish handlers on shared events. A driver with N queues and N MSI-X vectors has N bus_describe_intr calls.

19.21.7 Pattern: Quiesce Before Detach

```
mtx_lock(&sc->mtx);
sc->shutting_down = true;
mtx_unlock(&sc->mtx);

/* Let the interrupt handler drain. */
bus_teardown_intr(dev, sc->irq_res, sc->intr_cookie);
```

Why it matters: the `shutting_down` flag gives the handler a quick-exit path (the handler checks the flag before doing its normal work). The `bus_teardown_intr` is the definitive drain, but the flag makes the drain faster.

Chapter 19's driver uses `sc->pci_attached` for a similar purpose.

19.22 Reference: Common Mistakes Cheat Sheet

A compact list of interrupt-specific mistakes and their one-line fixes. Useful as a checklist when reviewing your own driver.

1. **No INTR_MPSAFE.** Fix: `flags = INTR_TYPE_MISC | INTR_MPSAFE.`
2. **Sleep lock in filter.** Fix: use atomic operations or a spin mutex.
3. **Missing acknowledgment.** Fix: `bus_write_4(res, INTR_STATUS, bits_handled);`.
4. **Acknowledging too many bits.** Fix: write back only the bits you handled.
5. **Missing FILTER_STRAY return.** Fix: if status is zero or unrecognised, return `FILTER_STRAY`.
6. **Missing FILTER_HANDLED return.** Fix: `rv |= FILTER_HANDLED;` for each recognised bit.
7. **Task using stale softc.** Fix: add `taskqueue_drain` to detach.
8. **Missing bus_teardown_intr.** Fix: before `bus_release_resource(SYS_RES_IRQ, ...)`.
9. **Missing INTR_MASK = 0 at detach.** Fix: clear the mask before teardown.
10. **Missing taskqueue_drain.** Fix: drain before freeing softc state.
11. **Wrong filter return value.** Fix: must be `FILTER_HANDLED`, `FILTER_STRAY`, `FILTER_SCHEDULE_THREAD`, or a bitwise OR.
12. **Enqueueing a task before TASK_INIT.** Fix: initialise the task in attach.
13. **Not setting INTR_MASK at attach.** Fix: write the bits you want enabled.
14. **Wrong rid for legacy IRQ.** Fix: use `rid = 0`.
15. **Wrong resource type.** Fix: use `SYS_RES_IRQ` for interrupts.
16. **Missing RF_SHAREABLE on shared line.** Fix: include the flag in allocation.
17. **Holding sc->mtx across bus_setup_intr.** Fix: release the lock before the call.
18. **Holding a spin lock across bus_teardown_intr.** Fix: never hold a filter's spin lock when tearing down.
19. **Taskqueue destroyed while task is still enqueued.** Fix: `taskqueue_drain` before `taskqueue_free`.
20. **Missing bus_describe_intr call.** Fix: add it after `bus_setup_intr` for diagnostic clarity.

19.23 Reference: ithread and Taskqueue Priorities

The Chapter 19 code uses `PI_NET` for the taskqueue. FreeBSD defines several priority constants in `/usr/src/sys/sys/priority.h`. A simplified view:

```

PI_REALTIME = PRI_MIN_ITHD + 0    (highest ithread priority)
PI_INTR     = PRI_MIN_ITHD + 4    (the common "hardware interrupt" level)
PI_AV       = PI_INTR             (audio/video)
PI_NET      = PI_INTR             (network)
PI_DISK     = PI_INTR             (block storage)
PI_TTY      = PI_INTR             (terminal/serial)
PI_DULL     = PI_INTR             (low-priority hardware ithreads)
PI_SOFT     = PRI_MIN_ITHD + 8    (soft interrupts)
PI_SOFTCLOCK = PI_SOFT            (soft clock)
PI_SWI(c)   = PI_SOFT            (per-category SWI)

```

A reader looking at this list will notice that most of the “hardware interrupt” aliases (`PI_AV`, `PI_NET`, `PI_DISK`, `PI_TTY`, `PI_DULL`) resolve to the same numeric value (`PI_INTR`). The comment at the top of that block in `priority.h` makes the reason explicit: “Most hardware interrupt threads run at the same priority, but can decay to lower priorities if they run for full time slices”. The category names exist because each one reads naturally at the call site, not because the numeric priorities differ.

Only `PI_REALTIME` (slightly above `PI_INTR`) and `PI_SOFT` (below `PI_INTR`) are actually distinct from the common hardware-interrupt level.

The `ithread`’s priority comes from the `INTR_TYPE_*` flag; the taskqueue’s priority is set explicitly. Passing `PI_NET` to `taskqueue_start_threads` puts the worker at the same nominal level as a network `ithread`, which is the right choice for work that cooperates with network-rate interrupt handling. A storage driver would pass `PI_DISK`; a low-priority background driver would pass `PI_DULL`. Because the constants all map to the same numeric value, the names are pragmatically interchangeable for correctness. They still matter for readability and for any future kernel where the distinction becomes real.

19.24 Reference: A Short Tour of /usr/src/sys/kern/kern_intr.c

A reader curious about what happens behind `bus_setup_intr(9)` and `bus_teardown_intr(9)` can open `/usr/src/sys/kern/kern_intr.c`. The file is about 1800 lines and has distinct sections:

- **intr_event management** (`intr_event_create`, `intr_event_destroy`): top-level creation and cleanup of the `intr_event` structure.
- **Handler management** (`intr_event_add_handler`, `intr_event_remove_handler`): the underlying operations that `bus_setup_intr` and `bus_teardown_intr` call.
- **Dispatch** (`intr_event_handle`, `intr_event_schedule_thread`): the code that actually runs when an interrupt fires.
- **Storm detection** (`intr_event_handle`): the `intr_storm_threshold` logic.
- **ithread creation and scheduling** (`ithread_create`, `ithread_loop`, `ithread_update`): the per-event `ithread` machinery.

- **SWI (software interrupt) management** (`swi_add`, `swi_sched`, `swi_remove`): soft interrupts.

A reader does not need to understand the whole file to write a driver. Browsing the top-level function list and reading the comments on `intr_event_handle` (the dispatch function) is a worthwhile half-hour.

19.24.1 Key Functions in `kern_intr.c`

Function	Purpose
<code>intr_event_create</code>	Allocate a new <code>intr_event</code> .
<code>intr_event_destroy</code>	Free an <code>intr_event</code> .
<code>intr_event_add_handler</code>	Attach a filter/ithread handler.
<code>intr_event_remove_handler</code>	Detach a handler.
<code>intr_event_handle</code>	Dispatch: called on each interrupt.
<code>intr_event_schedule_thread</code>	Wake the ithread.
<code>ithread_loop</code>	The body of an ithread.
<code>swi_add</code>	Register a software interrupt.
<code>swi_sched</code>	Schedule a software interrupt.

The `BUS_*` functions exposed to drivers (`bus_setup_intr`, `bus_teardown_intr`, `bus_bind_intr`, `bus_describe_intr`) call into these kernel-internal functions after platform-specific bus-driver hooks.

19.25 Wrapping Up

Chapter 19 gave the driver ears. At the start, `myfirst` at version `1.1-pci` was attached to a real PCI device but did not listen to it: every action the driver took was initiated by user space, and the device's own asynchronous events (if any) went unnoticed. At the end, `myfirst` at version `1.2-intr` has a filter handler wired to the device's IRQ line, a deferred-task pipeline that handles the bulk work in thread context, a simulated-interrupt path for testing on lab targets, a shared-IRQ discipline that coexists with other drivers on the same line, a clean teardown that releases every resource in the correct order, and a new `myfirst_intr.c` file plus `INTERRUPTS.md` document.

The transition walked through eight sections. Section 1 introduced interrupts at the hardware level, covering edge-triggered and level-triggered signalling, the CPU's dispatch flow, and the six obligations a driver's handler has. Section 2 introduced FreeBSD's kernel model: `intr_event`, `intr_handler`, `ithread`, the filter-plus-ithread split, `INTR_MPSAFE`, and the constraints on filter context. Section 3 wrote the minimal filter and the attach/detach wiring. Section 4 extended the filter with status decoding, per-bit acknowledgment, and taskqueue-based deferred work. Section 5 added the simulated-interrupt `sysctl` that lets the reader exercise the pipeline without real IRQ events. Section 6 codified the shared-IRQ discipline: check ownership, return `FILTER_STRAY` correctly, handle unrecognised bits defensively. Section 7 consolidated the teardown: mask at the device, tear down the handler, drain the taskqueue, release resources. Section 8 refactored the whole thing into a maintainable layout.

What Chapter 19 did not do is MSI, MSI-X, or DMA. The driver's interrupt path is a single legacy IRQ; the data path does not use DMA; the deferred work is a single taskqueue task. Chapter 20 introduces MSI and MSI-X (multiple vectors, per-vector filters, richer interrupt routing). Chapters 20 and 21 introduce DMA and the interaction between interrupts and DMA descriptor rings.

What Chapter 19 accomplished is the split between two threads of control. The driver's filter is short, runs in primary interrupt context, and handles the urgent per-interrupt work. The driver's deferred task is longer, runs in thread context, and handles the bulk per-event work. The discipline that keeps them cooperating (atomics for filter state, sleep locks for task state, strict ordering for teardown) is the discipline every later chapter's interrupt code assumes.

The file layout has grown: `myfirst.c`, `myfirst_hw.c`, `myfirst_hw_pci.c`, `myfirst_hw.h`, `myfirst_sim.c`, `myfirst_sim.h`, `myfirst_pci.c`, `myfirst_pci.h`, `myfirst_intr.c`, `myfirst_intr.h`, `myfirst_sync.h`, `cbuf.c`, `cbuf.h`, `myfirst.h`. The documentation has grown: `HARDWARE.md`, `LOCKING.md`, `SIMULATION.md`, `PCI.md`, `INTERRUPTS.md`. The test suite has grown: the simulated-interrupt pipeline, the Stage 4 regression script, a handful of challenge exercises to keep the reader practising.

19.25.1 A Reflection Before Chapter 20

A pause before the next chapter. Chapter 19 taught the filter-plus-task pattern, the `INTR_MPSAFE` promise, the constraints of interrupt context, and the shared-IRQ discipline. The patterns you practised here (read status, acknowledge, defer work, return the right `FILTER_*`, tear down cleanly) are patterns every FreeBSD interrupt handler uses. Chapter 20 will layer MSI-X on top; Chapter 21 will layer DMA on top. Neither chapter replaces the Chapter 19 patterns; both build on them.

A second observation worth making. The composition of Chapter 17's simulation, Chapter 18's real PCI attach, and Chapter 19's interrupt handling is now a complete driver in the architectural sense. A reader who understands the three layers can open any FreeBSD PCI driver and recognise the parts: the register map, the PCI attach, the interrupt filter. The specifics differ; the structure is constant. That recognition is what makes the book's investment pay off across the whole FreeBSD source tree.

Third observation: the payoff of Chapter 16's accessor layer continues. The `CSR_*` macros did not change in Chapter 19; the `ICSR_*` macros were added for filter-context use, but they call the same underlying `bus_read_4` and `bus_write_4`. The abstraction has now paid off three times: against the Chapter 17 simulation backend, against the Chapter 18 real PCI BAR, and against the Chapter 19 filter context. A reader who builds similar accessor layers in their own drivers will find the same dividend.

19.25.2 What to Do If You Are Stuck

Three suggestions.

First, focus on the simulated-interrupt path. If `sudo sysctl dev.myfirst.0.intr_simulate=1` makes counters tick and the task run, the pipeline is working. Every other piece of the chapter is optional in the sense that it decorates the pipeline, but if the pipeline fails, the whole chapter is not working and Section 5 is the right place to diagnose.

Second, open `/usr/src/sys/dev/mgb/if_mgb.c` and re-read the `mgb_legacy_intr` function slowly. It is about sixty lines of filter code. Every line maps to a Chapter 19 concept. Reading it once after completing the chapter should feel like familiar territory.

Third, skip the challenges on first pass. The labs are calibrated for Chapter 19's pace; the challenges assume the chapter's material is solid. Come back to them after Chapter 20 if they feel out of reach now.

Chapter 19's goal was to give the driver a way to listen to its device. If it has, Chapter 20's MSI-X machinery becomes a specialisation rather than an entirely new topic, and Chapter 21's DMA becomes a matter of wiring descriptor completions to the interrupt path you already have.

19.26 Bridge to Chapter 20

Chapter 20 is titled *Advanced Interrupt Handling*. Its scope is the specialisation that Chapter 19 deliberately did not take: MSI (Message Signaled Interrupts) and MSI-X, the modern PCIe interrupt mechanisms that replace the legacy INTx line with per-device (or per-queue) vectors delivered as memory writes.

Chapter 19 prepared the ground in four specific ways.

First, **you have a functioning filter handler**. The Chapter 19 filter reads status, handles bits, acknowledges, and defers. Chapter 20's filter looks similar, but is replicated per vector: each MSI-X vector has its own filter, and each handles a specific subset of the device's events.

Second, **you understand the attach/detach cascade**. Chapter 19 grew the cascade by two labels (`fail_release_irq`, `fail_teardown_intr`). Chapter 20 grows it further: one pair of labels per vector. The pattern does not change; the count does.

Third, **you have an interrupt teardown discipline**. Chapter 20 reuses Chapter 19's order: clear interrupts at the device, `bus_teardown_intr` for each vector, `bus_release_resource` for each IRQ resource. The per-vector nature adds a small loop; the order is the same.

Fourth, **you have a lab environment that exposes MSI-X**. On QEMU with `virtio-rng-pci`, MSI-X is available; on bhyve with `virtio-rnd`, only legacy INTx is exposed. Chapter 20's labs may require switching to QEMU or to a more richly-emulated bhyve device to exercise the MSI-X path.

Specific topics Chapter 20 will cover:

- Why MSI and MSI-X are an improvement over legacy INTx.
- How MSI differs from MSI-X (single vector vs table of vectors).
- `pci_alloc_msi(9)`, `pci_alloc_msix(9)`: allocating vectors.
- `pci_msi_count(9)`, `pci_msix_count(9)`: querying capability.
- `pci_release_msi(9)`: the teardown counterpart.
- Multi-vector interrupt handlers: per-queue filters.
- The MSI-X table layout and how to reach specific entries.
- CPU affinity across vectors for NUMA awareness.
- Interrupt coalescing: reducing the rate of interrupts when the device supports it.
- Interaction between MSI-X and iflib (the modern network driver framework).
- Migrating the `myfirst` driver from Chapter 19's legacy path to an MSI-X path, with a fallback to legacy for devices that do not support MSI-X.

You do not need to read ahead. Chapter 19 is sufficient preparation. Bring your `myfirst` driver at `1.2-intr`, your `LOCKING.md`, your `INTERRUPTS.md`, your WITNESS-enabled kernel, and your regression script. Chapter 20 starts where Chapter 19 ended.

Chapter 21 is one chapter further along; it is worth a brief forward pointer. DMA will introduce yet another interaction with interrupts: completion interrupts that signal “descriptor ring entry N is done”. The filter-plus-task discipline Chapter 19 taught carries through; the task’s work now involves walking a descriptor ring rather than reading a single register.

The vocabulary is yours; the structure is yours; the discipline is yours. Chapter 20 adds precision to all three.

19.27 Reference: Chapter 19 Quick-Reference Card

A compact summary of the vocabulary, APIs, macros, and procedures Chapter 19 introduced.

19.27.1 Vocabulary

- **Interrupt**: an asynchronous hardware-signalled event.
- **IRQ (Interrupt Request)**: the identifier of an interrupt line.
- **Edge-triggered**: signalled by a transition; one interrupt per transition.
- **Level-triggered**: signalled by a level held; an interrupt fires while the level is held.
- **intr_event**: FreeBSD’s kernel structure for one interrupt source.
- **ithread**: FreeBSD’s kernel thread that runs deferred interrupt handlers.
- **filter handler**: a function that runs in primary interrupt context.
- **ithread handler**: a function that runs in thread context after the filter.
- **FILTER_HANDLED**: the filter handled the interrupt; no ithread needed.
- **FILTER_SCHEDULE_THREAD**: the filter partially handled; run the ithread.
- **FILTER_STRAY**: the interrupt was not for this driver.
- **INTR_MPSAFE**: a flag promising the handler does its own synchronisation.
- **INTR_TYPE_*** (TTY, BIO, NET, CAM, MISC, CLK, AV): handler-category hints.
- **INTR_EXCL**: exclusive interrupt.

19.27.2 Essential APIs

- `bus_alloc_resource_any(dev, SYS_RES_IRQ, &rid, flags)`: claim an IRQ.
- `bus_release_resource(dev, SYS_RES_IRQ, rid, res)`: release the IRQ.
- `bus_setup_intr(dev, res, flags, filter, ihand, arg, &cookie)`: register a handler.
- `bus_teardown_intr(dev, res, cookie)`: unregister the handler.
- `bus_describe_intr(dev, res, cookie, "name")`: name the handler for tools.
- `bus_bind_intr(dev, res, cpu)`: route the interrupt to a CPU.
- `pci_msi_count(dev), pci_msix_count(dev)` (Chapter 20).
- `pci_alloc_msi(dev, &count), pci_alloc_msix(dev, &count)` (Chapter 20).
- `pci_release_msi(dev)` (Chapter 20).
- `taskqueue_create("name", M_WAITOK, taskqueue_thread_enqueue, &tq)`: create a taskqueue.
- `taskqueue_start_threads(&tq, n, PI_pri, "thread name")`: start workers.
- `taskqueue_enqueue(tq, &task)`: enqueue a task.
- `taskqueue_drain(tq, &task)`: wait for a task to finish, prevent new enqueues.
- `taskqueue_free(tq)`: free the taskqueue.
- `TASK_INIT(&task, pri, fn, arg)`: initialise a task.

19.27.3 Essential Macros

- `FILTER_HANDLED`, `FILTER_STRAY`, `FILTER_SCHEDULE_THREAD`.
- `INTR_TYPE_TTY`, `INTR_TYPE_BIO`, `INTR_TYPE_NET`, `INTR_TYPE_CAM`, `INTR_TYPE_MISC`, `INTR_TYPE_CLK`, `INTR_TYPE_AV`.
- `INTR_MPSAFE`, `INTR_EXCL`.
- `RF_SHAREABLE`, `RF_ACTIVE`.
- `SYS_RES_IRQ`.

19.27.4 Common Procedures

Allocate a legacy PCI interrupt and register a filter handler:

1. `sc->irq_rid = 0;`
2. `sc->irq_res = bus_alloc_resource_any(dev, SYS_RES_IRQ, &sc->irq_rid, RF_SHAREABLE | RF_ACTIVE);`
3. `bus_setup_intr(dev, sc->irq_res, INTR_TYPE_MISC | INTR_MPSAFE, filter, NULL, sc, &sc->intr_cookie);`
4. `bus_describe_intr(dev, sc->irq_res, sc->intr_cookie, "name");`

Tear down an interrupt handler:

1. Disable interrupts at the device (clear `INTR_MASK`).
2. `bus_tear_down_intr(dev, sc->irq_res, sc->intr_cookie);`
3. `taskqueue_drain(sc->intr_tq, &sc->intr_data_task);`
4. `taskqueue_free(sc->intr_tq);`
5. `bus_release_resource(dev, SYS_RES_IRQ, sc->irq_rid, sc->irq_res);`

Write a filter handler:

1. Read `INTR_STATUS`; if zero, return `FILTER_STRAY`.
2. For each recognised bit, increment a counter, acknowledge by writing back, and optionally enqueue a task.
3. Return `FILTER_HANDLED` (or `FILTER_SCHEDULE_THREAD`), or `FILTER_STRAY` if nothing was recognised.

19.27.5 Useful Commands

- `vmstat -i`: list interrupt sources with counts.
- `devinfo -v`: list devices with their IRQ resources.
- `sysctl hw.intrcnt` and `sysctl hw.intrnames`: raw counters.
- `sysctl hw.intr_storm_threshold`: enable kernel storm detection.
- `cpuset -g`: query interrupt CPU affinity (platform-specific).
- `sudo sysctl dev.myfirst.0.intr_simulate=1`: fire a simulated interrupt.

19.27.6 Files to Keep Bookmarked

- `/usr/src/sys/sys/bus.h`: `driver_filter_t`, `driver_intr_t`, `FILTER_*`, `INTR_*`.

- `/usr/src/sys/kern/kern_intr.c`: the kernel's interrupt-event machinery.
- `/usr/src/sys/sys/taskqueue.h`: taskqueue API.
- `/usr/src/sys/dev/mgb/if_mgb.c`: a readable filter-plus-task example.
- `/usr/src/sys/dev/ath/if_ath_pci.c`: a minimal ithread-only interrupt setup.

19.28 Reference: A Comparison Table Across Part 4

A compact summary of where each chapter of Part 4 fits, what it adds, and what it assumes. Useful for readers jumping in or back through the part.

Topic	Ch 16	Ch 17	Ch 18	Ch 19	Ch 20 (preview)	Ch 21 (preview)
BAR access	Simulated with malloc	Extended with sim layer	Real PCI BAR	Same	Same	Same
Chapter 17 simulation	N/A	Introduced	Inactive on PCI	Inactive on PCI	Inactive on PCI	Inactive on PCI
PCI attach	N/A	N/A	Introduced	Same + IRQ	MSI-X option	DMA init added
Interrupt handling	N/A	N/A	N/A	Introduced	MSI-X per-vector	Completion-driven
DMA	N/A	N/A	N/A	N/A	Preview	Introduced
Version	0.9-mmio	1.0-simulated	1.1-pci	1.2-intr	1.3-msi	1.4-dma
New file	<code>myfirst_hw.c</code>	<code>myfirst_sim.c</code>	<code>myfirst_pci.c</code>	<code>myfirst_intr.c</code>	<code>myfirst_msix.c</code>	<code>myfirst_dma.c</code>
Key discipline	Accessor abstraction	Fake device	Newbus attach	Filter/task split	Per-vector handlers	DMA maps

The table makes the book's cumulative structure visible at a glance. A reader who understands the row for a given topic can predict how Chapter 19's work fits into the bigger picture.

19.29 Reference: FreeBSD Manual Pages for Chapter 19

A list of the manual pages most useful for Chapter 19's material. Open each with `man 9 <name>` (for kernel APIs) or `man 4 <name>` (for subsystem overviews) on a FreeBSD system.

19.29.1 Kernel API Manual Pages

- **`bus_setup_intr(9)`**: registering an interrupt handler.

- **bus_teardown_intr(9)**: tearing down a handler.
- **bus_bind_intr(9)**: binding to a CPU.
- **bus_describe_intr(9)**: labelling a handler.
- **bus_alloc_resource(9)**: resource allocation (generic).
- **bus_release_resource(9)**: resource release.
- **atomic(9)**: atomic operations including `atomic_add_64`.
- **taskqueue(9)**: taskqueue primitives.
- **ppsratecheck(9)**: rate-limited logging helper.
- **swi_add(9)**: software interrupts (mentioned as alternative).
- **intr_event(9)**: interrupt event machinery (if present; some APIs are internal).

19.29.2 Device Subsystem Manual Pages

- **pci(4)**: PCI subsystem.
- **vmstat(8)**: `vmstat -i` for observing interrupts.
- **devinfo(8)**: device tree and resources.
- **devctl(8)**: runtime device control.
- **sysctl(8)**: reading and writing sysctls.
- **dtrace(1)**: dynamic tracing.

Most of these have been referenced in the chapter body. This consolidated list is for readers who want a single place to find them.

19.30 Reference: Driver Memorable Phrases

A few aphorisms that summarise Chapter 19's discipline. Useful for reading and for code review.

- **"Read, acknowledge, defer, return."** The four things a filter does.
- **"FILTER_STRAY if you didn't recognise anything."** The shared-IRQ protocol.
- **"Mask before teardown; teardown before release."** The detach ordering.
- **"Filter context is spin-lock-only."** The no-sleep-locks rule.
- **"Every enqueue needs a drain before free."** The taskqueue lifecycle.
- **"One filter, one device, one state."** The isolation that keeps per-device code sane.
- **"If WITNESS panics, believe it."** The debug kernel catches subtle mistakes.
- **"PROD first, interrupt second."** Program the device (`INTR_MASK`) before enabling the handler.
- **"Small in the filter; large in the task."** The work-size discipline.
- **"Storm detection is a safety net, not a design tool."** Do not rely on the kernel's throttling.

None of these are complete specifications. Each is a compact reminder that unpacks into the chapter's detailed treatment.

19.31 Reference: Glossary of Chapter 19 Terms

ack (acknowledge): the operation of writing back to `INTR_STATUS` to clear the pending bit and deassert the IRQ line.

driver_filter_t: the C typedef for a filter-handler function: `int f(void *)`.

driver_intr_t: the C typedef for an ithread handler function: `void f(void *)`.

edge-triggered: an interrupt signalling mode where the interrupt is signalled by a level transition.

FILTER_HANDLED: return value from a filter meaning “this interrupt is handled; no ithread needed”.

FILTER_SCHEDULE_THREAD: return value meaning “schedule the ithread to run”.

FILTER_STRAY: return value meaning “this interrupt is not for this driver”.

filter handler: a C function running in primary interrupt context.

Giant: the legacy single global kernel lock; modern drivers avoid it by setting `INTR_MPSAFE`.

IE (interrupt event): short for `intr_event`.

INTR_MPSAFE: a flag promising the handler does its own synchronisation and is safe without Giant.

INTR_STATUS: the device register that tracks pending interrupt causes (RW1C).

INTR_MASK: the device register that enables specific interrupt classes.

intr_event: kernel structure representing one interrupt source.

ithread: kernel interrupt thread; runs deferred handlers in thread context.

level-triggered: an interrupt signalling mode where the interrupt fires while the level is held.

MSI: Message Signaled Interrupts; a PCIe mechanism (Chapter 20).

MSI-X: the richer variant of MSI with a table of vectors (Chapter 20).

primary interrupt context: the context of a filter handler; no sleeping, no sleep locks.

PCIR_INTLINE / PCIR_INTPIN: PCI configuration-space fields specifying the legacy IRQ line and pin.

RF_ACTIVE: resource allocation flag; activate the resource in one step.

RF_SHAREABLE: resource allocation flag; allow sharing the resource with other drivers.

stray interrupt: an interrupt for which no filter returned a claim; counted separately by the kernel.

storm: a situation in which a level-triggered interrupt fires continuously because the driver does not acknowledge.

SYS_RES_IRQ: resource type for interrupts.

taskqueue: a kernel primitive for running deferred work in thread context.

trap stub: the small piece of kernel code that runs when the CPU takes an interrupt vector.

EOI (End of Interrupt): the signal sent to the interrupt controller to re-arm the IRQ line.

19.32 Reference: A Closing Note on Interrupt-Handling Philosophy

A paragraph to close the chapter with, worth returning to after the labs.

An interrupt handler’s job is not to do the device’s work. The device’s work (processing a packet, finishing an I/O, reading a sensor) is done by the rest of the driver, in thread context, under the driver’s full set of locks.

The handler's job is narrower: to notice that the device has something to say, to acknowledge the device so the conversation can continue, to schedule the real work that will happen later, and to return quickly enough that the CPU is free for the interrupted thread or for the next interrupt.

A reader who has written the Chapter 19 driver has written one interrupt handler. It is small. The rest of the driver is what makes it useful. Chapter 20 will specialise the handler to per-vector work on MSI-X. Chapter 21 will specialise the task to walking a DMA descriptor ring. Each of those is an extension, not a replacement. The Chapter 19 handler is the skeleton that both build on.

The skill Chapter 19 teaches is not “how to handle interrupts for the virtio-rnd device”. It is “how to split work between primary context and thread context, how to respect the filter's constraints, how to tear down cleanly, and how to cooperate with other drivers on a shared line”. Each of those is a transferable skill. Every driver in the FreeBSD tree exercises some of them; most drivers exercise all of them.

For this reader and for this book's future readers, the Chapter 19 filter and task are a permanent part of the `myfirst` driver's architecture. Every later chapter assumes them. Every later chapter extends them. The driver's overall complexity will grow, but the interrupt path will remain what Chapter 19 made it: a narrow, fast, correctly-ordered piece of code that gets out of the way so the rest of the driver can do its job.

20 Advanced Interrupt Handling

20.1 Reader Guidance & Outcomes

Chapter 19 ended with a driver that could listen to its device. The `myfirst` module at version `1.2-intr` has one filter handler registered on the legacy PCI INTx line, one deferred-work task on a taskqueue, a simulated-interrupt `sysctl` for testing on the bhyve lab target, a strict teardown order, and a new `myfirst_intr.c` file that keeps the interrupt code tidy. The Chapter 11 locking discipline carries through: atomics in the filter, sleep locks in the task, `INTR_MPSAFE` on the handler, shared-IRQ safety via `FILTER_STRAY`. The driver behaves like a small real driver that happens to have a single source of interrupts.

What the driver does not yet do is take advantage of everything PCIe offers. A modern PCIe device does not need to share a single line with its neighbours. It can request a dedicated interrupt through the message-signalled mechanism PCI 2.2 introduced (MSI) or the richer per-function table MSI-X added in PCIe. A device with several queues (a NIC with receive and transmit queues, an NVMe controller with admin and I/O submission queues, a modern USB3 host controller with event queues) usually wants an interrupt per queue rather than one shared interrupt for the whole device. Chapter 20 teaches the driver how to ask.

The chapter's scope is precisely this transition: what MSI and MSI-X are at the hardware level, how FreeBSD represents them in terms of additional IRQ resources, how a driver queries capability counts and allocates vectors, how the fallback ladder from MSI-X down to MSI down to legacy INTx works in practice, how to register several different filter functions on the same device, how to design per-vector data structures so each vector's handler touches its own state, how to give each vector a CPU affinity that matches the device's NUMA placement, how to label each vector with `bus_describe_intr(9)` so `vmstat -i` tells the operator which vector does what, and how to tear every one of these things down in the correct order. The chapter stops short of DMA, which is Chapter 21; per-queue receive and transmit vectors become especially valuable once a descriptor ring is in the picture, but teaching both at once would dilute both.

Chapter 20 keeps several neighbouring topics at arm's length. Full DMA (`bus_dma(9)` tags, descriptor rings, bounce buffers, cache coherence) is Chapter 21. `iflib`'s multi-queue framework, which wraps MSI-X with a layer of per-queue `iflib` machinery, is a Part 6 topic (Chapter 28) for readers who want the `iflib`-style networking path. The richer per-function MSI-X mask table operations (steering specific message addresses to specific CPUs through the MSI-X table directly) are discussed but not implemented end to end. Platform-specific interrupt remapping through the IOMMU, SR-IOV vector sharing, and PCIe AER-driven interrupt recovery are left for later chapters. Chapter 20 stays inside the ground it can cover well and hands off explicitly when a topic deserves its own chapter.

Multi-vector work rests on every earlier Part 4 layer. Chapter 16 gave the driver a vocabulary of register access. Chapter 17 taught it to think like a device. Chapter 18 introduced it to a real PCI device. Chapter 19 gave it ears on a single IRQ. Chapter 20 gives it a set of ears, one per conversation the device wants to have. Chapter 21 will teach those ears to cooperate with the device's own ability to reach into RAM. Each chapter adds one

layer. Each layer depends on the ones before it. Chapter 20 is where the driver stops pretending the device has only one thing to say and starts treating it as the multi-queue machine it really is.

20.1.1 Why MSI-X Earns a Chapter of Its Own

At this point you may be asking why MSI and MSI-X need a chapter of their own. The Chapter 19 driver has a working interrupt handler on the legacy IRQ line. If the filter-plus-task pipeline is already right, why not keep using it? Does MSI-X really justify a full chapter of new material?

Three reasons.

The first is scale. A single IRQ line on a shared system forces every driver on that line to serialise through one `intr_event`. On a host with dozens of PCIe devices, the legacy INTx mechanism would bottleneck the whole system if it were the only option. MSI-X lets each device (and each queue within a device) have its own dedicated `intr_event`, served by its own ithread or filter handler, pinned to its own CPU. The difference between a modern server handling ten million packets per second with MSI-X and the same workload on legacy INTx is the difference between “possible” and “impossible”; MSI-X is what makes the former a reality.

The second is locality. With a single interrupt line, the kernel has one choice of CPU to route the interrupt to, and that choice is global for the device. With MSI-X, each vector can be pinned to a different CPU, and good drivers pin each vector to a CPU that is NUMA-local to the queue it serves. The cache-line advantages of doing this are real: a receive queue whose interrupt fires on the same CPU that eventually consumes the packet avoids cross-socket cache traffic that dominates on legacy setups.

The third is cleanliness. Even for a driver that does not need high throughput, MSI or MSI-X can simplify the handler. With a dedicated line, the filter does not need to handle the shared-IRQ case. With a dedicated vector per event class (admin, receive, transmit, error), each handler is smaller and more specialised, and the whole driver becomes easier to read. Good drivers use MSI-X even when performance does not require it, because the code becomes better.

Chapter 20 earns its place by teaching all three of these benefits concretely. A reader finishes the chapter able to allocate vectors, route them, describe them, and tear them down, with a working driver that demonstrates the pattern end to end.

20.1.2 Where Chapter 19 Left the Driver

A brief recap of where you should stand. Chapter 20 extends the driver produced at the end of Chapter 19 Stage 4, tagged as version `1.2-intr`. If any of the items below feels uncertain, return to Chapter 19 before starting this chapter.

- Your driver compiles cleanly and identifies itself as `1.2-intr` in `kldstat -v`.
- On a bhyve or QEMU guest that exposes a `virtio-rnd` device, the driver attaches, allocates BAR 0 as `SYS_RES_MEMORY`, allocates the legacy IRQ as `SYS_RES_IRQ` with `rid = 0`, registers a filter handler through `bus_setup_intr(9)` with `INTR_TYPE_MISC | INTR_MPSAFE`, creates `/dev/myfirst0`, and supports the `dev.myfirst.N.intr_simulate` sysctl.
- The filter reads `INTR_STATUS`, increments per-bit counters, acknowledges, enqueues the deferred task for `DATA_AV`, and returns the right `FILTER_*` value.

- The task (`myfirst_intr_data_task_fn`) runs in thread context on a taskqueue named `myfirst_intr` at `PI_NET` priority, reads `DATA_OUT`, updates the `softc`, and broadcasts `sc->data_cv`.
- The detach path clears `INTR_MASK`, calls `bus_tearardown_intr`, drains and frees the taskqueue, releases the IRQ resource, detaches the hardware layer, and releases the BAR.
- `HARDWARE.md`, `LOCKING.md`, `SIMULATION.md`, `PCI.md`, and `INTERRUPTS.md` are current.
- `INVARIANTS`, `WITNESS`, `WITNESS_SKIPSPIN`, `DDB`, `KDB`, and `KDB_UNATTENDED` are enabled in your test kernel.

That driver is what Chapter 20 extends. The additions are noticeable in scope: one new file (`myfirst_msix.c`), one new header (`myfirst_msix.h`), several new `softc` fields to track per-vector state, a new family of per-vector filter functions, a new fallback ladder in the setup helper, per-vector `bus_describe_intr` calls, optional CPU binding, a version bump to 1.3-`msi`, a new `MSIX.md` document, and updates to the regression test. The mental model grows too: the driver starts thinking of interrupts as a vector of sources rather than a single event stream.

20.1.3 What You Will Learn

By the time you move on to the next chapter, you will be able to:

- Describe what an MSI and an MSI-X interrupt is at the hardware level, how each is signalled over PCIe (as a memory write rather than an electrical level change), and why the two mechanisms coexist with legacy INTx.
- Explain the key differences between MSI and MSI-X: the MSI vector count (1 to 32 from a contiguous block), the MSI-X vector count (up to 2048 independently-addressable vectors), and the per-vector address and mask capabilities MSI-X offers that MSI does not.
- Query a device's MSI and MSI-X capabilities through `pci_msi_count(9)` and `pci_msix_count(9)`, and know what the returned count means.
- Allocate MSI or MSI-X vectors through `pci_alloc_msi(9)` and `pci_alloc_msix(9)`, handle the case where the kernel allocates fewer vectors than requested, and recover from allocation failures.
- Build a three-tier fallback ladder: MSI-X first (if available), then MSI (if MSI-X is unavailable or allocation failed), then legacy INTx. Each tier uses the same `bus_setup_intr` pattern at its core, but the `rid` and handler-per-vector structure differ.
- Allocate per-vector IRQ resources with the correct `rid` (`rid=0` for legacy INTx; `rid=1, 2, 3, ...` for MSI and MSI-X vectors).
- Register a distinct filter handler per vector so that each vector has its own purpose (admin, receive-queue-N, transmit-queue-N, error).
- Design per-vector state (per-queue counters, per-queue task, per-queue lock) so that handlers running concurrently on different CPUs do not contend on shared data.
- Describe each vector with `bus_describe_intr(9)` so that `vmstat -i` and `devinfo -v` show each vector with a meaningful name.
- Bind each vector to a specific CPU with `bus_bind_intr(9)`, and query the device's NUMA-local CPU set with `bus_get_cpus(9)` using `LOCAL_CPUS` or `INTR_CPUS`.
- Handle partial-allocation failures: the device has eight vectors, the kernel gave us three; adjust the driver to use the three and do the remaining work through polling or scheduled tasks.
- Tear down a multi-vector driver correctly: per-vector `bus_tearardown_intr`, per-vector `bus_release_resource`, then a single `pci_release_msi(9)` at the end.

- Log a single clear dmesg summary line at attach time stating the interrupt mode (MSI-X / N vectors, MSI / K vectors, or legacy INTx), so the operator instantly sees which tier the driver ended up using.
- Split the multi-vector code into `myfirst_msix.c`, update the module's SRCS line, tag the driver as 1.3-`msi`, and produce `MSIX.md` documenting per-vector purposes and observed counter patterns.

The list is long; each item is narrow. The point of the chapter is the composition.

20.1.4 What This Chapter Does Not Cover

Several adjacent topics are explicitly deferred so Chapter 20 stays focused.

- **DMA.** `bus_dma(9)` tags, `bus_dmamap_load(9)`, scatter-gather lists, bounce buffers, cache coherence around DMA descriptors, and the way the device writes completions to RAM are Chapter 21. Chapter 20 gives the driver multiple vectors; Chapter 21 gives the device the ability to move data. Each half is independently valuable; together they are the backbone of every modern performance driver.
- **iflib(9) and the multi-queue networking framework.** `iflib` is a thick, opinionated framework that wraps MSI-X with per-queue ithreads, per-queue DMA pools, and a lot of machinery a generic driver does not need. Chapter 20 teaches the raw pattern; Part 6's networking chapter (Chapter 28) revisits it in `iflib`'s vocabulary.
- **PCIe AER recovery through MSI-X vectors.** Advanced Error Reporting can signal through its own MSI-X vector on some devices. Chapter 20 mentions the possibility; the full recovery path is a later-chapter topic.
- **SR-IOV and per-VF interrupts.** A Single-Root IO Virtualization virtual function has its own MSI-X capability and its own per-VF vectors. Chapter 20's driver is a physical function; the VF story is a later-chapter specialisation.
- **Thread priority tuning per vector.** A driver can pass a different priority to each vector's `bus_setup_intr` flags, or use `taskqueue_start_threads` at different priorities per vector. Chapter 20 uses `INTR_TYPE_MISC | INTR_MPSAFE` for each vector and does not tune priorities; Part 7's performance chapters (Chapter 33) cover the tuning story.
- **Modern-virtio-PCI transport using PCIe capabilities.** The `virtio_pci_modern(4)` driver puts the virtqueue notifications inside capability structures and uses MSI-X vectors for virtqueue completions. Chapter 20's driver still targets a legacy `virtio-rnd` BAR; a reader adapting it to a real production device would follow the Chapter 20 pattern but read from the modern virtio PCI layout.

Staying inside those lines keeps Chapter 20 a chapter about multi-vector interrupt handling. The vocabulary is what transfers; the specific chapters that follow apply the vocabulary to DMA, `iflib`, AER, and SR-IOV.

20.1.5 Estimated Time Investment

- **Reading only:** four to five hours. The MSI/MSI-X conceptual model is not complex, but the per-vector discipline, the fallback ladder, and the CPU-affinity story benefit from a careful read.
- **Reading plus typing the worked examples:** ten to twelve hours over two or three sessions. The driver evolves in four stages: fallback ladder, multiple vectors, per-vector handlers, refactor. Each stage is small but requires careful attention to per-vector state.

- **Reading plus all labs and challenges:** sixteen to twenty hours over four or five sessions, including reading real drivers (`virtio_pci.c`, `if_em.c`'s MSI-X code, `nvme.c`'s admin+IO vector split), setting up a bhyve or QEMU guest with MSI-X exposed, and running the chapter's regression test.

Sections 3, 5, and 6 are the densest. If the per-vector handler pattern feels unfamiliar on first pass, that is normal. Stop, re-read Section 3's diagram, and continue when the shape has settled.

20.1.6 Prerequisites

Before starting this chapter, confirm:

- Your driver source matches Chapter 19 Stage 4 (1.2-intr). The starting point assumes every Chapter 19 primitive: the filter plus task pipeline, the simulated-interrupt `sysctl`, the `ICSR_*` accessor macros, the clean teardown.
- Your lab machine runs FreeBSD 14.3 with `/usr/src` on disk and matching the running kernel.
- A debug kernel with `INVARIANTS`, `WITNESS`, `WITNESS_SKIPSPIN`, `DDB`, `KDB`, and `KDB_UNATTENDED` is built, installed, and booting cleanly.
- `bhyve(8)` or `qemu-system-x86_64` is available. For MSI-X labs the guest must expose a device whose MSI-X capability is enabled. QEMU's `virtio-rng-pci` has MSI-X; bhyve's `virtio-rnd` uses legacy `virtio` and does not expose MSI-X on the host driver as a matter of default configuration. The chapter calls out which labs require which environment.
- The `devinfo(8)`, `vmstat(8)`, `pciconf(8)`, and `cpuset(1)` tools are in your path.

If any item above is shaky, fix it now. MSI-X tends to expose any latent weakness in the driver's interrupt-context discipline because multiple handlers can run on multiple CPUs at the same time; the debug kernel's `WITNESS` is especially valuable during Chapter 20 development.

20.1.7 How to Get the Most Out of This Chapter

Four habits will pay off quickly.

First, keep `/usr/src/sys/dev/pci/pci.h` and `/usr/src/sys/dev/pci/pcivar.h` bookmarked along with the new files `/usr/src/sys/dev/pci/pci.c` and `/usr/src/sys/dev/virtio/pci/virtio_pci.c`. The first two are from Chapter 18 and define the capability constants (`PCIY_MSI`, `PCIY_MSIX`, `PCIM_MSIXCTRL_*`) and the accessor wrappers. The third is the kernel's implementation of `pci_msi_count_method`, `pci_alloc_msi_method`, `pci_alloc_msix_method`, and `pci_release_msi_method`. The fourth is a clean real-driver example of the full MSI-X allocation ladder with fallback. Each file rewards half an hour of reading.

Second, run `pciconf -lvc` on your lab host and a guest. The `-c` flag tells `pciconf` to print each device's capability list, and you will see which devices expose MSI, MSI-X, or both. Looking at your own machine is the quickest way to understand why MSI-X is the default everywhere in modern PCIe.

Third, type the changes by hand and run each stage. MSI-X code is where subtle per-vector mistakes produce bugs that only appear under concurrent load. Typing carefully, watching `dmesg` for attach banners, and running the regression test after each stage catches these mistakes at the point where they are cheap to fix.

Fourth, read `/usr/src/sys/dev/nvme/nvme_ctrlr.c`'s MSI-X setup (look for `nvme_ctrlr_allocate_bar` and `nvme_ctrlr_construct_admin_qpair`) after Section 5. `nvme(4)` is a clean real-driver example of the

admin-plus-N-queues pattern that Chapter 20 teaches. The file is long but the MSI-X code is a small fraction of it; the rest of the reading is optional but educational.

20.1.8 Roadmap Through the Chapter

The sections in order are:

1. **What Are MSI and MSI-X?** The hardware picture: how message-signalled interrupts work over PCIe, the difference between MSI and MSI-X, and why modern devices prefer them.
2. **Enabling MSI in Your Driver.** The simpler of the two modes. Query count, allocate, register a handler. Stage 1 of the Chapter 20 driver (`1.3-msi-stage1`).
3. **Managing Multiple Interrupt Vectors.** The core of the chapter. Per-vector rid, per-vector filter function, per-vector softc state, per-vector `bus_describe_intr`. Stage 2 (`1.3-msi-stage2`).
4. **Designing Interrupt-Safe Data Structures.** Why multi-vector means multi-CPU, what locks each vector's handler may and may not touch, how to structure per-queue state. A discipline, not a stage bump.
5. **Using MSI-X for High Flexibility.** The fuller mechanism. Table layout, per-vector binding, NUMA-aware placement with `bus_get_cpus`. Stage 3 (`1.3-msi-stage3`).
6. **Handling Vector-Specific Events.** Per-vector handler functions, per-vector deferred work, a pattern the `nvme(4)` driver uses at scale.
7. **Teardown and Cleanup With MSI/MSI-X.** Per-vector teardown, then a single `pci_release_msi` call. The ordering rules that keep everything safe.
8. **Refactoring and Versioning Your Multi-Vector Driver.** The final split into `myfirst_msix.c`, the new `MSIX.md`, the version bump to `1.3-msi`, and the regression pass. Stage 4.

After the eight sections come hands-on labs, challenge exercises, a troubleshooting reference, a Wrapping Up that closes Chapter 20's story and opens Chapter 21's, and a bridge to Chapter 21. The reference-and-cheat-sheet material at the end of the chapter is meant to be re-read as you work through Chapter 21; Chapter 20's vocabulary (vector, rid, per-vector softc, affinity, teardown ordering) is the foundation Chapter 21's DMA work assumes.

If this is your first pass, read linearly and do the labs in order. If you are revisiting, Sections 3 and 5 stand alone and make good single-sitting reads.

20.2 Section 1: What Are MSI and MSI-X?

Before the driver code, the hardware picture. Section 1 teaches what message-signalled interrupts are at the level of the PCIe bus and the interrupt controller, without any FreeBSD-specific vocabulary. A reader who understands Section 1 can read the rest of the chapter with the kernel's MSI/MSI-X path as a concrete object rather than a vague abstraction.

20.2.1 The Problem With Legacy INTx

Chapter 19 taught the legacy PCI INTx interrupt model: each PCI function has one interrupt line (usually one of INTA, INTB, INTC, INTD), the line is level-triggered, and multiple devices on the same physical line

share it. Chapter 19's driver handled the shared case correctly by reading `INTR_STATUS` first and returning `FILTER_STRAY` when nothing was set.

INTx works. But it has three problems that grow in importance as systems scale.

The first is **sharing overhead**. A shared line with ten devices requires every driver to be called on every interrupt, just to read its own status register and discover the interrupt was not for it. On a system where most interrupts are legitimate (the line is busy), this is a few extra `bus_read_4` calls per event; on a system where one device storms, every other driver's filter runs unnecessarily. The CPU cost is small per event but adds up across millions of events per second.

The second is **no per-queue separation**. A modern NIC has four, eight, sixteen, or sixty-four receive queues and a matching number of transmit queues. Each queue wants its own interrupt: when receive queue 3 has packets, only the handler for receive queue 3 should run, on a CPU close to the memory that queue uses. With INTx the device has only one line, so either the driver polls every queue from one handler (expensive and slow) or the device only supports one queue (unacceptable for a ten-gigabit NIC).

The third is **no CPU affinity per event type**. A shared line fires on one CPU (whichever the interrupt controller routes it to). On a NUMA system where the device is attached to socket 0, firing the interrupt on a CPU in socket 1 is worse than firing it on a CPU in socket 0: the handler's code runs on socket 1 but the device's memory lives on socket 0, and every register read crosses the inter-socket fabric. With INTx the driver cannot say "please fire this interrupt on CPU 3"; the kernel picks and the driver has no influence per event type.

MSI and MSI-X fix all three. The mechanism is fundamentally different from INTx: instead of electrical signalling over a dedicated wire, the device performs a memory write to a specific address, and the CPU's interrupt controller treats the write as an interrupt. This decouples the number of interrupts from the number of physical wires, lets each message-signalled interrupt have its own destination address (and therefore its own CPU), and eliminates the shared-line problem entirely.

20.2.2 How an MSI-Triggered Interrupt Actually Happens

Physically, an MSI interrupt is a write transaction on the PCIe fabric. The device issues a write of a specific value to a specific address. The memory controller recognises the address as belonging to the interrupt controller's MSI region, and routes the write to the APIC (or GIC, or whatever the platform's interrupt controller is). The interrupt controller decodes the address to determine which CPU should receive the interrupt, and decodes the written value to determine which vector (which entry in the IDT or equivalent) the CPU should run. The CPU then dispatches as it would for any interrupt: save state, jump to the vector's handler, run the kernel's interrupt dispatch.

From the driver's perspective, the flow is almost identical to legacy INTx:

1. The device has an event.
2. The interrupt fires.
3. The kernel calls the driver's filter handler.
4. The filter reads status, acknowledges, handles or defers, returns.
5. If deferred, the task runs later.
6. Teardown proceeds in the detach path.

What differs is the mechanism at step 2 and the allocation model at setup time. The device is not asserting a wire; it is writing to memory. The kernel does not need to have a pre-arranged destination wire; it has a pool of

message addresses and message values. Each vector corresponds to one (address, value) pair. The device stores these pairs in its MSI capability structure and issues a write using them when it needs an interrupt.

20.2.3 MSI: The Simpler of the Two

MSI (Message Signalled Interrupts) is the older and simpler of the two mechanisms. Introduced in PCI 2.2 in 1999, MSI lets a device request between 1 and 32 interrupt vectors, allocated as a contiguous power-of-two block (1, 2, 4, 8, 16, or 32). The device has a single MSI capability structure in its configuration space, which contains:

- A message address register (the write's destination address, typically the APIC's MSI region).
- A message data register (the value written, which encodes the vector number).
- A message control register (enable bit, function mask bit, number of vectors requested, and so on).

When the device wants to signal vector *N* (where *N* is 0 through count-1), it writes the message data register's base value OR'd with *N* to the message address. The interrupt controller demultiplexes the written value to dispatch the correct vector.

Key properties of MSI:

- **Single capability block.** The device has one MSI capability, not one per vector.
- **Contiguous vectors.** The block is power-of-two and allocated as a unit.
- **Limited count.** Maximum 32 vectors per function.
- **No per-vector masking.** The whole block is masked or unmasked as a group (via the function-mask bit if supported).
- **No per-vector address.** All vectors share a single message address register; the vector number goes in the low bits of the data written.

MSI is a significant improvement over legacy INTx, but it has limits: no per-vector masking and a 32-vector cap. Most drivers that want multiple vectors end up preferring MSI-X.

20.2.4 MSI-X: The Fuller Mechanism

MSI-X, introduced in PCI 3.0 in 2004 and extended in PCIe, removes MSI's limits. The device has an MSI-X capability structure plus an MSI-X **table** (an array of per-vector entries) and a **pending bit array** (PBA). The capability structure points into one or more of the device's BARs, where the table and PBA live.

Each MSI-X table entry contains:

- A message address register (per-vector).
- A message data register (per-vector).
- A vector control register (per-vector mask bit).

When the device wants to signal vector *N*, it looks up entry *N* in the table, reads that entry's address and data, and performs the write. The interrupt controller dispatches based on what was written.

Key properties of MSI-X:

- **Per-vector address and data.** Each vector can be routed to a different CPU by programming a different address.

- **Per-vector mask.** Individual vectors can be disabled without disabling the whole block.
- **Up to 2048 vectors per function.** An NVMe controller with many queues is happy here; a NIC with 64 receive queues plus 64 transmit queues plus some admin vectors fits.
- **Table in a BAR.** The table's location is discoverable through the MSI-X capability registers; `pci_msix_table_bar(9)` and `pci_msix_pba_bar(9)` return which BAR holds each.
- **More complex setup.** The driver has to allocate the table, program each entry, and then enable.

In practice, modern PCIe devices prefer MSI-X for any multi-vector use case, and reserve MSI for backwards compatibility or single-vector simple devices. The kernel handles most of the table programming internally; the driver's job is to query the count, allocate, and register per-vector handlers.

20.2.5 How FreeBSD Abstracts the Difference

The kernel hides most of the MSI-vs-MSI-X difference behind a small set of accessor functions. From `/usr/src/sys/dev/pci/pcivar.h`:

- `pci_msi_count(dev)` returns the MSI vector count the device advertises (0 if no MSI capability).
- `pci_msix_count(dev)` returns the MSI-X vector count (0 if no MSI-X capability).
- `pci_alloc_msi(dev, &count)` and `pci_alloc_msix(dev, &count)` allocate vectors. The count is an input-output: input is the desired count, output is the actual count allocated.
- `pci_release_msi(dev)` releases both MSI and MSI-X vectors (it handles either case internally).

The driver does not interact with the MSI-X table directly; the kernel does that on the driver's behalf. What the driver does see is that after a successful allocation, the device appears to have additional IRQ resources available through `bus_alloc_resource_any(9)` with `SYS_RES_IRQ`, using `rid = 1, 2, 3, ...` for the allocated vectors. The driver then registers a filter handler for each resource the same way Chapter 19 registered one for the legacy line.

The symmetry is deliberate. The same `bus_setup_intr(9)` call that handled the legacy IRQ at `rid = 0` handles each MSI or MSI-X vector at `rid = 1, 2, 3, ...`. Every `INTR_MPSAFE` rule, every `FILTER_*` return-value convention, every shared-IRQ discipline (for MSI, where vectors can technically share an `intr_event` in corner cases), and every teardown ordering from Chapter 19 carries through.

20.2.6 The Fallback Ladder

A robust driver tries the mechanisms in order of preference and falls back to the next one when allocation fails. The canonical ladder:

1. **MSI-X first.** If `pci_msix_count(dev)` is non-zero, try `pci_alloc_msix(dev, &count)`. If it succeeds, use MSI-X. On a modern PCIe device this is the preferred path.
2. **MSI second.** If MSI-X is unavailable or the allocation failed, check `pci_msi_count(dev)`. If it is non-zero, try `pci_alloc_msi(dev, &count)`. If it succeeds, use MSI.
3. **Legacy INTx last.** If both MSI-X and MSI are unavailable, fall back to the Chapter 19 legacy path with `rid = 0`.

Real drivers implement this ladder so they work on every system they might land on, from a brand-new NVMe drive that only supports MSI-X to a legacy chipset that only supports INTx. The Chapter 20 driver does the

same; Section 2 writes the MSI path, Section 5 writes the MSI-X path, and Section 8 ties them together into a single fallback ladder.

20.2.7 Real-World Examples

A short tour of devices that use MSI and MSI-X.

Modern NICs. A typical 10 or 25 Gbps NIC exposes 16 to 64 MSI-X vectors: one per receive queue, one per transmit queue, and a handful for admin, error, and link-state events. Intel's `igc(4)`, `em(4)`, `ix(4)`, and `ixl(4)` all follow this pattern; Broadcom's `bnxt(4)`, Mellanox's `mlx4(4)` and `mlx5(4)`, and Chelsio's `cxgbe(4)` do the same. The `iflib(9)` framework wraps the MSI-X allocation for many drivers.

NVMe storage controllers. An NVMe controller has one admin queue and up to 65535 I/O queues. In practice, drivers allocate one MSI-X vector for the admin queue and one per I/O queue up to NCPU. FreeBSD's `nvme(4)` driver does exactly this; the code is readable and worth studying.

Modern USB host controllers. An xHCI (USB 3) host controller typically advertises one MSI-X vector for the command-completion event ring and several more for per-slot event rings on high-performance variants. The `xhci(4)` driver's setup path shows the admin-plus-events pattern.

GPUs. A modern discrete GPU has many MSI-X vectors: one for the command buffer, one or more for display, one per engine, one for power management, and others. The out-of-tree `drm-kmod` drivers exercise MSI-X extensively.

Virtio devices in VMs. When a FreeBSD guest runs under bhyve, KVM, or VMware, the modern virtio-PCI transport uses MSI-X: one vector for configuration-change events, and one per virtqueue. The `virtio_pci_modern(4)` driver implements this.

Each of these drivers follows the same pattern Chapter 20 teaches: query, allocate, register per-vector handlers, bind to CPUs, describe. The specifics differ (how many vectors, how they are assigned to events, how they are bound to CPUs), but the structure is constant.

20.2.8 Why MSI-X and Not MSI

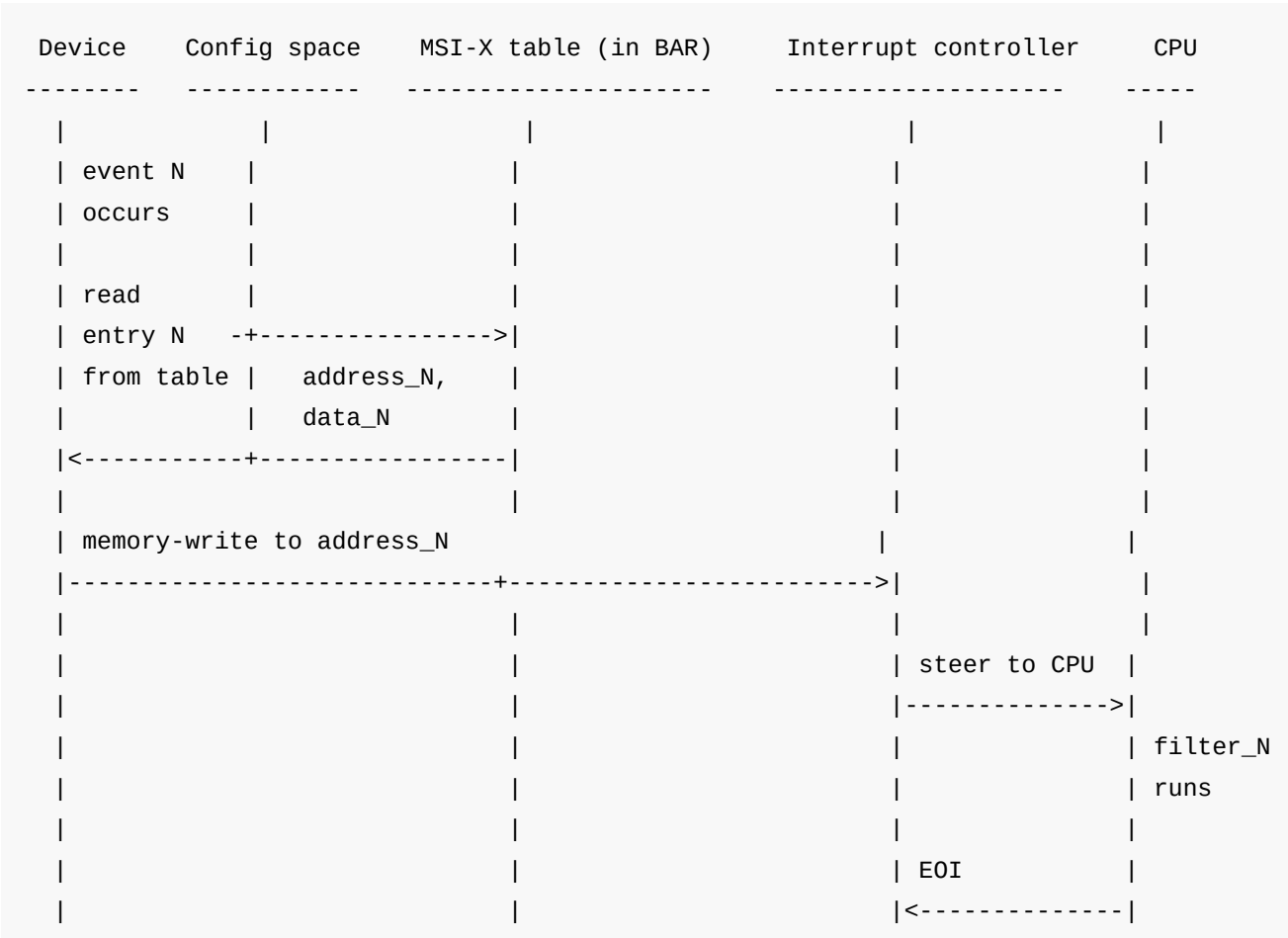
A reader might ask: given that MSI-X is strictly more capable than MSI, why does MSI even exist any more? Two reasons.

The first is backward compatibility. Devices and motherboards predating PCI 3.0 may support MSI but not MSI-X. A driver that wants to work on old hardware needs an MSI fallback. Most of the ecosystem has moved forward, but the long tail of older devices still exists.

The second is simplicity. MSI with one or two vectors is simpler to set up than MSI-X (no table to program, no BAR to consult). For devices whose interrupt needs fit within MSI's 32-vector cap and do not need per-vector masking, MSI is the lighter-weight choice. Many simple PCIe devices expose only MSI for this reason.

The practical answer for Chapter 20's driver: always try MSI-X first, fall back to MSI if MSI-X is unavailable, fall back to legacy INTx if neither is available. Every real FreeBSD driver written in the last decade uses this ladder.

20.2.9 A Diagram of the MSI-X Flow



The diagram elides the MSI-X-table reads (which the device performs internally before issuing the write) and the interrupt-controller’s demultiplexing logic, but it captures the mechanism’s essence: the device’s event triggers a memory write, the memory write becomes an interrupt, the interrupt dispatches to a filter. The filter does the same work Chapter 19’s filter did. The only difference is that on MSI-X, there is a different filter for each vector.

20.2.10 Exercise: Finding MSI-Capable Devices on Your System

Before moving to Section 2, a short exercise to make the capability picture concrete.

On your lab host, run:

```
sudo pciconf -lvc
```

The -c flag tells pciconf(8) to print each device’s capability list. You will see entries like:

```
vgapci0@pci0:0:2:0: ...
...
```

```
cap 05[d0] = MSI supports 1 message, 64 bit
cap 10[a0] = PCI-Express 2 endpoint max data 128(128)
em0@pci0:0:25:0: ...
...
cap 01[c8] = powerspec 2 supports D0 D3 current D0
cap 05[d0] = MSI supports 1 message, 64 bit
cap 11[e0] = MSI-X supports 4 messages
```

Each `cap 05` is an MSI capability. Each `cap 11` is an MSI-X capability. The description after the equals sign tells you how many messages (vectors) the device supports in that mode.

Pick three devices from your output. For each, note:

- The MSI count (if any).
- The MSI-X count (if any).
- Which one the driver is currently using. (You can deduce this from `vmstat -i` entries for the device: if you see multiple `name:queueN` lines, the driver is using MSI-X.)

A host without many PCIe devices may show only MSI capabilities; laptops often have limited MSI-X use. A modern server with multiple NICs and an NVMe drive shows many MSI-X capabilities with high vector counts (64 or more for some NICs).

Keep this output open while you read Section 2. The vocabulary of “`cap 11[XX] = MSI-X supports N messages`” is what the kernel’s `pci_msix_count(9)` returns to the driver, and what the allocation ladder queries at attach time.

20.2.11 Wrapping Up Section 1

MSI and MSI-X are the modern, message-signalled successors to legacy INTx. MSI offers up to 32 vectors allocated as a contiguous block with a single destination address; MSI-X offers up to 2048 vectors with per-vector addresses, per-vector data, and per-vector masking. Both are signalled over PCIe as memory writes that the interrupt controller decodes into vector dispatches.

The kernel abstracts the difference behind `pci_msi_count(9)`, `pci_msix_count(9)`, `pci_alloc_msi(9)`, `pci_alloc_msix(9)`, and `pci_release_msi(9)`. Each allocated vector becomes an IRQ resource at `rid = 1, 2, 3, ...` that the driver registers a filter handler for via `bus_setup_intr(9)`, exactly as Chapter 19 did for the legacy IRQ at `rid = 0`.

A robust driver implements a three-tier fallback ladder: MSI-X preferred, MSI as fallback, legacy INTx as last resort. Section 2 writes the MSI part of that ladder. Section 5 writes the MSI-X part. Section 8 assembles the full ladder.

20.3 Section 2: Enabling MSI in Your Driver

Section 1 established the hardware model. Section 2 puts the driver to work. The task is narrow: extend Chapter 19’s attach path so that before falling back to the legacy IRQ at `rid = 0`, the driver tries to allocate an MSI

vector. If the allocation succeeds, the driver uses the MSI vector instead of the legacy line. If the allocation fails (either because the device does not support MSI or the kernel cannot allocate), the driver falls back to the legacy path exactly as Chapter 19's code did.

The point of Section 2 is to introduce the MSI API in isolation, before the multi-vector complications of MSI-X make the picture busier. A single-vector MSI path is essentially the same as a single-vector legacy INTx path; only the allocation call and the rid change. That minimal change makes a good first stage.

20.3.1 What Stage 1 Produces

Stage 1 extends the Chapter 19 Stage 4 driver with a two-tier fallback: MSI first, legacy INTx as fallback. The filter handler is the same Chapter 19 filter. The taskqueue is the same. The sysctls are the same. What changes is the allocation path: `myfirst_intr_setup` first checks `pci_msi_count(9)` and, if non-zero, calls `pci_alloc_msi(9)` for one vector. If that succeeds, the IRQ resource is at `rid = 1`; if it fails, the driver falls through to `rid = 0` for legacy INTx.

The driver also logs the interrupt mode in a single `dmesg` line so the operator knows at a glance which tier the driver ended up using. This is a small but important observability feature that every real FreeBSD driver implements; Chapter 20 follows the convention.

20.3.2 The MSI Count Query

The first step is to ask the device how many MSI vectors it advertises:

```
int msi_count = pci_msi_count(sc->dev);
```

The return value is 0 if the device has no MSI capability; otherwise it is the number of vectors the device advertises in its MSI capability control register. Typical values are 1, 2, 4, 8, 16, or 32 (MSI requires a power-of-two count up to 32).

A return of 0 does not mean the device has no interrupts; it means the device does not expose MSI. The driver should fall through to the next tier.

20.3.3 The MSI Allocation Call

The second step is to ask the kernel to allocate the vectors:

```
int count = 1;
int error = pci_alloc_msi(sc->dev, &count);
```

The count is an input-output parameter. On input, it is the number of vectors the driver wants. On output, it is the number the kernel actually allocated. The kernel is allowed to allocate fewer than requested; a driver that needs at least a specific count must check the returned value.

For Chapter 20 Stage 1, the driver requests one vector. If the kernel returns 1, the driver proceeds. If the kernel returns 0 (rare but possible on a contended system) or returns an error, the driver releases any allocation and falls back to legacy INTx.

A subtle point: even when `pci_alloc_msi` returns non-zero, the driver **must** call `pci_release_msi(dev)` to undo the allocation at teardown time. Unlike `bus_alloc_resource_any` / `bus_release_resource`, the MSI family uses a single `pci_release_msi` call that undoes all vectors allocated via either `pci_alloc_msi` or `pci_alloc_msix` on the device.

20.3.4 The Per-Vector Resource Allocation

With MSI vectors allocated at the device level, the driver must now allocate a `SYS_RES_IRQ` resource for each vector. For a single MSI vector, the rid is 1:

```
int rid = 1; /* MSI vectors start at rid 1 */
struct resource *irq_res;

irq_res = bus_alloc_resource_any(sc->dev, SYS_RES_IRQ, &rid, RF_ACTIVE);
if (irq_res == NULL) {
    /* Release the MSI allocation and fall back. */
    pci_release_msi(sc->dev);
    goto fallback;
}
```

Note two differences from Chapter 19's legacy allocation:

First, **rid is 1, not 0**. MSI vectors are numbered starting from 1, leaving rid 0 for legacy INTx. If the driver ever used both (which it should not), the rids would not overlap.

Second, **RF_SHAREABLE is not set**. MSI vectors are per-function; they are not shared with other drivers. The `RF_SHAREABLE` flag is only relevant for legacy INTx. Setting it on an MSI resource allocation does no harm, but it is not meaningful.

20.3.5 The Filter Handler on an MSI Vector

The filter handler function is identical to Chapter 19's:

```
int myfirst_intr_filter(void *arg);
```

The kernel calls the filter when the vector fires, exactly as it called the Chapter 19 filter when the legacy line asserted. The filter reads `INTR_STATUS`, acknowledges, enqueues the task for `DATA_AV`, and returns `FILTER_HANDLED` (or `FILTER_STRAY` on no-bit). Nothing about the filter's body needs to change.

`bus_setup_intr(9)` is called identically:

```
error = bus_setup_intr(sc->dev, irq_res,
    INTR_TYPE_MISC | INTR_MPSAFE,
    myfirst_intr_filter, NULL, sc,
    &sc->intr_cookie);
```


The function signature, the flags, the argument (sc), and the out-cookie are all the Chapter 19 pattern.

A small improvement: `bus_describe_intr(9)` can now label the vector with a mode-specific name:

```
bus_describe_intr(sc->dev, irq_res, sc->intr_cookie, "msi");
```

After this, `vmstat -i` shows the handler as `irq<N>: myfirst0:msi` (for some N the kernel chose). The operator instantly sees the driver is using MSI.

20.3.6 Building the Fallback

Putting it all together, the Stage 1 `myfirst_intr_setup` becomes a fallback ladder with two tiers: try MSI first, fall back to legacy INTx. The code:

```
int
myfirst_intr_setup(struct myfirst_softc *sc)
{
    int error, msi_count, count;

    TASK_INIT(&sc->intr_data_task, 0, myfirst_intr_data_task_fn, sc);
    sc->intr_tq = taskqueue_create("myfirst_intr", M_WAITOK,
        taskqueue_thread_enqueue, &sc->intr_tq);
    taskqueue_start_threads(&sc->intr_tq, 1, PI_NET,
        "myfirst intr taskq");

    /*
     * Tier 1: attempt MSI.
     */
    msi_count = pci_msi_count(sc->dev);
    if (msi_count > 0) {
        count = 1;
        if (pci_alloc_msi(sc->dev, &count) == 0 && count == 1) {
            sc->irq_rid = 1;
            sc->irq_res = bus_alloc_resource_any(sc->dev,
                SYS_RES_IRQ, &sc->irq_rid, RF_ACTIVE);
            if (sc->irq_res != NULL) {
                error = bus_setup_intr(sc->dev, sc->irq_res,
                    INTR_TYPE_MISC | INTR_MPSAFE,
                    myfirst_intr_filter, NULL, sc,
                    &sc->intr_cookie);
                if (error == 0) {
                    bus_describe_intr(sc->dev,
                        sc->irq_res, sc->intr_cookie,
                        "msi");
                    sc->intr_mode = MYFIRST_INTR_MSI;
                }
            }
        }
    }
}
```

```

        device_printf(sc->dev,
            "interrupt mode: MSI, 1 vector\n");
        goto enabled;
    }
    bus_release_resource(sc->dev,
        SYS_RES_IRQ, sc->irq_rid, sc->irq_res);
    sc->irq_res = NULL;
}
pci_release_msi(sc->dev);
}
}

/*
 * Tier 2: fall back to legacy INTx.
 */
sc->irq_rid = 0;
sc->irq_res = bus_alloc_resource_any(sc->dev, SYS_RES_IRQ,
    &sc->irq_rid, RF_SHAREABLE | RF_ACTIVE);
if (sc->irq_res == NULL) {
    device_printf(sc->dev, "cannot allocate legacy IRQ\n");
    taskqueue_free(sc->intr_tq);
    sc->intr_tq = NULL;
    return (ENXIO);
}
error = bus_setup_intr(sc->dev, sc->irq_res,
    INTR_TYPE_MISC | INTR_MPSAFE,
    myfirst_intr_filter, NULL, sc,
    &sc->intr_cookie);
if (error != 0) {
    bus_release_resource(sc->dev, SYS_RES_IRQ,
        sc->irq_rid, sc->irq_res);
    sc->irq_res = NULL;
    taskqueue_free(sc->intr_tq);
    sc->intr_tq = NULL;
    return (error);
}
bus_describe_intr(sc->dev, sc->irq_res, sc->intr_cookie, "legacy");
sc->intr_mode = MYFIRST_INTR_LEGACY;
device_printf(sc->dev,
    "interrupt mode: legacy INTx (rid=0)\n");

enabled:
    /* Enable interrupts at the device. */
    MYFIRST_LOCK(sc);

```

```

if (sc->hw != NULL)
    CSR_WRITE_4(sc, MYFIRST_REG_INTR_MASK,
        MYFIRST_INTR_DATA_AV | MYFIRST_INTR_ERROR |
        MYFIRST_INTR_COMPLETE);
MYFIRST_UNLOCK(sc);

return (0);
}

```

The code has three distinct blocks:

1. The MSI attempt block (lines inside the `if (msi_count > 0)` guard).
2. The legacy fallback block.
3. The enabling block at `enabled:` that runs regardless of tier.

The MSI attempt does the full sequence: count query, allocate, allocate the IRQ resource at rid 1, register the handler. If any step fails, the code releases whatever succeeded (the resource if allocated, the MSI if successfully allocated) and falls through.

The legacy fallback is essentially the Chapter 19 setup, unchanged.

The `enabled:` block writes `INTR_MASK` at the device. Whether we got MSI or legacy, the device-side mask is the same.

The fallback structure is what real drivers do. A reader looking at `virtio_pci.c`'s setup code will see the same pattern at larger scale: several attempts with successive fallbacks.

20.3.7 The `intr_mode` Field and the `dmesg` Summary

The `softc` gains a new field:

```

enum myfirst_intr_mode {
    MYFIRST_INTR_LEGACY = 0,
    MYFIRST_INTR_MSI = 1,
    MYFIRST_INTR_MSIX = 2,
};

struct myfirst_softc {
    /* ... existing fields ... */
    enum myfirst_intr_mode intr_mode;
};

```

The field records which tier the driver ended up using. The attach-time `device_printf` prints it:

```
myfirst0: interrupt mode: MSI, 1 vector
```

or:

```
myfirst0: interrupt mode: legacy INTx (rid=0)
```

Operators reading `dmesg` see this line and know which path is active. The reader debugging their driver sees it too; if the driver is falling back to legacy when the reader expected MSI, the line flags the problem immediately.

The `intr_mode` field is also exposed through a read-only sysctl so that user-space tools can read it:

```
SYSCTL_ADD_INT(&sc->sysctl_ctx,
    SYSCTL_CHILDREN(sc->sysctl_tree), OID_AUTO, "intr_mode",
    CTLFLAG_RD, &sc->intr_mode, 0,
    "Interrupt mode: 0=legacy, 1=MSI, 2=MSI-X");
```

A script that wants to know whether any `myfirst` instance is using MSI-X can sum the `intr_mode` values across all units.

20.3.8 What the Teardown Needs to Change

The teardown path in Chapter 19 called `bus_teardown_intr`, drained and freed the taskqueue, and released the IRQ resource. For Stage 1, one additional call is needed: if the driver used MSI, it must call `pci_release_msi` after releasing the IRQ resource:

```
void
myfirst_intr_teardown(struct myfirst_softc *sc)
{
    /* Disable at the device. */
    MYFIRST_LOCK(sc);
    if (sc->hw != NULL && sc->bar_res != NULL)
        CSR_WRITE_4(sc, MYFIRST_REG_INTR_MASK, 0);
    MYFIRST_UNLOCK(sc);

    /* Tear down the handler. */
    if (sc->intr_cookie != NULL) {
        bus_teardown_intr(sc->dev, sc->irq_res, sc->intr_cookie);
        sc->intr_cookie = NULL;
    }

    /* Drain and destroy the taskqueue. */
    if (sc->intr_tq != NULL) {
```

```

    taskqueue_drain(sc->intr_tq, &sc->intr_data_task);
    taskqueue_free(sc->intr_tq);
    sc->intr_tq = NULL;
}

/* Release the IRQ resource. */
if (sc->irq_res != NULL) {
    bus_release_resource(sc->dev, SYS_RES_IRQ, sc->irq_rid,
        sc->irq_res);
    sc->irq_res = NULL;
}

/* Release MSI if used. */
if (sc->intr_mode == MYFIRST_INTR_MSI ||
    sc->intr_mode == MYFIRST_INTR_MSIX)
    pci_release_msi(sc->dev);

sc->intr_mode = MYFIRST_INTR_LEGACY;
}

```

The `pci_release_msi` call is conditional: only call it if the driver actually allocated MSI or MSI-X. Calling it when the driver used only legacy INTx is a no-op in modern FreeBSD, but the condition is clearer.

Notice the ordering: IRQ resource release first, then `pci_release_msi`. This is the opposite of the allocation order (where `pci_alloc_msi` preceded `bus_alloc_resource_any`). The rule is the general teardown rule from Chapter 18 and 19: undo in reverse order of setup.

20.3.9 Verifying Stage 1

On a guest where the device supports MSI (QEMU's `virtio-rng-pci` does; bhyve's `virtio-rnd` does not), the Stage 1 driver should attach with MSI:

```

myfirst0: <Red Hat Virtio entropy source (myfirst demo target)> ... on pci0
myfirst0: BAR0 allocated: 0x20 bytes at 0xfebf1000
myfirst0: hardware layer attached to BAR: 32 bytes
myfirst0: interrupt mode: MSI, 1 vector

```

On a guest where the device supports only legacy (bhyve's `virtio-rnd` typically):

```

myfirst0: BAR0 allocated: 0x20 bytes at 0xc1000000
myfirst0: hardware layer attached to BAR: 32 bytes
myfirst0: interrupt mode: legacy INTx (rid=0)

```

Both cases are correct. The driver runs in either mode; the behaviour (filter, task, counters, simulated-interrupt `sysctl`) is identical.

`sysctl dev.myfirst.0.intr_mode` returns 0 (legacy), 1 (MSI), or 2 (MSI-X, once Section 5 adds it). The regression script uses this to verify the expected mode.

20.3.10 What Stage 1 Does Not Do

Stage 1 adds MSI with one vector but does not yet use MSI's multi-vector potential. A single MSI vector is functionally almost identical to a single legacy IRQ (it only gains the scalability benefits on a system with many devices, which a single-device lab rarely shows). The value of Stage 1 is that it introduces the fallback ladder idiom and establishes the `intr_mode` observability; Stage 2 and beyond use those foundations to add multi-vector handling.

20.3.11 Common Mistakes at This Stage

A short list.

Using `rid = 0` for MSI. The MSI vector's `rid` is 1, not 0. Requesting `rid = 0` on a device that has allocated MSI vectors returns the legacy INTx resource, which is not the MSI vector. The driver ends up with a handler on the wrong line. Fix: `rid = 1` for the first MSI or MSI-X vector.

Forgetting `pci_release_msi` at teardown. The kernel's MSI allocation state survives `bus_release_resource` on the IRQ resource. Without `pci_release_msi`, the next attach attempt will fail because the kernel still thinks the driver owns the MSI vectors. Fix: always call `pci_release_msi` in teardown when MSI or MSI-X was used.

Forgetting the INTx fallback. A driver that only tries MSI and returns an error on failure works on systems that support MSI but fails on older systems. Fix: always provide a legacy INTx fallback.

Forgetting to restore `sc->intr_mode` on teardown. The `intr_mode` field records the tier. Without resetting, a future reattach could read a stale value. Not a serious bug (attach always sets it), but cleanliness matters. Fix: reset to LEGACY (or a neutral value) in teardown.

Mismatching the count. `pci_alloc_msi` can allocate fewer vectors than requested; if the driver assumes `count == 1` when it is 0, the code dereferences an un-allocated resource. Fix: always check the returned count.

Calling `pci_alloc_msi` twice without release. Only one MSI (or MSI-X) allocation can be active per device at a time. Attempting a second allocation without releasing the first returns an error. Fix: if the driver wants to change its allocation (from MSI to MSI-X, say), call `pci_release_msi` first.

20.3.12 Checkpoint: Stage 1 Working

Before Section 3, confirm Stage 1 is in place:

- `kldstat -v | grep myfirst` shows version 1.3-`msi-stage1`.
- `dmesg | grep myfirst` shows the attach banner with an `interrupt mode:` line indicating either MSI or legacy.

- `sysctl dev.myfirst.0.intr_mode` returns 0 or 1.
- `vmstat -i | grep myfirst` shows the handler with either `myfirst0:msi` or `myfirst0:legacy` as the descriptor.
- `sudo sysctl dev.myfirst.0.intr_simulate=1` still drives the Chapter 19 pipeline.
- `kldunload myfirst` runs cleanly; no leaks.

If the MSI path fails on your guest, try QEMU instead of bhyve. If the MSI path works on one and not the other, verify the device's MSI capability is exposed through `pciconf -lvc`.

20.3.13 Wrapping Up Section 2

Enabling MSI in a driver is three new calls (`pci_msi_count`, `pci_alloc_msi`, `pci_release_msi`), one change to the IRQ resource allocation (`rid = 1` instead of 0), and one new softc field (`intr_mode`). The fallback ladder adds a second tier: try MSI, fall back to legacy. Every `bus_setup_intr`, every filter, every taskqueue task, and every teardown step from Chapter 19 carries through unchanged.

Stage 1 handles a single MSI vector. Section 3 moves to multi-vector: several different filter functions, several per-vector softc states, and the beginning of the per-queue handler pattern that modern drivers use extensively.

20.4 Section 3: Managing Multiple Interrupt Vectors

Stage 1 added MSI with one vector. Section 3 extends the driver to handle multiple vectors, each with its own role. The motivating example is the device that has more than one thing to say: a NIC with a receive queue and a transmit queue, an NVMe controller with admin and I/O queues, a UART with receive-ready and transmit-empty events.

The Chapter 20 driver does not have a real multi-queue device; the `virtio-rnd` target has at most a single event class per interrupt. For teaching purposes, we simulate multi-vector behaviour the same way Chapter 19 simulated interrupts: the `sysctl` interface lets the reader fire simulated interrupts against specific vectors, and the driver's filter and task machinery demonstrates how real drivers handle multi-vector cases.

By the end of Section 3 the driver is at version `1.3-msi-stage2` and has three MSI-X vectors: an admin vector, an "rx" vector, and a "tx" vector. Each vector has its own filter function, its own deferred task, and its own counters. The filter reads `INTR_STATUS` and acknowledges only the bits relevant to its vector; the task does the vector-specific work.

An important note about the three-vector count. MSI is constrained to a power-of-two vector count (1, 2, 4, 8, 16, or 32), so a request for exactly 3 vectors is rejected by `pci_alloc_msi(9)` with `EINVAL` (see `/usr/src/sys/dev/pci/pci.c`'s `pci_alloc_msi_method`). MSI-X has no such restriction and readily allocates 3 vectors. The MSI tier of the fallback ladder therefore requests a single MSI vector and falls back to the Chapter 19 single-handler pattern; only the MSI-X tier gives the driver its three per-vector filters. Section 5 makes this explicit and Section 8's refactor keeps the MSI tier simple.

20.4.1 The Per-Vector Design

The design has three vectors:

- **Admin vector (vector 0, rid 1).** Handles ERROR and configuration-change events. Low rate; runs rarely.
- **RX vector (vector 1, rid 2).** Handles DATA_AV events (receive-ready). Runs at data-path rate.
- **TX vector (vector 2, rid 3).** Handles COMPLETE events (transmit-complete). Runs at data-path rate.

Each vector has:

- A separate struct resource * (the IRQ resource for that vector).
- A separate void *intr_cookie (the kernel's opaque handle for the handler).
- A separate filter function (myfirst_admin_filter, myfirst_rx_filter, myfirst_tx_filter).
- A separate set of counters (so per-CPU concurrent filter runs do not fight over a single shared counter).
- A separate bus_describe_intr name (admin, rx, tx).
- A separate deferred task (for RX; admin and TX handle their work inline).

The per-vector state lives in an array of per-vector structures inside the softc:

```
#define MYFIRST_MAX_VECTORS 3

enum myfirst_vector_id {
    MYFIRST_VECTOR_ADMIN = 0,
    MYFIRST_VECTOR_RX,
    MYFIRST_VECTOR_TX,
};

struct myfirst_vector {
    struct resource      *irq_res;
    int                  irq_rid;
    void                 *intr_cookie;
    enum myfirst_vector_id id;
    struct myfirst_softc *sc;
    uint64_t             fire_count;
    uint64_t             stray_count;
    const char           *name;
    driver_filter_t       *filter;
    struct task           task;
    bool                  has_task;
};

struct myfirst_softc {
    /* ... existing fields ... */
    struct myfirst_vector vectors[MYFIRST_MAX_VECTORS];
    int                  num_vectors; /* actually allocated */
};
```

A few design notes worth unpacking.

Per-vector struct myfirst_softc *sc back-pointer. The argument each filter receives via bus_setup_intr is the per-vector structure (struct myfirst_vector *), not the global softc. The per-vector structure contains

a back-pointer to the softc so the filter can reach shared state when needed. This is the pattern `nvme(4)` uses for per-queue vectors and the pattern every multi-queue driver follows.

Per-vector counters. Each vector has its own `fire_count` and `stray_count`. Two filters running on two CPUs can increment their own counters without atomic contention; atomics are still used, but each atomic hits a different cache line.

Per-vector filter pointer. The `filter` field stores a pointer to the vector's filter function. This is not strictly required (we could have a switch in a single generic filter), but it makes the per-vector specialisation explicit: each vector's filter is statically known.

Per-vector task. Not every vector needs a task. Admin and TX do their work inline (increment a counter, update a flag, maybe wake a waiter). RX defers to a task because it wants to broadcast a condition variable, which requires thread context. The `has_task` flag makes the per-vector difference explicit.

20.4.2 The Filter Functions

Three distinct filter functions, one per vector:

```
int
myfirst_admin_filter(void *arg)
{
    struct myfirst_vector *vec = arg;
    struct myfirst_softc *sc = vec->sc;
    uint32_t status;

    status = ICSR_READ_4(sc, MYFIRST_REG_INTR_STATUS);
    if ((status & (MYFIRST_INTR_ERROR)) == 0) {
        atomic_add_64(&vec->stray_count, 1);
        return (FILTER_STRAY);
    }

    atomic_add_64(&vec->fire_count, 1);
    ICSR_WRITE_4(sc, MYFIRST_REG_INTR_STATUS, MYFIRST_INTR_ERROR);
    atomic_add_64(&sc->intr_error_count, 1);
    return (FILTER_HANDLED);
}

int
myfirst_rx_filter(void *arg)
{
    struct myfirst_vector *vec = arg;
    struct myfirst_softc *sc = vec->sc;
    uint32_t status;

    status = ICSR_READ_4(sc, MYFIRST_REG_INTR_STATUS);
```

```

    if ((status & MYFIRST_INTR_DATA_AV) == 0) {
        atomic_add_64(&vec->stray_count, 1);
        return (FILTER_STRAY);
    }

    atomic_add_64(&vec->fire_count, 1);
    ICSR_WRITE_4(sc, MYFIRST_REG_INTR_STATUS, MYFIRST_INTR_DATA_AV);
    atomic_add_64(&sc->intr_data_av_count, 1);
    if (sc->intr_tq != NULL)
        taskqueue_enqueue(sc->intr_tq, &vec->task);
    return (FILTER_HANDLED);
}

int
myfirst_tx_filter(void *arg)
{
    struct myfirst_vector *vec = arg;
    struct myfirst_softc *sc = vec->sc;
    uint32_t status;

    status = ICSR_READ_4(sc, MYFIRST_REG_INTR_STATUS);
    if ((status & MYFIRST_INTR_COMPLETE) == 0) {
        atomic_add_64(&vec->stray_count, 1);
        return (FILTER_STRAY);
    }

    atomic_add_64(&vec->fire_count, 1);
    ICSR_WRITE_4(sc, MYFIRST_REG_INTR_STATUS, MYFIRST_INTR_COMPLETE);
    atomic_add_64(&sc->intr_complete_count, 1);
    return (FILTER_HANDLED);
}

```

Each filter has the same shape: read status, check the bit this vector cares about, acknowledge, update counters, optionally enqueue a task, return. The differences are which bit each filter cares about and which counters each increments.

Several details worth noticing.

The stray check is per-vector. Each filter checks for its own bit, not for any bit. If the filter is called for an event it does not handle (because the bit set is for a different vector), the filter returns `FILTER_STRAY`. This matters less for MSI-X (where each vector has its own dedicated message, so the device never fires the “wrong” vector), but it matters more for MSI with multiple vectors sharing a single capability.

Counter sharing. The per-vector counters (`vec->fire_count`, `vec->stray_count`) are specific to the vector. The global counters (`sc->intr_data_av_count`, etc.) are shared and still used for the chapter’s per-bit observability. Having both gives the reader a way to cross-check: the RX filter’s fire count should approximately

equal the global `data_av_count`.

The filter does not sleep. All of the Chapter 19 filter-context rules carry through: no sleep locks, no `malloc(M_WAITOK)`, no blocking. The filter uses only atomics and direct BAR accesses.

20.4.3 The Per-Vector Task

Only RX has a task; admin and TX handle their work in the filter. The RX task is essentially the Chapter 19 task:

```
static void
myfirst_rx_task_fn(void *arg, int npending)
{
    struct myfirst_vector *vec = arg;
    struct myfirst_softc *sc = vec->sc;

    MYFIRST_LOCK(sc);
    if (sc->hw != NULL && sc->pci_attached) {
        sc->intr_last_data = CSR_READ_4(sc, MYFIRST_REG_DATA_OUT);
        sc->intr_task_invocations++;
        cv_broadcast(&sc->data_cv);
    }
    MYFIRST_UNLOCK(sc);
}
```

The task runs in thread context on the shared `intr_tq` taskqueue (Chapter 19 created it at `PI_NET` priority). The same taskqueue serves all per-vector tasks; for a driver with truly independent per-queue work, each vector might have its own taskqueue, but Chapter 20 uses one.

20.4.4 Allocating Multiple Vectors

The setup code for Stage 2 is longer than Stage 1 because it handles multiple vectors:

```
int
myfirst_intr_setup(struct myfirst_softc *sc)
{
    int error, wanted, allocated, i;

    TASK_INIT(&sc->vectors[MYFIRST_VECTOR_RX].task, 0,
              myfirst_rx_task_fn, &sc->vectors[MYFIRST_VECTOR_RX]);
    sc->vectors[MYFIRST_VECTOR_RX].has_task = true;
    sc->vectors[MYFIRST_VECTOR_ADMIN].filter = myfirst_admin_filter;
    sc->vectors[MYFIRST_VECTOR_RX].filter = myfirst_rx_filter;
    sc->vectors[MYFIRST_VECTOR_TX].filter = myfirst_tx_filter;
```

```

sc->vectors[MYFIRST_VECTOR_ADMIN].name = "admin";
sc->vectors[MYFIRST_VECTOR_RX].name = "rx";
sc->vectors[MYFIRST_VECTOR_TX].name = "tx";
for (i = 0; i < MYFIRST_MAX_VECTORS; i++) {
    sc->vectors[i].id = i;
    sc->vectors[i].sc = sc;
}

sc->intr_tq = taskqueue_create("myfirst_intr", M_WAITOK,
    taskqueue_thread_enqueue, &sc->intr_tq);
taskqueue_start_threads(&sc->intr_tq, 1, PI_NET,
    "myfirst intr taskq");

/*
 * Try to allocate a single MSI vector. MSI requires a power-of-two
 * count (PCI specification and /usr/src/sys/dev/pci/pci.c's
 * pci_alloc_msi_method enforces this), so we cannot request the
 * MYFIRST_MAX_VECTORS = 3 we want; we ask for 1 and fall back to
 * the Chapter 19 single-handler pattern at rid=1, the same way
 * sys/dev/virtio/pci/virtio_pci.c's vtpci_alloc_msi() does.
 *
 * MSI-X, covered in Section 5, is the tier where we actually
 * obtain three distinct vectors; MSI-X is not constrained to
 * power-of-two counts.
 */
allocated = 1;
if (pci_msi_count(sc->dev) >= 1 &&
    pci_alloc_msi(sc->dev, &allocated) == 0 && allocated >= 1) {
    sc->vectors[MYFIRST_VECTOR_ADMIN].filter = myfirst_intr_filter;
    sc->vectors[MYFIRST_VECTOR_ADMIN].name = "msi";
    error = myfirst_intr_setup_vector(sc, MYFIRST_VECTOR_ADMIN, 1);
    if (error == 0) {
        sc->intr_mode = MYFIRST_INTR_MSI;
        sc->num_vectors = 1;
        device_printf(sc->dev,
            "interrupt mode: MSI, 1 vector "
            "(single-handler fallback)\n");
        goto enabled;
    }
    pci_release_msi(sc->dev);
}

/*
 * MSI allocation failed or was unavailable. Fall back to legacy

```

```

    * INTx with a single vector-0 handler that handles every event
    * class in one place.
    */

```

fallback_legacy:

```

sc->vectors[MYFIRST_VECTOR_ADMIN].irq_rid = 0;
sc->vectors[MYFIRST_VECTOR_ADMIN].irq_res = bus_alloc_resource_any(
    sc->dev, SYS_RES_IRQ,
    &sc->vectors[MYFIRST_VECTOR_ADMIN].irq_rid,
    RF_SHAREABLE | RF_ACTIVE);
if (sc->vectors[MYFIRST_VECTOR_ADMIN].irq_res == NULL) {
    device_printf(sc->dev, "cannot allocate legacy IRQ\n");
    taskqueue_free(sc->intr_tq);
    sc->intr_tq = NULL;
    return (ENXIO);
}
error = bus_setup_intr(sc->dev,
    sc->vectors[MYFIRST_VECTOR_ADMIN].irq_res,
    INTR_TYPE_MISC | INTR_MPSAFE,
    myfirst_intr_filter, NULL, sc,
    &sc->vectors[MYFIRST_VECTOR_ADMIN].intr_cookie);
if (error != 0) {
    bus_release_resource(sc->dev, SYS_RES_IRQ,
        sc->vectors[MYFIRST_VECTOR_ADMIN].irq_rid,
        sc->vectors[MYFIRST_VECTOR_ADMIN].irq_res);
    sc->vectors[MYFIRST_VECTOR_ADMIN].irq_res = NULL;
    taskqueue_free(sc->intr_tq);
    sc->intr_tq = NULL;
    return (error);
}
bus_describe_intr(sc->dev,
    sc->vectors[MYFIRST_VECTOR_ADMIN].irq_res,
    sc->vectors[MYFIRST_VECTOR_ADMIN].intr_cookie, "legacy");
sc->intr_mode = MYFIRST_INTR_LEGACY;
sc->num_vectors = 1;
device_printf(sc->dev,
    "interrupt mode: legacy INTx (1 handler for all events)\n");

```

enabled:

```

/* Enable interrupts at the device. */
MYFIRST_LOCK(sc);
if (sc->hw != NULL)
    CSR_WRITE_4(sc, MYFIRST_REG_INTR_MASK,
        MYFIRST_INTR_DATA_AV | MYFIRST_INTR_ERROR |

```

```

        MYFIRST_INTR_COMPLETE);
MYFIRST_UNLOCK(sc);

return (0);
}

```

The code has three phases: MSI attempt, MSI fallback cleanup, and legacy fallback. The MSI attempt loops through the vectors, calling a helper (`myfirst_intr_setup_vector`) to allocate and register each one. On failure at any vector, the code unwinds in reverse order and falls through to legacy.

The helper:

```

static int
myfirst_intr_setup_vector(struct myfirst_softc *sc, int idx, int rid)
{
    struct myfirst_vector *vec = &sc->vectors[idx];
    int error;

    vec->irq_rid = rid;
    vec->irq_res = bus_alloc_resource_any(sc->dev, SYS_RES_IRQ,
        &vec->irq_rid, RF_ACTIVE);
    if (vec->irq_res == NULL)
        return (ENXIO);

    error = bus_setup_intr(sc->dev, vec->irq_res,
        INTR_TYPE_MISC | INTR_MPSAFE,
        vec->filter, NULL, vec, &vec->intr_cookie);
    if (error != 0) {
        bus_release_resource(sc->dev, SYS_RES_IRQ, vec->irq_rid,
            vec->irq_res);
        vec->irq_res = NULL;
        return (error);
    }

    bus_describe_intr(sc->dev, vec->irq_res, vec->intr_cookie,
        "%s", vec->name);
    return (0);
}

```

The helper is small and symmetric: allocate the resource, set up the handler, describe it. The argument to `bus_setup_intr` is the per-vector structure (`vec`), not the `softc`. The filter receives `vec` as its void `*arg` and uses `vec->sc` when it needs the `softc`.

The per-vector teardown helper:

```
static void
myfirst_intr_teardown_vector(struct myfirst_softc *sc, int idx)
{
    struct myfirst_vector *vec = &sc->vectors[idx];

    if (vec->intr_cookie != NULL) {
        bus_teardown_intr(sc->dev, vec->irq_res, vec->intr_cookie);
        vec->intr_cookie = NULL;
    }
    if (vec->irq_res != NULL) {
        bus_release_resource(sc->dev, SYS_RES_IRQ, vec->irq_rid,
            vec->irq_res);
        vec->irq_res = NULL;
    }
}
```

Teardown is the inverse of setup: tear down the handler, release the resource.

20.4.5 The Full Teardown Path

The multi-vector teardown calls the per-vector helper for each active vector, then releases the MSI allocation once:

```
void
myfirst_intr_teardown(struct myfirst_softc *sc)
{
    int i;

    MYFIRST_LOCK(sc);
    if (sc->hw != NULL && sc->bar_res != NULL)
        CSR_WRITE_4(sc, MYFIRST_REG_INTR_MASK, 0);
    MYFIRST_UNLOCK(sc);

    /* Per-vector teardown. */
    for (i = 0; i < sc->num_vectors; i++)
        myfirst_intr_teardown_vector(sc, i);

    /* Drain tasks. */
    if (sc->intr_tq != NULL) {
        for (i = 0; i < sc->num_vectors; i++) {
            if (sc->vectors[i].has_task)
                taskqueue_drain(sc->intr_tq,
                    &sc->vectors[i].task);
        }
    }
}
```

```

    taskqueue_free(sc->intr_tq);
    sc->intr_tq = NULL;
}

/* Release MSI if used. */
if (sc->intr_mode == MYFIRST_INTR_MSI ||
    sc->intr_mode == MYFIRST_INTR_MSIX)
    pci_release_msi(sc->dev);

sc->num_vectors = 0;
sc->intr_mode = MYFIRST_INTR_LEGACY;
}

```

The ordering is the now-familiar one: mask at device, tear down handlers, drain tasks, release MSI. The per-vector loop does the per-vector work.

20.4.6 Simulating Interrupts Per Vector

Chapter 19's simulated-interrupt sysctl fires one handler at a time. Stage 2 extends the concept: a sysctl per vector, or a single sysctl with a vector-index field. The chapter's code goes with the simpler single-sysctl-per-vector form:

```

SYSCTL_ADD_PROC(ctx, kids, OID_AUTO, "intr_simulate_admin",
    CTLTYPE_UINT | CTLFLAG_WR | CTLFLAG_MPSAFE,
    &sc->vectors[MYFIRST_VECTOR_ADMIN], 0,
    myfirst_intr_simulate_vector_sysctl, "IU",
    "Simulate admin vector interrupt");
SYSCTL_ADD_PROC(ctx, kids, OID_AUTO, "intr_simulate_rx", ...);
SYSCTL_ADD_PROC(ctx, kids, OID_AUTO, "intr_simulate_tx", ...);

```

The handler:

```

static int
myfirst_intr_simulate_vector_sysctl(SYSCTL_HANDLER_ARGS)
{
    struct myfirst_vector *vec = arg1;
    struct myfirst_softc *sc = vec->sc;
    uint32_t mask = 0;
    int error;

    error = sysctl_handle_int(oidp, &mask, 0, req);
    if (error != 0 || req->newptr == NULL)
        return (error);
}

```



```

MYFIRST_LOCK(sc);
if (sc->hw == NULL || sc->bar_res == NULL) {
    MYFIRST_UNLOCK(sc);
    return (ENODEV);
}
bus_write_4(sc->bar_res, MYFIRST_REG_INTR_STATUS, mask);
MYFIRST_UNLOCK(sc);

/*
 * Invoke this vector's filter if it has one (MSI-X). On single-
 * handler tiers (MSI with 1 vector, or legacy INTx) only slot 0
 * has a registered filter, so we fall through to it. The Chapter 19
 * myfirst_intr_filter handles all three status bits in one pass.
 */
if (vec->filter != NULL)
    (void)vec->filter(vec);
else if (sc->vectors[MYFIRST_VECTOR_ADMIN].filter != NULL)
    (void)sc->vectors[MYFIRST_VECTOR_ADMIN].filter(
        &sc->vectors[MYFIRST_VECTOR_ADMIN]);
return (0);
}

```

From user space:

```

sudo sysctl dev.myfirst.0.intr_simulate_admin=2  # ERROR bit, admin vector
sudo sysctl dev.myfirst.0.intr_simulate_rx=1     # DATA_AV bit, rx vector
sudo sysctl dev.myfirst.0.intr_simulate_tx=4     # COMPLETE bit, tx vector

```

The `intr_count` counters per vector tick independently. A reader can verify per-vector behaviour by firing each `sysctl` and watching the corresponding `vec->fire_count` counter rise.

20.4.7 What Happens on Legacy Fallback

When the driver falls back to legacy INTx (because MSI was unavailable or failed), there is only one handler to cover all three event classes. The code assigns the Chapter 19 `myfirst_intr_filter` to the admin vector's slot and uses that single filter on `rid = 0`. The admin vector's filter becomes a multi-event handler that looks at all three status bits and dispatches accordingly.

This is a small but important detail: the Chapter 19 filter still exists and is reused on the legacy path, while the per-vector filters are only used when MSI or MSI-X is available. A reader inspecting the driver sees both, and the difference is explained in Section 3's comments.

20.4.8 The Stage 2 dmesg Banner

On a guest where the driver lands on the MSI-X tier (this is the only tier that delivers three vectors; the MSI tier falls back to a single-handler setup for the reasons explained earlier):

```
myfirst0: BAR0 allocated: 0x20 bytes at 0xfebf1000
myfirst0: hardware layer attached to BAR: 32 bytes
myfirst0: interrupt mode: MSI-X, 3 vectors
```

`vmstat -i | grep myfirst` shows three separate lines:

irq256: myfirst0:admin	12	1
irq257: myfirst0:rx	98	8
irq258: myfirst0:tx	45	4

(The exact IRQ numbers vary by platform; MSI-allocated IRQs on x86 start in the 256 range once the I/O APIC range is exhausted.)

On a guest where only MSI is available, the driver reports a single-handler fallback:

```
myfirst0: interrupt mode: MSI, 1 vector (single-handler fallback)
```

and `vmstat -i` shows one line, because the driver uses the Chapter 19 pattern on that one MSI vector.

The per-vector breakdown (three lines) is what makes the multi-vector driver observable. An operator watching the counters can tell which vector is active and how often.

20.4.9 Common Mistakes in Stage 2

A short list.

Passing the softc (not the vector) as the filter argument. If you pass `sc` instead of `vec`, the filter cannot tell which vector it is serving. Fix: pass `vec` to `bus_setup_intr`; the filter reaches `sc` through `vec->sc`.

Forgetting to initialise `vec->sc`. The per-vector structures are zero-initialised by `myfirst_init_softc`; `vec->sc` stays NULL unless explicitly set. Without it, the filter's `vec->sc->mtx` access is a null dereference. Fix: set `vec->sc = sc` during setup, before any handler is registered.

Using the same rid for multiple vectors. MSI rids are 1, 2, 3, ...; reusing rid 1 for both the admin and RX vectors means only one handler is actually registered. Fix: assign rids in sequence per vector.

Per-vector handler that touches shared state without locking. Two filters running on different CPUs both try to write to a single `sc->counter`. Without atomic operations or a spin lock, the increment loses updates. Fix: use per-vector counters where possible, and atomic ops for any shared counter.

Per-vector task stored in the wrong location. If the task is in the `softc` instead of in the vector structure, two vectors enqueueing the “same” task collide. Fix: store the task in the vector structure, and pass the vector as the task argument.

Missing per-vector teardown on partial setup failure. The goto cascade must undo exactly the vectors that succeeded. Missing cleanup leaves IRQ resources allocated. Fix: use the per-vector teardown helper and iterate backward on the partial-failure case.

20.4.10 Wrapping Up Section 3

Managing multiple interrupt vectors is a set of three new patterns: per-vector state (in a `struct my-first_vector` array), per-vector filter functions, and per-vector `bus_describe_intr` names. Each vector has its own IRQ resource at its own `rid`, its own filter function that reads only the status bits relevant to its vector, its own counters, and (optionally) its own deferred task. The single MSI or MSI-X allocation call handles the device-side state; the per-vector `bus_alloc_resource_any` and `bus_setup_intr` calls handle each handler individually.

The fallback ladder from Section 2 extends naturally: try MSI with N vectors first; on partial failure, release and try legacy INTx with a single handler. The per-vector teardown helper makes the partial-failure unwind and the clean teardown symmetric.

Section 4 is the locking-and-data-structures section. It examines what happens when multiple filters run on multiple CPUs at the same time, and what synchronisation discipline keeps the shared state sane.

20.5 Section 4: Designing Interrupt-Safe Data Structures

Section 3 added multiple vectors, each with its own handler. Section 4 examines the consequences: multiple handlers can run concurrently on multiple CPUs, and any data the handlers share must be protected accordingly. The discipline is not new; it is the multi-CPU specialisation of Chapter 11's locking model. What is new is that Chapter 20's driver has three (or more) concurrent filter-context paths instead of one.

Section 4 is the section where multi-vector changes the shape of the driver's state, not just the count of its handlers.

20.5.1 The New Concurrency Picture

Chapter 19's driver had one filter and one task. The filter ran on whichever CPU the kernel routed the interrupt to; the task ran on the taskqueue's worker thread. Two of them could in principle run simultaneously: the filter on CPU 0 and the task on CPU 3, for example. The atomic counters and the `sc->mtx` (held by the task, not by the filter) gave the needed synchronisation.

Chapter 20's multi-vector driver has three filters and one task. On an MSI-X system, each filter has its own `intr_event`, so each can fire on a different CPU independently. A burst of three interrupts arriving within a microsecond can see three filters running on three CPUs at the same time. The single task is still serialised through the taskqueue, but the filters are not.

The data the filters touch falls into three categories:

1. **Per-vector state.** Each vector's own counters, its own cookie, its own resource. No sharing between vectors. No synchronisation needed.

2. **Shared counters.** Counters updated by any filter (the global `intr_data_av_count`, `intr_error_count`, etc.). Must be atomic.
3. **Shared device state.** The BAR itself, the softc's `sc->hw` pointer, `sc->pci_attached`, the mutex-protected fields. Access rules depend on the context.

The discipline is to keep per-vector state truly per-vector, to use atomics for shared counters, and to obey the Chapter 11 locking rules for anything requiring a sleep mutex.

20.5.2 Per-Vector State: The Default

The easiest synchronisation is no synchronisation. If a piece of state is touched by only one vector's filter (and by nothing else), no lock is needed. This is the case for:

- `vec->fire_count`: incremented by this vector's filter only, read by sysctl handlers via the sysctl-reader path. Atomic add suffices; no lock between filter and sysctl because the sysctl reads atomically.
- `vec->stray_count`: same pattern.
- `vec->intr_cookie`: written once at setup, read at teardown. Single-writer, ordered access.
- `vec->irq_res`: same pattern.

Most per-vector state falls into this category. The struct `myfirst_vector` array in the softc is the key pattern: each vector's state lives in its own slot, touched only by its own filter.

20.5.3 Shared Counters: Atomic Operations

The global per-bit counters (Chapter 19 introduced `sc->intr_data_av_count`, etc.) are updated by the corresponding vector's filter. Only one filter updates each counter, so technically they are per-vector-except-by-name. But the reader can imagine a scenario where a bit pattern appears in `INTR_STATUS` that requires both the RX and the admin vectors to increment shared counters. The safer approach: make each update atomic.

Chapter 20 uses `atomic_add_64` throughout the filter paths:

```
atomic_add_64(&sc->intr_data_av_count, 1);
```

This is cheap (one locked instruction on x86, a barrier-plus-add on arm64), and it lets the filter run on any CPU without worrying about lost updates.

The cost of `atomic_add_64` on a heavily-shared counter is cache-line bouncing: every increment from a different CPU invalidates the cache line on the other CPUs. For a counter incremented a million times per second from several CPUs, this is a measurable performance hit. The mitigation is to make counters truly per-CPU (using `COUNTER(9)` or `DPCPU_DEFINE`) and sum them only when read; Chapter 20's driver is not at that scale, so plain atomics are fine.

20.5.4 Shared Device State: The Mutex Discipline

`sc->hw`, `sc->pci_attached`, `sc->bar_res`: these are set during attach and torn down during detach. During steady state, they are read-only. The filters access them without a lock because the lifetime discipline (attach before enabling, disable before detach) ensures the pointers are valid whenever the filter can run.

The rule: a filter that accesses `sc->hw` or `sc->bar_res` without a lock must be confident the attach-detach ordering guarantees the pointer is valid. Chapter 20's Section 7 walks the ordering in detail. For Section 4's purposes, trust the discipline: when the filter runs, the device is attached and the pointers are valid.

20.5.5 The Per-Vector Lock: When You Need It

Sometimes the per-vector state is richer than a counter. A vector that reads from a receive queue and updates a per-queue data structure (a ring of mbufs, say) needs a spin lock to protect the ring from two simultaneous firings of the same vector. Wait, can the same vector fire twice simultaneously on an MSI-X system?

On MSI-X, the kernel guarantees that each `intr_event` delivers to one CPU at a time; a single vector does not re-enter itself. Two different vectors can run on two CPUs at once, but vector N cannot run on CPU 3 and CPU 5 simultaneously.

This means: **per-vector state does not need a per-vector lock** for concurrent access from the same vector. It might need one for communication between the filter and the task (the task runs on a different CPU, possibly concurrently with the filter), but a spin lock suffices there, and the communication is usually through atomic operations anyway.

A spin lock becomes useful when:

- A driver uses a single filter function for multiple vectors, and the kernel can dispatch two vectors' worth of that filter concurrently. (Chapter 20's Stage 2 has separate filters per vector, so this does not apply.)
- A driver shares a receive ring between the filter (which fills it) and a task (which drains it). A spin lock protects the ring index; the filter acquires the spin lock, adds to the ring, releases. The task acquires, drains, releases.

Chapter 20's driver does not use spin locks in the filter; the per-vector counters are atomic and the shared state is handled through the existing `sc->mtx` in the task. Real drivers may need spin locks in richer scenarios.

20.5.6 Per-CPU Data: The Advanced Option

For very-high-rate drivers, even atomic counters on shared data become a bottleneck. The solution is per-CPU data: each CPU has its own copy of the counter, the filter increments its own CPU's copy (no cross-CPU traffic), and the `sysctl` reader sums the per-CPU values.

FreeBSD's `counter(9)` API provides this: a `counter_u64_t` is a handle to a per-CPU array, `counter_u64_add(c, 1)` increments the current CPU's slot, and `counter_u64_fetch(c)` sums all CPUs' slots on read. The implementation uses per-CPU data regions (`DPCPU_DEFINE` under the hood) and is as cheap as a normal non-atomic increment on the hot path.

Chapter 20's driver does not use `counter(9)`; plain atomics are enough for the demo's scale. Real high-throughput drivers (ten gigabit NICs, NVMe controllers at a million IOPS) use `counter(9)` extensively. A reader writing such a driver should study `counter(9)` after Chapter 20.

20.5.7 Lock Ordering and Multi-Vector Complications

Chapter 15 established the driver's lock order: `sc->mtx` -> `sc->cfg_sx` -> `sc->stats_cache_sx`. Chapter 19's filter took no locks (atomics only); the task took `sc->mtx`. Chapter 20's per-vector filters still take no

locks (atomics only), so the filter path does not contribute new lock-order edges. The per-vector tasks still take `sc->mtx`, same as Chapter 19's single task.

Lock ordering with multiple tasks running concurrently requires a small extension. When the admin task and the RX task both acquire `sc->mtx`, they serialise on the mutex. That is fine as long as each task releases the mutex promptly; if the admin task held `sc->mtx` while waiting on something slow, the RX task would stall. The Chapter 15 rule “no long-held mutexes” applies here too.

WITNESS catches most lock-order problems. For Chapter 20, the lock-order story is essentially unchanged from Chapter 19 because the filter paths are lock-free and the task paths all acquire the same single `sc->mtx`.

20.5.8 The Memory Model: Why Atomics Matter

A subtle point worth making explicit. In a multi-CPU system, writes from one CPU are not visible to other CPUs instantly. A write on CPU 0 to `sc->intr_count++` (without atomics) might land in CPU 0's store buffer and take nanoseconds or microseconds to propagate to CPU 3's view of the same memory. In that window, CPU 3 could read the pre-write value.

`atomic_add_64` includes a memory barrier that forces the write to become globally visible before the instruction returns. This is what makes the counter's value “consistent” across CPUs: any reader after the increment sees the new value.

For counter state, this level of consistency is sufficient. The counter's absolute value at any instant is not important; what matters is that the value grows monotonically and reaches the correct total. `atomic_add_64` guarantees both.

For richer shared state (say, a shared data-structure index that multiple filters update), the memory model gets more subtle. The driver would need a spin lock, which provides both mutual exclusion and a memory barrier. Chapter 20's driver does not need this level of machinery; Chapter 19's atomic discipline carries through.

20.5.9 Observability: Per-Vector Counters in sysctl

Each vector gets its own sysctl subtree so the operator can query it:

```
char name[32];
for (int i = 0; i < MYFIRST_MAX_VECTORS; i++) {
    snprintf(name, sizeof(name), "vec%d_fire_count", i);
    SYSCTL_ADD_U64(&sc->sysctl_ctx,
        SYSCTL_CHILDREN(sc->sysctl_tree), OID_AUTO, name,
        CTLFLAG_RD, &sc->vectors[i].fire_count, 0,
        "Fire count for this vector");
}
```

From user space:

```
sysctl dev.myfirst.0 | grep vec
```

```
dev.myfirst.0.vec0_fire_count: 42    # admin
dev.myfirst.0.vec0_stray_count: 0
dev.myfirst.0.vec1_fire_count: 9876  # rx
dev.myfirst.0.vec1_stray_count: 0
dev.myfirst.0.vec2_fire_count: 4523  # tx
dev.myfirst.0.vec2_stray_count: 0
```

The operator can see at a glance which vectors are firing and at what rates. The stray counts should stay at zero on MSI-X (each vector has its own dedicated message) but may tick on MSI or legacy when a shared filter sees an event for a different vector.

20.5.10 Wrapping Up Section 4

Multi-vector drivers change the concurrency picture: several filters can run on several CPUs simultaneously. The discipline is to design state per-vector where possible, use atomics for shared counters, and obey the existing Chapter 11 lock order for anything needing a sleep mutex. Per-CPU counters (`counter(9)`) are available for very-high-rate drivers but are overkill for Chapter 20.

The driver's lock order does not gain new edges because the filter path remains lock-free (atomics only) and the tasks all take `sc->mtx`. WITNESS still catches lock-order problems; the atomic discipline still catches the rest.

Section 5 moves to the more capable mechanism: MSI-X. The API is very similar (`pci_msix_count` + `pci_alloc_msix` instead of the MSI pair), but the scalability and CPU-affinity options are richer.

20.6 Section 5: Using MSI-X for High Flexibility

Section 2 introduced MSI with one vector, Section 3 extended to multiple MSI vectors, Section 4 walked through the concurrency implications. Section 5 moves to MSI-X: the fuller mechanism that modern PCIe devices use when they have more than a handful of interrupts to manage. The API is parallel to MSI's, so the code change is small; the conceptual change is that MSI-X lets the driver bind each vector to a specific CPU, through `bus_bind_intr(9)` and `bus_get_cpus(9)`, and that matters for real performance.

20.6.1 The MSI-X Count and Allocation API

The API mirrors MSI's:

```
int msix_count = pci_msix_count(sc->dev);
```

`pci_msix_count(9)` returns the number of MSI-X vectors the device advertises (0 if no MSI-X capability). The count comes from the MSI-X capability's `Table Size` field plus one; a device with `Table Size` = 7 advertises 8 vectors.

Allocating is similar:

```
int count = desired;
int error = pci_alloc_msix(sc->dev, &count);
```

Same input-output count parameter, same semantics: the kernel may allocate fewer than requested. Unlike MSI, MSI-X allows a non-power-of-two count, so if the driver asks for 3 vectors, the kernel can give 3.

The same `pci_release_msi(9)` call releases MSI-X vectors; there is no separate `pci_release_msix`. The function name is a historical artefact; it handles both MSI and MSI-X.

20.6.2 The Fallback Ladder Extended

The Chapter 20 driver's full fallback ladder is:

1. **MSI-X** with the desired vector count.
2. **MSI** with the desired vector count, if MSI-X is unavailable or allocation failed.
3. **Legacy INTx** with a single handler for everything, if both MSI-X and MSI fail.

The code structure parallels Section 3's two-tier ladder, extended with a third tier at the top:

```
/* Tier 0: MSI-X. */
wanted = MYFIRST_MAX_VECTORS;
if (pci_msix_count(sc->dev) >= wanted) {
    allocated = wanted;
    if (pci_alloc_msix(sc->dev, &allocated) == 0 &&
        allocated == wanted) {
        for (i = 0; i < wanted; i++) {
            error = myfirst_intr_setup_vector(sc, i, i + 1);
            if (error != 0)
                goto fail_msix;
        }
        sc->intr_mode = MYFIRST_INTR_MSIX;
        sc->num_vectors = wanted;
        device_printf(sc->dev,
            "interrupt mode: MSI-X, %d vectors\n", wanted);
        myfirst_intr_bind_vectors(sc);
        goto enabled;
    }
    if (allocated > 0)
        pci_release_msi(sc->dev);
}

/* Tier 1: MSI. */
/* ... Section 3 MSI code ... */

fail_msix:
```



```
for (i -= 1; i >= 0; i--)
    myfirst_intr_teardown_vector(sc, i);
pci_release_msi(sc->dev);
/* fallback to MSI attempt, then legacy. */
```

The structure is straightforward: each tier has the same pattern (query count, allocate, set up vectors, mark mode, describe). The code follows the well-worn waterfall.

20.6.3 Vector Binding With `bus_bind_intr`

Once MSI-X is allocated, the driver has the option of binding each vector to a specific CPU. The API is:

```
int bus_bind_intr(device_t dev, struct resource *r, int cpu);
```

The `cpu` is an integer CPU ID from 0 to `mp_ncpus - 1`. On success, the interrupt is routed to that CPU. On failure, the function returns an `errno`; the driver treats it as a non-fatal hint and continues without binding.

For Chapter 20's three-vector driver, a reasonable binding is:

- **Admin vector:** CPU 0 (control work, any CPU is fine).
- **RX vector:** CPU 1 (cache locality benefits for a real RX queue).
- **TX vector:** CPU 2 (similar locality benefit).

On a two-CPU system, the bindings would collapse; on a many-CPU system, the driver should use `bus_get_cpus(9)` to query which CPUs are local to the device's NUMA node and distribute vectors accordingly.

The bind helper:

```
static void
myfirst_intr_bind_vectors(struct myfirst_softc *sc)
{
    int i, cpu, ncpus;
    int err;

    if (mp_ncpus < 2)
        return; /* nothing to bind */

    ncpus = mp_ncpus;
    for (i = 0; i < sc->num_vectors; i++) {
        cpu = i % ncpus;
        err = bus_bind_intr(sc->dev, sc->vectors[i].irq_res, cpu);
        if (err != 0) {
            device_printf(sc->dev,
                "bus_bind_intr vector %d to CPU %d: %d\n",

```

```

        i, cpu, err);
    }
}

```

The code is a round-robin bind: vector 0 to CPU 0, vector 1 to CPU 1, and so on, wrapping modulo the CPU count. On a two-CPU system with three vectors, vector 0 and vector 2 both land on CPU 0; on a four-CPU system, each vector gets its own CPU.

A more sophisticated driver uses `bus_get_cpus(9)`:

```

cpuset_t local_cpus;
int ncpus_local;

if (bus_get_cpus(sc->dev, LOCAL_CPUS, sizeof(local_cpus),
    &local_cpus) == 0) {
    /* Use only CPUs in local_cpus for binding. */
    ncpus_local = CPU_COUNT(&local_cpus);
    /* ... pick from local_cpus ... */
}

```

The `LOCAL_CPUS` argument returns the CPUs that are in the same NUMA domain as the device. The `INTR_CPUS` argument returns CPUs suitable for handling device interrupts (usually excluding CPUs pinned to critical work). A driver that cares about NUMA performance uses these to place vectors on NUMA-local CPUs.

Chapter 20's driver does not use `bus_get_cpus(9)` by default; the simpler round-robin bind is enough for the lab. A challenge exercise adds NUMA-aware binding.

20.6.4 The MSI-X dmesg Summary

The Chapter 20 driver prints a line like:

```
myfirst0: interrupt mode: MSI-X, 3 vectors
```

With per-vector CPU bindings visible in `vmstat -i` (the per-CPU totals in `vmstat -i` are not per-vector; they are aggregated) and in `cpuset -g -x <irq>` output (one query per vector):

```

for irq in 256 257 258; do
    echo "IRQ $irq:"
    cpuset -g -x $irq
done

```

Typical output:

```
IRQ 256:
irq 256 mask: 0
IRQ 257:
irq 257 mask: 1
IRQ 258:
irq 258 mask: 2
```

(The IRQ numbers depend on the platform's assignment.)

An operator inspecting the driver's interrupt setup can see which vectors fire where.

20.6.5 Per-Vector `bus_describe_intr`

Each MSI-X vector should have a description. The code from Section 3 already sets them via `bus_describe_intr(9)`:

```
bus_describe_intr(sc->dev, vec->irq_res, vec->intr_cookie,
    "%s", vec->name);
```

After this, `vmstat -i` shows each vector with its role:

```
irq256: myfirst0:admin          42          4
irq257: myfirst0:rx            12345        1234
irq258: myfirst0:tx            5432         543
```

The operator sees which vector is the admin, which is the RX, which is the TX, and how busy each is. This is essential observability for a multi-vector driver.

20.6.6 MSI-X Table and BAR Considerations

A detail worth mentioning, though the driver does not interact with it directly. The MSI-X capability structure points to a **table** and a **pending bit array** (PBA), each living in one of the device's BARs. The BAR that holds each is discoverable through `pci_msix_table_bar(9)` and `pci_msix_pba_bar(9)`:

```
int table_bar = pci_msix_table_bar(sc->dev);
int pba_bar = pci_msix_pba_bar(sc->dev);
```

Each returns the BAR index (0 through 5) or -1 if the device has no MSI-X capability. For most devices, the table and PBA are in BAR 0 or BAR 1; for some devices, they share a BAR with memory-mapped registers (the driver's BAR 0).

The kernel handles the table programming internally. The driver's only interaction is:

- Ensure the BAR containing the table is allocated (so the kernel can reach it). On some devices this requires the driver to allocate additional BARs.

- Call `pci_alloc_msix` and let the kernel do the rest.

For Chapter 20's driver, the virtio-rnd target (or its QEMU equivalent with MSI-X) has the table in BAR 1 or a dedicated region. The Chapter 18 code allocated BAR 0; the kernel handles the MSI-X table BAR implicitly through the allocation infrastructure.

A driver that wants to inspect the table bar:

```
device_printf(sc->dev, "MSI-X table in BAR %d, PBA in BAR %d\n",
    pci_msix_table_bar(sc->dev), pci_msix_pba_bar(sc->dev));
```

This is useful for diagnostic purposes.

20.6.7 Allocating Fewer Vectors Than Requested

A subtle case: the device advertises 3 MSI-X vectors and the driver asks for 3, but the kernel only allocates 2. What does the driver do?

The answer depends on the driver's design. Options:

1. **Fail the attach.** If the driver cannot function with fewer vectors, return an error. This is rare for flexible drivers but possible for drivers with tight hardware requirements.
2. **Use what we got.** If the driver can function with 2 vectors (folding RX and TX into one, for example), use the 2 and adjust the configuration. This is common for drivers targeting ranges of hardware.
3. **Release and fall back.** If 2 vectors is worse than 1 MSI vector for some reason, release MSI-X and try MSI. This is uncommon.

Chapter 20's driver goes with option 1: if it does not get exactly `MYFIRST_MAX_VECTORS` (3) vectors, it releases MSI-X and falls back to MSI. A more sophisticated driver would use option 2; the Chapter 20 teaching focuses on the simpler pattern.

Real FreeBSD drivers often use option 2 with a helper function that figures out how to distribute the allocated vectors across the desired roles. The `nvme(4)` driver is an example: if it asks for `N` I/O queues worth of vectors and gets fewer, it reduces the number of I/O queues accordingly.

20.6.8 Testing MSI-X on bhyve vs QEMU

A practical detail about the lab. `bhyve`'s legacy virtio-rnd device (what Chapter 18 and 19 used) does not expose MSI-X; it is a legacy-only virtio transport. To exercise MSI-X in a guest, the reader needs one of:

- **QEMU with `-device virtio-rng-pci`** (not `-device virtio-rng`, which is legacy). The modern `virtio-rng-pci` exposes MSI-X.
- **Modern bhyve emulation** of a non-virtio-rnd device that has MSI-X. Chapter 20 does not use this path.
- **Real hardware** that supports MSI-X (most modern PCIe devices).

QEMU is the practical choice for Chapter 20's labs. The driver's fallback ladder ensures it still works on `bhyve` (falling back to legacy); testing MSI-X specifically requires QEMU or real hardware.

20.6.9 Common Mistakes in MSI-X Setup

A short list.

Using `pci_release_msix`. This function does not exist in FreeBSD; the release is handled by `pci_release_msi(9)` which works for both MSI and MSI-X. Fix: use `pci_release_msi`.

Binding to a CPU the device cannot reach. Some platforms (rarely) have CPUs that are not in the interrupt-controller's routable set. The `bus_bind_intr` call returns an error; ignore it and continue. Fix: log the error but do not fail attach.

Expecting `vmstat -i` to show per-CPU breakdown. `vmstat -i` aggregates per-event counts. The per-CPU breakdown is available via `cpuset -g -x <irq>` (or `sysctl hw.intrcnt` in raw form). The operator has to look in the right place. Fix: document the observability path for your driver.

Failing to check `allocated` against `wanted`. Accepting a partial allocation when the driver cannot handle it leads to subtle bugs (vectors that should fire never fire). Fix: decide the strategy upfront (fail, adapt, or release) and code accordingly.

20.6.10 Wrapping Up Section 5

MSI-X is the fuller mechanism: a per-vector addressable table that the kernel programs on behalf of the driver, with per-vector CPU affinity and per-vector masking available to drivers that need them. The API mirrors MSI's closely (`pci_msix_count` + `pci_alloc_msix` + `pci_release_msi`), and the per-vector resource allocation is the same as Section 3's MSI code. The new piece is `bus_bind_intr(9)` for CPU affinity and `bus_get_cpus(9)` for NUMA-local CPU queries.

For Chapter 20's driver, MSI-X is the preferred tier; the fallback ladder tries MSI-X first, falls back to MSI, and finally to legacy INTx. The per-vector handlers, counters, and tasks from Section 3 work unchanged on MSI-X; only the allocation call changes.

Section 6 is where vector-specific events become explicit. Each vector has its own purpose, its own filter logic, and its own observable behaviour. Chapter 20's Stage 3 is the stage where the driver looks like a real multi-queue device even though the underlying silicon (the `virtio-rnd` target) is simpler.

20.7 Section 6: Handling Vector-Specific Events

Sections 2 through 5 built the infrastructure for multi-vector handling. Section 6 is the section where the per-vector roles become explicit. Each vector has a specific event class it handles; each filter has a specific check it makes; each task has a specific wake-up it performs. The driver at Stage 3 treats vectors as named, purposeful entities rather than interchangeable slots.

The Chapter 20 driver's three vectors have distinct responsibilities:

- **Admin vector** handles `ERROR` events. The filter reads status, acknowledges, and (on real errors) logs a message. Admin work is infrequent but must not be dropped.
- **RX vector** handles `DATA_AV` (receive-available) events. The filter acknowledges and defers the data-handling work to a per-vector task that broadcasts a condition variable.

- **TX vector** handles COMPLETE (transmit-complete) events. The filter acknowledges and optionally wakes a thread waiting for transmit completion. The filter handles the bookkeeping inline.

Each vector is independently testable through the simulated-interrupt sysctl, independently observable through its counters, and independently bound to a CPU. The driver is starting to look like a small real multi-queue device.

20.7.1 The Admin Vector

The admin vector handles rare, important events: configuration changes, errors, link state changes (for a NIC), temperature alerts (for a sensor). Its work is usually small: log the event, update a state flag, wake a user-space waiter that polls the state.

For the Chapter 20 driver, the admin vector handles the Chapter 17 ERROR bit. The filter:

```
int
myfirst_admin_filter(void *arg)
{
    struct myfirst_vector *vec = arg;
    struct myfirst_softc *sc = vec->sc;
    uint32_t status;

    status = ICSR_READ_4(sc, MYFIRST_REG_INTR_STATUS);
    if ((status & MYFIRST_INTR_ERROR) == 0) {
        atomic_add_64(&vec->stray_count, 1);
        return (FILTER_STRAY);
    }

    atomic_add_64(&vec->fire_count, 1);
    ICSR_WRITE_4(sc, MYFIRST_REG_INTR_STATUS, MYFIRST_INTR_ERROR);
    atomic_add_64(&sc->intr_error_count, 1);
    return (FILTER_HANDLED);
}
```

On a real device, the admin filter might also examine a secondary register (an error code register, say) and decide based on the severity whether to schedule a recovery task. Chapter 20's driver keeps it simple: count and acknowledge.

20.7.2 The RX Vector

The RX vector is the data-path vector. For a NIC, it would handle received packets. For an NVMe drive, completions of read requests. For the Chapter 20 driver, it handles the Chapter 17 DATA_AV bit.

The filter is small (acknowledge and enqueue a task); the task does the real work. Section 3 showed both. The task:

```
static void
myfirst_rx_task_fn(void *arg, int npending)
{
    struct myfirst_vector *vec = arg;
    struct myfirst_softc *sc = vec->sc;

    MYFIRST_LOCK(sc);
    if (sc->hw != NULL && sc->pci_attached) {
        sc->intr_last_data = CSR_READ_4(sc, MYFIRST_REG_DATA_OUT);
        sc->intr_task_invocations++;
        cv_broadcast(&sc->data_cv);
    }
    MYFIRST_UNLOCK(sc);
}
```

In a real driver, the task would walk a receive descriptor ring, build mbufs, and pass them up the network stack. For Chapter 20's demo, it reads DATA_OUT, stores the value, broadcasts the condition variable, and lets any waiting cdev reader wake up.

The npending argument is the number of times the task was enqueued since its last run. For a high-rate RX path, a task that ran once and saw npending = 5 knows it is behind (5 interrupts coalesced into 1 task run) and can size its batch accordingly. Chapter 20's task ignores npending; real drivers use it for batching.

20.7.3 The TX Vector

The TX vector is the transmit-completion vector. For a NIC, it signals that a packet the driver handed to hardware has been transmitted and the buffer can be reclaimed. For an NVMe drive, it signals that a write request has completed.

For the Chapter 20 driver, it handles the Chapter 17 COMPLETE bit. The filter does the work inline (no task needed):

```
int
myfirst_tx_filter(void *arg)
{
    struct myfirst_vector *vec = arg;
    struct myfirst_softc *sc = vec->sc;
    uint32_t status;

    status = ICSR_READ_4(sc, MYFIRST_REG_INTR_STATUS);
    if ((status & MYFIRST_INTR_COMPLETE) == 0) {
        atomic_add_64(&vec->stray_count, 1);
        return (FILTER_STRAY);
    }
}
```

```
    atomic_add_64(&vec->fire_count, 1);
    ICSR_WRITE_4(sc, MYFIRST_REG_INTR_STATUS, MYFIRST_INTR_COMPLETE);
    atomic_add_64(&sc->intr_complete_count, 1);
    return (FILTER_HANDLED);
}
```

The TX filter's inline-only design is a deliberate choice. On a real TX completion path, the filter might record the completion count and the task might walk the TX descriptor ring to reclaim buffers. For the Chapter 20 demo, the completion is just counted.

An alternative design would have TX also use a task. Whether to do this depends on the work the task would do: if it is substantial (walking a ring, reclaiming dozens of buffers), a task is worth it; if it is trivial (a single decrement of an in-flight counter), inline in the filter is fine. Chapter 20 picks inline for TX to illustrate that not every vector needs a task.

20.7.4 Vector-to-Event Mapping

On MSI-X, each vector is independent; firing vector 1 delivers to the RX filter, firing vector 2 delivers to the TX filter. The mapping from vector to event is part of the driver's design, not a kernel choice.

On MSI with multiple vectors, the kernel may in principle dispatch multiple vectors in rapid succession if multiple events fire simultaneously. The driver's filters must each read the status register and claim only the bits belonging to their vector.

On legacy INTx, there is only one vector and one filter. The filter handles all three event classes in one pass.

Chapter 20's code handles all three cases: the per-vector filter on MSI-X reads only its own bit, the per-vector filter on MSI does the same (with the same bit-check logic), and the single filter on legacy INTx handles all three bits.

20.7.5 Simulated Per-Vector Events

The simulation sysctl from Section 3 lets a reader exercise each vector independently. From user space:

```
# Simulate an admin interrupt (ERROR).
sudo sysctl dev.myfirst.0.intr_simulate_admin=2

# Simulate an RX interrupt (DATA_AV).
sudo sysctl dev.myfirst.0.intr_simulate_rx=1

# Simulate a TX interrupt (COMPLETE).
sudo sysctl dev.myfirst.0.intr_simulate_tx=4
```

Each sysctl writes its specific bit to INTR_STATUS and invokes the corresponding vector's filter. On the MSI-X tier all three filters exist, so each sysctl hits its own per-vector filter and its per-vector counter ticks. On the MSI tier (single vector at slot 0) and on the legacy INTx tier (single vector at slot 0), slots 1 and 2 have no registered

filter, so the simulation helper routes the call through slot 0's filter. The Chapter 19 `myfirst_intr_filter` handles all three bits inline, so the global `intr_count`, `intr_data_av_count`, `intr_error_count`, and `intr_complete_count` counters still move correctly. The per-vector counters on slots 1 and 2 stay at zero in single-handler tiers, which is the right observability signal that the driver is not running with three vectors.

A reader can watch the pipeline from user space:

```
while true; do
    sudo sysctl dev.myfirst.0.intr_simulate_rx=1
    sleep 0.1
done &
watch sysctl dev.myfirst.0 | grep -E "vec|intr_"
```

The counters tick at roughly 10 per second. The RX vector's counter matches `intr_data_av_count`; the task's invocation count matches.

20.7.6 Dynamic Vector Assignment

A subtle but important point. The driver's design has three vectors in a fixed array with fixed roles. A more flexible driver might discover the number of available vectors at runtime and assign roles dynamically. The pattern looks like:

```
/* Discover how many vectors we got. */
int nvec = actually_allocated_msix_vectors(sc);

/* Assign roles based on nvec. */
if (nvec >= 3) {
    /* Full design: admin, rx, tx. */
    sc->vectors[0].filter = myfirst_admin_filter;
    sc->vectors[1].filter = myfirst_rx_filter;
    sc->vectors[2].filter = myfirst_tx_filter;
    sc->num_vectors = 3;
} else if (nvec == 2) {
    /* Compact: admin+tx share one vector, rx has its own. */
    sc->vectors[0].filter = myfirst_admin_tx_filter;
    sc->vectors[1].filter = myfirst_rx_filter;
    sc->num_vectors = 2;
} else if (nvec == 1) {
    /* Minimal: one filter handles everything. */
    sc->vectors[0].filter = myfirst_intr_filter;
    sc->num_vectors = 1;
}
```

This dynamic adaptation is what production drivers do. Chapter 20's driver uses the simpler fixed approach; a challenge exercise adds the dynamic variant.

20.7.7 A Pattern From nvme(4)

For a real example, the nvme(4) driver handles the admin queue separately from the I/O queues. Its filter functions differ per queue type; its interrupt counts are tracked per queue. The pattern is:

```
/* In nvme_ctrlr_construct_admin_qpair: */
qpair->intr_idx = 0; /* vector 0 for admin */
qpair->intr_rid = 1;
qpair->res = bus_alloc_resource_any(ctrlr->dev, SYS_RES_IRQ,
    &qpair->intr_rid, RF_ACTIVE);
bus_setup_intr(ctrlr->dev, qpair->res, INTR_TYPE_MISC | INTR_MPSAFE,
    NULL, nvme_qpair_msix_handler, qpair, &qpair->tag);

/* For each I/O queue: */
for (i = 0; i < ctrlr->num_io_queues; i++) {
    ctrlr->ioq[i].intr_rid = i + 2; /* I/O vectors at rid 2, 3, ... */
    /* ... similar bus_alloc_resource_any + bus_setup_intr ... */
}
```

Each queue has its own `intr_rid`, its own resource, its own tag (cookie), its own handler argument. The admin queue uses one vector; each I/O queue uses its own vector. The pattern scales linearly with the number of queues.

Chapter 20's driver is a small version of this: three fixed vectors instead of one admin plus N I/O. The scaling story transfers directly.

20.7.8 Observability: Per-Vector Rate

A useful diagnostic: compute each vector's rate over a sliding window:

```
#!/bin/sh
prev_admin=$(sysctl -n dev.myfirst.0.vec0_fire_count)
prev_rx=$(sysctl -n dev.myfirst.0.vec1_fire_count)
prev_tx=$(sysctl -n dev.myfirst.0.vec2_fire_count)
sleep 1
curr_admin=$(sysctl -n dev.myfirst.0.vec0_fire_count)
curr_rx=$(sysctl -n dev.myfirst.0.vec1_fire_count)
curr_tx=$(sysctl -n dev.myfirst.0.vec2_fire_count)

echo "admin: $((curr_admin - prev_admin)) /s"
echo "rx:    $((curr_rx - prev_rx )) /s"
echo "tx:    $((curr_tx - prev_tx )) /s"
```

The output is per-vector rates over the last second. A reader running the simulated-interrupt `sysctl` in a loop can see the rates tick up; a reader observing a real workload sees which vector is busy.

20.7.9 Wrapping Up Section 6

Handling vector-specific events means each vector has its own filter function, its own counters, its own (optional) task, and its own observable behaviour. The pattern scales: three vectors for the Chapter 20 demo, dozens for a production NIC, hundreds for an NVMe controller. The per-vector separation makes each piece small, specific, and maintainable.

Section 7 is the teardown section. Multi-vector drivers need to tear down each vector individually, in the right order, and then call `pci_release_msi` once at the end. The order is strict but not complex; Section 7 walks through it.

20.8 Section 7: Teardown and Cleanup With MSI/MSI-X

Chapter 19's teardown was a single pair: `bus_teardown_intr` on the one vector, then `bus_release_resource` on the one IRQ resource. Chapter 20's teardown is the same pair repeated per vector, followed by a single `pci_release_msi` call that undoes the MSI or MSI-X device-level allocation.

Section 7 makes the ordering precise, walks through the partial-failure cases, and highlights the observability checks that confirm a clean teardown.

20.8.1 The Required Order

For a multi-vector driver, the detach sequence is:

1. **Refuse if busy.** Same as Chapter 19: return `EBUSY` if the driver has open descriptors or in-flight work.
2. **Mark as no longer attached.**
3. **Destroy the cdev.**
4. **Disable interrupts at the device.** Clear `INTR_MASK` so the device stops asserting.
5. **For each vector in reverse order:**
 - a. `bus_teardown_intr` on the vector's cookie.
 - b. `bus_release_resource` on the vector's IRQ resource.
6. **Drain all per-vector tasks.** Each task that was initialised.
7. **Destroy the taskqueue.**
8. **Call `pci_release_msi` once,** unconditionally if `intr_mode` is MSI or MSI-X.
9. **Detach the hardware layer and release the BAR** as usual.
10. **Deinit the softc.**

The new steps are 5 (per-vector loop instead of a single pair) and 8 (`pci_release_msi`). Steps 1-4 and 9-10 are unchanged from Chapter 19.

20.8.2 Why Reverse Order Per Vector

The per-vector reverse-order loop is a defensive measure against dependencies between vectors. On a simple driver like Chapter 20's, the vectors are independent: tearing down vector 2 before vector 1 is fine. On a driver

where vector 2's filter reads state that vector 1's filter writes, the order matters: tear down the writer (vector 1) first, then the reader (vector 2).

For correctness on Chapter 20's driver, forward and reverse order are both safe. For robustness against future changes, reverse order is preferred.

20.8.3 The Per-Vector Teardown Code

From Section 3, the per-vector teardown helper:

```
static void
myfirst_intr_teardown_vector(struct myfirst_softc *sc, int idx)
{
    struct myfirst_vector *vec = &sc->vectors[idx];

    if (vec->intr_cookie != NULL) {
        bus_teardown_intr(sc->dev, vec->irq_res, vec->intr_cookie);
        vec->intr_cookie = NULL;
    }
    if (vec->irq_res != NULL) {
        bus_release_resource(sc->dev, SYS_RES_IRQ, vec->irq_rid,
            vec->irq_res);
        vec->irq_res = NULL;
    }
}
```

The helper is robust against partial setup: if the vector never had a cookie (setup failed before `bus_setup_intr`), the `if` check skips the teardown call. If the resource was never allocated, the second `if` check skips the release. The same helper works for partial-failure unwind during setup and for full teardown during detach.

20.8.4 The Full Teardown

```
void
myfirst_intr_teardown(struct myfirst_softc *sc)
{
    int i;

    MYFIRST_LOCK(sc);
    if (sc->hw != NULL && sc->bar_res != NULL)
        CSR_WRITE_4(sc, MYFIRST_REG_INTR_MASK, 0);
    MYFIRST_UNLOCK(sc);

    /* Tear down each vector's handler, in reverse. */
}
```

```

for (i = sc->num_vectors - 1; i >= 0; i--)
    myfirst_intr_teardown_vector(sc, i);

/* Drain and destroy the taskqueue, including per-vector tasks. */
if (sc->intr_tq != NULL) {
    for (i = 0; i < sc->num_vectors; i++) {
        if (sc->vectors[i].has_task)
            taskqueue_drain(sc->intr_tq,
                            &sc->vectors[i].task);
    }
    taskqueue_free(sc->intr_tq);
    sc->intr_tq = NULL;
}

/* Release the MSI/MSI-X allocation if used. */
if (sc->intr_mode == MYFIRST_INTR_MSI ||
    sc->intr_mode == MYFIRST_INTR_MSIX)
    pci_release_msi(sc->dev);

sc->num_vectors = 0;
sc->intr_mode = MYFIRST_INTR_LEGACY;
}

```

The code structure is straightforward: disable at the device, per-vector teardown in reverse, per-vector task drain, free the taskqueue, release MSI if used. The pattern transfers directly to any multi-vector driver.

20.8.5 Partial-Attach Failure Unwind

During setup, if vector N fails to register, the code must unwind vectors 0 through N-1 that succeeded. The pattern:

```

for (i = 0; i < MYFIRST_MAX_VECTORS; i++) {
    error = myfirst_intr_setup_vector(sc, i, i + 1);
    if (error != 0)
        goto fail_vectors;
}

/* Success, continue. */

fail_vectors:
    /* Undo vectors 0 through i-1. */
    for (i -= 1; i >= 0; i--)
        myfirst_intr_teardown_vector(sc, i);

```

```
pci_release_msi(sc->dev);  
/* Fall through to next tier or final failure. */
```

The `i -= 1` is important: after the `goto`, `i` is the vector that failed (it is past the successful setups). We undo vectors 0 through `i-1`, which is the successfully-registered set. The per-vector teardown helper is safe to call on the failing vector's slot too, because its fields are NULL (the setup did not get far enough to populate them).

20.8.6 Observability: Verifying a Clean Teardown

After `kldunload myfirst`, the following should hold:

- `kldstat -v | grep myfirst` returns nothing.
- `devinfo -v | grep myfirst` returns nothing.
- `vmstat -i | grep myfirst` returns nothing.
- `vmstat -m | grep myfirst` shows zero live allocations.

Any failure points to a cleanup bug:

- A `vmstat -i` entry remaining means `bus_teardown_intr` was not called for that vector.
- A `vmstat -m` leak means a per-vector task was not drained or the taskqueue was not freed.
- A `devinfo -v` entry remaining (rare) means the device's detach did not complete.

20.8.7 MSI Resource Leak Across Load-Unload Cycles

A specific concern for MSI/MSI-X drivers: forgetting `pci_release_msi` leaves the device's MSI state allocated. The next `kldload` of the same driver (or a different driver for the same device) will fail to allocate MSI vectors because the kernel thinks they are already in use.

The symptom in `dmesg`:

```
myfirst0: pci_alloc_msix returned EBUSY
```

or similar. The fix is to ensure `pci_release_msi` runs on every teardown path, including the partial-failure unwind.

A useful test: load, unload, load. If the second load succeeds with the same MSI mode, the teardown is correct. If the second load falls back to a lower tier, the teardown leaked.

20.8.8 Common Mistakes in Teardown

A short list.

Forgetting `pci_release_msi`. The most common bug. Symptom: next MSI allocation attempts fail. Fix: always call it when MSI or MSI-X was used.

Calling `pci_release_msi` on a driver that used legacy INTx only. Technically a no-op, but explicit check makes the intent clearer. Fix: check `intr_mode` before calling.

Wrong per-vector teardown order. For drivers with inter-vector dependencies, the reverse-order loop matters. For Chapter 20's driver, the order is not dependency-critical, but the reverse-order discipline is cheap and worth keeping.

Draining a task that was never initialised. If a vector does not have `has_task`, draining its uninitialised task field produces garbage. Fix: check `has_task` before draining.

Leaking the taskqueue. `taskqueue_drain` does not free the taskqueue; `taskqueue_free` does. Both are needed. Fix: call both.

Partial-setup unwind that undoes too much. If vector 2 fails and the unwind code also tears down vector 2 (which was never set up), NULL dereferences follow. The per-vector helper's NULL checks protect against this, but the cascade logic should also be careful. Fix: use `i -= 1` to start the unwind at the correct vector.

20.8.9 Wrapping Up Section 7

Teardown for a multi-vector driver is per-vector in a loop, followed by a single `pci_release_msi` at the end. The per-vector helper is shared between full-teardown and partial-failure-unwind. The observability checks after unload are the same ones Chapter 19 used; any leak points to a specific bug.

Section 8 is the refactor section: split the multi-vector code into `myfirst_msix.c`, update `INTERRUPTS.md` to reflect the new capabilities, bump the version to `1.3-msi`, and run the regression pass. The driver is functionally complete after Section 7; Section 8 makes it maintainable.

20.9 Section 8: Refactoring and Versioning Your Multi-Vector Driver

The multi-vector interrupt handler is working. Section 8 is the housekeeping section. It splits the MSI/MSI-X code into its own file, updates the module metadata, extends the `INTERRUPTS.md` document with the new multi-vector details, bumps the version to `1.3-msi`, and runs the regression pass.

This is the fourth chapter running in which the chapter closes with a refactor section. The refactors accumulate: Chapter 16 split out the hardware layer, Chapter 17 the simulation, Chapter 18 the PCI attach, Chapter 19 the legacy interrupt. Chapter 20 adds the MSI/MSI-X layer. Each responsibility has its own file; the main `myfirst.c` stays roughly constant in size; the driver scales.

20.9.1 The Final File Layout

At the end of Chapter 20:

<code>myfirst.c</code>	- Main driver
<code>myfirst.h</code>	- Shared declarations
<code>myfirst_hw.c</code>	- Ch16 hardware access layer
<code>myfirst_hw_pci.c</code>	- Ch18 hardware-layer extension
<code>myfirst_hw.h</code>	- Register map
<code>myfirst_sim.c</code>	- Ch17 simulation backend
<code>myfirst_sim.h</code>	- Simulation interface

```

myfirst_pci.c      - Ch18 PCI attach
myfirst_pci.h      - PCI declarations
myfirst_intr.c     - Ch19 interrupt handler (legacy + filter+task)
myfirst_intr.h     - Ch19 interrupt interface + ICSR macros
myfirst_msix.c     - Ch20 MSI/MSI-X multi-vector layer (NEW)
myfirst_msix.h     - Ch20 multi-vector interface (NEW)
myfirst_sync.h     - Part 3 synchronisation
cbuf.c / cbuf.h    - Ch10 circular buffer
Makefile          - kmod build
HARDWARE.md, LOCKING.md, SIMULATION.md, PCI.md, INTERRUPTS.md, MSIX.md (NEW)

```

myfirst_msix.c and myfirst_msix.h are new. MSIX.md is new. The Chapter 19 myfirst_intr.c stays; it now handles the legacy-INTx fallback while myfirst_msix.c handles the MSI and MSI-X path.

20.9.2 The myfirst_msix.h Header

```

#ifndef _MYFIRST_MSIX_H_
#define _MYFIRST_MSIX_H_

#include <sys/taskqueue.h>

struct myfirst_softc;

enum myfirst_intr_mode {
    MYFIRST_INTR_LEGACY = 0,
    MYFIRST_INTR_MSI = 1,
    MYFIRST_INTR_MSIX = 2,
};

enum myfirst_vector_id {
    MYFIRST_VECTOR_ADMIN = 0,
    MYFIRST_VECTOR_RX,
    MYFIRST_VECTOR_TX,
    MYFIRST_MAX_VECTORS
};

struct myfirst_vector {
    struct resource      *irq_res;
    int                  irq_rid;
    void                 *intr_cookie;
    enum myfirst_vector_id id;
    struct myfirst_softc *sc;
    uint64_t             fire_count;
};

```



```
uint64_t      stray_count;
const char    *name;
driver_filter_t *filter;
struct task   task;
bool          has_task;
};

int  myfirst_msix_setup(struct myfirst_softc *sc);
void myfirst_msix_teardown(struct myfirst_softc *sc);
void myfirst_msix_add_sysctls(struct myfirst_softc *sc);

#endif /* _MYFIRST_MSIX_H_ */
```

The public API is three functions: `setup`, `teardown`, `add_sysctls`. The enum types and the per-vector struct are exported so `myfirst.h` can include them and the softc can have the per-vector array.

20.9.3 The Full Makefile

```
# Makefile for the Chapter 20 myfirst driver.

KMOD=  myfirst
SRCS=  myfirst.c \
       myfirst_hw.c myfirst_hw_pci.c \
       myfirst_sim.c \
       myfirst_pci.c \
       myfirst_intr.c \
       myfirst_msix.c \
       cbuf.c

CFLAGS+= -DMYFIRST_VERSION_STRING=\"1.3-msi\"

.include <bsd.kmod.mk>
```

One additional source file in the SRCS list; the version string bumped.

20.9.4 The Version String

1.2-intr to 1.3-msi. The bump reflects a significant capability addition: multi-vector interrupt handling. A minor-version bump is appropriate; the user-visible interface (the cdev) did not change.

20.9.5 The MSIX.md Document

A new document lives next to the source:

MSI and MSI-X Support in the myfirst Driver

Summary

The driver probes the device's MSI-X, MSI, and legacy INTx capabilities in that order, and uses the first one that allocates successfully. The driver's interrupt counters, data path, and cdev behaviour are independent of which tier the driver ends up using.

Setup Sequence

``myfirst_msix_setup()`` tries three tiers:

1. MSI-X with MYFIRST_MAX_VECTORS (3) vectors. On success:
 - Allocates per-vector IRQ resources at rid=1, 2, 3.
 - Registers a distinct filter function per vector.
 - Calls `bus_describe_intr` with the per-vector name.
 - Binds each vector to a CPU (round-robin or NUMA-aware).
2. MSI with MYFIRST_MAX_VECTORS vectors. Same per-vector pattern.
3. Legacy INTx with a single handler that covers all three event classes at rid=0.

Per-Vector Assignment

Vector	Purpose	Handles	Inline/Deferred
0	admin	INTR_STATUS.ERROR	Inline
1	rx	INTR_STATUS.DATA_AV	Deferred (task)
2	tx	INTR_STATUS.COMPLETE	Inline

On MSI-X, each vector has its own `intr_event`, its own CPU affinity (via `bus_bind_intr`), and its own `bus_describe_intr` label ("admin", "rx", "tx"). On MSI, the driver obtains a single vector and falls back to the Chapter 19 single-handler pattern because MSI requires a power-of-two vector count (`pci_alloc_msi` rejects count=3 with `EINVAL`). On legacy INTx, a single filter covers all three bits.

sysctls

- ``dev.myfirst.N.intr_mode``: 0 (legacy), 1 (MSI), 2 (MSI-X).
- ``dev.myfirst.N.vec{0,1,2}_fire_count``: per-vector fire counts.
- ``dev.myfirst.N.vec{0,1,2}_stray_count``: per-vector stray counts.
- ``dev.myfirst.N.intr_simulate_admin``, ``dev.myfirst.N.intr_simulate_rx``, ``dev.myfirst.N.intr_simulate_tx``: simulate per-vector interrupts.

Teardown Sequence

1. Disable interrupts at the device (clear `INTR_MASK`).
2. Per-vector in reverse: `bus_teardown_intr`, `bus_release_resource`.
3. Drain and free per-vector tasks and the taskqueue.
4. If `intr_mode` is MSI or MSI-X, call `pci_release_msi` once.

dmesg Summary Line

A single line on attach:

- "interrupt mode: MSI-X, 3 vectors"
- "interrupt mode: MSI, 1 vector (single-handler fallback)"
- "interrupt mode: legacy INTx (1 handler for all events)"

Known Limitations

- `MYFIRST_MAX_VECTORS` is hardcoded at 3. A dynamic design that adapts to the allocated count is a Chapter 20 challenge exercise.
- CPU binding is round-robin. NUMA-aware binding via `bus_get_cpus` is a challenge exercise.
- DMA is Chapter 21.
- iflib integration is out of scope.

See Also

- `INTERRUPTS.md` for the Chapter 19 legacy path details.
- `HARDWARE.md` for the register map.
- `LOCKING.md` for the full lock discipline.
- `PCI.md` for the PCI attach behaviour.

The document gives a future reader the full picture of the multi-vector design in one page.

20.9.6 The Regression Pass

The Chapter 20 regression is a superset of Chapter 19's:

1. Compile cleanly. `make` produces `myfirst.ko` without warnings.
2. Load. `kldload` shows the attach banner including the `interrupt mode:` line.
3. Verify mode. `sysctl dev.myfirst.0.intr_mode` returns 0, 1, or 2 (depending on guest).
4. Per-vector attach. `vmstat -i | grep myfirst` shows N lines (1 for legacy, 3 for MSI or MSI-X).
5. Per-vector description. Each entry has the correct name (`admin`, `rx`, `tx`, or `legacy`).
6. Simulated interrupts. Each vector's counter ticks independently.

7. Task runs. RX vector's simulated interrupt drives `intr_task_invocations`.
8. Clean detach. `devctl detach myfirst0` tears down all vectors.
9. Load after unload. A second `kldload` uses the same tier (tests that `pci_release_msi` worked).
10. `vmstat -m` shows no leaks. After unload, no `myfirst` allocations remain.

The regression script runs all ten checks. On QEMU with `virtio-rng-pci`, the test exercises the MSI-X path; on bhyve with `virtio-rnd`, it exercises the legacy-INTx fallback. The driver's fallback ladder ensures it works on either.

20.9.7 What the Refactor Accomplished

At the start of Chapter 20, `myfirst` at 1.2-`intr` had one interrupt handler on the legacy line. At the end of Chapter 20, `myfirst` at 1.3-`msi` has a three-tier fallback ladder (MSI-X → MSI → legacy), three per-vector filters on MSI or MSI-X, per-vector counters, per-vector CPU affinity, and a single clean teardown path. The driver's file count has grown by two; its documentation has grown by one; its functional capabilities have grown substantially.

The code is recognisably FreeBSD. A contributor opening the driver for the first time finds a familiar structure: a per-vector array, per-vector filter functions, a three-tier setup ladder, a `bus_describe_intr` for each vector, a single `pci_release_msi` at teardown. These patterns appear in every multi-queue FreeBSD driver.

20.9.8 Wrapping Up Section 8

The refactor follows the established shape: a new file for the new layer, a new header exporting the public interface, a new document explaining the behaviour, a version bump, a regression pass. Chapter 20's layer is multi-vector interrupt handling; Chapter 19's remains the single-vector legacy fallback. Together they form the full interrupt story the driver needs.

The instructional body of Chapter 20 is complete. Labs, challenges, troubleshooting, a wrap-up, and the bridge to Chapter 21 follow.

20.10 Reading a Real Driver Together: `virtio_pci.c`

Before the labs, a short walk through a real FreeBSD driver that uses MSI-X extensively. `/usr/src/sys/dev/virtio/pci/virtio_pci.c` is the shared core of both the legacy and modern virtio-PCI transports; it holds the interrupt-allocation ladder that every virtio device uses. Reading this file after Chapter 20 is a short exercise in pattern recognition; almost everything in the interrupt section maps to something Chapter 20 just taught.

20.10.1 The Allocation Ladder

`virtio_pci.c` has a helper called `vtpci_alloc_intr_resources` (the exact name varies slightly by FreeBSD version). Its structure is:

```

static int
vtpci_alloc_intr_resources(struct vtpci_common *cn)
{
    int error;

    /* Tier 0: MSI-X. */
    error = vtpci_alloc_msix(cn, nvectors);
    if (error == 0) {
        cn->vtpci_flags |= VTPCI_FLAG_MSIX;
        return (0);
    }

    /* Tier 1: MSI. */
    error = vtpci_alloc_msi(cn);
    if (error == 0) {
        cn->vtpci_flags |= VTPCI_FLAG_MSI;
        return (0);
    }

    /* Tier 2: legacy INTx. */
    return (vtpci_alloc_intx(cn));
}

```

The three tiers are exactly the Chapter 20 ladder. Each tier, on success, sets a flag on the common state and returns 0. On failure, the next tier is tried.

20.10.2 The MSI-X Allocation Helper

`vtpci_alloc_msix` queries the count, decides how many vectors to request (based on the number of virtqueues the device uses), and calls `pci_alloc_msix`:

```

static int
vtpci_alloc_msix(struct vtpci_common *cn, int nvectors)
{
    int error, count;

    if (pci_msix_count(cn->vtpci_dev) < nvectors)
        return (ENOSPC);

    count = nvectors;
    error = pci_alloc_msix(cn->vtpci_dev, &count);
    if (error != 0)
        return (error);
    if (count != nvectors) {

```

```

        pci_release_msi(cn->vtpci_dev);
        return (ENXIO);
    }
    return (0);
}

```

The pattern: check count, allocate, verify the allocation matched the request, release on mismatch. If the device advertises fewer vectors than wanted, ENOSPC is returned immediately. If `pci_alloc_msix` allocates a smaller count than requested, the code releases and returns ENXIO.

Chapter 20's code follows this exact logic (Section 5 showed the full version).

20.10.3 The Per-Vector Resource Allocation

Once MSI-X is allocated, `virtio` walks the vectors and registers a handler per vector:

```

static int
vtpci_register_msix_vectors(struct vtpci_common *cn)
{
    int i, rid, error;

    rid = 1; /* MSI-X vectors start at rid 1 */
    for (i = 0; i < cn->vtpci_num_vectors; i++) {
        cn->vtpci_vectors[i].res = bus_alloc_resource_any(
            cn->vtpci_dev, SYS_RES_IRQ, &rid, RF_ACTIVE);
        if (cn->vtpci_vectors[i].res == NULL)
            /* ... fail ... */;
        rid++;
        error = bus_setup_intr(cn->vtpci_dev,
            cn->vtpci_vectors[i].res,
            INTR_TYPE_MISC | INTR_MPSAFE,
            NULL, vtpci_vq_handler,
            &cn->vtpci_vectors[i], &cn->vtpci_vectors[i].cookie);
        if (error != 0)
            /* ... fail ... */;
    }
    return (0);
}

```

Two things match Chapter 20:

- `rid = 1` for the first vector, incrementing per vector.
- The filter (here `NULL`) and handler (`vtpci_vq_handler`) pattern. Note that `virtio` uses an `ithread`-only handler (filter=`NULL`), not a filter-plus-task pipeline. This is a simpler option that works for `virtio`'s per-vector work.

The `vtpci_vq_handler` function is the per-vector worker. Each vector gets its own argument (`&cn->vtpci_vectors[i]`), and the handler uses that argument to identify which virtqueue to service.

20.10.4 The Teardown

Virtio's teardown follows the Chapter 20 pattern:

```
static void
vtpci_release_intr_resources(struct vtpci_common *cn)
{
    int i;

    for (i = 0; i < cn->vtpci_num_vectors; i++) {
        if (cn->vtpci_vectors[i].cookie != NULL) {
            bus_teardown_intr(cn->vtpci_dev,
                             cn->vtpci_vectors[i].res,
                             cn->vtpci_vectors[i].cookie);
        }
        if (cn->vtpci_vectors[i].res != NULL) {
            bus_release_resource(cn->vtpci_dev, SYS_RES_IRQ,
                                rman_get_rid(cn->vtpci_vectors[i].res),
                                cn->vtpci_vectors[i].res);
        }
    }

    if (cn->vtpci_flags & (VTPCI_FLAG_MSI | VTPCI_FLAG_MSIX))
        pci_release_msi(cn->vtpci_dev);
}
```

Per-vector teardown (`bus_teardown_intr` + `bus_release_resource`), then a single `pci_release_msi` at the end. The order matches Chapter 20's `myfirst_msix_teardown`.

A detail worth noting: virtio uses `rman_get_rid` to recover the rid from the resource, rather than storing it separately. Chapter 20's driver stores the rid in the per-vector struct; both approaches are fine, but the storage approach is clearer and easier to debug.

20.10.5 What the Virtio Walkthrough Teaches

Three lessons transfer directly to Chapter 20's design:

1. **The three-tier fallback ladder is the standard pattern.** Every driver that wants to work on a range of hardware implements it the same way.
2. **Per-vector resource management uses incrementing rids starting at 1.** This is universal in FreeBSD's PCI infrastructure.

3. **pci_release_msi** is called once, regardless of the number of vectors. The per-vector teardown releases IRQ resources; the device-level release handles the MSI state.

A reader who can follow `vtpci_alloc_intr_resources` end to end has internalised the Chapter 20 vocabulary. For a richer example, `/usr/src/sys/dev/nvme/nvme_ctrlr.c` shows the same pattern at scale, with one admin vector plus up to NCPU I/O vectors.

20.11 A Deeper Look at Vector-to-CPU Placement

Section 5 introduced `bus_bind_intr(9)` briefly. This section goes a level deeper into why CPU placement matters, how real drivers choose CPUs, and what the trade-offs are.

20.11.1 The NUMA Picture

On a single-socket system, all CPUs share a single memory controller and a single cache hierarchy. Placement between CPUs matters only for cache affinity (the handler's code and data will be warm on whichever CPU last ran it). The performance difference between "CPU 0" and "CPU 3" is small.

On a multi-socket NUMA system, the picture changes. Each socket has its own memory controller, its own L3 cache, and its own PCIe root complex. A PCIe device attached to socket 0 sits on that socket's root complex; its registers are memory-mapped to an address range handled by socket 0's controller. An interrupt from that device fires; the handler reads `INTR_STATUS`; the read goes to the device's BAR, which is on socket 0; the CPU that runs the handler must be on socket 0, or the read crosses the inter-socket interconnect.

The inter-socket interconnect (on Intel systems: UPI or earlier QPI; on AMD: Infinity Fabric) is much slower than intra-socket cache access. A handler running on the wrong socket sees register reads that take tens of nanoseconds instead of ones; a receive queue whose data lives on the wrong socket sees every packet crossing the interconnect on the way to user space.

Well-placed vectors keep the handler's work on the socket the device lives on.

20.11.2 Querying NUMA Locality

FreeBSD exposes the NUMA topology to drivers through `bus_get_cpus(9)`. The API:

```
int bus_get_cpus(device_t dev, enum cpu_sets op, size_t setsize,
                 struct _cpuset *cpuset);
```

The `op` argument selects which set to query:

- `LOCAL_CPUS`: CPUs in the same NUMA domain as the device.
- `INTR_CPUS`: CPUs suitable for handling device interrupts (usually `LOCAL_CPUS` unless the operator has excluded some).

The `cpuset` is an output parameter; on success, it contains the bitmap of CPUs in the queried set.

Example use:


```

cpuset_t local_cpus;
int num_local;

if (bus_get_cpus(sc->dev, INTR_CPUS, sizeof(local_cpus),
    &local_cpus) == 0) {
    num_local = CPU_COUNT(&local_cpus);
    device_printf(sc->dev, "device has %d interrupt-suitable CPUs\n",
        num_local);
}

```

The driver uses `CPU_FFS(&local_cpus)` to find the first CPU in the set, `CPU_CLR(cpu, &local_cpus)` to mark it used, and iterates.

A round-robin bind that respects NUMA locality:

```

static void
myfirst_msix_bind_vectors_numa(struct myfirst_softc *sc)
{
    cpuset_t local_cpus;
    int cpu, i;

    if (bus_get_cpus(sc->dev, INTR_CPUS, sizeof(local_cpus),
        &local_cpus) != 0) {
        /* No NUMA info; round-robin across all CPUs. */
        myfirst_msix_bind_vectors_roundrobin(sc);
        return;
    }

    if (CPU_EMPTY(&local_cpus))
        return;

    for (i = 0; i < sc->num_vectors; i++) {
        if (CPU_EMPTY(&local_cpus))
            bus_get_cpus(sc->dev, INTR_CPUS,
                sizeof(local_cpus), &local_cpus);
        cpu = CPU_FFS(&local_cpus) - 1; /* FFS returns 1-based */
        CPU_CLR(cpu, &local_cpus);
        (void)bus_bind_intr(sc->dev,
            sc->vectors[i].irq_res, cpu);
    }
}

```

The code grabs the local CPU set, picks the lowest-numbered CPU, binds vector 0 to it, clears that CPU from the set, picks the next lowest, binds vector 1 to it, and so on. If the set is exhausted (more vectors than local CPUs), it refreshes and continues.

Chapter 20's driver does not include this NUMA-aware binding; a challenge exercise asks the reader to add it.

20.11.3 The Operator View

An operator can override the kernel's placement with `cpuset`:

```
# Get current placement for IRQ 257.
sudo cpuset -g -x 257

# Bind IRQ 257 to CPU 3.
sudo cpuset -l 3 -x 257

# Bind to a set of CPUs (kernel picks one when the interrupt fires).
sudo cpuset -l 2,3 -x 257
```

These commands override whatever the driver set with `bus_bind_intr`. An operator might do this to pin critical interrupts away from user-workload CPUs (for real-time applications) or to concentrate traffic on specific CPUs (for diagnostic purposes).

The driver's `bus_bind_intr` call sets the initial placement; the operator can override. A well-behaved driver sets a sensible default and respects operator changes (which it does automatically, because `bus_bind_intr` just writes to an OS-managed CPU-affinity state that the operator then modifies).

20.11.4 Measuring the Effect

A concrete way to see NUMA locality's value: run a high-interrupt-rate workload with the handler pinned to a local CPU, then to a remote CPU, and compare latencies. On a two-socket system, the remote CPU's handler typically takes 1.5x to 3x longer per interrupt, measured in CPU cycles.

FreeBSD's DTrace provider can measure this:

```
sudo dtrace -n '
fbt::myfirst_intr_filter:entry { self->ts = vtimestamp; }
fbt::myfirst_intr_filter:return /self->ts/ {
    @[cpu] = quantize(vtimestamp - self->ts);
    self->ts = 0;
}'
```

The output is a per-CPU histogram of filter latencies. A reader can run this while observing vector placements and confirm the latency difference.

20.11.5 When Vector Placement Matters

- High interrupt rates (more than a few thousand per second per vector).

- Large cache-line footprint in the handler (the handler's code and data occupy multiple cache lines).
- Shared receive paths with downstream processing on the same socket.
- NUMA systems with more than one socket and PCIe devices attached to specific sockets.

20.11.6 When Vector Placement Doesn't Matter

- Low-rate interrupts (dozens per second or fewer).
- Single-socket systems.
- Handlers that do minimal work (Chapter 20's admin vector).
- Drivers that run on a single CPU regardless (single-CPU embedded systems).

Chapter 20's driver is in the "does not really matter" category for normal testing, but the patterns the chapter teaches transfer directly to drivers where it does.

20.12 A Deeper Look at Vector Assignment Strategies

Section 6 showed the fixed-assignment pattern (vector 0 = admin, 1 = rx, 2 = tx). This section explores other assignment strategies real drivers use.

20.12.1 One-Vector-Per-Queue

The simplest strategy and the most common. Each queue (rx queue, tx queue, admin queue, etc.) has its own dedicated vector. The driver allocates $N+M+1$ vectors for N receive queues, M transmit queues, and 1 admin.

Pros: - Simple per-vector handler logic. - Each queue's interrupt rate is independent. - CPU affinity is per-queue (easy to pin to the NUMA-local CPU).

Cons: - Consumes many vectors for drivers with many queues. - Each queue's ithread adds overhead on low-rate queues.

This is the pattern `nvme(4)` uses.

20.12.2 Coalesced RX+TX Vector

Some drivers coalesce the RX and TX of a single queue-pair into a single vector. A NIC with 8 queue pairs would use 8 coalesced vectors plus a few for admin. When the vector fires, the filter checks both RX and TX status bits and dispatches accordingly.

Pros: - Half the vectors per queue-pair. - RX and TX for the same queue pair tend to be NUMA-local to each other (they share the same descriptor-ring memory).

Cons: - The filter is slightly more complex. - RX and TX can interfere under load (a burst of RX fills the handler's time, delaying TX completions).

This is a middle-ground design, used by some consumer NICs.

20.12.3 One Vector For All Queues

Some very-constrained devices (low-cost NICs, small embedded devices) have only one or two MSI-X vectors total. The driver uses a single vector for all queues and dispatches to each queue based on a status register.

Pros: - Works on hardware with few vectors. - Simple allocation.

Cons: - No per-queue affinity. - The filter does more work to decide what to dispatch.

This is the pattern a driver on very low-end hardware uses.

20.12.4 Dynamic Per-CPU Assignment

A clever design: allocate one vector per CPU, and assign queues to vectors dynamically. An RX queue is “owned” by one CPU at a time; it processes on that CPU’s vector. If the workload shifts, the driver can remap queues to different CPUs.

Pros: - Optimal per-CPU cache affinity. - Adapts to workload changes.

Cons: - Complex allocation and remapping logic. - Not easy to reason about.

Some high-end NIC drivers (Mellanox ConnectX series, Intel 800 Series) use variants of this.

20.12.5 Chapter 20’s Strategy

Chapter 20’s driver uses the fixed-assignment strategy with three vectors. It is the simplest strategy that illustrates multi-vector design without getting into NUMA details or dynamic remapping. Real drivers often start with this design and evolve to more sophisticated patterns as requirements demand.

A challenge exercise asks the reader to implement the dynamic per-CPU-allocation strategy as an extension.

20.13 A Deeper Look at Interrupt Moderation and Coalescing

A concept adjacent to MSI-X that deserves a brief mention. Modern high-throughput devices often support **interrupt moderation** or **coalescing**: the device buffers events (incoming packets, completions) and fires a single interrupt for multiple events, either at a time threshold or a count threshold.

20.13.1 Why Moderation Matters

A NIC receiving ten million packets per second would fire ten million interrupts if each packet triggered one. That is far too many; the CPU would spend all its time entering and exiting interrupt handlers. The solution is to batch: the NIC fires one interrupt every 50 microseconds, and during those 50 microseconds the NIC accumulates whatever packets arrived. The handler processes all the accumulated packets in one go.

Coalescing trades latency for throughput: each packet takes up to 50 microseconds longer to be delivered to user space, but the CPU handles millions of packets per second with a manageable interrupt rate.

20.13.2 How Drivers Control Moderation

The mechanism is device-specific. Common forms:

- **Time-based:** the device fires after a configured interval (e.g., 50 microseconds).
- **Count-based:** the device fires after N events (e.g., 16 packets).
- **Combined:** whichever threshold is reached first.
- **Adaptive:** the device (or the driver) tunes the thresholds based on observed rates.

The driver typically programs the thresholds through device registers. The MSI-X mechanism itself does not provide moderation; it is a device feature that works with MSI-X because MSI-X allows per-vector assignment.

20.13.3 Chapter 20's Driver Does Not Moderate

The Chapter 20 driver has no moderation. Each simulated interrupt produces one filter call. On real hardware this would be a problem at high rates; on the lab it is fine.

Real drivers like `em(4)`, `ix(4)`, `ixl(4)`, and `mgb(4)` all have moderation parameters. The `sysctl` interface exposes them as tunable values:

```
sysctl dev.em.0 | grep itr
```

A reader who adapts the chapter's driver to a real device should study the moderation controls for that device. The mechanism is orthogonal to MSI-X; the two combine to give high-performance interrupt handling.

20.14 Patterns From Real FreeBSD Drivers

A tour of the multi-vector patterns that appear in `/usr/src/sys/dev/`. Each pattern is a short snippet from a real driver, with a note on what it teaches for Chapter 20.

20.14.1 Pattern: `nvme(4)` Admin + I/O Vector Split

`/usr/src/sys/dev/nvme/nvme_ctrlr.c` has the canonical admin-plus-N pattern:

```
/* Allocate one vector for admin + N for I/O. */
num_trackers = MAX(1, MIN(mp_ncpus, ctrlr->max_io_queues));
num_vectors_requested = num_trackers + 1; /* +1 for admin */
num_vectors_allocated = num_vectors_requested;
pci_alloc_msix(ctrlr->dev, &num_vectors_allocated);

/* Admin queue uses vector 0 (rid 1). */
ctrlr->adminq.intr_rid = 1;
ctrlr->adminq.res = bus_alloc_resource_any(ctrlr->dev, SYS_RES_IRQ,
```

```

    &ctrlr->adminq.intr_rid, RF_ACTIVE);
bus_setup_intr(ctrlr->dev, ctrlr->adminq.res,
    INTR_TYPE_MISC | INTR_MPSAFE,
    NULL, nvme_qpair_msix_handler, &ctrlr->adminq, &ctrlr->adminq.tag);

/* I/O queues use vectors 1..N (rid 2..N+1). */
for (i = 0; i < ctrlr->num_io_queues; i++) {
    ctrlr->ioq[i].intr_rid = i + 2;
    /* same pattern ... */
}

```

Why it matters: the admin-plus-N pattern is the right choice when one vector handles infrequent, high-priority work (errors, async events) and N vectors handle rate-limited, per-queue work. Chapter 20's admin/rx/tx split is a miniature version of this.

20.14.2 Pattern: ixgbe's Queue-Pair Vector

/usr/src/sys/dev/ixgbe/ix_txx.c uses a queue-pair design where each vector handles both the RX and TX of one queue pair:

```

/* One vector per queue pair + 1 for link. */
for (i = 0; i < num_qpairs; i++) {
    que[i].rid = i + 1;
    /* Filter checks both RX and TX status bits and dispatches. */
    bus_setup_intr(..., ixgbe_msix_que, &que[i], ...);
}
/* Link-state vector is the last one. */
link.rid = num_qpairs + 1;
bus_setup_intr(..., ixgbe_msix_link, sc, ...);

```

Why it matters: the coalesced RX+TX-per-queue-pair design halves the vector count without sacrificing per-queue affinity. Suitable when the device has many queues but few vectors.

20.14.3 Pattern: virtio_pci's Per-Virtqueue Vector

/usr/src/sys/dev/virtio/pci/virtio_pci.c has one vector per virtqueue:

```

int nvectors = ... /* count of virtqueues + 1 for config */;
pci_alloc_msix(dev, &nvectors);
for (i = 0; i < nvectors; i++) {
    vec[i].rid = i + 1;
    /* Each vector gets the per-virtqueue data as its arg. */
    bus_setup_intr(dev, vec[i].res, ..., virtio_vq_intr, &vec[i], ...);
}

```

Why it matters: virtio's per-virtqueue assignment is the model for any paravirtualised device. The vector count equals the virtqueue count plus admin/config.

20.14.4 Pattern: ahci's Per-Port Vector

`/usr/src/sys/dev/ahci/ahci_pci.c` uses one vector per SATA port:

```
for (i = 0; i < ahci->nports; i++) {
    ahci->ports[i].rid = i + 1;
    /* ... */
}
```

Why it matters: storage controllers often use per-port vector assignments so that I/O completions on different ports can be processed concurrently on different CPUs.

20.14.5 Pattern: iflib's Hidden Vector Management

Drivers using `iflib(9)` (such as `em(4)`, `igc(4)`, `ix(4)`, `ixl(4)`, `mgb(4)`) do not manage vectors directly. Instead, they register per-queue handler functions with `iflib`'s registration table, and `iflib` does the allocation and binding:

```
static struct if_shared_ctx em_sctx_init = {
    /* ... */
    .isc_driver = &em_if_driver,
    .isc_tx_maxsize = EM_TS0_SIZE,
    /* ... */
};

static int
em_if_msix_intr_assign(if_ctx_t ctx, int msix)
{
    struct e1000_softc *sc = iflib_get_softc(ctx);
    int error, rid, i, vector = 0;

    /* iflib has already called pci_alloc_msix; sc knows the count. */
    for (i = 0; i < sc->rx_num_queues; i++, vector++) {
        rid = vector + 1;
        error = iflib_irq_alloc_generic(ctx, ..., rid, IFLIB_INTR_RXTX,
            em_msix_que, ...);
    }
    return (0);
}
```

Why it matters: iflib abstracts MSI-X allocation and per-queue binding behind a clean API. Drivers using iflib are simpler than bare MSI-X drivers but give up some flexibility. The iflib pattern is the right choice for new FreeBSD network drivers; the bare MSI-X pattern is the right choice for non-network devices or when iflib does not fit.

20.14.6 What the Patterns Teach

All of these drivers follow the same structural pattern Chapter 20 teaches:

1. Query vector count.
2. Allocate vectors.
3. For each vector: allocate IRQ resource at `rid=i+1`, register handler, describe.
4. Bind vectors to CPUs.
5. On teardown: per-vector teardown in reverse, then `pci_release_msi`.

The differences among drivers are:

- How many vectors (1, a handful, dozens, or hundreds).
- How vectors are assigned (admin+N, queue-pair, per-port, per-virtqueue).
- Whether iflib handles the allocation.
- What each filter function does (admin vs data-path).

A reader who has Chapter 20's vocabulary can recognise these differences immediately.

20.15 A Performance Observation: Measuring the MSI-X Benefit

A section that grounds the chapter's performance claims in a concrete measurement.

20.15.1 The Test Setup

Suppose you have the Chapter 20 driver running on QEMU with `virtio-rng-pci` (so MSI-X is active) and a multi-CPU guest. The `intr_simulate_rx` sysctl lets you trigger interrupts from a user-space loop:

```
# In one shell, drive simulated RX interrupts as fast as possible.
while true; do
    sudo sysctl dev.myfirst.0.intr_simulate_rx=1 >/dev/null 2>&1
done
```

20.15.2 Measuring With DTrace

In another shell, measure the filter's CPU-time per invocation and which CPU it runs on:


```
sudo dtrace -n '  
fbt::myfirst_rx_filter:entry { self->ts = vtimestamp; self->c = cpu; }  
fbt::myfirst_rx_filter:return /self->ts/ {  
    @lat[self->c] = quantize(vtimestamp - self->ts);  
    self->ts = 0;  
    self->c = 0;  
}'
```

The output is a per-CPU histogram of filter latencies. If `bus_bind_intr` placed the RX vector on CPU 1, the histogram should show all invocations on CPU 1, with latencies in the hundreds of nanoseconds to single-digit microseconds.

20.15.3 What the Results Show

On a well-placed MSI-X vector:

- Every invocation is on the same CPU (the bound CPU).
- Latencies are consistently short (the hot cache lines stay on one CPU).
- No cross-CPU cache bouncing.

On a legacy INTx shared line:

- Invocations spread across CPUs (the kernel routes randomly).
- Latencies are more variable (cold cache lines on each new CPU).
- Cross-CPU cache traffic appears in performance counters.

The difference can be measured in nanoseconds per invocation. For a driver handling a few hundred interrupts per second, the difference is invisible. For a driver handling a million interrupts per second, the difference is the difference between “works” and “does not work”.

20.15.4 The General Lesson

Chapter 20’s machinery is overkill for low-rate drivers. It is essential for high-rate drivers. The patterns the chapter teaches scale from “demo driver doing a hundred interrupts per second” to “production NIC doing ten million”. Knowing where on that scale a specific driver lives determines how much of Chapter 20’s advice matters in practice.

20.16 A Deeper Look at sysctl Tree Design for Multi-Vector Drivers

Chapter 20’s driver exposes its per-vector counters as flat sysctls (`vec0_fire_count`, `vec1_fire_count`, `vec2_fire_count`). For a driver with many vectors, a flat namespace becomes unwieldy. This section shows how to use `SYSCTL_ADD_NODE` to build a per-vector sysctl tree.

20.16.1 The Flat vs Tree Trade-off

Flat namespace (what Chapter 20 uses):

```
dev.myfirst.0.vec0_fire_count: 42
dev.myfirst.0.vec1_fire_count: 9876
dev.myfirst.0.vec2_fire_count: 4523
dev.myfirst.0.vec0_stray_count: 0
dev.myfirst.0.vec1_stray_count: 0
dev.myfirst.0.vec2_stray_count: 0
```

Pros: simple, no SYSCTL_ADD_NODE calls. Cons: many siblings at the top level; no grouping.

Tree namespace:

```
dev.myfirst.0.vec.admin.fire_count: 42
dev.myfirst.0.vec.admin.stray_count: 0
dev.myfirst.0.vec.rx.fire_count: 9876
dev.myfirst.0.vec.rx.stray_count: 0
dev.myfirst.0.vec.tx.fire_count: 4523
dev.myfirst.0.vec.tx.stray_count: 0
```

Pros: groups per-vector state; scales to many vectors; named rather than numbered. Cons: more code to set up.

20.16.2 The Tree-Building Code

```
void
myfirst_msix_add_sysctls(struct myfirst_softc *sc)
{
    struct sysctl_ctx_list *ctx = &sc->sysctl_ctx;
    struct sysctl_oid *parent = sc->sysctl_tree;
    struct sysctl_oid *vec_node;
    struct sysctl_oid *per_vec_node;
    int i;

    /* Create the "vec" parent node. */
    vec_node = SYSCTL_ADD_NODE(ctx, SYSCTL_CHILDREN(parent),
        OID_AUTO, "vec", CTLFLAG_RD, NULL,
        "Per-vector interrupt statistics");

    for (i = 0; i < MYFIRST_MAX_VECTORS; i++) {
        /* Create "vec.<name>" node. */
```

```

    per_vec_node = SYSCTL_ADD_NODE(ctx,
        SYSCTL_CHILDREN(vec_node),
        OID_AUTO, sc->vectors[i].name,
        CTLFLAG_RD, NULL,
        "Per-vector statistics");

    /* Add fire_count under it. */
    SYSCTL_ADD_U64(ctx, SYSCTL_CHILDREN(per_vec_node),
        OID_AUTO, "fire_count", CTLFLAG_RD,
        &sc->vectors[i].fire_count, 0,
        "Times this vector's filter was called");

    /* Add stray_count. */
    SYSCTL_ADD_U64(ctx, SYSCTL_CHILDREN(per_vec_node),
        OID_AUTO, "stray_count", CTLFLAG_RD,
        &sc->vectors[i].stray_count, 0,
        "Stray returns from this vector");

    /* Other per-vector fields... */
}
}

```

The `SYSCTL_ADD_NODE` calls create the intermediate nodes; subsequent `SYSCTL_ADD_U64` calls attach leaf counters under them. The tree structure becomes visible in `sysctl` output automatically.

20.16.3 Querying the Tree

```

# Show all per-vector stats.
sysctl dev.myfirst.0.vec

# Show just the rx vector.
sysctl dev.myfirst.0.vec.rx

# Show only fire counts.
sysctl -n dev.myfirst.0.vec.admin.fire_count dev.myfirst.0.vec.rx.fire_count
↪ dev.myfirst.0.vec.tx.fire_count

```

The tree structure makes the `sysctl` namespace much more readable, especially for drivers with many vectors (NVMe with 32 I/O queues, or a NIC with 16 queue pairs).

20.16.4 When to Use the Tree

For Chapter 20's three-vector driver, the flat namespace is fine. For a driver with eight or more vectors, the tree becomes valuable. A reader writing a production driver should use the tree.

20.16.5 Common Mistakes

- **Leaking the parent node.** `SYSCTL_ADD_NODE` registers the node in `sc->sysctl_ctx`; it is freed with the rest of the context. No explicit free needed.
- **Forgetting `NULL` for the handler argument.** `SYSCTL_ADD_NODE` is not a `CTLPROC`; it is a pure grouping node. The handler argument is `NULL`.
- **Wrong parent passed to child `SYSCTL_ADD_*` calls.** `SYSCTL_CHILDREN(vec_node)` for children of `vec_node`, not `SYSCTL_CHILDREN(parent)`.

This tree-design pattern is the cleanest way to expose multi-vector state. Chapter 20's challenge exercise suggests implementing it as an extension.

20.17 A Deeper Look at Error Paths in MSI-X Setup

Section 3 and Section 5 showed the happy-path setup code. This section walks through what can go wrong and how to diagnose it.

20.17.1 Failure Mode 1: `pci_msix_count` Returns 0

Symptom: the MSI-X attempt is skipped because the count is 0.

Cause: the device has no MSI-X capability, or the PCI bus driver has not discovered it.

Fix: Confirm with `pciconf -lvc`. If the device advertises MSI-X but `pci_msix_count` returns 0, the device's PCI configuration is broken or the kernel's probe did not find it; rare and hard to fix from the driver.

20.17.2 Failure Mode 2: `pci_alloc_msix` Returns `EINVAL`

Symptom: allocation fails with `EINVAL`.

Cause: the driver is asking for a count greater than the device's advertised max, or it is asking for 0.

Fix: Clamp the requested count to `pci_msix_count`'s returned value. Always request at least 1.

20.17.3 Failure Mode 3: `pci_alloc_msix` Returns Fewer Vectors Than Requested

Symptom: count after the call is less than requested.

Cause: the kernel's vector pool was partially depleted; the device's allocation was given whatever remained.

Fix: Decide upfront whether to accept, adapt, or release. Chapter 20's driver releases and falls back to MSI.

20.17.4 Failure Mode 4: bus_alloc_resource_any Returns NULL for an MSI-X Vector

Symptom: after `pci_alloc_msix` succeeded, the per-vector IRQ allocation fails.

Causes: - Wrong rid (using 0 instead of `i+1`). - Already released previously (double-release). - Out of IRQ resources at the bus layer.

Fix: Check the rid is `i+1`. Audit the release code. Log the error.

20.17.5 Failure Mode 5: bus_setup_intr Returns EINVAL for a Per-Vector Handler

Symptom: `bus_setup_intr` fails.

Causes: - Filter and `ithread` both NULL. - Missing `INTR_TYPE_*` flag. - Already set up previously (double-setup).

Fix: Ensure the filter argument is non-NULL. Include an `INTR_TYPE_*` flag. Audit the setup code for double-registration.

20.17.6 Failure Mode 6: bus_bind_intr Returns an Error

Symptom: `bus_bind_intr` returns non-zero.

Causes: - Platform does not support rebinding. - CPU out of range. - Kernel configuration (`NO_SMP`, `NUMA` disabled).

Fix: Treat as non-fatal (`device_printf` a warning and continue). The driver still works without binding.

20.17.7 Failure Mode 7: vmstat -i Shows Vectors but Counters Don't Increment

Symptom: the kernel sees the vectors but the filters never fire.

Causes: - The device's `INTR_MASK` is zero (chapter 19 problem). - The device reset its interrupt state. - Hardware bug or bhyve/QEMU configuration problem.

Fix: Verify the device's `INTR_MASK`. Use the simulated-interrupt `sysctl` to confirm the filter works at all.

20.17.8 Failure Mode 8: Second kldload Falls Back to Lower Tier

Symptom: first load uses MSI-X; unload; second load uses legacy or MSI.

Cause: `pci_release_msi` not called on teardown.

Fix: Audit the teardown path. Make sure `pci_release_msi` runs on every successful allocation path.

20.17.9 Failure Mode 9: WITNESS Panics on Multi-Vector Setup

Symptom: WITNESS reports a lock-order violation or a "lock held during sleep" in the per-vector setup.

Cause: holding `sc->mtx` across a `bus_setup_intr` call. The bus hooks may sleep, and holding a mutex across a sleep is illegal.

Fix: Release `sc->mtx` before calling `bus_setup_intr`. Reacquire afterwards if needed.

20.17.10 Failure Mode 10: Partial Setup Doesn't Clean Up Properly

Symptom: attach fails; second attach fails with “resource in use”.

Cause: the partial-failure goto cascade doesn't undo all the way. Some per-vector state lingers.

Fix: Ensure the cascade unwinds to the vector that failed, not past it. Use the per-vector helper consistently.

20.18 Additional Troubleshooting

A handful of extra failure modes Chapter 20 readers might hit.

20.18.1 “QEMU guest does not expose MSI-X”

Causes: QEMU version too old, or the guest is booting with legacy virtio.

Fix: Update QEMU to a recent version. In the guest, check:

```
pciconf -lvc | grep -B 1 -A 2 'cap 11'
```

If no `cap 11` lines appear, MSI-X is not available. Switch to QEMU's modern `virtio-rng-pci` with `-device virtio-rng-pci,disable-legacy=on`.

20.18.2 “intr_simulate_rx increments fire_count but task never runs”

Cause: the task's `TASK_INIT` was not called, or the taskqueue was not started.

Fix: Verify `TASK_INIT(&vec->task, 0, myfirst_rx_task_fn, vec)` in setup. Verify `taskqueue_start_threads(&sc
>intr_tq, ...)`.

20.18.3 “Per-vector counters increment but the stray count goes up proportionally”

Cause: the filter's status check is wrong, or multiple vectors are triggering on the same bit.

Fix: Each filter should check for its specific bit(s). If two filters both try to handle `DATA_AV`, one will win and the other will see stray.

20.18.4 “cpuset -g -x \$irq reports mask 0 on all vectors”

Cause: `bus_bind_intr` has not been called, or it was called with CPU 0 (mask 1).

Fix: If intentionally unbound, “mask 0” might be platform-specific. If binding was attempted, check return value of `bus_bind_intr`.

20.18.5 “Driver load succeeds but dmesg shows no attach banner”

Cause: the `device_printf` came before the banner flush, or the banner is in a very early boot buffer.

Fix: `dmesg -a` shows the full message buffer. Verify `dmesg -a | grep myfirst`.

20.18.6 “Detach hangs after multi-vector setup”

Cause: a vector’s handler is still running when teardown tries to proceed. `bus_teardown_intr` blocks waiting for it.

Fix: Make sure the device’s `INTR_MASK` is cleared *before* `bus_teardown_intr`, so no new handlers can be dispatched. Make sure the filter does not loop forever; short-runtime discipline.

20.18.7 “pci_alloc_msix succeeds but only some vectors fire”

Cause: the device is not actually signalling on the vectors it should. Could be a driver bug (forgot to enable) or a device quirk.

Fix: Use the simulated-interrupt `sysctl` to confirm the filter works for each vector. If the simulated path works but real events don’t fire the vector, the issue is on the device side.

20.19 Worked Example: Tracing an Event Through Three Tiers

To make the fallback ladder concrete, here is a complete trace of the same event (a simulated `DATA_AV` interrupt) on each of the three tiers.

20.19.1 Tier 3: Legacy INTx

On bhyve with `virtio-rnd` (no MSI-X exposed), the driver falls back to legacy INTx with one handler at `rid 0`.

1. User runs `sudo sysctl dev.myfirst.0.intr_simulate_admin=1` (or `intr_simulate_rx=1`, etc.).
2. The `sysctl` handler acquires `sc->mtx`, writes the bit to `INTR_STATUS`, releases, calls the filter.
3. The single `myfirst_intr_filter` (from Chapter 19) runs. It reads `INTR_STATUS`, sees the bit, acknowledges, and either enqueues the task (for `DATA_AV`) or handles inline (for `ERROR/COMPLETE`).
4. `intr_count` increments.
5. On legacy, there is only one vector, so all three simulated-interrupt `sysctls` go through the same filter.

Observations: - `sysctl dev.myfirst.0.intr_mode` returns 0. - `vmstat -i | grep myfirst` shows one line. - The per-vector counters do not exist (legacy mode uses the Chapter 19 counters).

20.19.2 Tier 2: MSI

On a system that supports MSI but not MSI-X, the driver allocates a single MSI vector. MSI requires a power-of-two vector count, so the driver cannot ask for 3 here; it requests 1 and uses the Chapter 19 single-handler pattern.

1. User runs `sudo sysctl dev.myfirst.0.intr_simulate_admin=1` (or `intr_simulate_rx=1`, or `intr_simulate_tx=4`).
2. Because only one vector is set up on the MSI tier, all three per-vector simulation sysctls route through the same Chapter 19 `myfirst_intr_filter`.
3. The filter reads `INTR_STATUS`, sees the bit, acknowledges, and either handles inline or enqueues the task.

Observations: - `sysctl dev.myfirst.0.intr_mode` returns 1. - `vmstat -i | grep myfirst` shows one line (the single MSI handler at `rid=1`, labelled “msi”). - The per-vector counters on slots 1 and 2 stay at 0 because only slot 0 is in use; the Chapter 19 global counters (`intr_count`, `intr_data_av_count`, etc.) are the ones that move.

20.19.3 Tier 1: MSI-X

On QEMU with `virtio-rng-pci`, the driver allocates MSI-X with 3 vectors, each bound to a CPU.

1. User runs `sudo sysctl dev.myfirst.0.intr_simulate_rx=1`.
2. The sysctl calls the rx filter directly (simulated path does not go through the hardware).
3. `myfirst_rx_filter` runs (on whichever CPU the sysctl was invoked on, because simulation is not going through the kernel’s interrupt dispatch).
4. Counters increment; task runs.

Observations: - `sysctl dev.myfirst.0.intr_mode` returns 2. - `vmstat -i | grep myfirst` shows three lines; each has a different IRQ number. - `cpuset -g -x <irq>` for each IRQ shows different CPU masks.

A real (non-simulated) MSI-X interrupt would dispatch on the bound CPU; the simulation bypass makes it run on the calling thread’s CPU. This is a limitation of the simulation technique but does not affect correctness.

20.19.4 The Lesson

All three tiers drive the same filter logic and the same task. The only differences are:

- Which rid the IRQ resource uses (0 for legacy, 1+ for MSI/MSI-X).
- Whether `pci_alloc_msi` or `pci_alloc_msix` succeeded.
- How many filter functions are registered (1 for legacy, 3 for MSI/MSI-X).
- Which CPU real interrupts dispatch on.

A well-written driver works identically on all three tiers. Chapter 20’s fallback ladder ensures this.

20.20 A Practical Lab: Regression Testing Across All Three Tiers

A lab that exercises the fallback ladder to confirm all three tiers work.

20.20.1 Setup

You need two test environments:

- **Environment A:** bhyve with virtio-rnd. The driver falls back to legacy INTx.
- **Environment B:** QEMU with virtio-rng-pci. The driver uses MSI-X.

(A third environment with only MSI and no MSI-X is hard to construct reliably on modern platforms. The MSI path is exercised only if the reader has a system where MSI-X fails but MSI works.)

20.20.2 Procedure

1. On Environment A, load `myfirst.ko`. Verify:

```
sysctl dev.myfirst.0.intr_mode    # returns 0
vmstat -i | grep myfirst         # one line
```

2. Exercise the pipeline via the simulated-interrupt sysctls. All three should work, though on legacy they all go through the same filter.

```
sudo sysctl dev.myfirst.0.intr_simulate_admin=2
sudo sysctl dev.myfirst.0.intr_simulate_rx=1
sudo sysctl dev.myfirst.0.intr_simulate_tx=4
sysctl dev.myfirst.0.intr_count    # should be 3
```

3. Unload. Verify no leaks.
4. On Environment B, repeat:

```
sysctl dev.myfirst.0.intr_mode    # returns 2
vmstat -i | grep myfirst         # three lines
for irq in <IRQs>; do cpuset -g -x $irq; done
```

5. Exercise the per-vector pipeline. Each sysctl should increment its own vector's counter.

```
sudo sysctl dev.myfirst.0.intr_simulate_admin=2
sysctl dev.myfirst.0.vec.admin.fire_count    # 1
sysctl dev.myfirst.0.vec.rx.fire_count       # 0
sysctl dev.myfirst.0.vec.tx.fire_count       # 0
```

6. Unload. Verify no leaks.

20.20.3 Expected Observations

- Both environments attach cleanly.
- The `dmesg` summary line shows the correct mode for each.
- Per-vector counters tick independently on MSI-X.
- On legacy, a single counter covers all events.
- No leaks after unload in either environment.

20.20.4 What to Do if a Tier Fails

If the MSI-X tier fails on Environment B:

1. Verify QEMU is new enough. Older versions (pre-5.0) have quirks.
2. Check `pciconf -lvc` in the guest; MSI-X capability should be visible.
3. Check `dmesg` for errors from `pci_alloc_msix`.

If the legacy tier fails on Environment A:

1. Check `pciconf -lvc` for the device's interrupt line configuration.
2. Ensure `virtio_rnd` is not already attached (Chapter 18 caveat).
3. Look for `pci_alloc_resource` failures in `dmesg`.

20.21 Extended Challenge: Building a Production-Quality Driver

An optional exercise for readers who want to practise multi-vector design on a realistic scale.

20.21.1 The Goal

Take the Chapter 20 driver and extend it to handle *N* queues dynamically, where *N* is discovered at attach time based on the allocated MSI-X vector count. Each queue has:

- Its own vector (MSI-X vector 1+queue_id).
- Its own filter function (or a shared one that identifies the queue from the vector arg).
- Its own counters.
- Its own task on its own taskqueue.
- Its own NUMA-local CPU binding.

20.21.2 Implementation Outline

1. Replace `MYFIRST_MAX_VECTORS` with a runtime-chosen count.
2. Allocate the `vectors[]` array dynamically (using `malloc`).
3. Allocate a separate taskqueue per vector.
4. Use `bus_get_cpus(INTR_CPUS, ...)` to distribute vectors across NUMA-local CPUs.
5. Add sysctls that scale with the vector count.

20.21.3 Testing

Run the driver on a guest with varying MSI-X vector counts. For each count, verify: - The fire counters tick for the simulated interrupts. - The CPU affinity respects NUMA locality. - Teardown is clean.

20.21.4 What This Exercises

- Dynamic memory management in a driver.
- The `bus_get_cpus` API.
- Per-queue taskqueues (challenge 3 from earlier).
- Runtime sysctl tree construction (challenge 7 from earlier).

This is a significant exercise and will likely take several hours. The result is a driver recognisably similar to production NIC and NVMe drivers.

20.22 Reference: Priority Values for Interrupt and Task Work

For quick reference, the priority constants a Chapter 20 driver might use (from `/usr/src/sys/sys/priority.h`):

```
PI_REALTIME = PRI_MIN_ITHD + 0    (highest; rarely used)
PI_INTR      = PRI_MIN_ITHD + 4    (common hardware interrupt level)
PI_AV        = PI_INTR             (audio/video)
PI_NET       = PI_INTR             (network)
PI_DISK      = PI_INTR             (block storage)
PI_TTY       = PI_INTR             (terminal/serial)
PI_DULL      = PI_INTR             (low-priority hardware)
PI_SOFT      = PRI_MIN_ITHD + 8    (soft interrupts)
```

The common hardware priorities all map to `PI_INTR`; the names are distinctions of intent rather than of scheduling priority. Chapter 20's driver uses `PI_NET` for its taskqueue; any hardware-level priority would work equivalently.

20.23 Reference: Useful DTrace One-Liners for MSI-X Drivers

For readers who want to observe the Chapter 20 driver's behaviour dynamically.

20.23.1 Count filter invocations per CPU

```
sudo dtrace -n '
fbt::myfirst_admin_filter:entry, fbt::myfirst_rx_filter:entry,
fbt::myfirst_tx_filter:entry { @[probefunc, cpu] = count(); }'
```

Shows which filter runs on which CPU.

20.23.2 Time spent in each filter

```
sudo dtrace -n '  
fbt::myfirst_rx_filter:entry { self->ts = vtimestamp; }  
fbt::myfirst_rx_filter:return /self->ts/ {  
    @[probefunc] = quantize(vtimestamp - self->ts);  
    self->ts = 0;  
}'
```

Histogram of RX filter CPU time.

20.23.3 Rate of simulated vs real interrupts

```
sudo dtrace -n '  
fbt::myfirst_intr_simulate_vector_sysctl:entry { @sims = count(); }  
fbt::myfirst_rx_filter:entry { @filters = count(); }'
```

If `filters > sims`, some real interrupts are firing.

20.23.4 Task latency

```
sudo dtrace -n '  
fbt::myfirst_rx_filter:entry { self->ts = vtimestamp; }  
fbt::myfirst_rx_task_fn:entry /self->ts/ {  
    @lat = quantize(vtimestamp - self->ts);  
    self->ts = 0;  
}'
```

Histogram of time from filter to task invocation. Shows the taskqueue's scheduling latency.

20.24 Reference: A Closing Note Before Part 4's End

Chapters 16 through 20 built the full interrupt and hardware story for the `myfirst` driver. Each chapter added one layer:

- Chapter 16: register access.
- Chapter 17: device behaviour simulation.
- Chapter 18: PCI attach.
- Chapter 19: single-vector interrupt handling.
- Chapter 20: multi-vector MSI/MSI-X.

Chapter 21 will add DMA, completing Part 4's hardware layer. At that point, the `myfirst` driver will be structurally a real driver: a PCI device with MSI-X interrupts and DMA-based data transfer. What distinguishes it from a production driver is the specific protocol it speaks (none, really; it is a demo) and the device it targets (a `virtio-rnd` abstraction).

A reader who has internalised these five chapters can open any FreeBSD driver in `/usr/src/sys/dev/` and recognise the patterns. That recognition is Part 4's deepest payoff.

20.25 Hands-On Labs

The labs are graduated checkpoints. Each lab builds on the previous one and corresponds to one of the chapter's stages. A reader who works through all five has a complete multi-vector driver, a working QEMU test environment for MSI-X, and a regression script that validates all three tiers of the fallback ladder.

Time budgets assume the reader has already read the relevant sections.

20.25.1 Lab 1: Discover MSI and MSI-X Capabilities

Time: thirty minutes.

Objective: Build an intuition for which devices on your system support MSI and MSI-X.

Steps:

1. Run `sudo pciconf -lvc > /tmp/pci_caps.txt`. The `-c` flag includes capability lists.
2. Search for MSI capabilities: `grep -B 1 "cap 05" /tmp/pci_caps.txt`.
3. Search for MSI-X capabilities: `grep -B 1 "cap 11" /tmp/pci_caps.txt`.
4. For three devices that support MSI-X, note:
 - The device's name (`pci0:B:D:F`).
 - The number of MSI-X messages supported.
 - Whether the driver is currently using MSI-X (check `vmstat -i` for multiple lines of the same device name).
5. Compare the total number of MSI-capable devices to the total number of MSI-X-capable devices. Modern systems typically have more MSI-X devices than MSI-only devices.

Expected observations:

- NICs usually advertise MSI-X with many vectors (4 to 64).
- SATA and NVMe controllers advertise MSI-X (NVMe often with dozens of vectors).
- Some legacy devices (an audio chip, a USB controller) advertise only MSI.
- A few very old devices advertise neither and rely on legacy INTx.

This lab is about vocabulary. No code. The payoff is that Section 2 and 5's allocation calls become concrete.

20.25.2 Lab 2: Stage 1, MSI Fallback Ladder

Time: two to three hours.

Objective: Extend Chapter 19's driver with the MSI-first fallback ladder. Version target: 1.3-msi-stage1.

Steps:

1. Starting from Chapter 19 Stage 4, copy the driver source to a new working directory.
2. Add the `intr_mode` field and enum to `myfirst.h`.
3. Modify `myfirst_intr_setup` (in `myfirst_intr.c`) to attempt MSI allocation first, falling back to legacy INTx.
4. Modify `myfirst_intr_teardown` to call `pci_release_msi` when MSI was used.
5. Add the `dev.myfirst.N.intr_mode` sysctl.
6. Update the Makefile version string to 1.3-msi-stage1.
7. Compile (`make clean && make`).
8. Load on a guest. Note which mode the driver reports:

```
sudo kldload ./myfirst.ko
sudo dmesg | tail -5
sysctl dev.myfirst.0.intr_mode
```

On QEMU with `virtio-rng-pci`, the driver should report MSI, 1 vector (or similar). On bhyve with `virtio-rnd`, it should report legacy INTx.

9. Unload and verify no leaks.

Common failures:

- Missing `pci_release_msi`: next load fails or falls back to legacy.
- Wrong rid (using 0 for MSI): `bus_alloc_resource_any` returns NULL.
- Not checking the returned count: driver proceeds with fewer vectors than expected.

20.25.3 Lab 3: Stage 2, Multi-Vector Allocation (MSI)

Time: three to four hours.

Objective: Extend to three MSI vectors with per-vector handlers. Version target: 1.3-msi-stage2.

Steps:

1. Starting from Lab 2, add the `myfirst_vector` struct and per-vector array to `myfirst.h`.
2. Write three filter functions: `myfirst_admin_filter`, `myfirst_rx_filter`, `myfirst_tx_filter`.
3. Write the `myfirst_intr_setup_vector` and `myfirst_intr_teardown_vector` helpers.
4. Modify `myfirst_intr_setup` to try `pci_alloc_msi` for `MYFIRST_MAX_VECTORS` vectors, setting up each vector independently.
5. Modify `myfirst_intr_teardown` to loop per-vector.
6. Add per-vector counter sysctls (`vec0_fire_count`, `vec1_fire_count`, `vec2_fire_count`).
7. Add per-vector simulated-interrupt sysctls (`intr_simulate_admin`, `intr_simulate_rx`, `intr_simulate_tx`).
8. Bump the version to 1.3-msi-stage2.
9. Compile, load, verify:

```
sysctl dev.myfirst.0.intr_mode    # should be 1 on QEMU
vmstat -i | grep myfirst         # should show 3 lines
```

10. Exercise each vector:

```
sudo sysctl dev.myfirst.0.intr_simulate_admin=2 # ERROR
sudo sysctl dev.myfirst.0.intr_simulate_rx=1    # DATA_AV
sudo sysctl dev.myfirst.0.intr_simulate_tx=4    # COMPLETE
sysctl dev.myfirst.0 | grep vec
```

Each vector's counter should increment independently.

11. Unload, verify no leaks.

20.25.4 Lab 4: Stage 3, MSI-X With CPU Binding

Time: three to four hours.

Objective: Prefer MSI-X over MSI, bind each vector to a CPU. Version target: 1.3-msi-stage3.

Steps:

1. Starting from Lab 3, change the fallback ladder to attempt MSI-X first (via `pci_msix_count` and `pci_alloc_msix`), MSI as second tier, legacy as last.
2. Add the `myfirst_msix_bind_vectors` helper that calls `bus_bind_intr` for each vector.
3. Call the bind helper after all vectors are registered.
4. Update the `dmesg` summary line to distinguish MSI-X from MSI.
5. Bump the version to 1.3-msi-stage3.
6. Compile, load on QEMU with `virtio-rng-pci`. Verify:

```
sysctl dev.myfirst.0.intr_mode    # should be 2 on QEMU
sudo dmesg | grep myfirst | grep MSI-X
```

The attach line should read `interrupt mode: MSI-X, 3 vectors`.

7. Check per-vector CPU bindings:

```
# For each myfirst IRQ, show its CPU binding.
vmstat -i | grep myfirst
# (Note the IRQ numbers, then:)
for irq in <IRQ1> <IRQ2> <IRQ3>; do
    echo "IRQ $irq:"
    cpuset -g -x $irq
done
```

On a multi-CPU guest, each vector should be bound to a different CPU.

8. Exercise each vector (same as Lab 3).
9. Detach and reattach:

```
sudo devctl detach myfirst0
sudo devctl attach pci0:0:4:0
sysctl dev.myfirst.0.intr_mode # should still be 2
```

10. Unload, verify no leaks.

20.25.5 Lab 5: Stage 4, Refactor, Regression, Version

Time: three to four hours.

Objective: Move the multi-vector code into `myfirst_msix.c`, write `MSIX.md`, run the regression. Version target: 1.3-msi.

Steps:

1. Starting from Lab 4, create `myfirst_msix.c` and `myfirst_msix.h`.
2. Move the per-vector filter functions, helpers, setup, teardown, and sysctl registration into `myfirst_msix.c`.
3. Keep the legacy-INTx fallback in `myfirst_intr.c` (Chapter 19's file).
4. In `myfirst_pci.c`, replace the old interrupt setup/teardown calls with calls into `myfirst_msix.c`.
5. Update the Makefile to add `myfirst_msix.c` to SRCS. Bump the version to 1.3-msi.
6. Write `MSIX.md` documenting the multi-vector design.
7. Compile, load, run the full regression script (from the companion examples).
8. Confirm all three tiers work (by testing on bhyve with `virtio-rnd` for legacy and QEMU with `virtio-rng-pci` for MSI-X).

Expected outcomes:

- The driver at 1.3-msi works on both bhyve (legacy fallback) and QEMU (MSI-X).
- `myfirst_intr.c` now only contains the Chapter 19 single-handler fallback path.
- `myfirst_msix.c` contains the Chapter 20 multi-vector logic.
- `MSIX.md` documents the design clearly.

20.26 Challenge Exercises

The challenges build on the labs and extend the driver in directions the chapter did not take.

20.26.1 Challenge 1: Dynamic Vector-Count Adaptation

Modify the setup to adapt to whatever vector count the kernel actually allocates. If 3 are requested but 2 are allocated, the driver should still work with 2 (fold admin and tx into one combined vector). If 1 is allocated, fold everything into one.

This exercise teaches the “adapt” strategy from the fallback ladder.

20.26.2 Challenge 2: NUMA-Aware CPU Binding

Replace the round-robin CPU binding with a NUMA-aware binding using `bus_get_cpus(dev, INTR_CPUS, ...)`. Verify with `cpuset -g -x <irq>` that vectors land on CPUs in the same NUMA domain as the device.

On a single-socket system the exercise is academic; on a multi-socket test host it is measurable.

20.26.3 Challenge 3: Per-Vector Taskqueues

Each vector currently shares one taskqueue. Modify the driver so each vector has its own taskqueue (with its own worker thread). Measure the latency impact with DTrace.

This exercise introduces per-vector workers and shows when they help vs hurt.

20.26.4 Challenge 4: Per-Vector MSI-X Mask Control

The MSI-X table's vector-control register has a mask bit per vector. Add a sysctl that lets the operator mask an individual vector at runtime. Verify that a masked vector stops receiving interrupts.

Hint: the mask bit is programmed through direct MSI-X table access, which is a deeper topic than Chapter 20 covers. The FreeBSD MSI-X implementation may or may not expose this directly; a reader might need to use `bus_tearardown_intr` and later `bus_setup_intr` as a higher-level "soft mask".

20.26.5 Challenge 5: Implement Interrupt Moderation

For a simulated driver, moderation is easy to prototype: a sysctl that coalesces N simulated interrupts into one task run. Implement the coalescing, measure the latency-vs-throughput trade-off.

20.26.6 Challenge 6: Vector Reassignment at Runtime

Add a sysctl that lets the operator reassign which vector handles which event class (e.g., swap RX and TX). Demonstrate that after the reassignment, simulated-interrupt-RX triggers the TX filter and vice versa.

20.26.7 Challenge 7: Per-Queue Sysctl Tree

Restructure the per-vector sysctls into a proper tree: `dev.myfirst.N.vec.admin.fire_count`, `dev.myfirst.N.vec.rx.f` etc. Use `SYSCTL_ADD_NODE` to create the tree nodes.

20.26.8 Challenge 8: Dtrace Instrumentation

Write a DTrace script that shows the per-CPU distribution of each vector's filter invocations. Plot the per-CPU breakdown as a histogram. This is the diagnostic that confirms CPU binding is working.

20.27 Troubleshooting and Common Mistakes

20.27.1 “pci_alloc_msix returns EBUSY or ENXIO”

Possible causes:

1. The device is not connected in a way that supports MSI-X (legacy virtio-rnd on bhyve, for example). Check `pciconf -lvc`.
2. A previous load of the driver did not call `pci_release_msi` at teardown. Reboot or try `kldunload + kldload` again.
3. The kernel ran out of interrupt vectors. Rare on modern x86, possible on low-vector platforms.

20.27.2 “vmstat -i shows only one line on MSI-X guest”

Likely cause: `pci_alloc_msix` succeeded but allocated only 1 vector. Check the returned count vs requested. Either accept (fold work into one) or release and fall back.

20.27.3 “Filter fires but vec->fire_count stays at zero”

Likely cause: the `sc` argument is confused with `vec`. The handler receives `vec`, not `sc`. Check `bus_setup_intr`’s argument.

20.27.4 “Driver panics on kldunload after multiple load/unload cycles”

Likely cause: `pci_release_msi` not called on teardown. The device-level MSI state leaks across loads; eventually the kernel’s internal bookkeeping is confused.

20.27.5 “Different vectors all fire on the same CPU”

Likely cause: `bus_bind_intr` failed silently. Check the return value and log non-zero results.

20.27.6 “MSI-X allocation succeeds but vmstat -i shows no events”

Likely cause: the device’s `INTR_MASK` write targeted the wrong register or was skipped. Verify the mask is set (Chapter 17/Chapter 19 diagnostic).

20.27.7 “Stray interrupts accumulate on the MSI-X admin vector”

Likely cause: the admin filter’s status check is wrong; the filter returns `FILTER_STRAY` when it should handle. Check the `status & MYFIRST_INTR_ERROR` check.

20.27.8 “Shared-IRQ behaviour on legacy fallback differs from MSI-X”

Expected. On legacy `INTx` the single handler sees every event bit; on MSI-X each vector sees only its own event. Tests that exercise per-vector stray counts differ between the two modes.

20.27.9 “Stage 2 compiles but Stage 3 fails at bus_get_cpus link error”

Cause: `bus_get_cpus` may not be available in older FreeBSD versions or may require specific `#include <sys/bus.h>` placement. Check the include order.

20.27.10 “QEMU guest does not expose MSI-X despite using virtio-rng-pci”

Likely cause: older QEMU versions use legacy virtio by default. Check `pciconf -lvc` in the guest; if MSI-X is not listed, the guest is using legacy. Update QEMU or use `-device virtio-rng-pci,disable-modern=off,disable-legacy=on`.

20.28 Wrapping Up

Chapter 20 gave the driver the ability to handle multiple interrupt vectors. The starting point was `1.2-intr` with one handler on the legacy INTx line. The ending point is `1.3-msi` with a three-tier fallback ladder (MSI-X, MSI, legacy), three per-vector filter handlers, per-vector counters and tasks, per-vector CPU binding, a clean multi-vector teardown, and a new `myfirst_msix.c` file plus `MSIX.md` document.

The eight sections walked the full progression. Section 1 introduced MSI and MSI-X at the hardware level. Section 2 added MSI as a single-vector alternative to legacy INTx. Section 3 extended to multi-vector MSI. Section 4 examined the concurrency implications of multiple filters on multiple CPUs. Section 5 moved to MSI-X with per-vector CPU binding. Section 6 codified the per-vector event roles. Section 7 consolidated the teardown. Section 8 refactored into the final layout.

What Chapter 20 did not do is DMA. Each vector’s handler still only touches registers; the device does not yet have the ability to reach into RAM. Chapter 21 is where that changes. DMA introduces new complications (coherence, scatter-gather, mapping) that interact with interrupts (completion interrupts signal that a DMA transfer is done). The Chapter 20 interrupt machinery is ready to handle completion interrupts; Chapter 21 writes the DMA side.

The file layout has grown: 14 source files (including `cbuf`), 6 documentation files (`HARDWARE.md`, `LOCKING.md`, `SIMULATION.md`, `PCI.md`, `INTERRUPTS.md`, `MSIX.md`), and a growing regression test suite. The driver is structurally parallel to production FreeBSD drivers at this point.

20.28.1 A Reflection Before Chapter 21

Chapter 20 was the last chapter in Part 4 that is purely about interrupts. Chapter 21 moves to DMA, which is about moving data. The two are complementary: interrupts signal events; DMA moves the data those events are about. A high-performance driver uses both together: receive descriptors are populated by DMA from the device into RAM, then a completion interrupt signals the driver to process the descriptors.

Chapter 20’s per-vector handlers are already the right shape for this. Each receive queue’s completion interrupt fires its own vector; each vector’s filter acknowledges and defers; the task walks the receive ring (populated by DMA, Chapter 21) and passes packets up. Chapter 21 writes the DMA side; Chapter 20’s interrupt side is already in place.

The chapter's teaching also generalises. A reader who has internalised Chapter 20's three-tier fallback ladder, per-vector state design, CPU binding, and clean teardown will find similar patterns in every multi-queue FreeBSD driver. The specific device differs; the structure does not.

20.28.2 What to Do If You Are Stuck

Three suggestions.

First, read `/usr/src/sys/dev/virtio/pci/virtio_pci.c` carefully, focusing on the `vtpci_alloc_intr_resources` family of functions. The pattern matches Chapter 20 exactly and the code is compact enough to read in one sitting.

Second, run the chapter's regression test on both a bhyve guest (legacy fallback) and a QEMU guest (MSI-X). Seeing the same driver behave correctly on both targets confirms the fallback ladder is right.

Third, skip the challenges on first pass. The labs are calibrated for Chapter 20's pace; the challenges assume the material is solid. Come back to them after Chapter 21 if they feel out of reach now.

Chapter 20's goal was to give the driver a multi-vector interrupt path. If it has, Chapter 21's DMA work becomes a complement rather than an entirely new topic.

20.29 Bridge to Chapter 21

Chapter 21 is titled *DMA and High-Speed Data Transfer*. Its scope is the topic Chapter 20 deliberately did not take: the device's ability to read and write RAM directly, without the driver in the loop for each word. A NIC with a 64-entry receive descriptor ring populates those entries by DMA from the wire; a single interrupt signals "N entries are ready". The driver's handler walks the ring and processes the entries. Without DMA the driver would have to read each byte from a device register, which does not scale.

Chapter 20 prepared the ground in three specific ways.

First, **you have per-vector completion interrupts**. Each queue's receive completions and transmit completions can fire a dedicated vector. Chapter 21's DMA ring work plugs into Chapter 20's per-vector filter and task; the filter sees "completions N through M are ready", the task processes them.

Second, **you have per-CPU handler placement**. A DMA ring's memory is on a specific NUMA node; the handler that processes it should run on a CPU on that node. Chapter 20's `bus_bind_intr` work is the mechanism. Chapter 21 extends this: the DMA memory is allocated with NUMA awareness too, so the ring, the handler, and the processing all end up on the same node.

Third, **you have the teardown discipline**. DMA adds more resources (DMA tags, DMA maps, DMA memory regions), and each needs its own teardown step. The Chapter 19/Chapter 20 per-vector teardown pattern extends naturally to per-queue DMA cleanup.

Specific topics Chapter 21 will cover:

- What DMA is, the difference between memory-mapped I/O and DMA.
- `bus_dma(9)`: tags, maps, and the DMA state machine.
- `bus_dma_tag_create` to describe DMA requirements (alignment, boundaries, address range).
- `bus_dmamap_create` and `bus_dmamap_load` to set up DMA transfers.

- Synchronisation: `bus_dmamap_sync` around DMA.
- Bounce buffers: what they are and when they are used.
- Cache coherence: why CPUs and devices see different memory at different times.
- Scatter-gather lists: physical addresses that are not contiguous.
- Ring buffers: the producer-consumer descriptor ring pattern.

You do not need to read ahead. Chapter 20 is sufficient preparation. Bring your `myfirst` driver at `1.3-msi`, your `LOCKING.md`, your `INTERRUPTS.md`, your `MSIX.md`, your WITNESS-enabled kernel, and your regression script. Chapter 21 starts where Chapter 20 ended.

The hardware conversation is deepening. The vocabulary is yours; the structure is yours; the discipline is yours. Chapter 21 adds the next missing piece: the device's ability to move data without asking.

20.30 Reference: Chapter 20 Quick-Reference Card

A compact summary of the vocabulary, APIs, macros, and procedures Chapter 20 introduced.

20.30.1 Vocabulary

- **MSI (Message Signalled Interrupts)**: PCI 2.2 mechanism. 1 to 32 vectors, contiguous, single address.
- **MSI-X**: PCIe mechanism. Up to 2048 vectors, per-vector address, per-vector mask, table in a BAR.
- **vector**: a single interrupt source identified by an index.
- **rid**: the resource ID used with `bus_alloc_resource_any`. 0 for legacy INTx, 1+ for MSI and MSI-X.
- **intr_mode**: the driver's record of which tier it is using (legacy, MSI, or MSI-X).
- **fallback ladder**: try MSI-X first, then MSI, then legacy INTx.
- **per-vector state**: counters, filter, task, cookie, resource per vector.
- **CPU binding**: routing a vector to a specific CPU via `bus_bind_intr`.
- **LOCAL_CPUS / INTR_CPUS**: CPU-set queries for NUMA-aware placement.

20.30.2 Essential APIs

- `pci_msi_count(dev)`: query MSI vector count.
- `pci_msix_count(dev)`: query MSI-X vector count.
- `pci_alloc_msi(dev, &count)`: allocate MSI vectors.
- `pci_alloc_msix(dev, &count)`: allocate MSI-X vectors.
- `pci_release_msi(dev)`: release MSI or MSI-X vectors.
- `pci_msix_table_bar(dev)`, `pci_msix_pba_bar(dev)`: identify table/PBA BARs.
- `bus_alloc_resource_any(dev, SYS_RES_IRQ, &rid, RF_ACTIVE)`: allocate per-vector IRQ resource.
- `bus_setup_intr(dev, res, flags, filter, ihand, arg, &cookie)`: register per-vector handler.
- `bus_teardown_intr(dev, res, cookie)`: unregister per-vector handler.
- `bus_describe_intr(dev, res, cookie, "name")`: label per-vector handler.
- `bus_bind_intr(dev, res, cpu)`: bind vector to a specific CPU.
- `bus_get_cpus(dev, op, size, &set)`: query NUMA-local CPUs (`op` = `LOCAL_CPUS` or `INTR_CPUS`).

20.30.3 Essential Macros

- `PCIY_MSI` = 0x05: MSI capability ID.
- `PCIY_MSIX` = 0x11: MSI-X capability ID.
- `PCIM_MSIXCTRL_TABLE_SIZE` = 0x07FF: mask for vector count.
- `PCI_MSIX_MSGNUM(ctrl)`: macro to extract vector count from control register.
- `MYFIRST_MAX_VECTORS`: driver-defined constant (3 in Chapter 20).

20.30.4 Common Procedures

Implement the three-tier fallback ladder:

1. `pci_msix_count(dev)`; if > 0, try `pci_alloc_msix`.
2. On failure, `pci_msi_count(dev)`; if > 0, try `pci_alloc_msi`.
3. On failure, fall back to legacy INTx with `rid = 0` and `RF_SHAREABLE`.

Register per-vector handlers (MSI-X):

1. Loop from `i = 0` to `num_vectors - 1`.
2. For each: `bus_alloc_resource_any(dev, SYS_RES_IRQ, &rid, RF_ACTIVE)` with `rid = i + 1`.
3. `bus_setup_intr(dev, vec->irq_res, INTR_TYPE_MISC | INTR_MPSAFE, vec->filter, NULL, vec, &vec->intr_cookie)`.
4. `bus_describe_intr(dev, vec->irq_res, vec->intr_cookie, vec->name)`.
5. `bus_bind_intr(dev, vec->irq_res, target_cpu)`.

Tear down a multi-vector driver:

1. Clear `INTR_MASK` at the device.
2. For each vector (reverse order): `bus_tearardown_intr`, `bus_release_resource`.
3. Drain each per-vector task.
4. Free the taskqueue.
5. `pci_release_msi(dev)` if MSI or MSI-X was used.

20.30.5 Useful Commands

- `pciconf -lvc`: list devices with capability lists.
- `vmstat -i`: show per-handler interrupt counts.
- `cpuset -g -x <irq>`: query CPU affinity for an IRQ.
- `cpuset -l <cpu> -x <irq>`: set CPU affinity for an IRQ.
- `sysctl dev.myfirst.0.intr_mode`: query driver's interrupt mode.

20.30.6 Files to Keep Bookmarked

- `/usr/src/sys/dev/pci/pcivar.h`: MSI/MSI-X inline wrappers.
- `/usr/src/sys/dev/pci/pci.h`: capability IDs and bit fields.
- `/usr/src/sys/dev/pci/pci.c`: kernel-side implementation of `pci_alloc_msi/msix`.

- `/usr/src/sys/dev/virtio/pci/virtio_pci.c`: clean MSI-X fallback-ladder example.
- `/usr/src/sys/dev/nvme/nvme_ctrlr.c`: per-queue MSI-X pattern at scale.

20.31 Reference: Glossary of Chapter 20 Terms

affinity: the mapping from an interrupt vector to a specific CPU (or set of CPUs).

bus_bind_intr(9): function to route an interrupt vector to a specific CPU.

bus_get_cpus(9): function to query CPU sets associated with a device (local, interrupt-suitable).

capability list: the linked list of PCI device capabilities in configuration space.

coalescing: buffering multiple events into one interrupt to reduce rate.

cookie: the opaque handle returned by `bus_setup_intr(9)`, used by `bus_tearardown_intr(9)`.

fallback ladder: the sequence MSI-X → MSI → legacy INTx that drivers implement.

intr_mode: driver-side enum recording which interrupt tier is active.

INTR_CPUS: `cpu_sets` enum value; CPUs suitable for handling device interrupts.

LOCAL_CPUS: `cpu_sets` enum value; CPUs in the same NUMA domain as the device.

MSI: Message Signalled Interrupts, PCI 2.2.

MSI-X: the fuller mechanism, PCIe.

moderation: buffering interrupts at the device level to trade latency for throughput.

NUMA: Non-Uniform Memory Access; multi-socket system architecture.

per-vector state: the softc fields specific to one vector (counters, filter, task, cookie, resource).

pci_msi_count(9) / pci_msix_count(9): capability-count queries.

pci_alloc_msi(9) / pci_alloc_msix(9): vector allocation.

pci_release_msi(9): release of MSI/MSI-X (handles both).

rid: resource ID. 0 for legacy INTx, 1+ for MSI/MSI-X vectors.

stray interrupt: an interrupt that no filter claims.

taskqueue: FreeBSD's deferred-work primitive.

vector: a single interrupt source in the MSI or MSI-X mechanism.

vmstat -i: diagnostic showing per-handler interrupt counts.

20.32 Reference: The Complete Stage 4 `myfirst_msix.c` Walkthrough

For readers who want a single place to see the final multi-vector layer annotated, this appendix walks through `myfirst_msix.c` from the companion examples, showing each function and explaining the design choices.

20.32.1 The Top of the File

```
#include <sys/param.h>
#include <sys/systm.h>
#include <sys/kernel.h>
#include <sys/bus.h>
#include <sys/lock.h>
#include <sys/mutex.h>
#include <sys/condvar.h>
#include <sys/rman.h>
#include <sys/sysctl.h>
#include <sys/taskqueue.h>
#include <sys/types.h>
#include <sys/smp.h>

#include <machine/atomic.h>
#include <machine/bus.h>
#include <machine/resource.h>

#include <dev/pci/pci.h>
#include <dev/pci/pcireg.h>
#include <dev/pci/pcivar.h>

#include "myfirst.h"
#include "myfirst_hw.h"
#include "myfirst_intr.h"
#include "myfirst_msix.h"
```

The include list is longer than `myfirst_intr.c`'s: `<dev/pci/pci.h>` and `<dev/pci/pcivar.h>` for the MSI/MSI-X API, `<sys/smp.h>` for `mp_ncpus`, and `<machine/atomic.h>` for the per-vector counter increments. Note that `<dev/pci/pci.h>` is pulled in even though the file does not directly use `PCIY_MSI` or similar constants; the accessor inline functions in `pcivar.h` depend on it.

20.32.2 The Per-Vector Helpers

```
static int
myfirst_msix_setup_vector(struct myfirst_softc *sc, int idx, int rid)
{
    struct myfirst_vector *vec = &sc->vectors[idx];
    int error;

    vec->irq_rid = rid;
    vec->irq_res = bus_alloc_resource_any(sc->dev, SYS_RES_IRQ,
```



```

        &vec->irq_rid, RF_ACTIVE);
    if (vec->irq_res == NULL)
        return (ENXIO);

    error = bus_setup_intr(sc->dev, vec->irq_res,
        INTR_TYPE_MISC | INTR_MPSAFE,
        vec->filter, NULL, vec, &vec->intr_cookie);
    if (error != 0) {
        bus_release_resource(sc->dev, SYS_RES_IRQ,
            vec->irq_rid, vec->irq_res);
        vec->irq_res = NULL;
        return (error);
    }

    bus_describe_intr(sc->dev, vec->irq_res, vec->intr_cookie,
        "%s", vec->name);
    return (0);
}

static void
myfirst_msix_teardown_vector(struct myfirst_softc *sc, int idx)
{
    struct myfirst_vector *vec = &sc->vectors[idx];

    if (vec->intr_cookie != NULL) {
        bus_teardown_intr(sc->dev, vec->irq_res, vec->intr_cookie);
        vec->intr_cookie = NULL;
    }
    if (vec->irq_res != NULL) {
        bus_release_resource(sc->dev, SYS_RES_IRQ,
            vec->irq_rid, vec->irq_res);
        vec->irq_res = NULL;
    }
}

```

These helpers are the symmetry pair from Section 3. Each takes a vector index and operates on the vec at that slot. The setup helper is idempotent in the sense that it leaves the vector in a clean state on failure; the teardown helper is safe to call even if setup did not complete.

20.32.3 The Per-Vector Filter Functions

The three filters differ only in which bit they check. Their common shape:

```

int
myfirst_msix_rx_filter(void *arg)
{
    struct myfirst_vector *vec = arg;
    struct myfirst_softc *sc = vec->sc;
    uint32_t status;

    status = ICSR_READ_4(sc, MYFIRST_REG_INTR_STATUS);
    if ((status & MYFIRST_INTR_DATA_AV) == 0) {
        atomic_add_64(&vec->stray_count, 1);
        return (FILTER_STRAY);
    }

    atomic_add_64(&vec->fire_count, 1);
    ICSR_WRITE_4(sc, MYFIRST_REG_INTR_STATUS, MYFIRST_INTR_DATA_AV);
    atomic_add_64(&sc->intr_data_av_count, 1);
    if (sc->intr_tq != NULL)
        taskqueue_enqueue(sc->intr_tq, &vec->task);
    return (FILTER_HANDLED);
}

```

The admin filter checks MYFIRST_INTR_ERROR, the tx filter checks MYFIRST_INTR_COMPLETE. Each increments the appropriate global counter and the per-vector counter. Only the rx filter enqueues a task.

20.32.4 The RX Task

```

static void
myfirst_msix_rx_task_fn(void *arg, int npending)
{
    struct myfirst_vector *vec = arg;
    struct myfirst_softc *sc = vec->sc;

    MYFIRST_LOCK(sc);
    if (sc->hw != NULL && sc->pci_attached) {
        sc->intr_last_data = CSR_READ_4(sc, MYFIRST_REG_DATA_OUT);
        sc->intr_task_invocations++;
        cv_broadcast(&sc->data_cv);
    }
    MYFIRST_UNLOCK(sc);
}

```

The task runs in thread context and takes `sc->mtx` safely. It checks `sc->pci_attached` before touching shared state, guarding against a race where the task runs during detach.

20.32.5 The Main Setup Function

The setup function orchestrates the fallback ladder:

```
int
myfirst_msix_setup(struct myfirst_softc *sc)
{
    int error, wanted, allocated, i;

    /* Initialise per-vector state common to all tiers. */
    for (i = 0; i < MYFIRST_MAX_VECTORS; i++) {
        sc->vectors[i].id = i;
        sc->vectors[i].sc = sc;
    }
    TASK_INIT(&sc->vectors[MYFIRST_VECTOR_RX].task, 0,
        myfirst_msix_rx_task_fn,
        &sc->vectors[MYFIRST_VECTOR_RX]);
    sc->vectors[MYFIRST_VECTOR_RX].has_task = true;
    sc->vectors[MYFIRST_VECTOR_ADMIN].filter = myfirst_msix_admin_filter;
    sc->vectors[MYFIRST_VECTOR_RX].filter = myfirst_msix_rx_filter;
    sc->vectors[MYFIRST_VECTOR_TX].filter = myfirst_msix_tx_filter;
    sc->vectors[MYFIRST_VECTOR_ADMIN].name = "admin";
    sc->vectors[MYFIRST_VECTOR_RX].name = "rx";
    sc->vectors[MYFIRST_VECTOR_TX].name = "tx";

    sc->intr_tq = taskqueue_create("myfirst_intr", M_WAITOK,
        taskqueue_thread_enqueue, &sc->intr_tq);
    taskqueue_start_threads(&sc->intr_tq, 1, PI_NET,
        "myfirst intr taskq");

    wanted = MYFIRST_MAX_VECTORS;

    /* Tier 0: MSI-X. */
    if (pci_msix_count(sc->dev) >= wanted) {
        allocated = wanted;
        if (pci_alloc_msix(sc->dev, &allocated) == 0 &&
            allocated == wanted) {
            for (i = 0; i < wanted; i++) {
                error = myfirst_msix_setup_vector(sc, i,
                    i + 1);
                if (error != 0) {
                    for (i -= 1; i >= 0; i--)
                        myfirst_msix_takedown_vector(
                            sc, i);
                    pci_release_msi(sc->dev);
                }
            }
        }
    }
}
```

```

        goto try_msi;
    }
}
sc->intr_mode = MYFIRST_INTR_MSIX;
sc->num_vectors = wanted;
myfirst_msix_bind_vectors(sc);
device_printf(sc->dev,
    "interrupt mode: MSI-X, %d vectors\n", wanted);
goto enabled;
}
if (allocated > 0)
    pci_release_msi(sc->dev);
}

try_msi:
/*
 * Tier 1: MSI with a single vector. MSI requires a power-of-two
 * count, so we cannot request MYFIRST_MAX_VECTORS (3) here. We
 * request 1 vector and fall back to the Chapter 19 single-handler
 * pattern, matching the approach sys/dev/virtio/pci/virtio_pci.c
 * takes in vtpci_alloc_msi().
 */
allocated = 1;
if (pci_msi_count(sc->dev) >= 1 &&
    pci_alloc_msi(sc->dev, &allocated) == 0 && allocated >= 1) {
    sc->vectors[MYFIRST_VECTOR_ADMIN].filter = myfirst_intr_filter;
    sc->vectors[MYFIRST_VECTOR_ADMIN].name = "msi";
    error = myfirst_msix_setup_vector(sc, MYFIRST_VECTOR_ADMIN, 1);
    if (error == 0) {
        sc->intr_mode = MYFIRST_INTR_MSI;
        sc->num_vectors = 1;
        device_printf(sc->dev,
            "interrupt mode: MSI, 1 vector "
            "(single-handler fallback)\n");
        goto enabled;
    }
    pci_release_msi(sc->dev);
}

try_legacy:
/* Tier 2: legacy INTx. */
sc->vectors[MYFIRST_VECTOR_ADMIN].filter = myfirst_intr_filter;
sc->vectors[MYFIRST_VECTOR_ADMIN].irq_rid = 0;
sc->vectors[MYFIRST_VECTOR_ADMIN].irq_res = bus_alloc_resource_any(

```

```

        sc->dev, SYS_RES_IRQ,
        &sc->vectors[MYFIRST_VECTOR_ADMIN].irq_rid,
        RF_SHAREABLE | RF_ACTIVE);
if (sc->vectors[MYFIRST_VECTOR_ADMIN].irq_res == NULL) {
    taskqueue_free(sc->intr_tq);
    sc->intr_tq = NULL;
    return (ENXIO);
}
error = bus_setup_intr(sc->dev,
    sc->vectors[MYFIRST_VECTOR_ADMIN].irq_res,
    INTR_TYPE_MISC | INTR_MPSAFE,
    myfirst_intr_filter, NULL, sc,
    &sc->vectors[MYFIRST_VECTOR_ADMIN].intr_cookie);
if (error != 0) {
    bus_release_resource(sc->dev, SYS_RES_IRQ,
        sc->vectors[MYFIRST_VECTOR_ADMIN].irq_rid,
        sc->vectors[MYFIRST_VECTOR_ADMIN].irq_res);
    sc->vectors[MYFIRST_VECTOR_ADMIN].irq_res = NULL;
    taskqueue_free(sc->intr_tq);
    sc->intr_tq = NULL;
    return (error);
}
bus_describe_intr(sc->dev,
    sc->vectors[MYFIRST_VECTOR_ADMIN].irq_res,
    sc->vectors[MYFIRST_VECTOR_ADMIN].intr_cookie, "legacy");
sc->intr_mode = MYFIRST_INTR_LEGACY;
sc->num_vectors = 1;
device_printf(sc->dev,
    "interrupt mode: legacy INTx (1 handler for all events)\n");

enabled:
    MYFIRST_LOCK(sc);
    if (sc->hw != NULL)
        CSR_WRITE_4(sc, MYFIRST_REG_INTR_MASK,
            MYFIRST_INTR_DATA_AV | MYFIRST_INTR_ERROR |
            MYFIRST_INTR_COMPLETE);
    MYFIRST_UNLOCK(sc);

    return (0);
}

```

The function is long because it handles three tiers, each with its own allocation, per-vector setup loop, and partial-failure unwind. A reader tracing the flow sees MSI-X tried first, falling through to MSI on any failure, falling through to legacy on any failure there. The `enabled:` label is reached from any successful tier.

The legacy tier is the Chapter 19 path: one filter (`myfirst_intr_filter` from `myfirst_intr.c`), `rid = 0`, `RF_SHAREABLE`. The per-vector counters are not really used on this tier; the Chapter 19 code does its own counting.

20.32.6 The Teardown Function

```
void
myfirst_msix_teardown(struct myfirst_softc *sc)
{
    int i;

    MYFIRST_LOCK(sc);
    if (sc->hw != NULL && sc->bar_res != NULL)
        CSR_WRITE_4(sc, MYFIRST_REG_INTR_MASK, 0);
    MYFIRST_UNLOCK(sc);

    for (i = sc->num_vectors - 1; i >= 0; i--)
        myfirst_msix_teardown_vector(sc, i);

    if (sc->intr_tq != NULL) {
        for (i = 0; i < sc->num_vectors; i++) {
            if (sc->vectors[i].has_task)
                taskqueue_drain(sc->intr_tq,
                                &sc->vectors[i].task);
        }
        taskqueue_free(sc->intr_tq);
        sc->intr_tq = NULL;
    }

    if (sc->intr_mode == MYFIRST_INTR_MSI ||
        sc->intr_mode == MYFIRST_INTR_MSIX)
        pci_release_msi(sc->dev);

    sc->num_vectors = 0;
    sc->intr_mode = MYFIRST_INTR_LEGACY;
}
```

The function follows the strict ordering: disable at device, per-vector teardown in reverse, per-vector task drain, free taskqueue, release MSI. No surprises; the symmetry is the payoff.

20.32.7 The Bind Function

```
static void
myfirst_msix_bind_vectors(struct myfirst_softc *sc)
{
    int i, cpu;
    int err;

    if (mp_ncpus < 2)
        return;

    for (i = 0; i < sc->num_vectors; i++) {
        cpu = i % mp_ncpus;
        err = bus_bind_intr(sc->dev, sc->vectors[i].irq_res, cpu);
        if (err != 0)
            device_printf(sc->dev,
                "bus_bind_intr vec %d: %d\n", i, err);
    }
}
```

Round-robin binding. Only called on MSI-X (the function is not useful on MSI or legacy; the setup ladder skips it for those tiers). On single-CPU systems the function returns early without binding.

20.32.8 The sysctl Function

```
void
myfirst_msix_add_sysctls(struct myfirst_softc *sc)
{
    struct sysctl_ctx_list *ctx = &sc->sysctl_ctx;
    struct sysctl_oid_list *kids = SYSCTL_CHILDREN(sc->sysctl_tree);
    char name[32];
    int i;

    SYSCTL_ADD_INT(ctx, kids, OID_AUTO, "intr_mode",
        CTLFLAG_RD, &sc->intr_mode, 0,
        "0=legacy, 1=MSI, 2=MSI-X");

    for (i = 0; i < MYFIRST_MAX_VECTORS; i++) {
        snprintf(name, sizeof(name), "vec%d_fire_count", i);
        SYSCTL_ADD_U64(ctx, kids, OID_AUTO, name,
            CTLFLAG_RD, &sc->vectors[i].fire_count, 0,
            "Times this vector's filter was called");
    }
}
```

```

    snprintf(name, sizeof(name), "vec%d_stray_count", i);
    SYSCTL_ADD_U64(ctx, kids, OID_AUTO, name,
        CTLFLAG_RD, &sc->vectors[i].stray_count, 0,
        "Stray returns from this vector");
}

SYSCTL_ADD_PROC(ctx, kids, OID_AUTO, "intr_simulate_admin",
    CTLTYPE_UINT | CTLFLAG_WR | CTLFLAG_MPSAFE,
    &sc->vectors[MYFIRST_VECTOR_ADMIN], 0,
    myfirst_intr_simulate_vector_sysctl, "IU",
    "Simulate admin vector interrupt");
SYSCTL_ADD_PROC(ctx, kids, OID_AUTO, "intr_simulate_rx",
    CTLTYPE_UINT | CTLFLAG_WR | CTLFLAG_MPSAFE,
    &sc->vectors[MYFIRST_VECTOR_RX], 0,
    myfirst_intr_simulate_vector_sysctl, "IU",
    "Simulate rx vector interrupt");
SYSCTL_ADD_PROC(ctx, kids, OID_AUTO, "intr_simulate_tx",
    CTLTYPE_UINT | CTLFLAG_WR | CTLFLAG_MPSAFE,
    &sc->vectors[MYFIRST_VECTOR_TX], 0,
    myfirst_intr_simulate_vector_sysctl, "IU",
    "Simulate tx vector interrupt");
}

```

The function builds three read-only per-vector counter sysctls and three write-only simulated-interrupt sysctls. The tree-style (Challenge 7) is left as an exercise.

20.32.9 Lines of Code

The full `myfirst_msix.c` file is about 330 lines. That is a substantial addition to the driver, but it buys all of Chapter 20's capabilities: three-tier fallback, per-vector handlers, per-vector counters, CPU binding, clean teardown.

Compare to the Chapter 19 `myfirst_intr.c` which was about 250 lines. The Chapter 20 file is not much longer in absolute terms; the per-vector logic adds complexity but each piece is small.

20.33 Reference: A Closing Note on Multi-Vector Philosophy

A paragraph to close the chapter with.

A multi-vector driver does not fundamentally differ from a single-vector driver. It has the same filter shape, the same task pattern, the same teardown ordering, the same lock discipline. What changes is the count: *N* filters instead of one, *N* teardowns instead of one, *N* tasks instead of one. The design's quality comes from how cleanly those *N* pieces coexist.

Chapter 20's lesson is that multi-vector handling is an exercise in symmetry. Each vector looks like every other vector at the structural level; each has its own counter, its own filter, its own description. The code that allocates, the code that handles, the code that tears down: all of them loop over the vectors and do the same thing N times. The simplicity of the loop is what makes the N-vector driver manageable; a driver where each vector is special is a driver that does not scale.

For this reader and for this book's future readers, the Chapter 20 multi-vector pattern is a permanent part of the `myfirst` driver's architecture and a permanent tool in the reader's toolkit. Chapter 21 assumes it: per-queue DMA rings, per-queue completion interrupts, per-queue CPU placement. The vocabulary is the vocabulary every high-performance FreeBSD driver shares; the patterns are the patterns the kernel's own test drivers use; the discipline is the discipline production drivers live by.

The skill Chapter 20 teaches is not "how to allocate MSI-X for `virtio-rng-pci`". It is "how to design a multi-vector driver, allocate its vectors, place them on CPUs, route events per vector, and tear it all down cleanly". That skill applies across every multi-queue device the reader will ever work on.

21 DMA and High-Speed Data Transfer

21.1 Reader Guidance & Outcomes

Chapter 20 closed with a driver that handles interrupts well. The `myfirst` module at version `1.3-msi` has a three-tier fallback ladder (MSI-X first, MSI next, legacy INTx last); three per-vector filter handlers wired to separate roles (admin, receive, transmit); per-vector counters; per-vector CPU binding through `bus_bind_intr(9)`; a clean teardown that releases every vector in reverse order; a new `myfirst_msix.c` file that keeps the multi-vector code tidy; and an `MSIX.md` document that describes the per-vector design. Every vector's handler runs in the Chapter 19 discipline: a short filter in primary interrupt context that reads status, acknowledges, and either handles or defers, and a task in thread context that does the bulk work under the `sleep-mutex`.

What the driver still does not do is move data. Every byte the device has produced so far has been read one register at a time from the CPU's point of view. The simulated device writes a word into `DATA_OUT`; the CPU reads it with `bus_read_4`. That works when there is one word to read. It does not work when there are sixty-four receive descriptors to walk, or when a storage controller has completed a four-kilobyte block transfer, or when a NIC has just placed a nine-kilobyte jumbo frame into host memory. At those rates, reading word by word collapses the throughput the hardware was designed to deliver. The device needs to reach into RAM on its own, write the data there, and then tell the driver where to find it. That is what DMA means, and it is what Chapter 21 teaches.

The chapter's scope is precisely this transition: what DMA is at the level of the PCIe fabric; why a device writing to arbitrary memory would be a correctness and safety nightmare without an abstraction layer; how FreeBSD's `bus_dma(9)` interface provides that layer in a portable way; how to create DMA tags that describe a device's addressing constraints; how to allocate DMA-capable memory and load it into a mapping; how to synchronise the CPU's view with the device's view at the right moments; how to extend the Chapter 17 simulated backend with a DMA engine; how to process completion interrupts through the Chapter 20 per-vector path; how to recover from the failures that real DMA code must handle; and how to refactor all of this into a file the reader can read, test, and extend. The chapter stops short of the subsystems that build on top of DMA. The network driver chapter in Part 6 (Chapter 28) revisits DMA inside the `iflib(9)` framework; the power-management chapter that follows (Chapter 22) will teach how to quiesce DMA safely during suspend; the performance chapter in Part 7 (Chapter 33) will return to DMA with a tuning lens. Chapter 21 stays focused on the primitives that every later chapter assumes.

Chapter 21 stops short of several load paths that build on the same foundation. Scatter-gather DMA for mbuf chains (`bus_dmamap_load_mbuf_sg`), CAM control blocks (`bus_dmamap_load_ccb`), user I/O structures (`bus_dmamap_load_uio`), and crypto operations (`bus_dmamap_load_crp`) all use the same underlying mechanism and the same synchronisation discipline, but each comes with its own context (networking, storage, VFS, OpenCrypto) that belongs in its own chapter. Zero-copy user-space DMA, in which a driver lets user space map a DMA buffer and coordinate its synchronisation through `sys_msync(2)` or similar primitives, sits outside the scope of Chapter 21; Chapter 10 Section 8 introduced the `d_mmap(9)` path it would build on, and

Chapter 31 discusses the security implications of exposing kernel memory through `mmap(2)`-style interfaces. IOMMU-assisted remapping (`busdma_iommu`) is mentioned for context but not configured by hand. The reader who finishes Chapter 21 will understand the primitives well enough that the specialised load paths look like variations on the base case; that is the goal.

The `bus_dma(9)` story sits on every Part 4 layer we have built. Chapter 16 gave the driver a vocabulary of register access. Chapter 17 taught it to think like a device. Chapter 18 introduced it to a real PCI device. Chapter 19 gave it ears on one IRQ. Chapter 20 gave it several ears, one per queue the device wants. Chapter 21 gives it hands: the ability to hand the device a physical address and say “put your data here, tell me when you are done, and let me process it without bothering you for each byte”. That is the last missing primitive before Part 4 closes.

21.1.1 Why `bus_dma(9)` Earns a Chapter of Its Own

Before we go further, it is worth pausing on what `bus_dma(9)` buys us that a loop of `bus_read_4` calls cannot. The Chapter 20 driver has a complete interrupt pipeline. If the filter-plus-task pattern is already right, and the register accessor layer already handles reads and writes cleanly, why not just read bigger blocks word by word and be done?

Three reasons.

The first is **bandwidth**. A single `bus_read_4` is one MMIO transaction on the PCIe bus, and each transaction costs hundreds of nanoseconds once the overhead of address decoding, the read completion round-trip, and the CPU’s memory ordering are accounted for. On a PCIe 3.0 x4 link, a driver that uses `bus_read_4` word by word tops out around twenty megabytes per second of effective throughput; a driver that arranges for the device to DMA into a contiguous buffer reaches gigabytes per second on the same link. The order-of-magnitude gap is what separates a ten-gigabit NIC from an unusable NIC, a modern NVMe from a late-1990s IDE controller. DMA is not an optimisation for drivers that could work without it; it is the only mechanism that lets modern devices deliver their designed throughput.

The second is **CPU cost**. Every `bus_read_4` or `bus_write_4` keeps a CPU busy for the duration of the transaction. A driver that moves a megabyte of data one word at a time burns ten to twenty milliseconds of CPU time just shuttling bytes across MMIO. DMA offloads that cost to the device’s own bus master engine: the CPU hands over an address and a length, the device runs the transfer independently, and the CPU is free to do other work (including handling interrupts for other devices). On a server pushing millions of packets per second through several NICs at once, the CPU cannot afford to touch each byte; DMA is what makes aggregate throughput achievable.

The third is **correctness under concurrency**. A driver that reads a descriptor ring word by word is racing the device: the device may be writing new entries while the driver is reading old ones, and the driver sees torn reads of half-updated fields unless it takes out a global lock that serialises the whole transfer. DMA with proper synchronisation replaces that race with a clean producer-consumer protocol: the device writes entries in order, signals completion through a single register write or a completion interrupt, and the CPU processes the entries as a batch with the guarantee that every byte is there. The `bus_dmamap_sync` call makes the hand-off explicit; the `bus_dmamap_unload` call makes the cleanup explicit. The driver becomes easier to reason about, not harder, even though the mechanism is more sophisticated.

Chapter 21 earns its place by teaching all three benefits concretely. A reader finishes the chapter able to create a tag, allocate a buffer, load it into a map, trigger a transfer, synchronise around it, wait for completion, verify

the result, and tear everything down. With those primitives in hand, the reader can open any DMA-capable FreeBSD driver and recognise its structure, the same way Chapter 20's graduate can read any multi-vector driver.

21.1.2 Where Chapter 20 Left the Driver

A few prerequisites to verify before starting. Chapter 21 extends the driver produced at the end of Chapter 20 Stage 4, tagged as version 1.3-`msi`. If any of the items below feels uncertain, return to Chapter 20 before starting this chapter.

- Your driver compiles cleanly and identifies itself as 1.3-`msi` in `kldstat -v`.
- On a QEMU guest that exposes `virtio-rng-pci` with MSI-X, the driver attaches; chooses MSI-X; allocates three vectors (admin, rx, tx); registers a distinct filter per vector; binds each vector to a CPU; prints an interrupt mode: MSI-X, 3 vectors banner; and creates `/dev/myfirst0`.
- On a bhyve guest with `virtio-rnd`, the driver attaches; falls through to MSI with one vector, or further through to legacy INTx; prints the corresponding banner.
- Per-vector counters (`dev.myfirst.0.vec0_fire_count` through `vec2_fire_count`) increment when the matching simulation `sysctl` (`dev.myfirst.0.intr_simulate_admin`, `intr_simulate_rx`, or `intr_simulate_tx`) is written to.
- The detach path tears down vectors in reverse order, drains per-vector tasks, releases resources, and calls `pci_release_msi` exactly once.
- `HARDWARE.md`, `LOCKING.md`, `SIMULATION.md`, `PCI.md`, `INTERRUPTS.md`, and `MSIX.md` are current in your working tree.
- `INVARIANTS`, `WITNESS`, `WITNESS_SKIPSPIN`, `DDB`, `KDB`, and `KDB_UNATTENDED` are enabled in your test kernel.

That driver is what Chapter 21 extends. The additions are noticeable in scope: one new file (`myfirst_dma.c`), one new header (`myfirst_dma.h`), several new `softc` fields to track DMA tag, map, memory pointer, and the DMA engine's simulated state, a new pair of helper functions (`myfirst_dma_setup` and `myfirst_dma_teardown`), a new completion path in the receive filter, a version bump to 1.4-`dma`, a new `DMA.md` document, and updates to the regression test. The mental model grows too: the driver starts thinking of memory ownership as something that passes back and forth between the CPU and the device, and every hand-off becomes a deliberate `bus_dmamap_sync` call.

21.1.3 What You Will Learn

After finishing this chapter you will be able to:

- Describe what DMA is at the hardware level, how a PCIe device performs a bus-master write to host memory, and why that mechanism delivers bandwidth and CPU-offload benefits that MMIO cannot match.
- Explain why direct device access to arbitrary memory would be unsafe without an abstraction layer, and name the three hardware realities that layer has to hide (device addressing limits, IOMMU remapping, cache coherence).
- Recognise where bounce buffers fit in the picture: when the kernel silently inserts them, what they cost, and when an explicit address constraint can avoid them.

- Read and write the central `bus_dma(9)` vocabulary: `bus_dma_tag_t`, `bus_dmamap_t`, `bus_dma_segment_t`, the PRE/POST sync operations, the tag constraints (alignment, boundary, lowaddr, highaddr, maxsize, nsegments, maxsegsz), and the common flag set (`BUS_DMA_WAITOK`, `BUS_DMA_NOWAIT`, `BUS_DMA_COHERENT`, `BUS_DMA_ZERO`).
- Create a device-scoped DMA tag with `bus_dma_tag_create`, inheriting the parent bridge's constraints through `bus_get_dma_tag(9)`, and choosing alignment, boundary, and address limits that match the device's datasheet.
- Allocate DMA-capable memory with `bus_dmamem_alloc`, obtain its kernel virtual address, and understand why the allocator returns a single segment with `bus_dmamem_alloc` but may return several segments for arbitrary memory loaded later.
- Load a kernel buffer into a DMA map with `bus_dmamap_load`, extract the bus address from the single-segment callback, and understand the cases where the callback is deferred and what that means for the driver's locking discipline.
- Use `bus_dmamap_sync` with the correct PRE/POST flags around every device-visible memory access: `PREWRITE` before the device reads, `POSTREAD` after the device writes, and the combined pairs for descriptor rings where both directions happen.
- Extend the Chapter 17 simulated backend with a small DMA engine that takes a bus address, a length, and a direction from the softc, runs the transfer under a `callout(9)` to emulate latency, and raises a completion interrupt through the Chapter 19/20 filter path.
- Process completion interrupts by reading a status register inside the rx filter, acknowledging, enqueueing the task, and letting the task perform `bus_dmamap_sync` and access the buffer.
- Recover from every recoverable failure mode: `bus_dma_tag_create` returning `ENOMEM`, `bus_dmamap_load` returning `EINVAL` or `EINVAL`, the engine reporting a partial transfer, a timeout that expires before the device signals completion, and a detach that fires while a transfer is in flight.
- Refactor the DMA code into a dedicated `myfirst_dma.c` / `myfirst_dma.h` pair, with `myfirst_dma_setup` and `myfirst_dma_teardown` as the only entry points used by the rest of the driver.
- Version the driver as `1.4-dma`, update the Makefile's `SRCS` line, run the extended regression test, and produce `DMA.md` documenting the DMA flow, the buffer layouts, and the observable counters.
- Read the DMA code in a real driver (`/usr/src/sys/dev/re/if_re.c` is the chapter's running reference) and map each call to the concepts introduced in Chapter 21.

The list is long; each item is narrow. The point of the chapter is the composition.

21.1.4 What This Chapter Does Not Cover

Several adjacent topics are explicitly deferred so Chapter 21 stays focused.

- **Scatter-gather DMA for heterogeneous buffers.** `bus_dmamap_load_mbuf_sg` (networking), `bus_dmamap_load_ccb` (CAM storage), `bus_dmamap_load_uio` (VFS), and `bus_dmamap_load_crp` (OpenCrypto) all sit on top of the same `bus_dma(9)` foundation but are used in subsystem-specific ways. Part 6 (Chapter 27 storage, Chapter 28 network) cover them in context. Chapter 21 uses `bus_dmamap_load` on a single contiguous buffer; the rest is a specialisation.
- **iflib(9) and its hidden DMA pools.** The networking framework wraps `bus_dma` with per-queue helpers that allocate, load, and sync receive and transmit rings automatically. The framework is the subject of its own chapter in Part 6 (Chapter 28); Chapter 21 teaches the raw layer that `iflib` uses internally.

- **IOMMU-assisted DMA on amd64 with Intel VT-d or AMD-Vi.** The `busdma_iommu` machinery integrates transparently with the `bus_dma` API, so a driver written for the generic path automatically benefits from IOMMU remapping when the kernel is built with `DEV_IOMMU`. The chapter mentions IOMMU presence, explains what it does, and shows how to observe it; it does not configure it by hand.
- **NUMA-aware DMA memory placement.** `bus_dma_tag_set_domain(9)` lets a driver bind a tag's allocations to a specific NUMA domain. The function is named and mentioned; the full placement story is a Part 7 (Chapter 33) performance topic.
- **Power-aware DMA quiescing.** Stopping in-flight DMA before suspend is Chapter 22's topic. Chapter 21 arranges the primitives that Chapter 22 will use; the `myfirst_dma_tear` down path is designed so that Chapter 22's suspend handler can invoke it cleanly.
- **Zero-copy DMA to and from user-space buffers.** Mapping user pages into a DMA map requires `vm_fault_quick_hold_pages(9)` plus `bus_dmamap_load_ma_triv` or an equivalent, and touches on memory pinning, VM reference counts, and capability enforcement. These topics belong in a later chapter.
- **DMA descriptor rings with hardware tail/head registers.** A full ring design (producer/consumer indexes, wrap-around, doorbell writes) is a next-level pattern built on Chapter 21's primitives. The chapter shows a single-transfer pattern; descriptor rings are a natural extension the reader can build as a challenge.

Staying inside those lines keeps Chapter 21 a chapter about DMA primitives. The vocabulary is what transfers; the later chapters apply it to networking, storage, power, and performance in turn.

21.1.5 Estimated Time Investment

- **Reading only:** five to six hours. The DMA conceptual model is the densest in Part 4; the sync discipline in particular benefits from a careful read and then a second pass once the code examples have concretised it.
- **Reading plus typing the worked examples:** twelve to fifteen hours over two or three sessions. The driver evolves in four stages: tag and allocation, simulated engine with polling, simulated engine with interrupt completion, and final refactor. Each stage is small but builds on the previous one, and each requires careful attention to the PRE/POST pairs.
- **Reading plus all labs and challenges:** eighteen to twenty-four hours over four or five sessions, including reading real drivers (`if_re.c`, the `nvme_qpair.c` allocation code, and the `busdma_bufalloc.c` sources if curiosity takes you that far), running the chapter's regression test on both bhyve and QEMU targets, and attempting one or two of the production-style challenges.

Sections 3, 4, and 5 are the densest. If the sync discipline or the tag constraints feel opaque on first pass, that is normal. Stop, re-read Section 4's diagram, run the Section 4 exercise, and continue when the shape has settled. DMA is one of the topics where a working mental model pays off many times; it is worth building slowly.

21.1.6 Prerequisites

Before starting this chapter, confirm:

- Your driver source matches Chapter 20 Stage 4 (`1.3-msi`). The starting point assumes every Chapter 20 primitive: the three-tier fallback ladder, the per-vector filters, per-vector CPU binding, and the clean multi-vector teardown.

- Your lab machine runs FreeBSD 14.3 with `/usr/src` on disk and matching the running kernel.
- A debug kernel with `INVARIANTS`, `WITNESS`, `WITNESS_SKIPSPIN`, `DDB`, `KDB`, and `KDB_UNATTENDED` is built, installed, and booting cleanly. The `WITNESS` option is especially valuable for DMA work because several `bus_dma` functions must not be called with non-sleepable locks held, and `WITNESS` catches the violations early.
- `bhyve(8)` or `qemu-system-x86_64` is available. Chapter 21's labs work on either target; the DMA simulation happens inside the `myfirst` driver, so no particular DMA-capable guest device is required. The real-DMA sanity check at the end of Section 5 uses whatever PCI device the chapter's virtio host exposes, without depending on it for DMA.
- The `devinfo(8)`, `vmstat(8)`, `pciconf(8)`, `sysctl(8)`, and `procstat(1)` tools are in your path. `procstat -kke` is useful for watching the driver's threads during DMA transfers.

If any item above is shaky, fix it now. DMA tends to expose any latent weakness in the driver's lock discipline (the sync calls sit inside the task context, the allocation calls sit inside `attach`, and the unload calls sit inside `detach`; each context has different lock rules), and `WITNESS` in a debug kernel is what catches the mistakes at development time rather than in production.

21.1.7 How to Get the Most Out of This Chapter

Four habits will pay off quickly.

First, keep `/usr/src/sys/sys/bus_dma.h` and `/usr/src/share/man/man9/bus_dma.9` bookmarked. The header is compact (about four hundred lines) and lists every API the chapter uses, with a brief comment on each. The manual page is substantial (over eleven hundred lines) and is the authoritative reference on every parameter. Reading both once at the start of Section 2 and returning to them as you work each section is the single most useful thing you can do for fluency.

Second, keep `/usr/src/sys/dev/re/if_re.c` bookmarked as the running real-driver example. `if_re(4)` is a reasonably compact network driver that uses `bus_dma` in a textbook way: a parent tag that inherits from the PCI bridge, a per-ring tag for descriptors, a per-buffer tag for mbuf payloads, a single-segment callback, and clean teardown. Most of Chapter 21's patterns have a direct analogue in `if_re.c`. When the chapter shows a pattern, it will often say "look at `re_allocmem`" and give a specific line.

A note on line numbers. When the chapter pins a reference to a line inside `if_re.c`, treat the number as a pointer for the FreeBSD 14.3 tree at time of writing, not a stable coordinate. `re_allocmem` and `re_dma_map_addr` are durable names; the line each one sits on is not. Open the file, search for the function name, and let your editor report where it actually lives on the system in front of you.

Third, type the changes by hand and run each stage. DMA is where the cost of a wrong flag or a missed sync is paid in subtle corruption rather than an obvious crash, and typing carefully makes the common mistakes visible at the moment they happen. The chapter's regression test will catch many of the mistakes, but the typing itself is what builds the mental model.

Fourth, after finishing Section 4, re-read Sections 2 and 3 of Chapter 17. The simulated backend's MMIO register map is the base the simulated DMA engine extends, and seeing the two layered together is what makes the engine's design feel inevitable rather than arbitrary. The Chapter 17 simulation was built with DMA in mind; Chapter 21 is where that design pays off.

21.1.8 Roadmap Through the Chapter

The sections in order are:

1. **What Is DMA and Why Use It?** The hardware picture: bus-master writes, the bandwidth and CPU-cost argument, concrete device examples, and why modern driver development assumes DMA.
2. **Understanding FreeBSD's `bus_dma(9)` Interface.** The abstraction layer: what the layer hides (addressing limits, IOMMU, coherence), what a tag is, what a map is, what a segment is, and how the pieces fit together. Concepts first, code next.
3. **Allocating and Mapping DMA Memory.** The tag parameters in detail, `bus_dmamem_alloc`, the `bus_dmamap_load` callback pattern, the single-segment case, and the first running code: Stage 1 of the Chapter 21 driver (`1.4-dma-stage1`).
4. **Synchronising and Using DMA Buffers.** The sync discipline: `PREWRITE`, `POSTREAD`, the combined pairs, the coherent-vs-non-coherent nuance, and the mental model of memory ownership passing between CPU and device. The exercise at the end is a paper walkthrough before any code.
5. **Building a Simulated DMA Engine.** The simulated backend's DMA registers, the state machine, the `callout(9)`-driven transfer, and the completion path. Stage 2 (`1.4-dma-stage2`) makes the driver hand a physical address to the engine and poll for completion.
6. **Handling DMA Completion in the Driver.** The filter-plus-task path from Chapter 19/20 consumes the completion interrupt. The task performs `bus_dmamap_sync(..., POSTREAD)` and reads the buffer. Stage 3 (`1.4-dma-stage3`) wires the interrupt path to the DMA engine.
7. **DMA Error Handling and Edge Cases.** Every failure mode the chapter addresses, with the matching recovery pattern: mapping failures, partial transfers, timeouts, and detach-while-in-flight.
8. **Refactoring and Versioning Your DMA-Capable Driver.** The final split into `myfirst_dma.c` and `myfirst_dma.h`, the updated Makefile, the `DMA.md` document, and the version bump. Stage 4 (`1.4-dma`).

After the eight sections come an extended walkthrough of `if_re.c`'s DMA code, several deeper looks at bounce buffers, 32-bit devices, IOMMU, coherent flags, and parent tag inheritance, a set of hands-on labs, a set of challenge exercises, a troubleshooting reference, a Wrapping Up that closes Chapter 21's story and opens Chapter 22's, a bridge, and the usual quick-reference and glossary material at the end of the chapter. The reference material is meant to be re-read as you work through the next few chapters; Chapter 21's vocabulary (tag, map, segment, PRE/POST, coherent, bounce, callback) is the foundation every chapter after this one assumes.

If this is your first pass, read linearly and do the labs in order. If you are revisiting, Sections 3, 4, and 5 stand alone and make good single-sitting reads.

21.2 Section 1: What Is DMA and Why Use It?

Before the driver code, the hardware picture. Section 1 teaches what DMA is at the level of the PCIe bus, what it gains the driver over MMIO reads and writes, what the real-world examples look like, and why modern driver development takes DMA as a baseline assumption rather than an optimisation. A reader who finishes Section 1 can read the rest of the chapter with the kernel's `bus_dma` layer as a concrete object rather than a vague abstraction.

21.2.1 The Problem With MMIO-Only Data Movement

Chapter 16 introduced memory-mapped I/O: a device's registers live inside a region of physical memory, the kernel maps that region into the kernel virtual address space, and `bus_read_4` and `bus_write_4` wrappers turn reads and writes into bus transactions. Chapter 17 built a simulated backend that looks the same from the driver's perspective. Chapter 18 moved that backend onto a real PCI BAR. Every word of data the Chapter 19 and Chapter 20 driver has handled has come through `bus_read_4` from the device's `DATA_OUT` register.

For a driver that handles one word per event, that model works. For a driver that has to handle a megabyte of data, it does not. Each `bus_read_4` is one MMIO transaction; each MMIO transaction is a posted or completion bus cycle with its own overhead. On a typical PCIe 3.0 x4 link the round-trip time per transaction is a few hundred nanoseconds once the PCIe header, the read completion, and the CPU's memory ordering instructions are added up. A driver that moves data one word at a time tops out at a few tens of megabytes per second regardless of what the device is actually capable of.

A modern NIC is capable of tens of gigabytes per second. A modern NVMe is capable of several gigabytes per second of sustained I/O. A USB3 host controller handles thousands of descriptors per second. A GPU handles hundreds of megabytes of command buffers per frame. The hardware is not the bottleneck in any of these cases; the MMIO model is. Every driver for these devices has to use a different mechanism to move the bulk of the data, and that mechanism is DMA.

21.2.2 What DMA Means at the Hardware Level

DMA stands for Direct Memory Access. The word “direct” is the key. In an MMIO transfer, the CPU is the master: the CPU issues a read or write to the device's MMIO region, and the device responds. In a DMA transfer, the device is the master: the device issues a read or write to host memory, and the memory controller responds as if the CPU had made the request. The CPU is not in the loop for each word; the device moves the bytes on its own schedule.

Physically, a DMA transfer is a memory transaction on the same bus the CPU uses to reach RAM. On PCIe this is called a **bus-master** transaction, and PCI devices that can do it are said to be **bus-mastering**. The device holds an internal state (a source address, a destination address, a length, a direction, a status) and advances through the transfer under its own clock. The CPU sets up the transfer by writing the relevant parameters to the device's control registers, then issues a go-ahead write (often called a “doorbell”), and can then do something else until the device signals completion.

From the CPU's perspective, a DMA transfer passes through roughly these steps:

1. The driver allocates a region of host memory with properties the device can use (addressable by the device, aligned as the device requires, contiguous if the device requires that).
2. The driver arranges for the device to see that region at some **bus address** (which may or may not equal the host's physical address depending on the platform).
3. The driver writes the bus address, the length, and the direction to the device's registers.
4. The driver writes a doorbell register that tells the device to begin.
5. The device performs the transfer on its own, reading from or writing to the host memory at the bus address, through the memory controller, while the CPU is free.
6. The device signals completion, usually by writing a status bit in an MMIO register, raising an interrupt, or both.

7. The driver performs any necessary cache synchronisation between the CPU's view of the memory and the device's view.
8. The driver reads or uses the transferred data.

Each step looks simple. The subtlety lies in step 2 (what bus address to use, and how to make the device see the buffer) and step 7 (what “necessary” means, and how the driver knows when to do which kind of sync). Most of Chapter 21 is about those two steps.

21.2.3 What the Driver Gains

Three concrete benefits matter in every DMA-capable driver.

The first is **throughput**. On PCIe, a bus-master transfer uses the device's own width and clock. A ten-gigabit NIC's PCIe 3.0 x4 link can sustain roughly three and a half gigabytes per second of bus-mastered data. The same link used for MMIO alone tops out near twenty megabytes per second: the NIC cannot touch its advertised bandwidth without DMA. The ratio is worse for higher-performance devices. A PCIe 4.0 x16 GPU has a theoretical link budget over thirty gigabytes per second; no amount of `bus_write_4` can approach it.

The second is **CPU offload**. Bus-master transfers consume little to no CPU time once started. The CPU writes a few registers, triggers the doorbell, and is free until the completion interrupt arrives. On a system with many DMA-capable devices, the CPU can orchestrate dozens of concurrent transfers as long as the device has enough DMA engines. On a NIC that supports multi-queue transmit, the CPU can queue up receive descriptors, hand transmit buffers to the device, and process completions for both directions at once, with the actual byte movement happening in the device's internal DMA engines. The CPU's job becomes policy and bookkeeping; the data movement is elsewhere.

The third is **determinism under load**. An MMIO-based driver that is busy-looping through a large buffer can be preempted by an interrupt, the `ithread` that handles the interrupt can monopolise the CPU for an extended period, and the throughput becomes a function of kernel preemption timing rather than device capability. A DMA-based driver has the device do the work; the driver's own code runs for a predictable, small number of cycles per transfer and yields to the scheduler in between. Latency distributions tighten, tail latencies shrink, and the driver's performance becomes easier to reason about.

These three benefits compound. A driver that uses DMA consumes less CPU per byte, moves more bytes per unit time, and does both more predictably. The cost is the `bus_dma` setup code, which adds a few hundred lines to the driver. The chapter is about writing those lines well.

21.2.4 Real-World Examples

DMA is pervasive in modern hardware. A few concrete examples make the scale of its use clear.

A **network card** uses DMA for both receive and transmit. The device has a **receive ring** in host memory: an array of descriptors, each of which contains a bus address and a length pointing at a packet buffer. The device copies incoming packets from the wire into the buffers via bus-master writes, updates each descriptor's status field when the packet is complete, and raises an interrupt. The driver walks the ring, processes each completed descriptor, and refills the ring with fresh buffers. Transmit works the other way: the driver places outgoing packet headers and payloads into buffers, writes descriptors with the bus addresses, and kicks the device; the device reads the buffers, transmits, and updates the descriptor status. The whole protocol runs in host memory,

synchronised through `bus_dmamap_sync` calls and completion interrupts. Chapter 21's driver is a single-transfer simplification of this pattern; Part 6 (Chapter 28) generalises to full rings.

A **storage controller** (SATA, SAS, NVMe) uses DMA for every block transfer. The driver issues a command block containing a physical-address list (a scatter-gather list) that describes the pages making up the host buffer. The device walks the list, reads or writes each page, and signals completion. Modern NVMe controllers use a **Physical Region Page (PRP)** or **Scatter Gather List (SGL)** structure to describe the transfer, and the controller writes the completion into a completion queue that is itself in host memory via DMA. The `nvme(4)` driver is about four thousand lines of DMA-aware code; the structure is comfortable to read once Chapter 21's primitives are in place.

A **USB3 host controller** uses DMA for transfer descriptors, completion events, and bulk data. The driver hands the device a pointer to a transfer descriptor in host memory; the device fetches it via DMA, performs the transfer, writes the completion into an event ring, and raises an interrupt. USB is particularly interesting because each USB device's data goes through the host controller's DMA engine, so the driver for a USB NIC is actually two drivers stacked: the USB host controller does the DMA, and the USB device driver runs on top.

A **GPU** uses DMA for command buffers, texture uploads, and sometimes display scanout. A single frame may involve tens of megabytes of data moving between system RAM and GPU VRAM, orchestrated by the GPU's own DMA engine under the direction of the driver's command stream. FreeBSD's `drm-kmod` ports of Linux's DRM drivers use `bus_dma` to set up the command buffer mappings; the translation from Linux's DMA API is a thin layer because the abstractions are similar.

A **sound card** uses DMA for the audio buffer. The driver allocates a circular buffer, programs the device with its bus address and length, and the device reads samples in real time, wrapping around at the end. The driver refills the buffer ahead of the device's read pointer and relies on a position interrupt (or periodic timer) to schedule the refill. `sound(4)` uses `bus_dma` for this purpose; the pattern is a nice intermediate between single-transfer DMA and full scatter-gather rings.

Each of these examples follows the same abstract pattern: tag, map, memory, sync, trigger, complete, sync, consume, unload. The details differ; the shape is constant. Chapter 21 teaches the shape on a simulated device; later chapters apply the shape to the real subsystems.

21.2.5 Why Direct Memory Access Cannot Be Unrestricted

At first glance, “the device writes to memory” sounds straightforward. The device has an internal register; the register holds a bus address; the device writes to that address. Why does the driver need to do anything at all beyond telling the device an address?

Three hardware realities make unrestricted DMA unsafe, and those realities are what `bus_dma(9)` exists to hide.

Reality one: the device may not be able to reach all of host memory. Older PCI devices are 32-bit bus masters: they can only address the lower four gigabytes of bus address space. On a system with sixteen gigabytes of RAM, a buffer at physical address `0x4_0000_0000` is invisible to such a device; handing it that address would be a silent corruption (the write goes somewhere, but not where the driver expected). Some newer devices have 36-bit or 40-bit DMA engines and can reach more of RAM but not all of it. The driver has to describe the device's range to the kernel, and the kernel has to ensure every buffer the device sees is inside the range. When a buffer happens to be outside the range, the kernel silently inserts a **bounce buffer**: a region inside the range, into

which the kernel copies the data before the DMA read (or out of which it copies after the DMA write). Bounce buffers are correct but expensive; a driver that allocates its buffers inside the device's range avoids them.

Reality two: the bus address the device sees is not always the physical address the CPU uses. On modern amd64 systems with an IOMMU enabled (Intel VT-d, AMD-Vi), the memory controller inserts a translation layer between the device's bus address and the host's physical address. The device writes to bus address X; the IOMMU translates that to physical address Y; the memory controller writes physical address Y. The translation is per-device, and by default the IOMMU only permits a device to access memory the driver has explicitly mapped. This is a correctness and security win (a buggy or compromised device cannot write to arbitrary memory), but it requires the driver to participate: the driver tells the kernel "this buffer should be visible to this device", the kernel programs the IOMMU accordingly, and only then can the device reach the buffer. Without `bus_dma`, the driver would have to know whether an IOMMU is present, what its page tables look like, and how to program them.

Reality three: the CPU and the device may see the same memory differently at the same moment. Every CPU has caches; the caches hold copies of memory at the granularity of cache lines (typically sixty-four bytes on amd64, sometimes more on other platforms). When the CPU writes a value, the write goes into the cache first; the cache line is marked dirty, and the write only reaches the memory controller when the line is evicted or a coherence protocol synchronises it. When a DMA-capable device writes a value, the write goes directly to the memory controller (unless the platform is fully coherent, which amd64 typically is but ARM sometimes is not), so the CPU's cached copy may be stale. Conversely, when the device reads a value that the CPU has recently written and not flushed, the device reads a stale value. On fully coherent platforms, the hardware handles this automatically; on partially coherent or non-coherent platforms, the driver has to tell the kernel "the CPU is about to read this buffer, please ensure the caches are invalidated first" or "the CPU just wrote this buffer, please flush the caches before the device reads". This is what `bus_dmamap_sync` does.

These three realities are not visible to driver authors on amd64 with a modern IOMMU and a fully coherent device; the `bus_dma` API is still required, but most of its work is transparent. On 32-bit hardware, on non-coherent ARM platforms, or on systems where the IOMMU is configured restrictively, the API actually does something at every call. A driver written correctly against `bus_dma` works on all of them; a driver that bypasses the API works only on the subset of platforms where the naive model happens to be correct.

21.2.6 Why `bus_dma(9)` Exists

FreeBSD's `bus_dma(9)` interface is the portability layer that hides these three realities behind a single API. It was inherited and adapted from NetBSD's equivalent, and refined in FreeBSD over the 5.x and later releases. The design decisions that make it distinctive are worth understanding at the conceptual level before the code examples.

The API **separates description from execution**. A DMA **tag** describes the constraints of a group of transfers: what address range is reachable, what alignment is required, how large a transfer may be, how many segments it may span, what the boundary rules are. A DMA **map** represents one specific transfer's mapping onto that tag: which bus addresses the specific buffer occupies, which bounce buffers (if any) are in play. The driver creates the tag once at attach time; it creates or loads maps at transfer time. The separation lets the kernel cache expensive setup (parent tags, constraint checks) while keeping per-transfer work cheap.

The API **uses callbacks to hand back mapping information**. `bus_dmamap_load` does not return the list of bus addresses directly; it calls a callback function the driver provides, passing the segment list as an array. The

reason is historical and practical: on some platforms the load may need to wait for bounce buffers to become available, and in that case the callback runs later when the buffers are free. The driver's load code returns immediately with `EINPROGRESS`; the callback eventually runs and completes the mapping. This pattern is the one that trips most first-time readers, and Section 3 walks through it carefully. For simple cases (descriptor rings allocated at attach time, where the driver is willing to wait), the load completes synchronously and the callback runs before `bus_dmamap_load` returns.

The API **makes synchronisation explicit**. `bus_dmamap_sync` with a `PRE` flag says “the CPU is about to stop accessing this buffer and hand it to the device; please flush”. With a `POST` flag it says “the device has stopped accessing this buffer and the CPU is about to access it; please invalidate”. On coherent platforms, `bus_dmamap_sync` is sometimes a no-op; on non-coherent platforms, it runs the cache flush or invalidate. The driver writes the same code in both cases; the API handles the difference.

The API **supports hierarchies of constraints**. A tag can inherit from a parent tag, and the child's constraints are restrictions of the parent's. This matches the hardware: a PCI device's DMA capabilities are constrained by its parent bridge's capabilities, which are constrained by the platform's memory controller. `bus_get_dma_tag(9)` returns the device's parent tag, and the driver passes that to `bus_dma_tag_create` as the parent of any tag it creates. The kernel composes the constraints automatically; the driver describes only its own device's requirements.

These four design choices (separation, callbacks, explicit sync, hierarchies) are visible in every `bus_dma`-using driver, and the chapter's code follows them closely. The benefit of understanding the design is that real drivers become much easier to read; the six-thousand-line `nvme(4)` driver, for example, follows the same pattern as Chapter 21's toy driver.

21.2.7 The Concrete Flow Chapter 21 Will Build

Concretely, the Chapter 21 driver will learn to do the following sequence, in order:

1. **Create a parent-inheriting tag.** The driver calls `bus_get_dma_tag(sc->dev)` to obtain the device's parent tag and passes it to `bus_dma_tag_create` along with the myfirst device's own constraints (4 KB alignment, 4 KB buffer size, 1 segment, `BUS_SPACE_MAXADDR` for the address range because the simulation has no architectural limit).
2. **Allocate DMA memory.** The driver calls `bus_dmamem_alloc` with the tag. The kernel returns a kernel virtual address pointing at a four-kilobyte buffer that is both mapped for the CPU and suitable for the device. The call also returns a `bus_dmamap_t` handle that represents the mapping.
3. **Load the memory into the map.** The driver calls `bus_dmamap_load` with the kernel virtual address and a callback. The callback receives the segment list (one segment for this simple case) and stashes the bus address in the `softc`.
4. **Program the device.** The driver writes the bus address, the length, and the direction to the simulated DMA engine's registers.
5. **Sync PREWRITE (for host-to-device) or PREREAD (for device-to-host).** The driver calls `bus_dmamap_sync` with the appropriate flag, signalling that the CPU has finished touching the buffer and the device is about to access it.
6. **Kick the engine.** The driver writes the `DMA_CTRL` register's `START` bit; the simulated engine schedules a `callout(9)` a few milliseconds in the future.

7. **Wait for completion.** Via the Chapter 20 rx filter, which fires when the simulated engine raises DMA_COMPLETE. The filter enqueues the task; the task runs.
8. **Sync POSTREAD or POSTWRITE.** The task calls bus_dmamap_sync with the POST flag before accessing the buffer.
9. **Read and verify.** The task compares the buffer contents against the expected pattern and updates a softc counter.
10. **Tear down.** On detach, the driver unloads the map, frees the memory, destroys the tag. The order is the reverse of setup.

Every step maps to a single bus_dma call. The chapter's job is to teach each call in context, show how it fits into the driver's existing lifecycle, and explain what the kernel itself does on each call so the reader can debug when things go wrong. A driver that has internalised these ten steps can move on to descriptor rings, scatter-gather lists, and bus_dmamap_load_mbuf_sg without learning a new model; each of those is a variation on the same ten steps.

21.2.8 Exercise: Identify DMA-Capable Devices on Your System

Before the next section, take five minutes to look at your own system. The exercise is simple and builds intuition: identify three devices on your lab machine that use DMA, and note one property each.

Start with `pciconf -lv`. The tool lists every PCI function, and most of them are DMA-capable in some way. For each function, note what subsystem it belongs to (network, storage, graphics, audio, USB). Then look for the line starting with `cmdreg:` in `pciconf -c` output; if the bit that reads `BUSMASTEREN` is set, the device has bus-mastering enabled and is actively using DMA.

```
pciconf -lvc | grep -B1 BUSMASTEREN
```

Pick one network function and look at it more carefully:

```
pciconf -lvbc <devname>
```

`pciconf -lvbc` shows the BAR regions, the capability list, and whether the PCIe device's config space reports any DMA-relevant capabilities (MSI/MSI-X, power management, PCIe DevCtl, ASPM). On most modern systems the output reveals that the device has a large MMIO BAR (for register access) and smaller MMIO BARs (for MSI-X table and PBA), but no I/O ports; the bulk of the device's memory is in RAM, reached by DMA, not in the device's own BAR.

Then look at an nvme device if you have one, or at `dmesg` for "mapped DMA" messages. Most storage drivers log a brief DMA setup banner at attach time; `nvme_ctrlr_setup` is a good one to grep for.

Write down in a lab notebook:

1. One network device with bus-mastering enabled, and a guess at what it uses DMA for.
2. One storage device (if present) with its DMA setup banner.
3. One device you did not expect to use DMA, but does. A surprise finding pays off: once you know what to look for, the picture changes.

The exercise takes about ten minutes and gives you a map of your own lab target's DMA landscape before Chapter 21's work begins.

21.2.9 Wrapping Up Section 1

Section 1 established the hardware picture. DMA lets devices write to host memory directly, bypassing the CPU-by-word bottleneck of MMIO. The benefits are throughput, CPU offload, and determinism; the costs are the setup complexity and the need to synchronise between the CPU's cached view and the device's bus-visible view. FreeBSD's `bus_dma(9)` interface exists to hide three hardware realities (device addressing limits, IOMMU remapping, cache coherence) behind a single API, and the API's four design principles (tag/map separation, callback-based load, explicit sync, constraint inheritance) appear in every driver that uses it. Chapter 21's running example will exercise each principle in turn on a simulated device, with enough real FreeBSD grounding that the patterns carry over to production drivers unchanged.

Section 2 is the next step: the API's vocabulary in detail. What a tag actually contains, what a map actually is, what a segment is, what the PRE and POST flags mean at a finer level, and how the pieces fit together.

21.3 Section 2: Understanding FreeBSD's `bus_dma(9)` Interface

Section 1 established that `bus_dma(9)` is the portability layer between the driver and the platform's DMA realities. Section 2 opens the layer and looks at its pieces. The goal is to give the reader the vocabulary to talk about a DMA tag, a DMA map, a segment, a sync operation, and a callback without mystery. No Stage 1 code is written yet; Section 3 does that. This section is the mental map.

21.3.1 The Four Pieces of the API

Every `bus_dma` driver deals with four objects. Understand the four and the rest of the API falls into place.

The tag is a description. It is an opaque kernel object of type `bus_dma_tag_t`, created once and reused for many transfers. A tag carries:

- An optional parent tag, inherited from the parent bridge through `bus_get_dma_tag(dev)`.
- An alignment constraint in bytes (the starting address of every mapping made through this tag must be a multiple of this value).
- A boundary constraint in bytes (a mapping that crosses an address boundary of this size is not allowed).
- A low address and a high address that together describe a window of bus address space the device **cannot** reach.
- A maximum mapping size (the sum of segment lengths in one mapping).
- A maximum segment count (how many discontinuous pieces one mapping may span).
- A maximum segment size (the largest single piece).
- A set of flags, chiefly `BUS_DMA_WAITOK`, `BUS_DMA_NOWAIT`, and `BUS_DMA_COHERENT`.
- An optional lock function and its argument, used when the kernel needs to invoke the driver's load callback from a deferred context.

The tag is how the driver tells the kernel “this device has these constraints; please honour them on every mapping I make through this tag”. The kernel consults the tag during every map operation and ensures the mapping respects the constraints (or reports an error if the requested buffer cannot satisfy them).

The map is a mapping context. It is an opaque kernel object of type `bus_dmamap_t`, created (often implicitly) per transfer. A map carries enough state to describe one specific mapping: which bus addresses the buffer

occupies, whether any bounce pages are in use, whether the mapping is currently loaded or idle. Maps are cheap to create and cheap to load; the expensive setup work lives in the tag.

The memory is the kernel virtual address range the CPU uses to access the buffer. For static DMA regions (allocated at attach time, reused for many transfers), the memory is allocated by `bus_dmamem_alloc`, which returns a kernel virtual address and an implicitly-loaded map. For dynamic DMA (mapping arbitrary kernel buffers, mbufs, or user data), the memory already exists elsewhere and the driver uses `bus_dmamap_create` plus `bus_dmamap_load` to attach a map to it.

The segment is the bus-address-and-length pair the device actually sees. A `bus_dma_segment_t` is a small structure with two fields: `ds_addr` (a `bus_addr_t` giving the bus address) and `ds_len` (a `bus_size_t` giving the length). A single mapping may consist of one segment (physically contiguous) or several (scatter-gather). The driver programs the device with the segment list; that is the concrete hand-off.

Chapter 21's driver uses all four. The tag is created in `myfirst_dma_setup`. The map is returned by `bus_dmamem_alloc`. The memory is the four-kilobyte buffer returned by the same call. The segment is the one `bus_dma_segment_t` returned by the callback passed to `bus_dmamap_load`. Later chapters extend this to per-transfer maps and multi-segment scatter-gather, but the same four objects are always in play.

21.3.2 Static Versus Dynamic Transactions

The `bus_dma(9)` manual page introduces a distinction between **static** and **dynamic** transactions. The distinction matters because it shapes which API calls a driver uses.

Static transactions use memory regions allocated by `bus_dma` itself, typically at attach time, and reused for the driver's lifetime. Descriptor rings are the classic example: a NIC driver allocates the receive ring once and uses it for every packet, never unloading and reloading. The driver calls:

- `bus_dma_tag_create` once, to describe the ring's constraints.
- `bus_dmamem_alloc` once, to allocate the ring's memory and get an implicitly-loaded map.
- `bus_dmamap_load` once, to obtain the ring's bus address. (The manual page's wording is "an initial load operation is required to obtain the bus address"; this is an API quirk you just remember.)
- `bus_dmamap_sync` many times, around each use of the ring.

On teardown, the driver calls `bus_dmamap_unload`, `bus_dmamem_free`, and `bus_dma_tag_destroy`. No `bus_dmamap_create` or `bus_dmamap_destroy` is needed because `bus_dmamem_alloc` returned the map and `bus_dmamem_free` frees it.

Dynamic transactions use memory regions allocated by something else (an mbuf from `m_getcl`, a kernel buffer from `m_malloc`, a user page pinned by `vm_fault_quick_hold_pages`), and the driver maps them into the device's address space per transfer. A NIC driver that is transmitting packets does this for each outgoing packet: the packet's mbuf already exists, the driver maps it, programs the device, waits for transmission, unmmaps it. The driver calls:

- `bus_dma_tag_create` once, to describe the per-buffer constraints.
- `bus_dmamap_create` once per buffer slot, to get a map.
- `bus_dmamap_load_mbuf_sg` (or `bus_dmamap_load`) each time a packet is transmitted, to map the specific mbuf.
- `bus_dmamap_sync` around each use.

- `bus_dmamap_unload` after each transmission completes.
- `bus_dmamap_destroy` once per buffer slot, on detach.
- `bus_dma_tag_destroy` once, on detach.

Chapter 21's driver uses the static pattern: one buffer, allocated once, reused for every simulated DMA transfer. The dynamic pattern is introduced briefly in Section 7 as a contrast; Part 6's network chapter (Chapter 28) uses it in earnest.

Knowing which pattern a driver uses is the first thing to identify when reading DMA code. The call sequence is different, the teardown order is different, and the interpretation of the map is different. `bus_dmamem_alloc`'s "one static allocation" behaviour is what makes the static path shorter and simpler.

21.3.3 The Sync Discipline

Section 1 described `bus_dmamap_sync` in general terms; Section 2 is the place to pin down the four operations precisely, because the next section will assume this vocabulary.

The four operations are:

- `BUS_DMASYNC_PREREAD`. Called **before** the device writes the buffer (from the host's point of view, before the buffer is read by the host). Tells the kernel "the CPU has finished whatever it was doing with this buffer; the device is about to write into it". On non-coherent platforms, this invalidates the CPU's cache copy so a later read will see the device's writes. On coherent platforms, it is often a no-op.
- `BUS_DMASYNC_PWRITE`. Called **before** the device reads the buffer (from the host's point of view, before the buffer is written by the host). Tells the kernel "the CPU has just written the buffer; please flush any dirty cache lines so the device reads the current contents". On non-coherent platforms this is a cache flush; on coherent platforms it is often a memory barrier or no-op.
- `BUS_DMASYNC_POSTREAD`. Called **after** the device has written the buffer and **before** the CPU reads it. Tells the kernel "the device has finished writing; the CPU is about to read". On platforms with bounce buffers, this is the point at which the data is copied from the bounce region back to the driver's buffer.
- `BUS_DMASYNC_POSTWRITE`. Called **after** the device has read the buffer. Tells the kernel "the device is done reading; the CPU may reuse the buffer". Usually a no-op on coherent platforms; on systems with bounce buffers, this is when the bounce region may be released.

The names are worth internalising. "PRE" and "POST" refer to the DMA transaction: PRE is before, POST is after. "READ" and "WRITE" are from the **host's** perspective: READ means the host will read the result (the device writes), WRITE means the host has written what the device will read.

The pairs come together in the four common sequences:

- **Host-to-device (the driver sends data to the device):** write data → PWRITE → device reads → device done → POSTWRITE.
- **Device-to-host (the device sends data to the driver):** PREREAD → device writes → device done → POSTREAD → host reads.
- **Descriptor ring where the driver updates an entry, the device reads it, updates status, and the driver reads the status:** the driver writes the entry → PWRITE → device reads the entry → device updates status → device done → POSTREAD | POSTWRITE (combined flag) → host reads status.
- **Full ring shared between host and device:** at setup, PREREAD | PWRITE marks the entire ring as hand-off to device, with both directions of data flow open.

Real drivers use the combined flags often because descriptor rings are bidirectional. Chapter 21 starts with the simple one-direction cases and shows the combined flag in the descriptor-ring sketch at the end of Section 4.

21.3.4 The Load Callback Pattern

The most surprising part of `bus_dma` for a first-time reader is that `bus_dmamap_load` does not return the segment list directly. It takes a **callback function** as an argument, calls the callback with the segment list, and (normally) returns after the callback has run. Why this indirection?

The reason is that on platforms where the kernel may need to wait for bounce buffers, the load operation can be **deferred**. If bounce buffers are in short supply at the moment of the call, the kernel queues the request, returns `EINPROGRESS` to the caller, and runs the callback later when buffers are free. The driver has to be prepared for this case: the callback may run in a different context, possibly on a different thread, after the load call has already returned.

For most drivers on most platforms, the deferred case is rare. A driver that allocates its descriptor ring at attach time, on a system with enough bounce buffers and no IOMMU restriction, sees the callback run synchronously inside the load call, long before the load returns. The callback just stashes the bus address in a local variable that the driver reads immediately after the load returns.

A minimal callback looks like this:

```
static void
myfirst_dma_single_map(void *arg, bus_dma_segment_t *segs, int nseg, int error)
{
    bus_addr_t *addr = arg;

    if (error)
        return;

    KASSERT(nseg == 1, ("unexpected DMA segment count %d", nseg));
    *addr = segs[0].ds_addr;
}
```

The driver calls:

```
bus_addr_t bus_addr = 0;
int err = bus_dmamap_load(sc->dma_tag, sc->dma_map,
    sc->dma_vaddr, DMA_BUFFER_SIZE,
    myfirst_dma_single_map, &bus_addr, BUS_DMA_NOWAIT);
if (err != 0) {
    device_printf(sc->dev, "bus_dmamap_load failed: %d\n", err);
    return (err);
}
KASSERT(bus_addr != 0, ("single-segment callback did not set address"));
sc->dma_bus_addr = bus_addr;
```

The pattern is so common that many drivers define exactly this helper and name it similarly. `/usr/src/sys/dev/re/if_re.c` calls it `re_dma_map_addr`. `/usr/src/sys/dev/nvme/nvme_private.h` calls it `nvme_single_map`. Chapter 21’s driver will call it `myfirst_dma_single_map`. The function body is almost identical across drivers.

The `BUS_DMA_NOWAIT` flag on the load call tells the kernel “do not defer; if bounce buffers are unavailable, return `ENOMEM` immediately”. This is the common case for drivers that are willing to accept failure rather than wait. A driver that wants to wait passes `BUS_DMA_WAITOK` (or `0`, since `WAITOK` is the default zero value); in that case, if the load is deferred, the driver must return from the caller and expect the callback to run later.

When the load is deferred and runs later, the kernel may need to hold a driver-level lock while the callback runs. This is what the `lockfunc` and `lockfuncarg` parameters of `bus_dma_tag_create` are for: the driver provides a function that the kernel calls with `BUS_DMA_LOCK` before the callback and `BUS_DMA_UNLOCK` after. FreeBSD provides `busdma_lock_mutex` as a ready-made implementation for drivers that protect their DMA state with a standard mutex; the driver passes the mutex pointer as `lockfuncarg`. For drivers that never defer loads (because they use `BUS_DMA_NOWAIT` everywhere, or because they only load at attach time with no locks held), the kernel provides `_busdma_dflt_lock`, which panics if called; that is the safer default because it turns a silent threading bug into a loud one.

Chapter 21’s driver loads exactly once at attach time, under no contended lock, with `BUS_DMA_NOWAIT`. The `lockfunc` does not matter because the callback always runs synchronously. Section 3 passes `NULL` for both `lockfunc` and `lockfuncarg`; the kernel substitutes `_busdma_dflt_lock` automatically, and the panic-on-unexpected-call acts as a safety net.

21.3.5 Tag Hierarchies and Inheritance

Section 1 mentioned that a tag inherits from a parent tag, and that the hierarchy matches the hardware. Section 2 is the place to make this concrete.

When a PCI device is attached to a PCI bridge, the bridge’s own tag carries the constraints that the bridge itself imposes on DMA traffic: the DMA addresses it can route, the alignment it enforces, whether an IOMMU is in front of it. The kernel creates this tag at bus-attach time, and a driver can retrieve it for its own device with:

```
bus_dma_tag_t parent = bus_get_dma_tag(sc->dev);
```

The returned tag is the device’s **parent DMA tag**. When the driver creates its own tag, it passes the parent:

```
int err = bus_dma_tag_create(parent,
    /* alignment */    4,
    /* boundary */    0,
    /* lowaddr */    BUS_SPACE_MAXADDR,
    /* highaddr */    BUS_SPACE_MAXADDR,
    /* filtfunc */    NULL,
    /* filtfuncarg */ NULL,
    /* maxsize */    DMA_BUFFER_SIZE,
    /* nsegments */   1,
```

```

/* maxsegsz */ DMA_BUFFER_SIZE,
/* flags */ 0,
/* lockfunc */ NULL,
/* lockfuncarg */ NULL,
&sc->dma_tag);

```

The child inherits the parent’s restrictions; the child can only add more restrictions, never remove them. If the parent says “this bridge cannot address above 4 GB”, the child cannot say “actually my device can”; the kernel takes the intersection. This is what lets a driver write portable code: the driver describes only its device’s own constraints, and the kernel composes them with whatever the platform imposes.

Drivers that need to describe multiple groups of transfers (for example, one alignment for descriptor rings and a different alignment for data buffers) create multiple tags, all inheriting from the same parent. The `if_re(4)` driver is a clean example: it creates `rl_parent_tag` from the bridge, then `rl_tx_mtag` and `rl_rx_mtag` (for mbuf payloads, 1-byte aligned, multi-segment) and `rl_tx_list_tag` and `rl_rx_list_tag` (for descriptor rings, `RL_RING_ALIGN`-aligned, single-segment) all inherit from `rl_parent_tag`. Each tag describes a different set of buffers; the hierarchy composes them.

Chapter 21’s driver uses a single tag because it has a single buffer type. When the reader extends the driver with descriptor rings, the natural evolution is to add a second tag; Section 8’s challenge exercises sketch the step.

21.3.6 A Simple Hardware-to-Software Timeline

To solidify the vocabulary, here is a timeline for one complete DMA transfer on Chapter 21’s device, with each line annotated by which object and which API call is involved.

1. **Attach time, once.** Create tag via `bus_dma_tag_create`. Allocate memory and implicit map via `bus_dmamem_alloc`. Load map via `bus_dmamap_load` with single-segment callback. Record bus address in `softc`. Result: `sc->dma_tag`, `sc->dma_vaddr`, `sc->dma_map`, `sc->dma_bus_addr` all populated.
2. **First transfer, host-to-device:**
 - Fill `sc->dma_vaddr` with the pattern to send.
 - Call `bus_dmamap_sync(sc->dma_tag, sc->dma_map, BUS_DMASYNC_PREWRITE)`.
 - Write `DMA_ADDR_LOW`, `DMA_ADDR_HIGH`, `DMA_LEN`, `DMA_DIR=WRITE` via `bus_write_4` calls.
 - Write `DMA_CTRL = START`.
 - Wait for completion interrupt.
 - In the rx filter, read `DMA_STATUS`; if `DONE`, enqueue task.
 - In the task, call `bus_dmamap_sync(sc->dma_tag, sc->dma_map, BUS_DMASYNC_POSTWRITE)`.
 - Update statistics.
3. **Second transfer, device-to-host:**
 - Call `bus_dmamap_sync(..., BUS_DMASYNC_PREREAD)`.
 - Write `DMA_DIR=READ`, `DMA_CTRL=START`.
 - Wait for interrupt.

- In the task, call `bus_dmamap_sync( . . . , BUS_DMASYNC_POSTREAD)`.
- Read `sc->dma_vaddr` and verify the pattern.

4. **Detach time, once.** Unload map via `bus_dmamap_unload`. Free memory via `bus_dmamem_free` (which also frees the implicit map). Destroy tag via `bus_dma_tag_destroy`.

The timeline has four loops of sync: PREWRITE/POSTWRITE for host-to-device, PREREAD/POSTREAD for device-to-host. Each loop is one transfer. Each loop has exactly one sync call before and one after; neither may be omitted. Between the PRE sync and the POST sync, the CPU may not touch the buffer (the device owns it). Before the PRE sync and after the POST sync, the CPU owns the buffer freely.

The mental model “ownership passes between CPU and device, with sync calls marking each hand-off” is the single most useful intuition to take away from Section 2. Every real driver respects the ownership discipline, even on coherent platforms where the sync calls happen to be cheap or no-op.

21.3.7 Exercise: Walk Through a Sample DMA Setup Without a Real Device

A paper exercise before Section 3. Imagine a simple DMA-capable device with the following properties:

- PCI device with one MMIO BAR.
- One DMA engine with these registers: `DMA_ADDR_LOW` at offset 0x20, `DMA_ADDR_HIGH` at 0x24, `DMA_LEN` at 0x28, `DMA_DIR` at 0x2C (0 = host-to-device, 1 = device-to-host), `DMA_CTRL` at 0x30 (write 1 = start), `DMA_STATUS` at 0x34 (bit 0 = done, bit 1 = error).
- 4 KB transfer buffer required.
- 32-bit DMA engine (cannot reach memory above 4 GB).
- 16-byte alignment required.
- Buffer must not cross a 64 KB boundary.

Write out, on paper, the exact `bus_dma_tag_create` call the driver would issue. Identify each of the fourteen parameters, and name what concrete value you would use for each. Then write out the `bus_dmamem_alloc` call and the `bus_dmamap_load` call that follow. Do not write C; write prose.

A sample answer (for comparison):

- Parent tag: `bus_get_dma_tag(sc->dev)`, the PCI parent.
- Alignment: 16.
- Boundary: 65536.
- Low address: `BUS_SPACE_MAXADDR_32BIT`. Remember the confusing naming: `lowaddr` is the highest address the device can reach, not a lower bound. For a 32-bit engine, any address above `BUS_SPACE_MAXADDR_32BIT` must be excluded, so that value goes here.
- High address: `BUS_SPACE_MAXADDR`. This is the upper bound of the excluded window; passing `BUS_SPACE_MAXADDR` extends the exclusion to infinity, which correctly expresses “nothing above 4 GB is reachable”.
- Filter function: NULL (not needed, modern code avoids them).
- Filter argument: NULL.
- Max size: 4096.
- Nsegments: 1 (single-segment required).
- Max segment size: 4096.

- Flags: 0.
- Lockfunc: NULL.
- Lockfuncarg: NULL.
- Tag pointer: &sc->dma_tag.

Then `bus_dmamem_alloc` with `BUS_DMA_WAITOK` | `BUS_DMA_COHERENT` | `BUS_DMA_ZERO` to allocate the buffer as coherent, zero-initialised, sleeping allowed (attach time). Then `bus_dmamap_load` with `BUS_DMA_NOWAIT`, the single-segment callback, and `&sc->dma_bus_addr` as the callback argument.

Writing this out on paper before seeing the code makes Section 3's walkthrough feel like a confirmation rather than an introduction. The exercise takes ten minutes and is worth it.

21.3.8 Wrapping Up Section 2

Section 2 gave the reader the vocabulary: tag, map, memory, segment, PRE, POST, READ, WRITE, static, dynamic, parent, callback. The pieces fit together in a specific pattern: at attach time, create a tag, allocate memory, load a map, record the bus address; at transfer time, sync PRE, trigger the device, wait, sync POST, read the data; at detach time, unload the map, free the memory, destroy the tag. The pattern is the same across every DMA-capable driver in FreeBSD; the vocabulary is the same in every chapter of this book from now on.

Section 3 is the first running code. The driver creates a tag, allocates memory, loads it, and verifies that the bus address is sane. No transfers happen yet; the goal is to exercise the setup and teardown paths under WITNESS and confirm that the driver can stand up and tear down a DMA region cleanly across many `kldload`/`kldunload` cycles. That pair of operations is the foundation that every later section adds to.

21.4 Section 3: Allocating and Mapping DMA Memory

Section 2 built the vocabulary. Section 3 writes the code. The first running stage of the Chapter 21 driver creates a DMA tag, allocates a single four-kilobyte buffer, loads it into a map, records the bus address, and adds the setup-plus-teardown path to the attach and detach sequences. No DMA transfers happen yet. The goal is to exercise the allocation path, watch the new `softc` fields populate under `kldload`, confirm that `kldunload` cleans up without complaint, and do so repeatedly under WITNESS to catch any lock-order violations.

Stage 1's version tag is `1.4-dma-stage1`. The compile target after the stage is `myfirst.ko`. No new files are added yet; the tag creation, allocation, and teardown live in `myfirst.c` and `myfirst_pci.c` alongside the existing attach logic. Section 8 moves the DMA code into its own file; here we keep everything in one place so the stage's scope is visible in a single view.

21.4.1 New Softc Fields

The driver's `softc` grows by four fields:

```
/* In myfirst.h, inside struct myfirst_softc. */
bus_dma_tag_t      dma_tag;
bus_dmamap_t       dma_map;
```

```
void                *dma_vaddr;
bus_addr_t         dma_bus_addr;
```

`dma_tag` is the tag created at attach time. `dma_map` is the map returned by `bus_dmamem_alloc`. `dma_vaddr` is the kernel virtual address the CPU uses to access the buffer. `dma_bus_addr` is the bus address the device uses.

The fields follow the Chapter 17/18 naming convention: lowercase, underscores, short. They are initialised to zero by the softc's implicit zeroing at attach time; the driver adds the initialisation by relying on `device_get_softc`'s zero-fill guarantee. No explicit `= 0` assignments are needed.

The softc includes `<machine/bus.h>` and `<sys/bus_dma.h>` (the latter through `<machine/bus.h>`); Chapter 18 already pulled these in for `bus_space` work, so the includes are present and no new ones are required.

21.4.2 The Single-Segment Callback

Before the setup function, a one-function helper. The helper is the single-segment callback introduced in Section 2:

```
static void
myfirst_dma_single_map(void *arg, bus_dma_segment_t *segs, int nseg, int error)
{
    bus_addr_t *addr = arg;

    if (error != 0) {
        printf("myfirst: dma load callback error %d\n", error);
        return;
    }

    KASSERT(nseg == 1, ("myfirst: unexpected DMA segment count %d", nseg));
    *addr = segs[0].ds_addr;
}
```

The function is the same pattern as `re_dma_map_addr` in `/usr/src/sys/dev/re/if_re.c`. Pass it a pointer to a `bus_addr_t`; on success, the callback writes the one segment's bus address into the target. The `KASSERT` confirms there is exactly one segment; `bus_dmamem_alloc` always returns a single segment, so this assertion is a safety net rather than a runtime check.

The callback is marked `static` because it is not used outside `myfirst.c`. It takes no state beyond the bus-address pointer, so it is thread-safe by construction and can run in any context the kernel chooses.

21.4.3 The Setup Function

The Stage 1 setup function, named `myfirst_dma_setup`, lives in `myfirst.c`. It performs the four operations in order: create tag, allocate memory, load map, record bus address. Each step checks for error and unwinds

prior steps on failure. The function returns 0 on success, a nonzero error code on failure, and leaves the softc's DMA fields in a consistent state in both cases.

```
#define MYFIRST_DMA_BUFFER_SIZE 4096u

int
myfirst_dma_setup(struct myfirst_softc *sc)
{
    int err;

    err = bus_dma_tag_create(bus_get_dma_tag(sc->dev),
        /* alignment */      4,
        /* boundary */      0,
        /* lowaddr */       BUS_SPACE_MAXADDR,
        /* highaddr */      BUS_SPACE_MAXADDR,
        /* filtfunc */      NULL,
        /* filtfuncarg */   NULL,
        /* maxsize */       MYFIRST_DMA_BUFFER_SIZE,
        /* nsegments */     1,
        /* maxsegsz */      MYFIRST_DMA_BUFFER_SIZE,
        /* flags */         0,
        /* lockfunc */      NULL,
        /* lockfuncarg */   NULL,
        &sc->dma_tag);
    if (err != 0) {
        device_printf(sc->dev, "bus_dma_tag_create failed: %d\n", err);
        return (err);
    }

    err = bus_dmamem_alloc(sc->dma_tag, &sc->dma_vaddr,
        BUS_DMA_WAITOK | BUS_DMA_COHERENT | BUS_DMA_ZERO,
        &sc->dma_map);
    if (err != 0) {
        device_printf(sc->dev, "bus_dmamem_alloc failed: %d\n", err);
        bus_dma_tag_destroy(sc->dma_tag);
        sc->dma_tag = NULL;
        return (err);
    }

    sc->dma_bus_addr = 0;
    err = bus_dmamap_load(sc->dma_tag, sc->dma_map,
        sc->dma_vaddr, MYFIRST_DMA_BUFFER_SIZE,
        myfirst_dma_single_map, &sc->dma_bus_addr,
        BUS_DMA_NOWAIT);
    if (err != 0 || sc->dma_bus_addr == 0) {
```



```

    device_printf(sc->dev, "bus_dmamap_load failed: %d\n", err);
    bus_dmamem_free(sc->dma_tag, sc->dma_vaddr, sc->dma_map);
    sc->dma_vaddr = NULL;
    bus_dma_tag_destroy(sc->dma_tag);
    sc->dma_tag = NULL;
    return (err != 0 ? err : ENOMEM);
}

device_printf(sc->dev,
    "DMA buffer %zu bytes at KVA %p bus addr %#jx\n",
    (size_t)MYFIRST_DMA_BUFFER_SIZE,
    sc->dma_vaddr, (uintmax_t)sc->dma_bus_addr);

return (0);
}

```

A few details repay attention.

The constants. MYFIRST_DMA_BUFFER_SIZE is a #define named with the chapter's prefix, following the Chapter 17 naming convention. It is used three times in the setup function (in the tag's maxsize, the tag's maxsegsz, and the load's buffer length), so a symbolic name is worth it even if the value is small. Chapter 21 uses 4 KB because it matches a page and avoids any bounce-buffer corner cases; Section 7 revisits the choice.

The parent tag. bus_get_dma_tag(sc->dev) returns the parent DMA tag inherited from the PCI bridge. The kernel composes the parent's restrictions with the driver's own, so the child tag automatically respects any bridge-level limits. On amd64 with an IOMMU, the parent tag also carries the IOMMU's mapping requirements; the driver's tag inherits them transparently.

The alignment. 4 bytes is enough for a 32-bit-wide engine's DMA start address. A real device's datasheet would specify a value (NVMe uses the page size, if_re uses the descriptor size); Chapter 21's simulated engine happens to be content with 4.

The boundary. 0 means no boundary restriction. A device that cannot cross 64 KB boundaries (some DMA engines have this limitation because their internal counters are 16 bits) would pass 65536.

The addresses. BUS_SPACE_MAXADDR for both low and high means "no exclusion window". The simulation has no address-range limit; a real 32-bit-only device would pass BUS_SPACE_MAXADDR_32BIT for lowaddr to exclude addresses above 4 GB. Section 7 revisits this with a concrete walkthrough.

The size and segment count. A single 4 KB buffer that must be physically contiguous. nsegments = 1 plus maxsegsz = MYFIRST_DMA_BUFFER_SIZE means a one-segment mapping; bus_dmamem_alloc always returns a single segment, so this matches.

The flags. 0 at tag-create time (no special tag flags). BUS_DMA_WAITOK | BUS_DMA_COHERENT | BUS_DMA_ZERO at bus_dmamem_alloc time: the allocator may sleep if needed, the memory should be cache-coherent with the device (important on arm and arm64; a hint on amd64), and the buffer should be zero-initialised. BUS_DMA_NOWAIT at bus_dmamap_load time: the load must not defer; if bounce buffers are unavailable, the load should fail immediately. For Chapter 21's one-load-at-attach-time usage, the distinction

does not matter in practice; the convention is to use `NOWAIT` anywhere the driver has a clean error path, because it makes the control flow easier to reason about.

The lockfunc. `NULL` means “default”. On Chapter 21’s single load at attach time, the callback always runs synchronously, so the lockfunc is never invoked; the default (`_busdma_dflt_lock`, which panics if called) is the correct safety net.

The error handling. Each step checks the return value and unwinds prior steps on failure. The order matters: tag was created first, so it is destroyed last; memory was allocated second, so it is freed second-to-last. The softc fields are cleared to `NULL` after free so detach can safely walk them whether setup partially failed or ran to completion.

The final device_printf. A single banner line logs the buffer size, kernel virtual address, and bus address. The line is useful in `dmesg` and doubles as a regression-test marker: a working driver prints it; a broken setup does not.

21.4.4 The Teardown Function

The paired teardown function is:

```
void
myfirst_dma_teardown(struct myfirst_softc *sc)
{
    if (sc->dma_bus_addr != 0) {
        bus_dmamap_unload(sc->dma_tag, sc->dma_map);
        sc->dma_bus_addr = 0;
    }
    if (sc->dma_vaddr != NULL) {
        bus_dmamem_free(sc->dma_tag, sc->dma_vaddr, sc->dma_map);
        sc->dma_vaddr = NULL;
        sc->dma_map = NULL;
    }
    if (sc->dma_tag != NULL) {
        bus_dma_tag_destroy(sc->dma_tag);
        sc->dma_tag = NULL;
    }
}
```

The function is guarded by zero-value checks so it is safe to call even when setup did not complete. This is the same pattern the Chapter 19 interrupt teardown uses, and for the same reason: detach may run after a partial attach failure, and the cleanup code must not assume any particular prefix of the setup succeeded.

The order is the reverse of setup:

1. Unload the map. Safe to call even if the load succeeded; idempotent.
2. Free the memory and the implicit map. `bus_dmamem_free` frees both the buffer and the map object returned by `bus_dmamem_alloc`; the driver never calls `bus_dmamap_destroy` for this path.

3. Destroy the tag. Returns EBUSY if any maps are still associated; the earlier unload prevents that.

The `bus_dmamap_unload` call is subtle. After `bus_dmamem_alloc`, the map is implicitly loaded (this is what the manual page calls “An initial load operation is required to obtain the bus address”). The load either happens implicitly or via the explicit `bus_dmamap_load` call the driver already made. Either way, the map is in the loaded state after setup; it must be unloaded before `bus_dmamem_free`. The chapter’s pattern uses an explicit `bus_dmamap_load` to expose the callback and the bus address; `bus_dmamap_unload` is its paired teardown.

A reader who is curious about why `bus_dmamap_load` has to be called manually after `bus_dmamem_alloc` can read the answer in the `bus_dma(9)` manual page: the “STATIC TRANSACTIONS” section explicitly states “An initial load operation is required to obtain the bus address of the allocated memory”. The allocator returns a valid KVA and a map object, but the bus address itself is only made available via the callback after the load. This is a historical quirk; you remember it once and the pattern falls into place for every static driver.

21.4.5 Integrating Setup and Teardown Into Attach and Detach

The attach function from Chapter 20 ended with the MSI-X setup. Chapter 21 inserts the DMA setup just before it, so the tag and buffer are ready by the time the device starts firing interrupts:

```
/* ... earlier attach work: BAR allocation, hw layer, cdev ... */

err = myfirst_dma_setup(sc);
if (err != 0)
    goto fail_dma;

err = myfirst_msix_setup(sc);
if (err != 0)
    goto fail_msix;

/* ... remaining attach work ... */

return (0);

fail_msix:
    myfirst_dma_teardown(sc);
fail_dma:
    /* ... earlier failure labels ... */
```

The exact labels depend on the driver’s current layout, which is the Chapter 20 layout after Stage 4. The principle is: the DMA setup comes after the BAR is mapped (because the device’s registers must be reachable to eventually trigger transfers), after the hardware layer is built, but before the interrupt setup runs. On failure, the driver unwinds to the labels in reverse.

The detach function mirrors the attach order. The MSI-X teardown runs first (to stop any further interrupts), then the DMA teardown runs, then the earlier cleanup. The ordering guarantees that no interrupt can fire on the DMA path after the tag is destroyed, which would be a use-after-free.

21.4.6 Running Stage 1

Build and load:

```
make clean && make  
kldload ./myfirst.ko
```

Watch dmesg:

```
myfirst0: <myfirst DMA test device> ...  
myfirst0: DMA buffer 4096 bytes at KVA 0xfffffe0010af9000 bus addr 0x10af9000  
myfirst0: interrupt mode: MSI-X, 3 vectors
```

The KVA and bus address will differ on your machine; the format is what matters. The KVA is a typical kernel virtual address (well above the direct-map base on amd64). The bus address on amd64 with an IOMMU disabled is usually equal to the physical address; with an IOMMU enabled, it is an IOMMU-remapped address, often much lower than any physical address that could back it.

Verify the `softc` fields via `sysctl` or a `device_printf` in a debug build:

```
sysctl dev.myfirst.0
```

Unload the module:

```
kldunload myfirst
```

`dmesg` should show the detach:

```
myfirst0: detaching
```

No panic, no WITNESS complaint, no VFS warning about leaking memory. If WITNESS is enabled and the driver's setup or teardown acquires a lock that `bus_dma` also acquires, the output would show a lock-order warning; a clean `kldunload` is the confirmation that the locking is right.

Repeat the load/unload cycle several times. DMA-related memory leaks are rare but catastrophic when they happen (a leak in `bus_dma` leaks pages, not just bytes); the regression check is worth running fifty times in a row with a `for` loop and confirming that `vmstat -m` shows `bus_dmamap` and `bus_dma_tag` returning to their idle counts.

21.4.7 What Stage 1 Does and Does Not Do

Stage 1 exercises the allocation and cleanup paths. It does not perform any DMA transfers; `dma_vaddr` is an unused buffer (apart from its zero-fill), and `dma_bus_addr` is an unused address. The next sections will put both to work.

What Stage 1 proves is that:

- The driver can create a tag that inherits from the PCI parent.
- The driver can allocate a coherent, zero-initialised 4 KB buffer.
- The driver can obtain the bus address via the callback.
- The driver can tear everything down cleanly.
- The cycle can run many times without leaking resources.

That is a solid foundation. The rest of the chapter builds on it.

21.4.8 Common Mistakes at This Stage

Five mistakes are common in Stage 1. Each is easy to make and easy to fix once recognised.

Mistake one: forgetting to call `bus_dmamap_load` after `bus_dmamem_alloc`. The symptom is that `dma_bus_addr` stays at zero; programming the device with bus address zero either silently corrupts memory or hits an IOMMU fault. The fix is to call `bus_dmamap_load` explicitly, as the Stage 1 code does. The `KASSERT(bus_addr != 0, ...)` in the driver catches this during development.

Mistake two: passing an unaligned `maxsize` to `bus_dma_tag_create`. The alignment parameter must be a power of two, and `maxsize` must be at least alignment. A size of 3 bytes with alignment 4 produces an error. The chapter's 4096-byte size with alignment 4 is safely aligned.

Mistake three: using `BUS_DMA_NOWAIT` with `bus_dmamem_alloc` and then ignoring the error. `BUS_DMA_NOWAIT` makes the allocator return `ENOMEM` if memory is tight; a driver that ignores the error and proceeds with `NULL` dereferences panics immediately. Use `BUS_DMA_WAITOK` at attach time (where sleeping is safe), and always check the return value.

Mistake four: forgetting to zero the bus address before the load call. If the load fails, the callback is never called, and `dma_bus_addr` retains whatever garbage was there. The chapter's code assigns `sc->dma_bus_addr = 0` before calling `bus_dmamap_load` so the post-load check (`if (err != 0 || sc->dma_bus_addr == 0)`) is reliable.

Mistake five: skipping the tag destroy in the error-path unwind. The unwind in the chapter's code destroys the tag in every failure branch. A driver that fails `bus_dmamem_alloc` and returns without destroying the tag leaks a tag on every failed attach; over many retries this adds up. The pattern is: on any failure after `bus_dma_tag_create` succeeded, destroy the tag before returning.

None of these mistakes panic immediately; all of them produce subtle bugs that manifest under load or after repeated load/unload cycles. Running Stage 1 fifty times with `WITNESS` and `INVARIANTS` enabled catches most of them.

21.4.9 A Word on the `bus_dma` Template API

FreeBSD 13 and later also provide a template-based API for creating DMA tags, covered in the `bus_dma(9)` manual under `bus_dma_template_*`. The template API is an ergonomic alternative to `bus_dma_tag_create`'s fourteen-parameter list: the driver initialises a template from a parent tag, overrides individual fields with `BD_*` macros, and builds the tag with `bus_dma_template_tag`.

A template-based Stage 1 setup would look like:

```
bus_dma_template_t t;

bus_dma_template_init(&t, bus_get_dma_tag(sc->dev));
BUS_DMA_TEMPLATE_FILL(&t,
    BD_ALIGNMENT(4),
    BD_MAXSIZE(MYFIRST_DMA_BUFFER_SIZE),
    BD_NSEGMENTS(1),
    BD_MAXSEGSIZE(MYFIRST_DMA_BUFFER_SIZE));
err = bus_dma_template_tag(&t, &sc->dma_tag);
```

The template API is preferred in new drivers because it makes the overridden fields explicit. Chapter 21 uses the classic `bus_dma_tag_create` in its running example because that is what the majority of the existing FreeBSD source tree uses, and the reader who can read a fourteen-parameter call can always translate it to the template form. Section 8's refactor exercise suggests trying the template API as a stylistic variation.

21.4.10 Wrapping Up Section 3

Section 3 put the Chapter 21 driver into Stage 1: a tag, a map, a buffer, a bus address, and clean teardown. No transfers happen yet; the allocation path is the only thing exercised. That is enough to confirm that the pieces fit together, that the parent tag is inherited correctly, that the single-segment callback populates the bus address, and that `kldload/kldunload` cycles do not leak.

Section 4 is the sync discipline. Before the driver hands the buffer to a DMA engine, it needs to know exactly which sync calls happen where, why each one exists, and what would go wrong if one were omitted. Section 4 is a careful tour of `bus_dmamap_sync`, with concrete diagrams and the mental model that Section 5 then uses to build the simulated engine.

21.5 Section 4: Synchronising and Using DMA Buffers

Section 3 produced a driver that can allocate and release a DMA buffer. Section 4 teaches the discipline that every actual transfer must respect: when to call `bus_dmamap_sync`, with which flag, and why. The discipline is what keeps the CPU's and the device's view of memory consistent. No new code is written in this section (the simulated engine that will consume the sync calls is built in Section 5), but the mental model laid down here is what every later section depends on.

21.5.1 The Ownership Model

The single most useful intuition for DMA is this: at any moment, either the CPU owns the buffer or the device owns the buffer. Ownership passes back and forth through `bus_dmamap_sync` calls. Between a PRE sync and the next POST sync, the device owns the buffer and the CPU must not touch it. Between a POST sync and the next PRE sync, the CPU owns the buffer and the device must not touch it. The driver's job is to make every ownership boundary explicit with a sync call.

The ownership is physical, not logical. It does not mean the CPU or the device “knows” about the buffer in some abstract sense; it means the CPU’s caches and the device’s bus engine can actually observe different contents for the same memory at the same moment, and the sync calls are the kernel’s opportunity to reconcile the difference. On a fully coherent platform (modern amd64 with a coherent PCIe root complex and no non-coherent DMA peers), the hardware reconciles the difference transparently and `bus_dmamap_sync` is often a memory barrier or a no-op. On a partially coherent platform (some arm64 systems, some RISC-V configurations, anything with `BUS_DMA_COHERENT` not honoured at allocation), the sync calls do cache flushes and invalidates. The driver writes the same code regardless; the kernel does the right thing.

A driver that respects the ownership boundaries is portable. A driver that skips sync calls “because my amd64 system happens to work without them” is not; the same driver will corrupt data on arm64 or in a virtualised environment where the hypervisor’s IOMMU imposes its own coherence policy.

21.5.2 The Four Sync Flags in Depth

Section 2 named the four sync flags. Section 4 pins down exactly what each one means, what the kernel does for each on different platforms, and when the driver calls each in practice.

BUS_DMASYNC_PREREAD. The CPU has finished touching the buffer and the device is about to read from host memory is **not** the intent here; the flag is about **device writes into host memory**, which the host then **reads**. The flag is called `PREREAD` because the *host* is going to read after the operation; it is called `PRE` because this is before the DMA. On non-coherent platforms, the kernel invalidates the CPU’s cache copies of the buffer’s cache lines. The reason is that after the device writes new data to memory, stale cache lines in the CPU would cause subsequent reads to see the old values. Invalidation ensures the next read fetches from memory, where the device’s writes will land. On coherent platforms, invalidation is unnecessary because the hardware snoops the cache automatically; the kernel implements `PREREAD` as a memory barrier or no-op.

BUS_DMASYNC_PREWRITE. The CPU has just written the buffer and the device is about to read it. On non-coherent platforms, the kernel flushes the CPU’s dirty cache lines to memory. The reason is that the CPU’s writes might still sit in its cache; the device reads from memory directly and would see stale contents. Flushing ensures the memory controller has the current data before the device’s read. On coherent platforms, a memory barrier is usually sufficient to enforce ordering; the cache coherence protocol handles the flush implicitly.

BUS_DMASYNC_POSTREAD. The device has finished writing to host memory and the CPU is about to read. On non-coherent platforms, this is often the actual cache invalidate (in some implementations the invalidate happens at `POST` rather than `PRE`); on coherent platforms, it is a barrier or no-op. On platforms with bounce buffers, this is when the kernel copies data from the bounce region back into the driver’s buffer. That is why even on coherent platforms the `POSTREAD` call cannot be skipped: the coherence machinery may be coherent, but the bounce buffer machinery is not, and the `POSTREAD` call is the hook the bounce machinery uses to do its work.

BUS_DMASYNC_POSTWRITE. The device has finished reading from host memory. On most platforms this is a no-op because no host-visible state needs to change (the CPU already had the data it wrote; the device has finished reading and will not read again; nothing is out of sync). On bounce-buffer platforms, this is the hook where the bounce region may be released back to the pool.

Three observations that surface after the definitions:

- The `PRE` and `POST` of a pair are always both called. `PREREAD` is followed later by `POSTREAD`; `PREWRITE` is followed later by `POSTWRITE`. Skipping either breaks the contract.

- On coherent platforms many of these calls are cheap or free. That is an implementation detail the driver does not rely on. The driver writes them because they are required for portability and because bounce-buffer support is transparent only if the calls are there.
- The semantics of PRE and POST are “from the device’s viewpoint”: PRE means “before the device operation starts”; POST means “after the device operation completes”. READ and WRITE are “from the host’s viewpoint”: READ means “the host is going to read the result”; WRITE means “the host wrote the content”.

21.5.3 The Four Common Transfer Patterns

Four patterns cover almost every DMA transfer a driver performs. Each has its own PRE/POST sequence, and the sequence is predictable once the pattern is recognised.

Pattern A: Host-to-device data transfer. The driver fills a buffer with data and asks the device to send it.

```
CPU fills buffer at KVA
bus_dmamap_sync(..., BUS_DMASYNC_PREWRITE)
program device with bus_addr, length, direction
write doorbell
wait for completion
(device has read the buffer)
bus_dmamap_sync(..., BUS_DMASYNC_POSTWRITE)
CPU may now reuse buffer
```

Example: a driver transmitting a packet to a NIC. The driver copies the packet into a DMA buffer, issues the PREWRITE sync, writes the descriptor, pokes the doorbell, and waits for the transmit-complete interrupt. In the interrupt’s task, the driver issues the POSTWRITE sync and returns the buffer to the free pool.

Pattern B: Device-to-host data transfer. The driver asks the device to deliver data into a buffer; the driver then reads the buffer.

```
bus_dmamap_sync(..., BUS_DMASYNC_PREREAD)
program device with bus_addr, length, direction
write doorbell
wait for completion
(device has written the buffer)
bus_dmamap_sync(..., BUS_DMASYNC_POSTREAD)
CPU reads buffer at KVA
```

Example: a driver receiving a packet from a NIC. The driver hands the NIC a buffer’s bus address, issues the PREREAD sync, pokes the doorbell, and waits for the receive-complete interrupt. In the interrupt’s task, the driver issues the POSTREAD sync and processes the received data.

Pattern C: Descriptor ring entry. The driver writes a descriptor that the device reads (sees a new request), and the device later updates the same descriptor with a status field that the driver reads.


```

CPU writes descriptor fields (bus_addr, length, status = PENDING)
bus_dmamap_sync(..., BUS_DMASYNC_PREWRITE)
device reads descriptor
device processes the transfer
device writes descriptor status = DONE
bus_dmamap_sync(..., BUS_DMASYNC_POSTREAD | BUS_DMASYNC_POSTWRITE)
CPU reads descriptor status

```

The combined POST flag is the idiomatic way to pair the host-write-then-host-read when the host is both the producer and the consumer of different fields in the same memory region. The kernel does both operations in one call.

Pattern D: Bidirectional shared ring. The driver and the device share a descriptor ring indefinitely. At ring setup, both directions of data flow need to be synchronised.

```

bus_dmamem_alloc returns the ring's memory
bus_dmamap_sync(..., BUS_DMASYNC_PREREAD | BUS_DMASYNC_PREWRITE)
(ring is now live; both sides may read and write)
for each ring use:
    bus_dmamap_sync with the appropriate PRE
    doorbell / completion
    bus_dmamap_sync with the appropriate POST

```

Production drivers like `if_re(4)` and `nvme(4)` use this pattern; the combined PRE sync at ring setup is followed by per-transaction PRE/POST pairs. Chapter 21's single-buffer driver does not use this pattern; it is mentioned for completeness and because the reader will see it in real driver code.

21.5.4 The Simulated Transfer's Sync Sequence

Chapter 21's simulated DMA engine will run both host-to-device and device-to-host transfers. The test sequence the driver performs in Stage 2 is:

```

/* Host-to-device transfer: driver writes a pattern, engine reads it. */
memset(sc->dma_vaddr, 0xAA, MYFIRST_DMA_BUFFER_SIZE);
bus_dmamap_sync(sc->dma_tag, sc->dma_map, BUS_DMASYNC_PREWRITE);
/* program engine */
CSR_WRITE_4(sc, MYFIRST_REG_DMA_DIR, MYFIRST_DMA_DIR_WRITE);
CSR_WRITE_4(sc, MYFIRST_REG_DMA_CTRL, MYFIRST_DMA_CTRL_START);
/* wait for completion interrupt */
bus_dmamap_sync(sc->dma_tag, sc->dma_map, BUS_DMASYNC_POSTWRITE);

/* Device-to-host transfer: engine writes a pattern, driver reads it. */
bus_dmamap_sync(sc->dma_tag, sc->dma_map, BUS_DMASYNC_PREREAD);
CSR_WRITE_4(sc, MYFIRST_REG_DMA_DIR, MYFIRST_DMA_DIR_READ);

```

```
CSR_WRITE_4(sc, MYFIRST_REG_DMA_CTRL, MYFIRST_DMA_CTRL_START);
/* wait for completion interrupt */
bus_dmamap_sync(sc->dma_tag, sc->dma_map, BUS_DMASYNC_POSTREAD);
/* read sc->dma_vaddr and verify */
```

Two transfers, four sync calls, four flags. The order is the test’s contract. Section 5 will thread this through the driver’s code.

21.5.5 What Coherence Flags Mean and Do Not Mean

BUS_DMA_COHERENT is a flag both `bus_dma_tag_create` and `bus_dmamem_alloc` accept. Its meaning differs subtly between the two, and the difference is worth understanding so the reader knows when to use which.

On `bus_dma_tag_create`, BUS_DMA_COHERENT is an **architectural hint** that the driver is willing to spend extra cost at allocation to reduce cost at sync. On arm64, the flag tells the allocator to place the buffer in a memory domain that is inherently coherent with the DMA path, which may be a separate reserved region.

On `bus_dmamem_alloc`, BUS_DMA_COHERENT tells the allocator to produce a buffer that does not need cache flushes for sync. On arm and arm64, this uses a different allocator path (uncached or write-combined memory). On amd64, where the PCIe root complex is coherent and DMA is automatically snooped, the flag has little effect but is still passed for portability.

The rule: always pass BUS_DMA_COHERENT at allocation for descriptor rings and small control structures that are on the hot path for both the CPU and the device. Do not rely on the flag to remove the need for `bus_dmamap_sync`; the sync is still required by the API contract even if the flag makes it cheap. Some arm64 implementations do require the sync even with BUS_DMA_COHERENT (the flag is called a “hint” in the manual page for a reason); the driver is portable only if the sync is always called.

For bulk data buffers (packet payloads, storage blocks), BUS_DMA_COHERENT is usually not passed because the cache-hit rate is high when the CPU is about to process the data anyway, and the coherent allocator may use a slower memory domain. Chapter 21 passes BUS_DMA_COHERENT on its single small buffer because the chapter’s goal is to show the common case; a production driver would make the decision per-buffer based on access pattern.

21.5.6 What Happens If a Sync Is Omitted

A concrete failure scenario clarifies the abstraction. Suppose the driver performs a host-to-device transfer but forgets the PREWRITE sync:

```
memset(sc->dma_vaddr, 0xAA, MYFIRST_DMA_BUFFER_SIZE);
/* MISSING: bus_dmamap_sync(..., BUS_DMASYNC_PREWRITE); */
CSR_WRITE_4(sc, MYFIRST_REG_DMA_CTRL, MYFIRST_DMA_CTRL_START);
```

On amd64 with a coherent root complex, the transfer succeeds: the hardware snoops the CPU’s cache and the device reads the current contents. The bug is invisible.

On arm64 without a coherent PCIe fabric (some embedded systems), the transfer reads stale data: the CPU's write is still in the cache, the memory controller has old data, and the device reads 0x00 or garbage instead of 0xAA. The bug is visible immediately as data corruption.

On amd64 with bounce buffers (because the device had a 32-bit address limit and the buffer happened to be above 4 GB), the transfer reads stale data from the bounce region: the bounce machinery copies the driver's buffer into the bounce region at PREWRITE time, but without the PREWRITE call the bounce region is uninitialised (zero, or stale from a previous transfer). The bug is visible immediately as data corruption.

The pattern is consistent: the omission makes the driver work on the platform the author tested and fail on platforms with different coherence or addressing semantics. The sync calls are the portability guarantee. A driver that passes its test suite on amd64 and never has the sync calls exercised has a latent bug waiting for a different target.

21.5.7 Lock Context and Sync Calls

The Chapter 19/20 lock discipline extends naturally to sync calls. The sync itself does not acquire any lock inside `bus_dma`; it is safe to call from any context as long as the caller has the appropriate concurrency discipline for the map.

The relevant rules:

- The same map must not be touched by two threads simultaneously. If two filters on two vectors can both sync the same map, the driver must hold a mutex during the sync. Chapter 21 uses a single map and a single vector's task for completion, so there is no contention on the single-buffer case.
- The sync call may be made from a filter (primary interrupt context), from a task (thread context), or from process context (attach, detach, sysctl handlers). All are safe.
- The sync call must not be made from inside the `bus_dmamap_load` callback. The callback itself is the kernel's signal that the load has completed; the driver syncs later, after the callback returns and after the driver has programmed the device.
- The sync call must be paired with the PRE/POST flag matching the transfer direction, not with a flag matching what the driver "plans to do next". `BUS_DMASYNC_POSTREAD` after a host-to-device transfer is a bug, even if the CPU happens to be about to read the buffer (for verification, say) right after. Use `POSTWRITE` after a host-to-device transfer to match the transfer's direction.

Chapter 21's driver calls sync in three contexts: from `myfirst_dma_do_transfer` (called from process context or from a sysctl handler), from the rx filter (only after confirming the transfer is complete, but the chapter defers the sync to the task to keep the filter minimal), and from the rx task (under the softc mutex, before and after touching the buffer). Section 5 shows the exact placements.

21.5.8 The First Sync: When to Do It

One subtle question: does the driver need a sync call before the very first use of a freshly allocated buffer?

The answer is yes for bidirectional rings, and the required call is the combined PRE flag at setup time. The reason is that the allocator may not have zeroed the cache lines covering the buffer (even with `BUS_DMA_ZERO` setting the memory to zero, the cache lines may still hold whatever was there before the page was allocated), and without an initial sync the device might read garbage.

For Chapter 21's pattern (host-to-device transfer where the driver fills the buffer before PREWRITE), the initial sync is not needed: the driver writes the buffer, then issues PREWRITE, so the cache state before the write is irrelevant; PREWRITE handles any coherence concerns.

For device-to-host transfers, PREREAD serves the same purpose: the kernel does whatever invalidation is needed before the device writes, so any stale cache state is cleared.

The rule of thumb: as long as every hand-off to the device is preceded by a PRE sync, the buffer's initial state does not matter for correctness. The PRE sync is both an invalidate (for READ) and a flush (for WRITE), so it handles both directions. This is one of the API's small elegances; the driver does not need special "init" sync calls.

21.5.9 Wrapping Up Section 4

Section 4 pinned down the sync discipline. Four flags (PREREAD, PREWRITE, POSTREAD, POSTWRITE), one mental model (ownership passes between CPU and device), four common patterns (host-to-device, device-to-host, descriptor ring, shared ring), and one rule (every transfer has one PRE call and one POST call, both in the driver's code even if one is a no-op on the current platform). The `BUS_DMA_COHERENT` flag reduces sync cost but does not remove the sync requirement; `bus_dmamap_sync` is always called.

Section 5 builds the simulated DMA engine that will consume these sync calls. The engine accepts a bus address and a length, runs a `callout(9)`-scheduled transfer a few milliseconds later, and sets a completion bit in the status register. The driver's job is to program the engine, wait for completion, and verify the result. Each stage of Section 5 matches one of Section 4's four patterns.

21.6 Section 5: Building a Simulated DMA Engine

Section 4 laid down the sync discipline. Section 5 puts that discipline to work against a concrete DMA engine. The engine lives inside the Chapter 17 simulated backend and is built so the driver can drive it exactly as it would drive a real device. Tag, map, memory, sync, registers, completion, sync, verify: the full loop runs end to end, all on the simulator, inside the `myfirst` kernel module, with no real hardware required.

Stage 2's version tag is `1.4-dma-stage2`. The simulated engine is added to `myfirst_sim.c`. The driver's use of the engine is added to `myfirst.c`. The rx filter from Chapter 20 is unchanged at this stage; Section 6 wires the completion path through it. Here, the driver polls the engine's status register in a task because polling is the simpler path and isolates the DMA mechanics from the interrupt machinery for one stage.

21.6.1 The DMA Register Set

The simulated backend's MMIO register map from Chapter 17 already includes `INTR_STATUS`, `INTR_MASK`, `DATA_OUT`, `DATA_AV`, and `ERROR` at fixed offsets. Stage 2 extends the map with five new DMA registers at offsets `0x20` through `0x34`. The offsets and meanings are chosen to match a typical real device's DMA control block:

Offset	Name	R/W	Meaning
0x20	DMA_ADDR_LOW	RW	Low 32 bits of the DMA bus address.
0x24	DMA_ADDR_HIGH	RW	High 32 bits (zero on this device's 32-bit engine).
0x28	DMA_LEN	RW	Transfer length in bytes.
0x2C	DMA_DIR	RW	0 = host-to-device (engine reads), 1 = device-to-host.
0x30	DMA_CTRL	RW	Write 1 = start, write 2 = abort, bit 31 = reset.
0x34	DMA_STATUS	RO	Bit 0 = DONE, bit 1 = ERR, bit 2 = RUNNING.

The MYFIRST_REG_DMA_* constants live in `myfirst.h`:

```
#define MYFIRST_REG_DMA_ADDR_LOW    0x20
#define MYFIRST_REG_DMA_ADDR_HIGH  0x24
#define MYFIRST_REG_DMA_LEN        0x28
#define MYFIRST_REG_DMA_DIR        0x2C
#define MYFIRST_REG_DMA_CTRL        0x30
#define MYFIRST_REG_DMA_STATUS      0x34

#define MYFIRST_DMA_DIR_WRITE       0u
#define MYFIRST_DMA_DIR_READ        1u

#define MYFIRST_DMA_CTRL_START      (1u << 0)
#define MYFIRST_DMA_CTRL_ABORT      (1u << 1)
#define MYFIRST_DMA_CTRL_RESET      (1u << 31)

#define MYFIRST_DMA_STATUS_DONE     (1u << 0)
#define MYFIRST_DMA_STATUS_ERR      (1u << 1)
#define MYFIRST_DMA_STATUS_RUNNING  (1u << 2)
```

The naming follows the Chapter 17 convention: MYFIRST_REG_* for offsets, MYFIRST_DMA_* for named bits. The constants are used by both the simulation backend (which implements the engine) and the driver (which programs the engine); the shared header is the single source of truth.

The engine also sets MYFIRST_INTR_COMPLETE in INTR_STATUS when a transfer finishes, using the existing Chapter 19/20 interrupt mechanism. That is what Section 6 wires up; for Stage 2, the driver polls DMA_STATUS directly from a task and does not use the interrupt yet.

21.6.2 Extending the Simulation State

The simulated backend's struct `myfirst_sim` state grows with new fields to track the DMA engine:

```
struct myfirst_sim_dma {
    uint32_t    addr_low;
    uint32_t    addr_high;
    uint32_t    len;
    uint32_t    dir;
    uint32_t    ctrl;
    uint32_t    status;
    struct callout done_co;
    bool        armed;
};
```

The first six fields mirror the register contents. The `done_co` callout is the mechanism the engine uses to simulate transfer latency: when the driver writes `DMA_CTRL_START`, the engine schedules `done_co` to fire a few milliseconds later, which sets `DMA_STATUS_DONE` and raises `MYFIRST_INTR_COMPLETE`. The `armed` bool tracks whether the callout is scheduled so the abort path knows whether to call `callout_stop`.

The new state is embedded in the existing `struct myfirst_sim` structure:

```
struct myfirst_sim {
    /* ... existing Chapter 17 fields ... */
    struct myfirst_sim_dma dma;

    /* Backing store for the DMA engine's source or sink. */
    void          *dma_scratch;
    size_t        dma_scratch_size;

    /* Simulation back-channel: the host-visible KVA and bus address
     * the driver registers at myfirst_dma_setup time. A real device
     * never needs these; the simulator needs the KVA because it is
     * software and cannot reach RAM through the memory controller. */
    void          *dma_host_kva;
    bus_addr_t    dma_host_bus_addr;
    size_t        dma_host_size;
};
```

`dma_scratch` is a buffer that the engine uses as the other side of the transfer: when the driver programs a host-to-device transfer, the engine reads from the driver's buffer and writes into `dma_scratch`; when the driver programs a device-to-host transfer, the engine reads from `dma_scratch` and writes into the driver's buffer. In a real device, the "other side" is the device's own hardware (the wire, the storage media, the audio output); in the simulation, it is an in-kernel buffer that stands in.

The `dma_host_*` fields are the back-channel mentioned above. They are populated by a new helper `myfirst_sim_register_dma_buffer(sim, kva, bus_addr, size)` called from `myfirst_dma_setup`, and cleared by `myfirst_sim_unregister_dma_buffer` called from `myfirst_dma_tear down`. The helpers sit inside `myfirst_sim.c` and expose the sim's internal state only to the extent the simulator needs; they are the one explicit coupling between the DMA layer and the sim layer.

The scratch buffer is allocated at sim init time:

```
sc->sim.dma_scratch_size = 4096;
sc->sim.dma_scratch = malloc(sc->sim.dma_scratch_size,
    M_MYFIRST, M_WAITOK | M_ZERO);
```

and freed at sim teardown:

```
free(sc->sim.dma_scratch, M_MYFIRST);
```

The scratch buffer is not itself a DMA buffer (the engine is simulated; no real DMA happens). It is just kernel memory the simulation uses to model the device's internal storage.

21.6.3 The Register Access Hook

Chapter 17's simulation backend handles writes and reads on the simulated BAR by intercepting the `bus_read_4` and `bus_write_4` operations. The Stage 2 change extends the write hook to recognise the new DMA registers:

```
static void
myfirst_sim_write_4(struct myfirst_sim *sim, bus_size_t off, uint32_t val)
{
    switch (off) {
        /* ... existing registers from Chapter 17 ... */
        case MYFIRST_REG_DMA_ADDR_LOW:
            sim->dma.addr_low = val;
            break;
        case MYFIRST_REG_DMA_ADDR_HIGH:
            sim->dma.addr_high = val;
            break;
        case MYFIRST_REG_DMA_LEN:
            sim->dma.len = val;
            break;
        case MYFIRST_REG_DMA_DIR:
            sim->dma.dir = val;
            break;
        case MYFIRST_REG_DMA_CTRL:
            sim->dma.ctrl = val;
            myfirst_sim_dma_ctrl_written(sim, val);
            break;
        default:
            /* ... existing default handling ... */
            break;
    }
}
```

```

    }
}

```

The MYFIRST_REG_DMA_STATUS register is read-only, so the read hook handles it; the write hook ignores attempts to write it (or, in a defensive build, logs a warning and refuses). The DMA_STATUS read returns the current `sim->dma.status` value directly.

`myfirst_sim_dma_ctrl_written` is the engine's main decision point:

```

static void
myfirst_sim_dma_ctrl_written(struct myfirst_sim *sim, uint32_t val)
{
    if ((val & MYFIRST_DMA_CTRL_RESET) != 0) {
        if (sim->dma.armed) {
            callout_stop(&sim->dma.done_co);
            sim->dma.armed = false;
        }
        sim->dma.status = 0;
        sim->dma.addr_low = 0;
        sim->dma.addr_high = 0;
        sim->dma.len = 0;
        sim->dma.dir = 0;
        return;
    }
    if ((val & MYFIRST_DMA_CTRL_ABORT) != 0) {
        if (sim->dma.armed) {
            callout_stop(&sim->dma.done_co);
            sim->dma.armed = false;
        }
        sim->dma.status &= ~MYFIRST_DMA_STATUS_RUNNING;
        sim->dma.status |= MYFIRST_DMA_STATUS_ERR;
        return;
    }
    if ((val & MYFIRST_DMA_CTRL_START) != 0) {
        if ((sim->dma.status & MYFIRST_DMA_STATUS_RUNNING) != 0) {
            /* New START while an old transfer is in flight. */
            sim->dma.status |= MYFIRST_DMA_STATUS_ERR;
            return;
        }
        sim->dma.status = MYFIRST_DMA_STATUS_RUNNING;
        sim->dma.armed = true;
        callout_reset(&sim->dma.done_co, hz / 100,
            myfirst_sim_dma_done_co, sim);
        return;
    }
}

```



```

    }
}

```

The engine treats RESET, ABORT, and START as three distinct commands and handles each accordingly. The $\text{hz} / 100$ value schedules the completion roughly ten milliseconds in the future on a 1000 Hz kernel; the delay makes the transfer observably asynchronous and gives the driver a realistic window to poll or wait for the interrupt.

21.6.4 The Callout Handler

When the callout fires, the engine performs the simulated transfer and sets the status:

```

static void
myfirst_sim_dma_done_co(void *arg)
{
    struct myfirst_sim *sim = arg;
    bus_addr_t bus_addr;
    uint32_t len;
    void *kva;

    bus_addr = ((bus_addr_t)sim->dma.addr_high << 32) | sim->dma.addr_low;
    len = sim->dma.len;

    /* Back-channel lookup: find the KVA for this bus address. A real
     * device would not need this; the device's own DMA engine would
     * perform the memory-controller access. */
    if (sim->dma_host_kva == NULL ||
        bus_addr != sim->dma_host_bus_addr ||
        len == 0 || len > sim->dma_host_size ||
        len > sim->dma_scratch_size) {
        sim->dma.status = MYFIRST_DMA_STATUS_ERR;
        sim->dma.armed = false;
        myfirst_sim_dma_raise_complete(sim);
        return;
    }
    kva = sim->dma_host_kva;

    if (sim->dma.dir == MYFIRST_DMA_DIR_WRITE) {
        /* Host-to-device: sim reads host KVA, writes scratch. */
        memcpy(sim->dma_scratch, kva, len);
    } else {
        /* Device-to-host: sim reads scratch, writes host KVA.
         * Fill scratch with a recognisable pattern first so the
         * test can verify the transfer. */

```

```

        memset(sim->dma_scratch, 0x5A, len);
        memcpy(kva, sim->dma_scratch, len);
    }

    sim->dma.status = MYFIRST_DMA_STATUS_DONE;
    sim->dma.armed = false;
    myfirst_sim_dma_raise_complete(sim);
}

```

The two helpers `myfirst_sim_dma_copy_from_host` and `myfirst_sim_dma_copy_to_host` do the actual byte movement. Here the simulation runs into a fundamental limit: it is software running inside the kernel, not a real device. A real device uses the bus address to reach RAM through the memory controller; a software simulator cannot physically take that path. The simulator needs a kernel virtual address it can dereference, not the bus address.

We solve this with a deliberate back-channel. At `myfirst_dma_setup` time, the driver calls a new helper `myfirst_sim_register_dma_buffer(sc, sc->dma_vaddr, sc->dma_bus_addr, MYFIRST_DMA_BUFFER_SIZE)`. The helper stores the (KVA, bus address, size) triple inside the `sim`'s state. When the callout later fires, it reads the bus address the driver programmed into `DMA_ADDR_LOW/DMA_ADDR_HIGH`, looks it up in the stored triple, and recovers the KVA. The `memcpy` then works against the KVA.

This back-channel is a simulation-only crutch. Real hardware never receives a KVA; the device only ever uses the bus address, and the memory controller resolves it through the platform's address mapping (possibly via an IOMMU). The back-channel's sole purpose is to let the simulator pretend to do the DMA without actually programming the memory controller. Keeping the mechanism explicit and named is the best defence against confusion: the reader sees a clearly-labelled simulation bridge rather than a magical pointer cast, and the divide between “what the device sees” (the bus address) and “what the CPU sees” (the KVA) stays clean.

`myfirst_sim_dma_raise_complete` sets `MYFIRST_INTR_COMPLETE` in `INTR_STATUS` and, if `INTR_MASK` has the bit set, fires the simulated interrupt through the existing Chapter 19/20 path:

```

static void
myfirst_sim_dma_raise_complete(struct myfirst_sim *sim)
{
    sim->intr_status |= MYFIRST_INTR_COMPLETE;
    if ((sim->intr_mask & MYFIRST_INTR_COMPLETE) != 0)
        myfirst_sim_raise_intr(sim);
}

```

For Stage 2 we do not enable the `MYFIRST_INTR_COMPLETE` bit in `INTR_MASK` yet; the driver will poll. Section 6 enables the bit and wires the completion path through the filter.

21.6.5 The Driver's Side: Programming the Engine

On the driver side, a single helper function programs the engine and waits:

```

int
myfirst_dma_do_transfer(struct myfirst_softc *sc, int direction,
    size_t length)
{
    uint32_t status;
    int timeout;

    if (length == 0 || length > MYFIRST_DMA_BUFFER_SIZE)
        return (EINVAL);

    if (direction == MYFIRST_DMA_DIR_WRITE) {
        bus_dmamap_sync(sc->dma_tag, sc->dma_map,
            BUS_DMASYNC_PREWRITE);
    } else {
        bus_dmamap_sync(sc->dma_tag, sc->dma_map,
            BUS_DMASYNC_PREREAD);
    }

    CSR_WRITE_4(sc, MYFIRST_REG_DMA_ADDR_LOW,
        (uint32_t)(sc->dma_bus_addr & 0xFFFFFFFF));
    CSR_WRITE_4(sc, MYFIRST_REG_DMA_ADDR_HIGH,
        (uint32_t)(sc->dma_bus_addr >> 32));
    CSR_WRITE_4(sc, MYFIRST_REG_DMA_LEN, (uint32_t)length);
    CSR_WRITE_4(sc, MYFIRST_REG_DMA_DIR, direction);
    CSR_WRITE_4(sc, MYFIRST_REG_DMA_CTRL, MYFIRST_DMA_CTRL_START);

    /* Poll for completion. Stage 2 is polling-only. */
    for (timeout = 500; timeout > 0; timeout--) {
        pause("dma", hz / 1000); /* 1 ms. */
        status = CSR_READ_4(sc, MYFIRST_REG_DMA_STATUS);
        if ((status & MYFIRST_DMA_STATUS_DONE) != 0)
            break;
        if ((status & MYFIRST_DMA_STATUS_ERR) != 0) {
            if (direction == MYFIRST_DMA_DIR_WRITE)
                bus_dmamap_sync(sc->dma_tag, sc->dma_map,
                    BUS_DMASYNC_POSTWRITE);
            else
                bus_dmamap_sync(sc->dma_tag, sc->dma_map,
                    BUS_DMASYNC_POSTREAD);
            return (EIO);
        }
    }

    if (timeout == 0) {

```

```

        CSR_WRITE_4(sc, MYFIRST_REG_DMA_CTRL, MYFIRST_DMA_CTRL_ABORT);
    if (direction == MYFIRST_DMA_DIR_WRITE)
        bus_dmamap_sync(sc->dma_tag, sc->dma_map,
            BUS_DMASYNC_POSTWRITE);
    else
        bus_dmamap_sync(sc->dma_tag, sc->dma_map,
            BUS_DMASYNC_POSTREAD);
    return (ETIMEDOUT);
}

/* Acknowledge the completion. */
CSR_WRITE_4(sc, MYFIRST_REG_INTR_STATUS, MYFIRST_INTR_COMPLETE);

if (direction == MYFIRST_DMA_DIR_WRITE)
    bus_dmamap_sync(sc->dma_tag, sc->dma_map,
        BUS_DMASYNC_POSTWRITE);
else
    bus_dmamap_sync(sc->dma_tag, sc->dma_map,
        BUS_DMASYNC_POSTREAD);

return (0);
}

```

The function is sixty lines, and the shape is worth reading twice. The PRE sync happens once at the top; the POST sync happens exactly once on every return path (success, error, timeout). The polling loop uses `pause("dma", hz / 1000)` for one-millisecond sleeps; on a 1000 Hz kernel this gives five hundred one-millisecond tries before giving up at five hundred milliseconds total. `pause(9)` takes the global sleep mutex briefly but is safe from a process-context driver helper. A real device with an interrupt path would not poll; Section 6 replaces the polling with an interrupt wait.

The PRE and POST flag choice matches the direction. For host-to-device (DIR_WRITE), the driver has just written the buffer, so PREWRITE is required; after the transfer, POSTWRITE releases any bounce region. For device-to-host (DIR_READ), the driver will read the buffer, so PREREAD is required; after the transfer, POSTREAD completes the copy from bounce region (if any) and invalidates the cache (if needed).

21.6.6 Exercising the Engine From User Space

The driver already exposes a sysctl tree under `dev.myfirst.N`. from Chapter 20. Stage 2 adds two new write-only sysctls that trigger a transfer:

```

SYSCTL_ADD_PROC(ctx, kids, OID_AUTO, "dma_test_write",
    CTLTYPE_UINT | CTLFLAG_WR | CTLFLAG_MPSAFE, sc, 0,
    myfirst_dma_sysctl_test_write, "IU",
    "Trigger a host-to-device DMA transfer");

```

```

SYSCTL_ADD_PROC(ctx, kids, OID_AUTO, "dma_test_read",
    CTLTYPE_UINT | CTLFLAG_WR | CTLFLAG_MPSAFE, sc, 0,
    myfirst_dma_sysctl_test_read, "IU",
    "Trigger a device-to-host DMA transfer");

```

The handler for `dma_test_write`:

```

static int
myfirst_dma_sysctl_test_write(SYSCTL_HANDLER_ARGS)
{
    struct myfirst_softc *sc = arg1;
    unsigned int pattern;
    int err;

    err = sysctl_handle_int(oidp, &pattern, 0, req);
    if (err != 0 || req->newptr == NULL)
        return (err);

    memset(sc->dma_vaddr, (int)(pattern & 0xFF),
        MYFIRST_DMA_BUFFER_SIZE);
    err = myfirst_dma_do_transfer(sc, MYFIRST_DMA_DIR_WRITE,
        MYFIRST_DMA_BUFFER_SIZE);
    if (err != 0)
        device_printf(sc->dev, "dma_test_write: error %d\n", err);
    else
        device_printf(sc->dev,
            "dma_test_write: pattern 0x%02x transferred\n",
            pattern & 0xFF);
    return (err);
}

```

The user writes an integer to `dev.myfirst.0.dma_test_write`, the low byte becomes a fill pattern, the driver fills the buffer with that pattern, programs the engine, waits for completion, and logs the result. The handler for `dma_test_read` is symmetric: it triggers a device-to-host transfer, waits for completion, reads the first few bytes of the buffer, and logs them.

A user-space session runs like this:

```

sudo sysctl dev.myfirst.0.dma_test_write=0xAA
sudo dmesg | tail -1
# myfirst0: dma_test_write: pattern 0xaa transferred

sudo sysctl dev.myfirst.0.dma_test_read=1
sudo dmesg | tail -1
# myfirst0: dma_test_read: first bytes 5A 5A 5A 5A 5A 5A 5A 5A

```

The first transfer sends 0xAA to the simulated device; the second receives the device's 0x5A pattern back. Both transfers exercise the full PRE/POST discipline.

21.6.7 Observability: Per-Transfer Counters

Stage 2 adds four softc counters to the DMA state:

```
uint64_t dma_transfers_write;  
uint64_t dma_transfers_read;  
uint64_t dma_errors;  
uint64_t dma_timeouts;
```

Each is incremented by `myfirst_dma_do_transfer` on the appropriate exit path, and each is exposed as a read-only sysctl. The counters are the driver's Stage 2 equivalent of the per-vector counters Chapter 20 added to the interrupt path; they make it easy to confirm at a glance that transfers are happening as expected and to detect silent failures.

The counters are read with atomic operations (`atomic_add_64`) so they can be updated safely even if Section 6's interrupt-driven completion path ends up running from a filter context. Chapter 20's Section 4 covered the atomic discipline; Stage 2's counters follow the same pattern.

21.6.8 Verifying Stage 2

Build and load:

```
make clean && make  
sudo kldload ./myfirst.ko
```

`dmesg` shows the DMA banner from Stage 1 plus the new DMA-enabled banner:

```
myfirst0: DMA buffer 4096 bytes at KVA 0xfffffe... bus addr 0x...  
myfirst0: DMA engine present, scratch 4096 bytes  
myfirst0: interrupt mode: MSI-X, 3 vectors
```

Run the test:

```
sudo sysctl dev.myfirst.0.dma_test_write=0x33  
sudo sysctl dev.myfirst.0.dma_test_read=1  
sudo sysctl dev.myfirst.0.dma_transfers_write  
sudo sysctl dev.myfirst.0.dma_transfers_read
```

Expected:

```
dev.myfirst.0.dma_transfers_write: 1
dev.myfirst.0.dma_transfers_read: 1
```

Run it a thousand times in a loop:

```
for i in $(seq 1 1000); do
    sudo sysctl dev.myfirst.0.dma_test_write=$((i & 0xFF)) >/dev/null
    sudo sysctl dev.myfirst.0.dma_test_read=1 >/dev/null
done
sudo sysctl dev.myfirst.0.dma_transfers_write
sudo sysctl dev.myfirst.0.dma_transfers_read
sudo sysctl dev.myfirst.0.dma_errors
sudo sysctl dev.myfirst.0.dma_timeouts
```

Expected counts: 1000 write transfers, 1000 read transfers, 0 errors, 0 timeouts. Any timeout or error during the test indicates a bug in the simulation or a race in the driver; both are worth catching early.

Unload the module and verify that the callout is stopped, the scratch buffer is freed, and the tag is destroyed:

```
sudo kldunload myfirst
vmstat -m | grep myfirst || true
```

The line should be missing or show zero allocations.

21.6.9 What Stage 2 Does and Does Not Do

Stage 2 is the first stage where actual DMA transfers run. The driver programs the engine; the engine copies bytes through the simulated path; the driver observes the result. The sync discipline is exercised in both directions.

What Stage 2 does not do is use the interrupt path. Polling is acceptable for a teaching stage but inappropriate for a real driver: the driver holds a kernel thread in pause for the duration of each transfer. Section 6 replaces the poll with the Chapter 19/20 interrupt mechanism; the rx filter sees MYFIRST_INTR_COMPLETE, acknowledges it, enqueues the task, and the task does the POST sync and verification. Stage 3 is the fully interrupt-driven version.

Stage 2 also does not handle every error cleanly. The timeout and abort paths are present but minimal; Section 7 walks through them carefully and extends the recovery logic. The single-transfer pattern is also limited; Section 8's refactor exposes helper functions that make multi-transfer patterns easy to add.

21.6.10 Common Mistakes in Stage 2

Four mistakes are worth calling out.

Mistake one: calling `bus_dmamap_sync` inside the callout. The callout runs in softclock context, and the sync itself is safe from there, but the buffer accesses the simulation does (`myfirst_sim_dma_copy_from_host`) need

the driver's map to be loaded. If the driver detaches while a callout is armed, the map is unloaded under the callout's feet. The fix is Chapter 21's detach pattern: stop the callout (`callout_drain`) before unloading the map. Section 7 revisits this carefully.

Mistake two: forgetting to acknowledge `MYFIRST_INTR_COMPLETE`. The simulation raises the bit; the driver must clear it by writing the bit back to `INTR_STATUS`. If the driver does not, the next transfer sees a stale `DONE` bit and the polling loop returns immediately with correct status but before the new transfer has run. The acknowledge is in the polling-loop version (`CSR_WRITE_4(sc, MYFIRST_REG_INTR_STATUS, MYFIRST_INTR_COMPLETE);`); Section 6 moves the acknowledge into the filter path where interrupt driven.

Mistake three: using `DELAY` or `cpu_spinwait` instead of `pause`. The Stage 2 polling loop uses `pause("dma", hz / 1000)` to yield. A tight spin with `DELAY(1000)` pegs a CPU for the entire transfer window, which is acceptable on a uniprocessor test but unacceptable in production. The `pause` call puts the thread to sleep and lets other work run. This matters more in the Section 6 version, where the driver calls `cv_timedwait` instead of polling, but the principle is the same: yield the CPU whenever you are waiting for external events.

Mistake four: forgetting to cap length to the tag's `maxsize`. The driver's helper accepts a length argument; if the caller passes a larger value than `MYFIRST_DMA_BUFFER_SIZE`, the engine writes past the end of the scratch buffer (in the simulation) or past the end of the host buffer (on real hardware). Chapter 21's code guards against this in the helper's first lines; forgetting the guard is a common source of subtle corruption.

21.6.11 Wrapping Up Section 5

Section 5 made the simulated DMA engine real. The register map, the state machine, the callout-based latency, the driver's polling helper, the `sysctl` test interface, and the counters are all in place. The driver runs full transfers in both directions; the sync discipline from Section 4 is exercised on every one. Stage 2 is a working DMA driver even if it polls rather than handling interrupts.

Section 6 replaces the polling with the Chapter 19/20 interrupt path. The completion interrupt arrives through the rx filter, the filter enqueues the task, and the task performs the POST sync and the verification. The user-visible behaviour does not change; the driver's internals do. That change is what makes the driver scale: a polling driver pegs a CPU per transfer, an interrupt-driven driver handles the same transfers with negligible CPU cost.

21.7 Section 6: Handling DMA Completion in the Driver

Section 5 produced a polling-based DMA driver. Section 6 rewrites the completion path to use the Chapter 19/20 interrupt machinery. The goals are: no CPU is held in pause while a transfer runs; the completion is delivered through a filter that runs in primary interrupt context; the task in thread context does the POST sync and the verification; and the user-visible behaviour is unchanged from Stage 2. The version tag becomes `1.4-dma-stage3`.

The Chapter 20 per-vector machinery already has the right shape for this. One of the three vectors (the rx vector, or in the legacy fallback case the single admin vector) handles `MYFIRST_INTR_COMPLETE`. The filter acknowledges; the task processes. Chapter 21's change is to make the task's processing include a DMA POST sync and a buffer read.

21.7.1 Why Interrupt-Driven Is the Right Design

Section 5's polling path works, but it has four weaknesses that Stage 3 fixes.

One CPU is tied up per in-flight transfer. The pause call puts the thread to sleep on the dma wait channel, which is fine, but the thread's stack, cache state, and scheduling overhead still exist. A driver with many in-flight transfers would tie up many threads. Interrupt-driven completion uses one task per vector, reused across all transfers.

Latency is coarse. The polling loop wakes every millisecond. A real engine might complete in microseconds; the driver waits up to a full millisecond before noticing. Interrupt-driven completion is delivered at the hardware's natural latency.

The polling loop does not compose with multiple in-flight transfers. To issue two transfers back-to-back, the driver must either wait for the first to complete (serialising) or maintain its own per-transfer state (ad-hoc bookkeeping). Interrupt-driven completion naturally composes: each completion fires its own interrupt, the filter routes it to the task, and the task handles it.

Polling cannot run in interrupt context. If the driver ever needs to complete a transfer from inside an interrupt handler (for example, a receive-complete interrupt that immediately triggers a new descriptor fill), the polling loop will not work because pause cannot be called from the filter. The interrupt-driven path is the only composable design.

Stage 3 keeps the polling helper as a fallback for sysctl-driven tests (it is useful for deterministic testing) and adds an interrupt-driven helper for the real use case.

21.7.2 The Interrupt Path Modifications

Chapter 20's rx filter handled MYFIRST_INTR_DATA_AV. Stage 3 extends it (or the legacy-fallback admin filter) to handle MYFIRST_INTR_COMPLETE as well:

```
int
myfirst_msix_rx_filter(void *arg)
{
    struct myfirst_vector *vec = arg;
    struct myfirst_softc *sc = vec->sc;
    uint32_t status;
    bool handled = false;

    status = ICSR_READ_4(sc, MYFIRST_REG_INTR_STATUS);

    if ((status & MYFIRST_INTR_DATA_AV) != 0) {
        atomic_add_64(&vec->fire_count, 1);
        ICSR_WRITE_4(sc, MYFIRST_REG_INTR_STATUS,
            MYFIRST_INTR_DATA_AV);
        atomic_add_64(&sc->intr_data_av_count, 1);
        handled = true;
    }
}
```

```

if ((status & MYFIRST_INTR_COMPLETE) != 0) {
    atomic_add_64(&sc->dma_complete_intrs, 1);
    ICSR_WRITE_4(sc, MYFIRST_REG_INTR_STATUS,
        MYFIRST_INTR_COMPLETE);
    handled = true;
}

if (!handled) {
    atomic_add_64(&vec->stray_count, 1);
    return (FILTER_STRAY);
}

if (sc->intr_tq != NULL)
    taskqueue_enqueue(sc->intr_tq, &vec->task);
return (FILTER_HANDLED);
}

```

The filter reads `INTR_STATUS` once, checks for both bits, acknowledges each one it sees, increments per-bit counters, and enqueues the task if any bit was set. The pattern is exactly Chapter 19's generalised to two bits per vector; the only new thing is the `MYFIRST_INTR_COMPLETE` counter.

The task picks up the state the filter recorded:

```

static void
myfirst_msix_rx_task_fn(void *arg, int npending)
{
    struct myfirst_vector *vec = arg;
    struct myfirst_softc *sc = vec->sc;
    bool did_complete;

    MYFIRST_LOCK(sc);
    if (sc->hw == NULL || !sc->pci_attached) {
        MYFIRST_UNLOCK(sc);
        return;
    }

    /* Data-available path (Chapter 19/20). */
    sc->intr_last_data = CSR_READ_4(sc, MYFIRST_REG_DATA_OUT);
    sc->intr_task_invocations++;
    cv_broadcast(&sc->data_cv);

    /* DMA-complete path (Chapter 21). */
    did_complete = false;
    if (sc->dma_in_flight) {

```

```

    sc->dma_in_flight = false;
    did_complete = true;
    if (sc->dma_last_direction == MYFIRST_DMA_DIR_WRITE)
        bus_dmamap_sync(sc->dma_tag, sc->dma_map,
            BUS_DMASYNC_POSTWRITE);
    else
        bus_dmamap_sync(sc->dma_tag, sc->dma_map,
            BUS_DMASYNC_POSTREAD);
    sc->dma_last_status = CSR_READ_4(sc,
        MYFIRST_REG_DMA_STATUS);
    atomic_add_64(&sc->dma_complete_tasks, 1);
    cv_broadcast(&sc->dma_cv);
}
MYFIRST_UNLOCK(sc);

(void)did_complete;
}

```

The task takes the softc mutex (Chapter 11 discipline; the sleep mutex is held while accessing shared state). The `dma_in_flight` flag, set by the transfer helper before the doorbell write, tells the task that a DMA completion is pending. If it is set, the task clears it, issues the POST sync matching the recorded direction, reads the final status, and broadcasts `dma_cv` so any waiter can wake up.

The sync call from the task is safe: the task runs in thread context under the softc mutex, and the sync call does not acquire any extra locks. The Chapter 19 lock discipline carries through.

21.7.3 The New Interrupt-Driven Transfer Helper

The Stage 3 version of the helper starts the transfer, arms the callout or the real hardware, and sleeps until the completion task signals:

```

int
myfirst_dma_do_transfer_intr(struct myfirst_softc *sc, int direction,
    size_t length)
{
    int err;

    if (length == 0 || length > MYFIRST_DMA_BUFFER_SIZE)
        return (EINVAL);

    MYFIRST_LOCK(sc);
    if (sc->dma_in_flight) {
        MYFIRST_UNLOCK(sc);
        return (EBUSY);
    }
}

```

```

}

/* Issue the PRE sync before touching the device. */
if (direction == MYFIRST_DMA_DIR_WRITE)
    bus_dmamap_sync(sc->dma_tag, sc->dma_map,
        BUS_DMASYNC_PREWRITE);
else
    bus_dmamap_sync(sc->dma_tag, sc->dma_map,
        BUS_DMASYNC_PREREAD);

sc->dma_last_direction = direction;
sc->dma_in_flight = true;

CSR_WRITE_4(sc, MYFIRST_REG_DMA_ADDR_LOW,
    (uint32_t)(sc->dma_bus_addr & 0xFFFFFFFF));
CSR_WRITE_4(sc, MYFIRST_REG_DMA_ADDR_HIGH,
    (uint32_t)(sc->dma_bus_addr >> 32));
CSR_WRITE_4(sc, MYFIRST_REG_DMA_LEN, (uint32_t)length);
CSR_WRITE_4(sc, MYFIRST_REG_DMA_DIR, direction);
CSR_WRITE_4(sc, MYFIRST_REG_DMA_CTRL, MYFIRST_DMA_CTRL_START);

/* Wait for the task to set dma_in_flight = false. */
err = cv_timedwait(&sc->dma_cv, &sc->mtx, hz); /* 1 s timeout */
if (err == EWOULDBLOCK) {
    /* Abort the engine and issue the POST sync so we do not
     * leave the map in an inconsistent state. */
    CSR_WRITE_4(sc, MYFIRST_REG_DMA_CTRL, MYFIRST_DMA_CTRL_ABORT);
    if (direction == MYFIRST_DMA_DIR_WRITE)
        bus_dmamap_sync(sc->dma_tag, sc->dma_map,
            BUS_DMASYNC_POSTWRITE);
    else
        bus_dmamap_sync(sc->dma_tag, sc->dma_map,
            BUS_DMASYNC_POSTREAD);
    sc->dma_in_flight = false;
    atomic_add_64(&sc->dma_timeouts, 1);
    MYFIRST_UNLOCK(sc);
    return (ETIMEDOUT);
}

/* The task has issued the POST sync and recorded dma_last_status. */
if ((sc->dma_last_status & MYFIRST_DMA_STATUS_ERR) != 0) {
    atomic_add_64(&sc->dma_errors, 1);
    MYFIRST_UNLOCK(sc);
    return (EIO);
}

```

```

    }

    if (direction == MYFIRST_DMA_DIR_WRITE)
        atomic_add_64(&sc->dma_transfers_write, 1);
    else
        atomic_add_64(&sc->dma_transfers_read, 1);

    MYFIRST_UNLOCK(sc);
    return (0);
}

```

The helper is longer than Stage 2's because it coordinates with the task: the task issues the POST sync; the helper only does so on the timeout path where the task will never see completion. The coordination is the point of the redesign; the helper has moved from “poll until done” to “signal the device, wait on a condition variable, let the task complete me”.

The `cv_timedwait` call takes the softc mutex as its second argument; that is the standard `cv_timedwait` contract (mutex held going in, released for the wait, re-acquired on wake). The mutex scope covers every access to the shared DMA state (`dma_in_flight`, `dma_last_direction`, `dma_last_status`).

One subtle point: the PRE sync runs before the mutex is released for the wait, and the POST sync runs after the wake. Both are inside the critical section of the transfer, so no other thread can see the buffer in a partially-synchronised state. This is the locking payoff of Chapter 11.

21.7.4 The MYFIRST_INTR_COMPLETE Bit Goes Live

Stage 2 masked `MYFIRST_INTR_COMPLETE` out of `INTR_MASK`. Stage 3 enables it:

```

CSR_WRITE_4(sc, MYFIRST_REG_INTR_MASK,
    MYFIRST_INTR_DATA_AV | MYFIRST_INTR_ERROR | MYFIRST_INTR_COMPLETE);

```

The change is a single line in the interrupt setup. The simulation's engine raises the bit at completion; the driver's filter sees it and enqueues the task; the task handles it. The entire pipeline lights up with this one change.

21.7.5 Verifying Stage 3

Build and load:

```

make clean && make
sudo kldload ./myfirst.ko

```

Run the same `sysctl`-driven test as Stage 2, but wire it to the interrupt-driven helper:

```
static int
myfirst_dma_sysctl_test_write(SYSCTL_HANDLER_ARGS)
{
    struct myfirst_softc *sc = arg1;
    unsigned int pattern;
    int err;

    err = sysctl_handle_int(oidp, &pattern, 0, req);
    if (err != 0 || req->newptr == NULL)
        return (err);

    memset(sc->dma_vaddr, (int)(pattern & 0xFF),
           MYFIRST_DMA_BUFFER_SIZE);
    err = myfirst_dma_do_transfer_intr(sc, MYFIRST_DMA_DIR_WRITE,
                                       MYFIRST_DMA_BUFFER_SIZE);
    if (err != 0)
        device_printf(sc->dev,
                      "dma_test_write: intr err %d\n", err);
    return (err);
}
```

Run a thousand transfers and observe the counters:

```
for i in $(seq 1 1000); do
    sudo sysctl dev.myfirst.0.dma_test_write=$((i & 0xFF)) >/dev/null
    sudo sysctl dev.myfirst.0.dma_test_read=1 >/dev/null
done
sudo sysctl dev.myfirst.0.dma_complete_intrs
sudo sysctl dev.myfirst.0.dma_complete_tasks
sudo sysctl dev.myfirst.0.dma_transfers_write
sudo sysctl dev.myfirst.0.dma_transfers_read
```

Expected: `dma_complete_intrs` equals 2000 (one per transfer in both directions), `dma_complete_tasks` equals 2000 (the task processed each completion), `dma_transfers_write` and `dma_transfers_read` each equal 1000.

Run `vmstat -i` during the test to see the vector receiving interrupts. With MSI-X, the rx vector's count should climb visibly. With legacy fallback, the single vector's count climbs.

21.7.6 Stopping In-Flight Transfers at Detach

Stage 3 has a new detach hazard: the task may be running the POST sync at the moment `myfirst_dma_tear_down` is called. The teardown must not unload the map while the task is still holding references to it.

The fix is one line added to the detach path before `myfirst_dma_tear_down`:

```
taskqueue_drain(sc->intr_tq, &sc->vectors[MYFIRST_VECTOR_RX].task);
```

`taskqueue_drain` waits for the task to finish running; after it returns, the driver is guaranteed that no further task invocations will happen on that task (subject to the caveat that another filter might re-enqueue it; the detach path has already masked `INTR_MASK` so no more interrupts will fire). With the drain in place, the teardown is safe.

The MSI-X teardown from Chapter 20 already drains the task; Chapter 21 extends the Stage 3 teardown to also stop the simulation callout before unloading the map:

```
/* In myfirst_sim_destroy, called from detach: */  
callout_drain(&sc->sim.dma.done_co);
```

`callout_drain` waits for the callout to finish running; after it returns, the engine is guaranteed not to raise any more completions. Combined with the taskqueue drain, the detach is safe against every race.

21.7.7 What Stage 3 Produces

Stage 3 is a fully interrupt-driven DMA driver. Every transfer programs the engine, sleeps on a condition variable, wakes when the completion task signals, and returns with the transferred data synced and verified. The driver never holds the CPU while a transfer runs. The `sysctl` test interface works as before; the internals match production DMA code.

Stage 3 does not yet cover error recovery in depth. Section 7 walks through the specific failure modes and the patterns that handle each.

21.7.8 Common Mistakes in Stage 3

Four more mistakes are worth pointing out.

Mistake one: forgetting the `cv_broadcast` in the task. If the task does the `POST` sync but does not broadcast `dma_cv`, the helper waits the full timeout and returns `ETIMEDOUT` even though the transfer succeeded. The fix is the single `cv_broadcast` line; missing it is a subtle bug that manifests as every transfer timing out.

Mistake two: calling `cv_timedwait` without holding the mutex. `cv_timedwait` requires the mutex to be held at entry; the kernel panics under `INVARIANTS` if it is not. The helper's structure (lock, do work, wait, handle result, unlock) keeps the mutex held throughout; the wait releases it briefly during the sleep and re-acquires it on wake. Breaking this pattern (releasing the mutex before `cv_timedwait`) is a race that `WITNESS` catches.

Mistake three: handling the timeout path without a `POST` sync. The timeout path in the helper aborts the engine and then still issues the `POST` sync. The sync is required because `BUS_DMA_NOWAIT` may have inserted bounce buffers at `PRE` time; without `POST`, the bounce region is not released. Forgetting it leaks bounce pages slowly.

Mistake four: leaving `dma_in_flight` set on an error. Every exit path in the helper clears `dma_in_flight`. Forgetting to clear it on the error path means the next transfer returns `EBUSY` even though no transfer is running. The helper's structure (set at the top, cleared either by the task or explicitly on timeout/error) is the robust pattern.

21.7.9 Wrapping Up Section 6

Section 6 replaced Stage 2's polling with Stage 3's interrupt-driven completion. The filter sees the completion bit, the task issues the POST sync and signals, the helper wakes and returns. Throughput and latency both improve; CPU cost drops; the driver composes with other work naturally. The code is longer than Stage 2's because it coordinates across two contexts (task and helper), but the pattern is a direct application of Chapter 11's lock discipline and Chapter 20's per-vector task design; no new ideas are needed beyond the DMA-specific sync placement.

Section 7 is the error-handling tour. Every failure mode the chapter addresses gets a walkthrough: what happens, what the symptom looks like, and what the driver does about it. The patterns are the same ones every production DMA driver has to handle; the chapter's simulated environment is a good place to rehearse them.

21.8 Section 7: DMA Error Handling and Edge Cases

Section 6 produced a working interrupt-driven DMA driver. Section 7 is the chapter where the driver stops assuming success. Every step of the DMA pipeline can fail: the tag creation can fail, the memory allocation can fail, the load can fail, the load can be deferred, the engine can report an error, the engine can never complete (timeout), the detach can race with a running transfer. Each failure mode has a pattern that handles it; Section 7 walks through each in turn.

The goal is not to make Chapter 21's driver resistant to every conceivable failure; it is to teach the patterns so the reader recognises them in production code and applies them when writing real drivers. Many of the patterns are short; the explanation is the learning object, not the code.

21.8.1 Failure Mode 1: `bus_dma_tag_create` Returns `ENOMEM`

`bus_dma_tag_create` can return `ENOMEM` if the kernel cannot allocate the tag structure itself. The allocation is small (the tag is a few hundred bytes) and only happens once at attach, so the failure is rare in practice, but the driver must still handle it.

The pattern: check the return value, log, return the error to the caller, do not touch any of the downstream DMA state. The Stage 1 code already does this:

```
err = bus_dma_tag_create(bus_get_dma_tag(sc->dev), ...);
if (err != 0) {
    device_printf(sc->dev, "bus_dma_tag_create failed: %d\n", err);
    return (err);
}
```

The higher-level attach function sees the error, runs its failure-path unwind, and the kernel reports the probe failure cleanly. No DMA resources are leaked because none were allocated.

A `BUS_DMA_ALLOCNOW` flag tells the kernel to pre-allocate bounce buffer resources at tag creation time, so some kinds of allocation failure move from load time to tag time. This is useful for drivers that cannot tolerate a later load failure; Chapter 21's driver does not use it (the simulation does not need bounce buffers), but production drivers that interface with 32-bit hardware often do.

21.8.2 Failure Mode 2: `bus_dmamem_alloc` Returns `ENOMEM`

`bus_dmamem_alloc` can return `ENOMEM` if the allocator cannot satisfy the tag's constraints at the moment of the call. For large buffers, this is more likely than it sounds: a request for a contiguous four-megabyte buffer with a 4 KB alignment on a fragmented system can genuinely fail. For Chapter 21's 4 KB buffer, the failure is extremely unlikely, but the code still checks.

The pattern: check, log, destroy the tag that was just created, propagate the error:

```
err = bus_dmamem_alloc(sc->dma_tag, &sc->dma_vaddr,
    BUS_DMA_WAITOK | BUS_DMA_COHERENT | BUS_DMA_ZERO, &sc->dma_map);
if (err != 0) {
    device_printf(sc->dev, "bus_dmamem_alloc failed: %d\n", err);
    bus_dma_tag_destroy(sc->dma_tag);
    sc->dma_tag = NULL;
    return (err);
}
```

The key point is the tag destroy in the error path. Forgetting it leaks the tag across every failed attach; over many retries it becomes a noticeable leak. The Stage 1 code follows this pattern; every new allocation-path call in later sections must too.

21.8.3 Failure Mode 3: `bus_dmamap_load` Returns `EINVAL`

`bus_dmamap_load` returns `EINVAL` in two related cases:

1. The buffer is larger than the tag's `maxsize`. For example, loading an 8 KB buffer with a tag whose `maxsize` is 4 KB fails immediately with `EINVAL`. The callback is called with `error = EINVAL` as a confirmation.
2. The buffer's properties violate the tag's constraints in a way the kernel can detect statically (alignment, boundary). These are rarer because the tag's constraints usually match the buffer's characteristics by design.

The pattern: check, log, free the memory, destroy the tag, propagate:

```
err = bus_dmamap_load(sc->dma_tag, sc->dma_map,
    sc->dma_vaddr, MYFIRST_DMA_BUFFER_SIZE,
    myfirst_dma_single_map, &sc->dma_bus_addr, BUS_DMA_NOWAIT);
if (err != 0) {
    device_printf(sc->dev, "bus_dmamap_load failed: %d\n", err);
    bus_dmamem_free(sc->dma_tag, sc->dma_vaddr, sc->dma_map);
    sc->dma_vaddr = NULL;
    bus_dma_tag_destroy(sc->dma_tag);
    sc->dma_tag = NULL;
    return (err);
}
```

The Stage 1 code has this pattern. Note that `EINVAL` is distinct from `EFBIG`: `EINVAL` is passed to the callback if the arguments are invalid; `EFBIG` comes back in the callback when the mapping cannot be achieved within the tag's segment constraints. Chapter 21's single-segment load is too simple to see `EFBIG`; scatter-gather loads of large buffers can.

21.8.4 Failure Mode 4: `bus_dmamap_load` Returns `EINPROGRESS`

`EINPROGRESS` means the kernel has queued the load because bounce buffers (or other mapping resources) are not currently available. The callback will run later, in a different context, when resources free up. Chapter 21 uses `BUS_DMA_NOWAIT`, which forbids this behaviour (the kernel returns `ENOMEM` instead), so `EINPROGRESS` does not occur in the chapter's driver. A driver that uses `BUS_DMA_WAITOK` or zero flags and sees `EINPROGRESS` has to do more work:

```
err = bus_dmamap_load(sc->dma_tag, sc->dma_map, buf, len,
    my_callback, sc, 0);
if (err == EINPROGRESS) {
    /* Do not free buf or destroy the tag here; the callback will
     * run later. The caller must be prepared to handle the load
     * completing at any time. */
    return (0);
}
```

The driver then has to arrange for the callback to record the result in a place the rest of the driver can see (often a softc field guarded by the tag's lockfunc), and the caller that is “waiting” for the load has to sleep on a condition variable the callback broadcasts. This is the deferred-load pattern; it is complex enough that most drivers prefer `BUS_DMA_NOWAIT` and retry-on-failure over the deferred path.

Chapter 21's driver does not use this pattern. Section 7 mentions it for completeness; the reader who encounters `EINPROGRESS` in another driver will know what it means and where to look.

21.8.5 Failure Mode 5: Engine Reports Partial Transfer

The simulated engine does not report partial transfers in its baseline behaviour; it either completes the full `DMA_LEN` or sets `DMA_STATUS_ERR`. Real engines sometimes report a partial transfer: the engine copied fewer bytes than requested and signalled completion with the partial count in a length register.

A driver that sees a partial transfer must decide what to do: retry, treat as fatal, pass the partial result to user space with an error flag. The chapter's simulation can be extended to model partial transfers by having the callout set a smaller transferred-length field:

```
uint32_t actual_len = len;
/* For the lab in Section 7, force a partial transfer every 100 tries. */
if ((sim->dma_transfer_count++ % 100) == 0)
    actual_len = len / 2;
/* ... perform memcpy of actual_len bytes ... */
```

```
sim->dma.transferred_len = actual_len;
sim->dma.status = MYFIRST_DMA_STATUS_DONE;
```

The driver reads the length after the sync:

```
uint32_t xferred;
xferred = CSR_READ_4(sc, MYFIRST_REG_DMA_XFERRED);
if (xferred < expected_len) {
    device_printf(sc->dev,
        "partial DMA: requested %u, got %u\n",
        expected_len, xferred);
    atomic_add_64(&sc->dma_partials, 1);
    /* Decide: retry, report, ignore. */
}
```

Chapter 21's baseline code does not implement partial-transfer reporting; the simulated engine always completes the full length. Lab 5 in the Section 7 challenges adds the partial-transfer behaviour and walks through a retry strategy. The pattern is what matters, not the specific `xferred` register; every partial transfer has the same driver-side shape.

21.8.6 Failure Mode 6: Engine Never Signals Completion (Timeout)

The engine might set `STATUS_RUNNING` and never advance to `DONE`. In the simulation this happens if the callout is dropped (for example, the callout pool is exhausted or the callout is cancelled by a different path). On real hardware it happens if the device is wedged, the PCIe link is down, or the firmware has a bug.

Chapter 21's interrupt-driven helper (`myfirst_dma_do_transfer_intr`) uses `cv_timedwait` with a one-second timeout. If the wait returns `EWOULDBLOCK`, the transfer is considered hung. The driver's response:

```
if (err == EWOULDBLOCK) {
    CSR_WRITE_4(sc, MYFIRST_REG_DMA_CTRL, MYFIRST_DMA_CTRL_ABORT);
    if (direction == MYFIRST_DMA_DIR_WRITE)
        bus_dmamap_sync(sc->dma_tag, sc->dma_map,
            BUS_DMASYNC_POSTWRITE);
    else
        bus_dmamap_sync(sc->dma_tag, sc->dma_map,
            BUS_DMASYNC_POSTREAD);
    sc->dma_in_flight = false;
    atomic_add_64(&sc->dma_timeouts, 1);
    MYFIRST_UNLOCK(sc);
    return (ETIMEDOUT);
}
```

Four things happen: the engine is aborted (so it does not complete after the helper returns), the POST sync is issued (so the map is not left in a PRE state), `dma_in_flight` is cleared (so the next transfer can start), a counter is incremented (for observability). The helper then returns `ETIMEDOUT` to the caller.

A real driver would also consider whether to reset the device at this point. If timeouts are rare, abort-and-continue is appropriate; if they are frequent, a full reset may be needed. Chapter 21's pattern is abort-and-continue; the reset path is a Section 8 refactor exercise.

The chapter's simulation can be extended to never complete, to let the reader exercise the timeout path:

```
/* Comment out the callout_reset to make transfers hang: */
// callout_reset(&sim->dma.done_co, hz / 100, myfirst_sim_dma_done_co, sim);
```

After this change, every transfer hits the timeout. The driver's counters climb, the caller sees `ETIMEDOUT`, and `kldunload` still succeeds (because the abort path cleared `dma_in_flight` and the teardown has no in-flight transfers to wait for). Running this experiment once gives the reader a concrete sense of what a “stuck device” driver looks like from the outside.

21.8.7 Failure Mode 7: Detach While a Transfer Is In Flight

The detach path has to handle the case where a transfer is in flight at the moment the driver is unloaded. The risks are:

1. The callout fires after `myfirst_dma_tear_down` has unloaded the map, writing to freed memory.
2. The task runs after `myfirst_dma_tear_down` has destroyed the tag, syncing against a freed tag.
3. The helper is waiting on `dma_cv` at the moment the device is disappearing.

The Chapter 20 teardown ordering already addresses several of these. Chapter 21 adds two more barriers:

```
void
myfirst_detach_dma_path(struct myfirst_softc *sc)
{
    /* 1. Tell callers no new transfers may start. */
    MYFIRST_LOCK(sc);
    sc->detaching = true;
    MYFIRST_UNLOCK(sc);

    /* 2. If a transfer is in flight, wait for it to complete or time out. */
    MYFIRST_LOCK(sc);
    while (sc->dma_in_flight) {
        cv_timedwait(&sc->dma_cv, &sc->mtx, hz);
    }
    MYFIRST_UNLOCK(sc);

    /* 3. Mask the completion interrupt at the device. */
    CSR_WRITE_4(sc, MYFIRST_REG_INTR_MASK,
```

```

        CSR_READ_4(sc, MYFIRST_REG_INTR_MASK) &
        ~MYFIRST_INTR_COMPLETE);

    /* 4. Drain callouts and tasks. */
    callout_drain(&sc->sim.dma.done_co);
    taskqueue_drain(sc->intr_tq, &sc->vectors[MYFIRST_VECTOR_RX].task);

    /* 5. Tear down the DMA resources. */
    myfirst_dma_teardown(sc);
}

```

The five steps are in the order that keeps each subsequent step safe. Setting `detaching = true` prevents new transfers from starting (the helper checks it before arming). Waiting for in-flight transfers to finish or time out ensures no callout is scheduled and no task is pending. Masking the interrupt prevents any late completion from firing the filter. Draining the callout and the task ensures that any in-progress callback has finished. Tearing down the DMA resources is safe because nothing else can touch them now.

The first step (setting `detaching = true`) requires the transfer helper to check the flag:

```

if (sc->detaching) {
    MYFIRST_UNLOCK(sc);
    return (ENXIO);
}

```

This is a small addition to the helper and prevents the rare race where a user-space test fires a transfer while detach is running.

21.8.8 Failure Mode 8: Bounce Buffer Exhaustion

On 32-bit systems or on 64-bit systems with 32-bit-only devices, `bus_dma` may need to allocate bounce buffers to satisfy the address constraint. The bounce pool has a fixed size; if it is exhausted, `bus_dmamap_load` with `BUS_DMA_NOWAIT` returns `ENOMEM`.

Chapter 21's simulation has no real address constraints and the chapter's primary path does not hit this mode. Production drivers for 32-bit-capable devices on 64-bit systems must handle it:

```

err = bus_dmamap_load(sc->dma_tag, map, buf, len,
    my_callback, sc, BUS_DMA_NOWAIT);
if (err == ENOMEM) {
    /* The bounce pool is exhausted. Options:
     * - Retry later (queue the request).
     * - Fail this transfer and let the caller retry.
     * - Allocate a fresh buffer inside the device's address range. */
    ...
}

```

The practical mitigation is usually to allocate all buffers inside the device's addressable range (so bounce buffers are never needed). That is what `bus_dmamem_alloc` with an appropriate tag does automatically: the allocator sees the tag's `highaddr` (or `lowaddr`, depending on which side of the window the address constraint is expressed) and allocates inside the range. A driver that uses `bus_dmamem_alloc` for its DMA buffers never sees bounce-buffer exhaustion; a driver that uses `bus_dmamap_load` on arbitrary kernel memory (say, an mbuf from the network stack) can.

Chapter 21's static pattern is immune. The dynamic pattern discussed in Part 6 (Chapter 28) is not, and the network-driver chapter there covers the retry strategies in detail.

21.8.9 Failure Mode 9: Unaligned Buffer

The tag's `alignment` parameter describes the required alignment of DMA addresses. If the buffer's address is not aligned, the load fails.

For Chapter 21's `bus_dmamem_alloc`-allocated buffer, the allocator always returns an aligned buffer; the failure does not occur on the static path. For dynamic loads of arbitrary buffers, the driver must either ensure the buffer is aligned (by copying it into an aligned temp buffer) or rely on bounce-buffer machinery to do the alignment automatically. The kernel handles this transparently on `bus_dmamap_load_mbuf_sg` (the mbuf machinery produces aligned segments via bounce); for raw `bus_dmamap_load` the driver is on its own.

The chapter's simulation does not model alignment constraints beyond accepting `alignment = 4`. A Section 7 challenge invites the reader to set `alignment = 64`, observe that `bus_dmamem_alloc` still returns an aligned buffer (because allocators are alignment-aware), and then try loading a deliberately misaligned buffer to see the failure.

21.8.10 Failure Mode 10: The Load Callback Runs with Non-Zero error

`bus_dmamap_load` may call the callback with `error = EFBIG` if the load is logically valid but cannot be achieved with the tag's segment constraints. The callback should handle this:

```
static void
myfirst_dma_single_map(void *arg, bus_dma_segment_t *segs, int nseg, int error)
{
    bus_addr_t *addr = arg;

    if (error != 0) {
        printf("myfirst: dma load callback error %d\n", error);
        /* Do not write *addr; the caller checks for zero. */
        return;
    }

    KASSERT(nseg == 1, ("myfirst: unexpected DMA segment count %d", nseg));
    *addr = segs[0].ds_addr;
}
```

The caller then detects the failure by checking the bus-address output:

```
sc->dma_bus_addr = 0;
err = bus_dmamap_load(...);
if (err != 0 || sc->dma_bus_addr == 0) {
    /* Failure: either the load returned non-zero, or the callback
     * ran with error != 0 and did not populate the address. */
    ...
}
```

The two checks together cover all failure modes: synchronous error from the load (`err != 0`), callback error with the load returning zero (`dma_bus_addr == 0`). Chapter 21's Stage 1 code has both checks and handles both cases uniformly.

21.8.11 Failure Mode 11: A Sync Call Is Skipped

A driver that skips a sync call does not fail immediately on coherent platforms; the bug is latent. Section 4 already covered this. The key point for Section 7 is that there is no automatic detection for missing syncs; the driver must be structured so that syncs are impossible to forget. Chapter 21's pattern (the PRE is always the first action in the helper; the POST is always the last) is one way. Another is to use a wrapper function that does the sync around a device-specific callback; this is what iflib and some other frameworks do internally.

WITNESS does catch certain sync-related mistakes. If the driver holds the wrong lock when calling a `bus_dmamap_*` function that the tag's `lockfunc` expects, WITNESS warns. If the driver calls a sync from a context where `busdma_lock_mutex` would try to acquire a lock, WITNESS catches the mismatch. Running the driver under WITNESS regularly is the best defence against sync-related latent bugs.

21.8.12 Failure Mode 12: The Callback Sets *addr to Zero and the Driver Proceeds

A subtle variant of Failure Mode 10. The callback runs, the load returns zero, the error argument to the callback happens to be zero, but `nseg == 0`. This cannot happen for `bus_dmamem_alloc`-backed buffers (which always produce one segment), but it can happen for some corner cases of arbitrary loads. The pattern is:

```
static void
myfirst_dma_single_map(void *arg, bus_dma_segment_t *segs, int nseg, int error)
{
    bus_addr_t *addr = arg;

    if (error != 0) {
        printf("myfirst: dma load callback error %d\n", error);
        *addr = 0;
        return;
    }
}
```

```
if (nseg != 1) {
    printf("myfirst: unexpected DMA segment count %d\n", nseg);
    *addr = 0;
    return;
}

*addr = segs[0].ds_addr;
}
```

Explicit zeroing on every failure path keeps the caller's error detection reliable. The stricter version is the right default; the `KASSERT` in the Stage 1 code is a debug-only check, and the runtime check is what production code should use. The chapter's example code uses both: `KASSERT` in debug builds, explicit zeroing as a fallback.

21.8.13 A Pattern: The `dma_setup` / `dma_tearardown` Pair

Section 8 moves the DMA code into its own file, but Section 7 is the right place to name the pattern that motivates the refactor. Every DMA-capable driver ends up with a pair of functions: one that performs all the tag/map/memory/load operations in the correct order with full error unwinding, and one that reverses them in the correct order. The pair is the driver's DMA ABI: the rest of the driver only needs to call `dma_setup` at attach and `dma_tearardown` at detach. The functions themselves handle all the intermediate error paths.

Chapter 21's pair is `myfirst_dma_setup` and `myfirst_dma_tearardown`. Production drivers often have per-subsystem pairs (`re_dma_alloc` and `re_dma_free` for `if_re`, `nvme_qpair_dma_create` and `nvme_qpair_dma_destroy` for NVMe). The shape is always the same; the depth is what varies with the driver's complexity.

21.8.14 Wrapping Up Section 7

Section 7 walked through twelve failure modes, each with its pattern. The patterns are: check every return value, unwind every prior allocation, explicitly zero output fields on failure, drain callouts and tasks before tearing down resources, mask interrupts before destroying associated DMA state, and keep the setup/teardown pair symmetric so the detach path is the reverse of the attach path. Every production DMA driver follows these patterns; the chapter's code provides a template for applying them.

Section 8 is the final refactor. The DMA code moves into `myfirst_dma.c` and `myfirst_dma.h`; the Makefile updates; the version bumps to 1.4-dma; the `DMA.md` document captures the design. The chapter's running driver is then ready for Chapter 22's power-management work and Chapter 28's descriptor-ring extension.

21.9 Section 8: Refactoring and Versioning Your DMA-Capable Driver

Section 7 closed the functional scope. Section 8 is the housekeeping pass. The DMA code has accumulated in `myfirst.c`, the simulation backend's DMA extension has grown inside `myfirst_sim.c`, the softc has grown several new fields, the sysctls have grown, and the interrupt filter has gained a `MYFIRST_INTR_COMPLETE` branch. This section collects the DMA-specific code into `myfirst_dma.c` and `myfirst_dma.h`, cleans up the naming,

updates the Makefile, bumps the version, adds the `DMA.md` document, and runs the final regression pass. Version tag `1.4-dma`.

The refactor is small in absolute terms (about 200 lines of code moved, plus a new header with public function prototypes and macros), but the structural benefit is large: the driver's architecture now shows, at a glance, that DMA is a first-class subsystem with its own file, its own public API, and its own documentation.

21.9.1 The Final File Layout

After Section 8, the driver's file layout is:

```
myfirst.c      # Top-level: attach/detach, cdev, ioctl
myfirst.h      # Shared macros, softc struct, public prototypes
myfirst_hw.c   # Chapter 16: register accessor layer
myfirst_hw.h
myfirst_hw_pci.c # Chapter 18: real PCI backend
myfirst_sim.c  # Chapter 17: simulated backend (now includes DMA engine)
myfirst_sim.h
myfirst_pci.c  # Chapter 18: PCI attach/detach
myfirst_pci.h
myfirst_intr.c # Chapter 19: legacy interrupt path
myfirst_intr.h
myfirst_msix.c # Chapter 20: MSI/MSI-X path
myfirst_msix.h
myfirst_dma.c  # Chapter 21: DMA setup/teardown/transfer
myfirst_dma.h
myfirst_sync.h # Chapter 11: locking macros
cbuf.c        # Chapter 15: circular buffer
cbuf.h
```

Fifteen source files plus shared headers. Each file has a narrow responsibility. The refactor maintains this separation: `myfirst_dma.c` is self-contained and only depends on `myfirst.h` and the kernel's public headers.

21.9.2 The `myfirst_dma.h` Header

The header declares the public DMA API and the shared constants:

```
/* myfirst_dma.h */
#ifndef _MYFIRST_DMA_H_
#define _MYFIRST_DMA_H_

/* DMA buffer size used by myfirst. Matches the Chapter 21 simulated
 * engine's scratch size. A real device would use a value from the
 * hardware's documented capabilities. */
```

```

#define MYFIRST_DMA_BUFFER_SIZE      4096u

/* DMA register offsets (relative to the BAR base). */
#define MYFIRST_REG_DMA_ADDR_LOW     0x20
#define MYFIRST_REG_DMA_ADDR_HIGH    0x24
#define MYFIRST_REG_DMA_LEN          0x28
#define MYFIRST_REG_DMA_DIR           0x2C
#define MYFIRST_REG_DMA_CTRL          0x30
#define MYFIRST_REG_DMA_STATUS        0x34

/* DMA_DIR values. */
#define MYFIRST_DMA_DIR_WRITE         0u /* host-to-device */
#define MYFIRST_DMA_DIR_READ          1u /* device-to-host */

/* DMA_CTRL bits. */
#define MYFIRST_DMA_CTRL_START        (1u << 0)
#define MYFIRST_DMA_CTRL_ABORT        (1u << 1)
#define MYFIRST_DMA_CTRL_RESET        (1u << 31)

/* DMA_STATUS bits. */
#define MYFIRST_DMA_STATUS_DONE        (1u << 0)
#define MYFIRST_DMA_STATUS_ERR         (1u << 1)
#define MYFIRST_DMA_STATUS_RUNNING     (1u << 2)

/* Public API. */
struct myfirst_softc;

int myfirst_dma_setup(struct myfirst_softc *sc);
void myfirst_dma_teardown(struct myfirst_softc *sc);
int myfirst_dma_do_transfer(struct myfirst_softc *sc,
    int direction, size_t length);
void myfirst_dma_handle_complete(struct myfirst_softc *sc);
void myfirst_dma_add_sysctls(struct myfirst_softc *sc);

#endif /* _MYFIRST_DMA_H_ */

```

Five public functions. `myfirst_dma_setup` is called once from `attach`; `myfirst_dma_teardown` is called once from `detach`; `myfirst_dma_do_transfer` is called by `sysctl` handlers or by other parts of the driver that want to trigger a DMA; `myfirst_dma_handle_complete` is called from the rx task when `MYFIRST_INTR_COMPLETE` was observed; `myfirst_dma_add_sysctls` registers the DMA counters and the test `sysctls`.

The header uses forward declaration (`struct myfirst_softc`) to avoid circular includes. The implementation sees the full `softc` definition via `myfirst.h`.

21.9.3 The `myfirst_dma.c` File

The file gathers the DMA code that has accumulated across Sections 3 through 7. The top of the file includes the standard headers the DMA API needs:

```
/* myfirst_dma.c */
#include <sys/param.h>
#include <sys/system.h>
#include <sys/kernel.h>
#include <sys/bus.h>
#include <sys/module.h>
#include <sys/rman.h>
#include <sys/lock.h>
#include <sys/mutex.h>
#include <sys/condvar.h>
#include <sys/sysctl.h>

#include <machine/bus.h>
#include <machine/resource.h>

#include "myfirst.h"
#include "myfirst_hw.h"
#include "myfirst_dma.h"
```

The implementation then defines the single-segment callback, `myfirst_dma_setup`, `myfirst_dma_tear_down`, `myfirst_dma_do_transfer`, `myfirst_dma_handle_complete`, and `myfirst_dma_add_sysctls`. Each function is lifted from its earlier section with no behaviour change; the refactor is purely a location move.

The one notable addition is `myfirst_dma_handle_complete`, which centralises the POST sync that Section 6's task performed inline:

```
void
myfirst_dma_handle_complete(struct myfirst_softc *sc)
{
    MYFIRST_ASSERT_LOCKED(sc);

    if (!sc->dma_in_flight)
        return;

    sc->dma_in_flight = false;
    if (sc->dma_last_direction == MYFIRST_DMA_DIR_WRITE)
        bus_dmamap_sync(sc->dma_tag, sc->dma_map,
            BUS_DMASYNC_POSTWRITE);
    else
        bus_dmamap_sync(sc->dma_tag, sc->dma_map,
```

```

        BUS_DMASYNC_POSTREAD));

sc->dma_last_status = CSR_READ_4(sc, MYFIRST_REG_DMA_STATUS);
atomic_add_64(&sc->dma_complete_tasks, 1);
cv_broadcast(&sc->dma_cv);
}

```

The rx task's body becomes:

```

static void
myfirst_msix_rx_task_fn(void *arg, int npending)
{
    struct myfirst_vector *vec = arg;
    struct myfirst_softc *sc = vec->sc;

    MYFIRST_LOCK(sc);
    if (sc->hw == NULL || !sc->pci_attached) {
        MYFIRST_UNLOCK(sc);
        return;
    }
    sc->intr_last_data = CSR_READ_4(sc, MYFIRST_REG_DATA_OUT);
    sc->intr_task_invocations++;
    cv_broadcast(&sc->data_cv);
    myfirst_dma_handle_complete(sc);
    MYFIRST_UNLOCK(sc);
}

```

The task is shorter by four lines, and the DMA-specific logic lives entirely in `myfirst_dma.c`. If a later chapter (or a future contributor) wants to change the POST-sync or completion behaviour, the change touches one file.

21.9.4 The Updated Makefile

The Makefile from Chapter 20 was:

```

KMOD=  myfirst
SRCS=  myfirst.c \
        myfirst_hw.c myfirst_hw_pci.c \
        myfirst_sim.c \
        myfirst_pci.c \
        myfirst_intr.c \
        myfirst_msix.c \
        cbuf.c

```

```
CFLAGS+= -DMYFIRST_VERSION_STRING=\"1.3-msi\"  
  
.include <bsd.kmod.mk>
```

The Chapter 21 Stage 4 Makefile adds `myfirst_dma.c` and bumps the version string:

```
KMOD= myfirst  
SRCS= myfirst.c \  
      myfirst_hw.c myfirst_hw_pci.c \  
      myfirst_sim.c \  
      myfirst_pci.c \  
      myfirst_intr.c \  
      myfirst_msix.c \  
      myfirst_dma.c \  
      cbuf.c  
  
CFLAGS+= -DMYFIRST_VERSION_STRING=\"1.4-dma\"  
  
.include <bsd.kmod.mk>
```

The change is two lines: the added source file in `SRCS` and the updated version string. Both are required; a Makefile missing the new source fails to link because `myfirst.c` calls `myfirst_dma_setup` which is not in any compiled object.

21.9.5 The Version String in `kldstat`

The `MYFIRST_VERSION_STRING` macro is used in `MODULE_VERSION` and in an `SYSCTL_STRING` exposing the version. After the refactor, `kldstat -v | grep myfirst` reports:

```
myfirst 1.4-dma (pseudo-device)
```

and `sysctl dev.myfirst.0.version` returns the same string. Operators who see the driver attached can check the version at a glance; the string is a useful diagnostic when mixing development and production driver versions on the same test system.

21.9.6 The `DMA.md` Document

Chapter 20 introduced `MSIX.md`. Chapter 21 adds `DMA.md`, a one-file reference that documents the DMA subsystem's public API, the register layout, the flow diagrams, and the counters. A sample outline:

```
# DMA Subsystem
```

Purpose

The DMA layer allows the driver to transfer data between host memory and the device without CPU-per-byte involvement. It is used for every transfer larger than a few words; smaller register reads and writes still use MMIO directly.

Public API

- ``myfirst_dma_setup(sc)``: called from attach. Creates the tag, allocates the buffer, loads the map, populates ``sc->dma_bus_addr``.
- ``myfirst_dma_teardown(sc)``: called from detach. Reverses setup.
- ``myfirst_dma_do_transfer(sc, dir, len)``: triggers one DMA transfer and waits for completion.
- ``myfirst_dma_handle_complete(sc)``: called from the rx task when ``MYFIRST_INTR_COMPLETE`` was observed.
- ``myfirst_dma_add_sysctls(sc)``: registers the DMA counters and test sysctls under ``dev.myfirst.N.``.

Register Layout

... (table from Section 5) ...

Flow Diagrams

Host-to-device:

... (diagram from Section 4, Pattern A) ...

Device-to-host:

... (diagram from Section 4, Pattern B) ...

Counters

- ``dev.myfirst.N.dma_transfers_write``: successful host-to-device transfers.
- ``dev.myfirst.N.dma_transfers_read``: successful device-to-host transfers.
- ``dev.myfirst.N.dma_errors``: transfers that returned EIO.
- ``dev.myfirst.N.dma_timeouts``: transfers that hit the 1-second timeout.
- ``dev.myfirst.N.dma_complete_intrs``: completion-bit observations in the filter.
- ``dev.myfirst.N.dma_complete_tasks``: completion processing in the task.

Observability

``sysctl dev.myfirst.N.dma_*`` returns the full counter set. A healthy driver has ``dma_complete_intrs == dma_complete_tasks`` and both equal

```
to `dma_transfers_write + dma_transfers_read + dma_errors + dma_timeouts`.
```

Testing

The sysctls ``dma_test_write`` and ``dma_test_read`` trigger transfers from user space. Writing any value to ``dma_test_write`` fills the buffer with the low byte of the value and runs a host-to-device transfer; writing any value to ``dma_test_read`` runs a device-to-host transfer and logs the first eight bytes to ``dmesg``.

Known Limitations

- Single buffer, single transfer at a time.
- No descriptor-ring support (Part 6, Chapter 28).
- No per-NUMA-node allocation (Part 7, Chapter 33).
- No partial-transfer reporting (exercise for the reader).

See Also

- ``bus_dma(9)``, ``/usr/src/sys/sys/bus_dma.h``.
- ``/usr/src/sys/dev/re/if_re.c`` for a production descriptor-ring driver.
- ``INTERRUPTS.md``, ``MSIX.md`` for the interrupt path the completion uses.

The document is about a page long and useful both as a contributor reference and as a reader's recap. Real production drivers often have their README files structured similarly: subsystem-by-subsystem, with a short flow diagram and a counter list.

21.9.7 The Regression Pass

The chapter's regression test from Chapter 20 is extended with DMA-specific checks. The full test (call it `full_regression_ch21.sh`) runs these steps on a fresh boot:

1. Kernel is in the expected state (`uname -v` shows 1.4-dma in the `myfirst` line, `INVARIANTS/WITNESS` in the config).
2. Load the module (`kldload ./myfirst.ko`). `dmesg` shows the DMA banner and the interrupt-mode banner.
3. Verify initial counter state (`dma_transfers_write == 0`, etc.).
4. Run 1000 write transfers via the sysctl. Assert counter increments.
5. Run 1000 read transfers via the sysctl. Assert counter increments.
6. Run 100 transfers with the simulated error injection enabled. Assert `dma_errors` counter climbs.
7. Run 10 transfers with the callout disabled (engine hangs). Assert `dma_timeouts` counter climbs and `dma_in_flight` is cleared after each timeout.
8. Unload the module (`kldunload myfirst`). Assert clean unload with no `WITNESS` warnings.
9. Repeat the load/unload cycle 50 times to detect leaks.

The full script lives under `examples/part-04/ch21-dma/labs/full_regression_ch21.sh`. A successful run prints a single line per step with the observed counters and an overall PASS at the end. Any step that fails prints a FAIL line with the actual versus expected values.

21.9.8 What the Refactor Accomplished

After Section 8:

- The DMA code lives in its own file with a documented public API.
- The Makefile knows about the new source.
- The version tag reflects the chapter's work.
- A `DMA.md` document serves as a reference for contributors and future chapters.
- The regression test covers every DMA path the chapter teaches.

The driver is now `1.4-dma`, and the diff from `1.3-msi` is about four hundred lines added plus fifty lines moved. Every line is traceable: either it implements a concept from one of the earlier sections or it is housekeeping (headers, Makefile, documentation).

21.9.9 Exercise: Create a Utility That Verifies DMA Data

A hands-on exercise for Section 8. Write a small user-space utility that exercises the DMA path and verifies the data end-to-end.

The utility's job: 1. Open the `dev.myfirst.0.dma_test_write` sysctl, write a known pattern (say, `0xA5`). 2. Open `dev.myfirst.0.dma_test_read`, trigger a read. 3. Check `dmesg` (or a custom sysctl that exposes the first bytes of the buffer) for the expected pattern (`0x5A` from the simulator's scratch). 4. Repeat 100 times with different patterns; report a summary.

The utility is a shell script for simplicity:

```
#!/bin/sh
fail=0
for pat in $(jot 100 1); do
    hex=$(printf "0x%02x" $pat)
    sysctl -n dev.myfirst.0.dma_test_write=$pat >/dev/null
    sysctl -n dev.myfirst.0.dma_test_read=1 >/dev/null
    # ... check the result ...
done
echo "failures: $fail"
```

The exercise is open-ended: the reader can extend it with timing measurements, error-rate estimates, or a comparison against `dma_errors` counter values. Section 8's challenge list suggests a few directions.

21.9.10 Wrapping Up Section 8

Section 8 is the finish line. The DMA code has its own file; the Makefile compiles it; the version tag says `1.4-dma`; the documentation captures the design; the regression test exercises every path. The chapter's running

driver is now ready for Chapter 22's power-management work, which will use the DMA teardown path to quiesce transfers during suspend, and ready for Chapter 28, which will extend the single-buffer pattern to a full descriptor ring.

The chapter's sections are complete. The remaining material is reference and practice: a walkthrough of a production DMA driver (`if_re`), deeper looks at bounce buffers, 32-bit devices, IOMMU, coherent flags, and parent-tag inheritance, the labs, the challenges, the troubleshooting reference, and the final wrap-up.

21.10 Reading a Real Driver Together: `if_re.c`

A tour of `/usr/src/sys/dev/re/if_re.c` that maps its DMA code back to the concepts in Chapter 21. `if_re(4)` is the driver for the RealTek 8139C+ / 8169 / 8168 family of Gigabit Ethernet chips, which is common enough to be present in many lab environments and compact enough (around four thousand lines total) to read in a week of commute-length sessions. Its DMA code is about four hundred lines and sits inside a few well-named functions, so it is a good first real-driver read.

Reading this walkthrough. The listings in the subsections that follow are abbreviated excerpts from `re_allocmem()`, `re_dma_map_addr()`, `re_encap()`, and `re_detach()` in `/usr/src/sys/dev/re/if_re.c`. We have preserved each call's argument list and the surrounding control flow, but we show only the fragments that illustrate a `bus_dma(9)` idea under discussion; the real functions include more error handling, more child tags, and more bookkeeping. Every symbol the listings name, from `bus_dma_tag_create` to `bus_dmamap_load` to `re_dma_map_addr`, is a real FreeBSD identifier you can find with a symbol search. The same abbreviation convention appears in the `nvme(4)` driver's `nvme_single_map` in `/usr/src/sys/dev/nvme/nvme_private.h` and in any driver that packages a single-segment callback as a helper.

21.10.1 The Parent Tag

`re_allocmem` begins by creating the parent tag:

```
lowaddr = BUS_SPACE_MAXADDR;
if ((sc->rl_flags & RL_FLAG_PCIE) == 0)
    lowaddr = BUS_SPACE_MAXADDR_32BIT;
error = bus_dma_tag_create(bus_get_dma_tag(dev), 1, 0,
    lowaddr, BUS_SPACE_MAXADDR, NULL, NULL,
    BUS_SPACE_MAXSIZE_32BIT, 0, BUS_SPACE_MAXSIZE_32BIT, 0,
    NULL, NULL, &sc->rl_parent_tag);
```

Two things worth noticing.

First, the `lowaddr` decision. On PCIe variants of the chip, DMA can reach all of memory (`BUS_SPACE_MAXADDR`). On older non-Pcie variants, the chip is 32-bit-only (`BUS_SPACE_MAXADDR_32BIT`, i.e. the device cannot reach memory above 4 GB). The driver probes this at attach time and sets `lowaddr` accordingly. Any child tag created from `rl_parent_tag` inherits this limit automatically. On a 64-bit amd64 system with 16 GB of RAM,

the driver's tags ensure that DMA buffers are allocated below 4 GB for the non-PCIe variants; bounce buffers kick in only if the kernel cannot satisfy that allocation.

Second, the `maxsize = BUS_SPACE_MAXSIZE_32BIT` and `nsegments = 0`. These values produce a very permissive parent tag: any child with any segment count is valid. This is a common idiom; the parent carries only the addressing constraint, and children carry the specifics.

21.10.2 The Per-Buffer Child Tags

The driver creates four child tags:

- `rl_tx_mtag`: for transmit mbuf payloads. Alignment 1, max size `MCLBYTES * RL_NTXSEGS`, up to `RL_NTXSEGS` segments per mapping.
- `rl_rx_mtag`: for receive mbuf payloads. Alignment 8, max size `MCLBYTES`, 1 segment per mapping.
- `rl_tx_list_tag`: for the transmit descriptor ring. Alignment `RL_RING_ALIGN` (256 bytes), max size `tx_list_size`, 1 segment.
- `rl_rx_list_tag`: for the receive descriptor ring. Alignment `RL_RING_ALIGN`, max size `rx_list_size`, 1 segment.

Each child passes `sc->rl_parent_tag` as the parent. Each carries its own alignment and segment constraints. The kernel composes the parent's 32-bit addressing limit with each child's own constraints; the driver does not have to repeat the addressing limit in every child tag.

The descriptor-ring tags are single-segment: the ring is allocated contiguously. The mbuf-payload tags are multi-segment: a single received packet may be a chain of mbufs, and the NIC's DMA engine is fine with a scatter-gather list of segments as long as the total fits within the maximum size.

21.10.3 The Ring Allocation

The descriptor ring allocation uses `bus_dmamem_alloc` with `BUS_DMA_COHERENT`:

```
error = bus_dmamem_alloc(sc->rl_ldata.rl_tx_list_tag,  
    (void **)&sc->rl_ldata.rl_tx_list,  
    BUS_DMA_WAITOK | BUS_DMA_COHERENT | BUS_DMA_ZERO,  
    &sc->rl_ldata.rl_tx_list_map);
```

The flags are `WAITOK` | `COHERENT` | `ZERO`: allow sleeping, prefer coherent memory, zero-initialise. The coherent hint is appropriate because the ring is on the hot path for both the CPU (producing TX entries) and the NIC (consuming them, writing back status). The zero-initialise is important because the ring's initial state must have all entries marked "not in use"; uninitialised memory could cause the NIC to attempt DMA based on garbage descriptor contents.

After `bus_dmamem_alloc`, the driver calls `bus_dmamap_load` with a single-segment callback:

```
sc->rl_ldata.rl_tx_list_addr = 0;  
error = bus_dmamap_load(sc->rl_ldata.rl_tx_list_tag,
```

```
sc->rl_ldata.rl_tx_list_map, sc->rl_ldata.rl_tx_list,
tx_list_size, re_dma_map_addr,
&sc->rl_ldata.rl_tx_list_addr, BUS_DMA_NOWAIT);
```

And the callback:

```
static void
re_dma_map_addr(void *arg, bus_dma_segment_t *segs, int nseg, int error)
{
    bus_addr_t *addr;

    if (error)
        return;

    KASSERT(nseg == 1, ("too many DMA segments, %d should be 1", nseg));
    addr = arg;
    *addr = segs->ds_addr;
}
```

The callback is almost identical to Chapter 21's `myfirst_dma_single_map`. The driver's code is portable across every DMA-capable FreeBSD platform because the callback, the tag, and the allocator all respect the `bus_dma` discipline.

21.10.4 The Per-Transfer Pattern

For each transmitted packet, the driver calls `bus_dmamap_load_mbuf_sg` on the packet's mbuf:

```
error = bus_dmamap_load_mbuf_sg(sc->rl_ldata.rl_tx_mtag, txd->tx_dmamap,
    *m_head, segs, &nsegs, BUS_DMA_NOWAIT);
```

The `_mbuf_sg` variant fills a pre-allocated segment array, avoiding the callback for the common case. If the load returns `EFBIG`, the driver tries to compact the mbuf chain with `m_collapse` and retries.

Once the load succeeds, the driver syncs:

```
bus_dmamap_sync(sc->rl_ldata.rl_tx_mtag, txd->tx_dmamap,
    BUS_DMASYNC_PREWRITE);
```

Then writes the descriptors with the segment addresses, pokes the doorbell, and lets the NIC transmit. On completion, the driver syncs again:

```
bus_dmamap_sync(sc->rl_ldata.rl_tx_list_tag, sc->rl_ldata.rl_tx_list_map,  
    BUS_DMASYNC_POSTREAD | BUS_DMASYNC_POSTWRITE);
```

The combined POSTREAD|POSTWRITE sync is the descriptor-ring variant from Section 4: the driver both wrote the descriptor (the address, length, flags) and wants to read the device's status update (transmitted, error, etc.).

21.10.5 The Cleanup

The detach path (`re_dma_free`) mirrors the allocation order, in reverse:

```
if (sc->rl_ldata.rl_rx_list_tag) {  
    if (sc->rl_ldata.rl_rx_list_addr)  
        bus_dmamap_unload(sc->rl_ldata.rl_rx_list_tag,  
            sc->rl_ldata.rl_rx_list_map);  
    if (sc->rl_ldata.rl_rx_list)  
        bus_dmamem_free(sc->rl_ldata.rl_rx_list_tag,  
            sc->rl_ldata.rl_rx_list, sc->rl_ldata.rl_rx_list_map);  
    bus_dma_tag_destroy(sc->rl_ldata.rl_rx_list_tag);  
}
```

The per-buffer dmamaps created with `bus_dmamap_create` are destroyed with `bus_dmamap_destroy` in a loop, and finally the tags are destroyed.

21.10.6 What the Walkthrough Teaches

`if_re`'s DMA code is a near-textbook application of Chapter 21's concepts. Every call the chapter teaches appears in the driver. The only concepts not in the chapter are the scatter-gather variant (`bus_dmamap_load_mbuf_sg`) and the per-buffer dynamic maps (`bus_dmamap_create`/`bus_dmamap_destroy`), and both are natural extensions of the static pattern.

A reader who finishes Chapter 21 and then reads `if_re.c` for half an hour sees the same shapes. The chapter's investment pays off against real driver code.

21.11 A Deeper Look at Bounce Buffers

Section 1 introduced bounce buffers; Section 7 described the error mode. This deeper look pins down what bounce buffers are, when they are used, and what the cost looks like.

21.11.1 What Bounce Buffers Are

A bounce buffer is a region of physical memory inside the device's addressable range, used to stage data when the driver's actual buffer is outside the range. The `bus_dma` layer manages a pool of bounce pages as an internal

resource; a driver that allocates buffers with `bus_dmamem_alloc` inside the device's range never touches the pool.

When a driver loads a buffer via `bus_dmamap_load` and the buffer is outside the device's range (say, a mbuf allocated above 4 GB on a 32-bit-only device), the kernel:

1. Allocates a bounce page inside the device's range.
2. At `PREWRITE` time, copies the driver's buffer into the bounce page.
3. Programs the device with the bounce page's address.
4. At `POSTREAD` time, copies the bounce page back to the driver's buffer.
5. Releases the bounce page.

The driver never knows that a bounce happened. The `bus_addr_t` returned by the callback is the bounce page's address; the sync operations move the data back and forth.

21.11.2 When Bounce Buffers Matter

Bounce buffers are needed in three situations:

1. **32-bit-only devices on systems with more than 4 GB of RAM.** The device cannot reach above 4 GB; any buffer up there needs a bounce below 4 GB.
2. **Devices with addressing holes.** Some legacy devices have unusable address regions (for example, the ISA-style 16 MB DMA limit). A buffer inside the hole needs to bounce.
3. **IOMMU-enabled systems with devices that are not IOMMU-registered.** If the IOMMU is in enforcing mode and a device has not been mapped, the DMA target is bounced to a mapped region.

On modern amd64 with 64-bit-capable devices and 64-bit-capable drivers (which most are), bounce buffers are rare. On embedded systems or in compatibility contexts, they are common.

21.11.3 The Performance Cost

Each bounce involves a `memcpy` from the driver's buffer to the bounce region (at `PREWRITE`) and another `memcpy` back (at `POSTREAD`). For a 1500-byte mbuf this is about 3 KB of memory traffic per packet; on a 10 Gbps NIC receiving a million packets per second, that is 3 GB/s of bounce traffic, which measurably loads the memory bus.

The bounce buffer pool is also sized (tunable via `hw.busdma.* sysctls`, with defaults that scale with physical memory). On a system where many drivers are bouncing, the pool can be exhausted, at which point loads with `BUS_DMA_NOWAIT` return `ENOMEM` and loads without `NOWAIT` defer.

21.11.4 How to Avoid Bouncing

Three strategies:

1. **Allocate DMA memory with `bus_dmamem_alloc`.** The allocator respects the tag's addressing constraints by construction.

2. **Set the tag's `highaddr/lowaddr` to match the device's true range.** Do not pass `BUS_SPACE_MAXADDR` for a 32-bit-only device; the incorrect value leads the allocator to hand out too-high buffers and forces bouncing.
3. **Use `BUS_DMA_ALLOCNOW`.** This pre-allocates the bounce pool at tag creation, turning a later allocation failure into an immediate failure the driver can handle at attach time.

Chapter 21's simulation does not exercise bouncing, but the concept is important for understanding real driver behaviour. A driver that “works on my laptop” but corrupts data on a server with 64 GB of RAM almost certainly has an address-range misconfiguration that is triggering silent bouncing on the laptop and failing differently on the server.

21.11.5 Observability

The `hw.busdma` sysctl tree exposes bounce-related counters:

```
sysctl hw.busdma
```

The interesting lines include `total_bpages`, `free_bpages`, `total_bounced`, `total_deferred`, `lowpriority_bounces`. `total_bounced` is the total number of pages bounced since boot; a non-zero value on a system where no bouncing should happen is a clue that some driver's tag is misconfigured.

21.12 A Deeper Look at 32-Bit-Capable Devices

Section 1 and the bounce-buffer discussion both touched on this; this deeper look collects the practical guidance.

21.12.1 The Setting

Some PCI and PCIe devices have 32-bit DMA engines. The engines accept only 32-bit bus addresses, so the DMA traffic is confined to the lower 4 GB of bus address space. On a 64-bit host with more than 4 GB of RAM, every DMA buffer must be below the 4 GB boundary.

21.12.2 The Tag Setting

The relevant constants:

- `BUS_SPACE_MAXADDR`: no addressing limit. Use for 64-bit-capable devices.
- `BUS_SPACE_MAXADDR_32BIT`: `0xFFFFFFFF`. Use as the `lowaddr` for 32-bit-only devices.
- `BUS_SPACE_MAXADDR_24BIT`: `0x00FFFFFF`. Use as the `lowaddr` for legacy devices with a 16 MB limit (rare).

Together, `lowaddr` and `highaddr` describe the *excluded* window of bus address space. The manual page wording is: “The window contains all addresses greater than `lowaddr` and less than or equal to `highaddr`.” Addresses inside the window cannot be reached by the device and must be bounced; addresses outside the window are reachable.

For a 32-bit device, the excluded window is everything above 4 GB, so `lowaddr = BUS_SPACE_MAXADDR_32BIT` and `highaddr = BUS_SPACE_MAXADDR`. The window becomes `(0xffffffff, BUS_SPACE_MAXADDR]`, which captures exactly “anything above 4 GB”.

For a 64-bit-capable device with no restrictions, the excluded window should be empty. The idiomatic way to express that is `lowaddr = BUS_SPACE_MAXADDR` and `highaddr = BUS_SPACE_MAXADDR`, which collapses the window to `(BUS_SPACE_MAXADDR, BUS_SPACE_MAXADDR]`, an empty range.

The naming is confusing at first: `lowaddr` sounds like a lower bound but is actually the *upper end* of the device’s reachable range (and the *lower end* of the excluded window). The naming is historical: `lowaddr` is the low end of the excluded region, and the excluded region is above the device’s range. The mnemonic: `lowaddr` is “the last address the device can reach”.

21.12.3 Double-Addressing-Cycle (DAC)

PCI has a mechanism called Double Address Cycle (DAC) that allows 32-bit slots to address 64 bits by issuing two cycles. Some devices that appear 32-bit actually support DAC and can reach 64-bit addresses. The `if_re` driver checks a chip-family flag to decide: PCIe variants use full 64-bit; non-PCIe variants use 32-bit only. A driver that does not know whether its device supports DAC should default to 32-bit (safer) and enable 64-bit if the datasheet confirms support.

21.12.4 The Impact on Bounce Traffic

A 32-bit-only device on a 64 GB server: - Most buffers are allocated above 4 GB by the default kernel allocator (since most memory is up there). - Every load triggers a bounce. - The bounce pool caps the driver’s throughput at whatever the pool can churn.

The fix is always to use `bus_dmamem_alloc` for the driver’s static buffers and to ensure the tag’s `lowaddr` is set correctly so the allocator places the buffers below 4 GB. For the dynamic case (mbufs, user buffers), the network stack provides pre-allocated mbuf clusters below 4 GB for drivers that request them via `bus_dma`-aware mbuf types, though this path is inside `iflib` and not a concern for Chapter 21.

21.13 A Deeper Look at IOMMU Integration

Section 1 mentioned the IOMMU. This deeper look covers what the IOMMU does, how to tell if it is active, and why the `bus_dma` API is designed to be transparent to it.

21.13.1 What the IOMMU Does

An IOMMU (Input-Output Memory Management Unit) is a translation layer between a device’s bus addresses and the host’s physical addresses. With the IOMMU active:

- The device sees bus address X.
- The IOMMU translates X to physical address Y.
- The memory controller writes physical address Y.

The translation is per-device (each device has its own mapping table), and by default the IOMMU only allows a device to access memory the kernel has explicitly mapped for it. This provides two benefits:

1. **Security.** A compromised device cannot DMA to arbitrary memory; it can only reach memory the kernel has explicitly mapped.
2. **Flexibility.** The kernel can present a 32-bit-capable device with a 32-bit view of any region of memory, including regions above 4 GB, by setting up an appropriate mapping. The bounce-buffer strategy becomes unnecessary when the IOMMU can just remap.

21.13.2 How the Kernel Integrates

FreeBSD's `busdma_iommu` (built when `DEV_IOMMU` is set, typically on x86 with VT-d or AMD-Vi) integrates with the `bus_dma` API transparently. When a driver calls `bus_dmamap_load`, the `busdma` layer asks the IOMMU to allocate IOVA (I/O virtual address) space and program the mapping; the `bus_addr_t` returned to the driver is the IOVA, not the physical address. The driver programs the device with the IOVA; the device issues a DMA to the IOVA; the IOMMU translates to the physical address; the memory controller completes the transfer.

From the driver's perspective, nothing changes. The same `bus_dma` calls do the right thing on IOMMU systems as on non-IOMMU systems. This is the portability promise of `bus_dma`: drivers written against the API work on whatever the platform provides.

21.13.3 Detecting the IOMMU

```
sysctl hw.vmm.ppt.devices  # shows pass-through devices (bhyve)
sysctl hw.dmar              # Intel VT-d if enabled
sysctl hw.iommu             # generic IOMMU flag
```

On a system where the kernel booted with the IOMMU active, `dmesg` shows a line like `dmar0: <DMAR>` near boot.

21.13.4 Performance Implications

IOMMU translation is not free. Each translation goes through the IOMMU's page tables; the IOMMU caches recently-used translations in a TLB-like structure, but misses cost real time. For drivers that make millions of small mappings per second, the IOMMU can become a bottleneck.

The mitigations are:

1. **Batch mappings.** A driver that keeps a long-lived buffer mapped once has no recurring IOMMU cost. Chapter 21's static pattern is ideal.
2. **Use the IOMMU's super-page support.** Larger pages mean fewer TLB entries.
3. **Disable the IOMMU for trusted devices.** Not generally advised, but possible for specific deployments.

Chapter 21's driver is not affected either way; the static buffer is mapped once at attach and never remapped. Drivers with high remap rates (storage controllers processing many small transfers per second) are affected more.

21.14 A Deeper Look at BUS_DMA_COHERENT vs Explicit Sync

Section 4 covered the BUS_DMA_COHERENT flag at the API level. This deeper look explains the interaction with the sync discipline, which is subtle on some platforms.

21.14.1 The Relationship

BUS_DMA_COHERENT is a hint passed to `bus_dma_tag_create` (architectural preference) and `bus_dmamem_alloc` (allocation mode). It asks the allocator to place the buffer in a memory domain that is coherent with the DMA path: on arm64 with write-combining or uncached regions, this is a specific memory type; on amd64, the allocator chooses normal memory because the PCIe root complex is already coherent.

The flag does not eliminate the requirement to call `bus_dmamap_sync`. The API contract requires the sync; BUS_DMA_COHERENT makes the sync cheap or free, not unnecessary.

21.14.2 Why the Sync Is Still Required

Two reasons. First, the driver is portable: the same source file compiles on every platform FreeBSD supports. On a platform where BUS_DMA_COHERENT is honoured but bounce buffers are in play (for an addressing constraint, say), the sync is the hook that triggers the bounce copy. Without it, the bounce data is stale.

Second, the driver is forward-compatible: a future platform may not honour the flag, or may honour it only for certain allocators. The explicit sync is what makes the driver correct against any future platform.

The rule: always pass BUS_DMA_COHERENT when the access pattern justifies it, and always call `bus_dmamap_sync` regardless. The flag makes sync cheap; the sync makes the driver correct.

21.14.3 When to Use the Flag

For static buffers (descriptor rings, control structures): always pass BUS_DMA_COHERENT. The buffer is accessed often by both the CPU and the device; coherent memory reduces the per-access cost.

For dynamic buffers (packet payloads, disk blocks): usually do not pass it. The cache-hit rate when the CPU processes the data is usually high enough that coherent memory's slower access patterns hurt more than they help.

The `if_re` driver follows exactly this: descriptor tags use BUS_DMA_COHERENT, payload tags do not. NVMe's per-qpair tag uses BUS_DMA_COHERENT for the same reason.

21.15 A Deeper Look at Parent and Child Tag Inheritance

Section 3's setup function passed `bus_get_dma_tag(sc->dev)` as the parent. This deeper look covers why inheritance matters and when a driver should create an explicit parent tag rather than inheriting from the bridge.

21.15.1 Inheritance Semantics

A child tag inherits every constraint from its parent that is more restrictive than the child's own. The intersection semantics mean:

- The child's `lowaddr` is `min(parent_lowaddr, child_lowaddr)`. The tighter limit wins.
- The child's alignment is `max(parent_alignment, child_alignment)`. More alignment wins.
- The child's `maxsize` is `min(parent_maxsize, child_maxsize)`. The smaller win.
- The child's `nsegments` is `min(parent_nsegments, child_nsegments)`.
- Flags are composed: `BUS_DMA_COHERENT` propagates, `BUS_DMA_ALLOCNOW` does not.

The kernel applies these rules at tag-create time; the child's internal constraints are the intersection of the parent's and its own.

21.15.2 Why Drivers Create Explicit Parent Tags

A driver that has several per-subsystem tags often finds it cleaner to create one explicit parent tag that carries the device-level constraints (addressing limits, platform-level alignment), then create children for each specific purpose (rings, payloads). The parent is then destroyed at detach after all children are destroyed.

`if_re` does this: `rl_parent_tag` is the device-level parent, inheriting from the PCI bridge and adding the device's 32-bit limit (for non-PCIe chips). The four child tags (`rl_tx_mtag`, `rl_rx_mtag`, `rl_tx_list_tag`, `rl_rx_list_tag`) inherit from `rl_parent_tag`. Destroying `rl_parent_tag` is the last DMA cleanup step, because all four children must be destroyed first (`bus_dma_tag_destroy` returns `EBUSY` if any child still exists).

21.15.3 When to Skip the Explicit Parent

For drivers with a single tag (like Chapter 21's), creating an explicit parent is overkill. The driver's single tag inherits from `bus_get_dma_tag(sc->dev)` directly and applies its own constraints; the bridge's tag is the de-facto parent.

For drivers with two or three related tags that share constraints, an explicit parent reduces duplication and makes the design clearer.

For drivers with many tags (descriptor rings, multiple buffer pools, shared memory regions), an explicit parent is always the right choice.

21.16 A Deeper Look at Descriptor Rings as a Future Topic

Chapter 21's driver uses a single DMA buffer. Production drivers use descriptor rings. This deeper look previews what a descriptor ring is, so the reader who wants to extend the chapter's work has a shape to aim at, but the detailed teaching is deferred.

A descriptor ring is an array of fixed-size entries in DMA-coherent memory. Each entry contains at least a bus address and a length, plus flags that describe the transfer (direction, type, status). The driver and the device

communicate through the ring: the driver writes entries, the device reads them, performs the transfer, and writes status back.

The ring has two indexes: a **producer** index (the next entry the writer will fill) and a **consumer** index (the next entry the reader will process). For a transmit ring, the driver is the producer and the device is the consumer. For a receive ring, the roles are reversed. Both indexes wrap modulo the ring size.

The driver signals new entries by writing a **doorbell** register (an MMIO write that the device treats as “look at the ring”). The device signals completions by raising an interrupt; the driver then walks the ring from the last consumer index to the current one, processes each entry, and advances the consumer index.

The complications that make rings their own topic are: head-of-line blocking when the ring is full, partial-transfer handling across ring entries, reorder tolerance (does the device complete entries in order, or out of order?), flow control between driver and device, wrap-around correctness, and the interaction with multi-queue hardware where each queue has its own ring.

Extending Chapter 21’s driver to use a small ring is a natural exercise. The challenges at the end of the chapter include a sketch. The full descriptor-ring topic is the network chapter in Part 6 (Chapter 28).

21.17 A Mental Model for Chapter 21’s Sync Discipline

A paragraph-length mental model that captures the sync discipline in one sentence: *the PRE and POST sync calls mark every moment a DMA buffer changes ownership between the CPU and the device, and the driver writes them into its code even when the current platform makes them free.* Chapter 21’s driver respects this model everywhere. Every `bus_dmamap_sync` call in `myfirst_dma.c` is paired with a partner: PRE marks the CPU releasing ownership, POST marks the CPU re-acquiring it. Between the two, the buffer belongs to the device, and no driver code reads or writes it. The discipline sounds absolute because it is; every production driver follows it; every exception the reader might imagine is actually a buggy driver that happens to work on the test platform.

21.18 Patterns From Real FreeBSD Drivers

A short catalogue of DMA patterns as they appear in the real tree. Each pattern has a representative file; reading the file’s DMA-related functions after this chapter is the recommended follow-up practice.

21.18.1 Pattern: `bus_dmamem_alloc` for Static Descriptor Rings

Where: `/usr/src/sys/dev/re/if_re.c`, `re_allocmem`; `/usr/src/sys/dev/nvme/nvme_qpair.c`, `nvme_qpair_construct`.

The driver allocates a contiguous region at attach time, loads it with a single-segment callback, uses it for the driver’s lifetime, unloads it on detach. This is Chapter 21’s static pattern and the foundation for almost every DMA-capable driver.

21.18.2 Pattern: `bus_dmamap_load_mbuf_sg` for Per-Packet Mappings

Where: `/usr/src/sys/dev/re/if_re.c`, `re_encap`; `/usr/src/sys/dev/e1000/if_em.c`, `em_xmit`.

For each outgoing packet, the driver loads the packet's mbuf into a dynamic map, programs the descriptor, transmits, and unloads. This is the dynamic pattern Chapter 21 mentions but does not implement.

21.18.3 Pattern: Parent-Child Tag Hierarchy

Where: `/usr/src/sys/dev/re/if_re.c`, `re_allocmem`; `/usr/src/sys/dev/bce/if_bce.c`, `bce_dma_alloc`.

The driver creates a parent tag for device-level constraints and child tags for per-subsystem allocations. The hierarchy composes constraints automatically and separates the driver's cleanup into clear steps.

21.18.4 Pattern: `BUS_DMASYNC_PREREAD` | `BUS_DMASYNC_PREWRITE` for Shared Rings

Where: `/usr/src/sys/dev/nvme/nvme_qpair.c`, around `nvme_qpair_reset` and the submission queue; `/usr/src/sys/dev/ahci/ahci.c`, around the command table.

For rings where both the driver and the device read and write, the combined PRE flag at setup and the combined POST flag at transaction boundaries make the sync discipline explicit.

21.18.5 Pattern: Single-Segment Callback as a Universal Helper

Where: `/usr/src/sys/dev/nvme/nvme_private.h`, `nvme_single_map`; `/usr/src/sys/dev/re/if_re.c`, `re_dma_map_addr`; many others.

Almost every driver defines a single-segment callback named in its own style. Chapter 21's `my_first_dma_single_map` is a clean example; the real-driver variants are all functionally identical.

21.18.6 Pattern: Coherent Tag for Rings, Non-Coherent for Payloads

Where: `/usr/src/sys/dev/re/if_re.c` and many others.

The driver passes `BUS_DMA_COHERENT` for descriptor-ring tags and omits it for payload tags. The allocator handles both; the driver makes the right choice per tag.

21.19 Hands-On Labs

Five hands-on labs walk the reader through the chapter's stages on a lab target. Each lab is self-contained; completing all five produces a working 1.4-dma driver.

21.19.1 Lab 1: Identify DMA-Capable Devices on Your System

Goal: build a map of your lab machine's DMA landscape before the chapter's coding work begins.

Time: 20 minutes.

Steps:

1. Run `pciconf -lv` on your lab host and note the devices.
2. Run `pciconf -lvc | grep -B1 BUSMASTEREN` to identify devices with bus-mastering enabled.
3. Pick one device, run `pciconf -lvbc <devname>`, and identify its BAR layout and capability list.
4. Run `sysctl hw.busdma` and note the bounce-buffer counters.
5. Write in your lab notebook: three devices, one property each, your current bounce-buffer usage.

Expected: a short list of devices with their bus-mastering status, and an understanding of whether your system is bouncing.

21.19.2 Lab 2: Stage 1, Allocate and Map a DMA Buffer

Goal: build the Stage 1 driver that creates a tag, allocates a buffer, loads a map, and cleans up.

Time: 1.5 hours.

Steps:

1. Start from the Chapter 20 Stage 4 driver (1.3-msi).
2. Add the four softc fields (`dma_tag`, `dma_map`, `dma_vaddr`, `dma_bus_addr`).
3. Add the single-segment callback (`myfirst_dma_single_map`).
4. Add `myfirst_dma_setup` and `myfirst_dma_tearardown` to `myfirst.c`.
5. Call them from `attach` and `detach`.
6. Build and load. Verify the `dmesg` banner shows the KVA and bus address.
7. Unload and reload 50 times; check that `vmstat -m` does not show leaks.

Expected: a driver that stands up and tears down a DMA region cleanly. `dmesg` shows the banner on each load; no panic on unload.

Hint: if the load fails immediately with `EINVAL`, check that `MYFIRST_DMA_BUFFER_SIZE` is both the tag's `maxsize` and the argument to `bus_dmamap_load`; a mismatch is the most common cause.

21.19.3 Lab 3: Stage 2, Simulated DMA Engine with Polling

Goal: extend the driver with the simulated DMA engine and the polling-based transfer helper.

Time: 2 hours.

Steps:

1. Add the `MYFIRST_REG_DMA_*` constants to `myfirst.h`.
2. Extend `struct myfirst_sim` with the DMA engine state.
3. Implement the engine's callout handler and the write hook.
4. Implement `myfirst_dma_do_transfer` (polling version).

5. Add the `dma_test_write` and `dma_test_read` sysctls.
6. Build and load.
7. Run `sudo sysctl dev.myfirst.0.dma_test_write=0xAA` and verify the `dmesg` output.
8. Run the 1000-iteration test from Section 5 and check the counters.

Expected: 1000 successful transfers in each direction; zero errors; zero timeouts.

Hint: if the transfer times out every time, the simulation's callout is probably not scheduling. Check `callout_reset` is called and that `hz / 100` matches your kernel's `hz` (use `sysctl kern.hz` to verify).

21.19.4 Lab 4: Stage 3, Interrupt-Driven Completion

Goal: replace the polling helper with the interrupt-driven helper.

Time: 2 hours.

Steps:

1. Extend the rx filter to handle `MYFIRST_INTR_COMPLETE`.
2. Extend the rx task to call `myfirst_dma_handle_complete`.
3. Add `myfirst_dma_do_transfer_intr` to `myfirst.c`.
4. Enable `MYFIRST_INTR_COMPLETE` in the `INTR_MASK` write.
5. Rewire the sysctl handlers to use the interrupt-driven version.
6. Build and load.
7. Run the 1000-iteration test. Verify `dma_complete_intrs` and `dma_complete_tasks` climb together.
8. Verify no CPU is pegged during the test (`top` or `vmstat` should show low system CPU).

Expected: 1000 interrupts, 1000 tasks, 1000 completions. No CPU pegged.

Hint: if the completion counter increments but the helper hangs in `cv_timedwait` until the 1-second timeout, the task is not calling `cv_broadcast` on `dma_cv`. The wake-up is what unblocks the helper.

21.19.5 Lab 5: Stage 4, Refactor and Regression

Goal: complete the refactor into `myfirst_dma.c` and `myfirst_dma.h`; run the final regression test.

Time: 1.5 hours.

Steps:

1. Create `myfirst_dma.h` with the public API.
2. Create `myfirst_dma.c` with the DMA code moved from `myfirst.c`.
3. Update the Makefile to include `myfirst_dma.c` and bump the version to 1.4-dma.
4. Create `DMA.md` with the subsystem reference.
5. Run `make clean && make` and confirm a clean build.
6. Load the driver, verify the version string.
7. Run the full regression script (`labs/full_regression_ch21.sh`) and confirm it prints PASS.

Expected: clean build, clean load, clean regression pass, clean unload.

Hint: if the link fails with an undefined symbol referring to one of the DMA functions, the SRCS line in the Makefile has not been updated to include `myfirst_dma.c`. Run `make clean` after changing the Makefile to ensure the new source is compiled.

21.20 Challenge Exercises

Challenge exercises that stretch Chapter 21's scope without introducing foundations from later chapters. Pick the ones that interest you; the list is meant to be a menu, not a checklist.

21.20.1 Challenge 1: Multi-Buffer Rotation

Extend the driver to maintain a pool of four DMA buffers instead of one. Each transfer picks the next buffer in rotation. The test program runs four transfers back-to-back without waiting for completions and observes that all four complete before the fifth is submitted.

What it exercises: the allocation and load paths in a loop; per-buffer softc state; the interaction with a single completion vector when multiple transfers are in flight.

21.20.2 Challenge 2: Scatter-Gather Load

Extend the driver to accept a scatter-gather list of up to three segments per transfer. Use `bus_dmamap_load` with a multi-segment callback; store the segment list in the softc; program the simulated engine with three (address, length) pairs.

What it exercises: the non-single-segment callback; the tag's `nsegments` parameter; the engine's ability to handle multiple segments.

21.20.3 Challenge 3: Descriptor Ring Sketch

Extend the driver with a small descriptor ring (eight entries). Each entry is a struct with a bus address, a length, a direction, and a status. The driver fills entries, writes the ring's head doorbell, and the simulated engine walks the ring and updates status.

What it exercises: the full producer-consumer ring pattern; the combined sync flag for bidirectional rings; the head/tail doorbell protocol.

21.20.4 Challenge 4: Partial Transfer Reporting

Modify the simulated engine to occasionally (say, every fifth transfer) report a partial transfer. Modify the driver to detect the partial result, log it, and retry the remainder.

What it exercises: the partial-transfer pattern from Section 7; retry logic; per-transfer state tracking.

21.20.5 Challenge 5: IOMMU Awareness

On a system with the IOMMU active, add sysctls that expose the difference between the driver's bus address and the underlying physical address. Use `pmap_kextract` to get the physical address of the buffer's KVA, and compare it against `sc->dma_bus_addr`.

What it exercises: understanding the IOMMU's transparent remapping; observability of the abstraction.

21.20.6 Challenge 6: Bounce Buffer Observability

Create a tag with `lowaddr = BUS_SPACE_MAXADDR_32BIT` to force 32-bit-only addressing. Allocate buffers above 4 GB (via `contigmalloc` with appropriate flags) and observe that `bus_dmamap_load` triggers bouncing. Expose counters for `total_bounced` from the driver's perspective.

What it exercises: understanding when bouncing happens; observability of the bounce path.

21.20.7 Challenge 7: Tag Template Refactor

Rewrite the Stage 4 setup function to use the `bus_dma_template_*` API instead of `bus_dma_tag_create`. The behaviour should be identical; only the syntax changes.

What it exercises: the modern template API; the equivalence between the two tag-creation styles.

21.20.8 Challenge 8: DTrace-Based Transfer Profiling

Write a DTrace script that measures the latency of DMA transfers from the helper's perspective. Attach to `myfirst_dma_do_transfer_intr` entry and exit; print a histogram of elapsed times; compare against the one-millisecond engine delay.

What it exercises: DTrace FBT probes; DMA timing observability; understanding of the cost breakdown.

21.21 Troubleshooting and Common Mistakes

A reference guide to the most common problems and their fixes. Each entry has the symptom, the likely cause, and the fix.

21.21.1 “bus_dma_tag_create fails with EINVAL”

Symptom: `device_printf` reports `bus_dma_tag_create` failed: 22.

Likely cause: the tag parameters are inconsistent. The alignment must be a power of two; the boundary must be a power of two and at least as large as `maxsegsz`; `maxsize` must be at least `alignment`.

Fix: check the parameters against the `bus_dma(9)` manual page's constraint descriptions. Common mistakes: alignment of 3 (not a power of two); boundary smaller than `maxsegsz`.

21.21.2 “bus_dmamap_load callback runs with error != 0”

Symptom: the single-segment callback’s error argument is non-zero; dma_bus_addr is zero.

Likely cause: the buffer is larger than the tag’s maxsize, or the mapping cannot satisfy the tag’s segment constraints (EFBIG).

Fix: check that the load’s buflen matches the tag’s maxsize; check that the buffer is contiguous if the tag allows only one segment.

21.21.3 “dma_bus_addr is zero after bus_dmamap_load returned zero”

Symptom: the load returns success but the bus address is not populated.

Likely cause: the callback was not called (unlikely, but possible if the kernel’s internal state is inconsistent), or the callback did not write *addr because of an internal error it swallowed.

Fix: check the callback for early returns that do not populate the output. Chapter 21’s pattern is to zero dma_bus_addr before the load and check for zero after.

21.21.4 “Transfers succeed but the buffer contents are wrong”

Symptom: the transfer completes successfully according to the counters, but the data in the buffer is not what the sender sent.

Likely cause: a missing or wrong-flavour bus_dmamap_sync call. On a coherent platform the bug may be invisible; on a non-coherent platform or with bounce buffers active it is visible.

Fix: verify every transfer has exactly one PRE sync and one POST sync matching the transfer direction. PREWRITE/POSTWRITE for host-to-device; PREREAD/POSTREAD for device-to-host.

21.21.5 “bus_dmamap_unload panics with ‘map not loaded’ ”

Symptom: kldunload panics (or prints a WITNESS warning) at bus_dmamap_unload.

Likely cause: the driver’s teardown runs bus_dmamap_unload twice, or runs it when the map was never loaded.

Fix: guard the call with if (sc->dma_bus_addr != 0) and zero the field after unload. Chapter 21’s teardown pattern is correct; verify the driver’s teardown matches.

21.21.6 “bus_dmamem_free panics with ‘no allocation’ ”

Symptom: kldunload panics at bus_dmamem_free.

Likely cause: dma_vaddr is stale (was freed but not zeroed), and the teardown is running again on a second unload path.

Fix: set dma_vaddr = NULL immediately after bus_dmamem_free. Chapter 21’s teardown pattern is correct; verify.

21.21.7 “dma_complete_intrs is zero even though transfers succeed”

Symptom: the transfer helper returns success (via the polling path), but the completion-interrupt counter stays at zero.

Likely cause: MYFIRST_INTR_COMPLETE is not enabled in the INTR_MASK write, or the filter is not checking the bit.

Fix: verify the mask write enables MYFIRST_INTR_COMPLETE; verify the filter has the second if for the complete bit. Stage 3’s changes include both.

21.21.8 “cv_timedwait always returns EWOULDBLOCK”

Symptom: every interrupt-driven transfer times out after one second.

Likely cause: the task never calls cv_broadcast(&sc->dma_cv), or the task is not running because the filter did not enqueue it.

Fix: verify the task calls cv_broadcast(&sc->dma_cv) in myfirst_dma_handle_complete. Verify the filter calls taskqueue_enqueue after detecting the complete bit.

21.21.9 “Driver hangs on kldunload”

Symptom: kldunload myfirst blocks indefinitely.

Likely cause: a transfer is in flight and the helper is waiting on dma_cv; the detach path is waiting for dma_in_flight to clear; but the device is not completing.

Fix: the detach path must call callout_drain before waiting for dma_in_flight. If the callout has not been drained, the simulation can complete the transfer and the detach proceeds. Verify the order: set detaching = true, drain callout, wait for in-flight to clear, tear down.

21.21.10 “WITNESS warns about lock order reversal”

Symptom: dmesg shows a WITNESS warning involving sc->mtx, dma_cv, or the taskqueue lock.

Likely cause: the helper holds sc->mtx while calling cv_timedwait, which is correct; but an assertion may be triggered if a different path also holds a conflicting lock.

Fix: review the lock ordering. The Chapter 11/19/20 discipline: sc->mtx before dma_cv; no taskqueue locks held by the driver; no dma_cv wait while holding an atomic lock. Chapter 21’s patterns are correct; the warning likely indicates a driver-local issue.

21.21.11 “Counters climb but the buffer is not populated”

Symptom: dma_transfers_read counter increases, but sc->dma_vaddr is all zeros after the transfer.

Likely cause: the simulation’s dma_scratch pattern is not being copied into the host buffer, or the POST sync is skipped.

Fix: verify the simulated engine's `myfirst_sim_dma_copy_to_host` is called with the correct length; verify the POST sync runs. If the simulation uses a simplified mapping (bus address == KVA), verify that the host's KVA actually corresponds to the tag's allocated buffer.

21.21.12 “Load-unload cycle leaks memory”

Symptom: after 50 load-unload cycles, `vmstat -m` shows `bus_dmamap` or `bus_dma_tag` with non-zero counts.

Likely cause: a failure path in setup did not destroy the tag or free the memory before returning.

Fix: review every failure return in `myfirst_dma_setup`. Chapter 21's pattern destroys the tag on any post-tag-create failure, frees memory on any post-alloc failure, unloads the map on any post-load failure. Verify each failure path unwinds correctly.

21.21.13 “Transfers are slow compared to expected”

Symptom: the 1000-transfer test takes many seconds instead of milliseconds.

Likely cause: the polling loop's pause is too coarse, or the interrupt path is mis-configured so every transfer hits the 1-second timeout.

Fix: if using the polling helper, reduce the pause interval; if using the interrupt helper, verify the complete bit is reaching the filter (Stage 3's `dma_complete_intrs` should match the transfer count exactly).

21.22 Worked Example: Tracing a DMA Transfer End to End

A detailed walk through a single host-to-device DMA transfer in Chapter 21's Stage 4 driver, annotated at every step with which line of code runs, what the kernel is doing underneath, what WITNESS would check, and what an operator would see in `dmesg` or a counter. The goal is to give the reader a mental picture of the full pipeline as a sequence of concrete events.

21.22.1 The Starting State

The driver has attached. `dma_tag`, `dma_map`, `dma_vaddr`, and `dma_bus_addr` are populated. The interrupt mask has `MYFIRST_INTR_COMPLETE` set. The rx vector's task is initialised. All counters are at zero. A user logs in and types:

```
sudo sysctl dev.myfirst.0.dma_test_write=0xAA
```

21.22.2 Event 1: The sysctl Handler Runs

The sysctl framework calls `myfirst_dma_sysctl_test_write`. The handler parses the value (0xAA), fills `sc->dma_vaddr` with 0xAA bytes, and calls `myfirst_dma_do_transfer_intr(sc, MYFIRST_DMA_DIR_WRITE, MYFIRST_DMA_BUFFER_SIZE)`.

Context: process context, the user's `sysctl` binary's kernel side. No locks held yet. WITNESS has nothing to check.

21.22.3 Event 2: The Helper Takes the Softc Lock

`myfirst_dma_do_transfer_intr` acquires `sc->mtx` via `MYFIRST_LOCK(sc)`. WITNESS checks the lock order: the softc mutex is an `MTX_DEF` lock, no higher-ordered locks are held, the acquisition is valid.

The helper checks `sc->dma_in_flight`. It is false (no transfer is running). The helper proceeds.

21.22.4 Event 3: The PRE Sync

`bus_dmamap_sync(sc->dma_tag, sc->dma_map, BUS_DMASYNC_PREWRITE)` runs. On amd64, this is a memory barrier (`mfence`). On arm64 with `BUS_DMA_COHERENT` honoured, it is a data-memory barrier. On older arm without coherent DMA, it would be a cache flush of the buffer's lines.

The kernel's internal state: the tag's map tracker marks the map as pending PRE. If bounce pages were in use (they are not on Chapter 21's amd64 test), the bounce data would be copied now.

21.22.5 Event 4: The Device Programming

The helper writes four registers in sequence: `DMA_ADDR_LOW`, `DMA_ADDR_HIGH`, `DMA_LEN`, `DMA_DIR`. Each write goes through `bus_write_4`, which dispatches through the Chapter 16 accessor to the simulation's write hook. The simulation records the values in `sim->dma`.

The helper writes `DMA_CTRL = MYFIRST_DMA_CTRL_START`. The write triggers `myfirst_sim_dma_ctrl_written` in the simulation.

21.22.6 Event 5: The Simulation Arms the Callout

Inside the simulation, `myfirst_sim_dma_ctrl_written` sees `MYFIRST_DMA_CTRL_START`, checks that no transfer is already running, sets `status = RUNNING`, sets `armed = true`, and calls `callout_reset(&sim->dma.done_co, hz / 100, myfirst_sim_dma_done_co, sim)`.

The callout is scheduled ten milliseconds in the future. The simulation returns; the helper's `CSR_WRITE_4` returns.

21.22.7 Event 6: The Helper Waits

The helper sets `sc->dma_last_direction = MYFIRST_DMA_DIR_WRITE`, sets `sc->dma_in_flight = true`, and calls `cv_timedwait(&sc->dma_cv, &sc->mtx, hz)`.

`cv_timedwait` releases `sc->mtx`, puts the thread to sleep on `dma_cv`, and schedules a wakeup at `hz` (1 second) in the future. The thread is now off the run queue.

Context: the kernel scheduler is free to run other work. WITNESS tracks that the thread went to sleep while holding no non-sleepable locks; this is correct because the mutex was released.

21.22.8 Event 7: The Callout Fires

Ten milliseconds later, the callout subsystem runs `myfirst_sim_dma_done_co` on a softclock thread. The function extracts the bus address (`((sim->dma.addr_high << 32) | sim->dma.addr_low)`), confirms `len <= scratch_size`, and calls `myfirst_sim_dma_copy_from_host` to copy the 0xAA pattern from the host buffer into the scratch buffer.

The function sets `sim->dma.status = DONE` and calls `myfirst_sim_dma_raise_complete`.

Context: softclock thread. No driver lock held. WITNESS checks that the callout's callback is not acquiring driver locks in a way that conflicts with the `callout_mtx`; the sim's data is protected by per-sim state or atomic updates.

21.22.9 Event 8: The Simulation Raises the Completion Bit

`myfirst_sim_dma_raise_complete` sets `MYFIRST_INTR_COMPLETE` in `sim->intr_status` and, since the bit is enabled in `intr_mask`, calls `myfirst_sim_raise_intr`. The latter queues the simulated interrupt event through the Chapter 19/20 path.

21.22.10 Event 9: The Filter Runs

The kernel's interrupt machinery dispatches the rx vector. The filter (`myfirst_msix_rx_filter`) runs in primary interrupt context. It reads `INTR_STATUS`, sees `MYFIRST_INTR_COMPLETE` set, acknowledges the bit, increments `sc->dma_complete_intrs` via `atomic_add_64`, and enqueues the rx task.

The filter returns `FILTER_HANDLED`. The kernel's interrupt dispatch ends.

Context: primary interrupt context. No sleep, no malloc, no lock above the spinlock level. WITNESS checks that the atomic increment does not imply any lock.

21.22.11 Event 10: The Task Runs

The taskqueue machinery schedules the rx task on its thread. The thread acquires `sc->mtx` via `MYFIRST_LOCK(sc)`. It reads `DATA_OUT` (from the Chapter 19/20 pipeline) and broadcasts `data_cv`. Then it calls `myfirst_dma_handle_complete(sc)`.

21.22.12 Event 11: The POST Sync

`myfirst_dma_handle_complete` sees `sc->dma_in_flight == true`. It clears the flag, checks the direction (`DIR_WRITE`), issues `bus_dmamap_sync(..., BUS_DMASYNC_POSTWRITE)`, reads `DMA_STATUS` (which shows `DONE`), increments `dma_complete_tasks`, and calls `cv_broadcast(&sc->dma_cv)`.

The POST sync on amd64 is a barrier. On arm64 with coherent memory, a barrier. On systems with bounce buffers, the bounce data would be copied back now (for a POSTREAD; POSTWRITE usually just releases the bounce page).

21.22.13 Event 12: The Task Releases the Lock

The task's body completes. It releases `sc->mtx` via `MYFIRST_UNLOCK(sc)`. The task returns.

21.22.14 Event 13: The Helper Wakes

`cv_broadcast(&sc->dma_cv)` in Event 11 scheduled the helper's thread back to the run queue. The helper wakes inside `cv_timedwait`, re-acquires `sc->mtx`, and returns from the wait with `err = 0` (not `EWOULD-BLOCK`).

21.22.15 Event 14: The Helper Examines the Status

The helper reads `sc->dma_last_status`. It is `DONE` with `ERR = 0`. The helper increments `dma_transfers_write` via `atomic_add_64`. It releases `sc->mtx` and returns 0.

21.22.16 Event 15: The Sysctl Handler Returns

`myfirst_dma_sysctl_test_write` sees the successful return, prints a banner (`dma_test_write: pattern 0xaa transferred`), and returns 0 to the sysctl framework.

21.22.17 Event 16: The User Sees the Result

The user's sysctl command returns. A subsequent `dmesg | tail` shows the banner. The counters now show:

```
dma_transfers_write: 1
dma_complete_intrs: 1
dma_complete_tasks: 1
```

21.22.18 The Total Time

The critical path's wall-clock time is roughly: 10 ms for the callout delay, plus a few microseconds for the filter dispatch, plus a few microseconds for the task, plus the context-switch latency for the helper's wakeup. Total: about 10-11 ms.

If the helper had used the Stage 2 polling path instead, the wall-clock would be similar (the callout delay dominates), but the helper would hold a CPU busy polling for the entire ten milliseconds. The interrupt-driven version frees the CPU to do other work.

21.22.19 The Lesson

Sixteen events across five contexts: process (helper), softclock (callout), primary interrupt (filter), taskqueue thread (task), process (helper wake). Each context has its own lock discipline; the transitions between contexts

happen via the kernel's scheduling and interrupt machinery, not through driver code. The driver's code lives inside each context; the kernel moves the execution between them.

A reader who can trace this sequence from beginning to end understands Chapter 21. A reader who cannot can re-read Sections 5 and 6 with the sequence in mind; the individual steps become concrete when mapped to the narrative.

21.23 A Deeper Look at Locking and DMA

Chapter 11 and Chapter 19 established the driver's lock discipline. Chapter 21 applies it to the DMA path, which introduces a few specifics worth calling out.

21.23.1 The Softc Mutex Covers the DMA State

The softc mutex (`sc->mtx`) protects every DMA field that is read or written outside of attach/detach: `dma_in_flight`, `dma_last_direction`, `dma_last_status`, the per-transfer counters. Every code path that touches these fields acquires `sc->mtx` first.

The counters are updated with `atomic_add_64` so the filter (which cannot acquire a sleep mutex) can increment them without taking the lock. The same counters are read with `atomic_load_64` (implicitly via the `sysctl` handler's copy) when the reader wants a consistent snapshot. Using atomics instead of the mutex for counters is the Chapter 11 pattern; it is faster and filter-safe.

The condition variable `dma_cv` is waited on and broadcast under `sc->mtx`. This is standard `cv_*` discipline.

21.23.2 The Tag's Lockfunc

The `lockfunc` parameter of `bus_dma_tag_create` is for deferred loads. Chapter 21 passes `NULL`, which substitutes `_busdma_dflt_lock`, which panics if called. The panic is the safety net: it tells the developer that a deferred load happened (which Chapter 21's `BUS_DMA_NOWAIT` usage should prevent), at which point the developer would add a proper `lockfunc`.

For drivers that do use deferred loads, the common pattern is:

```
err = bus_dma_tag_create(parent, align, bdry, lowaddr, highaddr,
    NULL, NULL, maxsize, nseg, maxsegsz, 0,
    busdma_lock_mutex, &sc->mtx, &sc->tag);
```

`busdma_lock_mutex` is a kernel-provided implementation that acquires and releases the mutex passed as the last argument. The kernel calls it before and after the deferred callback.

21.23.3 No Locks in the Sync Path

`bus_dmamap_sync` does not internally acquire any driver lock. It is safe to call from filter, task, or process context, as long as the caller has the appropriate concurrency discipline for the map. Chapter 21 calls `sync` from the helper (with `sc->mtx` held) and from the task (with `sc->mtx` held, inside `myfirst_dma_handle_complete`). Both are safe because no thread outside the lock scope can concurrently sync the same map.

21.23.4 The Detach Race

The detach path's race is between “transfer is completing” and “driver is being unloaded”. Chapter 21's pattern uses the detaching flag plus the `dma_in_flight` wait to ensure the detach cannot tear down resources while a transfer is in progress. The `callout_drain` and `taskqueue_drain` ensure no callback is pending.

This pattern is the Chapter 11 detach discipline generalised to one more kind of in-flight work (DMA transfers). The pattern composes: interrupts were drained in Chapter 19, per-vector tasks in Chapter 20, DMA transfers in Chapter 21. Chapter 22 will add one more (power-transition callbacks).

21.24 A Deeper Look at Observability for DMA Drivers

Chapter 21's driver exposes several observability hooks. This deeper look covers what each one provides and how an operator would use them.

21.24.1 The Per-Transfer Counters

```
dev.myfirst.N.dma_transfers_write  
dev.myfirst.N.dma_transfers_read  
dev.myfirst.N.dma_errors  
dev.myfirst.N.dma_timeouts  
dev.myfirst.N.dma_complete_intrs  
dev.myfirst.N.dma_complete_tasks
```

Six counters, each a 64-bit atomic. The invariants:

- `dma_complete_intrs == dma_complete_tasks`: every filter-observed completion produced a task invocation.
- `dma_complete_intrs == dma_transfers_write + dma_transfers_read + dma_errors + dma_timeouts`: every completion interrupt eventually produced a transfer outcome.

Violations of either invariant indicate a bug. The regression test checks both.

21.24.2 The Ring Buffer of Recent Transfers

A challenge exercise would add a small circular buffer of recent transfers, each recording the direction, length, result, and time. A read-only `sysctl` exposes the last `N` transfers as a kernel-formatted string. This is useful for post-mortem debugging (when the driver has hung, the ring buffer shows what was happening just before).

21.24.3 DTrace Probes

The driver has no explicit DTrace probes, but FBT (function boundary tracing) works automatically. Useful one-liners:


```
# Count DMA transfer calls
dtrace -n 'fbt::myfirst_dma_do_transfer_intr:entry { @[probecount] = count(); }'

# Histogram of transfer durations
dtrace -n 'fbt::myfirst_dma_do_transfer_intr:entry { self->start = vtimestamp; }
↳ fbt::myfirst_dma_do_transfer_intr:return /self->start/ { @[probecount] =
↳ quantize(vtimestamp - self->start); self->start = 0; }'

# Interrupt rate per vector
dtrace -n 'fbt::myfirst_msix_rx_filter:entry { @ = count(); } tick-1s { printa(@);
↳ clear(@); }'
```

DTrace is the best tool for understanding driver behaviour under load. The probes run with low overhead (tens of nanoseconds per hit on amd64) and can be enabled on a production system.

21.24.4 vmstat -i and systat -vmstat

`vmstat -i` shows the interrupt count per vector. For Chapter 21's driver with MSI-X, three vectors appear with their `bus_describe_intr` labels (admin, rx, tx). The rx line's count should equal `dma_complete_intrs` plus `data_av_count` (the combined activity on that vector).

`systat -vmstat` provides a live view. Running it in a second terminal while the test runs shows the vector's rate changing in real time.

21.24.5 procstat -kke

`procstat -kke` shows the kernel stack of every thread. The rx task's thread, when running the DMA handler, has a stack containing `myfirst_dma_handle_complete` and `bus_dmamap_sync`. Observing the stack confirms the thread is in the expected place; an unexpected stack suggests a hang or deadlock.

21.24.6 sysctl hw.busdma

The `hw.busdma` `sysctl` tree exposes DMA-subsystem-wide counters:

```
hw.busdma.total_bpages
hw.busdma.free_bpages
hw.busdma.reserved_bpages
hw.busdma.active_bpages
hw.busdma.total_bounced
hw.busdma.total_deferred
```

`total_bounced` is the most useful for Chapter 21 purposes: on a system where no driver should be bouncing, this stays at zero. A climbing value indicates some driver's tag constraints are forcing bouncing; this is usually a configuration issue.

21.25 A Deeper Look at Performance of DMA Transfers

Chapter 21's simulation runs every transfer in about ten milliseconds (the callout delay). Real hardware is much faster. This deeper look covers what the measurements tell us and what they do not.

21.25.1 The Latency Budget

A modern PCIe 3.0 device's DMA latency is dominated by:

1. **The PCIe posted-write latency.** The device writes to the CPU's memory controller via a posted write, which takes a few hundred nanoseconds round-trip.
2. **The cache flush/invalidate overhead.** On coherent systems this is tens of nanoseconds via snooping; on non-coherent systems it is microseconds.
3. **The interrupt delivery latency.** MSI-X delivery to a CPU takes microseconds including the ithread wakeup.
4. **The task dispatch latency.** The rx task's wakeup and dispatch takes microseconds.

Total: a DMA transfer of a 4 KB buffer on a modern amd64 system takes about 5-10 microseconds from kick to completion, for a sustained throughput of several gigabytes per second.

21.25.2 The Throughput Ceiling

With a 4 KB buffer and 10 microseconds per transfer, the throughput is 400 MB/s. To reach the link's 3.5 GB/s, the driver has to batch: multiple transfers in flight simultaneously. A descriptor ring with N entries can have N transfers in flight, and if each takes 10 microseconds but the driver issues one per microsecond, the aggregate throughput is 4 GB/s.

Chapter 21's single-buffer driver cannot batch; it issues one transfer at a time. The descriptor-ring extension (challenge exercise) can. Part 6's network driver (Chapter 28) uses this pattern in full.

21.25.3 Measuring the Chapter's Driver

A simple timing test:

```
struct timespec start, end;
nanouptime(&start);
for (int i = 0; i < 1000; i++) {
    myfirst_dma_do_transfer_intr(sc, DIR_WRITE, 4096);
}
nanouptime(&end);
elapsed = timespec_sub(end, start);
```

With the simulation's ten-millisecond callout, 1000 transfers should take about 10 seconds total, for a throughput of 400 KB/s. Small, because the simulation is designed to be observably asynchronous rather than fast. Real hardware would be orders of magnitude faster.

21.25.4 What the Measurement Tells Us

The measurement confirms the pipeline is end-to-end functional. The absolute number is not interesting (it is a simulation, not real hardware). What matters is that 1000 transfers complete without error, without timeout, without the helper hanging, and without the counters diverging from their invariants. That is the regression criterion for Chapter 21.

On a real DMA-capable device, replacing the simulation backend with the real hardware driver would swap the 10 ms callout delay for the hardware's actual latency. The Chapter 21 code is otherwise unchanged; this is the portability payoff of building against `bus_dma`.

21.26 Additional Troubleshooting: Ten More Failure Patterns

Ten more common problems that surface in Chapter 21-style DMA drivers, each with diagnosis and fix.

21.26.1 “Kernel panics on first transfer with ‘null pointer dereference’ ”

The likely cause is that `sc->dma_tag` or `sc->dma_map` is NULL because the setup failed silently. The Stage 1 setup's error paths must propagate failure up so `attach` fails cleanly; if they do not, `attach` appears to succeed but the transfer path hits the NULL pointer.

The fix: review every return path in `myfirst_dma_setup`. Each post-allocation failure must free the preceding allocations and set the `softc` fields to NULL.

21.26.2 “First transfer works, subsequent transfers hang”

Likely cause: `dma_in_flight` is not being cleared on the first transfer's completion. The task's `dma_handle_complete` must clear it; the helper's timeout path must also clear it.

The fix: verify `dma_in_flight = false` is set in both `dma_handle_complete` and the helper's `EWOLDBLOCK` branch.

21.26.3 “Transfers succeed but `dma_complete_intrs` is zero”

Likely cause: the polling helper is in use, not the interrupt helper. The polling path does not increment `dma_complete_intrs`.

The fix: verify the `sysctl` handler calls `myfirst_dma_do_transfer_intr`, not the Stage 2 polling version. The Stage 3 refactor should have replaced the callers.

21.26.4 “Buffer contents are correct but verification fails”

Likely cause: a stride mismatch between what the simulator wrote and what the driver expected. The driver expects 0x5A bytes; the simulator writes 0x5A; but the driver checks a different offset.

The fix: verify the sim's `memset` matches the driver's verification. This is almost always a test-code bug rather than a DMA bug.

21.26.5 “dma_errors counter climbs but the driver does not propagate the error”

Likely cause: the helper increments the counter but returns 0 instead of EIO.

The fix: verify the error branch returns EIO and the sysctl handler propagates the error to the caller.

21.26.6 “taskqueue_drain hangs on kldunload”

Likely cause: a task is pending, but something keeps re-enqueueing it. Most commonly, the filter is still running because the interrupt mask was not set to zero.

The fix: mask the interrupt before draining the taskqueue. The detach order matters: mask, drain callout, drain task, unload map.

21.26.7 “bus_dmamem_alloc returns a buffer above 4 GB despite lowaddr = MAXADDR_32BIT”

Likely cause: the tag’s highaddr is BUS_SPACE_MAXADDR_32BIT instead of BUS_SPACE_MAXADDR. The meaning of lowaddr is “the lowest address in the excluded window”; highaddr is “the highest address in the excluded window”. To exclude addresses above 4 GB, set lowaddr = BUS_SPACE_MAXADDR_32BIT and highaddr = BUS_SPACE_MAXADDR.

The fix: swap the addresses. The idiom is: lowaddr is “the last address the device can reach”; highaddr is usually BUS_SPACE_MAXADDR.

21.26.8 “Test script runs fine once, fails after module reload”

Likely cause: the module’s module-level state is not being reset on reload. A static variable in myfirst_sim.c that holds the sim state across loads would cause this.

The fix: verify every module-level variable is initialised at module_init time and cleared at module_fini. Chapter 17’s simulation backend should be doing this correctly.

21.26.9 “WITNESS warns about ‘DMA map used while not loaded’ or similar”

Likely cause: the sync call runs after the map has been unloaded, or before it has been loaded. The order of operations must be: load, sync, unload.

The fix: review the teardown path; the unload must come after the last sync and before the tag destroy.

21.26.10 “Transfers work on my amd64 test VM but fail on the arm64 build server”

Likely cause: platform-dependent coherence behaviour. The driver may be skipping a sync call that is a no-op on amd64 but required on arm64.

The fix: verify every transfer has both PRE and POST syncs; re-run the test with INVARIANTS on arm64 and inspect any panic.

21.27 Reference: The Complete Stage 4 `myfirst_dma.c` Walkthrough

A section-by-section tour of the refactored `myfirst_dma.c` file. The file is about 250 lines; this walkthrough explains each function's shape.

21.27.1 The Includes and Macros

```
#include <sys/param.h>
#include <sys/systm.h>
#include <sys/kernel.h>
#include <sys/bus.h>
#include <sys/lock.h>
#include <sys/mutex.h>
#include <sys/condvar.h>
#include <sys/sysctl.h>

#include <machine/bus.h>
#include <machine/resource.h>

#include "myfirst.h"
#include "myfirst_hw.h"
#include "myfirst_dma.h"
```

Standard set. `<sys/bus_dma.h>` is included transitively through `<machine/bus.h>`; no need to include it explicitly.

21.27.2 The Single-Segment Callback

Covered in Section 3. Ten lines.

21.27.3 The Setup Function

Covered in Section 3. About 50 lines with extensive error handling.

21.27.4 The Teardown Function

Covered in Section 3. About 15 lines.

21.27.5 The Polling Transfer Helper

Covered in Section 5. About 60 lines. Kept in Stage 4 as a fallback or a debug path; the `sysctl` handlers can be pointed at it for comparative testing.

21.27.6 The Interrupt-Driven Transfer Helper

Covered in Section 6. About 70 lines.

21.27.7 The Completion Handler

Covered in Section 8. About 20 lines.

21.27.8 The Sysctl Registration

```
void
myfirst_dma_add_sysctls(struct myfirst_softc *sc)
{
    struct sysctl_ctx_list *ctx = &sc->sysctl_ctx;
    struct sysctl_oid_list *kids = SYSCTL_CHILDREN(sc->sysctl_tree);

    SYSCTL_ADD_U64(ctx, kids, OID_AUTO, "dma_transfers_write",
        CTLFLAG_RD, &sc->dma_transfers_write, 0,
        "Successful host-to-device DMA transfers");
    SYSCTL_ADD_U64(ctx, kids, OID_AUTO, "dma_transfers_read",
        CTLFLAG_RD, &sc->dma_transfers_read, 0,
        "Successful device-to-host DMA transfers");
    SYSCTL_ADD_U64(ctx, kids, OID_AUTO, "dma_errors",
        CTLFLAG_RD, &sc->dma_errors, 0,
        "DMA transfers that returned EIO");
    SYSCTL_ADD_U64(ctx, kids, OID_AUTO, "dma_timeouts",
        CTLFLAG_RD, &sc->dma_timeouts, 0,
        "DMA transfers that timed out");
    SYSCTL_ADD_U64(ctx, kids, OID_AUTO, "dma_complete_intrs",
        CTLFLAG_RD, &sc->dma_complete_intrs, 0,
        "DMA completion interrupts observed");
    SYSCTL_ADD_U64(ctx, kids, OID_AUTO, "dma_complete_tasks",
        CTLFLAG_RD, &sc->dma_complete_tasks, 0,
        "DMA completion task invocations");
    SYSCTL_ADD_UQUAD(ctx, kids, OID_AUTO, "dma_bus_addr",
        CTLFLAG_RD, &sc->dma_bus_addr,
        "Bus address of the DMA buffer");

    SYSCTL_ADD_PROC(ctx, kids, OID_AUTO, "dma_test_write",
        CTLTYPE_UINT | CTLFLAG_WR | CTLFLAG_MPSAFE, sc, 0,
        myfirst_dma_sysctl_test_write, "IU",
        "Trigger a host-to-device DMA transfer");
    SYSCTL_ADD_PROC(ctx, kids, OID_AUTO, "dma_test_read",
```

```

        CTLTYPE_UINT | CTLFLAG_WR | CTLFLAG_MPSAFE, sc, 0,
        myfirst_dma_sysctl_test_read, "IU",
        "Trigger a device-to-host DMA transfer");
}

```

Seven read-only counters, one UQUAD bus address (for operator visibility), two write-only test sysctls.

21.27.9 The Sysctl Handlers

```

static int
myfirst_dma_sysctl_test_write(SYSCTL_HANDLER_ARGS)
{
    struct myfirst_softc *sc = arg1;
    unsigned int pattern;
    int err;

    err = sysctl_handle_int(oidp, &pattern, 0, req);
    if (err != 0 || req->newptr == NULL)
        return (err);

    memset(sc->dma_vaddr, (int)(pattern & 0xFF),
           MYFIRST_DMA_BUFFER_SIZE);
    err = myfirst_dma_do_transfer_intr(sc, MYFIRST_DMA_DIR_WRITE,
                                       MYFIRST_DMA_BUFFER_SIZE);
    if (err != 0)
        device_printf(sc->dev,
                      "dma_test_write: err %d\n", err);
    else
        device_printf(sc->dev,
                      "dma_test_write: pattern 0x%02x transferred\n",
                      pattern & 0xFF);
    return (err);
}

static int
myfirst_dma_sysctl_test_read(SYSCTL_HANDLER_ARGS)
{
    struct myfirst_softc *sc = arg1;
    unsigned int ignore;
    int err;
    uint8_t *bytes;

    err = sysctl_handle_int(oidp, &ignore, 0, req);

```

```

if (err != 0 || req->newptr == NULL)
    return (err);

err = myfirst_dma_do_transfer_intr(sc, MYFIRST_DMA_DIR_READ,
    MYFIRST_DMA_BUFFER_SIZE);
if (err != 0) {
    device_printf(sc->dev, "dma_test_read: err %d\n", err);
    return (err);
}

bytes = (uint8_t *)sc->dma_vaddr;
device_printf(sc->dev,
    "dma_test_read: first bytes %02x %02x %02x %02x "
    "%02x %02x %02x %02x\n",
    bytes[0], bytes[1], bytes[2], bytes[3],
    bytes[4], bytes[5], bytes[6], bytes[7]);
return (0);
}

```

The write handler fills the buffer and triggers host-to-device. The read handler triggers device-to-host and logs the first eight bytes. Both are small; both make the DMA subsystem user-accessible without requiring a custom ioctl.

21.27.10 Lines of Code

The full `myfirst_dma.c` file is about 280 lines. For comparison, Chapter 20's `myfirst_msix.c` was 330 lines; Chapter 19's `myfirst_intr.c` was 250 lines. Chapter 21's file is in the middle of that range, which matches the complexity: DMA setup is more intricate than single-vector interrupt setup, but less than multi-vector routing.

Taken together with the 330-line `myfirst_msix.c` and the 250-line `myfirst_intr.c`, the three Chapter 19-21 files account for about 860 lines. A reader who has written and understood them has written three production-grade driver subsystems in miniature.

21.28 Reference: FreeBSD Manual Pages for Chapter 21

The manual pages Chapter 21 has drawn on directly. Each is worth a read after the chapter's material has settled.

- `bus_dma(9)`: the central reference for the bus-DMA API.
- `bus_space(9)`: the accessor layer the driver builds on (Chapter 16).
- `contigmalloc(9)`: the contiguous allocator used by `bus_dma` underneath.
- `busdma(9)` cross-references, including the `bus_dma_tag_template(9)` entry.
- `callout(9)`: the timer subsystem the simulation's engine uses.
- `condvar(9)`: the `cv_*` discipline the helper uses.

- `device(9)`: the general device framework.

The manual pages are maintained with the kernel and track changes to the API. Reading them once per major release (or after a significant change in the book's coverage) is a useful habit.

21.29 A Deeper Look at How DMA and Interrupts Fit Together

Chapters 19, 20, and 21 have built three interlocking pieces: interrupts, multiple vectors, DMA. This deeper look explains how they fit together in a complete high-performance driver, so the reader has a mental model of the whole before Chapter 22's power-management discipline arrives.

21.29.1 The Three-Piece Architecture

A modern DMA-capable driver has three cooperating subsystems:

1. **The DMA subsystem** (Chapter 21) owns the data path. It sets up buffers, programs the device, synchronises caches, and handles the transfer's data. Any time the driver moves data of non-trivial size, DMA is involved.
2. **The interrupt subsystem** (Chapters 19 and 20) owns the signalling path. The device signals events (data arrival, transfer completion, errors) via interrupts. The filter handles the signal in primary interrupt context; the task handles the follow-up in thread context.
3. **The coordination layer** (the `softc` and its mutex plus condition variables) owns the state between the two. The DMA subsystem writes `dma_in_flight` before triggering a transfer; the interrupt subsystem reads it during completion processing. The condition variable lets the DMA subsystem wait for the interrupt subsystem to signal completion.

The three pieces are designed to compose. A driver can have many DMA operations in flight, each signalling its completion through its own interrupt vector; the per-vector machinery from Chapter 20 handles this naturally, and the per-transfer state from Chapter 21 plugs into the per-vector tasks.

21.29.2 The Common Flow

For every DMA transfer in a production driver, the flow is:

1. Driver code wants to transfer data.
2. Driver locks the `softc`, sets up the per-transfer state.
3. Driver issues the PRE sync on the map.
4. Driver programs the device's registers with the bus address, length, direction, and flags.
5. Driver writes the doorbell.
6. Driver releases the lock (or sleeps on a condition variable under the lock).
7. Kernel does other work.
8. Device completes the transfer and raises its interrupt.
9. Filter runs, acknowledges, enqueues a task.
10. Task runs, acquires the `softc` lock, processes the completion: POST sync, status read, result recording.

11. Task broadcasts the condition variable and releases the lock.
12. Driver code (or its waiter) wakes up, reads the result, and proceeds.

The flow is the same for a simple single-buffer driver and for a multi-queue network driver with thousands of transfers in flight. The difference is scale: the network driver has many concurrent transfers, each with its own per-transfer state, but each one follows the same twelve-step flow.

21.29.3 The Scale Story

On Chapter 21's single-buffer simulation, the flow runs once per ten milliseconds. On a modern NIC, the flow runs millions of times per second across many queues. The machinery scales because:

- Interrupts are per-queue via MSI-X (Chapter 20).
- DMA buffers are per-queue via descriptor rings (a future chapter).
- The filter is short and filter-safe (Chapter 19).
- The task is per-queue and CPU-bound to a NUMA-local core (Chapter 20).
- The softc state is partitioned per-queue to avoid contention.

The cumulative design decisions Parts 4 and 5 teach are what make this scaling possible. A driver that respects the discipline scales naturally; a driver that violates it (a global lock instead of per-queue locks, a shared DMA map instead of per-transfer maps) hits scaling limits early.

21.29.4 The Interaction Patterns

Three specific interaction patterns worth recognising:

Pattern A: Interrupt delivers “new data is in the DMA ring”. The filter reads a completion index register, acknowledges, enqueues the task. The task walks the ring from the last index to the current, processes each entry, and updates its per-queue state. This is how a NIC's receive path works.

Pattern B: Interrupt delivers “DMA transfer N is complete”. The filter looks up transfer N in a per-transfer state array, enqueues the task. The task looks at transfer N's recorded state, performs the POST sync, and calls the completion callback. This is how a storage controller's command-completion path works.

Pattern C: Interrupt delivers “error occurred on DMA engine”. The filter may need more care here because errors often require device-level reset, which cannot happen in filter context. The filter enqueues a task with an error-specific payload; the task quiesces the engine, logs the error, and decides whether to reset.

Chapter 21's driver exercises a simplified Pattern B. Real drivers exercise all three, sometimes on separate vectors.

21.30 A Deeper Look at Comparisons with Other Kernel DMA APIs

For readers who come from other kernel contexts, a brief comparison between FreeBSD's `bus_dma(9)` and the equivalent APIs elsewhere. Not exhaustive; just enough to orient a reader who needs to translate between mental models.

21.30.1 Linux's DMA API

Linux uses `dma_alloc_coherent` / `dma_map_single` / `dma_sync_single_for_cpu` / `dma_sync_single_for_device` / `dma_unmap_single` for what FreeBSD calls `bus_dmamem_alloc` / `bus_dmamap_load` / `bus_dmamap_sync` (POST) / `bus_dmamap_sync` (PRE) / `bus_dmamap_unload`.

The semantic model is nearly identical:

- `dma_alloc_coherent` returns a CPU-visible virtual address and a DMA-visible bus address for a coherent buffer; `bus_dmamem_alloc` plus `bus_dmamap_load` does the same in two steps.
- `dma_map_single` maps an arbitrary buffer to a bus address, similar to `bus_dmamap_load` with a single-segment callback.
- `dma_sync_single_for_cpu` corresponds to `bus_dmamap_sync(..., POSTREAD)`; `for_device` corresponds to `PREWRITE` or `PREREAD`.
- `dma_unmap_single` is the Linux version of `bus_dmamap_unload`.

The FreeBSD API's distinctive features are the tag (an explicit constraint descriptor; Linux's approach is more implicit), the callback (for deferred loads; Linux uses DMA fence APIs instead), and the hierarchical tag inheritance (which Linux does not formalise the same way).

For Linux driver code being ported to FreeBSD, the rough translation is:

- Define a `bus_dma_tag_t` tag with the device's constraints.
- Replace `dma_alloc_coherent` with `bus_dma_tag_create` (once) plus `bus_dmamem_alloc` plus `bus_dmamap_load`.
- Replace `dma_map_single` per-transfer with `bus_dmamap_create` plus `bus_dmamap_load` (for dynamic maps).
- Replace every `dma_sync_*_for_cpu` with `bus_dmamap_sync(..., POSTREAD)` or `POSTWRITE`.
- Replace every `dma_sync_*_for_device` with `bus_dmamap_sync(..., PREREAD)` or `PREWRITE`.
- Replace `dma_unmap_single` with `bus_dmamap_unload`.

The DRM-kmod ports of Linux GPU drivers to FreeBSD use this translation extensively; a driver ported from Linux to FreeBSD typically gains explicit tag setup and explicit PRE/POST pairs where Linux had implicit ones.

21.30.2 NetBSD's `bus_dma`

NetBSD has the original `bus_dma` API that FreeBSD's is derived from. The function names are almost identical; the semantics are almost identical. The differences are mostly in peripheral APIs (template support, IOMMU integration, platform-specific extensions).

A driver written for NetBSD's `bus_dma` generally compiles on FreeBSD with minor adjustments. The portability is not accidental; it is the API's design goal.

21.30.3 Windows's DMA Abstractions

Windows uses a different abstraction (`AllocateCommonBuffer`, `IoMapTransfer`, `FlushAdapterBuffers`) with different semantics. Translating a Windows driver to FreeBSD is a more involved task because the Windows model does not have FreeBSD's tag concept and its synchronisation is less explicit.

A reader who works in both ecosystems benefits from understanding that the underlying hardware realities are the same everywhere; only the API surface differs. The `bus_dma` discipline from Chapter 21 transfers, albeit with a translation layer, to every kernel environment.

21.31 Reference: A Short Tour of `/usr/src/sys/kern/subr_bus_dma.c`

The kernel's implementation of the `bus_dma` API lives under `/usr/src/sys/kern/subr_bus_dma.c` and the architecture-specific `busdma_*.c` files. A short tour of the central file gives the curious reader a sense of where the machinery actually runs.

The file contains generic helpers that the architecture-specific backends call. The key entry points:

- `bus_dmamap_load_uio`: load a struct `uio` (user I/O).
- `bus_dmamap_load_mbuf_sg`: load an mbuf chain with a pre-allocated segment array.
- `bus_dmamap_load_ccb`: load a CAM control block.
- `bus_dmamap_load_bio`: load a struct `bio`.
- `bus_dmamap_load_crp`: load a crypto operation.

Each of these is a thin wrapper around the platform's `bus_dmamap_load_ma` (load array-of-mappings) or `bus_dmamap_load_phys` (load physical address), which are the architecture-specific primitives.

The file also contains the template API (`bus_dma_template_*`) and its helpers. The template code is about 200 lines and straightforward to read.

The generic file does not contain the per-architecture sync or bounce logic; that lives in `/usr/src/sys/x86/x86/busdma_bounce.c` for amd64 and i386, or in the equivalent platform directory. A reader who wants to understand what `bus_dmamap_sync` actually does on amd64 can read `busdma_bounce.c`'s `bounce_bus_dmamap_sync` function; it is about 100 lines and shows the bounce-copy logic plus the memory barriers.

21.32 Reference: Driver Memorable Phrases

A short list of the phrases worth memorising because they compress the chapter's discipline into a few words.

- “Tag describes, map is specific, sync signals ownership, unload reverses load.”
- “Every PRE has a POST; every setup has a teardown.”
- “`BUS_DMA_COHERENT` makes sync cheap, not unnecessary.”
- “The bus address is not always the physical address.”
- “`BUS_SPACE_MAXADDR_32BIT` is `lowaddr` for 32-bit devices; `highaddr` stays `BUS_SPACE_MAXADDR`.”
- “The callback may run later; use `BUS_DMA_NOWAIT` to avoid that case.”
- “The tag lockfunc is for deferred loads; `NULL` panics if deferred, which is the safer default.”
- “Single-segment callbacks look the same in every driver; the pattern is universal.”
- “Drain callouts before unloading maps; drain tasks before destroying tags.”

21.33 Reference: A Comparison Table Across Part 4

A table that summarises what each Part 4 chapter added, how the driver's version tag changed, and what new resource type the driver gained access to.

Chapter	Version	Added Subsystem	New Resource Types
16	0.9-mmio	Register access via <code>bus_space</code>	<code>bus_space_tag_t</code> , <code>bus_space_handle_t</code>
17	1.0-sim	Simulated hardware backend	sim backend, simulated register map
18	1.1-pci	Real PCI attach	BAR resource, PCI config access
19	1.2-intr	Legacy interrupt handling	IRQ resource, filter, task
20	1.3-msi	MSI/MSI-X multi-vector	per-vector IRQ resources, per-vector tasks
21	1.4-dma	DMA with <code>bus_dma(9)</code>	DMA tag, DMA map, DMA memory, bus address

Each chapter adds exactly one subsystem; the driver's complexity grows monotonically with its capability. A reader tracing the version history can see at a glance what the driver can do at any stage.

Part 4 closes with Chapter 22's power-management work, which does not add a new subsystem but adds discipline across all existing ones: every subsystem must be quiescable and resumable.

21.34 Reference: What the Reader Should Now Be Able to Do

A one-page self-check. After finishing Chapter 21 and completing the labs, the reader should be able to:

1. Open a random DMA-capable driver in `/usr/src/sys/dev/` and identify its tag creation, memory allocation, map loading, sync pattern, and teardown.
2. Explain why a driver calls `bus_dmamap_sync` with a PRE flag before triggering the device and a POST flag after the device completes.
3. Write a tag-creation call for a hypothetical device with documented constraints (alignment, size, addressing range).
4. Write a single-segment callback and use it to extract the bus address from a `bus_dmamap_load` call.
5. Recognise when a driver is using the static pattern versus the dynamic pattern.
6. Identify and explain the three portability realities `bus_dma(9)` hides: addressing limits, IOMMU remapping, cache coherence.
7. Debug a transfer that succeeds in terms of counters but produces corrupt data, by checking for missing or wrong-flavour syncs.
8. Construct a clean detach path that drains callouts, drains tasks, masks interrupts, unloads maps, frees memory, and destroys tags in the right order.
9. Distinguish between `BUS_DMA_COHERENT` at tag-create time (architectural hint) and at `bus_dmamem_alloc` time (allocation mode).
10. Explain why `BUS_DMA_NOWAIT` is the safer default for most drivers and when `BUS_DMA_WAITOK` is appropriate.

11. Write a correct single-buffer DMA driver from scratch, given a datasheet that specifies the register layout and the engine's behaviour.
12. Explain why the `bus_dma(9)` layer exists and what happens when a driver bypasses it.

A reader who checks off ten or more of these has internalised the chapter. The remaining items come with practice on real drivers.

21.35 Reference: When to Reach for Each API Variant

The `bus_dma(9)` API has several load variants; a quick reference to help readers choose correctly in future work:

- **`bus_dmamap_load`**: a generic kernel buffer at a known KVA and length. The simplest and most common call. Used by Chapter 21.
- **`bus_dmamap_load_mbuf_sg`**: an mbuf chain, common in network drivers. The `_sg` variant fills a pre-allocated segment array and avoids the callback.
- **`bus_dmamap_load_ccb`**: a CAM control block, used by storage drivers.
- **`bus_dmamap_load_uio`**: a struct `uio`, used when mapping user-supplied buffers.
- **`bus_dmamap_load_bio`**: a block I/O request, used in GEOM consumers.
- **`bus_dmamap_load_crp`**: a crypto operation, used by OpenCrypto transforms.

The chapter uses only the first. The reader who moves to networking, storage, or crypto later will see the specialised variants; each is a thin wrapper around the same underlying mechanism, with a helper that unpacks the subsystem's preferred buffer representation into segments.

For dynamic transactions where the driver maintains its own pool of maps, `bus_dmamap_create` and `bus_dmamap_destroy` complement the load calls. Chapter 21's static pattern does not use these; the dynamic pattern in Part 6's network chapter (Chapter 28) does.

A subtle point worth remembering: `bus_dmamem_alloc` both allocates memory and creates an implicit map. The map does not need `bus_dmamap_create` or `bus_dmamap_destroy`. The driver still calls `bus_dmamap_load` once to obtain the bus address (via the callback), and `bus_dmamap_unload` once at tear-down; but the map itself is managed by the allocator, not by the driver. A driver that calls `bus_dmamap_destroy` on an allocator-returned map triggers a panic. Keep the two lifecycles distinct.

21.36 Wrapping Up

Chapter 21 gave the driver the ability to move data. At the start, `myfirst` at version 1.3-`msi` could listen to its device through multiple interrupt vectors but touched every byte through `bus_read_4`. At the end, `myfirst` at version 1.4-`dma` has a `bus_dma(9)` setup and teardown path, a simulated DMA engine with a callout-driven completion, an interrupt-driven transfer helper that sleeps on a condition variable while the device works, a full PRE/POST sync discipline on every transfer, per-transfer counters for observability, a refactored `myfirst_dma.c` file, a `DMA.md` document, and a regression test that exercises every code path the chapter teaches.

The eight sections walked the full progression. Section 1 established the hardware picture: what DMA is, why it matters, what the real-world examples look like. Section 2 pinned down the `bus_dma(9)` vocabulary: tag, map,

memory, segment, PRE, POST, static, dynamic, parent, callback. Section 3 wrote the first running code: tag creation, memory allocation, map load, clean teardown. Section 4 laid down the sync discipline: the ownership model, the four flags, the four common patterns. Section 5 built the simulated DMA engine with register map, state machine, callout, and a polling-based driver helper. Section 6 rewrote the completion path to use the Chapter 20 per-vector interrupt machinery. Section 7 walked through twelve failure modes and the patterns that handle each. Section 8 refactored the code into `myfirst_dma.c`, updated the Makefile, bumped the version, added documentation, and closed the chapter's scope.

What Chapter 21 did not do is scatter-gather, descriptor rings, iflib integration, or user-space buffer mapping. Each of those is a natural extension built on Chapter 21's primitives, and each belongs in a later chapter (Part 6 for networking specifics, Part 7 for performance tuning). The foundation is in place; the specialisations add vocabulary without needing to rebuild the foundation.

The file layout has grown: 15 source files (including `cbuf`), 7 documentation files (`HARDWARE.md`, `LOCKING.md`, `SIMULATION.md`, `PCI.md`, `INTERRUPTS.md`, `MSIX.md`, `DMA.md`), and an extended regression suite. The driver is structurally parallel to production FreeBSD drivers; a reader who has worked through Chapters 16 through 21 can open `if_re.c`, `nvme_qpair.c`, or `ahci.c` and recognise the architectural parts: register accessors, simulation backend, PCI attach, interrupt filter and task, per-vector machinery, DMA setup and teardown, sync discipline, clean detach.

21.36.1 A Reflection Before Chapter 22

Chapter 21 was the last chapter in Part 4 that introduced a new hardware primitive. Chapters 16 through 21 took the driver from “no hardware awareness” to “fully functional hardware-backed DMA driver”. Chapter 22 is the discipline chapter: how to make this driver survive suspend and resume cycles, how to save and restore state around power transitions, how to quiesce every subsystem (interrupts, DMA, timers, tasks) before the device loses power and resume it cleanly afterwards. The DMA teardown path Chapter 21 built is what Chapter 22's suspend handler will call; the DMA setup path is what Chapter 22's resume handler will call.

Chapter 21's teaching also generalises. A reader who has internalised the tag-map-memory-sync discipline, the PRE/POST ownership model, the single-segment callback, and the clean setup/teardown pattern will find similar shapes in every DMA-capable FreeBSD driver. The specific device differs; the structure does not.

21.36.2 What to Do If You Are Stuck

Three suggestions.

First, focus on the Stage 2 polling path. If `sudo sysctl dev.myfirst.0.dma_test_write=0xAA` produces a correct banner in `dmesg`, the polling path is working. Every other piece of the chapter is optional in the sense that it decorates the pipeline, but if the pipeline fails, the whole chapter is not working and Section 5 is the right place to diagnose.

Second, open `/usr/src/sys/dev/re/if_re.c` and re-read `re_allocmem` slowly. It is about one hundred fifty lines of tag and memory setup code. Every line maps to a Chapter 21 concept. Reading it once after completing the chapter should feel like familiar territory; the real driver's patterns will look like elaborations of the chapter's simpler ones.

Third, skip the challenges on first pass. The labs are calibrated for Chapter 21's pace; the challenges assume the chapter's material is solid. Come back to them after Chapter 22 if they feel out of reach now.

Chapter 21's goal was to give the driver the ability to move data. If it has, Chapter 22's power-management machinery becomes a specialisation rather than an entirely new topic.

21.37 Bridge to Chapter 22

Chapter 22 is titled *Power Management*. Its scope is the discipline of saving and restoring a driver's state around system suspend and resume. Modern systems suspend aggressively: laptops suspend when the lid closes; servers suspend individual devices that are idle (PCIe's D0, D1, D2, D3 power states); virtual machines migrate between hosts. A driver that does not handle these transitions cleanly leaves its device in an inconsistent state, causes stuck resumes, or corrupts in-flight data.

Chapter 21 prepared the ground in three specific ways.

First, **you have a complete teardown path**. The DMA resources can be torn down and rebuilt cleanly; Chapter 22's suspend handler will call `myfirst_dma_tear_down`, and the resume handler will call `myfirst_dma_setup`. The versioning of state (attaching, detaching, and now suspending) is an extension of the patterns Chapter 21 already introduced.

Second, **you have an in-flight transfer tracker**. Chapter 21's `dma_in_flight` field and the `cv_timedwait` wait protocol are exactly what Chapter 22's suspend handler needs to ensure no transfer is pending when the device loses power. Reusing the tracker keeps the code uniform.

Third, **you have a clean interrupt mask API**. The driver masks and unmask `MYFIRST_INTR_COMPLETE` through the Chapter 19/20 machinery. Chapter 22's suspend handler will mask all interrupts before the power transition; Chapter 22's resume handler will unmask them after the device has stabilised.

Specific topics Chapter 22 will cover:

- What ACPI suspend states mean (S1, S3, S4) and what they require of drivers.
- PCIe device power states (D0, D1, D2, D3hot, D3cold) and how FreeBSD transitions between them.
- The `device_suspend` and `device_resume` methods; how to implement them.
- Quiescing DMA: how to ensure no in-flight transfer is outstanding when power is lost.
- Re-attaching after resume: re-initialising hardware state, reloading tables, restoring interrupts.
- Handling devices that reset across suspend: detecting the reset and rebuilding state.
- Integrating with the rest of the Chapter 16-21 machinery (`bus_space`, simulation, PCI, interrupts, DMA).

You do not need to read ahead. Chapter 21 is sufficient preparation. Bring your `myfirst` driver at `1.4-dma`, your `LOCKING.md`, your `INTERRUPTS.md`, your `MSIX.md`, your `DMA.md`, your WITNESS-enabled kernel, and your regression script. Chapter 22 starts where Chapter 21 ended.

Part 4 is nearly complete. Chapter 22 closes the part by adding the one remaining discipline that separates a prototype driver from a production driver: the ability to survive the power transitions that real systems impose.

The vocabulary is yours; the structure is yours; the discipline is yours. Chapter 22 adds the next missing piece: the driver's ability to gracefully stop and gracefully start, in response to events the driver itself did not initiate.

21.38 Reference: Chapter 21 Quick-Reference Card

A compact summary of the vocabulary, APIs, flags, and procedures Chapter 21 introduced.

21.38.1 Vocabulary

- **DMA (Direct Memory Access):** device reading or writing host memory without CPU per-byte involvement.
- **Bus-master:** a device capable of initiating bus transactions to host memory.
- **Tag (`bus_dma_tag_t`):** a description of DMA constraints for a group of transfers.
- **Map (`bus_dmamap_t`):** a mapping context for one specific transfer.
- **Segment (`bus_dma_segment_t`):** a (`bus_addr`, `length`) pair describing one contiguous piece of a mapping.
- **Bounce buffer:** a kernel-managed staging region used when the device cannot reach the driver's actual buffer.
- **Coherent memory:** memory allocated with `BUS_DMA_COHERENT`; sync operations are cheap.
- **Callback:** the function `bus_dmamap_load` calls with the segment list.
- **Static transaction:** a long-lived mapping allocated at attach time.
- **Dynamic transaction:** a per-transfer mapping created and destroyed per use.
- **Parent tag:** the tag inherited from the parent bridge; constraints compose via intersection.
- **PREWRITE/POSTWRITE/PREREAD/POSTREAD:** the four sync flags.
- **IOMMU:** input-output MMU; transparent remapping between device and host address spaces.

21.38.2 Essential APIs

- `bus_dma_tag_create(parent, align, bdry, low, high, filt, filtarg, maxsz, nseg, maxsegsz, flags, lockfn, lockarg, &tag)`: create a DMA tag.
- `bus_dma_tag_destroy(tag)`: destroy a DMA tag; fails if children remain.
- `bus_get_dma_tag(dev)`: return the device's parent DMA tag.
- `bus_dmamem_alloc(tag, &vaddr, flags, &map)`: allocate DMA-capable memory and get a map.
- `bus_dmamem_free(tag, vaddr, map)`: free `bus_dmamem_alloc` memory.
- `bus_dmamap_create(tag, flags, &map)`: create a map for dynamic loads.
- `bus_dmamap_destroy(tag, map)`: destroy a map.
- `bus_dmamap_load(tag, map, buf, len, callback, cbarg, flags)`: load a buffer into a map.
- `bus_dmamap_unload(tag, map)`: unload a map.
- `bus_dmamap_sync(tag, map, op)`: synchronise caches for `op` (one of the PRE/POST flags).

21.38.3 Essential Flags

- `BUS_DMA_WAITOK`: allocator may sleep.
- `BUS_DMA_NOWAIT`: allocator must not sleep; return `ENOMEM` if resources unavailable.
- `BUS_DMA_COHERENT`: prefer cache-coherent memory.
- `BUS_DMA_ZERO`: zero-initialise allocated memory.
- `BUS_DMA_ALLOCNOW`: pre-allocate bounce resources at tag-create time.

21.38.4 Essential Sync Operations

- `BUS_DMASYNC_PREREAD`: before device writes, host will read.
- `BUS_DMASYNC_PWRITE`: before device reads, host has written.

- `BUS_DMASYNC_POSTREAD`: after device writes, before host reads.
- `BUS_DMASYNC_POSTWRITE`: after device reads, host may reuse.

21.38.5 Common Procedures

Allocate static DMA buffer:

```
bus_dma_tag_create(bus_get_dma_tag(dev), ..., &sc->tag);
bus_dmamem_alloc(sc->tag, &sc->vaddr, BUS_DMA_WAITOK | BUS_DMA_COHERENT |
↳ BUS_DMA_ZERO, &sc->map);
bus_dmamap_load(sc->tag, sc->map, sc->vaddr, size, single_map_cb, &sc->bus_addr,
↳ BUS_DMA_NOWAIT);
```

Free static DMA buffer:

```
bus_dmamap_unload(sc->tag, sc->map);
bus_dmamem_free(sc->tag, sc->vaddr, sc->map);
bus_dma_tag_destroy(sc->tag);
```

Host-to-device transfer:

```
/* Fill sc->vaddr. */
bus_dmamap_sync(sc->tag, sc->map, BUS_DMASYNC_PREWRITE);
/* Program device, trigger, wait for completion. */
bus_dmamap_sync(sc->tag, sc->map, BUS_DMASYNC_POSTWRITE);
```

Device-to-host transfer:

```
bus_dmamap_sync(sc->tag, sc->map, BUS_DMASYNC_PREREAD);
/* Program device, trigger, wait for completion. */
bus_dmamap_sync(sc->tag, sc->map, BUS_DMASYNC_POSTREAD);
/* Read sc->vaddr. */
```

21.38.6 Useful Commands

- `sysctl hw.busedma`: DMA subsystem statistics.
- `vmstat -m | grep bus_dma`: memory use.
- `pciconf -lvbc <dev>`: device capability listing.
- `procstat -kke`: thread states and stacks.

21.38.7 Files to Keep Bookmarked

- `/usr/src/sys/sys/bus_dma.h`: the public header.
- `/usr/src/share/man/man9/bus_dma.9`: the manual page.
- `/usr/src/sys/dev/re/if_re.c`: production descriptor-ring driver.
- `/usr/src/sys/dev/nvme/nvme_qpair.c`: NVMe queue construction.

21.39 Reference: Glossary of Chapter 21 Terms

A short glossary of the chapter's new terms.

- **Alignment constraint**: the requirement that a buffer's starting address be a multiple of a specific value.
- **Boundary constraint**: the requirement that a buffer not cross a specific address boundary.
- **Bounce page**: a single page of bounce buffer used to stage data for a mapping.
- **Bus address**: the address a device uses to reach a memory region; may differ from the physical address.
- **Bus-master**: a device that can initiate transactions on the bus (i.e. do DMA).
- **Callback (DMA)**: the function `bus_dmamap_load` calls with the segment list after the load completes.
- **Coherent memory**: memory in a domain where CPU and DMA see the same data without explicit synchronisation.
- **Deferred load**: a load that returned `EINPROGRESS` and will have its callback called later.
- **Descriptor**: a small structure (in DMA memory) that describes one transfer: address, length, flags, status.
- **Descriptor ring**: an array of descriptors used for producer-consumer communication.
- **DMA tag**: see `bus_dma_tag_t`.
- **Doorbell**: an MMIO register a driver writes to signal new ring entries to the device.
- **Dynamic transaction**: see Section 2.
- **IOMMU**: input-output MMU; translates device-side addresses to host physical addresses.
- **KVA**: kernel virtual address; the pointer the CPU uses.
- **Load callback**: synonym for Callback (DMA).
- **Mapping**: a binding between a kernel buffer and a set of bus-visible segments.
- **Parent tag**: a tag from which another inherits constraints.
- **PREREAD, PREWRITE, POSTREAD, POSTWRITE**: sync flags; see Section 4.
- **Scatter-gather**: a mapping of multiple discontinuous segments as a single logical transfer.
- **Segment**: see `bus_dma_segment_t`.
- **Static transaction**: see Section 2.
- **Sync**: `bus_dmamap_sync`; the hand-off between CPU and device ownership of a buffer.

21.40 Reference: A Closing Note on DMA Philosophy

A paragraph to close the chapter with.

DMA is the primitive that turns a driver from a byte-at-a-time controller into a data-moving subsystem. Before DMA, every byte of data the driver handled went through the CPU's hands, one MMIO transaction at a time; after DMA, the CPU handles only policy and bookkeeping, and the device moves the bytes. The difference is the difference between a driver that can keep up with a 10 Mbit line and a driver that can keep up with a 100 Gbit line.

Chapter 21's lesson is that DMA is disciplined, not magical. The `bus_dma(9)` API hides three hardware realities (addressing limits, IOMMU remapping, cache coherence) behind a single set of calls, and the calls follow a predictable pattern: create a tag, allocate memory, load a map, sync PRE, trigger the device, wait, sync POST, read the result, eventually unload and free. The pattern is the same across every DMA-capable driver in FreeBSD; internalising it once pays off across dozens of later chapters and thousands of lines of real driver code.

For this reader and for this book's future readers, the Chapter 21 DMA pattern is a permanent part of the `myfirst` driver's architecture and a permanent tool in the reader's toolkit. Chapter 22 assumes it: suspend needs to quiesce DMA; resume needs to reinitialise it. Part 6's networking chapters assume it: every packet path uses DMA. Part 7's performance chapters (Chapter 33) assume it: every tuning measurement is against DMA throughput. The vocabulary is the vocabulary every high-performance FreeBSD driver shares; the patterns are the patterns production drivers live by; the discipline is the discipline that keeps coherent platforms coherent and non-coherent platforms correct.

The skill Chapter 21 teaches is not "how to set up a single 4 KB DMA buffer". It is "how to think about memory ownership between the CPU and the device, how to describe a device's constraints to the kernel, and how to move data under a sync discipline that works portably on every platform FreeBSD supports". That skill applies across every DMA-capable device the reader will ever work on.

22 Power Management

22.1 Reader Guidance & Outcomes

Chapter 21 ended with the driver at version 1.4-dma. That driver attaches to a PCI device, allocates MSI-X vectors, services interrupts through a filter-plus-task pipeline, moves data through a `bus_dma(9)` buffer, and cleans up its resources when it is asked to detach. For a driver that boots, runs, and eventually gets unloaded, those mechanics are complete. What the driver does not yet handle is the third kind of event a modern system throws at it: the moment when power itself is about to change.

Power changes are different from attach and detach. An attach starts from nothing and ends with a working device. A detach starts from a working device and ends with nothing. Both are one-shot transitions the driver itself can take its time over. A suspend is neither. The driver enters suspend already running, with live interrupts, live DMA transfers, live timers, and a device the kernel still expects to answer questions. The driver has to bring all of that to a stop within a narrow time window, hand the device over to a lower-power state, survive the power loss without forgetting what it needs to know, and then reassemble everything on the other side as if nothing happened. The user, ideally, notices nothing at all. The lid closes, a second later it opens, and the video conference resumes in the same browser tab as if the interruption had never happened.

Chapter 22 teaches how the driver earns that illusion. The chapter's scope is precisely this: what power management is at the driver level; how the kernel lets the driver see a power transition coming; what it means to quiesce a device so that no activity leaks across the transition; how to preserve the state the driver will need after resume; how to restore that state so the device comes back to the same behavior the user saw before; how to extend the same discipline to idle-device power saving through runtime suspend; how to test transitions from user space; how to debug the characteristic failures a power-aware driver faces; and how to organise the new code so that the driver stays readable as it grows. The chapter stops short of the later topics that build on this discipline. Chapter 23 teaches debugging and tracing in depth; the power-aware regression script in Chapter 22 is a first taste, not the full toolkit. The network-driver chapter in Part 6 (Chapter 28) adds `iflib`'s power hooks and multi-queue suspend coordination; Chapter 22 stays with the single-queue `myfirst` driver. The advanced real-world chapters in Part 7 explore hotplug and power-domain management on embedded platforms; Chapter 22 focuses on the desktop and server cases where ACPI and PCIe dominate.

The arc of Part 4 closes here with a discipline rather than a new primitive. Chapter 16 gave the driver a vocabulary of register access through `bus_space(9)`. Chapter 17 taught it to think like a device by simulating one. Chapter 18 introduced it to a real PCI device. Chapter 19 gave it one pair of ears on a single IRQ. Chapter 20 gave it several ears, one per queue the device cares about. Chapter 21 gave it hands: the ability to hand the device a physical address and let the device run a transfer on its own. Chapter 22 teaches the driver to stop doing all of that on request, to wait politely while the system sleeps, and to pick up again cleanly when the system wakes. That discipline is the last missing ingredient before the driver can call itself production-ready in the Part 4 sense. The later chapters add observability, specialisation, and polish; they assume the power discipline is in place.

22.1.1 Why Suspend and Resume Earn a Chapter of Their Own

A natural question at this stage is whether suspend and resume really need a full chapter, after the depth of Chapter 21. The `myfirst` driver already has a clean detach path. Detach already releases interrupts, drains tasks, tears down DMA, and returns the device to a quiet state. Can the driver not simply call detach on suspend and attach on resume, and be done?

The answer is no, for three linked reasons.

The first is that **suspend is not detach**. A detach is permanent. The driver does not need to remember anything about the device after detach finishes; when the device comes back, it is a fresh attach, from scratch. A suspend is temporary, and the driver does need to remember things across it. It needs to remember its software state so the user's session can pick up where it left off. It needs to remember which interrupt vectors it had allocated. It needs to remember its configuration sysctls. It needs to remember which clients had the device open. Detach forgets all of that; suspend must not. The two paths share cleanup steps in their middle, but they diverge at the ends. Treating suspend as a detach plus a later attach would be correct in the narrow mechanical sense and wrong in every other sense: it would drop the user's session, invalidate open file descriptors on `/dev/myfirst0`, lose the sysctl state, and ask the kernel to re-probe the device from its raw PCI identity on every resume. That is not how modern FreeBSD drivers work, and Chapter 22 shows the better pattern.

The second is that **the time budget is different**. A detach can afford to be thorough. A driver that takes five hundred milliseconds to detach has no user-facing impact; detaches happen at boot, at module unload, or at device removal, and those moments are understood to be slow. A suspend has to finish within a budget measured in tens of milliseconds per device on a laptop with a hundred devices, because the sum is what the user notices as lid-close latency. A driver that does a full detach-like cleanup, waits for queues to drain at their natural pace, unwinds every allocation, and rebuilds everything on resume will be measurably slow across the whole fleet of devices the system has. The Chapter 22 pattern is to stop activity fast, save what needs to be saved, leave allocations in place, and restore from saved state. That pattern is what keeps suspend-resume under a second on a typical laptop.

The third is that **the kernel gives the driver a specific contract for power transitions**, and that contract has its own vocabulary, its own order of operations, and its own failure modes. The `DEVICE_SUSPEND` and `DEVICE_RESUME` kobj methods are not just “detach and attach with different names”. They are called at specific points in the system-wide suspend sequence, with the driver tree traversed in a specific order, and they interact with the PCI layer's automatic config-space save and restore, with ACPI's sleep-state machinery, with the interrupt subsystem's mask and unmask calls, and with the `bus_generic_suspend` and `bus_generic_resume` helpers that walk the device tree. A driver that ignores the contract may still look correct during detach, during DMA, and during interrupt handling, and then fail only when the user closes the lid. That class of failure is notoriously hard to debug because it is hard to reproduce, and Chapter 22 invests time in making the contract explicit so the failures do not happen in the first place.

Chapter 22 earns its place by teaching those three ideas together, concretely, with the `myfirst` driver as the running example. A reader who finishes Chapter 22 can add `device_suspend`, `device_resume`, and `device_shutdown` methods to any FreeBSD driver, knows which of the chapter's discipline applies where, and understands the interactions between the ACPI layer, the PCI layer, and the driver's own state. That skill transfers directly to every FreeBSD driver the reader will ever work on.

22.1.2 Where Chapter 21 Left the Driver

A short checkpoint before we continue. Chapter 22 extends the driver produced at the end of Chapter 21 Stage 4, tagged as version 1.4-dma. If any of the items below is uncertain, return to Chapter 21 before starting this chapter.

- Your driver compiles cleanly and identifies itself as 1.4-dma in `kldstat -v`.
- The driver allocates one or three MSI-X vectors (depending on platform), registers per-vector filters and tasks, binds each vector to a CPU, and prints an interrupt banner during attach.
- The driver allocates a `bus_dma` tag, allocates a 4 KB DMA buffer, loads it into a map, and exposes the bus address through `dev.myfirst.N.dma_bus_addr`.
- Writing to `dev.myfirst.N.dma_test_write=0xAA` triggers a host-to-device transfer; writing to `dev.myfirst.N.dma_test_read=1` triggers a device-to-host transfer; both log success to `dmesg`.
- The detach path drains the rx task, drains the simulation's callout, waits for any in-flight DMA to complete, calls `myfirst_dma_tear_down`, tears down MSI-X vectors in reverse order, and releases resources.
- `HARDWARE.md`, `LOCKING.md`, `SIMULATION.md`, `PCI.md`, `INTERRUPTS.md`, `MSIX.md`, and `DMA.md` are current in your working tree.
- `INVARIANTS`, `WITNESS`, `WITNESS_SKIPSPIN`, `DDB`, `KDB`, and `KDB_UNATTENDED` are enabled in your test kernel.

That driver is what Chapter 22 extends. The additions are modest in lines but important in discipline: a new `myfirst_power.c` file, a matching `myfirst_power.h` header, a small set of new softc fields to track suspend state and saved runtime state, new `myfirst_suspend` and `myfirst_resume` entry points wired into the `device_method_t` table, a new `myfirst_shutdown` method, a call to the Chapter 21 quiesce primitives from the new suspend path, a restoration path that reinitialises the device without repeating attach, a version bump to 1.5-power, a new `POWER.md` document, and updates to the regression test. The mental model grows too: the driver starts thinking about its own lifecycle as attach, run, quiesce, sleep, wake, run again, and eventually detach, rather than just attach, run, detach.

22.1.3 What You Will Learn

Walking away from this chapter, you should be able to:

- Describe what power management means for a device driver, distinguish system-level and device-level power saving, and name the difference between a full suspend-resume cycle and a runtime power transition.
- Recognise the ACPI system sleep states (S0, S1, S3, S4, S5) and the PCI device power states (D0, D1, D2, D3hot, D3cold), explain how they compose in a single transition, and identify which parts of each are the driver's responsibility.
- Explain the role of PCIe link states (L0, L0s, L1, L1.1, L1.2) and active-state power management (ASPM), at a level sufficient to read a datasheet and recognise what the platform controls automatically versus what the driver must configure explicitly.
- Add `DEVICE_SUSPEND`, `DEVICE_RESUME`, `DEVICE_SHUTDOWN`, and (optionally) `DEVICE_QUIESCE` entries to a driver's `device_method_t` table, and implement each of them so they compose with `bus_generic_suspend(9)` and `bus_generic_resume(9)` in the device tree.

- Explain what it means to quiesce a device safely before a power transition, and apply the pattern: mask interrupts at the device, stop submitting new DMA work, drain the in-flight transfers, drain the callouts and taskqueues, flush or discard buffers as policy dictates, and leave the device in a defined, quiet state.
- Explain why the PCI layer automatically saves and restores configuration space around `device_suspend` and `device_resume`, when the driver has to supplement that with its own `pci_save_state/pci_restore_state` calls, and when it should not.
- Implement a clean resume path that re-enables bus-mastering, restores device registers from the driver's saved state, re-arms the interrupt mask, revalidates device identity, and re-admits clients to the device without losing data or raising spurious interrupts.
- Recognise the cases where a device silently resets across suspend, how to detect the reset, and how to rebuild only the state that was actually lost.
- Implement a runtime power management helper that puts an idle device into D3 and wakes it back to D0 on demand, and discuss the latency versus power tradeoff.
- Trigger a full-system suspend from user space with `acpiconf -s 3` or `zzz`, a per-device suspend with `devctl suspend` and `devctl resume`, and observe the transitions through `devinfo -v`, `sysctl hw.acpi.*`, and the driver's own counters.
- Debug the characteristic failures of power-aware code: frozen devices, lost interrupts, bad DMA after resume, missed PME# wake events, and WITNESS complaints about sleep-with-locks-held inside suspend. Apply the matching recovery patterns.
- Refactor the driver's power-management code into a dedicated `myfirst_power.c/myfirst_power.h` pair, bump the driver's version to 1.5-power, extend the regression test to cover suspend and resume, and produce a `POWER.md` document that explains the subsystem to the next reader.
- Read the power-management code in a real driver such as `/usr/src/sys/dev/re/if_re.c`, `/usr/src/sys/dev/xl/if_xl.c`, or `/usr/src/sys/dev/virtio/block/virtio_blk.c`, and map each call back to the concepts introduced in Chapter 22.

The list is long. The items are narrow. The chapter's goal is the composition, not any single item in isolation.

22.1.4 What This Chapter Does Not Cover

Several adjacent topics are explicitly deferred so Chapter 22 stays focused on the driver-side discipline.

- **Advanced ACPI internals** such as the AML interpreter, the SSDT/DSDT tables, the `_PSW/_PRW/_PSR` method semantics, and the ACPI button subsystem. The chapter uses ACPI only through the layer the kernel exposes to a driver; the internals belong in a later, platform-oriented chapter.
- **Hibernate-to-disk mechanics (S4)**. FreeBSD's S4 support has historically been partial on x86, and the driver-side contract is essentially a stricter version of S3. The chapter mentions S4 for completeness and treats it like S3 for driver purposes.
- **Cpufreq, powerd, and CPU frequency scaling**. These affect CPU power, not device power. A driver whose device is in D0 is unaffected by the CPU's P-state; the chapter does not pursue CPU power management.
- **SR-IOV suspend coordination between PF and VFs**. Virtual Function suspend has its own ordering constraints and belongs in a specialised chapter.
- **Hotplug and surprise removal**. Removing a device by physical unplug is similar in spirit to suspend but uses different code paths (`BUS_CHILD_DELETED`, `device_delete_child`). Part 7 covers hotplug in depth; Chapter 22 mentions the relationship and moves on.

- **Thunderbolt and USB-C dock suspend.** These compose ACPI, PCIe hotplug, and USB power management and belong in a dedicated later section.
- **Embedded-platform power-domain and clock-gating frameworks** such as the device-tree power-domains and clocks properties on arm64 and RISC-V. The chapter uses x86 ACPI and PCI conventions throughout, and mentions the embedded counterparts in passing when the concept is parallel.
- **Custom wake-on-LAN policy, wake-on-pattern policy, and application-specific wake sources.** The chapter explains how a wake source is plumbed (PME#, USB remote wakeup, GPIO wake) without trying to teach every hardware-specific variation.
- **The internals of the ksuspend/kresume paths and the kernel's cpuset migration around suspend.** The driver does not see those directly; they affect interrupt routing and CPU offlining, not the driver's visible contract.

Staying inside those lines keeps Chapter 22 a chapter about driver-side power discipline. The vocabulary transfers; the specialisations add detail in later chapters without needing a new foundation.

22.1.5 Estimated Time Investment

- **Reading only:** four to five hours. The power management conceptual model is neither as dense as DMA nor as mechanical as interrupts; much of the time is in building the mental picture of how ACPI, PCI, and the driver compose during a transition.
- **Reading plus typing the worked examples:** ten to twelve hours over two or three sessions. The driver evolves in three stages: skeleton suspend and resume with logging, full quiesce and restore, and finally refactor into `myfirst_power.c`. Each stage is short, but the testing is deliberate: a forgotten `bus_dmamap_sync` or a missed interrupt mask can produce silent corruption that only shows up on the fifth or sixth suspend-resume cycle.
- **Reading plus all labs and challenges:** fifteen to twenty hours over four or five sessions, including the lab that stresses the driver through repeated suspend-resume cycles, the lab that forces an intentional post-resume failure and debugs it, and the challenge material that extends the driver with runtime idle detection.

Sections 3 and 4 are the densest. If the quiesce discipline or the resume ordering feels opaque on first pass, that is normal. Stop, re-read the corresponding diagram, run the matching exercise on the simulated device, and continue when the shape has settled. Power management is one of those topics where a working mental model pays off repeatedly; it is worth building slowly.

22.1.6 Prerequisites

Before starting this chapter, confirm:

- Your driver source matches Chapter 21 Stage 4 (1.4-dma). The starting point assumes every Chapter 21 primitive: the DMA tag and buffer, the `dma_in_flight` tracker, the `dma_cv` condition variable, and the clean teardown path.
- Your lab machine runs FreeBSD 14.3 with `/usr/src` on disk and matching the running kernel.
- A debug kernel with `INVARIANTS`, `WITNESS`, `WITNESS_SKIPSPIN`, `DDB`, `KDB`, and `KDB_UNATTENDED` is built, installed, and booting cleanly. The `WITNESS` option is especially valuable for suspend and resume

work, because the code paths run under non-obvious locks and the kernel's power machinery tightens several invariants during the transition.

- `bhyve(8)` or `qemu-system-x86_64` is available. Chapter 22's labs work on either target. The suspend-resume testing does not require real hardware; `devctl suspend` and `devctl resume` let you drive the driver's power methods directly without involving ACPI.
- The `devinfo(8)`, `sysctl(8)`, `pciconf(8)`, `procstat(1)`, `devctl(8)`, `acpiconf(8)` (if on real hardware with ACPI) and `zzz(8)` commands are in your path.

If any item above is shaky, fix it now. Power management, like DMA, is a topic where latent weaknesses show up under stress. A driver that almost works on detach often breaks on suspend; a driver that handles one suspend cleanly often fails on the tenth cycle because a counter wrapped, a map leaked, or a condition variable was reinitialised incorrectly. The WITNESS-enabled debug kernel is what surfaces those mistakes at development time.

22.1.7 How to Get the Most Out of This Chapter

Four habits will pay off quickly.

First, keep `/usr/src/sys/kern/device_if.m` and `/usr/src/sys/kern/subr_bus.c` bookmarked. The first file defines the `DEVICE_SUSPEND`, `DEVICE_RESUME`, `DEVICE_SHUTDOWN`, and `DEVICE_QUIESCE` methods; the second contains `bus_generic_suspend`, `bus_generic_resume`, `device_quiesce`, and the `devctl` machinery that turns user-space requests into method calls. Reading both once at the start of Section 2 and returning to them as you work each section is the single most useful thing you can do for fluency.

Second, keep three real-driver examples close to hand: `/usr/src/sys/dev/re/if_re.c`, `/usr/src/sys/dev/xl/if_xl.c`, and `/usr/src/sys/dev/virtio/block/virtio_blk.c`. Each illustrates a different power-management style. `if_re.c` is a full network driver with wake-on-LAN support, config-space save and restore, and a careful resume path. `if_xl.c` is simpler: its `xl_shutdown` just calls `xl_suspend`, and `xl_suspend` stops the chip and sets up wake-on-LAN. `virtio_blk.c` is minimal: `vtblk_suspend` sets a flag and quiesces the queue, `vtblk_resume` clears the flag and restarts I/O. Chapter 22 will refer back to each of them at the moment where their pattern best illustrates what the `myfirst` driver is doing.

Third, type the changes by hand and exercise each stage with `devctl suspend` and `devctl resume`. Power management is where small omissions produce characteristic failures: a forgotten interrupt mask causes a stuck resume; a forgotten `bus_dmamap_sync` causes stale data; a forgotten state variable causes the driver to think a transfer is still in flight. Typing carefully and running the regression script after each stage exposes those mistakes at the moment they happen.

Fourth, after finishing Section 4, re-read Chapter 21's detach path. The quiesce discipline in Section 3 and the restore discipline in Section 4 both share infrastructure with Chapter 21's detach: `callout_drain`, `taskqueue_drain`, `bus_dmamap_sync`, `pci_release_msi`. Seeing suspend-resume and attach-detach side by side is what makes the differences visible. Suspend is not detach; resume is not attach; but they use the same building blocks, composed differently, and seeing that composition in two passes is worth the extra half-hour.

22.1.8 Roadmap Through the Chapter

The sections in order are:

1. **What Is Power Management in Device Drivers?** The big picture: why a driver cares about power, how system-level and device-level power management differ, what the ACPI S-states and PCI D-states mean, what PCIe link states and ASPM add, and what wake sources look like on the systems the reader is most likely to have. Concepts first, APIs next.
2. **FreeBSD's Power Management Interface.** The kobj methods: `DEVICE_SUSPEND`, `DEVICE_RESUME`, `DEVICE_SHUTDOWN`, `DEVICE_QUIESCE`. The order in which the kernel delivers them. The `bus_generic_suspend` helper, the `pci_suspend_child` path, and the interaction with ACPI. The first running code: Stage 1 of the Chapter 22 driver (`1.5-power-stage1`), with skeleton handlers that only log.
3. **Quiescing a Device Safely.** Stopping activity before a power transition. Masking interrupts, stopping DMA submission, draining in-flight work, draining callouts and taskqueues, flushing policy-sensitive buffers. Stage 2 (`1.5-power-stage2`) turns the skeletons into a real quiesce.
4. **Restoring State on Resume.** Reinitialising the device from saved state. What PCI save/restore does for you and what it does not. Re-enabling bus-master, restoring device registers, re-arming interrupts, validating identity, handling device reset. Stage 3 (`1.5-power-stage3`) adds the resume path that matches the Stage 2 quiesce.
5. **Handling Runtime Power Management.** Idle-device power saving. Detecting idleness. Putting the device into D3 and bringing it back to D0 on demand. Latency versus power. The chapter's optional section, but with a practical sketch a reader can experiment with.
6. **Interacting With the Power Framework.** Testing transitions from user space. `acpiconf -s 3` and `zzz` for full-system suspend. `devctl suspend` and `devctl resume` for per-device suspend. `devinfo -v` for observing power states. The regression script that wraps all of it.
7. **Debugging Power Management Issues.** The characteristic failure modes: frozen device, lost interrupts, bad DMA after resume, missing PME# wake, WITNESS complaints. The debugging patterns that find each of them.
8. **Refactoring and Versioning Your Power-Aware Driver.** The final split into `myfirst_power.c` and `myfirst_power.h`, the updated Makefile, the `POWER.md` document, and the version bump. Stage 4 (`1.5-power`).

After the eight sections come an extended walkthrough of `if_re.c`'s power-management code, several deeper looks at ACPI sleep states, PCIe link states, wake sources, and the `devctl` user-space interface, a set of hands-on labs, a set of challenge exercises, a troubleshooting reference, a Wrapping Up that closes Chapter 22's story and opens Chapter 23's, a bridge, and the usual quick-reference and glossary material at the end of the chapter. The reference material is meant to be re-read as you work through the next few chapters; Chapter 22's vocabulary (suspend, resume, quiesce, shutdown, D0, D3, ASPM, PME#) is the foundation every production FreeBSD driver shares.

If this is your first pass, read linearly and do the labs in order. If you are revisiting, Sections 3, 4, and 7 stand alone and make good single-sitting reads.

22.2 Section 1: What Is Power Management in Device Drivers?

Before the code, the picture. Section 1 teaches what power management means at the level the driver sees: the layers of the system that cooperate to save power, the sleep states and device power states those layers define, the PCIe link-level machinery that happens below the driver's visibility, and the wake sources that bring the system back. A reader who finishes Section 1 can read the rest of the chapter with the vocabulary of ACPI and

PCI power management as concrete objects rather than as vague acronyms.

22.2.1 Why the Driver Must Care About Power

The reader has spent the previous six chapters teaching a driver how to talk to a device. Each chapter added a capability: memory-mapped registers, simulated backends, real PCI, interrupts, multi-vector interrupts, DMA. In every chapter, the device was always ready to respond. The BAR was always mapped. The interrupt vector was always armed. The DMA engine was always on. That assumption is convenient for teaching, and it is also the assumption the driver makes during normal operation. It is not, however, the assumption the user makes. The user assumes that when the laptop sleeps, the battery drains slowly; that when the NVMe is idle, it cools itself; that when the Wi-Fi card has nothing to transmit, it is not pulling watts from the power supply. Those assumptions are real platform engineering, and the driver is one of the layers that has to cooperate for the engineering to work.

Cooperating means acknowledging that the device's power state can change under the driver's feet. The change is always announced: the kernel calls a method in the driver to tell it what is about to happen. But the announcement only works if the driver handles it correctly. A driver that ignores the announcement leaves the device in an inconsistent state, and the cost shows up in ways that are specific to power transitions: a laptop that does not wake up, a server whose RAID controller refuses to respond after a device-level reset, a USB dock that loses connectivity across a lid-close. Each of these failures maps back to a driver that treated a power event as optional when the platform's correctness assumed it was mandatory.

The stakes are not only about idle power. A driver that does not quiesce DMA before the system suspends can corrupt memory at the moment the CPU stops. A driver that does not mask interrupts before a bus enters a lower power state can cause spurious wake events. A driver that does not restore its configuration after resume can read zeros from a register that used to hold a valid address. Each of those is a kernel bug whose symptom is reported by the user as "sometimes my machine doesn't wake up". Chapter 22's discipline is what prevents that class of bug.

22.2.2 System-Level Versus Device-Level Power Management

Two words that sound similar describe different things. It is worth pinning them down now, because the chapter uses both and the distinction matters throughout.

System-level power management is a transition of the whole computer. The user presses the power button, closes the lid, or issues `shutdown -p now`. The kernel walks the device tree, asks every driver to suspend its device, and then either parks the CPU in a low-power state (S1, S3), writes the memory contents to disk (S4), or turns the power off (S5). Every driver in the system participates in the transition. If any driver refuses, the whole transition fails; the kernel prints a message like `DEVICE_SUSPEND(foo0) failed: 16`, the system stays awake, and the user sees a laptop whose screen went dark for half a second and then came back.

Device-level power management is a transition of one device. The kernel decides (or is told by `devctl suspend`) that one specific device can be put into a lower power state, independent of any other device. The PCIe NIC, for example, can go to D3 when its link has been idle for a few seconds, and come back to D0 when a packet arrives. The whole system stays in S0 throughout. The rest of the devices continue to work. The user does not notice anything except a slight increase in the latency of the first packet after an idle period, because the NIC had to wake up from D3.

System-level and device-level transitions use the same driver methods. `DEVICE_SUSPEND` is called both for a full S3 transition (where every device suspends together) and for a targeted `devctl suspend myfirst0` (where only the `myfirst0` device suspends). The driver usually does not distinguish the two; the same quiesce discipline works for both. What differs is the context around the call: a system-wide suspend also disables all CPUs except the boot CPU, stops most kernel threads, and expects every driver to be done quickly; a per-device suspend leaves the rest of the system running and is called from normal kernel context. The driver mostly does not have to care. It does, however, need to be aware that the two contexts exist, because a driver that only tests per-device suspend may miss a bug that only shows up in a full-system suspend, and vice versa.

Chapter 22 exercises both paths. The labs use `devctl suspend` and `devctl resume` for fast iteration, because they take milliseconds and do not involve ACPI. The integration test uses `acpiconf -s 3` (or `zzz`) to exercise the full path through the ACPI layer and the bus hierarchy. A driver that passes both tests is much more likely to be correct in production than one that passes only the first.

22.2.3 ACPI System Sleep States (S0 to S5)

On the x86 laptops and servers most readers will work with, the system's power state is described by the ACPI specification as a small set of letters and numbers called the **S-states**. Each S-state defines a distinct level of “how much of the system is still running”. The driver does not choose an S-state (the user, the BIOS, or the kernel's policy chooses), but it has to know which ones exist and what each one implies for the device.

S0 is the working state. The CPU is running, RAM is powered, all devices are in whatever device power state they need. Everything the reader has done so far has been in S0. This is the state the system boots into and the state it leaves only when sleep is requested.

S1 is called “standby” or “light sleep”. The CPU stops executing, but the CPU's registers and caches are preserved, RAM stays powered, and most devices stay in D0 or D1. Wake-up is fast (usually a second or less). On modern hardware S1 is rarely used, because S3 is more power-efficient and almost as fast to wake from. FreeBSD supports S1 if the platform advertises it; most platforms no longer do.

S2 is a rarely-implemented state between S1 and S3. On most platforms it is not advertised, and when it is, FreeBSD treats it similarly to S1. The chapter does not return to S2.

S3 is “suspend to RAM”, also known as “standby” or “sleep” in user-facing language. The CPU stops, the CPU context is lost and must be saved before the transition, RAM stays powered through its self-refresh mechanism, and most devices go to D3 or D3cold. Wake-up takes one to three seconds on a typical laptop. This is the state the user enters by closing the lid on a laptop. On a server, S3 is the state `acpiconf -s 3` or `zzz` produces. Chapter 22's main test is an S3 transition.

S4 is “suspend to disk” or “hibernate”. The full contents of RAM are written to a disk image, power is removed, and the system comes back by reading the image on the next boot. On FreeBSD, S4 support has historically been partial on x86 (the memory image can be produced but the restore path is not as polished as Linux or Windows). For driver purposes, S4 looks like S3 with an extra step at the end: the driver suspends exactly as it would for S3. The difference is invisible to the driver.

S5 is “soft off”. The system is powered off; only the wake-up circuitry (power button, wake-on-LAN) is receiving power. From the driver's point of view, S5 is similar to a system shutdown; the `DEVICE_SHUTDOWN` method is what gets called, not `DEVICE_SUSPEND`.

On real hardware, the reader can see which sleep states the platform supports with:

```
sysctl hw.acpi.supported_sleep_state
```

A typical laptop prints something like:

```
hw.acpi.supported_sleep_state: S3 S4 S5
```

A server might print only S5, because ACPI suspend is rarely meaningful on a datacenter machine. A VM might print varied combinations depending on the hypervisor. The `sysctl hw.acpi.s4bios` tells whether S4 is BIOS-assisted (it rarely is on modern systems). The `sysctl hw.acpi.sleep_state` lets the reader enter a sleep state manually; `acpiconf -s 3` is the preferred command-line wrapper.

For Chapter 22's purposes, the reader needs to be aware of S3 (the common case) and S5 (the shutdown case). S1 and S2 are handled by the driver the same way as S3; S4 is a superset of S3. The chapter treats S3 as the canonical example throughout.

22.2.4 PCI and PCIe Device Power States (D0 to D3cold)

A device's power state is described by the **D-states**, which the PCI specification defines independently from the system's S-state. The driver's methods most directly control the D-state of its device, and it is worth understanding each state in detail.

D0 is the fully-on state. The device is powered, clocked, accessible through its configuration space and BARs, and able to perform any operation the driver asks for. All the Chapter 16 to 21 work has been done with the device in D0. `PCI_POWERSTATE_D0` is the symbolic constant in `/usr/src/sys/dev/pci/pcivar.h`.

D1 and **D2** are intermediate low-power states that the PCI spec defines but does not tightly constrain. A device in D1 still has its configuration registers accessible and can respond to some I/O; a device in D2 may have lost more context. These states are rarely used on modern PCs because the jump from D0 to D3 is usually preferable for power savings. Most drivers do not bother with D1 and D2.

D3hot is the low-power state in which the device is effectively off, but the bus is still powered, the configuration space can still be accessed (reads return mostly zero or the preserved configuration), and the device can raise a PME# signal if it is configured to do so. Most devices enter D3hot during suspend.

D3cold is the even-lower power state in which the bus itself has lost power. The device cannot be accessed at all; reads from its configuration space return all-ones. The only way out of D3cold is for the platform to restore power, which usually happens under platform (not driver) control. D3cold is common during full-system S3 and S4.

When a driver calls `pci_set_powerstate(dev, PCI_POWERSTATE_D3)`, the PCI layer transitions the device from its current state to D3 (D3hot specifically; the transition to D3cold is the platform's job). When the driver calls `pci_set_powerstate(dev, PCI_POWERSTATE_D0)`, the PCI layer brings the device back to D0.

The PCI layer in FreeBSD also automatically manages these transitions during system suspend and resume. The `pci_suspend_child` function, which the PCI bus driver registers for `bus_suspend_child`, calls the driver's `DEVICE_SUSPEND` first, and then (if the `hw.pci.do_power_suspend` sysctl is true, which it is by default) transitions the device to D3. On resume, `pci_resume_child` transitions the device back to D0, restores the configuration space from a cached copy, clears any pending PME# signal, and then calls the driver's `DEVICE_RESUME`. The reader can observe the behavior with:

```
sysctl hw.pci.do_power_suspend  
sysctl hw.pci.do_power_resume
```

Both default to 1. A reader who wants to disable the automatic D-state transition (for debugging, or for a device that misbehaves in D3) can set them to 0, in which case the driver's `DEVICE_SUSPEND` and `DEVICE_RESUME` run but the device stays in D0 through the transition.

For Chapter 22, the important facts are:

- The driver's `DEVICE_SUSPEND` method runs before the D-state changes. The driver quiesces while the device is still in D0.
- The driver's `DEVICE_RESUME` method runs after the device has been returned to D0. The driver restores while the device is accessible.
- The driver does not (usually) call `pci_set_powerstate` directly during suspend and resume. The PCI layer handles that automatically.
- The driver does not (usually) call `pci_save_state` and `pci_restore_state` directly. The PCI layer handles those automatically too, through `pci_cfg_save` and `pci_cfg_restore`.
- The driver *does* save and restore its own device-specific state: BAR-local register contents that the hardware may have lost, softc fields that track runtime configuration, interrupt mask values. The PCI layer does not know about these.

The boundary is where the PCI configuration space ends and the BAR-accessed registers begin. The PCI layer protects the first; the driver protects the second.

22.2.5 PCIe Link States and Active-State Power Management (ASPM)

At a layer below the device's D-state sits the PCIe link itself. A link between a root complex and an endpoint can be in one of several **L-states**, and transitions between L-states happen automatically when the traffic on the link is low enough.

L0 is the fully-on link state. Data flows normally. Latency is at its minimum. This is the state the link is in whenever the device is active.

L0s is a low-power state the link enters when it has been idle for a few microseconds. The transmitter turns off its output drivers on one side; the link is bidirectional, so the other side's L0s is independent. Recovery from L0s takes hundreds of nanoseconds. This is cheap power saving that the platform can do automatically when traffic is bursty.

L1 is a deeper low-power state the link enters after a longer idle period (tens of microseconds). Both sides turn off more of their physical-layer circuitry. Recovery takes microseconds. This is used during light-load periods when the latency penalty is acceptable.

L1.1 and **L1.2** are PCIe 3.0 and later refinements of L1 that add further power gating, allowing even lower idle current at the cost of slower wake-up.

L2 is a near-off link state used during D3cold and S3; the link is effectively turned off, and wake-up requires a full re-negotiation. The driver typically does not manage L2 directly; it is a side-effect of the device entering D3cold.

The mechanism that controls the transitions between L0 and L0s/L1 is called **Active-State Power Management (ASPM)**. ASPM is a per-link feature configured through the PCIe capability registers of both ends of the link. It can be enabled, disabled, or restricted to L0s-only by platform policy. On FreeBSD, ASPM is usually controlled by the firmware through ACPI (the `_OSC` method tells the OS which capabilities to manage); the kernel does not second-guess the firmware's policy unless explicitly told to.

For Chapter 22 and for most FreeBSD drivers, ASPM is a platform concern, not a driver concern. The driver does not configure ASPM; the platform does. The driver does not need to save or restore ASPM state around suspend; the PCI layer handles the PCIe capability registers as part of the automatic config-space save and restore. A driver that wants to disable ASPM for a specific device (for example, because the device has a known errata that makes L0s unsafe) can do so by reading and writing the PCIe Link Control register explicitly, but this is rare and specific.

The reader does not need to add ASPM code to the `myfirst` driver. It is enough to be aware that L-states exist, that they transition automatically based on traffic, that the driver's D-state and the link's L-state are related but distinct, and that the platform handles the ASPM configuration. If a future driver the reader works on has a datasheet that specifies ASPM errata, the reader will know where to look.

22.2.6 The Anatomy of a Suspend-Resume Cycle

Putting the pieces together, a full system suspend-resume cycle looks like this from the driver's point of view, tracing the `myfirst` driver through an S3 transition:

1. The user closes the laptop lid. The ACPI button driver (`acpi_lid`) notices the event and, based on the system's policy, triggers a sleep request to state S3.
2. The kernel starts the suspend sequence. Userland daemons are paused; the kernel freezes non-essential threads.
3. The kernel walks the device tree and calls `DEVICE_SUSPEND` on each device, in reverse child order. The PCI bus driver's `bus_suspend_child` calls the `myfirst` driver's `device_suspend` method.
4. The `myfirst` driver's `device_suspend` runs. It masks interrupts on the device, stops accepting new DMA requests, waits for any in-flight DMA to complete, drains its task queue, logs the transition, and returns 0 to indicate success.
5. The PCI layer notes that `myfirst`'s suspend succeeded. It calls `pci_cfg_save` to cache the PCI configuration space. If `hw.pci.do_power_suspend` is 1 (the default), it transitions the device to D3hot via `pci_set_powerstate(dev, PCI_POWERSTATE_D3)`.
6. Higher in the tree, the PCI bus itself, the host bridge, and eventually the platform go through their own `DEVICE_SUSPEND` calls. ACPI arms its wake events. The CPU enters the low-power state corresponding to S3. The memory subsystem enters self-refresh. The PCIe link goes to L2 or thereabouts.
7. Time passes. On the scale the driver observes, no time passes at all; the kernel is not running.
8. The user opens the lid. The platform's wake circuitry wakes the CPU. ACPI performs the early resume steps: the CPU context is restored, memory is refreshed, the platform's firmware reinitialises what it must.
9. The kernel's resume sequence starts. It walks the device tree in forward order, calling `DEVICE_RESUME` on each device.
10. For `myfirst`, the PCI bus driver's `bus_resume_child` transitions the device back to D0 via `pci_set_powerstate(dev, PCI_POWERSTATE_D0)`. It calls `pci_cfg_restore` to write the cached

configuration space back into the device. It clears any pending PME# signal with `pci_clear_pme`. Then it calls the driver's `device_resume` method.

11. The driver's `device_resume` runs. The device is in D0, its configuration space is restored, its BAR registers are zero or default values. The driver re-enables bus-mastering if needed, writes its device-specific registers back from saved state, re-arms the interrupt mask, and marks the device as resumed. It returns 0.
12. The kernel's resume sequence continues up the tree. Userland threads are unfrozen. The user sees a working system, usually in one to three seconds.

Every step at which the driver has work is step 4 and step 11. Everything else is either platform or generic kernel machinery. The driver's job is to make those two steps correct, and to understand enough about the steps around them to interpret the behavior it observes.

22.2.7 Wake Sources

A device that is suspended can be the reason the system wakes. The way that happens depends on the bus:

- On **PCIe**, a device in D3hot can assert the **PME#** signal (Power Management Event). The platform's root complex translates PME# into a wake event, the ACPI `_PRW` method identifies which GPE (General-Purpose Event) it uses, and the ACPI subsystem translates the GPE into a wake from S3. On FreeBSD, the `pci_enable_pme(dev)` function enables the device's PME# output; `pci_clear_pme(dev)` clears any pending signal. The `pci_has_pm(dev)` helper tells whether the device has a power-management capability at all.
- On **USB**, a device can request **remote wakeup** through its standard USB descriptor. The host controller (`xhci`, `ohci`, `uhci`) translates the wake into a PME# or equivalent signal upstream. The driver does not usually handle this directly; the USB stack does.
- On **embedded platforms**, a device can assert a **GPIO** pin that the platform wires to its wake logic. The device-tree `interrupt-extended` or `wakeup-source` properties identify which pins are wake sources. FreeBSD's GPIO intr framework handles this.
- On **Wake-on-LAN**, a network controller watches for magic packets or pattern matches while suspended and asserts PME# when one is seen. Both the driver and the platform have to be configured; `re_setwol` in `if_re.c` is a good example of the driver side.

For the `myfirst` driver, the simulated device does not really have a wake source (it has no physical existence outside the simulation). The chapter explains the mechanism at the appropriate point in Section 4, shows what `pci_enable_pme` does and where it would be called, and leaves the actual wake-triggering to the sim backend's manual triggers. A real-hardware driver would call `pci_enable_pme` in its suspend path when wake-on-X is requested, and `pci_clear_pme` in its resume path to acknowledge any pending signal.

22.2.8 Real-World Examples: Wi-Fi, NVMe, USB

It helps to anchor the ideas in devices the reader has actually used. Consider three.

A **Wi-Fi adapter** like those handled by `iwlwifi` on Linux or `iwn` on FreeBSD is a constant power-management citizen. In S0, it spends most of its time in a low-power idle state on the chip itself, associated with the access point but not actively exchanging packets; when a packet is seen, it wakes to D0 for a few milliseconds, exchanges the packet, and goes back to idle. On system suspend (S3), the kernel asks the driver to save its state,

the driver tells the chip to disassociate cleanly (or to set up WoWLAN patterns if the user wants wake-on-wireless), and the PCI layer transitions the chip to D3. On resume, the reverse happens: the chip comes back to D0, the driver restores its state, and re-associates to the access point. The user perceives a one- or two-second delay after lid-open before Wi-Fi is back, which is almost entirely the re-association time and not the driver's resume time.

An **NVMe SSD** handles power states internally through its own power-state machinery (defined in the NVMe spec as PSx states, where PS0 is full power and higher numbers are lower power). The NVMe driver participates in system suspend by flushing its queues, waiting for in-flight commands to complete, and then telling the controller to enter a low-power state. On resume, the driver restores the queue configuration, tells the controller to come back to PS0, and the system resumes disk I/O. Because NVMe queues are large and DMA-heavy, the NVMe suspend path is a classic place where a missed `bus_dmamap_sync` or a missed queue drain shows up as filesystem corruption after resume.

A **USB device** is handled by the USB host controller driver (`xhci`, typically). The host controller driver is the one that implements `DEVICE_SUSPEND` and `DEVICE_RESUME`; individual USB drivers (for keyboards, storage, audio, etc.) are notified through the USB framework's own suspend and resume mechanism. A driver for a USB device rarely needs its own `DEVICE_SUSPEND` method; the USB framework handles the translation.

The `myfirst` driver in Chapter 22 uses a PCI endpoint model, which is the most common case and the one whose contract the other cases specialise. Learning the PCI pattern first gives the reader what is needed to understand the Wi-Fi pattern, the NVMe pattern, and the USB pattern when those drivers are looked at later.

22.2.9 What the Reader Has Gained

Section 1 is conceptual. The reader should not feel obligated to remember every detail of every state mentioned. What the reader should take from it is:

- Power management is a layered system. ACPI defines the system state. PCI defines the device state. PCIe defines the link state. The driver sees its device state most directly.
- Each layer's state can transition, and the transitions compose. A system S3 implies every device's D3 implies every link's L2. A per-device D3 (from runtime PM) does not imply system S3; the system stays in S0.
- The driver has a specific contract with the PCI and ACPI layers. The driver is responsible for quiescing its device's activity on suspend and restoring its device's state on resume. The PCI layer automatically handles configuration-space save, D-state transitions, and PME# wake signalling. ACPI handles the system-wide wake up.
- Wake sources exist and are plumbed through a specific chain (PME#, remote wakeup, GPIO). The driver usually enables and disables them through a helper API; it does not talk to the wake hardware directly.
- Testing is layered too. `devctl suspend/devctl resume` exercises only the driver methods. `acpicnf -s 3` exercises the whole system. A good regression script uses both.

With that picture in place, Section 2 can introduce the FreeBSD-specific APIs the driver uses to join this system.

22.2.10 Wrapping Up Section 1

Section 1 established why a driver has to care about power, what system-level and device-level power management mean, what the ACPI S-states and the PCI D-states look like, what the PCIe L-states add, how a suspend-resume cycle flows from the driver's point of view, and what wake sources are. It did not show any driver code; that is Section 2's job. What the reader now has is a vocabulary and a mental model: suspend is a transition the platform announces, the driver quiesces; the PCI layer moves the device to D3; the system sleeps; on wake, the PCI layer moves the device back to D0; the driver restores.

With that picture in mind, the next section introduces FreeBSD's concrete API for all of this: the four kobj methods (`DEVICE_SUSPEND`, `DEVICE_RESUME`, `DEVICE_SHUTDOWN`, `DEVICE_QUIESCE`), how the kernel invokes them, how they compose with `bus_generic_suspend` and `pci_suspend_child`, and how the `myfirst` driver's method table grows to include them.

22.3 Section 2: FreeBSD's Power Management Interface

Section 1 described the layered world of ACPI, PCI, PCIe, and wake sources. Section 2 narrows the view to the FreeBSD kernel's interface: the specific kobj methods the driver implements, the way the kernel dispatches them, and the generic helpers that make the whole scheme tractable. By the end of this section the `myfirst` driver has a skeleton power-management implementation that compiles, logs transitions, and can be exercised with `devctl suspend` and `devctl resume`. The skeleton does not yet quiesce DMA or restore state; that is Section 3's and Section 4's work. Section 2's job is to get the kernel to call the driver's methods so the rest of the chapter has something concrete to build on.

22.3.1 The Four Kobj Methods

The FreeBSD device framework treats a driver as an implementation of a kobj interface defined in `/usr/src/sys/kern/device_if.m`. That file is a small domain-specific language (`make -V` rules turn it into a header of function pointers and wrappers), and it defines the set of methods every driver can implement. The Chapter 16 to 21 work has populated the common methods: `DEVICE_PROBE`, `DEVICE_ATTACH`, `DEVICE_DETACH`. Power management adds four more, all documented in the same file with comments the reader can read directly:

1. **DEVICE_SUSPEND** is called when the kernel has decided to put the device into a suspended state. The method runs with the device still in D0 and with the driver still responsible for it. The method's job is to stop activity and, if necessary, save any state that will not be restored automatically. Returning 0 indicates success. Returning non-zero vetoes the suspend.
2. **DEVICE_RESUME** is called after the device has been returned to D0 on the way back from suspend. The method's job is to restore any state the hardware lost and to resume activity. Returning 0 indicates success. Returning non-zero causes the kernel to log a complaint; the resume cannot be meaningfully vetoed at this stage because the rest of the system has already come back.
3. **DEVICE_SHUTDOWN** is called during system shutdown to let the driver leave the device in a safe state for reboot or power-off. Many drivers implement this by calling their suspend method, because the two tasks are similar (stop the device cleanly). Returning 0 indicates success.

4. **DEVICE_QUIESCE** is called when the framework wants the driver to stop accepting new work but has not yet decided to detach. It is a softer form of detach: the device is still attached, the resources are still allocated, but the driver should refuse new submissions and let in-flight work drain. This method is optional and less commonly implemented than the other three; `device_quiesce` is called automatically before `device_detach` by the devctl layer, so a driver that implements both suspend and quiesce often shares code between them.

The file also contains default no-op implementations: `null_suspend`, `null_resume`, `null_shutdown`, `null_quiesce`. A driver that does not implement one of the methods gets the no-op, which returns 0 and does nothing. That is why Chapter 16 through Chapter 21 did not explicitly mention these methods: the no-ops were quietly being used, and for a driver whose device stays powered up forever and whose detach happens only at module unload, the no-ops give correct behavior for most workloads.

Chapter 22's first step is to replace those no-ops with real implementations.

22.3.2 Adding the Methods to the Driver's Method Table

The `device_method_t` array in a FreeBSD driver lists the kobj methods the driver implements. The `myfirst` driver's current method array (in `myfirst_pci.c`) looks something like this:

```
static device_method_t myfirst_pci_methods[] = {
    DEVMETHOD(device_probe,     myfirst_pci_probe),
    DEVMETHOD(device_attach,    myfirst_pci_attach),
    DEVMETHOD(device_detach,    myfirst_pci_detach),

    DEVMETHOD_END

};
```

Adding power management is mechanically simple: the driver adds three (or four) `DEVMETHOD` lines. The names on the left are the kobj method names from `device_if.m`; the names on the right are the driver's implementations. A complete set looks like this:

```
static device_method_t myfirst_pci_methods[] = {
    DEVMETHOD(device_probe,     myfirst_pci_probe),
    DEVMETHOD(device_attach,    myfirst_pci_attach),
    DEVMETHOD(device_detach,    myfirst_pci_detach),
    DEVMETHOD(device_suspend,   myfirst_pci_suspend),
    DEVMETHOD(device_resume,    myfirst_pci_resume),
    DEVMETHOD(device_shutdown,  myfirst_pci_shutdown),

    DEVMETHOD_END

};
```

The `myfirst_pci_suspend`, `myfirst_pci_resume`, and `myfirst_pci_shutdown` functions are new; they do not exist yet. The rest of Section 2 shows what each of them does at the skeleton level.

22.3.3 Prototypes and Return Values

Each of the four methods has the same signature: an `int` return value and one `device_t` argument. The `device_t` is the device the method is being called on, and the driver can use `device_get_softc(dev)` to recover the `softc` pointer.

```
static int myfirst_pci_suspend(device_t dev);
static int myfirst_pci_resume(device_t dev);
static int myfirst_pci_shutdown(device_t dev);
static int myfirst_pci_quiesce(device_t dev); /* optional */
```

Return values follow the usual FreeBSD convention. Zero means success. Non-zero is a regular `errno` value that indicates what went wrong: `EBUSY` if the driver cannot suspend because the device is busy, `EIO` if the hardware reported an error, `EINVAL` if the driver was called in an impossible state. The kernel's reaction differs per method.

For `DEVICE_SUSPEND`, a non-zero return **veto**es the suspend. The kernel aborts the suspend sequence and calls `DEVICE_RESUME` on the drivers that had already suspended successfully, unwinding the partial suspend. This is the mechanism that prevents a system from going to S3 while a critical device is in the middle of something the driver cannot interrupt. It should be used sparingly; returning `EBUSY` from every suspend whenever something is happening is a sure way to make suspend unreliable. A good driver only vetoes when the device is in a state that truly cannot be suspended.

For `DEVICE_RESUME`, a non-zero return is logged but mostly ignored. By the time resume is running, the system is coming back whether the driver likes it or not. The driver should log the error, mark its device as broken so that subsequent I/O fails cleanly, and return. A veto at resume time is too late to be useful.

For `DEVICE_SHUTDOWN`, a non-zero return is likewise mostly informational. The system is shutting down; the driver should try its best to leave the device in a safe state, but a failed shutdown is not an emergency.

For `DEVICE_QUIESCE`, a non-zero return prevents the subsequent operation (usually `detach`) from proceeding. A driver that returns `EBUSY` from `DEVICE_QUIESCE` forces the user to wait or to use `devctl detach -f` to force the detach.

22.3.4 The Order of Event Delivery

The kernel does not call `DEVICE_SUSPEND` on every driver at once. It walks the device tree and calls the method in a specific order, usually **reverse child order** on suspend and **forward child order** on resume. This is because a suspend is safest when each device is suspended *after* the devices that depend on it, and each device is resumed *before* the devices that depend on it.

Consider a simplified tree:

```
nexus0
  acpi0
    pci0
      pcib0
```

```

    myfirst0
pcib1
    em0
    xhci0
    umass0

```

On an S3 suspend, the traversal for the subtree under `pci0` suspends `myfirst0` before `pcib0`, `em0` before `pcib1`, and `umass0` before `xhci0`. Then `pcib0`, `pcib1`, and `xhci0` are suspended. Then `pci0`. Then `acpi0`. Then `nexus0`. Each parent is suspended only after all its children are suspended.

On resume, the order is reversed. `nexus0` resumes first, then `acpi0`, then `pci0`, then `pcib0`, `pcib1`, `xhci0`. Each of those calls `pci_resume_child` on its children, which transitions the child back to D0 before calling the child driver's `DEVICE_RESUME`. So `myfirst0`'s `device_resume` runs with `pcib0` already active and `pci0` already reconfigured.

The practical consequence for the driver is that during `DEVICE_SUSPEND` it can still access the device normally (the parent bus is still up), and during `DEVICE_RESUME` it can also access the device normally (the parent bus has been resumed first). The driver does not need to handle the edge case of a suspended parent.

There is one subtlety: if a parent bus reports that its children must be suspended in a specific order (ACPI can do this to express implicit dependencies), the generic helper `bus_generic_suspend` respects that order. The `myfirst` driver, whose parent is a PCI bus, does not need to worry about order beyond “children before parent”; the PCI bus has no strong ordering between its child devices.

22.3.5 `bus_generic_suspend`, `bus_generic_resume`, and the PCI Bus

A **bus driver** is itself a device driver, and when the kernel calls `DEVICE_SUSPEND` on a bus, the bus usually has to suspend all its children before it can itself go quiet. Implementing that by hand would be repetitive, so the kernel provides two helpers in `/usr/src/sys/kern/subr_bus.c`:

```

int bus_generic_suspend(device_t dev);
int bus_generic_resume(device_t dev);

```

The first iterates over the bus's children in reverse order and calls `BUS_SUSPEND_CHILD` on each. The second iterates forward and calls `BUS_RESUME_CHILD`. If any child's suspend fails, `bus_generic_suspend` unwinds by resuming the children that had already suspended.

A typical bus driver uses these helpers directly:

```

static device_method_t mybus_methods[] = {
    /* ... */
    DEVMETHOD(device_suspend, bus_generic_suspend),
    DEVMETHOD(device_resume, bus_generic_resume),
    DEVMETHOD_END
};

```

The `virtio_pci_modern` bus driver does exactly this in `/usr/src/sys/dev/virtio/pci/virtio_pci_modern.c`, where `vtpci_modern_suspend` and `vtpci_modern_resume` each just call `bus_generic_suspend(dev)` and `bus_generic_resume(dev)`.

The **PCI bus itself** does something more sophisticated: its `bus_suspend_child` is `pci_suspend_child`, and its `bus_resume_child` is `pci_resume_child`. These helpers (in `/usr/src/sys/dev/pci/pci.c`) do exactly what Section 1 described: on suspend they call `pci_cfg_save` to cache the configuration space, then call the driver's `DEVICE_SUSPEND`, and then if `hw.pci.do_power_suspend` is true they call `pci_set_powerstate(child, PCI_POWERSTATE_D3)`. On resume they reverse the sequence: transition back to D0, restore configuration from cache, clear pending PME#, and call the driver's `DEVICE_RESUME`.

The `myfirst` driver, which attaches directly to a PCI device, does not implement bus methods itself; it is a leaf driver. Its power methods are the ones that matter for its own device's state. But the reader should be aware that the PCI bus has already done work on both sides of the driver's methods: on suspend, by the time the driver's `DEVICE_SUSPEND` runs, the PCI layer has already saved the configuration; on resume, by the time the driver's `DEVICE_RESUME` runs, the PCI layer has already restored the configuration and brought the device back to D0.

22.3.6 `pci_save_state` and `pci_restore_state`: When the Driver Calls Them

The automatic save and restore handled by `pci_cfg_save/pci_cfg_restore` cover the standard PCI configuration registers: the BAR assignments, the command register, the cache-line size, the interrupt line, the MSI/MSI-X state. For most drivers, this is enough, and the driver does not need to call `pci_save_state` or `pci_restore_state` explicitly.

There are, however, situations where a driver does want to save the configuration manually. The PCI API exposes two helper functions for this:

```
void pci_save_state(device_t dev);
void pci_restore_state(device_t dev);
```

`pci_save_state` is a wrapper around `pci_cfg_save` that caches the current configuration. `pci_restore_state` writes the cached configuration back; if the device is not in D0 when `pci_restore_state` is called, the helper transitions it to D0 before restoring.

A driver typically calls these in two scenarios:

1. **Before and after a manual `pci_set_powerstate` that the driver itself initiates**, for example in a runtime power-management helper. If the driver decides to put an idle device into D3 while the system is in S0, it calls `pci_save_state`, then `pci_set_powerstate(dev, PCI_POWERSTATE_D3)`. When it wakes the device back up, it calls `pci_set_powerstate(dev, PCI_POWERSTATE_D0)` followed by `pci_restore_state`.
2. **Inside `DEVICE_SUSPEND` and `DEVICE_RESUME`, when the automatic save/restore is disabled**. Some drivers set `hw.pci.do_power_suspend` to 0 for devices that misbehave in D3, and manage the power state themselves. In that case the driver is also responsible for saving and restoring configuration. This is an uncommon pattern.

Chapter 22's `myfirst` driver uses scenario 1 in Section 5 (runtime PM), where the driver chooses to park the device in D3 while the system stays in S0. For system suspend, the driver does not call these helpers directly; the PCI layer handles it.

22.3.7 The `pci_has_pm` Helper

Not every PCI device has the PCI power-management capability. Older devices and some special-purpose ones do not advertise the capability, which means the driver cannot rely on `pci_set_powerstate` or `pci_enable_pme` working. The kernel provides a helper to check:

```
bool pci_has_pm(device_t dev);
```

Returns true if the device exposes a power-management capability, false otherwise. Most modern PCIe devices return true. Drivers that want to be robust against unusual hardware guard their power-related calls:

```
if (pci_has_pm(sc->dev))
    pci_enable_pme(sc->dev);
```

The Realtek driver in `/usr/src/sys/dev/re/if_re.c` uses this pattern in its `re_setwol` and `re_clrwol` functions: if the device does not have a PM capability, the function returns early without trying to touch power management.

22.3.8 PME#: Enabling, Disabling, and Clearing

On a device that does have the PM capability, the driver can ask the hardware to raise PME# when a wake-relevant event occurs. The API is three short functions:

```
void pci_enable_pme(device_t dev);
void pci_clear_pme(device_t dev);
/* there is no explicit pci_disable_pme; pci_clear_pme both clears pending
 * events and disables the PME_En bit. */
```

`pci_enable_pme` sets the `PME_En` bit in the device's power-management status/control register, so that the next power-management event the device detects causes it to assert PME#. `pci_clear_pme` clears any pending PME status bit and clears `PME_En`.

A driver that wants to enable wake-on-LAN, for example, typically:

1. Configures the device's own wake logic (set up pattern filters, set the magic-packet flag, etc.).
2. Calls `pci_enable_pme(dev)` in the suspend path so the device can actually assert PME#.
3. In the resume path, calls `pci_clear_pme(dev)` to acknowledge the wake event.

If `pci_enable_pme` is not called, the device will not raise PME# even if its own wake logic fires. If `pci_clear_pme` is not called on resume, a stale PME status bit can cause spurious future wake events.

The `myfirst` driver does not implement wake-on-X (the simulated device has nothing to wake on), so these calls do not appear in the main driver code. Section 4 includes a short sketch showing where they would go in a real driver.

22.3.9 A First Skeleton: Stage 1

With all of the background in place, we can write the first version of `myfirst`'s suspend, resume, and shutdown methods. Stage 1 of the Chapter 22 driver does nothing of substance; it only logs and returns success. The point is to get the kernel to call the methods so the rest of the chapter can test progressively.

First, add prototypes near the top of `myfirst_pci.c`:

```
static int myfirst_pci_suspend(device_t dev);
static int myfirst_pci_resume(device_t dev);
static int myfirst_pci_shutdown(device_t dev);
```

Next, extend the method table:

```
static device_method_t myfirst_pci_methods[] = {
    DEVMETHOD(device_probe,     myfirst_pci_probe),
    DEVMETHOD(device_attach,    myfirst_pci_attach),
    DEVMETHOD(device_detach,    myfirst_pci_detach),
    DEVMETHOD(device_suspend,   myfirst_pci_suspend),
    DEVMETHOD(device_resume,    myfirst_pci_resume),
    DEVMETHOD(device_shutdown,  myfirst_pci_shutdown),

    DEVMETHOD_END

};
```

Then implement the three functions at the end of the file:

```
static int
myfirst_pci_suspend(device_t dev)
{
    struct myfirst_softc *sc = device_get_softc(dev);

    device_printf(dev, "suspend (stage 1 skeleton)\n");
    atomic_add_64(&sc->power_suspend_count, 1);
    return (0);
}

static int
myfirst_pci_resume(device_t dev)
{
    struct myfirst_softc *sc = device_get_softc(dev);

    device_printf(dev, "resume (stage 1 skeleton)\n");
    atomic_add_64(&sc->power_resume_count, 1);
    return (0);
}
```

```

}

static int
myfirst_pci_shutdown(device_t dev)
{
    struct myfirst_softc *sc = device_get_softc(dev);

    device_printf(dev, "shutdown (stage 1 skeleton)\n");
    atomic_add_64(&sc->power_shutdown_count, 1);
    return (0);
}

```

Add the counter fields to the softc in myfirst.h:

```

struct myfirst_softc {
    /* ... existing fields ... */

    uint64_t power_suspend_count;
    uint64_t power_resume_count;
    uint64_t power_shutdown_count;
};

```

Expose them through sysctls next to the Chapter 21 counters, in whatever function already adds the myfirst sysctl tree:

```

SYSCTL_ADD_U64(ctx, kids, OID_AUTO, "power_suspend_count",
    CTLFLAG_RD, &sc->power_suspend_count, 0,
    "Number of times DEVICE_SUSPEND has been called");
SYSCTL_ADD_U64(ctx, kids, OID_AUTO, "power_resume_count",
    CTLFLAG_RD, &sc->power_resume_count, 0,
    "Number of times DEVICE_RESUME has been called");
SYSCTL_ADD_U64(ctx, kids, OID_AUTO, "power_shutdown_count",
    CTLFLAG_RD, &sc->power_shutdown_count, 0,
    "Number of times DEVICE_SHUTDOWN has been called");

```

Bump the version string in the Makefile:

```
CFLAGS+= -DMYFIRST_VERSION_STRING=\"1.5-power-stage1\"
```

Build, load, and test:

```
cd /path/to/driver
make clean && make
sudo kldload ./myfirst.ko
sudo devctl suspend myfirst0
sudo devctl resume myfirst0
sysctl dev.myfirst.0.power_suspend_count
sysctl dev.myfirst.0.power_resume_count
dmesg | tail -6
```

The expected output in `dmesg` is:

```
myfirst0: suspend (stage 1 skeleton)
myfirst0: resume (stage 1 skeleton)
```

And the counters should each read 1. If they do not, one of three things is wrong: the driver did not rebuild, the method table did not include the new entries, or `devctl` reported an error because the kernel cannot find the device by that name.

22.3.10 What the Skeleton Proves

Stage 1 looks trivial, but it proves three things that matter for the rest of the chapter:

1. **The kernel is delivering the methods.** If the counters increment, the `kobj` dispatch from `devctl` through the PCI bus to the `myfirst` driver is wired correctly. Every later stage builds on this, and it is easier to catch wiring bugs now than after adding real quiesce code.
2. **The method table is sourced correctly.** The `DEVMETHOD` lines with the method name and the driver's function pointer were typed correctly, the header includes are right, and the `DEVMETHOD_END` terminator is in place. A mistake here produces a kernel panic at load time, not a subtle runtime failure.
3. **The driver counts transitions.** The counters will be useful throughout the chapter as a cheap invariant check. `power_suspend_count` should always equal `power_resume_count` once the system is idle; any drift indicates a bug in one of the two methods.

With the skeleton in place, Section 3 can turn the suspend method from a log-only call into a real quiesce of the device's activity.

22.3.11 A Note on Detach, Quiesce, and Suspend

The reader may wonder how `DEVICE_DETACH`, `DEVICE_QUIESCE`, and `DEVICE_SUSPEND` relate. They look similar; each asks the driver to stop doing something. Here is the practical distinction, as the kernel enforces it:

- **DEVICE_QUIESCE** is the softest. It asks the driver to stop accepting new work and to drain in-flight work, but the device stays attached, its resources stay allocated, and another request can reactivate it. The kernel calls this before `DEVICE_DETACH` to give the driver a chance to refuse if the device is busy.

- **DEVICE_SUSPEND** is in the middle. It asks the driver to stop activity but leaves the resources allocated, because the driver will need them again on resume. The device's state is preserved (partly by the kernel through PCI config save, partly by the driver through its own saved state).
- **DEVICE_DETACH** is the hardest. It asks the driver to stop activity, release all resources, and forget the device. The only way to come back is through a fresh attach.

Many drivers implement `DEVICE_QUIESCE` by reusing parts of the suspend path (stop interrupts, stop DMA, drain queues) and `DEVICE_SHUTDOWN` by calling the suspend method directly. `/usr/src/sys/dev/xl/if_xl.c` does exactly that: `xl_shutdown(dev)` just calls `xl_suspend(dev)`. The relationship is:

- `shutdown` $\approx$ `suspend` (for most drivers that do not distinguish shutdown-specific behavior like wake-on-LAN defaults differently)
- `quiesce` $\approx$ half of `suspend` (stop activity, do not save state)
- `suspend` = `quiesce` + save state
- `resume` = restore state + un-quiesce
- `detach` = `quiesce` + free resources

Chapter 22 implements suspend, resume, and shutdown in full. It does not implement `DEVICE_QUIESCE` for the `myfirst` driver, because the Chapter 21 detach path already quiesces correctly, and `device_quiesce` would be redundant. A driver that wanted to allow a “stop I/O but keep the device attached” state (for example, to support `devctl detach -f gracefully`) would add `DEVICE_QUIESCE` as a separate method. The chapter mentions this for completeness and moves on.

22.3.12 The Detach Path as Inspiration

The Chapter 21 detach path is a useful reference because it already does most of what suspend needs to do. The detach path masks interrupts, drains the rx task and the simulation callout, waits for any in-flight DMA to complete, and calls `myfirst_dma_tear_down`. The suspend path will do the first three (mask, drain, wait) and skip the last (tear_down). A well-structured driver factors those common steps into shared helpers so that both paths use the same code.

Section 3 introduces exactly those helpers: `myfirst_stop_io`, `myfirst_drain_workers`, `myfirst_mask_interrupts`. Each is a small function extracted from the Chapter 21 detach path. Suspend uses them without tearing down resources; detach uses them and then tears down. The reuse keeps the two paths correct by construction.

22.3.13 Observing the Skeleton with WITNESS

A reader who has built the debug kernel with `WITNESS` can now run the skeleton and watch for any lock-order warnings. There should be none, because the skeleton does not acquire any locks. Section 3 will add lock acquisition in the suspend path, and `WITNESS` will immediately notice if the order disagrees with the order used in detach. That is one of the benefits of working in stages: the baseline is quiet, so any later warnings are clearly attributable to the stage that introduced them.

22.3.14 Wrapping Up Section 2

Section 2 established the FreeBSD-specific API for power management: the four kobj methods (`DEVICE_SUSPEND`, `DEVICE_RESUME`, `DEVICE_SHUTDOWN`, `DEVICE_QUIESCE`), their return-value semantics, the order the kernel delivers them, the generic helpers (`bus_generic_suspend`, `bus_generic_resume`) that walk the device tree, and the PCI-specific helpers (`pci_suspend_child`, `pci_resume_child`) that automatically save and restore configuration space around the driver's methods. The Stage 1 skeleton gave the driver log-only implementations of the three main methods, added counters to the softc, and verified that `devctl suspend` and `devctl resume` invoke them correctly.

What the skeleton does not do is interact with the device at all. The interrupt handlers keep firing; DMA transfers can still be in flight when suspend returns; the device does not go quiet. Section 3 fixes all of that: it introduces the quiesce discipline, factors the common stop-I/O helpers out of the Chapter 21 detach path, and turns the Stage 1 skeleton into a Stage 2 driver that actually quiesces the device before reporting success.

22.4 Section 3: Quiescing a Device Safely

Section 2 gave the driver skeleton suspend, resume, and shutdown methods that only log. Section 3 turns the suspend skeleton into a real suspend: one that stops interrupts, stops DMA, drains deferred work, and leaves the device in a defined quiet state before returning. Doing that correctly is the hardest single piece of power management, and it is where the Chapter 21 primitives pay off. If you have a clean DMA teardown path, a clean taskqueue drain, and a clean callout drain, you have most of what you need; quiesce is the art of applying them in the right order without tearing down the resources you will need back on resume.

22.4.1 What Quiesce Really Means

The word “quiesce” appears in several places in FreeBSD (`DEVICE_QUIESCE`, `device_quiesce`, `pcie_wait_for_pending`) and it means something specific: **to bring a device to a state where no activity is ongoing, no activity can start, and the hardware is not about to raise any more interrupts or do any more DMA**. The device is still fully attached, still has all its resources, still has its interrupt handlers registered, but it is not doing anything and is not going to do anything until something tells it to start again.

Quiescing is different from detach because detach unwinds the resource allocations. Quiescing is also different from a simple “set a flag that says the device is busy, so that future requests block”, because that flag only protects against new work entering the driver; it does not stop the hardware itself or the kernel-side infrastructure (tasks, callouts) from doing anything.

A quiesced device, in Chapter 22's sense, has these properties:

1. The device is not asserting, and cannot be provoked into asserting, an interrupt. Any interrupt mask the device has is set to suppress all sources. The interrupt handler, if it is called, has nothing to do.
2. The device is not performing DMA. Any in-flight DMA transfer has either completed or been aborted. The engine's control register is either in an idle state or has been explicitly reset.
3. The driver's deferred work has drained. Any task queued to the taskqueue has been executed or explicitly waited for. Any callout has been drained and will not fire.

4. The driver's softc fields reflect the quiesced state. The `dma_in_flight` flag is false. Any in-flight counter the driver keeps is zero. The suspended flag is true, so any new request the user space or another driver might submit gets an error.

Only when all four properties are true is the device actually quiet. A driver that masks interrupts but forgets to drain the taskqueue still has a task running in the background. A driver that drains the taskqueue but forgets the callout still has a timer that can fire. A driver that drains both but forgets to stop DMA can have a transfer commit bytes to memory after the CPU stopped looking. Each omission produces its own characteristic failure, and the cheapest way to avoid them all is to have one function that does the whole discipline in a known order.

22.4.2 The Order Matters

The four steps above are not independent. They have a dependency order that the driver has to respect, because the kernel-side infrastructure and the device interact. Consider what happens if the order is wrong.

If DMA is stopped before interrupts are masked, an interrupt can arrive between the DMA stop and the mask. The filter runs, sees a stale status bit, schedules a task. The task runs, expects a DMA buffer to be populated, finds stale data, and may corrupt the driver's internal state. Better to mask interrupts first so no new interrupts arrive while the stop is in progress.

If the taskqueue is drained before the device stops producing interrupts, a new interrupt can fire after the drain returns and schedule a task on a queue that was just drained. The task runs later, out of sync with the suspend sequence. Better to stop interrupts first so no new tasks are scheduled.

If the callout is drained before DMA is stopped, a simulated engine driven by a callout might see its callout torn down while a transfer is still in progress. The transfer never completes; `dma_in_flight` stays true; the driver hangs waiting for a completion that cannot come. Better to stop DMA first, wait for completion, then drain the callout.

The safe order, used by Chapter 21's detach path and adapted for Chapter 22's suspend, is:

1. Mark the driver as suspended (set a softc flag so new requests bounce).
2. Mask all interrupts at the device (write the interrupt-mask register).
3. Stop DMA: if a transfer is in flight, abort it and wait for it to reach a terminal state.
4. Drain the taskqueue (any task already running gets to finish; no new tasks start).
5. Drain any callouts (any fire in flight gets to finish; no new firings happen).
6. Verify the invariant (`dma_in_flight == false`, `softc->busy == 0`, etc.).

Each step builds on the previous one. By step 6, the device is quiet.

22.4.3 Helpers, Not Inline Code

A naive implementation would put all six steps inline in `myfirst_pci_suspend`. That works, but it duplicates code the detach path already has, and it makes both paths harder to maintain. The chapter's preferred pattern is to factor the steps into small helper functions that both paths call.

Three helpers are enough to cover the whole discipline:

```
static void myfirst_mask_interrupts(struct myfirst_softc *sc);
static int myfirst_drain_dma(struct myfirst_softc *sc);
static void myfirst_drain_workers(struct myfirst_softc *sc);
```

Each has one job:

- `myfirst_mask_interrupts` writes the device's interrupt-mask register to disable every vector the driver cares about. After it returns, no interrupt can arrive from this device.
- `myfirst_drain_dma` asks any in-flight DMA transfer to stop (sets the ABORT bit) and waits until `dma_in_flight` is false. It returns 0 on success, a non-zero `errno` if the device did not stop within the timeout.
- `myfirst_drain_workers` calls `taskqueue_drain` on the driver's taskqueue and `callout_drain` on the simulation's callout. After it returns, no deferred work is pending.

The suspend path calls all three in order. The detach path also calls all three, plus `myfirst_dma_tearardown` and the resource-release calls. The two paths share the quiesce steps and differ only at the end.

Here is the quiesce entry point the suspend path uses:

```
static int
myfirst_quiesce(struct myfirst_softc *sc)
{
    int err;

    MYFIRST_LOCK(sc);
    if (sc->suspended) {
        MYFIRST_UNLOCK(sc);
        return (0); /* already quiet, nothing to do */
    }
    sc->suspended = true;
    MYFIRST_UNLOCK(sc);

    myfirst_mask_interrupts(sc);

    err = myfirst_drain_dma(sc);
    if (err != 0) {
        device_printf(sc->dev,
            "quiesce: DMA did not stop cleanly (err %d)\n", err);
        /* Do not fail the quiesce; we still have to drain workers. */
    }

    myfirst_drain_workers(sc);

    return (err);
}
```

Note the design choice: `myfirst_quiesce` does not unwind on a DMA drain failure. A DMA that will not stop is a hardware problem, and the driver cannot undo the mask or unmark the suspend flag in response. The driver logs the problem, reports the error to the caller, and continues draining workers so that the rest of the state is still consistent. The caller (`myfirst_pci_suspend`) decides what to do with the error.

22.4.4 Implementing `myfirst_mask_interrupts`

For the `myfirst` driver, masking interrupts means writing the device's interrupt-mask register. The Chapter 19 simulation backend already has an `INTR_MASK` register at a known offset; the driver writes all-ones to it to disable every source.

```
static void
myfirst_mask_interrupts(struct myfirst_softc *sc)
{
    MYFIRST_ASSERT_UNLOCKED(sc);

    /*
     * Disable all interrupt sources at the device. After this write,
     * the hardware will not assert any interrupt vector. Any already-
     * pending status bits remain, but the filter will not be called
     * to notice them.
     */
    CSR_WRITE_4(sc, MYFIRST_REG_INTR_MASK, 0xFFFFFFFF);

    /*
     * For real hardware: also clear any pending status bits so we
     * don't see a stale interrupt on resume.
     */
    CSR_WRITE_4(sc, MYFIRST_REG_INTR_STATUS, 0xFFFFFFFF);
}
```

The function does not hold the `softc` lock. It talks only to the device through `CSR_WRITE_4`, which is a wrapper around `bus_write_4` that does not require any particular lock discipline. The `MYFIRST_ASSERT_UNLOCKED` call is a WITNESS-enabled invariant that catches a caller who mistakenly held the lock when calling this function; it is cheap and useful.

The mask value `0xFFFFFFFF` assumes the simulation's `INTR_MASK` register uses 1-means-masked semantics. The Chapter 19 simulation uses that convention; a real driver should consult its device's datasheet. The register map for `myfirst` is documented in `INTERRUPTS.md`; the reader can double-check the convention there.

A subtlety: on some real devices, masking interrupts only prevents the device from asserting new interrupts; the currently-active one continues to assert until the driver acknowledges it through the status register. That is why the function also clears `INTR_STATUS`: to make sure no stale bit is left asserted that could fire again after resume. On the simulation, the status register behaves similarly, so the same write is correct.

22.4.5 Implementing myfirst_drain_dma

Draining DMA is the most delicate of the three helpers, because it has to wait for the device. The Chapter 21 driver tracks in-flight DMA with `dma_in_flight` and notifies completion through `dma_cv`. The suspend path reuses that exact machinery.

```
static int
myfirst_drain_dma(struct myfirst_softc *sc)
{
    int err = 0;

    MYFIRST_LOCK(sc);
    if (sc->dma_in_flight) {
        /*
         * Tell the engine to abort. The abort bit produces an
         * ERR status that the filter will translate into a task
         * that wakes us via cv_broadcast.
         */
        CSR_WRITE_4(sc, MYFIRST_REG_DMA_CTRL,
                    MYFIRST_DMA_CTRL_ABORT);

        /*
         * Wait up to one second for the abort to land. The
         * cv_timedwait drops the lock while we sleep.
         */
        err = cv_timedwait(&sc->dma_cv, &sc->mtx, hz);
        if (err == EWOULDBLOCK) {
            device_printf(sc->dev,
                          "drain_dma: timeout waiting for abort\n");
            /*
             * Force the state forward. The device is
             * beyond our reach at this point; treat the
             * transfer as failed.
             */
            sc->dma_in_flight = false;
        }
    }
    MYFIRST_UNLOCK(sc);

    return (err == EWOULDBLOCK ? ETIMEDOUT : 0);
}
```

The function asks the DMA engine to abort, then sleeps on the condition variable used by the Chapter 21 completion path. If the completion arrives, the filter acknowledges it and enqueues a task; the task calls `myfirst_dma_handle_complete`, which does the POSTREAD/POSTWRITE sync, clears `dma_in_flight`, and

broadcasts the CV. The suspend path's `cv_timedwait` returns, the drain function returns 0, and the suspend continues.

If the completion does not arrive within one second (a second is a generous timeout for a simulated device whose callout fires every few milliseconds), the function logs a warning and forces `dma_in_flight` to false. This is a defensive choice: a real device that does not honor abort within a second is misbehaving, and the driver has to move on. Leaving `dma_in_flight` true would deadlock the suspend. The cost of the defensive clear is that a very slow transfer could in principle complete after the suspend returned, writing into a buffer the driver no longer expects to be live. On the simulation, that cannot happen because the callout is drained in the next step. On real hardware, the risk is hardware-specific and a real driver would add device-specific recovery here.

The return value is 0 on a clean drain (including the timeout case, which has been force-cleared) and `ETIMEDOUT` if the caller needs to know a timeout happened. The suspend path logs the error but does not veto the suspend; by the time the drain has timed out, the device is effectively broken anyway.

22.4.6 Implementing `myfirst_drain_workers`

Draining deferred work is easier because it does not involve the device. The rx task and the simulation's callout each have well-known drain primitives.

```
static void
myfirst_drain_workers(struct myfirst_softc *sc)
{
    /*
     * Drain the per-vector rx task. Any task currently running is
     * allowed to finish; no new tasks will be scheduled because
     * interrupts are already masked.
     */
    if (sc->rx_vector.has_task)
        taskqueue_drain(taskqueue_thread, &sc->rx_vector.task);

    /*
     * Drain the simulation's DMA callout. Any fire in flight is
     * allowed to finish; no new firings will happen.
     *
     * This is a simulation-only call; a real-hardware driver would
     * omit it.
     */
    if (sc->sim != NULL)
        myfirst_sim_drain_dma_callout(sc->sim);
}
```

The function is safe to call with the lock released. `taskqueue_drain` is documented to do its own synchronisation; `callout_drain` (which `myfirst_sim_drain_dma_callout` wraps internally) is similarly safe.

An important property of both drain calls: they wait for running work to finish, but they do not cancel it in the middle. A task that is halfway through `myfirst_dma_handle_complete` will complete its work, including any

bus_dmamap_sync and counter update, before the drain returns. That is the behavior we want: suspend should not interrupt a task partway through, because the task's invariants must hold for the resume path to be correct.

22.4.7 Updating the Suspend Method

With the three helpers in place, the suspend method is short:

```
static int
myfirst_pci_suspend(device_t dev)
{
    struct myfirst_softc *sc = device_get_softc(dev);
    int err;

    device_printf(dev, "suspend: starting\n");

    err = myfirst_quiesce(sc);
    if (err != 0) {
        device_printf(dev,
            "suspend: quiesce returned %d; continuing anyway\n",
            err);
    }

    atomic_add_64(&sc->power_suspend_count, 1);
    device_printf(dev,
        "suspend: complete (dma in flight=%d, suspended=%d)\n",
        sc->dma_in_flight, sc->suspended);
    return (0);
}
```

The suspend method does not return the quiesce error to the kernel. That is a policy decision, and it deserves explanation.

Returning a non-zero value from `DEVICE_SUSPEND` vetoes the suspend, which has a large downstream effect: the kernel unwinds the partial suspend, reports a failure to the user, and leaves the system in S0. For Chapter 22's driver, a quiesce timeout does not justify that level of disruption. The device is still accessible; masking the interrupt and marking the suspended flag is enough to prevent any activity. A real driver might choose differently: a storage controller with an outstanding write might veto the suspend until the write completes, because losing that write across the transition would corrupt the filesystem. Each driver makes its own call.

The logging is verbose at this stage. Section 7 will introduce the ability to turn off the verbose logging (or crank it up for debugging) through a sysctl. For now, the extra detail helps when stepping through the suspend sequence for the first time.

22.4.8 Saving Runtime State the Hardware Loses

So far the suspend path has only stopped activity. It has not saved any state. For most devices that is correct: the PCI layer saves configuration space automatically, and the device's runtime registers (the ones the driver writes through BARs) are either restored from the driver's softc on resume or regenerated from the driver's software state. The `myfirst` simulation does not have any BAR-local state the driver cares about preserving; the simulation starts fresh on resume, and the driver writes whatever registers it needs at that point.

A real driver may have more. Consider the `re(4)` driver: its `re_setwol` function writes wake-on-LAN-related registers into the NIC's EEPROM-backed configuration space before suspend. Those values are device-private; the PCI layer does not know about them. If the driver did not write them in suspend, the NIC would not know it was supposed to wake on a magic packet, and wake-on-LAN would not work.

For Chapter 22, the only state the `myfirst` driver saves is the interrupt mask's pre-suspend value. The Stage 2 suspend writes `0xFFFFFFFF` to the mask register, but the Stage 2 resume needs to know what value was there before (which determines which vectors are enabled in normal operation). The driver stores that in a `softc` field:

```
struct myfirst_softc {
    /* ... */
    uint32_t saved_intr_mask;
};
```

And the mask helper saves it:

```
static void
myfirst_mask_interrupts(struct myfirst_softc *sc)
{
    sc->saved_intr_mask = CSR_READ_4(sc, MYFIRST_REG_INTR_MASK);

    CSR_WRITE_4(sc, MYFIRST_REG_INTR_MASK, 0xFFFFFFFF);
    CSR_WRITE_4(sc, MYFIRST_REG_INTR_STATUS, 0xFFFFFFFF);
}
```

The resume path will write `sc->saved_intr_mask` back to the mask register after the device has been reinitialised. This is a minimal example of state-save-and-restore; Section 4 elaborates when it shows the full resume flow.

22.4.9 The suspended Flag as a User-Facing Invariant

Setting `sc->suspended = true` during quiesce has a second purpose beyond suppressing new requests: it makes the state observable to user space. The driver can expose the flag through a `sysctl`:

```
SYCTL_ADD_BOOL(ctx, kids, OID_AUTO, "suspended",
    CTLFLAG_RD, &sc->suspended, 0,
    "Whether the driver is in the suspended state");
```

After `devctl suspend myfirst0`, the reader sees:

```
# sysctl dev.myfirst.0.suspended
dev.myfirst.0.suspended: 1
```

After `devctl resume myfirst0`, the value should go back to 0 (Section 4 wires the resume path to clear it). This is a quick way to check the driver's state without inferring it from other counters.

22.4.10 Handling the Case Where DMA Is Not in Flight

The `myfirst_drain_dma` helper handles the case where a transfer is actively running. It should also handle the far more common case where nothing is in flight at the moment of suspend, without doing anything unnecessary.

The pseudocode above does handle this: the `if (sc->dma_in_flight)` guard skips the abort and wait entirely when the flag is false. The function returns 0 immediately and the suspend proceeds.

That path is fast: on an idle device, `myfirst_drain_dma` is a lock acquire, a flag check, and a lock release. The cost of the quiesce is dominated by `taskqueue_drain` (which does a full round-trip through the taskqueue thread) and `callout_drain` (similar). A typical idle-device suspend takes a few hundred microseconds to a few milliseconds, dominated by the deferred-work drains, not by the device.

22.4.11 Testing Stage 2

With the quiesce code in place, the Stage 2 test is more interesting. The reader runs a transfer, then immediately suspends, and observes:

```
# Start with no activity.
sysctl dev.myfirst.0.dma_transfers_read
# 0

# Trigger a transfer. The transfer should complete quickly.
sudo sysctl dev.myfirst.0.dma_test_read=1
sysctl dev.myfirst.0.dma_transfers_read
# 1

# Now stress the path: start a transfer and immediately suspend.
sudo sysctl dev.myfirst.0.dma_test_read=1 &
sudo devctl suspend myfirst0

# Check the state.
sysctl dev.myfirst.0.suspended
# 1

sysctl dev.myfirst.0.power_suspend_count
```

```
# 1

sysctl dev.myfirst.0.dma_in_flight
# 0 (the transfer completed or was aborted)

dmesg | tail -8
```

The expected dmesg output shows the suspend log, the in-flight DMA being aborted, and the completion. If `dma_in_flight` is still 1 after the suspend returned, the abort did not take effect, and the reader should check the simulation's abort handling.

Then resume:

```
sudo devctl resume myfirst0

sysctl dev.myfirst.0.suspended
# 0 (after Section 4 is implemented)

sysctl dev.myfirst.0.power_resume_count
# 1

# Try another transfer to check the device is back.
sudo sysctl dev.myfirst.0.dma_test_read=1
dmesg | tail -4
```

The last transfer should succeed; if it does not, the suspend left the device in a state the resume did not recover. Section 4 teaches the resume path that makes this correct.

22.4.12 A Cautious Note on Locking

The quiesce code runs from the `DEVICE_SUSPEND` method, which is called by the kernel's power-management path. That path holds no driver lock when it calls the method; the driver is responsible for its own synchronisation. The helpers in this section follow a specific discipline:

- `myfirst_mask_interrupts` holds no lock. It only writes hardware registers, which are atomic on PCIe.
- `myfirst_drain_dma` takes the softc lock to read `dma_in_flight` and uses `cv_timedwait` to sleep while holding the lock (which is the correct use of a sleep-mutex CV).
- `myfirst_drain_workers` holds no lock. `taskqueue_drain` and `callout_drain` do their own synchronisation and must be called without the lock to avoid deadlock (the task being drained may try to acquire the lock itself).

The full quiesce sequence thus acquires and releases the lock multiple times: once briefly at the top of `myfirst_quiesce` to set `suspended`, once inside `myfirst_drain_dma` for the sleep, and never inside `myfirst_drain_workers`. That is intentional. Holding the lock across `taskqueue_drain` would deadlock, because the drained task acquires the same lock on entry.

A reader who runs this code under WITNESS will not see any lock-order warnings, because the lock is only held over the CV sleep and no other locks are acquired during that window. If later work adds more locks to the driver (for example, a per-vector lock), the quiesce code should remain careful about which locks are held around the drain calls.

22.4.13 Integrating with the Shutdown Method

The shutdown method shares almost all of its logic with suspend. A reasonable implementation is:

```
static int
myfirst_pci_shutdown(device_t dev)
{
    struct myfirst_softc *sc = device_get_softc(dev);

    device_printf(dev, "shutdown: starting\n");
    (void)myfirst_quiesce(sc);
    atomic_add_64(&sc->power_shutdown_count, 1);
    device_printf(dev, "shutdown: complete\n");
    return (0);
}
```

The only difference from suspend is the absence of state-save calls (shutdown is final; there is no resume to save state for) and the absence of a return-value check on the quiesce (shutdown cannot be vetoed meaningfully). Many real drivers follow the same pattern; `/usr/src/sys/dev/xl/if_xl.c`'s `xl_shutdown` just calls `xl_suspend`. The `myfirst` driver can use either style; the chapter prefers the slightly more explicit version above because it makes the intent clearer in the code.

22.4.14 Wrapping Up Section 3

Section 3 turned the Stage 1 skeleton into a Stage 2 driver that actually quiesces the device on suspend. It introduced the quiesce discipline (mark suspended, mask interrupts, stop DMA, drain workers, verify), factored the steps into three helper functions (`myfirst_mask_interrupts`, `myfirst_drain_dma`, `myfirst_drain_workers`), explained the order in which they must run and why, showed how to integrate them into the suspend and shutdown methods, and discussed the locking discipline.

What the Stage 2 driver does not yet do is come back correctly. The resume method is still the Stage 1 skeleton; it logs and returns 0 without restoring anything. If the suspended device had any state the hardware lost, that state is gone, and a subsequent transfer will fail. Section 4 fixes the resume path so that a full suspend-resume cycle leaves the device in the same state it was in before.

22.5 Section 4: Restoring State on Resume

Section 3 gave the driver a correct suspend path. Section 4 writes the matching resume. The resume is the complement of the suspend: every thing suspend stopped, resume restarts; every value suspend saved, resume

writes back; every flag suspend set, resume clears. The sequence is not an exact mirror (resume runs in a different kernel context, with the PCI layer having already done work, and the device in a different state than where suspend left it), but the contents correspond one-to-one. Doing the resume correctly is a matter of respecting the contract the PCI layer has already partly fulfilled, and filling in the rest.

22.5.1 What the PCI Layer Has Already Done

When the kernel's `DEVICE_RESUME` method is called on the driver, several things have already happened:

1. The CPU has come out of the S-state (resumed from S3 or S4 back to S0).
2. Memory has been refreshed and the kernel has re-established its own state.
3. The parent bus has been resumed. For `myfirst`, that means the PCI bus driver has already handled the host bridge and the PCIe root complex.
4. The PCI layer has called `pci_set_powerstate(dev, PCI_POWERSTATE_D0)` on the device, transitioning it from whatever low-power state it was in (typically D3hot) back to full power.
5. The PCI layer has called `pci_cfg_restore(dev, dinfo)`, which writes the cached configuration space values (BARs, command register, cache-line size, etc.) back into the device.
6. The PCI layer has called `pci_clear_pme(dev)` to clear any pending power-management event bits.
7. The MSI or MSI-X configuration, which is part of the cached state, has been restored. The driver's interrupt vectors are usable again.

At this point the PCI bus driver calls into `myfirst`'s `DEVICE_RESUME`. The device is in D0, with its BARs mapped, its MSI/MSI-X table restored, and its generic PCI state intact. What the driver has to restore is the device-specific state that the PCI layer did not know about: the BAR-local registers the driver wrote during or after attach.

For the `myfirst` simulation, the relevant BAR-local registers are the interrupt mask (which the suspend path deliberately set to all-masked) and the DMA registers (which may have been left in an aborted state). The driver needs to put them back to values that reflect normal operation.

22.5.2 The Resume Discipline

A correct resume path does four things, in order:

1. **Re-enable bus-mastering**, in case the configuration-space restore did not do so or the PCI layer's automatic restore was disabled. This is `pci_enable_busmaster(dev)`. On modern FreeBSD it is usually redundant but harmless; older code paths or buggy BIOSes sometimes leave bus-mastering disabled. Calling it defensively is cheap.
2. **Restore any device-specific state** the driver saved during suspend. For `myfirst`, that means writing `saved_intr_mask` back to the `INTR_MASK` register. A real driver would also restore things like vendor-specific configuration bits, DMA engine programming, hardware timers, etc.
3. **Unmask interrupts and clear the suspended flag**, so the device can resume activity. This is the pivot point: before it, the device is still quiet; after it, the device can raise interrupts and accept work.
4. **Log the transition and update counters**, for observability and regression testing.

Here is what the pattern looks like in code:


```

static int
myfirst_pci_resume(device_t dev)
{
    struct myfirst_softc *sc = device_get_softc(dev);
    int err;

    device_printf(dev, "resume: starting\n");

    err = myfirst_restore(sc);
    if (err != 0) {
        device_printf(dev,
            "resume: restore failed (err %d)\n", err);
        atomic_add_64(&sc->power_resume_errors, 1);
        /*
         * Continue anyway. By the time we're here, the system
         * is coming back whether we like it or not.
         */
    }

    atomic_add_64(&sc->power_resume_count, 1);
    device_printf(dev, "resume: complete\n");
    return (0);
}

```

The helper `myfirst_restore` does the three real steps:

```

static int
myfirst_restore(struct myfirst_softc *sc)
{
    /* Step 1: re-enable bus-master (defensive). */
    pci_enable_busmaster(sc->dev);

    /* Step 2: restore device-specific state.
     *
     * For myfirst, this is just the interrupt mask. A real driver
     * would restore more: DMA engine programming, hardware timers,
     * vendor-specific configuration, etc.
     */
    if (sc->saved_intr_mask == 0xFFFFFFFF) {
        /*
         * Suspend saved a fully-masked mask, which means the
         * driver had no idea what the mask should be. Use the
         * default: enable DMA completion, disable everything
         * else.
         */
    }
}

```

```

        */
        sc->saved_intr_mask = ~MYFIRST_INTR_COMPLETE;
    }
    CSR_WRITE_4(sc->dev, MYFIRST_REG_INTR_MASK, sc->saved_intr_mask);

    /* Step 3: clear the suspended flag and unmask the device. */
    MYFIRST_LOCK(sc);
    sc->suspended = false;
    MYFIRST_UNLOCK(sc);

    return (0);
}

```

The function returns 0 because no step above can fail in the `myfirst` simulation. A real driver would check the return values of its hardware initialisation calls and propagate any errors.

22.5.3 Why `pci_enable_busmaster` Matters

Bus-mastering is a bit in the PCI command register that controls whether the device can issue DMA transactions. Without it, the device cannot read or write host memory; any DMA trigger would be silently ignored by the PCI host bridge.

Chapter 18 enabled bus-mastering during attach. The PCI layer's automatic config-space restore writes the command register back to its saved value, which includes the bus-master bit. So in principle the driver does not need to call `pci_enable_busmaster` again on resume. In practice, several things can go wrong:

- The platform firmware may reset the command register as part of waking the device.
- The `hw.pci.do_power_suspend` sysctl may be 0, in which case the PCI layer does not save and restore the config space.
- A device-specific quirk might clear bus-mastering as a side effect of the D3-to-D0 transition.

Calling `pci_enable_busmaster` unconditionally defensively in resume is a low-cost safety net. Several production FreeBSD drivers follow this pattern; `if_re.c`'s resume path is one example. The call is idempotent: if bus-mastering is already on, the call just re-asserts it.

22.5.4 Restoring Device-Specific State

The `myfirst` simulation does not have much state the driver needs to restore manually. The BAR-local registers are:

- The interrupt mask (restored from `saved_intr_mask`).
- The interrupt status bits (were cleared in suspend; they should stay cleared until new activity arrives).
- The DMA engine registers (`DMA_ADDR_LOW`, `DMA_ADDR_HIGH`, `DMA_LEN`, `DMA_DIR`, `DMA_CTRL`, `DMA_STATUS`). These are transient: they hold the parameters of the current transfer. After resume, no transfer is in progress, so the values do not matter; the next transfer will overwrite them.

A real driver would have more. Consider a few examples:

- A storage driver might have a DMA descriptor ring whose base address the device learned during attach. After resume, the BAR-level register holding that base address may have been reset; the driver needs to reprogram it.
- A network driver might have filter tables (MAC addresses, multicast lists, VLAN tags) programmed into device registers. After resume, those tables may be empty; the driver rebuilds them from softc-side copies.
- A GPU driver might have register state for display timing, colour tables, hardware cursors. After resume, the driver restores the active mode.

For `myfirst`, the interrupt mask is the only BAR-local state that needs restoring. The pattern shown above is the template a real driver would adapt to its device.

22.5.5 Validating Device Identity After Resume

Some devices are reset completely across a suspend-to-D3-cold cycle. The device that comes back is functionally the same, but its entire state has been reinitialised as if it had just powered on. A driver that assumed nothing changed would silently get wrong behaviour.

A defensive resume path can detect this by reading a known register value and comparing to what it read at attach time. For a PCI device, the vendor ID and device ID in configuration space are always the same (the PCI layer restored them), but some device-private register (a revision ID, a self-test register, a firmware version) can be checked:

```
static int
myfirst_validate_device(struct myfirst_softc *sc)
{
    uint32_t magic;

    magic = CSR_READ_4(sc->dev, MYFIRST_REG_MAGIC);
    if (magic != MYFIRST_MAGIC_VALUE) {
        device_printf(sc->dev,
            "resume: device identity mismatch (got %#x, "
            "expected %#x)\n", magic, MYFIRST_MAGIC_VALUE);
        return (EIO);
    }
    return (0);
}
```

For the `myfirst` simulation, there is no magic register (the simulation was not built with post-resume validation in mind). A reader who wants to add one as a challenge can extend the simulation backend's register map with a read-only `MAGIC` register, and have the driver check it. The chapter's Lab 3 includes this as an option.

A real driver whose device truly does reset across D3cold needs this check, because without it a subtle failure can occur: the driver assumes the device's internal state machine is in state `IDLE`, but after the reset the state machine is actually in state `RESETTING`. Any command the driver sends is rejected, the driver interprets the

rejection as a hardware fault, and the device is marked broken. Catching the reset explicitly and rebuilding state is cleaner.

22.5.6 Detecting and Recovering from a Device Reset

If the validation finds a mismatch, the driver's recovery options depend on the hardware. For the `myfirst` simulation, the simplest response is to log, mark the device broken, and fail subsequent operations:

```
if (myfirst_validate_device(sc) != 0) {
    MYFIRST_LOCK(sc);
    sc->broken = true;
    MYFIRST_UNLOCK(sc);
    return (EIO);
}
```

The `softc` grows a broken flag, and any user-facing request checks the flag and fails with an error. The detach path still works (detach always succeeds, even on a broken device), so the user can unload the driver and reload it.

A real driver that detects a reset has more options. A network driver might re-run its attach sequence from the point after `pci_alloc_msi` (which has been restored by the PCI layer). A storage driver might re-initialise its controller using the same code path attach used. The implementation depends heavily on the device; the pattern is “detect, then do whatever attach-time initialisation is still required”.

The chapter's `myfirst` driver takes the simpler approach: it does not implement reset detection for the simulation, and the resume path does not include the validation call by default. The code above is provided as reference for a reader who wants to extend the driver as an exercise.

22.5.7 Restoring DMA State

The Chapter 21 DMA setup allocates a tag, allocates memory, loads the map, and retains the bus address in the `softc`. None of that is visible in the BAR-local register map; the DMA engine learns the bus address only when the driver writes it to `DMA_ADDR_LOW` and `DMA_ADDR_HIGH` as part of starting a transfer.

This means the DMA state does not need restoration in the sense of “write registers”. The tag, map, and memory are all kernel-side data structures; they survive suspend intact. The next transfer will program the DMA registers as part of its normal submission.

What might need restoration on a real device is:

- **The DMA descriptor ring base address**, if the device keeps a persistent pointer. A real NIC writes a base-address register once at attach and points the device at a ring of descriptors; after D3cold, that register may have been reset and the driver must reprogram it.
- **The DMA engine's enable bit**, if it is separate from individual transfers.
- **Any per-channel configuration** (burst size, priority, etc.) that is held in registers the PCI layer did not cache.

For `myfirst`, none of this applies. The DMA engine is programmed per transfer. Resume does not need any DMA-specific restoration beyond what the generic state restoration already covered.

22.5.8 Re-Arming Interrupts

Masking interrupts was step 2 of suspend. Unmasking them is step 3 of resume. The Stage 3 resume writes `saved_intr_mask` back to the `INTR_MASK` register, which (by convention) writes 0 to the bits corresponding to enabled vectors and 1 to the bits for disabled vectors. After the write, the device is ready to assert interrupts on the enabled vectors as soon as there is reason to.

There is a subtlety around ordering. The resume path unmask interrupts before it clears the suspended flag. That means a very unfortunate interrupt could arrive, call the filter, and find `suspended == true`. The filter would refuse to handle it and return `FILTER_STRAY`, which would leave the interrupt asserted.

To avoid that, the resume path takes the softc lock around the state change and does the unmask and the flag clear in the opposite order: clear suspended first, then unmask. That way any interrupt the device raises after the mask clears sees `suspended == false` and is handled normally.

The code in the previous snippet does this correctly: `myfirst_restore` writes the mask, then acquires the lock, clears the flag, and releases the lock. The order is important; reversing it creates a narrow window where interrupts could be lost.

22.5.9 Wake Source Cleanup

If the driver enabled a wake source during suspend (`pci_enable_pme`), the resume path should clear any pending wake event (`pci_clear_pme`). The PCI layer's `pci_resume_child` helper already calls `pci_clear_pme(child)` before the driver's `DEVICE_RESUME`, so the driver does not usually need to call it again.

The one case where the driver might want to call `pci_clear_pme` explicitly is in a runtime-PM context where the driver is resuming the device while the system stays in S0. In that case `pci_resume_child` was not involved, and the driver is responsible for clearing the PME status itself.

A hypothetical sketch for a driver with wake-on-X:

```
static int
myfirst_pci_resume(device_t dev)
{
    struct myfirst_softc *sc = device_get_softc(dev);

    if (pci_has_pm(dev))
        pci_clear_pme(dev); /* defensive; PCI layer already did this */

    /* ... rest of the resume path ... */
}
```

For `myfirst`, there is no wake source, so the call does nothing useful; the chapter omits it from the main code and mentions the pattern here for completeness.

22.5.10 Updating the Stage 3 Driver

Stage 3 brings together everything above into a single working resume. The diff against Stage 2 is:

- myfirst.h grows a saved_intr_mask field (added for Stage 2) and a broken flag.
- myfirst_pci.c gets a myfirst_restore helper and a rewritten myfirst_pci_resume.
- The Makefile version bumps to 1.5-power-stage3.

Build and test:

```
cd /path/to/driver
make clean && make
sudo kldunload myfirst      # unload any previous version
sudo kldload ./myfirst.ko

# Quiet baseline.
sysctl dev.myfirst.0.dma_transfers_read
# 0
sysctl dev.myfirst.0.suspended
# 0

# Full cycle.
sudo devctl suspend myfirst0
sysctl dev.myfirst.0.suspended
# 1

sudo devctl resume myfirst0
sysctl dev.myfirst.0.suspended
# 0

# A transfer after resume should work.
sudo sysctl dev.myfirst.0.dma_test_read=1
sysctl dev.myfirst.0.dma_transfers_read
# 1

# Do it several times to make sure the path is stable.
for i in 1 2 3 4 5; do
    sudo devctl suspend myfirst0
    sudo devctl resume myfirst0
    sudo sysctl dev.myfirst.0.dma_test_read=1
done
sysctl dev.myfirst.0.dma_transfers_read
# 6 (1 + 5)

sysctl dev.myfirst.0.power_suspend_count dev.myfirst.0.power_resume_count
# should be equal, around 6 each
```

If the counters drift (suspend count not equal to resume count) or if `dma_test_read` starts failing after a suspend, something in the restore path is not putting the device back into a usable state. The first debugging step is to read the `INTR_MASK` and compare against `saved_intr_mask`; the second is to trace the DMA engine's status register and see if it is reporting an error.

22.5.11 Interaction with the Chapter 20 MSI-X Setup

The `myfirst` driver from Chapter 20 uses MSI-X when available, with a three-vector layout (admin, rx, tx). The MSI-X configuration lives in the device's MSI-X capability registers and in a kernel-side table. The PCI layer's config-space save-and-restore covers the capability registers; the kernel-side state is not affected by the D-state transition.

This means the `myfirst` driver does not need to do anything special to restore its MSI-X vectors. The interrupt resources (`irq_res`) remain allocated, the cookies remain registered, the CPU bindings remain in place. When the device raises an MSI-X vector on resume, the kernel delivers it to the filter that was registered at attach time.

A reader who wants to verify this can write to one of the `simulate sysctls` after resume and observe that the corresponding per-vector counter increments:

```
sudo devctl suspend myfirst0
sudo devctl resume myfirst0
sudo sysctl dev.myfirst.0.intr_simulate_admin=1
sysctl dev.myfirst.0.vec0_fire_count
# should be incremented
```

If the counter does not increment, the MSI-X path has been disturbed. The most likely cause is a bug in the driver's own state management (the suspended flag was not cleared, or the filter is rejecting the interrupt for a different reason). The chapter's troubleshooting section has more detail.

22.5.12 Handling a Failed Resume Gracefully

If some step of the resume fails, the driver has limited options. It cannot veto the resume (the kernel has no unwind path at this point). It cannot usually retry (the hardware state is uncertain). The best it can do is:

1. Log the failure prominently with `device_printf` so the user sees it in `dmesg`.
2. Increment a counter (`power_resume_errors`) that a regression script or an observability tool can check.
3. Mark the device broken so that subsequent requests fail cleanly rather than silently corrupting data.
4. Keep the driver attached, so the device-tree state stays consistent and the user can eventually unload and reload the driver.
5. Return 0 from `DEVICE_RESUME`, because the kernel expects it to succeed.

The “mark broken, keep attached” pattern is common in production drivers. It moves the failure from “mysterious later corruption” to “immediate user-visible error”, which is a better debugging experience.

22.5.13 A Short Detour: pci_save_state / pci_restore_state in Runtime PM

Section 2 mentioned that pci_save_state and pci_restore_state are sometimes called by the driver itself, typically in a runtime power-management helper. This is worth a concrete sketch before Section 5 builds it out.

A runtime PM helper that puts an idle device into D3 looks like:

```
static int
myfirst_runtime_suspend(struct myfirst_softc *sc)
{
    int err;

    err = myfirst_quiesce(sc);
    if (err != 0)
        return (err);

    pci_save_state(sc->dev);
    err = pci_set_powerstate(sc->dev, PCI_POWERSTATE_D3);
    if (err != 0) {
        /* roll back */
        pci_restore_state(sc->dev);
        myfirst_restore(sc);
        return (err);
    }

    return (0);
}

static int
myfirst_runtime_resume(struct myfirst_softc *sc)
{
    int err;

    err = pci_set_powerstate(sc->dev, PCI_POWERSTATE_D0);
    if (err != 0)
        return (err);
    pci_restore_state(sc->dev);

    return (myfirst_restore(sc));
}
```

The pattern is similar to the system suspend/resume but uses the explicit PCI helpers because the PCI layer is not in the loop. Section 5 will turn this sketch into a real implementation and wire it to an idle-detection policy.

22.5.14 A Reality Check Against a Real Driver

Before moving on, it is worth pausing and looking at a real driver's resume path. `/usr/src/sys/dev/re/if_re.c`'s `re_resume` function is about thirty lines. Its structure is:

1. Lock the softc.
2. If a MAC-sleep flag is set, take the chip out of sleep mode by writing a GPIO register.
3. Clear any wake-on-LAN patterns so normal receive filtering is not interfered with.
4. If the interface is administratively up, re-initialise it via `re_init_locked`.
5. Clear the suspended flag.
6. Unlock the softc.
7. Return 0.

The `re_init_locked` call is the substantive work: it reprograms the MAC address, resets the receive and transmit descriptor rings, re-enables interrupts on the NIC, and starts the DMA engines. For `myfirst`, the equivalent work is much shorter because the device is much simpler, but the shape is the same: acquire state, do hardware-specific reinitialisation, unlock, return.

A reader who reads `re_resume` after implementing `myfirst`'s resume will recognise the structure immediately. The vocabulary is the same; only the details differ.

22.5.15 Wrapping Up Section 4

Section 4 completed the resume path. It showed what the PCI layer has already done by the time `DEVICE_RESUME` is called (D0 transition, config-space restore, PME# clear, MSI-X restore), what the driver still has to do (re-enable bus-master, restore device-specific registers, clear the suspended flag, unmask interrupts), and why each step is important. The Stage 3 driver can now do a full suspend-resume cycle and continue operating normally; the regression test can run several cycles in a row and verify the counters are consistent.

With Sections 3 and 4 together, the driver is power-aware in the system-suspend sense: it handles S3 and S4 transitions cleanly. What it still does not do is any device-level power saving while the system is running. That is runtime power management, and Section 5 teaches it.

22.6 Section 5: Handling Runtime Power Management

System suspend is a big, visible transition: the lid closes, the screen goes dark, the battery saves power for hours. Runtime power management is the opposite: dozens of small, invisible transitions a second, each saving a little, together saving much of the idle power a modern system draws. The user never notices them; the platform engineer lives or dies by their correctness.

This section is marked optional in the chapter outline because not every driver needs runtime PM. A driver for a device that is always active (a NIC on a busy server, a disk controller for the root filesystem) does not save power by attempting to suspend its device; the device is busy, and trying to suspend it wastes cycles setting up transitions that never complete. A driver for a device that is frequently idle (a webcam, a fingerprint reader, a WLAN card on a laptop) does benefit. Whether to add runtime PM is a policy decision driven by the device's usage profile.

For Chapter 22, we implement runtime PM on the `myfirst` driver as a learning exercise. The device is already simulated; we can pretend it is idle whenever no `sysctl` has been written in the last few seconds, and watch the driver go through the motions. The implementation is short, and it teaches the PCI-level primitives that a real runtime-PM driver uses.

22.6.1 What Runtime PM Means in FreeBSD

FreeBSD does not currently have a centralised runtime-PM framework the way Linux does. There is no kernel-side “if the device has been idle for N milliseconds, call its idle hook” machinery. Instead, runtime PM is driver-local: the driver decides when to suspend and resume its device, using the same PCI-layer primitives (`pci_set_powerstate`, `pci_save_state`, `pci_restore_state`) it would use inside `DEVICE_SUSPEND` and `DEVICE_RESUME`.

This has two consequences. First, every driver that wants runtime PM implements its own policy: how long the device must be idle before suspending, what counts as idle, how quickly the device must wake on demand. Second, the driver must integrate its runtime PM with its system PM; the two paths share a lot of code and must not step on each other.

The pattern Chapter 22 uses is straightforward:

1. The driver adds a small state machine with states `RUNNING` and `RUNTIME_SUSPENDED`.
2. When the driver observes idleness (Section 5 uses a callout-based “no requests in the last 5 seconds” policy), it calls `myfirst_runtime_suspend`.
3. When the driver observes a new request while in `RUNTIME_SUSPENDED`, it calls `myfirst_runtime_resume` before processing the request.
4. On system suspend, if the device is in `RUNTIME_SUSPENDED`, the system-suspend path adjusts for it (the device is already quiesced; the system-suspend quiesce is a no-op, but the system resume has to bring the device back to D0).
5. On system resume, the driver returns to `RUNNING` unless it was explicitly runtime-suspended and wants to stay that way.

This is simpler than Linux’s runtime PM framework, which has richer concepts (parent/child ref-counting, autosuspend timers, barriers). For a single driver on simple hardware, the FreeBSD approach is enough.

22.6.2 The Runtime-PM State Machine

The `softc` gains a state variable and a timestamp:

```
enum myfirst_runtime_state {
    MYFIRST_RT_RUNNING = 0,
    MYFIRST_RT_SUSPENDED = 1,
};

struct myfirst_softc {
    /* ... */
    enum myfirst_runtime_state runtime_state;
```

```

    struct timeval      last_activity;
    struct callout      idle_watcher;
    int                 idle_threshold_seconds;
    uint64_t            runtime_suspend_count;
    uint64_t            runtime_resume_count;
};

```

The `idle_threshold_seconds` is a policy knob exposed through a `sysctl`; defaulting to five seconds gives quick observability without being so aggressive as to cause unnecessary wake-ups during normal use. A production driver would tune this per-device; five seconds is a learning-friendly value that makes the transitions visible without requiring hours of waiting.

The `idle_watcher` callout fires once a second to check the idle time. If the device has been idle longer than `idle_threshold_seconds` and is currently in `RUNNING`, the callout triggers `myfirst_runtime_suspend`.

22.6.3 Implementation

The attach path starts the idle watcher:

```

static void
myfirst_start_idle_watcher(struct myfirst_softc *sc)
{
    sc->idle_threshold_seconds = 5;
    microtime(&sc->last_activity);
    callout_init_mtx(&sc->idle_watcher, &sc->mtx, 0);
    callout_reset(&sc->idle_watcher, hz, myfirst_idle_watcher_cb, sc);
}

```

The callout is initialised with the softc mutex, so it acquires the mutex automatically when firing. That simplifies the callback: it runs under the lock.

The callback checks the time since the last activity and suspends if needed:

```

static void
myfirst_idle_watcher_cb(void *arg)
{
    struct myfirst_softc *sc = arg;
    struct timeval now, diff;

    MYFIRST_ASSERT_LOCKED(sc);

    if (sc->runtime_state == MYFIRST_RT_RUNNING && !sc->suspended) {
        microtime(&now);
        timersub(&now, &sc->last_activity, &diff);
    }
}

```

```

        if (diff.tv_sec >= sc->idle_threshold_seconds) {
            /*
             * Release the lock while suspending. The
             * runtime_suspend helper acquires it again as
             * needed.
             */
            MYFIRST_UNLOCK(sc);
            (void)myfirst_runtime_suspend(sc);
            MYFIRST_LOCK(sc);
        }

        /* Reschedule. */
        callout_reset(&sc->idle_watcher, hz, myfirst_idle_watcher_cb, sc);
    }
}

```

Note the lock-drop around `myfirst_runtime_suspend`. The suspend helper calls `myfirst_quiesce`, which acquires the lock itself. Holding the lock across it would deadlock.

Activity is recorded whenever the driver services a request. The Chapter 21 DMA path is a good hook: every time a user writes to `dma_test_read` or `dma_test_write`, the `sysctl` handler records activity:

```

static int
myfirst_dma_sysctl_test_write(SYSCTL_HANDLER_ARGS)
{
    struct myfirst_softc *sc = arg1;
    /* ... existing code ... */

    /* Mark the device active before processing. */
    myfirst_mark_active(sc);

    /* If runtime-suspended, bring the device back before running. */
    if (sc->runtime_state == MYFIRST_RT_SUSPENDED) {
        int err = myfirst_runtime_resume(sc);
        if (err != 0)
            return (err);
    }

    /* ... proceed with the transfer ... */
}

```

The `myfirst_mark_active` helper is a one-liner:

```
static void
myfirst_mark_active(struct myfirst_softc *sc)
{
    MYFIRST_LOCK(sc);
    microtime(&sc->last_activity);
    MYFIRST_UNLOCK(sc);
}
```

22.6.4 The Runtime-Suspend and Runtime-Resume Helpers

These were sketched in Section 4. Here are the fleshed-out versions:

```
static int
myfirst_runtime_suspend(struct myfirst_softc *sc)
{
    int err;

    device_printf(sc->dev, "runtime suspend: starting\n");

    err = myfirst_quiesce(sc);
    if (err != 0) {
        device_printf(sc->dev,
            "runtime suspend: quiesce failed (err %d)\n", err);
        /* Undo the suspended flag the quiesce set. */
        MYFIRST_LOCK(sc);
        sc->suspended = false;
        MYFIRST_UNLOCK(sc);
        return (err);
    }

    pci_save_state(sc->dev);
    err = pci_set_powerstate(sc->dev, PCI_POWERSTATE_D3);
    if (err != 0) {
        device_printf(sc->dev,
            "runtime suspend: set_powerstate(D3) failed "
            "(err %d)\n", err);
        pci_restore_state(sc->dev);
        myfirst_restore(sc);
        return (err);
    }

    MYFIRST_LOCK(sc);
    sc->runtime_state = MYFIRST_RT_SUSPENDED;
    MYFIRST_UNLOCK(sc);
}
```

```
    atomic_add_64(&sc->runtime_suspend_count, 1);

    device_printf(sc->dev, "runtime suspend: device in D3\n");
    return (0);
}

static int
myfirst_runtime_resume(struct myfirst_softc *sc)
{
    int err;

    MYFIRST_LOCK(sc);
    if (sc->runtime_state != MYFIRST_RT_SUSPENDED) {
        MYFIRST_UNLOCK(sc);
        return (0); /* nothing to do */
    }
    MYFIRST_UNLOCK(sc);

    device_printf(sc->dev, "runtime resume: starting\n");

    err = pci_set_powerstate(sc->dev, PCI_POWERSTATE_D0);
    if (err != 0) {
        device_printf(sc->dev,
            "runtime resume: set_powerstate(D0) failed "
            "(err %d)\n", err);
        return (err);
    }
    pci_restore_state(sc->dev);

    err = myfirst_restore(sc);
    if (err != 0) {
        device_printf(sc->dev,
            "runtime resume: restore failed (err %d)\n", err);
        return (err);
    }

    MYFIRST_LOCK(sc);
    sc->runtime_state = MYFIRST_RT_RUNNING;
    MYFIRST_UNLOCK(sc);
    atomic_add_64(&sc->runtime_resume_count, 1);

    device_printf(sc->dev, "runtime resume: device in D0\n");
    return (0);
}
```

The shape is identical to system suspend/resume except that the driver explicitly calls `pci_set_powerstate` and `pci_save_state/pci_restore_state`. The PCI layer's automatic transitions are not in the loop for runtime PM because the kernel is not coordinating a system-wide power change; the driver is on its own.

22.6.5 Interaction Between Runtime PM and System PM

The two paths have to cooperate. Consider what happens if the device is runtime-suspended (in D3) when the user closes the laptop lid:

1. The kernel starts system suspend.
2. The PCI bus calls `myfirst_pci_suspend`.
3. Inside `myfirst_pci_suspend`, the driver notices that the device is already runtime-suspended. The quiesce is a no-op (nothing is happening). The PCI layer's automatic config-space save runs; it reads the config space (which is still accessible in D3) and caches it.
4. The PCI layer transitions the device from D3 to... wait, it is already in D3. The transition to D3 is a no-op.
5. The system sleeps.
6. On wake, the PCI layer transitions the device back to D0. The driver's `myfirst_pci_resume` runs. It restores state. But now the driver thinks the device is RUNNING (because system resume cleared the suspended flag), while conceptually it was runtime-suspended before. The next activity will use the device normally and set `last_activity`; the idle watcher will eventually re-suspend it if still idle.

The interaction is mostly benign; the worst that happens is that the device gets one extra trip through D0 before the idle watcher re-suspends it. A more polished implementation would remember the runtime-suspended state across the system suspend and restore it, but for a learning driver the simple approach is enough.

The reverse (system-suspending a device that is already runtime-suspended) is already correct in our implementation because `myfirst_quiesce` checks `suspended` and returns 0 if already set. The runtime-suspended path set `suspended = true` as part of its quiesce, so the system suspend's quiesce sees the flag and skips.

22.6.6 Exposing Runtime-PM Controls Through Sysctl

The driver's runtime-PM policy can be controlled and observed through sysctls:

```

SYSCTL_ADD_INT(ctx, kids, OID_AUTO, "idle_threshold_seconds",
    CTLFLAG_RW, &sc->idle_threshold_seconds, 0,
    "Runtime PM idle threshold (seconds)");
SYSCTL_ADD_U64(ctx, kids, OID_AUTO, "runtime_suspend_count",
    CTLFLAG_RD, &sc->runtime_suspend_count, 0,
    "Runtime suspends performed");
SYSCTL_ADD_U64(ctx, kids, OID_AUTO, "runtime_resume_count",
    CTLFLAG_RD, &sc->runtime_resume_count, 0,
    "Runtime resumes performed");
SYSCTL_ADD_INT(ctx, kids, OID_AUTO, "runtime_state",
    CTLFLAG_RD, (int *)&sc->runtime_state, 0,
    "Runtime state: 0=running, 1=suspended");

```

A reader can now do this:

```
# Watch the device idle out.
while ;; do
    sysctl dev.myfirst.0.runtime_state dev.myfirst.0.runtime_suspend_count
    sleep 1
done &
```

After five seconds of inactivity, `runtime_state` flips from 0 to 1 and `runtime_suspend_count` increments. A write to any active sysctl triggers a resume and flips the state back:

```
sudo sysctl dev.myfirst.0.dma_test_read=1
# The log shows: runtime resume, then the test read
```

22.6.7 Tradeoffs

Runtime PM trades wake-up latency for idle power. Every D3-to-D0 transition costs time (tens of microseconds on a PCIe link, including the ASPM exit), and on some devices costs energy (the transition itself draws current). For a device that is idle most of the time with rare bursts of activity, the trade is favorable. For a device that is active most of the time with rare idle periods, the cost of the transitions dominates.

The `idle_threshold_seconds` knob lets the platform tune this. A value of 0 or 1 is aggressive and suitable for a webcam that is used for seconds at a time and idle for minutes. A value of 60 is conservative and suitable for a NIC whose idle periods are short but frequent. A value of 0 (if allowed) would disable runtime PM entirely, which is appropriate for devices that should stay on at all times.

A second tradeoff is in code complexity. Runtime PM adds a state machine, a callout, an idle watcher, two more `kobj`-like helpers, and additional ordering concerns between the runtime and system PM paths. Each of those is small, but together they increase the surface area for bugs. Many FreeBSD drivers deliberately omit runtime PM for this reason; they let the device stay in D0 and rely on the device's internal low-power states (clock gating, PCIe ASPM) to save power. That is a defensible choice, and for drivers where correctness matters more than milliwatts, it is the right one.

Chapter 22's `myfirst` driver keeps runtime PM as an optional feature, gated by a build-time flag:

```
CFLAGS+= -DMYFIRST_ENABLE_RUNTIME_PM
```

A reader can build with or without the flag; the Section 5 code is only compiled in when the flag is defined. The default for Stage 3 is to leave runtime PM off; Stage 4 enables it in the consolidated driver.

22.6.8 A Word on Platform Runtime PM

Some platforms provide their own runtime-PM mechanism alongside the driver-local one. On arm64 and RISC-V embedded systems, the device tree may describe `power-domains` and `clocks` properties that the driver uses

to turn off power domains and gate clocks. FreeBSD's `ext_resources/clk`, `ext_resources/regulator`, and `ext_resources/power` subsystems handle these.

Runtime PM on such a platform is more capable than PCI-only runtime PM because the platform can turn off entire SoC blocks (a USB controller, a display engine, a GPU) rather than just moving the PCI device to D3. The driver uses the same pattern (mark idle, turn off resources on idle, turn back on for activity) but through different APIs.

Chapter 22 stays with the PCI path because that is where the `myfirst` driver lives. A reader who later works on an embedded platform will find the same conceptual structure with platform-specific APIs. The chapter mentions the distinction here so the reader knows the territory exists.

22.6.9 Wrapping Up Section 5

Section 5 added runtime power management to the driver. It defined a two-state machine (`RUNNING`, `RUNTIME_SUSPENDED`), a callout-based idle watcher, a pair of helpers (`myfirst_runtime_suspend`, `myfirst_runtime_resume`) that use the PCI layer's explicit power-state and state-save APIs, the activity-recording hooks in the DMA `sysctl` handlers, and the `sysctl` knobs that expose the policy to user space. It also discussed the interaction between runtime PM and system PM, the latency-vs-power tradeoff, and the alternative of platform-level runtime PM on embedded systems.

With Sections 2 through 5 in place, the driver now handles system suspend, system resume, system shutdown, runtime suspend, and runtime resume. What it does not yet do cleanly is explain how the reader tests all of these from user space. Section 6 turns to the user-space interface: `acpiconf`, `zzz`, `devctl suspend`, `devctl resume`, `devinfo -v`, and the regression test that wraps them together.

22.7 Section 6: Interacting With the Power Framework

A driver that handles suspend and resume correctly is only half of the story. The other half is being able to *test* that correctness, repeatedly and deliberately, from user space. Section 6 surveys the tools FreeBSD provides for that purpose, explains how each fits the driver's state model, and shows how to combine them into a regression script that exercises every path Sections 2 through 5 built.

22.7.1 The Four User-Space Entry Points

Four commands cover almost everything a driver developer needs:

- **`acpiconf -s 3`** (and its variants) asks ACPI to put the whole system into sleep state S3. This is the most realistic test; it exercises the full path from user space through the kernel's suspend machinery through the PCI layer to the driver's methods.
- **`zzz`** is a thin wrapper around `acpiconf -s 3`. It reads `hw.acpi.suspend_state` (defaulting to S3) and enters the corresponding sleep state. For most users it is the most convenient way to suspend from a shell.
- **`devctl suspend myfirst0`** and **`devctl resume myfirst0`** trigger per-device suspend and resume through the `DEV_SUSPEND` and `DEV_RESUME` ioctls on `/dev/devctl2`. These only call the driver's methods; the rest of the system stays in S0. This is the fastest iteration target and what Chapter 22 uses for most development.

- **devinfo -v** lists all devices in the device tree with their current state. It shows whether a device is attached, suspended, or detached.

Each has strengths and weaknesses. `acpiconf` is realistic but slow (one to three seconds per cycle on typical hardware) and disruptive (the system actually sleeps). `devctl` is fast (milliseconds per cycle) but exercises only the driver, not the ACPI or platform code. `devinfo -v` is passive and cheap; it observes without changing state.

A good regression strategy uses all three: `devctl` for unit testing of the driver's methods, `acpiconf` for integration testing of the full suspend path, and `devinfo -v` as a quick sanity check.

22.7.2 Using `acpiconf` to Suspend the System

On a machine with working ACPI, `acpiconf -s 3` is what Section 1 called a full system suspend. The command:

```
sudo acpiconf -s 3
```

does the following:

1. It opens `/dev/acpi` and checks that the platform supports S3 via the `ACPIIO_ACKSLPSTATE` ioctl.
2. It sends the `ACPIIO_REQSLPSTATE` ioctl to request S3.
3. The kernel begins the suspend sequence: paused userland, frozen threads, device tree traversal with `DEVICE_SUSPEND` on each device.
4. Assuming no driver vetoes, the kernel enters S3. The machine sleeps.
5. A wake event (the lid opens, the power button is pressed, a USB device sends a remote-wakeup signal) wakes the platform.
6. The kernel runs the resume sequence: `DEVICE_RESUME` on each device, unfreezing threads, resuming userland.
7. The shell prompt returns. The machine is back in S0.

For the `myfirst` driver to be exercised, the driver must be loaded before the suspend. The entire sequence from user perspective looks like:

```
sudo kldload ./myfirst.ko
sudo sysctl dev.myfirst.0.dma_test_read=1 # exercise it a bit
sudo acpiconf -s 3
# [laptop sleeps; user opens lid]
dmesg | grep myfirst
```

The `dmesg` output should show two lines from Chapter 22's logging:

```
myfirst0: suspend: starting
myfirst0: suspend: complete (dma in flight=0, suspended=1)
myfirst0: resume: starting
myfirst0: resume: complete
```

If those lines are present and in that order, the driver's methods were called correctly by the full system path.

If the machine does not come back, the suspend path broke at some layer below `myfirst`. If the machine comes back but the driver is in a strange state (the sysctls return errors, the counters have strange values, DMA transfers fail), the problem is in `myfirst`'s suspend or resume implementation.

22.7.3 Using `zzz`

On FreeBSD, `zzz` is a small shell script that reads `hw.acpi.suspend_state` and calls `acpiconf -s <state>`. It is not a binary; it is usually installed at `/usr/sbin/zzz` and is a few lines long. A typical invocation is:

```
sudo zzz
```

The default `hw.acpi.suspend_state` is `S3` on machines that support it. A reader who wants to test `S4` (hibernate) can:

```
sudo sysctl hw.acpi.suspend_state=S4
sudo zzz
```

`S4` support on FreeBSD has historically been partial; whether it works depends on the platform firmware and the filesystem layout. For Chapter 22's purposes, `S3` is sufficient, and `zzz` is the convenient shorthand.

22.7.4 Using `devctl` for Per-Device Suspend

The `devctl(8)` command was built to let a user manipulate the device tree from user space. It supports attach, detach, enable, disable, suspend, resume, and more. For Chapter 22, suspend and resume are the two that matter.

```
sudo devctl suspend myfirst0
sudo devctl resume myfirst0
```

The first command issues `DEV_SUSPEND` through `/dev/devctl2`; the kernel translates that into a call to `BUS_SUSPEND_CHILD` on the parent bus, which for a PCI device ends up calling `pci_suspend_child`, which saves config space, puts the device in `D3`, and calls the driver's `DEVICE_SUSPEND`. The reverse happens for resume.

The key differences from `acpiconf`:

- Only the target device and its children go through the transition. The rest of the system stays in `S0`.
- The CPU does not park. Userland does not freeze. The kernel does not sleep.
- The PCI device actually goes to `D3hot` (assuming `hw.pci.do_power_suspend` is 1). The reader can verify with `pciconf`:

```
# Before suspend: device should be in D0
pciconf -lvbc | grep -A 2 myfirst

# After devctl suspend myfirst0: device should be in D3
sudo devctl suspend myfirst0
pciconf -lvbc | grep -A 2 myfirst
```

The power state is usually shown in the powerspec line of `pciconf -lvbc`. Moving from D0 to D3 is the observable signal that the transition really happened.

22.7.5 Using devinfo to Inspect Device State

The `devinfo(8)` utility lists the device tree with various levels of detail. The `-v` flag shows verbose information, including the device state (attached, suspended, or not present).

```
devinfo -v | grep -A 5 myfirst
```

Typical output:

```
myfirst0 pnpinfo vendor=0x1af4 device=0x1005 subvendor=0x1af4 subdevice=0x0004
↳ class=0x008880 at slot=5 function=0 dbsf=pci0:0:5:0
  Resource: <INTERRUPT>
    10
  Resource: <MEMORY>
    0xfeb80000-0xfeb80fff
```

The state is implicit in the output: if the device is suspended, the line shows the device and its resources without the “active” marker. An explicit state query can be done through the `softc sysctl`; the `dev.myfirst.0.%parent` and `dev.myfirst.0.%desc` keys tell the user where the device sits.

For Chapter 22, `devinfo -v` is most useful as a sanity check after a failed transition: if the device is missing from the output, the detach path ran; if the device is present but the resources are wrong, the attach or resume path left the device in an inconsistent state.

22.7.6 Inspecting Power States Through sysctl

The PCI layer exposes power-state information through `sysctl` under `hw.pci`. Two variables are most relevant:

```
sysctl hw.pci.do_power_suspend
sysctl hw.pci.do_power_resume
```

Both default to 1, meaning the PCI layer transitions devices to D3 on suspend and back to D0 on resume. Setting either to 0 disables the automatic transition for debugging.

The ACPI layer exposes system-state information:

```
sysctl hw.acpi.supported_sleep_state
sysctl hw.acpi.suspend_state
sysctl hw.acpi.s4bios
```

The first lists which sleep states the platform supports (typically something like S3 S4 S5). The second is the state zzz enters (usually S3). The third says whether S4 is implemented through BIOS assistance.

For per-device observation, the driver exposes its own state through `dev.myfirst.N.*`. The Chapter 22 driver adds:

- `dev.myfirst.N.suspended`: 1 if the driver considers itself suspended, 0 otherwise.
- `dev.myfirst.N.power_suspend_count`: number of times `DEVICE_SUSPEND` has been called.
- `dev.myfirst.N.power_resume_count`: number of times `DEVICE_RESUME` has been called.
- `dev.myfirst.N.power_shutdown_count`: number of times `DEVICE_SHUTDOWN` has been called.
- `dev.myfirst.N.runtime_state`: 0 for `RUNNING`, 1 for `RUNTIME_SUSPENDED`.
- `dev.myfirst.N.runtime_suspend_count`, `dev.myfirst.N.runtime_resume_count`: runtime-PM counters.
- `dev.myfirst.N.idle_threshold_seconds`: runtime-PM idle threshold.

Between these `sysctls` and `dmesg`, a reader can see in full detail what the driver did during any transition.

22.7.7 A Regression Script

The `labs` directory grows a new script: `ch22-suspend-resume-cycle.sh`. The script:

1. Records the baseline values of every counter.
2. Runs one DMA transfer to confirm the device is working.
3. Calls `devctl suspend myfirst0`.
4. Verifies `dev.myfirst.0.suspended` is 1.
5. Verifies `dev.myfirst.0.power_suspend_count` has incremented by 1.
6. Calls `devctl resume myfirst0`.
7. Verifies `dev.myfirst.0.suspended` is 0.
8. Verifies `dev.myfirst.0.power_resume_count` has incremented by 1.
9. Runs one more DMA transfer to confirm the device still works.
10. Prints a PASS/FAIL summary.

The full script is in the `examples` directory; a short outline of the logic:

```
#!/bin/sh
set -e

DEV="dev.myfirst.0"
```

```
if ! sysctl -a | grep -q "^${DEV}"; then
    echo "FAIL: ${DEV} not present"
    exit 1
fi

before_s=$(sysctl -n ${DEV}.power_suspend_count)
before_r=$(sysctl -n ${DEV}.power_resume_count)
before_xfer=$(sysctl -n ${DEV}.dma_transfers_read)

# Baseline: run one transfer.
sysctl -n ${DEV}.dma_test_read=1 > /dev/null

# Suspend.
devctl suspend myfirst0
[ "$(sysctl -n ${DEV}.suspended)" = "1" ] || {
    echo "FAIL: device did not mark suspended"
    exit 1
}

# Resume.
devctl resume myfirst0
[ "$(sysctl -n ${DEV}.suspended)" = "0" ] || {
    echo "FAIL: device did not clear suspended"
    exit 1
}

# Another transfer.
sysctl -n ${DEV}.dma_test_read=1 > /dev/null

after_s=$(sysctl -n ${DEV}.power_suspend_count)
after_r=$(sysctl -n ${DEV}.power_resume_count)
after_xfer=$(sysctl -n ${DEV}.dma_transfers_read)

if [ $((after_s - before_s)) -ne 1 ]; then
    echo "FAIL: suspend count did not increment by 1"
    exit 1
fi
if [ $((after_r - before_r)) -ne 1 ]; then
    echo "FAIL: resume count did not increment by 1"
    exit 1
fi
if [ $((after_xfer - before_xfer)) -ne 2 ]; then
    echo "FAIL: expected 2 transfers (pre+post), got $((after_xfer - before_xfer))"
```

```
    exit 1
fi

echo "PASS: one suspend-resume cycle completed cleanly"
```

Running the script repeatedly (say a hundred times in a tight loop) is a good stress test. A driver that passes one cycle but fails on the fiftieth usually has a resource leak or an edge case that only shows up under repetition. That class of bug is exactly what a regression script is meant to find.

22.7.8 Running the Stress Test

The chapter's `labs/` directory also includes `ch22-suspend-stress.sh`, which runs the cycle script a hundred times:

```
#!/bin/sh
N=100
i=0
while [ $i -lt $N ]; do
    if ! sh ./ch22-suspend-resume-cycle.sh > /dev/null; then
        echo "FAIL on iteration $i"
        exit 1
    fi
    i=$((i + 1))
done
echo "PASS: $N cycles"
```

On a modern machine with the simulation-only `myfirst` driver, a hundred cycles takes about a second. If any iteration fails, the script stops and reports the iteration number. Running this after each change during development catches regressions immediately.

22.7.9 Combining Runtime PM and User-Space Testing

The runtime-PM path needs a different test, because it is not triggered by user commands; it is triggered by idleness. The test looks like:

```
# Ensure runtime_state is running.
sysctl dev.myfirst.0.runtime_state
# 0

# Do nothing for 6 seconds.
sleep 6

# The callout should have fired and runtime-suspended the device.
```

```

sysctl dev.myfirst.0.runtime_state
# 1

# Counter should have incremented.
sysctl dev.myfirst.0.runtime_suspend_count
# 1

# Any activity should bring it back.
sysctl dev.myfirst.0.dma_test_read=1
sysctl dev.myfirst.0.runtime_state
# 0

sysctl dev.myfirst.0.runtime_resume_count
# 1

```

A reader watching `dmesg` during this will see the “runtime suspend: starting” and “runtime suspend: device in D3” lines after about five seconds of inactivity, then “runtime resume: starting” when the `sysctl` write arrives.

The chapter’s lab directory includes `ch22-runtime-pm.sh` to automate this sequence.

22.7.10 Interpreting Failure Modes

When a user-space test fails, the diagnostic path depends on which layer failed:

- **If `devctl suspend` returns a non-zero exit code:** the driver’s `DEVICE_SUSPEND` returned a non-zero value, vetoing the suspend. Check `dmesg` for the driver’s log output; the suspend method should be logging what went wrong.
- **If `devctl suspend` succeeds but `dev.myfirst.0.suspended` is 0 afterwards:** the driver’s quiesce set the flag briefly but something cleared it. This usually means the quiesce is re-entering itself, or the detach path is racing the suspend.
- **If `devctl resume` succeeds but the next transfer fails:** the restore path did not fully reinitialise the device. Most commonly, an interrupt mask or a DMA register was not written; check the per-vector fire counters before and after resume to see whether interrupts are reaching the driver.
- **If `acpicnf -s 3` succeeds but the system does not come back:** a driver below `myfirst` in the tree is blocking resume. This is unusual in a test VM; it is the classic failure mode on real hardware with new drivers.
- **If `acpicnf -s 3` returns `EOPNOTSUPP`:** the platform does not support S3. Check `sysctl hw.acpi.supported_sleep_state`.

In all cases, the first source of information is `dmesg`. The Chapter 22 driver logs every transition; if the log lines do not appear, the method was not called, and the problem is at a layer below the driver.

22.7.11 A Minimal Troubleshooting Flow

A compact flowchart for a failed suspend-resume cycle:

1. Is the driver loaded? `kldstat | grep myfirst.`
2. Is the device attached? `sysctl dev.myfirst.0.%driver.`
3. Do the suspend and resume methods log? `dmesg | tail.`
4. Did `dev.myfirst.0.suspended` toggle correctly? `sysctl dev.myfirst.0.suspended.`
5. Do the counters increment? `sysctl dev.myfirst.0.power_suspend_count dev.myfirst.0.power_resume_count.`
6. Does a post-resume transfer succeed? `sudo sysctl dev.myfirst.0.dma_test_read=1; dmesg | tail -2.`
7. Do the per-vector interrupt counters increment? `sysctl dev.myfirst.0.vec0_fire_count dev.myfirst.0.vec1_fire_count dev.myfirst.0.vec2_fire_count.`

Any “no” answer points to a specific layer of the implementation. Section 7 goes deeper into the common failure modes and how to debug them.

22.7.12 Wrapping Up Section 6

Section 6 surveyed the user-space interface to the kernel’s power-management machinery: `acpiconf`, `zzz`, `devctl suspend`, `devctl resume`, `devinfo -v`, and the relevant `sysctl` variables. It showed how to combine these tools into a regression script that exercises one suspend-resume cycle, and a stress script that runs a hundred cycles in a row. It discussed the runtime-PM test flow, the interpretation of the most common failure modes, and the minimal troubleshooting flowchart a reader can follow when a test fails.

With the user-space tools in hand, the next section dives into the characteristic failure modes the reader is likely to encounter while writing power-aware code, and how to debug each one.

22.8 Section 7: Debugging Power Management Issues

Power management code has a special class of bugs. The machine sleeps; the machine wakes; the bug shows up an unknown time after the wake and looks like a generic malfunction rather than anything related to the power transition. The chain of cause and effect is longer than with most driver bugs, the reproduction is slower, and the user’s bug report is usually “my laptop doesn’t wake up sometimes”, which contains almost no information the driver developer can use.

Section 7 is about recognising the characteristic symptoms, tracing them back to their likely causes, and applying the matching debugging patterns. It draws on the Chapter 22 `myfirst` driver for concrete examples, but the patterns apply to any FreeBSD driver.

22.8.1 Symptom 1: Frozen Device After Resume

The most common power-management bug, both in learning drivers and in production ones, is a device that stops responding after resume. The driver attaches correctly at boot, works normally in S0, handles a suspend-resume cycle without visible error, and then on the next command it is silent. Interrupts do not fire. DMA transfers do not complete. Any read from a device register returns stale values or zeros.

The usual cause is that the device’s registers were not written after resume. The device came back in a default state (interrupt mask all-masked, DMA engine disabled, whatever registers the hardware resets on D0 entry), the driver did not reprogram them, and so from the device’s perspective nothing is configured to run.

Debugging pattern. Compare the device's register values before and after suspend. The `myfirst` driver exposes several of its registers through `sysctls` (if the reader adds them); otherwise, the reader can write a short kernel-space helper that reads each register and prints it. After a suspend-resume cycle:

1. Read the interrupt mask register. If it is `0xFFFFFFFF` (all masked), the resume path did not restore the mask.
2. Read the DMA control register. If it has the ABORT bit set, the abort from the quiesce never cleared.
3. Read the device's configuration space via `pciconf -lvbc`. The command register should have the bus-master bit set; if not, `pci_enable_busmaster` was missed in the resume path.

Fix pattern. The resume path should include an unconditional reprogram of every device-specific register the driver's normal operation depends on. Saving them at suspend time into the `softc` and restoring them at resume time is one approach; re-deriving them from `softc` state (the approach `re_resume` takes) is another. Either works; the choice depends on which is easier to prove correct for the specific device.

22.8.2 Symptom 2: Lost Interrupts

A subtler variant of the frozen-device problem is lost interrupts: the device is responding to some calls, but its interrupts are not reaching the driver. The DMA engine accepts a START command, performs the transfer, raises the completion interrupt... and the interrupt count does not increment. The task queue does not get an entry. The CV does not broadcast. The transfer eventually times out, and the driver reports EIO.

Several things can cause this:

- The **interrupt mask** at the device is still all-masked. The device wants to raise the interrupt but the mask suppresses it. (Resume path bug.)
- The **MSI or MSI-X configuration** was not restored. The device is raising the interrupt, but the kernel does not route it to the driver's handler. (Unusual; the PCI layer should handle this automatically.)
- The **filter function pointer** was corrupted. Extremely unusual; usually indicates memory corruption somewhere else in the driver.
- The **suspended flag** is still true, and the filter is returning early. (Resume path bug: flag not cleared.)

Debugging pattern. Read the per-vector fire counters before and after the suspend-resume cycle. If the counter does not increment, the interrupt is not reaching the filter. Then check, in order:

1. Is the suspended flag cleared? `sysctl dev.myfirst.0.suspended`.
2. Is the interrupt mask at the device correct? Read the register.
3. Is the MSI-X table in the device correct? `pciconf -c` dumps the capability registers.
4. Is the kernel's MSI dispatch state consistent? `procstat -t` shows the interrupt threads.

Fix pattern. Make sure the resume path (a) clears the suspended flag under the lock, (b) unmask the device's interrupt register after clearing the flag, (c) does not rely on MSI-X restoration the driver must do itself (unless specifically disabled via `sysctl`).

22.8.3 Symptom 3: Bad DMA After Resume

A more dangerous class of bug is DMA that appears to work but produces wrong data. The driver programs the engine, the engine runs, the completion interrupt fires, the task runs, the sync is called, the driver reads the

buffer... and the bytes are wrong. Not zeros, not garbage, just subtly incorrect: the pattern written previously, or the pattern from two cycles ago, or a pattern that indicates the DMA addressed the wrong page.

Causes:

- The **bus address cached in the softc** is stale. This is unusual for a static allocation (the address is set once at attach and should not change), but it can happen if the driver re-allocates the DMA buffer at resume time (a bad idea; see below).
- The **DMA engine's base-address register** was not reprogrammed after resume, and it has a stale value that points elsewhere.
- The **bus_dmamap_sync calls are missing or mis-ordered**. This is the classic DMA-correctness bug, and it is worth being alert for in resume paths because the driver-side code adjacent to the sync calls is often edited during a refactor.
- The **IOMMU translation table** was not restored. Very rare on FreeBSD because the IOMMU configuration is per-session and survives suspend on most platforms; but if the driver is running on a system where DEV_IOMMU is unusual, this can bite.

Debugging pattern. Add a known-pattern write before each DMA, a verify after each DMA, and log both. Reducing the cycle to “write 0xAA, sync, read, expect 0xAA” makes data-corruption bugs visible immediately.

```
memset(sc->dma_vaddr, 0xAA, MYFIRST_DMA_BUFFER_SIZE);
bus_dmamap_sync(sc->dma_tag, sc->dma_map, BUS_DMASYNC_PREWRITE);
/* run transfer */
bus_dmamap_sync(sc->dma_tag, sc->dma_map, BUS_DMASYNC_POSTWRITE);
if (((uint8_t *)sc->dma_vaddr)[0] != 0xAA) {
    device_printf(sc->dev,
        "dma: corruption detected after transfer\n");
}
```

For the simulation, this should always succeed because the simulation does not modify the buffer on a write transfer. On real hardware, the pattern depends on the device. A reader debugging a real-hardware bug adapts the test.

Fix pattern. If the bus address is the problem, rebuild it on resume:

```
/* In resume, after PCI restore is complete. */
err = bus_dmamap_load(sc->dma_tag, sc->dma_map,
    sc->dma_vaddr, MYFIRST_DMA_BUFFER_SIZE,
    myfirst_dma_single_map, &sc->dma_bus_addr,
    BUS_DMA_NOWAIT);
```

Only do this if the bus address actually changed, which is rare. More commonly, the fix is to write the base-address register at the start of every transfer (rather than relying on a persistent value) and to make sure the sync calls are in the right order.

22.8.4 Symptom 4: Lost PME# Wake Events

On a device that supports wake-on-X, the symptom is “the device should have woken the system but did not”. The driver reported a successful suspend; the system went to S3; the expected event (magic packet, button press, timer) happened; and the system stayed asleep.

Causes:

- **pci_enable_pme was not called** in the suspend path. The device’s PME_En bit is 0, so even when the device would normally assert PME#, the bit is suppressed.
- **The device’s own wake logic is not configured.** For a NIC, the wake-on-LAN registers must be programmed before suspend. For a USB host controller, the remote-wakeup capability must be enabled per-port.
- **The platform’s wake GPE is not enabled.** This is usually a firmware matter; the ACPI _PRW method should have registered the GPE, but on some machines the BIOS disables it by default.
- **The PME status bit is set at the time of suspend**, and a stale PME# is what triggers the wake (instead of the expected event). The system appears to wake immediately after sleeping.

Debugging pattern. Read the PCI configuration space via `pciconf -lvbc`. The power-management capability’s status/control register shows PME_En and the PME_Status bit. Before suspending, PME_Status should be 0 (no pending wake). After suspending with wake enabled, PME_En should be 1.

On a machine where the wake does not happen, check the BIOS settings for “wake on LAN”, “wake on USB”, etc. The driver can be perfect and the system still not wake if the platform is not configured.

Fix pattern. In the suspend path of a wake-capable driver:

```
static int
myfirst_pci_suspend(device_t dev)
{
    struct myfirst_softc *sc = device_get_softc(dev);
    int err;

    /* ... quiesce as before ... */

    if (sc->wake_enabled && pci_has_pm(dev)) {
        /* Program device-specific wake logic here. */
        myfirst_program_wake(sc);
        pci_enable_pme(dev);
    }

    /* ... rest of suspend ... */
}
```

In the resume path:

```
static int
myfirst_pci_resume(device_t dev)
{
    if (pci_has_pm(dev))
        pci_clear_pme(dev);
    /* ... rest of resume ... */
}
```

The myfirst driver in Chapter 22 does not implement wake (the simulation has no wake logic). The pattern above is shown for reference.

22.8.5 Symptom 5: WITNESS Complaints During Suspend

A debug kernel with WITNESS enabled often produces messages like:

```
witness: acquiring sleepable lock foo_mtx @ /path/to/driver.c:123
witness: sleeping with non-sleepable lock bar_mtx @ /path/to/driver.c:456
```

These are lock-order violations or sleep-while-locked violations, and they often show up in suspend code because suspend does things the driver does not normally do: acquire locks, sleep, and coordinate multiple threads.

Causes:

- The suspend path acquires a lock and then calls a function that sleeps without explicit tolerance for sleeping-with-that-lock-held.
- The suspend path acquires locks in a different order than the rest of the driver, and WITNESS notices the reversal.
- The suspend path calls `taskqueue_drain` or `callout_drain` while holding the softc lock, which causes a deadlock if the task or callout tries to acquire the same lock.

Debugging pattern. Read the WITNESS message carefully. It includes the lock names and the source-line numbers where each was acquired. Trace the path from the acquisition to the sleep or lock reversal.

Fix pattern. The Chapter 22 `myfirst_quiesce` drops the softc lock before calling `myfirst_drain_workers` for exactly this reason. When extending the driver:

- Do not call `taskqueue_drain` with any driver lock held.
- Do not call `callout_drain` with the lock that the callout acquires.
- Sleep primitives (`pause`, `cv_wait`) must be called with only sleep-mutexes held (not spin-mutexes).
- If you need to drop a lock for a sleep, do so explicitly and reacquire after.

22.8.6 Symptom 6: Counters That Do Not Match

The chapter's regression script expects `power_suspend_count == power_resume_count` after each cycle. When they drift, something is wrong.

Causes:

- The driver's `DEVICE_SUSPEND` was called but the driver returned early before incrementing the counter. (Often because of a sanity check that fired.)
- The driver's `DEVICE_RESUME` was not called, because `DEVICE_SUSPEND` returned non-zero and the kernel unwound.
- The counters are not atomic and a concurrent update lost an increment. (Unlikely if the code uses `atomic_add_64`.)
- The driver was unloaded and reloaded between counts, resetting them.

Debugging pattern. Run the regression script with `dmesg -c` cleared beforehand, and `dmesg` after each cycle. The log shows every method invocation; counting the log lines is an alternative to counting the counters, and any difference indicates a bug.

22.8.7 Symptom 7: Hangs During Suspend

A hang during suspend is the worst diagnostic: the kernel is still running (the console still responds to break-to-DDB), but the suspend sequence is stuck in some driver's `DEVICE_SUSPEND`. Break into DDB and `ps` to see which thread is where:

```
db> ps
...  0 myfirst_drain_dma+0x42 myfirst_pci_suspend+0x80 ...
```

Debugging pattern. Identify the hanging thread and the function it is stuck in. Usually it is a `cv_wait` or `cv_timedwait` that never completed, or a `taskqueue_drain` waiting on a task that will not finish.

Fix pattern. Add a timeout to any wait the suspend path does. The `myfirst_drain_dma` function uses `cv_timedwait` with a one-second timeout; a variant that uses `cv_wait` (no timeout) can hang indefinitely. The chapter's implementation always uses timed variants for this reason.

22.8.8 Using DTrace to Trace Suspend and Resume

DTrace is an excellent tool for observing the power-management path at fine granularity without adding print statements. A simple D script that times each call:

```
fbt::device_suspend:entry,
fbt::device_resume:entry
{
    self->ts = timestamp;
    printf("%s: %s %s\n", probefunc,
        args[0] != NULL ? stringof(args[0]->name) : "?",
        args[0] != NULL ? stringof(args[0]->desc) : "?");
}

fbt::device_suspend:return,
```

```
fbt::device_resume: return
/self->ts/
{
    printf("%s: returned %d after %d us\n",
        probefunc, arg1,
        (timestamp - self->ts) / 1000);
    self->ts = 0;
}
```

Save this as `trace-devpower.d` and run with `dtrace -s trace-devpower.d`. Any `devctl suspend` or `acpiconf -s 3` will produce output showing each device's suspend and resume times, and their return values.

For the `myfirst` driver specifically, `fbt::myfirst_pci_suspend:entry` and `fbt::myfirst_pci_resume:entry` are the probes. A D script focused on the driver:

```
fbt::myfirst_pci_suspend:entry {
    self->ts = timestamp;
    printf("myfirst_pci_suspend: entered\n");
    stack();
}

fbt::myfirst_pci_suspend: return
/self->ts/ {
    printf("myfirst_pci_suspend: returned %d after %d us\n",
        arg1, (timestamp - self->ts) / 1000);
    self->ts = 0;
}
```

The `stack()` call prints the call stack at entry, which is useful to confirm that the method is being called from where you expect (the PCI bus's `bus_suspend_child`, for example).

22.8.9 A Note on Logging Discipline

The Chapter 22 code logs generously during suspend and resume: every method logs entry and exit, and each helper logs its own events. That verbosity is helpful during development but annoying in production (every laptop suspend prints half a dozen lines to `dmesg`).

A good production driver exposes a `sysctl` that controls log verbosity:

```
static int myfirst_power_verbose = 1;
SYSCTL_INT(_dev_myfirst, OID_AUTO, power_verbose,
    CTLFLAG_RWTUN, &myfirst_power_verbose, 0,
    "Verbose power-management logging (0=off, 1=on, 2=debug)");
```

And the logging becomes conditional:

```
if (myfirst_power_verbose >= 1)
    device_printf(dev, "suspend: starting\n");
```

A reader who wants to enable debugging on a production system can set `dev.myfirst.power_verbose=2` temporarily, trigger the problem, and reset the variable. The Chapter 22 driver does not implement this tiering; the learning driver logs everything and accepts the noise.

22.8.10 Using the INVARIANTS Kernel for Assertion Coverage

A debug kernel with INVARIANTS compiled in causes KASSERT macros to actually evaluate their conditions and panic on failure. The `myfirst_dma.c` and `myfirst_pci.c` code uses several KASSERTs; the power-management code adds more. For example, the quiesce invariant:

```
static int
myfirst_quiesce(struct myfirst_softc *sc)
{
    /* ... */

    KASSERT(sc->dma_in_flight == false,
        ("myfirst: dma_in_flight still true after drain"));

    return (0);
}
```

On an INVARIANTS kernel, a bug that leaves `dma_in_flight` true causes an immediate panic with a useful message. On a production kernel, the assertion is compiled out and nothing happens. The learning driver deliberately runs on an INVARIANTS kernel to catch this class of bug.

Similarly, the resume path can assert:

```
KASSERT(sc->suspended == true,
    ("myfirst: resume called but not suspended"));
```

This catches a bug where the driver somehow gets resume called without the matching suspend having happened (usually a bug in a parent bus driver, not in the `myfirst` driver itself).

22.8.11 A Debugging Case Study

To bring the patterns together, consider a concrete scenario. The reader writes the Stage 2 suspend, runs a regression cycle, and sees:


```
myfirst0: suspend: starting
myfirst0: drain_dma: timeout waiting for abort
myfirst0: suspend: complete (dma in flight=0, suspended=1)
myfirst0: resume: starting
myfirst0: resume: complete
```

Then:

```
sudo sysctl dev.myfirst.0.dma_test_read=1
# Returns EBUSY after a long delay
```

The user-visible symptom is that the post-resume transfer does not work. The log shows a drain timeout during suspend, which is the first anomaly.

Hypothesis. The DMA engine did not honor the ABORT bit. The driver force-cleared `dma_in_flight`, but the engine is still running; when the user triggers a new transfer, the engine is not ready.

Test. Check the engine's status register before and after the abort:

```
/* In myfirst_drain_dma, after writing ABORT: */
uint32_t pre_status = CSR_READ_4(sc->dev, MYFIRST_REG_DMA_STATUS);
DELAY(100); /* let the engine notice */
uint32_t post_status = CSR_READ_4(sc->dev, MYFIRST_REG_DMA_STATUS);
device_printf(sc->dev, "drain: status %#x -> %#x\n", pre_status, post_status);
```

Running the cycle again produces:

```
myfirst0: drain: status 0x4 -> 0x4
```

Status 0x4 is RUNNING. The engine ignored the ABORT. That points to the simulation backend: the simulated engine might not implement abort, or might do so only when the simulation callout fires.

Fix. Look at the simulation's DMA engine code and verify the abort semantics. In this case, the simulation's engine handles the abort in its callout callback, which does not fire for a few milliseconds. Extend the drain timeout from 1 second (plenty) to... wait, 1 second is plenty for a callout that fires every few milliseconds. The real issue is elsewhere.

Further investigation reveals that the simulation's callout was drained *before* the DMA drain completed. The order in `myfirst_drain_workers` (task first, callout second) was wrong; it should be callout first, task second, because the callout is what drives the abort completion.

Resolution. Reorder the drain:

```
static void
myfirst_drain_workers(struct myfirst_softc *sc)
{
```

```

/*
 * Drain the callout first: it runs the simulated engine's
 * completion logic, and the drain-DMA path waits on that
 * completion. Draining the callout after drain_dma would let
 * drain_dma time out and force-clear the in-flight flag.
 *
 * Wait - actually, drain_dma has already completed by the time
 * we get here, because myfirst_quiesce calls it first. So the
 * order of the two drains inside this function does not matter
 * for that reason. But drain_workers is also called from detach,
 * where drain_dma may not have been called, and the order there
 * does matter.
 */
if (sc->sim != NULL)
    myfirst_sim_drain_dma_callout(sc->sim);

if (sc->rx_vector.has_task)
    taskqueue_drain(taskqueue_thread, &sc->rx_vector.task);
}

```

But wait: by the time `myfirst_drain_workers` is called from `myfirst_quiesce`, `myfirst_drain_dma` has already completed. The drain-dma wait is inside the drain-dma call; the drain-workers call only cleans up residual state. The order inside drain-workers is mostly aesthetic for suspend.

The real fix is earlier: `myfirst_drain_dma` itself should not have timed out. The 1-second timeout should have been plenty. The actual cause is different: perhaps the simulation's callout was not firing because the driver held a sysctl lock that blocked it. Or the writing of the ABORT bit did not reach the simulation because the simulation's MMIO handler was also blocked.

Lesson. Debugging power management issues is iterative. Each symptom suggests a hypothesis; each test narrows it down; the fix is often in a different layer than the one the symptom pointed to. The patience to follow the chain is what distinguishes good power-aware code from code that mostly works.

22.8.12 Wrapping Up Section 7

Section 7 walked through the characteristic failure modes of power-aware drivers: frozen devices, lost interrupts, bad DMA, missed wake events, WITNESS complaints, counter drift, and outright hangs. For each, it showed the typical cause, a debugging pattern that narrows the problem down, and the fix pattern that removes it. It also introduced DTrace for measurement, discussed log discipline, and showed how INVARIANTS and WITNESS catch the class of bug that only shows up under specific conditions.

The debugging discipline in Section 7, like the quiesce discipline in Section 3 and the restore discipline in Section 4, is meant to stay with the reader beyond the `myfirst` driver. Every power-aware driver has some variation of these bugs lurking in its implementation; the patterns above are how to find them before they reach the user.

Section 8 brings Chapter 22 to a close by consolidating the Sections 2 through 7 code into a refactored `myfirst_power.c` file, bumping the version to 1.5-power, adding a `POWER.md` document, and wiring up a final integration test.

22.9 Section 8: Refactoring and Versioning Your Power-Aware Driver

Stages 1 through 3 added the power-management code inline in `myfirst_pci.c`. That was convenient for teaching, because every change appeared next to the attach and detach code the reader already knew. It is less convenient for readability: `myfirst_pci.c` now has attach, detach, three power methods, and several helpers, and the file is long enough that a first-time reader has to scroll to find things.

Stage 4, the final version of the Chapter 22 driver, pulls all of the power-management code out of `myfirst_pci.c` and into a new file pair, `myfirst_power.c` and `myfirst_power.h`. This follows the same pattern as Chapter 20's `myfirst_msix.c` split and Chapter 21's `myfirst_dma.c` split: the new file has a narrow, well-documented API, and the caller in `myfirst_pci.c` uses only that API.

22.9.1 The Target Layout

After Stage 4, the driver's source files are:

- `myfirst.c` - top-level glue, shared state, sysctl tree.
- `myfirst_hw.c`, `myfirst_hw_pci.c` - register-access helpers.
- `myfirst_sim.c` - simulation backend.
- `myfirst_pci.c` - PCI attach, detach, method table, and thin forwarding to subsystem modules.
- `myfirst_intr.c` - single-vector interrupt (Chapter 19 legacy path).
- `myfirst_msix.c` - multi-vector interrupt setup (Chapter 20).
- `myfirst_dma.c` - DMA setup, teardown, transfer (Chapter 21).
- `myfirst_power.c` - power management (Chapter 22, new).
- `cbuf.c` - circular-buffer support.

The new `myfirst_power.h` declares the public API of the power subsystem:

```
#ifndef _MYFIRST_POWER_H_
#define _MYFIRST_POWER_H_

struct myfirst_softc;

int myfirst_power_setup(struct myfirst_softc *sc);
void myfirst_power_teardown(struct myfirst_softc *sc);

int myfirst_power_suspend(struct myfirst_softc *sc);
int myfirst_power_resume(struct myfirst_softc *sc);
int myfirst_power_shutdown(struct myfirst_softc *sc);

#ifdef MYFIRST_ENABLE_RUNTIME_PM
```

```

int myfirst_power_runtime_suspend(struct myfirst_softc *sc);
int myfirst_power_runtime_resume(struct myfirst_softc *sc);
void myfirst_power_mark_active(struct myfirst_softc *sc);
#endif

void myfirst_power_add_sysctls(struct myfirst_softc *sc);

#endif /* _MYFIRST_POWER_H_ */

```

The `_setup` and `_teardown` pair initialise and tear down the subsystem-level state (the callout, the sysctls). The per-transition functions wrap the same logic the Section 3 through Section 5 code built. The runtime-PM functions are compiled only when the build-time flag is defined.

22.9.2 The `myfirst_power.c` File

The new file is about three hundred lines. Its structure mirrors `myfirst_dma.c`: header includes, static helpers, public functions, sysctl handlers, and `_add_sysctls`.

The helpers are the three from Section 3:

- `myfirst_mask_interrupts`
- `myfirst_drain_dma`
- `myfirst_drain_workers`

Plus one from Section 4:

- `myfirst_restore`

And, if runtime PM is enabled, two from Section 5:

- `myfirst_idle_watcher_cb`
- `myfirst_start_idle_watcher`

The public functions `myfirst_power_suspend`, `myfirst_power_resume`, and `myfirst_power_shutdown` become thin wrappers that call the helpers in the right order and update counters. The sysctl handlers expose the policy knobs and the observability counters.

22.9.3 Updating `myfirst_pci.c`

The `myfirst_pci.c` file is now much shorter. Its three power methods each just forward to the power subsystem:

```

static int
myfirst_pci_suspend(device_t dev)
{
    return (myfirst_power_suspend(device_get_softc(dev)));
}

```

```
}

static int
myfirst_pci_resume(device_t dev)
{
    return (myfirst_power_resume(device_get_softc(dev)));
}

static int
myfirst_pci_shutdown(device_t dev)
{
    return (myfirst_power_shutdown(device_get_softc(dev)));
}
```

The method table stays the same as Stage 1 set it up. The three prototypes above are now the only power-related code in `myfirst_pci.c`, apart from the call to `myfirst_power_setup` from `attach` and `myfirst_power_teardown` from `detach`.

The attach path grows one call:

```
static int
myfirst_pci_attach(device_t dev)
{
    /* ... existing attach code ... */

    err = myfirst_power_setup(sc);
    if (err != 0) {
        device_printf(dev, "power setup failed\n");
        /* unwind */
        myfirst_dma_teardown(sc);
        /* ... rest of unwind ... */
        return (err);
    }

    myfirst_power_add_sysctls(sc);

    return (0);
}
```

The detach path grows a matching call:

```
static int
myfirst_pci_detach(device_t dev)
{
```

```

    /* ... existing detach code ... */

    myfirst_power_tearardown(sc);

    /* ... rest of detach ... */

    return (0);
}

```

myfirst_power_setup initialises the saved_intr_mask, the suspended flag, the counters, and (if runtime PM is enabled) the idle watcher callout. myfirst_power_tearardown drains the callout and cleans up any subsystem-level state. The teardown must be done before the DMA teardown because the callout may still reference DMA state.

22.9.4 Updating the Makefile

The new source file goes into the SRCS list, and the version bumps:

```

KMOD=  myfirst
SRCS=  myfirst.c \
        myfirst_hw.c myfirst_hw_pci.c \
        myfirst_sim.c \
        myfirst_pci.c \
        myfirst_intr.c \
        myfirst_msix.c \
        myfirst_dma.c \
        myfirst_power.c \
        cbuf.c

CFLAGS+= -DMYFIRST_VERSION_STRING=\"1.5-power\"

# Optional: enable runtime PM.
# CFLAGS+= -DMYFIRST_ENABLE_RUNTIME_PM

.include <bsd.kmod.mk>

```

The MYFIRST_ENABLE_RUNTIME_PM flag is off by default in Stage 4; the runtime-PM code compiles but is wrapped in #ifdef. A reader who wants to experiment enables the flag at build time.

22.9.5 Writing POWER.md

The Chapter 21 pattern set the precedent: every subsystem gets a markdown document that describes its purpose, its API, its state model, and its testing story. POWER.md is next.

A good POWER.md has these sections:

1. **Purpose:** a paragraph explaining what the subsystem does.
2. **Public API:** a table of function prototypes with one-line descriptions.
3. **State Model:** a diagram or text description of the states and transitions.
4. **Counters and Sysctls:** the read-only and read-write sysctls the subsystem exposes.
5. **Transition Flows:** what happens during each of suspend, resume, shutdown.
6. **Interaction with Other Subsystems:** how power management relates to DMA, interrupts, and the simulation.
7. **Runtime PM (optional):** how runtime PM works and when it is enabled.
8. **Testing:** the regression and stress scripts.
9. **Known Limitations:** what the subsystem does not do yet.
10. **See Also:** cross-references to `bus(9)`, `pci(9)`, and the chapter text.

The full document is in the examples directory (`examples/part-04/ch22-power/stage4-final/POWER.md`); the chapter does not reproduce it inline, but a reader who wants to check the expected structure can open it.

22.9.6 Regression Script

The Stage 4 regression script exercises every path:

```
#!/bin/sh
# ch22-full-regression.sh

set -e

# 1. Basic sanity.
sudo kldload ./myfirst.ko

# 2. One suspend-resume cycle.
sudo sh ./ch22-suspend-resume-cycle.sh

# 3. One hundred cycles in a row.
sudo sh ./ch22-suspend-stress.sh

# 4. A transfer before, during, and after a cycle.
sudo sh ./ch22-transfer-across-cycle.sh

# 5. If runtime PM is enabled, test it.
if sysctl -N dev.myfirst.0.runtime_state >/dev/null 2>&1; then
    sudo sh ./ch22-runtime-pm.sh
fi

# 6. Unload.
sudo kldunload myfirst

echo "FULL REGRESSION PASSED"
```

Each sub-script is a few dozen lines and tests one thing. Running the full regression after each change catches regressions immediately.

22.9.7 Integration With the Existing Regression Tests

The Chapter 21 regression script checked:

- `dma_complete_intrs == dma_complete_tasks` (the task always sees every interrupt).
- `dma_complete_intrs == dma_transfers_write + dma_transfers_read + dma_errors + dma_timeouts`.

The Chapter 22 script adds:

- `power_suspend_count == power_resume_count` (every suspend has a matching resume).
- The suspended flag is 0 outside of a transition.
- After a suspend-resume cycle, the DMA counters still add up to the expected total (no phantom transfers).

The combined regression is the Chapter 22 full script. It exercises DMA, interrupts, MSI-X, and power management together. A driver that passes it is in good shape.

22.9.8 Version History

The driver has now evolved through several versions:

- 1.0 - Chapter 16: MMIO-only driver, simulation backend.
- 1.1 - Chapter 18: PCI attach, real BAR.
- 1.2-intx - Chapter 19: single-vector interrupt with filter+task.
- 1.3-msi - Chapter 20: multi-vector MSI-X with fallback.
- 1.4-dma - Chapter 21: `bus_dma` setup, simulated DMA engine, interrupt-driven completion.
- 1.5-power - Chapter 22: suspend/resume/shutdown, refactored into `myfirst_power.c`, optional run-time PM.

Each version builds on the previous one. A reader who has the Chapter 21 driver working can apply the Chapter 22 changes incrementally and end up at 1.5-power without having to rewrite any earlier code.

22.9.9 A Final Integration Test on Real Hardware

If the reader has access to real hardware (a machine with a working S3 implementation), the Chapter 22 driver can be exercised through a full system suspend:

```
sudo kldload ./myfirst.ko
sudo sh ./ch22-suspend-resume-cycle.sh
sudo acpiconf -s 3
# [laptop sleeps; user opens lid]
# After resume, the DMA test should still work.
sudo sysctl dev.myfirst.0.dma_test_read=1
```


On most platforms where ACPI S3 works, the driver survives the full cycle. The `dmesg` output shows the suspend and resume lines just as `devctl` would trigger, confirming that the same method code runs in both contexts.

If the full-system test fails where the per-device test succeeded, the extra work that system suspend does (ACPI sleep-state transitions, CPU parking, RAM self-refresh) has exposed something the per-device test missed. The usual culprits are device-specific register values that the system's low-power state resets but the per-device D3 does not. A driver that tests only with `devctl` can miss these; a driver that tests with `acpiconf -s 3` at least once before claiming correctness is more reliable.

22.9.10 The Chapter 22 Code in One Place

A compact summary of what the Stage 4 driver added:

- **One new file:** `myfirst_power.c`, about three hundred lines.
- **One new header:** `myfirst_power.h`, about thirty lines.
- **One new markdown document:** `POWER.md`, about two hundred lines.
- **Five new softc fields:** `suspended`, `saved_intr_mask`, `power_suspend_count`, `power_resume_count`, `power_shutdown_count`, plus the runtime-PM fields when that feature is enabled.
- **Three new DEVMETHOD lines:** `device_suspend`, `device_resume`, `device_shutdown`.
- **Three new helper functions:** `myfirst_mask_interrupts`, `myfirst_drain_dma`, `myfirst_drain_workers`.
- **Two new subsystem entry points:** `myfirst_power_setup`, `myfirst_power_teardown`.
- **Three new transition functions:** `myfirst_power_suspend`, `myfirst_power_resume`, `myfirst_power_shutdown`.
- **Six new sysctls:** the counter nodes and the suspended flag.
- **Several new lab scripts:** `cycle`, `stress`, `transfer-across-cycle`, `runtime-PM`.

The overall line increment is about seven hundred lines of code, plus a couple of hundred lines of documentation and script. For the capability the chapter added (a driver that correctly handles every power transition the kernel can throw at it), that is a proportionate investment.

22.9.11 Wrapping Up Section 8

Section 8 closed the Chapter 22 driver's construction by splitting the power code into its own file, bumping the version to 1.5-power, adding a `POWER.md` document, and wiring the final regression test. The pattern was familiar from Chapters 20 and 21: take the inline code, extract it into a subsystem with a small API, document the subsystem, and integrate it into the rest of the driver through function calls rather than direct field access.

The resulting driver is power-aware in every sense the chapter introduced: it handles `DEVICE_SUSPEND`, `DEVICE_RESUME`, and `DEVICE_SHUTDOWN`; it quiesces the device cleanly; it restores state correctly on resume; it optionally implements runtime power management; it exposes its state through sysctls; it has a regression test; and it survives full-system suspend on real hardware when the platform supports it.

22.10 Deep Look: Power Management in /usr/src/sys/dev/re/if_re.c

The Realtek 8169 and compatible gigabit NICs are handled by the `re(4)` driver. It is an informative driver to read for Chapter 22 purposes because it implements the full suspend-resume-shutdown trio with wake-on-LAN support, and its code has been stable enough to represent a canonical FreeBSD pattern. A reader who has worked through Chapter 22 can open `/usr/src/sys/dev/re/if_re.c` and recognise the structure immediately.

Reading this walkthrough. The paired `re_suspend()` and `re_resume()` listings in the subsections below are taken from `/usr/src/sys/dev/re/if_re.c`, and the method-table excerpt abbreviates the full `re_methods[]` array with a `/* ... other methods ... */` comment so the three power-related `DEVMETHOD` entries stand out. We kept the signatures, the lock-acquire and lock-release pattern, and the order of device-specific calls (`re_stop`, `re_setwol`, `re_clrwol`, `re_init_locked`) intact; the real method table has many more entries, and the surrounding file carries the helper implementations. Every symbol the listings name is a real FreeBSD identifier in `if_re.c` that you can find with a symbol search.

22.10.1 The Method Table

The `re(4)` driver's method table includes the three power methods near the top:

```
static device_method_t re_methods[] = {
    DEVMETHOD(device_probe,      re_probe),
    DEVMETHOD(device_attach,     re_attach),
    DEVMETHOD(device_detach,     re_detach),
    DEVMETHOD(device_suspend,    re_suspend),
    DEVMETHOD(device_resume,     re_resume),
    DEVMETHOD(device_shutdown,   re_shutdown),
    /* ... other methods ... */
};
```

This is exactly the pattern Chapter 22 teaches. The `myfirst` driver's method table looks the same.

22.10.2 re_suspend

The suspend function is about a dozen lines:

```
static int
re_suspend(device_t dev)
{
    struct rl_softc *sc;

    sc = device_get_softc(dev);

    RL_LOCK(sc);
    re_stop(sc);
```

```

    re_setwol(sc);
    sc->suspended = 1;
    RL_UNLOCK(sc);

    return (0);
}

```

Three calls do the work: `re_stop` quiesces the NIC (disables interrupts, halts DMA, stops the RX and TX engines), `re_setwol` programs the wake-on-LAN logic and calls `pci_enable_pme` if WoL is enabled, and `sc->suspended = 1` sets the softc flag.

Compare to `myfirst_power_suspend`:

```

int
myfirst_power_suspend(struct myfirst_softc *sc)
{
    int err;

    device_printf(sc->dev, "suspend: starting\n");
    err = myfirst_quiesce(sc);
    /* ... error handling ... */
    atomic_add_64(&sc->power_suspend_count, 1);
    return (0);
}

```

The structure is identical. `re_stop` and `re_setwol` together are the equivalent of `myfirst_quiesce`; the chapter's driver does not have wake-on-X, so there is no analogue of `re_setwol`.

22.10.3 re_resume

The resume function is about thirty lines:

```

static int
re_resume(device_t dev)
{
    struct rl_softc *sc;
    if_t ifp;

    sc = device_get_softc(dev);

    RL_LOCK(sc);

    ifp = sc->rl_ifp;
    /* Take controller out of sleep mode. */
}

```

```

    if ((sc->rl_flags & RL_FLAG_MACSLEEP) != 0) {
        if ((CSR_READ_1(sc, RL_MACDBG) & 0x80) == 0x80)
            CSR_WRITE_1(sc, RL_GPIO,
                CSR_READ_1(sc, RL_GPIO) | 0x01);
    }

    /*
     * Clear WOL matching such that normal Rx filtering
     * wouldn't interfere with WOL patterns.
     */
    re_clrwol(sc);

    /* reinitialize interface if necessary */
    if (if_getflags(ifp) & IFF_UP)
        re_init_locked(sc);

    sc->suspended = 0;
    RL_UNLOCK(sc);

    return (0);
}

```

The steps map cleanly to Chapter 22's discipline:

1. **Take the controller out of sleep mode** (MAC sleep bit on some Realtek parts). This is a device-specific restore step.
2. **Clear any WOL patterns** via `re_clrwol`, which reverses what `re_setwol` did. This also calls `pci_clear_pme` implicitly through the clear.
3. **Re-initialise the interface** if it was up before suspend. `re_init_locked` is the same function attach calls to bring up the NIC; it reprograms the MAC, resets the descriptor rings, enables interrupts, and starts the DMA engines.
4. **Clear the suspended flag** under the lock.

The `myfirst_power_resume` equivalent:

```

int
myfirst_power_resume(struct myfirst_softc *sc)
{
    int err;

    device_printf(sc->dev, "resume: starting\n");
    err = myfirst_restore(sc);
    /* ... */
    atomic_add_64(&sc->power_resume_count, 1);
}

```

```
        return (0);  
    }
```

Again the structure is identical. `myfirst_restore` corresponds to the combination of the MAC-sleep exit, `re_clrwol`, `re_init_locked`, and the flag clear.

22.10.4 re_shutdown

The shutdown function is:

```
static int  
re_shutdown(device_t dev)  
{  
    struct rl_softc *sc;  
  
    sc = device_get_softc(dev);  
  
    RL_LOCK(sc);  
    re_stop(sc);  
    /*  
     * Mark interface as down since otherwise we will panic if  
     * interrupt comes in later on, which can happen in some  
     * cases.  
     */  
    if_setflagbits(sc->rl_ifp, 0, IFF_UP);  
    re_setwol(sc);  
    RL_UNLOCK(sc);  
  
    return (0);  
}
```

Similar to `re_suspend`, plus the interface-flag clear (shutdown is final; marking the interface down prevents spurious activity). The pattern is nearly identical; `re_shutdown` is essentially a more defensive version of `re_suspend`.

22.10.5 re_setwol

The wake-on-LAN setup is worth looking at because it shows how a real driver calls the PCI PM APIs:

```
static void  
re_setwol(struct rl_softc *sc)  
{  
    if_t ifp;
```

```
uint8_t v;

RL_LOCK_ASSERT(sc);

if (!pci_has_pm(sc->rl_dev))
    return;

/* ... programs device-specific wake registers ... */

/* Request PME if WOL is requested. */
if ((if_getcapenable(ifp) & IFCAP_WOL) != 0)
    pci_enable_pme(sc->rl_dev);
}
```

Three key patterns appear here that are worth copying into any power-aware driver that supports wake-on-X:

1. **pci_has_pm(dev) guard.** The function returns early if the device does not support power management. This prevents writes to registers that do not exist.
2. **Device-specific wake programming.** The bulk of the function writes Realtek-specific registers through CSR_WRITE_1. A driver for a different device would write different registers, but the placement (inside the suspend path, before pci_enable_pme) is the same.
3. **Conditional pci_enable_pme.** Only enable PME# if the user has actually asked for wake-on-X. If the user has not, the function still sets the relevant configuration bits (for consistency with the driver's interface capabilities) but does not call pci_enable_pme.

The inverse is re_clrwol:

```
static void
re_clrwol(struct rl_softc *sc)
{
    uint8_t v;

    RL_LOCK_ASSERT(sc);

    if (!pci_has_pm(sc->rl_dev))
        return;

    /* ... clears the wake-related config bits ... */
}
```

Note that re_clrwol does not explicitly call pci_clear_pme; the PCI layer's pci_resume_child has already called it before the driver's DEVICE_RESUME. re_clrwol is responsible for undoing the driver-visible side of WoL configuration, not the kernel-visible PME status.

22.10.6 What the Deep Look Shows

The Realtek driver is more complex than `myfirst` by every measure (more registers, more state, more device variants), and yet its power-management discipline is less complex. That is because complexity of the *device* does not map one-to-one to complexity of the *power-management code*. Chapter 22's discipline scales down as well as it scales up: a simple device has a simple power path; a complex device has a modestly more complex power path. The structure is the same.

A reader who has finished Chapter 22 can now open `if_re.c`, recognise every function and every pattern, and understand why each exists. That comprehension transfers: the same recognition applies to `if_xl.c`, `virtio_blk.c`, and hundreds of other FreeBSD drivers. Chapter 22 is not teaching a `myfirst`-specific API; it is teaching the FreeBSD power-management idiom, and the `myfirst` driver is the vehicle that made it concrete.

22.11 Deep Look: Simpler Patterns in `if_xl.c` and `virtio_blk.c`

For contrast, two other FreeBSD drivers implement power management in even simpler ways.

22.11.1 `if_xl.c`: Shutdown Calls Suspend

The 3Com EtherLink III driver in `/usr/src/sys/dev/xl/if_xl.c` has the minimal three-method setup:

```
static int
xl_shutdown(device_t dev)
{
    return (xl_suspend(dev));
}

static int
xl_suspend(device_t dev)
{
    struct xl_softc *sc;

    sc = device_get_softc(dev);

    XL_LOCK(sc);
    xl_stop(sc);
    xl_setwol(sc);
    XL_UNLOCK(sc);

    return (0);
}

static int
xl_resume(device_t dev)
```

```
{  
    struct xl_softc *sc;  
    if_t ifp;  
  
    sc = device_get_softc(dev);  
    ifp = sc->xl_ifp;  
  
    XL_LOCK(sc);  
  
    if (if_getflags(ifp) & IFF_UP) {  
        if_setdrvflagbits(ifp, 0, IFF_DRV_RUNNING);  
        xl_init_locked(sc);  
    }  
  
    XL_UNLOCK(sc);  
  
    return (0);  
}
```

Two things stand out:

1. `xl_shutdown` is one line: it just calls `xl_suspend`. For this driver, shutdown and suspend do the same work, and the code does not need two copies.
2. There is no suspended flag in the softc. The driver assumes the normal lifecycle of attach → run → suspend → resume, and uses the `IFF_DRV_RUNNING` flag (which the TX path already checks) as the equivalent. This is a perfectly valid approach for a NIC whose main user-visible state is the interface's running state.

For the `myfirst` driver, the explicit suspended flag is preferred because the driver has no natural equivalent of `IFF_DRV_RUNNING`. A NIC driver can reuse what it already has; a learning driver declares what it needs.

22.11.2 `virtio_blk.c`: Minimal Quiesce

The `virtio` block driver in `/usr/src/sys/dev/virtio/block/virtio_blk.c` has an even shorter suspend path:

```
static int  
vtblk_suspend(device_t dev)  
{  
    struct vtblk_softc *sc;  
    int error;  
  
    sc = device_get_softc(dev);
```



```
    VTBLK_LOCK(sc);
    sc->vtblk_flags |= VTBLK_FLAG_SUSPEND;
    /* XXX BMV: virtio_stop(), etc needed here? */
    error = vtblk_quiesce(sc);
    if (error)
        sc->vtblk_flags &= ~VTBLK_FLAG_SUSPEND;
    VTBLK_UNLOCK(sc);

    return (error);
}

static int
vtblk_resume(device_t dev)
{
    struct vtblk_softc *sc;

    sc = device_get_softc(dev);

    VTBLK_LOCK(sc);
    sc->vtblk_flags &= ~VTBLK_FLAG_SUSPEND;
    vtblk_startio(sc);
    VTBLK_UNLOCK(sc);

    return (0);
}
```

The comment `/* XXX BMV: virtio_stop(), etc needed here? */` is an honest acknowledgement that the author was not sure how thorough the quiesce should be. The existing code sets a flag, waits for the queue to drain (that is what `vtblk_quiesce` does), and returns. On resume, it clears the flag and restarts I/O.

For a virtio block device, this is enough because the virtio host (the hypervisor) implements its own quiesce when the guest says it is suspending. The driver only needs to stop submitting new requests; the host deals with the rest.

This shows an important pattern: **the driver's quiesce depth depends on how much of the hardware's state is the driver's responsibility**. A bare-metal driver (like `re(4)`) has to program hardware registers carefully because the hardware has no other ally. A virtio driver has the hypervisor as an ally; the host can handle most of the state for the guest. The `myfirst` driver, running on a simulated backend, is in a similar position: the simulation is an ally, and the driver's quiesce can be correspondingly simpler.

22.11.3 What the Comparison Shows

Reading multiple drivers' power-management code side by side is one of the best ways to build fluency. Each driver adapts the Chapter 22 pattern to its context: `re(4)` handles wake-on-LAN, `xl(4)` reuses `xl_shutdown`

= `xl_suspend`, `virtio_blk(4)` trusts the hypervisor. The common thread is the structure: stop activity, save state, flag suspended, return 0 from suspend; on resume, clear flag, restore state, restart activity, return 0.

A reader who has Chapter 22 in memory can open any FreeBSD driver, find its `device_suspend` and `device_resume` in the method table, and read the two functions. Within a few minutes the driver's power policy is clear. That skill transfers to every driver the reader will ever work on; it is the single most useful takeaway from the chapter.

22.12 Deep Look: ACPI Sleep States in More Detail

Section 1 introduced the ACPI S-states as a list. It is worth revisiting them with the driver's point of view in focus, because the driver sees slightly different things depending on which S-state the kernel is entering.

22.12.1 S0: Working

S0 is the state the reader has worked in throughout Chapters 16 to 21. The CPU is executing, RAM is refreshed, the PCIe links are up. From the driver's point of view, S0 is continuous; everything is normal.

Within S0, however, there can still be fine-grained power transitions. The CPU may enter idle states (C1, C2, C3, etc.) between scheduler ticks. The PCIe link may enter L0s or L1 based on ASPM. Devices may enter D3 based on runtime PM. None of these require the driver to do anything beyond its own runtime-PM logic; they are transparent.

22.12.2 S1: Standby

S1 is historically the lightest sleep state. The CPU stops executing but its registers are preserved; RAM stays powered; device power stays at D0 or D1. Wake latency is fast (under a second).

On modern hardware, S1 is rarely supported. The platform's BIOS advertises only S3 and deeper. If the platform does advertise S1 and the user enters it, the driver's `DEVICE_SUSPEND` is still called; the driver does its usual quiesce. The difference is that the PCI layer typically does not transition to D3 for S1 (because the bus stays powered), so the device stays in D0 through the transition. The driver's save and restore are largely unused.

A driver that supports S1 cleanly also supports S3, because the driver-side work is a subset. No driver written for Chapter 22 needs to treat S1 specially.

22.12.3 S2: Reserved

S2 is defined in the ACPI specification but almost never implemented. A driver can safely ignore it; FreeBSD's ACPI layer treats S2 as S1 or S3 depending on platform support.

22.12.4 S3: Suspend to RAM

S3 is the canonical sleep state Chapter 22 targets. When the user enters S3:

1. The kernel's suspend sequence traverses the device tree, calling `DEVICE_SUSPEND` on each driver.

2. The PCI layer's `pci_suspend_child` caches configuration space for each PCI device.
3. The PCI layer transitions each PCI device to D3hot.
4. Higher-level subsystems (ACPI, the CPU's idle machinery) enter their own sleep states.
5. The CPU's context is saved to RAM; the CPU halts.
6. RAM enters self-refresh; the memory controller maintains the contents with minimal power.
7. The platform's wake circuitry is armed: the power button, lid switch, and any configured wake sources.
8. The system waits for a wake event.

When a wake event arrives:

1. The CPU resumes; its context is restored from RAM.
2. Higher-level subsystems resume.
3. The PCI layer walks the device tree and calls `pci_resume_child` for each device.
4. Each device is transitioned to D0; its configuration is restored; pending PME# is cleared.
5. Each driver's `DEVICE_RESUME` is called.
6. User space unfreezes.

The driver sees only steps 1 (suspend) and 5 (resume) of each sequence. The rest is kernel and platform machinery.

A subtle point: during S3, RAM is refreshed but the kernel is not running. This means any kernel-side state (the softc, the DMA buffer, the pending tasks) survives S3 unchanged. The only thing that may be lost is hardware state: configuration registers in the device may be reset; BAR-mapped registers may return to default values. The driver's job on resume is to re-program the hardware from the preserved kernel state.

22.12.5 S4: Suspend to Disk (Hibernate)

S4 is the “hibernate” state. The kernel writes the full contents of RAM to a disk image, then enters S5. On wake, the platform boots, the kernel reads the image back, and the system continues from where it left off.

On FreeBSD, S4 has historically been partial. The kernel can produce the hibernation image on some platforms, but the restore path is not as mature as Linux's. For driver purposes, S4 is the same as S3: the `DEVICE_SUSPEND` and `DEVICE_RESUME` methods are called; the driver's quiesce and restore paths work without change. The extra platform-level work (writing the image) is transparent.

The one difference the driver might notice is that after S4 resume, the PCI configuration space is always restored from scratch (the platform has fully rebooted), so even if the driver were relying on `hw.pci.do_power_suspend` being 0 to keep the device in D0, after S4 the device will still have been through a full power cycle. This matters only for drivers that do platform-specific tricks during suspend; most drivers are oblivious.

22.12.6 S5: Soft Off

S5 is system power-off. The power button, the battery (if any), and the wake circuitry still receive power; everything else is off.

From the driver's point of view, S5 looks like a shutdown: `DEVICE_SHUTDOWN` is called (not `DEVICE_SUSPEND`), the driver places the device in a safe state for power-off, and the system halts. There is no resume corresponding to S5; if the user presses the power button, the system boots from scratch.

Shutdown is not a power transition in the reversible sense; it is a termination. The driver's `DEVICE_SHUTDOWN` method is called once, and the driver does not expect to run again until the next boot. The chapter's `my-first_power_shutdown` handles this correctly by quiescing the device (same as suspend) and not trying to save any state (because there is no resume to save for).

22.12.7 Observing Which States the Platform Supports

On any FreeBSD 14.3 system with ACPI, the supported states are exposed through a `sysctl`:

```
sysctl hw.acpi.supported_sleep_state
```

Typical outputs:

- A modern laptop: S3 S4 S5
- A server: S5 (suspend not supported on many server platforms)
- A VM on bhyve: varies; usually S5 only
- A VM on QEMU/KVM with `-machine q35`: often S3 S4 S5

If a driver is meant to work on a specific platform, the supported-state list tells you which transitions you need to test. A driver that only runs on servers does not need S3 testing; a driver meant for laptops does.

22.12.8 What to Test

For Chapter 22's purposes, the minimum test is:

- `devctl suspend / devctl resume`: always possible; tests the driver-side code path.
- `acpiconf -s 3` (if supported): tests the full system suspend.
- System shutdown (`shutdown -p now`): tests the `DEVICE_SHUTDOWN` method.

S4 and runtime PM are optional; they exercise less-used code paths. A driver that passes the minimum test on a platform that supports S3 is in good shape; extensions are icing.

22.12.9 Mapping Sleep States to the Driver's Methods

A compact table of which `kobj` method is called for each transition:

Transition	Method	Driver Action
S0 → S1	<code>DEVICE_SUSPEND</code>	Quiesce; save state
S0 → S3	<code>DEVICE_SUSPEND</code>	Quiesce; save state (device likely goes to D3)
S0 → S4	<code>DEVICE_SUSPEND</code>	Quiesce; save state (followed by hibernate)
S0 → S5 (shutdown)	<code>DEVICE_SHUTDOWN</code>	Quiesce; leave in safe state for power-off
S1/S3 → S0	<code>DEVICE_RESUME</code>	Restore state; unmask interrupts
S4 → S0 (resume)	(attach from boot)	Normal attach, because the kernel booted fresh

Transition	Method	Driver Action
devctl suspend	DEVICE_SUSPEND	Quiesce; save state (device goes to D3)
devctl resume	DEVICE_RESUME	Restore state; unmask interrupts

The driver does not distinguish S1, S3, and S4 from its own code; it always does the same work. The differences are at the platform and kernel levels. That uniformity is what makes the pattern scalable: one suspend path, one resume path, multiple contexts.

22.13 Deep Look: PCIe Link States and ASPM in Action

Section 1 sketched the PCIe link states (L0, L0s, L1, L1.1, L1.2, L2). It is worth seeing how they behave in practice, because understanding them helps the driver developer interpret latency measurements and power observations.

22.13.1 Why the Link Has Its Own States

A PCIe link is a pair of high-speed differential lanes between two endpoints (root complex and device, or root complex and switch). Each lane has a transmitter and a receiver; each lane's transmitter consumes power to keep the channel in a known state. When traffic is low, the transmitters can be turned off in various degrees, and the link can be re-established quickly when traffic resumes. The L-states describe those degrees.

The link's state is separate from the device's D-state. A device in D0 can have its link in L1 (the link is idle; the device is not transmitting or receiving). A device in D3 has its link in L2 or similar (the link is off). A device in D0 with a busy link is in L0.

22.13.2 L0: Active

L0 is the normal operating state. Both sides of the link are active; data can flow in either direction; latency is at its minimum (a few hundred nanoseconds round-trip on a modern PCIe).

When a DMA transfer is running or an MMIO read is pending, the link is in L0. The device's own logic and the PCIe host bridge both require L0 for the transaction.

22.13.3 L0s: Transmitter Standby

L0s is a low-power state where one side of the link's transmitter is turned off. The receiver stays on; the link can be brought back to L0 in under a microsecond.

L0s is entered automatically by the link logic when no traffic has been sent for a few microseconds. The platform's PCIe host bridge and the device's PCIe interface cooperate: when the transmit FIFO is empty and ASPM is enabled, the transmitter goes off. When new traffic arrives, the transmitter comes back on.

L0s is "asymmetric": each side independently enters and exits the state. A device's transmitter can be in L0s while the root complex's transmitter is in L0. This is useful because traffic is typically bursty: the CPU sends a

DMA trigger, then does not send anything else for a while; the CPU's transmitter enters L0s quickly, while the device's transmitter stays in L0 because it is actively sending the DMA response.

22.13.4 L1: Both Sides Standby

L1 is a deeper state where both transmitters are off. Neither side can send anything until the link is brought back to L0; the latency is measured in microseconds (5 to 65, depending on platform).

L1 is entered after a longer idle period than L0s. The exact threshold is configurable through ASPM settings; typical values are tens of microseconds of inactivity. L1 saves more power than L0s but costs more to exit.

22.13.5 L1.1 and L1.2: Deeper L1 Sub-States

PCIe 3.0 and later define sub-states of L1 that turn off additional parts of the physical layer. L1.1 (also called "L1 PM Substate 1") keeps the clock running but turns off more circuitry; L1.2 turns off the clock as well. The wake latencies increase (tens of microseconds for L1.1; hundreds for L1.2), but the idle power draws decrease.

Most modern laptops use L1.1 and L1.2 aggressively to extend battery life. A laptop that stays in L1.2 most of the idle time can have PCIe power draw in the single-digit milliwatts, compared to hundreds of milliwatts in L0.

22.13.6 L2: Near-Off

L2 is the state the link enters when the device is in D3cold. The link is effectively off; re-establishing it requires a full link-training sequence (tens of milliseconds). L2 is entered as part of the full-system suspend sequence; the driver does not manage it directly.

22.13.7 Who Controls ASPM

ASPM is a per-link feature configured through the PCIe Link Capability and Link Control registers in both the root complex and the device. The configuration specifies:

- Whether L0s is enabled (one-bit field).
- Whether L1 is enabled (one-bit field).
- The exit latency thresholds the platform considers acceptable.

On FreeBSD, ASPM is usually controlled by the platform firmware through ACPI's `_OSC` method. The firmware tells the OS which capabilities to manage; if the firmware keeps ASPM control, the OS does not touch it. If the firmware hands over control, the OS may enable or disable ASPM per link based on policy.

For Chapter 22's `myfirst` driver, ASPM is the platform's job. The driver does not configure ASPM; it does not need to know whether the link is in L0 or L1 at any moment. The link's state is invisible to the driver from a functional standpoint (latency is the only observable effect).

22.13.8 When ASPM Matters to the Driver

There are specific situations where a driver does have to worry about ASPM:

1. **Known errata.** Some PCIe devices have bugs in their ASPM implementation that cause the link to wedge or produce corrupted transactions. The driver may need to explicitly disable ASPM for those devices. The kernel provides the PCIe Link Control register access through `pcie_read_config` and `pcie_write_config` for this purpose.
2. **Latency-sensitive devices.** A real-time audio or video device may not tolerate the microsecond-scale latency of L1. The driver may disable L1 while keeping L0s enabled.
3. **Power-sensitive devices.** A battery-powered device may want L1.2 always enabled. The driver may force L1.2 if the platform's default is less aggressive.

For the `myfirst` driver, none of these apply. The simulated device does not have a link at all; the real PCIe link (if any) is handled by the platform. The chapter mentions ASPM for completeness and moves on.

22.13.9 Observing Link States

On a system where the platform supports ASPM observation, the link state is exposed through `pciconf -lvbc`:

```
pciconf -lvbc | grep -A 20 myfirst
```

Look for lines like:

```
cap 10[ac] = PCI-Express 2 endpoint max data 128(512) FLR NS
            link x1(x1) speed 5.0(5.0)
            ASPM disabled(L0s/L1)
            exit latency L0s 1us/<1us L1 8us/8us
            slot 0
```

The “ASPM disabled” on this line says ASPM is not currently active. “disabled(L0s/L1)” says the device supports both L0s and L1 but neither is enabled. On a system with aggressive ASPM, the line would read “ASPM L1” or similar.

The exit latencies tell the driver how long the transition back to L0 takes; a latency-sensitive driver can decide whether L1 is tolerable by looking at this number.

22.13.10 Link State and Power Consumption

A rough table of PCIe power draws (typical values; actual depend on implementation):

State	Power (x1 link)	Exit Latency
L0	100-200 mW	0
L0s	50-100 mW	<1 μ s
L1	10-30 mW	5-65 μ s
L1.1	1-5 mW	10-100 μ s
L1.2	<1 mW	50-500 μ s
L2	near 0	1-100 ms

For a laptop with a dozen PCIe links all in L1.2 during idle, the aggregate savings relative to all-L0 can be in the watts. For a server with high-throughput links always in L0, ASPM is disabled and the power saving is zero.

Chapter 22 does not implement ASPM for `myfirst`. The chapter mentions it because understanding the link state machine is part of understanding the full power-management picture. A reader who later works on a driver with known ASPM errata will know where to look.

22.14 Deep Look: Wake Sources Explained

Wake sources are the mechanisms that bring a suspended system or device back to active. Chapter 1 mentioned them briefly; this deeper look walks through the most common ones.

22.14.1 PME# on PCIe

The PCI spec defines the **PME#** signal (Power Management Event). When asserted, it tells the upstream root complex that the device has an event worth waking for. The root complex converts **PME#** into an ACPI GPE or interrupt, which the kernel handles.

A device that supports **PME#** has a PCI power-management capability (checked via `pci_has_pm`). The capability's control register includes:

- **PME_En** (bit 8): enable **PME#** generation.
- **PME_Status** (bit 15): set by the device when **PME#** is raised, cleared by software.
- **PME_Support** (read-only, bits 11-15 in PMC register): which D-states the device can raise **PME#** from (D0, D1, D2, D3hot, D3cold).

The driver's job is to set **PME_En** at the right time (usually before suspend) and to clear **PME_Status** at the right time (usually after resume). The `pci_enable_pme(dev)` and `pci_clear_pme(dev)` helpers do both jobs.

On a typical laptop, the root complex routes **PME#** to an ACPI GPE, which the kernel's ACPI driver picks up as a wake event. The chain looks like:

```
device asserts PME#
→ root complex receives PME
→ root complex sets GPE status bit
```


- ACPI hardware interrupts CPU
- kernel wakes from S3
- kernel's ACPI driver services the GPE
- eventually: `DEVICE_RESUME` on the device that woke

The whole chain takes one to three seconds. The driver's role is minimal: it enabled `PME#` before suspend, and it will clear `PME_Status` after resume. Everything else is platform.

22.14.2 USB Remote Wakeup

USB has its own wake mechanism called “remote wakeup”. A USB device requests wake capability through its standard descriptor; the host controller enables the capability at enumeration time; when the device asserts a resume signal on its upstream port, the host controller propagates it.

From a FreeBSD driver perspective, USB remote wakeup is almost entirely handled by the USB host controller driver (`xhci`, `ohci`, `uhci`). Individual USB device drivers (for keyboards, storage, audio, etc.) participate through the USB framework's suspend and resume callbacks, but they do not deal with `PME#` directly. The USB host controller's own `PME#` is what actually wakes the system.

For Chapter 22 purposes, USB wake is a black box that works through the USB host controller driver. A reader who eventually writes a USB device driver will learn the framework's conventions then.

22.14.3 GPIO-Based Wake on Embedded Platforms

On embedded platforms (arm64, RISC-V), wake sources are typically GPIO pins connected to the SoC's wake logic. The device tree describes which pins are wake sources via `wakeup-source` properties and `interrupts-extended` pointing to the wake controller.

FreeBSD's GPIO intr framework handles these. A device driver whose hardware is wake-capable reads the device-tree `wakeup-source` property during attach, registers the GPIO as a wake source with the framework, and the framework does the rest. The mechanism is very different from PCIe `PME#`, but the driver-side API (mark wake enabled, clear wake status) is conceptually similar.

Chapter 22 does not exercise GPIO wake; the `myfirst` driver is a PCI device. Part 7 revisits embedded platforms and covers the GPIO path in detail.

22.14.4 Wake on LAN (WoL)

Wake on LAN is a specific implementation pattern for a network controller. The controller watches incoming packets for a “magic packet” (a specific pattern containing the controller's MAC address repeated many times) or for user-configured patterns. When a match is detected, the controller asserts `PME#` upstream.

From the driver's perspective, WoL requires:

1. Configuring the NIC's wake logic (magic-packet filter, pattern filters) before suspend.
2. Enabling `PME#` via `pci_enable_pme`.

3. On resume, disabling the wake logic (because normal packet processing would otherwise be influenced by the filters).

The `re(4)` driver's `re_setwol` is the canonical FreeBSD example. A reader building a NIC driver copies its structure and adapts the device-specific register programming.

22.14.5 Wake on Lid, Power Button, etc.

The laptop's lid switch, power button, keyboard (in some cases), and other platform inputs are wired to the platform's wake logic through ACPI. The ACPI driver handles the wake; individual device drivers are not involved.

The ACPI `_PRW` method on a device's object in the ACPI namespace declares which GPE that device's wake event uses. The OS reads `_PRW` during boot to configure the wake routing. The `myfirst` driver, as a simple PCI endpoint with no platform-specific wake source, does not have a `_PRW` method; its wake capability (if any) is purely through PME#.

22.14.6 When the Driver Must Enable Wake

A simple heuristic: the driver must enable wake if the user has asked for it (through an interface capability flag like `IFCAP_WOL` for NICs) and the hardware supports it (`pci_has_pm` returns true, the device's own wake logic is operational). Otherwise, the driver leaves wake disabled.

A driver that enables wake for every device by default wastes platform power; the wake circuitry and PME# routing cost a few milliwatts continuously. A driver that never enables wake frustrates users who want their laptop to wake on a network packet. The policy is “enable only when asked”.

FreeBSD's interface capabilities (set via `ifconfig em0 wol wol_magic`) are the standard way users express the desire. The NIC driver reads the flags and configures WoL accordingly.

22.14.7 Testing Wake Sources

Testing wake is harder than testing suspend and resume, because testing wake requires the system to actually sleep and then an external event to wake it. Common approaches:

- **Magic packet from another machine.** Send a WoL magic packet to the suspended machine's MAC address. If WoL is working, the machine wakes in a few seconds.
- **Lid switch.** Close the lid, wait, open the lid. If the platform's wake routing is working, the machine wakes on open.
- **Power button.** Press the power button briefly while suspended. The machine should wake.

For a learning driver like `myfirst`, there is no meaningful wake source to test against. The chapter mentions wake mechanics for pedagogical completeness, not because the driver exercises them.

22.15 Deep Look: The `hw.pci.do_power_suspend` Tunable

One of the most important tunables for power-management debugging is `hw.pci.do_power_suspend`. It controls whether the PCI layer automatically transitions devices to D3 during system suspend. Understanding what it does and when to change it is worth a dedicated look.

22.15.1 What the Default Does

With `hw.pci.do_power_suspend=1` (the default), the PCI layer's `pci_suspend_child` helper, after calling the driver's `DEVICE_SUSPEND`, transitions the device to D3hot by calling `pci_set_power_child(dev, child, PCI_POWERSTATE_D3)`. On resume, `pci_resume_child` transitions back to D0.

This is the “power-save” mode. A device that supports D3 uses its lowest-power idle state during suspend. A laptop benefits because battery life during sleep is extended; a device that can sleep at a few milliwatts instead of a few hundred is worth the extra D-state transition.

22.15.2 What `hw.pci.do_power_suspend=0` Does

With the tunable set to 0, the PCI layer does not transition the device to D3. The device stays in D0 throughout the suspend. The driver's `DEVICE_SUSPEND` runs; the driver quiesces activity; the device stays powered.

From a power-saving perspective, this is worse: the device continues to draw its D0 power budget during sleep. From a correctness perspective, it can be better for some devices:

- A device with broken D3 implementation may misbehave when transitioned. Staying in D0 avoids the transition bug.
- A device whose context is expensive to save and restore may prefer to stay in D0 during a short suspend. If the suspend is only a few seconds, the context-save cost exceeds the power-saving benefit.
- A device that is critical to the machine's core function (a console keyboard, for example) may need to stay alert even during suspend.

22.15.3 When to Change It

For development and debugging, setting `hw.pci.do_power_suspend=0` can isolate bugs:

- If a resume bug appears only with the tunable at 1, the bug is in the D3-to-D0 transition (either in the PCI layer's config restore, or in the driver's handling of a device that has been reset).
- If a resume bug appears with the tunable at 0 as well, the bug is in the driver's `DEVICE_SUSPEND` or `DEVICE_RESUME` code, not in the D-state machinery.

For production, the default (1) is almost always right. Changing it globally affects every PCI device on the system; a better approach is a per-device override if one is needed, which typically lives in the driver itself.

22.15.4 Verifying the Tunable Is in Effect

A quick way to verify is to check the device's power state with `pciconf` before and after a suspend:

```
# Before suspend (device should be in D0):
pciconf -lvbc | grep -A 5 myfirst

# With hw.pci.do_power_suspend=1 (default):
sudo devctl suspend myfirst0
pciconf -lvbc | grep -A 5 myfirst
# "powerspec" should show D3

# With hw.pci.do_power_suspend=0:
sudo sysctl hw.pci.do_power_suspend=0
sudo devctl resume myfirst0
sudo devctl suspend myfirst0
pciconf -lvbc | grep -A 5 myfirst
# "powerspec" should show D0

# Reset to default.
sudo sysctl hw.pci.do_power_suspend=1
sudo devctl resume myfirst0
```

The powerspec line in `pciconf -lvbc` output shows the current power state. Watching it change between D0 and D3 confirms the automatic transition is happening.

22.15.5 Interaction with pci_save_state

When `hw.pci.do_power_suspend` is 1, the PCI layer automatically calls `pci_cfg_save` before transitioning to D3. When it is 0, the PCI layer does not call `pci_cfg_save`.

This has a subtle implication: if the driver wants to save configuration explicitly in the 0 case, it must call `pci_save_state` itself. The Chapter 22 pattern assumes the default (1) and does not call `pci_save_state` explicitly; a driver that wants to support both modes would need additional logic.

22.15.6 Does the Tunable Affect System Suspend or devctl suspend?

Both. `pci_suspend_child` is called for both `acpiconf -s 3` and `devctl suspend`, and the tunable gates the D-state transition in both cases. A reader debugging with `devctl suspend` will see the same behavior as with a full system suspend, modulo the other platform work (CPU park, ACPI sleep state entry).

22.15.7 A Concrete Debugging Scenario

Suppose the `myfirst` driver's resume fails intermittently: sometimes it works, sometimes `dma_test_read` after resume returns EIO. The counters are consistent (suspend count = resume count), the logs show both methods ran, but the post-resume DMA fails.

Hypothesis 1. The D3-to-D0 transition is producing an inconsistent device state. Verify by setting `hw.pci.do_power_suspend=0` and retrying.

If the bug disappears with the tunable at 0, the D-state machinery is involved. The fix might be in the driver's resume path (add a delay after the transition to let the device stabilise), in the PCI layer's config restore, or in the device itself.

Hypothesis 2. The bug is in the driver's own suspend/resume code, independent of D3. Verify by setting the tunable to 0 and retrying.

If the bug persists with the tunable at 0, the driver's code is the problem. The D3 transition is innocent.

This kind of bisection is common in power-management debugging. The tunable is the tool that lets you isolate the variable.

22.16 Deep Look: DEVICE_QUIESCE and When You Need It

Section 2 briefly mentioned `DEVICE_QUIESCE` as the third power-management method alongside `DEVICE_SUSPEND` and `DEVICE_SHUTDOWN`. It is rarely implemented explicitly in FreeBSD drivers; a search of `/usr/src/sys/dev/` shows only a handful of drivers define their own `device_quiesce`. Understanding when you do need it and when you do not is worth a short section.

22.16.1 What DEVICE_QUIESCE Is For

The `device_quiesce` wrapper in `/usr/src/sys/kern/subr_bus.c` is called in several places:

- `devclass_driver_deleted`: when a driver is being unloaded, the framework calls `device_quiesce` on every instance before calling `device_detach`.
- `DEV_DETACH` via `devctl`: when the user runs `devctl detach myfirst0`, the kernel calls `device_quiesce` before `device_detach` unless the `-f` (force) flag is given.
- `DEV_DISABLE` via `devctl`: when the user runs `devctl disable myfirst0`, the kernel calls `device_quiesce` similarly.

In each case, the quiesce is a pre-check: “can the driver safely stop what it is doing?”. A driver that returns `EBUSY` from `DEVICE_QUIESCE` prevents the subsequent detach or disable. The user gets an error, and the driver stays attached.

22.16.2 What the Default Does

If a driver does not implement `DEVICE_QUIESCE`, the default (`null_quiesce` in `device_if.m`) returns 0 unconditionally. The kernel proceeds with detach or disable.

For most drivers, this is fine. The driver's detach path handles any in-flight work, so there is nothing the quiesce would do that detach does not also do.

22.16.3 When You Would Implement It

A driver implements `DEVICE_QUIESCE` explicitly when:

1. **Returning EBUSY is more informative than waiting.** If the driver has a concept of “busy” (a transfer in flight, an open file descriptor count, a filesystem mount), and the user can wait for it to become non-busy, the driver might refuse quiesce until busy is zero. `DEVICE_QUIESCE` returning `EBUSY` tells the user “the device is busy; wait and retry”.
2. **The quiesce can be done faster than a full detach.** If detach is expensive (frees large resource tables, drains slow queues) but the device can be stopped cheaply, `DEVICE_QUIESCE` lets the kernel probe for readiness without paying detach’s cost.
3. **The driver wants to distinguish quiesce from suspend.** If the driver wants to stop activity but not save state (because no resume is coming), implementing quiesce separately from suspend is a way to express that distinction in code.

For the `myfirst` driver, none of these apply. The Chapter 21 detach path already handles in-flight work; the Chapter 22 suspend path handles quiesce in the power-management sense. Adding a separate `DEVICE_QUIESCE` would be redundant.

22.16.4 An Example from `bce(4)`

The Broadcom NetXtreme driver in `/usr/src/sys/dev/bce/if_bce.c` has a commented-out `DEVMETHOD(device_quiesce, bce_quiesce)` entry in its method table. The comment suggests the author considered implementing quiesce but did not. This is common: many drivers keep the line commented as a `TODO` that never gets implemented, because the default handles their use case.

The implementation, if the driver enabled it, would stop the NIC’s TX and RX paths without freeing the hardware resources. A subsequent `device_detach` would then do the actual freeing. The split between “stop” and “free” is what `DEVICE_QUIESCE` would express.

22.16.5 Relation to `DEVICE_SUSPEND`

`DEVICE_QUIESCE` and `DEVICE_SUSPEND` do similar things: they stop the device’s activity. The differences:

- **Lifecycle:** quiesce is between run and detach; suspend is between run and eventual resume.
- **Resources:** quiesce does not require the driver to save any state; suspend does.
- **Ability to veto:** both can return `EBUSY`; the consequences differ (quiesce prevents detach; suspend prevents the power transition).

A driver that implements both usually shares code: `foo_quiesce` might do “stop activity” and `foo_suspend` might do “call quiesce; save state; return”. The `myfirst` driver’s `myfirst_quiesce` helper is the shared code; the chapter does not wire it to a `DEVICE_QUIESCE` method, but doing so would be a small addition.

22.16.6 An Optional Addition to `myfirst`

As a challenge, the reader can add `DEVICE_QUIESCE` to `myfirst`:

```
static int
myfirst_pci_quiesce(device_t dev)
{
    struct myfirst_softc *sc = device_get_softc(dev);

    device_printf(dev, "quiesce: starting\n");
    (void)myfirst_quiesce(sc);
    atomic_add_64(&sc->power_quiesce_count, 1);
    device_printf(dev, "quiesce: complete\n");
    return (0);
}
```

And the matching method-table entry:

```
DEVMETHOD(device_quiesce, myfirst_pci_quiesce),
```

Testing it: `devctl detach myfirst0` calls `quiesce` before detach; the reader can verify by reading `dev.myfirst.0.power_quiesce_count` immediately before the detach takes effect.

The challenge is short and does not change the driver's overall structure; it just wires one more method. Chapter 22's consolidated Stage 4 does not include it by default, but the reader who wants the method can add it in a few lines.

22.17 Hands-On Labs

Chapter 22 includes three hands-on labs that exercise the power-management path in progressively harder ways. Each lab has a script in `examples/part-04/ch22-power/labs/` that the reader can run as-is, plus extension ideas.

22.17.1 Lab 1: Single-Cycle Suspend-Resume

The first lab is the simplest: one clean suspend-resume cycle with counter verification.

Setup. Load the Chapter 22 Stage 4 driver:

```
cd examples/part-04/ch22-power/stage4-final
make clean && make
sudo kldload ./myfirst.ko
```

Verify attach:

```
sysctl dev.myfirst.0.%driver
# Should return: myfirst
sysctl dev.myfirst.0.suspended
# Should return: 0
```

Run. Execute the cycle script:

```
sudo sh ../labs/ch22-suspend-resume-cycle.sh
```

Expected output:

```
PASS: one suspend-resume cycle completed cleanly
```

Verify. Inspect the counters:

```
sysctl dev.myfirst.0.power_suspend_count
# Should return: 1
sysctl dev.myfirst.0.power_resume_count
# Should return: 1
```

Check dmesg:

```
dmesg | tail -6
```

Should show four lines (suspend start, suspend complete, resume start, resume complete) plus the pre-and-post transfer log lines.

Extension. Modify the cycle script to run two suspend-resume cycles instead of one, and verify that the counters increment by exactly 2 each.

22.17.2 Lab 2: One-Hundred-Cycle Stress

The second lab runs the cycle script one hundred times in a row and checks that nothing drifts.

Run.

```
sudo sh ../labs/ch22-suspend-stress.sh
```

Expected output after a few seconds:

```
PASS: 100 cycles
```

Verify. After the stress run, the counters should each be 100 (or 100 plus whatever was there before):


```
sysctl dev.myfirst.0.power_suspend_count  
# 100 (or however many cycles were added)
```

Observations to make.

- How long does one cycle take? On the simulation, it should be a few milliseconds. On real hardware with D-state transitions, expect a few hundred microseconds to a few milliseconds.
- Does the system's load average change during the stress? The simulation is cheap; a hundred cycles on a modern machine should barely register.
- What happens if you run the DMA test during the stress? (`sudo sysctl dev.myfirst.0.dma_test_read=1` concurrently with the cycle loop.) A well-written driver should handle this gracefully; the DMA test succeeds if it happens during a `RUNNING` window and fails with `EBUSY` or similar if it happens during a transition.

Extension. Run the stress script with `dmesg -c` before to clear the log, then afterwards:

```
dmesg | wc -l
```

Should be close to 400 (four log lines per cycle, times 100 cycles). A log-line-per-cycle count lets you verify that every cycle actually executed through the driver.

22.17.3 Lab 3: Transfer Across a Cycle

The third lab is the hardest: it starts a DMA transfer and immediately suspends in the middle of it, then resumes and verifies that the driver recovers.

Setup. The lab script is `ch22-transfer-across-cycle.sh`. It runs a DMA transfer in the background, sleeps a few milliseconds, calls `devctl suspend`, sleeps, calls `devctl resume`, and then starts another transfer.

Run.

```
sudo sh ../labs/ch22-transfer-across-cycle.sh
```

Observations to make.

- Does the first transfer complete, error out, or time out? The expected behavior is that the quiesce aborts it cleanly; the transfer reports `EIO` or `ETIMEDOUT`.
- Does the counter `dma_errors` or `dma_timeouts` increment? One of them should.
- Does `dma_in_flight` go back to false after the suspend?
- Does the post-resume transfer succeed normally? If yes, the driver's state is consistent and the cycle worked.

Extension. Reduce the sleep between the transfer start and the suspend to hit the corner case where the transfer is mid-execution at the moment of the suspend. That is where race conditions live; a driver that passes this test under aggressive timing has a solid quiesce implementation.

22.17.4 Lab 4: Runtime PM (Optional)

For readers building with `MYFIRST_ENABLE_RUNTIME_PM`, a fourth lab exercises the runtime-PM path.

Setup. Rebuild with runtime PM enabled:

```
cd examples/part-04/ch22-power/stage4-final
# Uncomment the CFLAGS line in the Makefile:
# CFLAGS+= -DMYFIRST_ENABLE_RUNTIME_PM
make clean && make
sudo kldload ./myfirst.ko
```

Run.

```
sudo sh ../labs/ch22-runtime-pm.sh
```

The script:

1. Sets the idle threshold to 3 seconds (instead of the default 5).
2. Records baseline counters.
3. Waits 5 seconds without any activity.
4. Verifies `runtime_state` is `RUNTIME_SUSPENDED`.
5. Triggers a DMA transfer.
6. Verifies `runtime_state` is back to `RUNNING`.
7. Prints PASS.

Observations to make.

- During the idle wait, `dmesg` should show the “runtime suspend” log line approximately 3 seconds in.
- The `runtime_suspend_count` and `runtime_resume_count` should each be 1 at the end.
- The DMA transfer should succeed normally after the runtime resume.

Extension. Set the idle threshold to 1 second. Run the DMA test repeatedly in a tight loop. You should see no runtime-suspend transitions during the loop (because each test resets the idle timer), but as soon as the loop stops, the runtime suspend fires.

22.17.5 Lab Notes

All of the labs assume the driver is loaded and the system is idle enough that transitions happen on-demand. If another process is actively using the device (unlikely for `myfirst`, but common in real setups), the counters drift by unexpected amounts and the scripts’ exact-increment checks fail. The scripts are designed for a quiet test environment, not a noisy one.

For realistic testing of the `re(4)` driver or other production drivers, the same script structure applies with the device name adjusted. The `devctl suspend/devctl resume` dance works for any PCI device the kernel manages.

22.18 Challenge Exercises

The Chapter 22 challenge exercises push the reader beyond the baseline driver into territory that real-world drivers eventually have to handle. Each exercise is scoped to be achievable with the chapter’s material and a few hours of work.

22.18.1 Challenge 1: Implement a Wake-on-Sysctl Mechanism

Extend the `myfirst` driver with a simulated wake source. The simulation already has a callout that can fire; add a new simulation feature that sets a “wake” bit on the device while it is in D3, and have the driver’s `DEVICE_RESUME` path log the wake event.

Hints.

- Add a `MYFIRST_REG_WAKE_STATUS` register to the simulation backend.
- Add a `MYFIRST_REG_WAKE_ENABLE` register the driver writes during suspend.
- Have the simulation callout set the wake status bit after a random delay.
- On resume, the driver reads the register and logs whether a wake was observed.

Verification. After `devctl suspend; sleep 1; devctl resume`, the log should show the wake status. A follow-up `sysctl dev.myfirst.0.wake_events` should increment.

Why this matters. Wake source handling is one of the trickiest parts of real-hardware power management. Building it into the simulation lets the reader exercise the full contract without needing hardware.

22.18.2 Challenge 2: Save and Restore a Descriptor Ring

The Chapter 21 simulation does not yet use a descriptor ring (transfers are one-at-a-time). Extend the simulation with a small descriptor ring, program its base address through a register at attach, and have the suspend path save the ring’s base address into softc state. Have the resume path write the saved base address back.

Hints.

- The ring’s base address is a `bus_addr_t` held in the softc.
- The register is `MYFIRST_REG_RING_BASE_LOW/_HIGH`.
- Saving and restoring is trivial; the point is to verify that *not* saving and restoring would break things.

Verification. After suspend-resume, the ring base register should hold the same value as before. Without the restore, it should hold zero.

Why this matters. Descriptor rings are what real high-throughput drivers use; a power-aware driver with a ring has to restore the base address on every resume. This exercise is a stepping stone to the kind of state management that production drivers like `re(4)` and `em(4)` perform.

22.18.3 Challenge 3: Implement a Veto Policy

Extend the suspend path with a policy knob that lets the user specify whether the driver should veto a suspend when the device is busy. Specifically:

- Add `dev.myfirst.0.suspend_veto_if_busy` as a read-write sysctl.
- If the sysctl is 1 and a DMA transfer is in flight, `myfirst_power_suspend` returns `EBUSY` without quiescing.
- If the sysctl is 0 (default), `suspend` always succeeds.

Hints. Set `suspend_veto_if_busy` to 1. Start a long DMA transfer (add a `DELAY` to the simulation's engine to make it last a second or two). Call `devctl suspend myfirst0` during the transfer. Verify that the `suspend` returns an error and `dev.myfirst.0.suspended` stays 0.

Verification. The kernel's unwind path runs; the driver is still in `RUNNING`; the transfer completes normally.

Why this matters. Vetoing is an effective tool and a dangerous one. Real-world policy decisions about whether to veto are nuanced (storage drivers often veto; NIC drivers usually do not). Implementing the mechanism makes the policy question tangible.

22.18.4 Challenge 4: Add a Post-Resume Self-Test

After resume, do a minimum-viable test of the device: write a known pattern to the DMA buffer, trigger a write transfer, read it back with a read transfer, and verify. If the test fails, mark the device broken and fail subsequent operations.

Hints.

- Add the self-test as a helper that runs from `myfirst_power_resume` after `myfirst_restore`.
- Use a well-known pattern like `0xDEADBEEF`.
- Use the existing DMA path; the self-test is just one write and one read.

Verification. Under normal operation, the self-test always passes. To verify it catches failures, add an artificial “fail once” mechanism to the simulation and trigger it; the driver should log the failure and mark itself broken.

Why this matters. Self-tests are a lightweight form of reliability engineering. A driver that catches its own failures at well-defined points is easier to debug than one that silently corrupts data until a user notices.

22.18.5 Challenge 5: Implement Manual `pci_save_state` / `pci_restore_state`

Most drivers let the PCI layer handle config-space save-and-restore automatically. Extend the Chapter 22 driver to optionally do it manually, gated by a sysctl `dev.myfirst.0.manual_pci_save`.

Hints.

- Read `hw.pci.do_power_suspend` and `hw.pci.do_power_resume` and set them to 0 when manual mode is enabled.
- Call `pci_save_state` explicitly in the suspend path, `pci_restore_state` in the resume path.
- Verify that the device still works after suspend-resume.

Verification. The device should function identically whether or not manual mode is enabled. Set the sysctl before a stress test and verify no drift.

Why this matters. Some real drivers need manual save/restore because the PCI layer's automatic handling interferes with device-specific quirks. Knowing when and how to take over the save/restore is a useful intermediate skill.

22.19 Troubleshooting Reference

This section collects the common problems a reader may encounter while working through Chapter 22, with a short diagnostic and fix for each. The list is meant to be skimmable; if a problem matches, skip to the corresponding entry.

22.19.1 “devctl: DEV_SUSPEND failed: Operation not supported”

The driver does not implement `DEVICE_SUSPEND`. Either the method table is missing the `DEVMETHOD(device_suspend, ...)` line, or the driver has not been rebuilt and reloaded.

Fix. Check the method table. Rebuild with `make clean && make`. Unload and reload.

22.19.2 “devctl: DEV_SUSPEND failed: Device busy”

The driver returned `EBUSY` from `DEVICE_SUSPEND`, probably because of the veto logic from Challenge 3, or because the device is genuinely busy (DMA in flight, task running) and the driver chose to veto.

Fix. Check whether the `suspend_veto_if_busy` knob is set. Check `dma_in_flight`. Wait for activity to complete before suspending.

22.19.3 “devctl: DEV_RESUME failed”

`DEVICE_RESUME` returned non-zero. The log should have more detail.

Fix. Check `dmesg | tail`. The resume log line should tell you what failed. Usually it is a hardware-specific init step that did not succeed.

22.19.4 Device is suspended but `dev.myfirst.0.suspended` reads 0

The driver’s flag is out of sync with the kernel’s state. Probably a bug in the quiesce path: the flag was never set, or was cleared prematurely.

Fix. Add a `KASSERT(sc->suspended == true)` at the top of the resume path; run under `INVARIANTS` to catch the bug.

22.19.5 `power_suspend_count != power_resume_count`

A cycle got one side but not the other. Check `dmesg` for errors; the log should show where the sequence broke.

Fix. Fix the code path that is missing. Usually an early return without the counter update.

22.19.6 DMA transfers fail after resume

The restore path did not reinitialise the DMA engine. Check the `INTR_MASK` register, the DMA control registers, the `saved_intr_mask` value. Enable verbose logging to see the resume path’s restoration sequence.

Fix. Add a missing register write to `myfirst_restore`.

22.19.7 WITNESS complains about a lock held during suspend

The suspend path acquired a lock and then called a function that sleeps or tries to acquire another lock. Read the WITNESS message for the offending lock names.

Fix. Drop the lock before the sleeping call, or restructure the code so the lock is acquired only when needed.

22.19.8 System does not wake from S3

A driver below `myfirst` is blocking resume. Unlikely to be `myfirst` itself unless the logs show an error from the driver specifically.

Fix. Boot into single-user mode, or load fewer drivers, and bisect. Check `dmesg` in the live system for the offending driver.

22.19.9 Runtime PM never fires

The idle watcher callout is not running, or the `last_activity` timestamp is being updated too often.

Fix. Verify `callout_reset` is being called from the attach path. Verify `myfirst_mark_active` is not being called from unexpected code paths. Add logging to the callout callback to confirm it fires.

22.19.10 Kernel panic during suspend

A KASSERT failed (on an INVARIANTS kernel) or a lock is held incorrectly. The panic message identifies the offending file and line.

Fix. Read the panic message. Match the file and line to the code. The fix is usually straightforward once the location is identified.

22.20 Wrapping Up

Chapter 22 closes Part 4 by giving the `myfirst` driver the discipline of power management. At the start, `myfirst` at version 1.4-dma was a capable driver: it attached to a PCI device, handled multi-vector interrupts, moved data through DMA, and cleaned up its resources on detach. What it lacked was the ability to participate in the system's power transitions. It would crash, leak, or silently fail if the user closed the laptop lid or asked the kernel to suspend the device. At the end, `myfirst` at version 1.5-power handles every power transition the kernel can throw at it: system suspend to S3 or S4, per-device suspend through `devctl`, system shutdown, and optional runtime power management.

The eight sections walked the full progression. Section 1 established the big picture: why a driver cares about power, what ACPI S-states and PCI D-states are, what PCIe L-states and ASPM add, and what wake sources look like. Section 2 introduced FreeBSD's concrete APIs: the `DEVICE_SUSPEND`, `DEVICE_RESUME`, `DEVICE_SHUTDOWN`, and `DEVICE_QUIESCE` methods, the `bus_generic_suspend` and `bus_generic_resume` helpers, and the PCI layer's automatic config-space save and restore. The Stage 1 skeleton made the methods log and count transitions without doing any real work. Section 3 turned the suspend skeleton into a real quiesce:

mask interrupts, drain DMA, drain workers, in that order, with helper functions shared between suspend and detach. Section 4 wrote the matching resume path: re-enable bus-master, restore device-specific state, clear the suspended flag, unmask interrupts. Section 5 added optional runtime power management with an idle-watcher callout and explicit `pci_set_powerstate` transitions. Section 6 surveyed the user-space interface: `acpiconf`, `zzz`, `devctl suspend`, `devctl resume`, `devinfo -v`, and the matching `sysctls`. Section 7 catalogued the characteristic failure modes and their debugging patterns. Section 8 refactored the code into `myfirst_power.c`, bumped the version to 1.5-power, added `POWER.md`, and wired the final regression test.

What Chapter 22 did not do is scatter-gather power management for multi-queue drivers (that is a Part 6 topic, Chapter 28), hotplug and surprise-removal integration (a Part 7 topic), embedded-platform power domains (Part 7 again), or the internals of ACPI's AML interpreter (never covered in this book). Each of those is a natural extension built on Chapter 22's primitives, and each belongs in a later chapter where the scope matches. The foundation is in place; the specialisations add vocabulary without needing a new foundation.

The file layout has grown: 16 source files (including `cbuf`), 8 documentation files (`HARDWARE.md`, `LOCKING.md`, `SIMULATION.md`, `PCI.md`, `INTERRUPTS.md`, `MSIX.md`, `DMA.md`, `POWER.md`), and an extended regression suite that covers every subsystem. The driver is structurally parallel to production FreeBSD drivers; a reader who has worked through Chapters 16 through 22 can open `if_re.c`, `if_xl.c`, or `virtio_blk.c` and recognise every architectural part: register accessors, simulation backend, PCI attach, interrupt filter and task, per-vector machinery, DMA setup and teardown, sync discipline, power suspend, power resume, clean detach.

22.20.1 A Reflection Before Chapter 23

Chapter 22 is the last chapter of Part 4, and Part 4 is the part that taught the reader how a driver talks to hardware. Chapters 16 through 21 introduced the primitives: MMIO, simulation, PCI, interrupts, multi-vector interrupts, DMA. Chapter 22 introduced the discipline: how those primitives survive power transitions. Together, the seven chapters take the reader from “no idea what a driver is” to “a working multi-subsystem driver that handles every hardware event the kernel can throw at it”.

Chapter 22's teaching generalises. A reader who has internalised the suspend-quiesce-save-restore pattern, the interaction between driver and PCI layer, the runtime-PM state machine, and the debugging patterns will find similar shapes in every power-aware FreeBSD driver. The specific hardware differs; the structure does not. A driver for a NIC, a storage controller, a GPU, or a USB host controller applies the same vocabulary to its own hardware.

Part 5, which begins with Chapter 23, shifts focus. Part 4 was about the driver-to-hardware direction: how the driver talks to the device. Part 5 is about the driver-to-kernel direction: how the driver is debugged, traced, tooled, and stressed by the humans who maintain it. Chapter 23 starts that shift with debugging and tracing techniques that apply across every driver subsystem.

22.20.2 What to Do If You Are Stuck

Three suggestions.

First, focus on the Stage 2 suspend and Stage 3 resume paths. If `devctl suspend myfirst0` followed by `devctl resume myfirst0` succeeds and a subsequent DMA transfer works, the core of the chapter is working. Every other piece of the chapter is optional in the sense that it decorates the pipeline, but if the pipeline fails, the chapter is not working and Section 3 or Section 4 is the right place to diagnose.

Second, open `/usr/src/sys/dev/re/if_re.c` and re-read `re_suspend`, `re_resume`, and `re_setwol`. Each function is about thirty lines. Every line maps to a Chapter 22 concept. Reading them once after completing the chapter should feel like familiar territory; the real driver's patterns will look like elaborations of the chapter's simpler ones.

Third, skip the challenges on the first pass. The labs are calibrated for Chapter 22's pace; the challenges assume the chapter's material is solid. Come back to them after Chapter 23 if they feel out of reach now.

Chapter 22's goal was to give the driver power-management discipline. If it has, Chapter 23's debugging and tracing machinery becomes a generalisation of what you already do instinctively rather than a new topic.

22.21 Part 4 Checkpoint

Part 4 has been the longest and densest stretch of the book so far. Seven chapters covered hardware resources, register I/O, PCI attach, interrupts, MSI and MSI-X, DMA, and power management. Before Part 5 changes the mode from “writing drivers” to “debugging and tracing them,” confirm that the hardware-facing story is internalized.

By the end of Part 4 you should be able to do each of the following without searching:

- Claim a hardware resource with `bus_alloc_resource_any` or `bus_alloc_resource_anywhere`, access it through the `bus_space(9)` read/write and barrier primitives, and release it cleanly in detach.
- Read and write device registers through the `bus_space(9)` abstraction rather than raw pointer dereferences, with correct barrier discipline around sequences that must not be reordered.
- Match a PCI device through vendor, device, subvendor, and subdevice IDs; claim its BARs; and survive a forced detach without leaking resources.
- Register a top-half filter together with a bottom-half task or ithread via `bus_setup_intr`, in the order the kernel requires, and tear them down in reverse order under detach.
- Set up MSI or MSI-X vectors with a graceful fallback ladder from MSI-X to MSI to legacy INTx, and bind vectors to specific CPUs when the workload calls for it.
- Allocate, map, sync, and release DMA buffers using `bus_dma(9)` including the bounce-buffer case.
- Implement `device_suspend` and `device_resume` with register save and restore, I/O quiescing, and a post-resume self-test.

If any of those still requires a lookup, the labs to revisit are:

- Registers and barriers: Lab 1 (Observe the Register Dance) and Lab 8 (The Watchdog-Meets-Register Scenario) in Chapter 16.
- Simulated hardware under load: Lab 6 (Inject Stuck-Busy and Watch the Driver Wait) and Lab 10 (Inject a Memory-Corruption Attack) in Chapter 17.
- PCI attach and detach: Lab 4 (Claim the BAR and Read a Register) and Lab 5 (Exercise the `cdev` and Verify Detach Cleanup) in Chapter 18.
- Interrupt handling: Lab 3 (Stage 2, Real Filter and Deferred Task) in Chapter 19.
- MSI and MSI-X: Lab 4 (Stage 3, MSI-X With CPU Binding) in Chapter 20.
- DMA: Lab 4 (Stage 3, Interrupt-Driven Completion) and Lab 5 (Stage 4, Refactor and Regression) in Chapter 21.
- Power management: Lab 2 (One-Hundred-Cycle Stress) and Lab 3 (Transfer Across a Cycle) in Chapter 22.

Part 5 will expect the following as a baseline:

- A hardware-capable driver with observability already baked in: counters, sysctls, and `devctl_notify` calls at the important transitions. Chapter 23's debugging machinery works best when the driver already reports on itself.
- A regression script that can cycle the driver reliably, since Part 5 turns reproducibility into a first-class skill.
- A kernel built with `INVARIANTS` and `WITNESS`. Part 5 leans on both even more heavily than Part 4, especially in Chapter 23.
- The understanding that a bug in driver code is a bug in kernel code, which means user-space debuggers alone will not be enough and Part 5 will teach the kernel-space tools.

If those hold, Part 5 is ready for you. If one still looks shaky, a short lap through the relevant lab will pay back its time several times over.

22.22 Bridge to Chapter 23

Chapter 23 is titled *Debugging and Tracing*. Its scope is the professional practice of finding bugs in drivers: tools like `ktrace`, `ddb`, `kgdb`, `dtrace`, and `procstat`; techniques for analysing panics, deadlocks, and data corruption; strategies for turning vague user reports into reproducible test cases; and the mindset of a driver developer who has to debug code running in kernel space with limited visibility.

Chapter 22 prepared the ground in four specific ways.

First, **you have observability counters everywhere**. The Chapter 22 driver exposes suspend, resume, shutdown, and runtime-PM counters through sysctls. Chapter 23's debugging techniques rely on observability; a driver that already tracks its own state is much easier to debug than one that does not.

Second, **you have a regression test**. The cycle and stress scripts from Section 6 are a first taste of what Chapter 23 expands: the ability to reproduce a bug on demand. A bug you cannot reproduce is a bug you cannot fix; Chapter 22's scripts are a foundation for the heavier testing Chapter 23 adds.

Third, **you have a working `INVARIANTS` / `WITNESS` debug kernel**. Chapter 22 leaned on both throughout; Chapter 23 builds on the same kernel for `ddb` sessions, post-mortem analysis, and kernel-crash reproduction.

Fourth, **you understand that bugs in driver code are bugs in kernel code**. Chapter 22 ran into hangs, frozen devices, lost interrupts, and `WITNESS` complaints. Each of those is a kernel bug in the user-visible sense; each requires a kernel-space debugging approach. Chapter 23 teaches that approach systematically.

Specific topics Chapter 23 will cover:

- Using `ktrace` and `kdump` to observe a process's system call trace in real time.
- Using `ddb` to break into the kernel debugger for post-mortem analysis or live inspection.
- Using `kgdb` with a core dump to recover the state of a crashed kernel.
- Using `dtrace` for in-kernel tracing without modifying the source.
- Using `procstat`, `top`, `pmcstat`, and related tools for performance observation.
- Strategies for minimising a bug: shrinking a reproducer, bisecting a regression, hypothesising and testing.
- Patterns for instrumenting a driver in production without disturbing behaviour.

You do not need to read ahead. Chapter 22 is sufficient preparation. Bring your `myfirst` driver at 1.5-power, your `LOCKING.md`, your `INTERRUPTS.md`, your `MSIX.md`, your `DMA.md`, your `POWER.md`, your WITNESS-enabled kernel, and your regression script. Chapter 23 starts where Chapter 22 ended.

Part 4 is complete. Chapter 23 opens Part 5 by adding the observability and debugging discipline that separates a driver you wrote last week from a driver you can maintain for years.

The vocabulary is yours; the structure is yours; the discipline is yours. Chapter 23 adds the next missing piece: the ability to find and fix bugs that only show up in production.

22.23 Reference: Chapter 22 Quick-Reference Card

A compact summary of the vocabulary, APIs, flags, and procedures Chapter 22 introduced.

22.23.1 Vocabulary

- **Suspend:** a transition from D0 (full operation) to a lower-power state from which the device can be brought back.
- **Resume:** the transition back from the lower-power state to D0.
- **Shutdown:** the transition to a final state from which the device will not return.
- **Quiesce:** to bring a device to a state with no activity and no pending work.
- **System sleep state (S0, S1, S3, S4, S5):** ACPI-defined levels of system power.
- **Device power state (D0, D1, D2, D3hot, D3cold):** PCI-defined levels of device power.
- **Link state (L0, L0s, L1, L1.1, L1.2, L2):** PCIe-defined levels of link power.
- **ASPM (Active-State Power Management):** automatic transitions between L0 and L0s/L1.
- **PME# (Power Management Event):** a signal a device asserts when it wants to wake the system.
- **Wake source:** a mechanism by which a suspended device can request wakeup.
- **Runtime PM:** device-level power saving while the system stays in S0.

22.23.2 Essential Kobj Methods

- `DEVMETHOD(device_suspend, foo_suspend)`: called to quiesce the device before a power transition.
- `DEVMETHOD(device_resume, foo_resume)`: called to restore the device after the power transition.
- `DEVMETHOD(device_shutdown, foo_shutdown)`: called to leave the device in a safe state for reboot.
- `DEVMETHOD(device_quiesce, foo_quiesce)`: called to stop activity without tearing down resources.

22.23.3 Essential PCI APIs

- `pci_has_pm(dev)`: true if the device has a power-management capability.
- `pci_set_powerstate(dev, state)`: transition to `PCI_POWERSTATE_D0`, `D1`, `D2`, or `D3`.
- `pci_get_powerstate(dev)`: current power state.
- `pci_save_state(dev)`: cache the configuration space.
- `pci_restore_state(dev)`: write the cached configuration space back.
- `pci_enable_pme(dev)`: enable PME# generation.

- `pci_clear_pme(dev)`: clear pending PME status.
- `pci_enable_busmaster(dev)`: re-enable bus-master after a reset.

22.23.4 Essential Bus Helpers

- `bus_generic_suspend(dev)`: suspend all children in reverse order.
- `bus_generic_resume(dev)`: resume all children in forward order.
- `device_quiesce(dev)`: call the driver's `DEVICE_QUIESCE`.

22.23.5 Essential Sysctls

- `hw.acpi.supported_sleep_state`: list of S-states the platform supports.
- `hw.acpi.suspend_state`: default S-state for zzz.
- `hw.pci.do_power_suspend`: automatic D0->D3 transition on suspend.
- `hw.pci.do_power_resume`: automatic D3->D0 transition on resume.
- `dev.N.M.suspended`: driver's own suspended flag.
- `dev.N.M.power_suspend_count`, `power_resume_count`, `power_shutdown_count`.
- `dev.N.M.runtime_state`, `runtime_suspend_count`, `runtime_resume_count`.

22.23.6 Useful Commands

- `acpiconf -s 3`: enter S3.
- `zzz`: wrapper around `acpiconf`.
- `devctl suspend <device>`: per-device suspend.
- `devctl resume <device>`: per-device resume.
- `devinfo -v`: device tree with state.
- `pciconf -lvbc`: PCI devices with power state.
- `sysctl -a | grep acpi`: all ACPI-related variables.

22.23.7 Common Procedures

Method table addition:

```
DEVMETHOD(device_suspend,  foo_suspend),
DEVMETHOD(device_resume,   foo_resume),
DEVMETHOD(device_shutdown, foo_shutdown),
```

Suspend skeleton:

```
int foo_suspend(device_t dev) {
    struct foo_softc *sc = device_get_softc(dev);
    FOO_LOCK(sc);
    sc->suspended = true;
```

```
    FOO_UNLOCK(sc);
    foo_mask_interrupts(sc);
    foo_drain_dma(sc);
    foo_drain_workers(sc);
    return (0);
}
```

Resume skeleton:

```
int foo_resume(device_t dev) {
    struct foo_softc *sc = device_get_softc(dev);
    pci_enable_busmaster(dev);
    foo_restore_registers(sc);
    FOO_LOCK(sc);
    sc->suspended = false;
    FOO_UNLOCK(sc);
    foo_unmask_interrupts(sc);
    return (0);
}
```

Runtime-PM helper:

```
int foo_runtime_suspend(struct foo_softc *sc) {
    foo_quiesce(sc);
    pci_save_state(sc->dev);
    return (pci_set_powerstate(sc->dev, PCI_POWERSTATE_D3));
}

int foo_runtime_resume(struct foo_softc *sc) {
    pci_set_powerstate(sc->dev, PCI_POWERSTATE_D0);
    pci_restore_state(sc->dev);
    return (foo_restore(sc));
}
```

22.23.8 Files to Keep Bookmarked

- /usr/src/sys/kern/device_if.m: the kobj method definitions.
- /usr/src/sys/kern/subr_bus.c: bus_generic_suspend, bus_generic_resume, device_quiesce.
- /usr/src/sys/dev/pci/pci.c: pci_suspend_child, pci_resume_child, pci_save_state, pci_restore_state.
- /usr/src/sys/dev/pci/pcivar.h: PCI_POWERSTATE_* constants and inline API.
- /usr/src/sys/dev/re/if_re.c: production reference for suspend/resume with WoL.
- /usr/src/sys/dev/xl/if_xl.c: minimal suspend/resume pattern.
- /usr/src/sys/dev/virtio/block/virtio_blk.c: virtio-style quiesce.

22.24 Reference: Glossary of Chapter 22 Terms

A short glossary of the chapter's new terms.

- **ACPI (Advanced Configuration and Power Interface):** the industry-standard interface between OS and platform firmware for power management.
- **ASPM (Active-State Power Management):** automatic PCIe link-state transitions.
- **D-state:** a device power state (D0 through D3cold).
- **DEVICE_QUIESCE:** the kobj method that stops activity without tearing down resources.
- **DEVICE_RESUME:** the kobj method called to restore a device to operation.
- **DEVICE_SHUTDOWN:** the kobj method called at system shutdown.
- **DEVICE_SUSPEND:** the kobj method called to quiesce a device before a power transition.
- **GPE (General-Purpose Event):** an ACPI wake event source.
- **L-state:** a PCIe link power state.
- **Link state machine:** the automatic transitions between L0 and L0s/L1.
- **PME# (Power Management Event):** the PCI signal a device asserts to request wake.
- **Power management capability:** the PCI capability structure that contains PM registers.
- **Quiesce:** to bring a device to a state with no activity and no pending work.
- **Runtime PM:** device-level power saving while the system stays in S0.
- **S-state:** an ACPI system sleep state (S0 through S5).
- **Shutdown:** final power-down, typically leading to reboot or power-off.
- **Sleep state:** see S-state.
- **Suspend:** temporary power-down from which the system or device can return.
- **Suspended flag:** a driver-local flag indicating the device is in a suspended state.
- **Wake source:** a mechanism by which a suspended system or device can be woken.
- **WoL (Wake on LAN):** a wake source triggered by a network packet.

22.25 Reference: A Closing Note on Power-Management Philosophy

A paragraph to close the chapter with.

Power management is the discipline that separates a driver prototype from a production driver. Before power management, a driver assumes its device is always on and always available. After power management, the driver knows the device can be put to sleep and knows how to put it to sleep correctly, and the driver can be trusted in the kinds of environments real users run: laptops that close and open dozens of times a day, servers that suspend idle devices to save power, VMs that migrate between hosts, embedded systems that turn off whole power domains to extend battery life.

Chapter 22's lesson is that power management is disciplined, not magical. The FreeBSD kernel gives the driver a specific contract (the four kobj methods, the order of invocation, the interaction with the PCI layer), and following the contract is most of the work. The rest is hardware-specific: understanding which registers the device loses across a D-state transition, which wake sources the hardware supports, which policy the driver should apply for runtime PM. The pattern is the same across every power-aware driver in FreeBSD; internalising it once pays off across dozens of later chapters and thousands of lines of real driver code.

For this reader and for this book's future readers, the Chapter 22 power-management pattern is a permanent part of the `myfirst` driver's architecture and a permanent tool in the reader's toolkit. Chapter 23 assumes it:

debugging a driver assumes the driver has the observability counters and the structured lifecycle Chapter 22 introduced. Part 6's specialisation chapters assume it: every production-style driver has a power path. Part 7's performance chapters (Chapter 33) assume it: every tuning measurement has to account for power-state transitions. The vocabulary is the vocabulary every production FreeBSD driver shares; the patterns are the patterns production drivers live by; the discipline is the discipline that keeps power-aware platforms correct.

The skill Chapter 22 teaches is not “how to add suspend and resume methods to a single PCI driver”. It is “how to think about a driver's lifecycle as attach, run, quiesce, sleep, wake, run, and eventually detach, rather than just attach, run, detach”. That skill applies across every driver the reader will ever work on.

Part 4 is complete. The `myfirst` driver is at 1.5-power, structurally parallel to a production FreeBSD driver, and ready for the debugging, tooling, and specialisation chapters that follow in Parts 5 and 6.

23 Debugging and Tracing

23.1 Reader Guidance & Outcomes

Chapter 22 closed Part 4 with a driver that survives suspend, resume, and shutdown. The `myfirst` driver at version 1.5-power attaches to a PCI device, allocates MSI-X vectors, runs a DMA pipeline, handles the full kobj power-management contract, exposes counters through sysctls, and passes a regression script that exercises attach, detach, suspend, resume, and runtime PM. For a driver that boots, runs, suspends, wakes, and eventually unloads, those mechanics are complete. What the driver does not yet have is the ability to tell its developer what is going on inside it when something goes wrong.

Chapter 23 adds that ability. Part 5 is titled *Debugging, Tools, and Real-World Practices*, and Chapter 23 opens it by teaching the observability and diagnosis discipline that turns a driver you just wrote into a driver you can maintain for years. The reader will spend this chapter learning how kernel-space debugging differs from userland debugging, how to use `printf()` and `device_printf()` with intent rather than by habit, how to follow log messages from the moment they are written to the moment a user reads them with `dmesg`, how to build a debug kernel that catches mistakes the normal kernel misses, how to ask DTrace to show the driver's live behavior without rebuilding anything, how to use `ktrace` and `kdump` to follow a user program as it crosses into the kernel through the driver's interface, how to read the characteristic patterns of common driver bugs the first time a symptom appears, and how to refactor the driver's debugging support so that it remains useful without turning the source into a pile of scattered `printf` calls.

Chapter 23's scope is precisely this. It teaches the everyday kernel-debugging toolkit a FreeBSD driver author uses in lab work, in regression testing, and in responding to bug reports. It teaches enough DTrace to instrument a driver and measure its behavior. It teaches the mental model behind the kernel's debug options so the reader can choose the right set for the problem in front of them. It shows where in the source tree these facilities live and how to read their documentation directly. It does not teach the deeper mechanics of post-mortem kernel crash analysis, because that material belongs further along in this book as a specialised topic. It does not teach driver architecture or integration with specific subsystems, because Chapter 24 is about that. It does not teach advanced performance tuning or large-scale profiling, because Part 6 covers hardware-aware tuning in depth. The discipline Chapter 23 introduces is the foundation the later chapters assume: you cannot tune what you cannot observe, and you cannot fix what you cannot reproduce.

Part 5 is where the driver earns the qualities that separate a working prototype from production software. Chapter 22 made the driver survive a change of power state. Chapter 23 makes the driver tell you what it is doing. Chapter 24 will make the driver integrate cleanly with the kernel subsystems users actually reach from their programs. Chapter 25 will make the driver resilient under stress. Each chapter adds one more quality. Chapter 23 adds observability, which turns out to be the quality every later topic depends on.

23.1.1 Why Kernel Debugging Earns a Chapter of Its Own

Before the first line of code, it helps to understand why DTrace, dmesg, and the debug-kernel options get a whole chapter. You may already be wondering why this is any harder than debugging a user program: add some `printf` statements, run under `gdb`, step through the code, done. That mental model is reasonable in user space and catastrophic in kernel space, for three linked reasons.

The first is that **a driver bug is a kernel bug**. When a userland program has a bug, the operating system protects the rest of the machine from it: the kernel kills the process, the shell prints a message, the user tries again. None of that protection applies inside the kernel. A single bad pointer, a single missed lock, a single unfreed allocation, a single race between an interrupt handler and a task, can panic the whole system. A driver that crashes in testing drops the running VM. A driver that leaks memory eventually starves the machine. A driver that corrupts memory destroys whatever data was unlucky enough to sit next to the bug. Chapter 23 exists because these failures do not look or behave like userland failures, and the tools to find them do not look or behave like userland tools.

The second is that **visibility is limited and costly**. In a userland debugger the program is stopped for you. You can step, you can print variables, you can set conditional breakpoints, you can rewind the clock with `rr` or `gdb --record`. In the kernel, none of that is generally available at runtime without special preparation. You cannot stop a running kernel the way you stop a process, because the kernel is also what runs everything else, including the keyboard, the terminal, the network, and the disk. Every piece of visibility you add has a cost: a `printf` is cheap but slow; an SDT probe is fast but has to be compiled in and enabled; a DTrace script is flexible but runs in the kernel itself and touches every probe it fires. Chapter 23 exists because you have to spend visibility budget deliberately, and knowing what each tool costs is part of knowing how to use it.

The third is that **feedback loops are longer**. In userland the write-compile-run cycle is seconds. In kernel work, the cycle is the same under best conditions, but the consequence of a mistake is a reboot or a rollback of the VM snapshot. A bug that takes one try to reproduce in userland may take ten suspends and resumes in kernel land before the driver's internal counter wraps and the symptom appears. A bug that reproduces in every run under `gdb` may reproduce only under an unloaded kernel, at a specific load, or on a specific CPU. Chapter 23 exists because kernel debugging is only tolerable if the feedback loop is as short as you can make it, and short loops require the right combination of tools for each class of bug.

Chapter 23 earns its place by teaching those three realities together, concretely, with the `myfirst` driver as the running example. A reader who finishes Chapter 23 can add disciplined, toggleable debug output to any FreeBSD driver; read and interpret `dmesg`, `/var/log/messages`, and DTrace output without help; build a debug kernel and know what each option buys; find and fix the common driver bug classes the first time a symptom surfaces; and turn a vague report ("my machine sometimes hangs when the driver loads") into a reproducible test case, a hypothesis, a fix, and a regression.

23.1.2 Where Chapter 22 Left the Driver

A few prerequisites to verify before starting. Chapter 23 extends the driver produced at the end of Chapter 22 Stage 4, tagged as version `1.5-power`. If any of the items below is uncertain, return to Chapter 22 and fix it before starting this chapter, because the debugging topics assume the driver has a stable baseline to debug.

- Your driver compiles cleanly and identifies itself as `1.5-power` in `kldstat -v`.

- The driver allocates one or three MSI-X vectors, registers a filter-plus-task interrupt pipeline, binds each vector to a CPU, and prints an interrupt banner during attach.
- The driver allocates a `bus_dma` tag and a 4 KB DMA buffer, exposes the bus address through `dev.myfirst.N.dma_bus_addr`, and cleans up the DMA resources on detach.
- The driver implements `DEVICE_SUSPEND`, `DEVICE_RESUME`, and `DEVICE_SHUTDOWN`. A `devctl suspend myfirst0` followed by `devctl resume myfirst0` succeeds cleanly, and a subsequent DMA transfer works without error.
- Counters for suspend, resume, shutdown, and runtime-PM transitions appear as `sysctls` under `dev.myfirst.N.`
- `HARDWARE.md`, `LOCKING.md`, `SIMULATION.md`, `PCI.md`, `INTERRUPTS.md`, `MSIX.md`, `DMA.md`, and `POWER.md` are current in your working tree.
- `INVARIANTS`, `WITNESS`, `WITNESS_SKIPSPIN`, `DDB`, `KDB`, and `KDB_UNATTENDED` are enabled in your test kernel.

That driver is what Chapter 23 extends. The additions are modest in source lines but significant in maintainability: a new `myfirst_debug.c` file, a matching `myfirst_debug.h` header, a bank of logging macros with compile-time and runtime toggles, a set of statically defined DTrace probe points, a verbose-mode `sysctl` knob, a set of debug counters that distinguish normal operation from abnormal events, a version bump to 1.6-debug, a `DEBUG.md` document that explains the subsystem, and a small collection of helper scripts that turn raw log and trace output into something a human can read.

23.1.3 What You Will Learn

By the end of this chapter you will be able to:

- Describe what makes kernel debugging different from userland debugging, in concrete terms, and explain how the difference shapes the rest of the chapter's toolkit.
- Use `printf()`, `device_printf()`, `device_log()`, and `log()` correctly in driver code, with an understanding of when each is appropriate and what log priority levels mean in FreeBSD.
- Read and filter `dmesg`, follow the kernel message buffer through its lifecycle, correlate boot-time and runtime messages, and know when the messages you want are in `dmesg`, when they are in `/var/log/messages`, and when they are in both.
- Build a custom debug kernel with `DDB`, `KDB`, `KDB_UNATTENDED`, `INVARIANTS`, `WITNESS`, and related options, and know what each option actually does at runtime, what it costs, and when you want it.
- Enter `ddb` deliberately during a test, run the basic inspection commands (`show`, `ps`, `bt`, `trace`, `show alllocks`, `show pcpu`, `show mbufs`), leave `ddb` cleanly, and understand what you are looking at in each case.
- Understand what a kernel crash dump is, how `dumpdev` and `savecore(8)` produce one, and how `kgdb` opens a dump for post-mortem analysis at a level sufficient to open one and inspect a few frames.
- Explain what DTrace is, load the `dtraceall` module, list the providers the kernel exposes, write a short script that traces a kernel function, and use the `fbt`, `syscall`, `sched`, `io`, `vfs`, and `sdt` providers for common driver questions.
- Add SDT (Statically Defined Tracing) probes to a driver using `SDT_PROVIDER_DEFINE`, `SDT_PROBE_DEFINE`, and the `SDT_PROBE` macros, and use `dtrace -n 'myfirst:::entry'` to observe them.
- Run `ktrace` against a user program that talks to the driver, read the output with `kdump`, and map syscalls like `open`, `ioctl`, `read`, and `write` back to the driver's entry points.

- Recognise the characteristic patterns of driver bugs: unreleased allocations visible in `vmstat -m`, lock order violations reported by `WITNESS`, `bus_space` access past the end of a BAR, `bus_dmamap_sync` calls missed before or after a transfer, access to a device after its `DEVICE_DETACH` has run, and the symptoms each produces.
- Refactor the driver's debugging support into a dedicated `myfirst_debug.c` and `myfirst_debug.h` pair with runtime-toggable verbose output, `ENTRY` and `EXIT` macros for function-level tracing, `ERROR` and `WARN` macros for structured problem reporting, SDT probes for external tracing, and a `sysctl` subtree for the debug controls.
- Read real driver debug code in the FreeBSD source tree, such as the `DPRINTF` family in `/usr/src/sys/dev/ath/if_ath_debug.h`, the `bootverbose` usage in `/usr/src/sys/dev/virtio/block-virtio_blk.c`, the SDT probes in `/usr/src/sys/dev/virtio/virtqueue.c`, and the debug `sysctls` in drivers like `/usr/src/sys/dev/iwm/if_iwm.c`.

The list is long because debugging is itself a family of skills. Each item is narrow and teachable. The chapter's work is the composition.

23.1.4 What This Chapter Does Not Cover

Several adjacent topics are explicitly deferred so Chapter 23 stays focused on everyday debugging discipline.

- **Deep kgdb scripting and Python extensions.** The chapter shows how to open a crash dump and inspect a few frames. Building Python helpers on top of `kgdb`, walking kernel data structures with user-defined commands, and scripting dump analysis belong in a later advanced chapter.
- **pmcstat and hardware performance counters.** These are performance-tuning tools, not debugging tools in the narrow sense. Part 6 returns to them.
- **Lock profiling beyond WITNESS.** The options `LOCK_PROFILING` path and the `lockstat` DTrace provider are mentioned in context, but the full tuning workflow for contention analysis belongs with Part 6's performance chapters.
- **Detailed KTR event tracing.** The chapter mentions `KTR` in contrast to `ktrace` so the reader does not confuse the two, but the heavyweight in-kernel event buffer and its circular-log workflow are advanced material that rarely needs a driver's attention.
- **Rate-limited logging frameworks.** Chapter 25 covers logging etiquette, static counters, and time-based rate limiting in depth. Chapter 23 introduces the need for rate limiting informally and hands off.
- **The bhyve debug backend, GDB server mode, and live kernel-level breakpoints.** These are possible and occasionally useful, but they are an advanced niche and do not belong in the first debugging chapter.
- **Error injection and fault testing frameworks.** `fail(9)`, `fail_point(9)`, and related facilities are briefly introduced in context but left to Chapter 25 and the testing material that follows.

Staying inside those lines keeps Chapter 23 a chapter about how to find bugs, not a chapter about all the places the kernel has ever tried to help you find one.

23.1.5 Estimated Time Investment

- **Reading only:** four to five hours. Chapter 23's ideas are conceptually lighter than Chapter 21 or Chapter 22, but they connect to a lot of tools, and a first pass through the tool survey takes time.

- **Reading plus typing the worked examples:** ten to twelve hours over two or three sessions. The driver evolves in three stages: first the macro-based logging and verbose mode, then the SDT probes, and finally the refactor into its own file pair. Each stage is short, and each stage comes with a short lab that confirms it works.
- **Reading plus all labs and challenges:** fifteen to eighteen hours over three or four sessions. The labs include an intentional ddb session, a DTrace exercise that measures driver latency, a ktrace/kdump walk-through, a deliberate memory leak the reader has to find with `vmstat -m`, and an intentional lock order violation the reader has to reproduce under WITNESS.

Sections 4 and 5 are the densest in terms of new vocabulary. If the ddb commands or DTrace syntax feel opaque on first pass, that is normal. Stop, run the matching exercise on the simulated driver, and come back when the shape has settled. Debugging is one of those skills where concrete practice settles the ideas faster than further reading.

23.1.6 Prerequisites

Before starting this chapter, confirm:

- Your driver source matches Chapter 22 Stage 4 (1.5-power). The starting point assumes every Chapter 22 primitive: the `device_suspend`, `device_resume`, and `device_shutdown` methods; the softc's suspended flag and power counters; the `POWER.md` document; and the regression test script. Chapter 23 builds on top of all of that.
- Your lab machine runs FreeBSD 14.3 with `/usr/src` on disk and matching the running kernel.
- A debug kernel with `INVARIANTS`, `WITNESS`, `WITNESS_SKIPSPIN`, `DDB`, `KDB`, `KDB_UNATTENDED`, `KDTRACE_HOOKS`, and `DDB_CTF` is built, installed, and booting cleanly. These are the exact options Chapter 22 already asked for; Section 4 of this chapter explains each one in detail so you know what they do.
- `bhyve(8)` or `qemu-system-x86_64` is available. Chapter 23's labs deliberately run in a VM. Several of the labs can panic the kernel (that is the lesson), and a VM snapshot is the right tool to keep the feedback loop short.
- The following user-space commands are in your path: `dmesg`, `sysctl`, `kldstat`, `kldload`, `kldunload`, `devctl`, `vmstat`, `ktrace`, `kdump`, `dtrace`, `procstat`, `savecore`, `kgdb`.
- You have snapshotted your VM at the 1.5-power state and named the snapshot clearly. Several labs in this chapter intentionally panic the kernel; you want to be able to roll back to a known good state in under a minute.

If any item above is shaky, fix it now. Debugging, unlike many topics in this book, is a skill that rewards preparation and punishes shortcuts. A lab session that starts without a snapshot is a lab session that ends with an hour of recovery; a DTrace exercise that starts without `KDTRACE_HOOKS` is a DTrace exercise that fails silently.

23.1.7 How to Get the Most Out of This Chapter

Five habits pay off in this chapter more than in any previous one.

First, keep `/usr/src/sys/kern/subr_prf.c`, `/usr/src/sys/kern/kern_ktrace.c`, `/usr/src/sys/kern/subr_witness.c`, and `/usr/src/sys/sdt.h` bookmarked. The first defines the kernel's `printf`, `log`, `vprintf`, and `tprintf` implementations and is the canonical answer to any question about what happens

when you call them. The second is the `ktrace` implementation. The third is `WITNESS`, the lock-order verifier that Chapter 23 leans on heavily. The fourth is the SDT probe macros. Reading each once at the start of the corresponding section is the single most effective thing you can do for fluency. None of these files are long; the longest is a few thousand lines and most of that is comment.

Second, keep three real driver examples close to hand: `/usr/src/sys/dev/ath/if_ath_debug.h`, `/usr/src/sys/dev/virtio/virtqueue.c`, and `/usr/src/sys/dev/iwm/if_iwm.c`. The first illustrates the classic kernel `DPRINTF` macro pattern with compile-time and runtime toggles. The second contains real SDT probes. The third shows how a driver wires a debug `sysctl` through `device_get_sysctl_tree()`. Chapter 23 points back to each at the right moment. Reading them once now, without trying to memorise, gives the rest of the chapter concrete anchors to hang its ideas on.

A note on line numbers. When these real drivers are referenced later, the reference is always anchored on a named symbol: a `DPRINTF` definition, an SDT probe macro, a specific `sysctl` callback. Those names will carry across future FreeBSD 14.x point releases; the lines where they sit will not. Jump to the symbol in your editor and trust what it reports; the example `WITNESS` output later in the chapter quotes source locations for the same reason, as pointers rather than fixed addresses.

Third, type every code change into the `myfirst` driver by hand. Debugging support is the kind of code that is easy to paste and hard to remember later. Typing out the macros, the SDT definitions, the verbose-mode `sysctl`, and the `myfirst_debug.c` refactor builds familiarity that a copy-paste pass does not. The goal is not to have the code; the goal is to be the person who could write it again from scratch in twenty minutes.

Fourth, work inside a VM, keep a snapshot, and do not be afraid to panic the kernel on purpose. Several of the chapter's labs ask you to enter `ddb` on a running system. Several more ask you to build a driver that intentionally misuses a lock, then to watch `WITNESS` catch it. A real panic in a real VM is, for debugging practice, cheap and safe. A real panic on the reader's development laptop is expensive and annoying. The snapshot is the cheap path.

Fifth, after finishing Section 4, re-read Chapter 22's debugging notes on suspend and resume. The suspend and resume paths are classic targets for the Chapter 23 toolkit: lock issues show up in `WITNESS`, DMA state mistakes show up as `KASSERT` failures under `INVARIANTS`, race conditions show up as intermittent `devctl` suspend failures. Seeing the Chapter 22 material in the light of Chapter 23's tools reinforces both chapters.

23.1.8 Roadmap Through the Chapter

The sections in order are:

1. **Why Kernel Debugging Is Challenging.** The mental model: what makes kernel-space bugs different; why the tools from userland do not carry over; why controlled, incremental changes are the right first response; why you always debug from a position of reproducibility.
2. **Logging With `printf()` and `device_printf()`.** The everyday workhorse. Kernel `printf`; `device_printf` as the preferred form for device drivers; the `log(9)` family with priority levels; formatting conventions; where the output actually goes; what to log and what to leave out. Stage 1 of the Chapter 23 driver extends the `myfirst` log lines to use `device_printf` consistently and prepares the ground for the debug macros in Section 8.
3. **Using `dmesg` and `Syslog` for Diagnostics.** Following the output. The kernel message buffer; `dmesg` and its filters; `/var/log/messages` and `newsyslog`; the relationship between console output, `dmesg`, and

syslog; what it means for a message to be “lost”; how to correlate messages across a boot cycle and across a module reload.

4. **Kernel Debug Builds and Options.** The invariants your kernel enforces. options KDB, DDB, KDB_UNATTENDED, KDB_TRACE; INVARIANTS and INVARIANT_SUPPORT; WITNESS and its variants; KDTRACE_HOOKS and DDB_CTF; a survey of the DDB commands a driver author uses most; a brief introduction to kgdb on a crash dump. This is the longest section of the chapter because the debug kernel is the ground the rest stands on.
5. **Using DTrace for Live Kernel Inspection.** The scalpel. What DTrace is; providers and probes; how to list them with `dtrace -l`; short scripts using `fbt`, `syscall`, `sched`, `io`, and `sdt`; how to add SDT probes to the `myfirst` driver. Stage 2 of the chapter driver adds SDT probes.
6. **Tracing Kernel Activity With `kttrace` and `kdump`.** The user-side viewpoint. What `kttrace` records, what it does not, how to invoke it, how to interpret `kdump` output, how to follow a user program as it crosses into the driver through `open`, `ioctl`, `read`, and `write`.
7. **Diagnosing Common Driver Bugs.** The field guide. Memory leaks and `vmstat -m`; lock order violations and WITNESS; `bus_space` and `bus_dma` misuse patterns; post-detach access; interrupt handler mistakes; lifecycle errors; the debug checklist to run before filing a bug.
8. **Refactoring and Versioning With Trace Points.** The clean house. The final split into `myfirst_debug.c` and `myfirst_debug.h`; runtime-toggable verbose mode through `sysctl`; ENTRY/EXIT/ERROR macros; a small SDT probe bank; the `DEBUG.md` document; the version bump to 1.6-debug.

After the eight sections come an extended look at real-world driver debug patterns in `if_ath_debug.h`, `virtqueue.c`, and `if_re.c`, a set of hands-on labs, a set of challenge exercises, a troubleshooting reference, a Wrapping Up that closes Chapter 23’s story and opens Chapter 24’s, a bridge, a quick-reference card, and the usual glossary of new terms.

If this is your first pass, read linearly and do the labs in order. If you are revisiting, Sections 4 and 7 stand alone and make good single-sitting reads. Section 5 assumes Sections 1 through 4; do not read DTrace in isolation on the first pass.

23.2 Section 1: Why Kernel Debugging Is Challenging

Before the tools, the mindset. Section 1 spells out what makes kernel debugging different from what a reader may already know. A reader who has written shell scripts, small C programs, or even a multi-threaded application server has probably used `printf`, `gdb`, `strace`, `valgrind`, `lldb`, or the browser’s devtools. Those tools are excellent in user space, and the mental model they encourage is reasonable in user space. In kernel space, the mental model needs to shift. Section 1 describes that shift, names the specific difficulties driver authors keep running into, and sets the stage for the tool-by-tool work that follows.

23.2.1 The Nature of Kernel Code

A driver is, in the simplest possible statement, C code that runs in kernel address space. That one fact drives most of the differences between kernel debugging and userland debugging.

Kernel address space contains everything: the code the kernel itself runs, the data structures the kernel uses to manage processes and files and memory and devices, the page tables for every process on the system, the

buffers of every pending network packet, the queues of every disk request, and the memory behind every device's mapped registers. Every pointer the driver holds potentially reaches every other part of the kernel. There is no separation like the one user-space programs enjoy, where a process can only touch its own address space and the operating system enforces the boundary.

The consequence is sharp: a single wrong pointer in a driver can corrupt any kernel data structure, not just the driver's own. A driver that writes one byte past the end of its softc may happen to write into the kernel's process table, or into the filesystem's free-block bitmap, or into another driver's interrupt vector table. The corruption often does not produce an immediate panic; it produces a panic five minutes later when some other code reads the structure the driver silently damaged. That delay between cause and symptom is the single most common reason kernel bugs feel harder than userland bugs. You do not catch the wrong pointer at the moment it happens; you catch a different piece of code stumbling over the mess.

Kernel code also runs without the usual safety nets. In user space, dereferencing a null pointer produces a segmentation fault, the kernel kills the process, and the user sees a message. In kernel space, dereferencing a null pointer produces a page fault, the kernel itself faults, and the fault handler decides whether the system can continue. If the page fault happens in a place where it would leave the kernel in an inconsistent state (inside an interrupt handler, inside a critical section, inside `malloc`), the kernel panics to prevent further damage. On a development kernel with `INVARIANTS` enabled, the kernel panics more aggressively than on a release kernel, because the trade-off is exactly the point: catch the bug loudly in development, rather than let it silently propagate.

There is also no standard output stream the way user programs have one. A user-space C program calls `printf` and the output appears on the terminal because the C runtime links the call to a chain that eventually writes to a file descriptor the shell connected to the terminal. In the kernel, there is no shell, no terminal in the user sense, and no C runtime. When a driver calls `printf`, it reaches the kernel's own implementation of `printf`, which writes to the kernel message buffer and, depending on settings, to the system console. Section 2 covers this in detail; the point for now is that the kernel's logging pipeline is not the same pipeline user programs use, and assuming familiarity leads to confusion later. A driver that "just calls `printf`" is calling a different function from the one a userland program calls, with different semantics and a different output path.

23.2.2 Risks of Debugging in Kernel Space

The consequences of a wrong debug change in user space are usually small. A `printf` that prints the wrong variable is an inconvenience. A `printf` inside a loop that runs a billion times slows the program down but does not damage anything. A misuse of `malloc` in a debug statement leaks a few bytes, and the process eventually exits, and the leak disappears with it.

The consequences of a wrong debug change in kernel space can be much larger. A `printf` on the wrong path can deadlock the kernel, if the `printf`'s path needs a lock the calling context has already taken. A `printf` in an interrupt handler that calls into code which may sleep can panic under `INVARIANTS`. A debug message that calls a sleeping allocator (`malloc(..., M_WAITOK)` instead of `M_NOWAIT`) while holding a spin lock will make `WITNESS` very unhappy, and if `WITNESS_KDB` is enabled, will drop the machine into the kernel debugger. A global counter used for debug purposes that is incremented without proper synchronisation is itself a source of bugs. Each of these is a real mistake driver authors make; each is a lesson this chapter will teach.

The nuance is that debugging code is, strictly, code, and it has to be as correct as any other code in the driver. "I'll clean it up later" is not safe in kernel context. A hasty debug change can introduce a new bug that is harder

to find than the original bug. Chapter 23 teaches habits that avoid that trap: use macros that compile out when disabled; use the kernel's own log facility instead of inventing one; wrap conditional blocks so verbose-mode code is paid for only when verbose mode is on; and test debug code at the same level of care as production code.

Another risk is observational: adding debug output can change the behavior you are trying to observe. A race condition that reproduces reliably without debug output may disappear entirely when you add `printf` calls, because the `printf`s serialise through a lock and happen to hide the race window. A performance bug may look different after instrumentation, because the instrumentation itself adds cost. Kernel developers call this the Heisenberg problem of kernel debugging. The chapter's tools are chosen in part because they minimise the problem: DTrace, SDT probes, and KTR add very little overhead when inactive; `device_printf` is chatty but consistent. Sprinkling `printf("HERE\n")` into code is always available, but as a habit it costs more than it helps.

23.2.3 Differences From Userland Debugging

The difference between kernel-space and user-space debugging is worth spelling out tool by tool.

A user can run a program under `gdb`, set a breakpoint, and watch the program stop. You cannot do this directly with a running kernel, because a paused kernel means a paused system. The rough equivalent is `kgdb` against a crash dump, which lets you inspect the kernel's state at the moment of the panic, but which cannot single-step forward. The other rough equivalent is `ddb`, the in-kernel debugger, which lets you pause a live kernel but runs as part of the kernel itself and has its own limited command set. Both are useful; neither behaves the way `gdb` on a live process behaves.

A user can use `strace` on Linux or `truss` on FreeBSD to watch a program's system calls. FreeBSD's `kttrace` is the closest equivalent; it records a trace of a process's syscalls, signals, namei events, I/O, and context switches into a file that `kdump` can read. But `kttrace` sees the user-side of the driver's interface, not the driver's internals. It can show you that the user process called `ioctl(fd, CMD_FOO, ...)`, but not what happened inside the `ioctl` implementation in the driver. Section 6 spends time on `kttrace` precisely because it is the tool that shows the boundary most cleanly.

A user can use `valgrind` to watch for memory errors and leaks in their program. No direct equivalent exists for the kernel. The closest analogs are `vmstat -m` for summarising kernel allocations by type, the `WITNESS` option for lock order errors, `INVARIANTS` for consistency checks the developer wrote into the code, and `MEMGUARD` for a heavier memory-debug allocator. Section 7 walks through the practical patterns.

A user can profile a program with `perf` on Linux or `gprof` on FreeBSD. FreeBSD provides `pmcstat` and DTrace for profiling. DTrace is the more flexible tool for the kind of questions a driver developer asks, and Chapter 23 focuses on DTrace rather than `pmcstat`.

A user can often just re-run a program to see if the bug happens again. Kernel developers often cannot; a kernel that panicked has to be rebooted, and bugs that only appear under load, or that depend on timing, may take hours to reproduce. The response to this difficulty is to build reproducibility into the process: snapshot the VM, script the test case, and log everything. Every tool in this chapter exists to support that response.

One final difference is cultural. The kernel-development community has built a specific vocabulary around debugging that is worth learning on its own. Terms like “lock order”, “critical section”, “preemption disabled”, “epoch”, “giant lock”, “sleepable context”, “non-sleepable context”, and “interrupt context” all have precise

meanings in FreeBSD, and using them correctly is part of being fluent in kernel debugging. Chapter 23 uses them as it goes; the glossary at the end collects the ones that recur.

23.2.4 Importance of Controlled, Incremental Changes

In user space, when a program misbehaves, the first instinct is often to change several things at once and see if the behavior changes. In kernel space this is a losing strategy. Each change is potentially a new bug. Each rebuild costs a VM reboot. Each test cycle eats time. A driver developer who changes five things between compiles spends hours puzzling over which change caused the new symptom, because no single change is isolated.

The right pattern is **controlled, incremental change**. Make one change. Rebuild. Test. Observe the effect. Note what happened. Decide the next change based on the evidence. Move on. This is slower per change and faster per working fix, because you never have to unwind a combined change when one of the five fixes actually introduced a new bug. The discipline feels artificial the first time. After a dozen sessions, it stops feeling artificial and starts feeling like the only way to make progress.

A related discipline is **always start from a known-good state**. Do not debug on a driver that is already in a state you have not verified. Do not start a session by applying six local changes and then running a test. Commit your baseline, verify it works, and only then start the new work. If the new work goes wrong, `git checkout` returns you to known-good, and you can start again. A driver author without this habit spends half of every debugging session wondering whether the baseline or the new change is at fault.

A third discipline is **reproduce before fixing**. A bug that cannot be reproduced cannot be proved fixed. The first job of a debugging session is not to fix the bug; it is to produce a short, reliable way to trigger the bug, ideally in a script. Once the trigger is reliable, the fix is straightforward: apply the candidate, re-run the trigger, observe whether the bug is gone. Without a reliable trigger, you are guessing.

A fourth discipline is **small, scripted reproduction cases**. The difference between a bug that takes an hour to fix and a bug that takes a week is often the size of the reproduction script. A bug you can reproduce with five lines of shell is a bug you can bisect with `git bisect`. A bug you can only reproduce after three hours of complex usage is a bug you will be afraid to even start debugging. Time spent shrinking a reproduction is time well spent.

Chapter 23's labs are deliberately small. The panics are triggered with three lines. The DTrace scripts are five lines. The `kttrace` demonstrations use a one-line C program. That is not accidental: it is the same pattern good driver debugging uses at any scale. Minimal reproductions and minimal changes scale to real bugs.

23.2.5 The Mental Shift

What changes when a developer moves from user space to kernel space is not just the tools; it is the assumptions. In user space, it is reasonable to assume the kernel will catch your mistakes, the runtime will clean up after you, and the worst case is usually a crashed program you can restart. In kernel space, those assumptions are wrong. The kernel catches nothing; the runtime is the thing you are writing; the worst case is a panic that brings down everything.

This is not a reason to be afraid. It is a reason to be careful. Careful, here, has a specific meaning: a driver author writes code that could be debugged by someone else, in six months, from the logs and the source alone. Careful code uses clear function names. Careful code logs what it does at the levels of detail that are useful.

Careful code checks its own invariants. Careful code is structured so that when it breaks, the breakage is obvious. Careful code does not try to be clever; clever code is code you cannot debug under stress.

Chapter 23 teaches the tools of kernel debugging, but the habit it actually teaches is carefulness. Every tool in the chapter is most useful when applied by a careful developer on careful code. The same tool applied to reckless code produces information you cannot interpret. The point is not to memorise DTrace syntax or ddb commands; the point is to develop the habit of debugging from evidence, in small steps, against a known baseline, with good logs, in a way that any collaborator can follow.

23.2.6 Wrapping Up Section 1

Section 1 established the shape of the problem. Kernel code runs in a single shared address space, without the safety nets userland relies on, with failure modes that damage the whole system. Kernel debugging tools are different, more costly, and less interactive than their userland cousins. The discipline of controlled, incremental, reproducible change is not optional; it is the only strategy that works. The mental shift required is real, but it is learnable, and it is the foundation the rest of the chapter's tools stand on.

Section 2 starts the tool survey with the simplest and most universal tool every driver ends up using: the kernel's `printf` family.

23.3 Section 2: Logging With `printf()` and `device_printf()`

The kernel's `printf` family is the humblest debugging tool in the toolkit. It is also the most used, by a wide margin. Almost every FreeBSD driver in `/usr/src/sys/dev` calls `device_printf` somewhere. Almost every boot message a user ever sees was written by a kernel `printf`. Before DTrace, before ddb, before SDT probes, before debug kernels, a driver author reaches for `device_printf` because it is always available, it is always safe (subject to the qualifications this section will spell out), and it is understood by every other driver author on the planet.

This section teaches how to use that family of functions deliberately. The goal is not just to show how to call `device_printf`, because that part is trivial. The goal is to build the habit of using logging as a designed feature of the driver, not as a scattered set of debugging notes that accumulate over time and never get cleaned up.

23.3.1 Basics of `printf()` in Kernel Context

FreeBSD's kernel `printf` lives in `/usr/src/sys/kern/subr_prf.c`. Its signature is familiar:

```
int printf(const char *fmt, ...);
```

It takes a format string and a variable argument list and writes the formatted text into the kernel message buffer. The buffer is a circular area of memory, usually around 192 KB in size on a modern FreeBSD system, into which every kernel message is written. The kernel does not keep a permanent log on its own; the user-space `syslogd(8)` daemon is what reads the buffer and writes selected lines to `/var/log/messages`. Section 3 covers the full pipeline.

The format string understands most of the same conversions userland's `printf` understands: `%d`, `%u`, `%x`, `%s`, `%c`, `%p`, and so on. It understands a few kernel-specific extras, such as `%D` for printing a hex dump of a byte buffer (with a separator argument). It does not understand floating-point conversions like `%f` and `%g`, because the kernel does not use floating-point math by default and the conversion code is intentionally absent. Trying to print a `double` or `float` inside the kernel is almost always a mistake, and the compiler will usually warn about it.

The kernel's `printf` has different safety properties from the userland one. Userland `printf` can allocate, can take locks inside the C runtime, and can block on I/O. The kernel's `printf` must not allocate; it must not block; it must be safe to call from almost any context. The implementation in `subr_prf.c` is deliberately simple: it formats into a small stack buffer, copies the result into the message buffer, wakes up `syslogd` if appropriate, and returns. That simplicity is part of why `printf` is the universal fallback tool.

There are a few contexts where even the kernel's `printf` is not safe. Fast interrupt handlers, for example, run before the kernel has stabilised the console state, and a `printf` from inside a fast interrupt can produce a hang or a deadlock on some platforms. Paths that hold a lock the `printf` path also tries to take can deadlock too. In practice, driver authors avoid the problem by using `printf` only from the ordinary driver entry points (`attach`, `detach`, `ioctl`, `read`, `write`, `suspend`, `resume`) and from interrupt tasks (the deferred half of a filter-plus-task interrupt pipeline, as Chapter 19 introduced), not from the filter itself. The `myfirst` driver already follows this pattern; Chapter 23 preserves it.

23.3.2 Preferred Use: `device_printf()` for Structured, Device-Aware Logging

Raw `printf` is generic. A driver author can do better. The kernel provides `device_printf`, declared in `/usr/src/sys/sys/bus.h`:

```
int device_printf(device_t dev, const char *fmt, ...);
```

The function prefixes every message with the device's name, typically the driver's name concatenated with the unit number: `myfirst0:`, `em1:`, `virtio_blk0:`. That prefix is not cosmetic. It is the single most important piece of information in the kernel log for a multi-device system. Without it, a message like "interrupt received" is useless in a machine with eight network cards and three storage controllers. With it, "myfirst0: interrupt received" is immediately locatable in any tool that reads logs.

The rule is simple: **in driver code, prefer `device_printf` over `printf`**. Every place the driver has a `device_t` in scope, use `device_printf`. Every place the driver does not have a `device_t` (a global init function, for example), use `printf` with an explicit prefix:

```
printf("myfirst: module loaded\n");
```

This is the pattern that makes the kernel log readable across a system. It is also the pattern almost every FreeBSD driver follows. Open any file under `/usr/src/sys/dev/` and the pattern is visible within the first `attach` function.

The `myfirst` driver already uses `device_printf` in most places, because Chapter 18 introduced the convention when the driver first became a PCI device. Section 8 of this chapter wraps `device_printf` in macros that add severity, category, and a verbose-mode toggle, but the underlying function remains the same. Nothing in Chapter 23 replaces `device_printf`; the chapter elaborates on it.

23.3.3 `device_log()` and Log Priority Levels

`device_printf` is plain: it writes the message at the kernel's default log priority (`LOG_INFO`, in syslog terms). Sometimes a driver wants to write at a higher priority, to flag a serious problem, or at a lower priority, to tag a routine informational message that should usually be filtered out.

FreeBSD 14 exposes `device_log` for that purpose, also in `/usr/src/sys/sys/bus.h`:

```
int device_log(device_t dev, int pri, const char *fmt, ...);
```

The `pri` argument is one of the syslog priority constants from `/usr/src/sys/sys/syslog.h`:

- `LOG_EMERG` (0): system is unusable
- `LOG_ALERT` (1): action must be taken immediately
- `LOG_CRIT` (2): critical conditions
- `LOG_ERR` (3): error conditions
- `LOG_WARNING` (4): warning conditions
- `LOG_NOTICE` (5): normal but significant condition
- `LOG_INFO` (6): informational
- `LOG_DEBUG` (7): debug-level messages

For driver work, the four priorities that matter in practice are `LOG_ERR`, `LOG_WARNING`, `LOG_NOTICE`, and `LOG_INFO`. A driver rarely needs `LOG_EMERG`, `LOG_ALERT`, or `LOG_CRIT`, because most driver failures are local to the driver, not to the whole system. `LOG_DEBUG` is useful during active debugging; Chapter 23 uses it for verbose-mode output.

The underlying `log(9)` function, also in `subr_prf.c`, is simpler: `void log(int pri, const char *fmt, ...)`. `device_log` wraps `log` and adds the device-name prefix.

The practical difference between `printf` and `log` is where the output goes when the kernel is configured conservatively. By default, FreeBSD sends everything at or above `LOG_NOTICE` to the console; `LOG_INFO` and below go to the kernel message buffer but not to the console. That matters in two places. First, during boot: a high-priority message is visible on the screen during boot; a `LOG_INFO` message is not. Second, during operation: a serious error on a production system should appear on the console (so a sysadmin watching `tail` on messages sees it immediately), while a routine probe or attach message should not clutter the screen.

The `myfirst` driver in Chapter 23 uses this convention:

- Probe and attach success: `device_printf` (which is `LOG_INFO`).
- Suspend, resume, and shutdown transitions: `device_printf` (routine).
- Errors from hardware, unexpected interrupt counts, DMA failures, and failed device state: `device_log(dev, LOG_ERR, ...)` or `device_log(dev, LOG_WARNING, ...)`.
- Verbose-mode output (enabled by a `sysctl`): `device_log(dev, LOG_DEBUG, ...)` inside the debug macros.

23.3.4 What to Log and What to Leave Out

Knowing what not to log matters as much as knowing what to log. A driver with too much output clutters `dmesg`, wastes message-buffer space, and pushes older informative messages out of the circular buffer before anyone

reads them. A driver with too little output is silent when something goes wrong and gives the reader nothing to work with.

The rule of thumb for the `myfirst` driver in Chapter 23 is:

Always log, at `device_printf` level (`LOG_INFO`):

- A one-line banner at attach, stating the driver name, the version, and any key resources allocated (interrupt type, MSI-X vector count, DMA buffer size). The reader should be able to see at a glance what the driver did.
- A one-line note at detach. Detach is usually silent, but logging its completion is useful when a driver hangs on unload and the reader needs to know the detach was at least started.
- A one-line note at each suspend, resume, and shutdown. Power transitions matter, and the reader often wants to correlate a bug with a transition that preceded it.

Always log, at `device_log` with `LOG_ERR` or `LOG_WARNING`:

- Any failure to allocate a resource (interrupt, DMA tag, DMA buffer, mutex). The error path should always state what failed.
- Any hardware error the driver detects: a response the device did not expect, a timeout, a register value that indicates the hardware has faulted.
- Any state inconsistency the driver catches: a DMA completion for a transfer that was not started, a suspend that finds the device already suspended, a detach that finds the device still holding a client reference.

Never log in the normal steady-state path:

- Every interrupt. Even if the device is quiet, a driver that logs every interrupt floods the log within seconds under real load. Use DTrace or counters for per-interrupt observability.
- Every DMA transfer. Same reason.
- Every read or write call. Same reason.
- Internal function entry and exit in production. Use the verbose-mode macros (Section 8) so the developer can turn entry/exit tracing on or off.

Log under verbose mode (enabled by the Section 8 macros and the debug `sysctl`):

- Function entry and exit for key functions, with a short argument summary.
- Individual interrupt events with the interrupt status register value.
- DMA transfer submissions and completions.
- `sysctl` and `ioctl` operations and the values they returned.

The line between “always” and “verbose” is a judgement call, and different drivers draw it differently. `/usr/src/sys/dev/re/if_re.c` logs very little in production and uses a `re_debug` `sysctl` to enable detailed output. `/usr/src/sys/dev/virtio/block/virtio_blk.c` uses `bootverbose` to gate extra attach-time detail. `/usr/src/sys/dev/ath/if_ath_debug.h` defines a sophisticated per-subsystem mask that selectively enables different categories of output. Chapter 23’s `myfirst` driver adopts a simpler pattern that is enough for teaching purposes and can be extended when the reader starts working on a real driver.

23.3.5 Tracing Function Flow Without Spamming

One of the debugging idioms that is tempting but produces more harm than good is writing a `device_printf` at the start and end of every function. The idea is sound: if you can see the call sequence, you can often deduce what is happening. The practice is wrong, because the sequence produces an overwhelming amount of output, and the output serialises through the `printf` path, which changes the timing of the code under test.

The right pattern is to use a **verbose mode controlled by a `sysctl`**. Define a pair of macros:

```
#define MYF_LOG_ENTRY(sc)      \
do {                          \
    if ((sc)->debug_verbose) \
        device_printf((sc)->dev, "ENTRY: %s\n", __func__); \
} while (0)

#define MYF_LOG_EXIT(sc, err) \
do {                          \
    if ((sc)->debug_verbose) \
        device_printf((sc)->dev, "EXIT: %s err=%d\n", __func__, (err)); \
} while (0)
```

Then, at the top of every function the driver wants to trace, call `MYF_LOG_ENTRY(sc);`. At every return, call `MYF_LOG_EXIT(sc, err);`. When the `debug_verbose` flag is zero (the default), the macros compile down to a branch that is never taken and have essentially zero cost. When the flag is one (enabled at runtime through `sysctl dev.myfirst.0.debug_verbose=1`), the macros produce a trace that any driver developer can read.

The boolean flag shown above is deliberately simple. It introduces the idea of a runtime-controlled knob without requiring the reader to learn a new layout at the same time. Section 8 of this chapter fleshes this idiom out into a full macro bank, replaces the single boolean with a 32-bit bitmask (so the operator can enable just the interrupt trace, or just the I/O trace, rather than everything at once), moves the whole apparatus into `myfirst_debug.h`, and adds SDT probes alongside. For now, the pattern to internalise is that trace-level output should be **off by default** and **runtime toggleable**. Compile-time debug flags (the old `#ifdef DEBUG` style) are a poor second: you have to rebuild the module to change them. Runtime toggles let a developer enable verbose mode on a running system the moment they need it and disable it the moment the evidence is gathered.

23.3.6 Redirecting Logs to `dmesg` and `/var/log/messages`

The kernel's `printf` writes to the kernel message buffer. That buffer is what `dmesg(8)` reads when the user runs it. Section 3 covers `dmesg` in detail; the short version is that the buffer is circular, so old messages eventually get overwritten by new ones as the buffer fills.

The `syslogd(8)` daemon reads new messages out of the buffer and writes them to the appropriate syslog file, usually `/var/log/messages`. The mapping between priority and file is configured in `/etc/syslog.conf`. By default, messages at `LOG_INFO` or higher go to `/var/log/messages`; `LOG_DEBUG` messages are dropped unless explicitly enabled. `/var/log/messages` is rotated by `newsyslog(8)` according to rules in `/etc/newsyslog.conf`; that rotation is what produces the `messages.0.gz`, `messages.1.gz`, and so on that the reader will find in a long-running system's log directory.

The practical consequence is that a driver's log output lives in two places at once. A recent message is in the kernel buffer (accessible through `dmesg`) and in `/var/log/messages` (accessible through any file-reading tool). An old message may have aged out of the buffer but still be in `/var/log/messages` or its archives. A very old message may have rolled out of `/var/log/messages` into a `messages.N.gz` archive, and can be read with `zcat`, `zgrep`, or `zless`. Section 3 walks through these tools.

There is one subtlety worth mentioning here because it trips up new driver authors: **a `LOG_DEBUG` message produced by the kernel is not written to `/var/log/messages` by default**. If the reader's debug output is at `LOG_DEBUG` and they cannot find it in the log file, the cause is usually `/etc/syslog.conf`, not the driver. A quick test: run the driver, and then run `dmesg | grep myfirst`. If the message appears in `dmesg` but not in `/var/log/messages`, the message was produced but was filtered out by `syslog`. Enabling `LOG_DEBUG` in `/etc/syslog.conf` (if `syslog` is configured to care about kernel debug messages) is one fix; using `device_printf` or `device_log(dev, LOG_INFO, ...)` is a more common fix, because it bypasses the problem by using a higher priority.

23.3.7 A Concrete Walkthrough: Logging in the `myfirst` Driver

The `myfirst` driver at the start of Chapter 23 already has log lines in the obvious places: `attach` prints a banner, `detach` confirms completion, `suspend` and `resume` log their transitions, and DMA errors log a warning. Chapter 23's Stage 1 upgrades this in three targeted ways.

First, every `printf` inside the driver that has a `device_t` in scope becomes a `device_printf`. A quick `grep` turns up about eight lines that still use plain `printf` from the module's `MOD_LOAD` and `MOD_UNLOAD` handlers. The module-level ones stay as `printf` (they do not have a `device_t`), but they use the literal `"myfirst: "` prefix so the reader can still find them in the log. A driver that calls `printf` without a prefix in any place is a driver that produces unfindable log lines.

Second, every error path becomes a `device_log` at an explicit priority:

```
/* Before */
if (error) {
    device_printf(dev, "bus_dma_tag_create failed: %d\n", error);
    return (error);
}

/* After */
if (error) {
    device_log(dev, LOG_ERR, "bus_dma_tag_create failed: %d\n", error);
    return (error);
}
```

The change is small, and the behavior is the same in most test kernels. The value is in the priority. A production system watching for `LOG_ERR` messages will now see the driver's errors; a development environment filtering at `LOG_INFO` will see everything.

Third, a new pattern of logging for “rare but important” state changes is introduced. Specifically, the Chapter 23 driver logs at `LOG_NOTICE` on three events the reader may want to know about but does not want every interrupt

for: the first successful DMA transfer after attach, a transition into runtime suspend from idle detection, and the first successful resume after a suspend. These are not errors, but they are not routine either; they are the events a developer reading the log wants to see highlighted.

The exact code changes for Stage 1 are laid out in Section 2's hands-on exercise at the end of this section, and the corresponding files are under `examples/part-05/ch23-debug/stage1-logging/`.

23.3.8 A Note on `uprntf` and `tprntf`

Two sibling functions appear in `subr_prf.c` that driver authors occasionally see but rarely use.

`uprntf(const char *fmt, ...)` writes a message to the current user process's controlling terminal, if there is one, *instead of* to the kernel message buffer. It is useful in rare cases where a driver's `ioctl` handler wants to produce an error message that only the user running the command sees, without polluting the system log. In practice, most drivers return an error code through the usual `errno` mechanism and do not use `uprntf`. The `/usr/src/sys/kern/` tree has a handful of callers, mostly in places where the kernel wants to warn a user about their own action (a security-relevant event, for example).

`tprntf(struct proc *p, int pri, const char *fmt, ...)` writes to a specific process's terminal. It is rarer still. Chapter 23 does not use either function directly. The Chapter 25 logging material returns to them in the context of user-facing diagnostic messages.

23.3.9 Formatting Conventions

A few small formatting habits make log output much easier to read:

1. **End every line with `\n`.** The kernel's `printf` does not add a newline. A message without a newline will run into the next message in `dmesg`, producing output like `myfirst0: interrupt receivedmyfirst0: interrupt received` that is nearly impossible to parse visually.
2. **Use hex for register values, decimal for counters.** `status=0x80a1` is a device register; `count=27` is a counter. Mixing the two conventions in a single driver produces output that is hard to read at a glance. The `%x` and `%d` conversions are the cheap way to keep them visually distinct.
3. **Put the variable values on the right, not the left.** `myfirst0: dma transfer failed (err=%d)` scans better than `myfirst0: %d was the err from dma transfer failed`. Human readers parse English left-to-right, and putting the variable data at the end of the line matches that parse.
4. **Use short tags for repeated message categories.** `INFO:`, `WARN:`, `ERR:`, `DEBUG:` at the start of the content after the device prefix turn a log into something a `grep` can filter. Section 8 uses this convention in its macros.
5. **Do not include timestamps in the message.** The kernel and `syslog` add their own timestamps; a manual timestamp produces duplicate information and visual noise.
6. **Do not include the function name in production messages.** Function names are useful in verbose-mode output (where the entry/exit macros print them automatically) but wasteful in every production message. A message like `myfirst0: DMA completed` is more readable than `myfirst0: my-first_dma_completion_handler: DMA completed`. If you want function names, turn on verbose mode.

7. **Match the driver's style.** If the rest of the driver uses “DMA” in uppercase, use “DMA” in every log line. If the rest uses “dma” in lowercase, match that. Consistency is easier to grep than mixed case.

These are small habits, but they pay off the first time a log becomes the only evidence of a bug that happened on a system the reader cannot access interactively.

23.3.10 Exercise: Add `device_printf()` at Key Points in Your Driver

The hands-on companion for Section 2 is a short exercise. The reader should:

1. Open `myfirst_pci.c`, `myfirst_sim.c`, `myfirst_intr.c`, `myfirst_dma.c`, and `myfirst_power.c` in turn.
2. For every `printf` call that has a `device_t` or a `softc` (which holds a `device_t`) in scope, replace it with `device_printf(dev, ...)`.
3. For every error-path message, change the call to `device_log(dev, LOG_ERR, ...)` if the condition is a real error, or `device_log(dev, LOG_WARNING, ...)` if the condition is recoverable. Keep `device_printf` for routine informational messages.
4. For the attach banner, add a single line that names the driver version: `device_printf(dev, "myfirst PCI driver %s attached\n", MYFIRST_VERSION);`. Define `MYFIRST_VERSION` in the top of `myfirst_pci.c` as a string literal. Chapter 23 will bump it from "1.5-power" to "1.6-debug" at the final refactor in Section 8; for now, keep it at "1.5-power-stage1".
5. Rebuild the module, load it, and verify that `dmesg | grep myfirst0` shows the attach banner. Generate a contrived DMA failure (the simulator's `sim_force_dma_error` sysctl is one way; Chapter 17 introduced the hook) and verify that the error message appears with the `LOG_ERR` priority.
6. Commit the changes with a message like “Chapter 23 Section 2: use `device_printf` consistently”.

The reader should feel free to check their work against `examples/part-05/ch23-debug/stage1-logging/myfirst_pci.c` and the other `stage1` files, but the learning happens in the typing, not in the checking.

23.3.11 Wrapping Up Section 2

Section 2 established the ground rules for logging. Use `device_printf` when you have a `device_t`. Use `device_log` with an explicit priority when the priority matters. Log enough to make the log readable, and not so much that the log becomes noise. Route routine information at `LOG_INFO`, real errors at `LOG_ERR`, and verbose tracing at `LOG_DEBUG` gated behind a sysctl.

Section 3 now follows the log output from the moment the driver writes it to the moment a user reads it, through `dmesg`, the kernel message buffer, `/var/log/messages`, and the syslog pipeline. The driver's job ends when it calls `device_printf`; the tools the reader uses to see what the driver said begin from there.

23.4 Section 3: Using `dmesg` and Syslog for Diagnostics

Section 2 ended with the driver's `device_printf` call. Section 3 picks up from the other end: the moment a user runs `dmesg`, opens `/var/log/messages`, or greps through syslog archives. In between sits the kernel's

message buffer, the `syslogd` daemon, the `newsyslog` rotator, and a small set of configuration files that quietly determine which messages appear where. Understanding the pipeline is how a driver author stops worrying about “why don’t my messages appear?” and starts debugging the actual driver.

23.4.1 The Kernel Message Buffer

Every kernel message the driver produces goes first into the **kernel message buffer**. This is a single circular region of memory the kernel allocates at boot, sized by default to about 192 KB on amd64 FreeBSD 14. The size is set by the `msgbufsize` tunable; the current size is visible as:

```
sysctl kern.msgbufsize
```

The buffer is “circular” in the sense that when it fills, new messages overwrite the oldest. This matters for a reader hunting an old bug: if you rebooted five hours ago, produced a bunch of driver log output, and only now went looking for it, the output may have rolled out of the buffer already. The safe assumption is that the buffer holds several days of routine messages on a quiet system and a few hours of messages on a busy one.

Because the buffer is a single block of memory, reading it is inexpensive. The `dmesg(8)` command works by calling the `sysctl(8)` function to read `kern.msgbuf` and printing the contents. The `dmesg` binary is small; its source is at `/usr/src/sbin/dmesg/dmesg.c` and worth a quick look if the reader is curious.

The boot-time portion of the buffer is preserved more carefully than the runtime portion. During boot, the kernel writes every `autoconf` message, every driver’s attach banner, every hardware detection note, into the buffer. The `/var/run/dmesg.boot` file, produced by the `/etc/rc.d/dmesg` startup script, captures the buffer’s contents at the moment the boot sequence finishes and before runtime output starts overwriting it. Reading `dmesg.boot` is the way to see boot messages after the running system has long since pushed them out of the live buffer:

```
cat /var/run/dmesg.boot | grep myfirst
```

For a driver author, `dmesg.boot` is the answer to “did my driver attach on boot?” when the answer is needed hours later. The live `dmesg` is the answer to “what has my driver said since then?”.

23.4.2 dmesg in Practice

The common invocations are worth committing to memory.

`dmesg` with no arguments prints the current kernel buffer contents in order from oldest to newest. On a system that has been running for a while, this is a lot of output:

```
dmesg
```

Pipe through `less` to scroll, or through `grep` to filter:

```
dmesg | less
dmesg | grep myfirst
dmesg | grep -i error
dmesg | tail -50
```

`dmesg -a` prints all messages including those at priorities the kernel is configured to suppress from the normal output. On some configurations this surfaces additional low-priority detail. On most configurations the output is the same as plain `dmesg`.

`dmesg -M /var/crash/vmcore.0 -N /var/crash/kernel.0` reads the message buffer out of a saved crash dump, rather than from the live kernel. This is how a reader looks at the last few messages a kernel produced before it panicked, when the panic itself prevented the normal logging path from running. Section 4's `kgdb` material returns to this.

The `dmesg -c` form (which on some systems clears the buffer after reading) is not available in the FreeBSD kernel's way of exposing the buffer, because the buffer is not drained by reading it; reading produces a copy, and the live buffer continues to receive new messages. Users coming from Linux sometimes expect `dmesg -c` and are surprised.

A useful habit during driver development is to run `dmesg | grep myfirst | tail` after every module load, every test, and every interesting event. The repetition builds a reflex of “what did the driver just say?” that pays off the first time something unexpected happens.

23.4.3 Permanent Logs in `/var/log/messages`

The kernel message buffer is volatile; a reboot empties it. For anything longer-term, FreeBSD uses `syslogd(8)`, the standard UNIX syslog daemon, which reads new messages as the kernel produces them and writes selected ones to files on disk.

The canonical file, on almost every FreeBSD system, is `/var/log/messages`. This file receives most kernel messages at priority `LOG_INFO` or higher, and most userland daemon messages too. The mapping between priority and destination file is in `/etc/syslog.conf`. A relevant excerpt looks like:

```
*.notice;kern.debug;lpr.info;mail.crit;news.err    /var/log/messages
```

The syntax is “facility.priority” pairs. `kern.debug` means “all kernel messages at priority `LOG_DEBUG` or higher”. `*.notice` means “all facilities at priority `LOG_NOTICE` or higher”. The reader should know this syntax exists; understanding it is useful when a specific kind of message needs to go to a specific file.

Because `/var/log/messages` is an ordinary text file, every tool in the reader's toolkit works on it:

```
tail -f /var/log/messages
tail -50 /var/log/messages
grep myfirst /var/log/messages
less /var/log/messages
wc -l /var/log/messages
```

Particularly useful during driver development:

```
tail -f /var/log/messages | grep myfirst
```

This gives a live stream of only the driver's output, which is what most debugging sessions actually need.

When messages age out, they move to `/var/log/messages.0`, then `.1`, then `.N.gz` as rotation happens. The rotation is done by `newsyslog(8)`, which is invoked by a periodic job from `cron` and reads its rules from `/etc/newsyslog.conf`. A typical rule for `/var/log/messages` rotates the file when it reaches a size threshold, keeps some number of compressed archives, and restarts `syslogd` to reopen the new file.

To search through the archives, `zgrep` is the tool:

```
zgrep myfirst /var/log/messages.*.gz
```

Or, for the whole history including the current file:

```
grep myfirst /var/log/messages && zgrep myfirst /var/log/messages.*.gz
```

Readers who are hunting a bug that happened days ago reach for these commands routinely.

23.4.4 Console Output Versus Buffered Output

One nuance confuses many driver authors the first time they encounter it. The kernel has two audiences for its messages: the in-memory buffer, and the system console. The console is the physical terminal on a laptop (or the VGA console in a VM, or the serial console in a headless box). A message can go to one, the other, or both, depending on its priority and the current console log level.

The console log level is controlled by the `kern.consmgbuf_size` (how much memory the console uses) and more importantly `kern.msgbuflock` and the console priority filter. The simplest knob is:

```
sysctl -d kern.log_console_level
```

On FreeBSD, high-priority messages (`LOG_WARNING` and above) are always displayed on the console. Lower-priority messages are buffered but not displayed. This is why a panic message appears on the console regardless of runlevel: it has priority `LOG_CRIT` or higher, and the kernel's console output path bypasses the usual buffering.

For a driver author, the practical consequence is that `device_log(dev, LOG_ERR, ...)` is more likely to be noticed by a human than `device_printf(dev, ...)`, because the former reaches the console and the latter goes only to the buffer. Chapter 23's Section 2 pointed this out; Section 3 reinforces it with the mechanics.

A driver that produces a torrent of `LOG_ERR` messages floods the console and, in extreme cases, slows the system down because the console output path is not cheap. Rate limiting is the cure; Chapter 25 teaches it properly. For now, the lesson is that priority matters.

23.4.5 Reviewing Logs in Practice

The common pattern for reading driver logs during development looks like this:

1. Load the module.
2. Run the test.
3. Unload the module.
4. Copy dmesg output to a file for analysis.

The sequence in shell form:

```
# Clean slate
sudo kldunload myfirst 2>/dev/null
sudo kldload ./myfirst.ko

# Test
dev_major=$(stat -f '%Hr' /dev/myfirst0)
echo "Running driver test..."
some_test_program

# Capture
dmesg | tail -200 > /tmp/myfirst-test-$(date +%Y%m%d-%H%M%S).log

# Clean slate again
sudo kldunload myfirst

# Now inspect the log at leisure
less /tmp/myfirst-test-*.log
```

Several habits that this pattern encourages are worth stating explicitly:

Clean slate first, every time. Unload any existing instance of the module before loading the one you want to test. Otherwise you may accidentally be running an older compile and wondering why your new code doesn't have the expected effect. An unload also leaves the dmesg buffer more readable because your new session starts from a known state.

Capture every interesting session. A log you saved to a file is a log you can diff, grep, share, and return to next week. A log you left in the live buffer is a log that may get pushed out by the time you come back to it.

Use timestamps in filenames. The date `+%Y%m%d-%H%M%S` convention in the snippet produces filenames like `myfirst-test-20260419-143027.log`. When you accumulate a dozen log files from a day of testing, this is the only way to find the one you want.

Narrow the tail -n to a reasonable window. The default is 10 lines, which is almost never enough for a driver test. 200 to 500 is usually the right size: long enough to capture a full test cycle, short enough to read in one pass. If your test produces more output than that, either your test is too large or your logging is too chatty.

23.4.6 Filtering Output

The `grep` toolchain is your friend. A few patterns recur often enough to be worth building into muscle memory.

Show only the driver’s output:

```
dmesg | grep '^myfirst'
```

The `^myfirst` anchor matches only lines that start with the device name. Without the anchor, you would also match lines that happen to contain “myfirst” in their text (for example, kernel messages about the module’s file descriptor).

Show only errors:

```
dmesg | grep -iE 'error|fail|warn|fault'
```

Case-insensitive, with an extended regex that matches several common error words.

Show a specific interval:

```
awk '/START OF TEST/,/END OF TEST/' /var/log/messages
```

This relies on the reader’s test harness printing “START OF TEST” and “END OF TEST” markers at the boundaries. A script that does this is a script whose logs are easy to slice.

Extract just the timestamps and first word:

```
awk '{print $1, $2, $3, $5}' /var/log/messages | grep myfirst
```

For high-level scanning of a busy log.

Count how many of each message type:

```
dmesg | grep myfirst | awk '{print $2}' | sort | uniq -c | sort -rn
```

This counts messages grouped by their second word, which (with the convention in Section 2) is usually the category like INFO, WARN, ERR, or DEBUG. Useful for spotting “why is my driver producing 30,000 DEBUG messages and 2 INFO messages?”

23.4.7 Permanent Configuration: `/etc/syslog.conf` and `/etc/newsyslog.conf`

For most driver work, the default syslog configuration is fine. Chapter 23 does not ask the reader to change it. But it is worth knowing the two files exist and roughly what they do.

`/etc/syslog.conf` is read by `syslogd` at startup and on `SIGHUP`. It contains rules that map “facility and priority” to “destination”. Facility is the kind of producer (`kern`, `auth`, `mail`, `daemon`, `local0` through `local7`),

and priority is the syslog priority level. A driver's messages always have facility kern (because they come from the kernel) and the priority the driver specified.

A useful experiment, after the reader has read Chapter 23, is to add a line like:

```
kern.debug                /var/log/myfirst-debug.log
```

to `/etc/syslog.conf`, touch the file, and reload syslogd with `service syslogd reload`. Now every kernel debug-level message goes to a separate file. If the reader then enables verbose mode on the `myfirst` driver (Section 8's `sysctl`), the separate file captures only the verbose output and the main messages file stays clean. This is a trick worth knowing; it lets a reader enable deep verbose logging without flooding `/var/log/messages` with everything.

`/etc/newsyslog.conf` is read by `newsyslog` when it rotates logs. A typical rule looks like:

```
/var/log/messages    644  5    100    *    JC
```

The fields, left to right: file path, mode, number of rotations to keep, max size in KB, rotation time, flags. For the `myfirst-debug.log` file the reader just added, an appropriate `newsyslog` rule keeps the logs from growing without bound:

```
/var/log/myfirst-debug.log 644  3   500    *    JC
```

Again, this is not a chapter requirement; it is a pattern worth knowing.

23.4.8 Correlating Messages Across a Boot Cycle

One of the common debugging tasks is “the driver worked after the last boot but now it doesn't”. The reader wants to compare what happened at the last boot to what is happening now. Two paths:

Path one: read `/var/run/dmesg.boot`, which captured the boot-time messages, and compare to the current `dmesg`:

```
diff <(grep myfirst /var/run/dmesg.boot) <(dmesg | grep myfirst)
```

This shows exactly which messages are new since boot. Useful for finding runtime surprises.

Path two: `grep` through the archives of `/var/log/messages`:

```
zgrep myfirst /var/log/messages.*.gz | head -100
```

This shows the driver's history over the last few days, across reboots. Useful for finding patterns like “it started failing on Tuesday”.

A driver that logs its version at attach makes this much easier. If `dmesg.boot` shows `myfirst0: myfirst PCI driver 1.5-power attached` and the current boot shows `myfirst0: myfirst PCI driver 1.6-debug attached`, the reader knows immediately that the two are different versions and that a code change happened between them. Chapter 23's Stage 1 exercise adds exactly this version-at-attach line for that reason.

23.4.9 Exercise: Generate a Known Driver Message and Confirm It Appears

A small hands-on companion for Section 3. The reader should:

1. Rebuild and load the `1.5-power-stage1` driver from Section 2's exercise. Confirm the attach banner appears in `dmesg`.
2. Trigger the simulator's `sim_force_dma_error=1` sysctl to make the next DMA transfer fail. Run a DMA transfer. Confirm the error message appears both in `dmesg` and in `/var/log/messages`.
3. Produce a verbose informational line: set the (not yet existing) `debug_verbose` sysctl to 1. Because the sysctl does not yet exist, this step will fail, and the reader should note which way it failed. This is intentional; Section 8 adds the sysctl.
4. For the lines that did appear, use `awk` or `grep` to extract only the ones with `myfirst0:` prefix, and count them.
5. Create a `/var/log/myfirst-debug.log` file by adding a `kern.debug` line to `/etc/syslog.conf` as sketched above. Reload `syslogd`. Confirm the file is created. Confirm no messages appear in it yet (because the `myfirst` driver does not yet produce `LOG_DEBUG` output). Make a note to return to this in Section 8.

The exercise is short. Its value is that the reader has now actually used `dmesg`, `/var/log/messages`, `grep`, and `syslog.conf` on real driver output. Future chapters will assume these tools are familiar.

23.4.10 Wrapping Up Section 3

Section 3 followed the log from the moment the driver calls `device_printf` or `device_log` to the moment a user sees the output. Messages go first into the kernel's circular message buffer, accessible through `dmesg`. A boot-time snapshot is preserved in `/var/run/dmesg.boot`. The `syslogd` daemon reads new messages and writes them to `/var/log/messages` according to the rules in `/etc/syslog.conf`. The `newsyslog` daemon rotates the files according to `/etc/newsyslog.conf`. Console output is automatic for high-priority messages and gated for low-priority ones.

With the logging pipeline clear, Section 4 turns to the other essential piece of the debugging foundation: the debug kernel. A driver whose bugs always panic quietly and produce nothing for the log is a driver that cannot be debugged. A debug kernel makes those silent bugs loud.

23.5 Section 4: Kernel Debug Builds and Options

A debug kernel is not a different kernel; it is the same kernel with additional checks, additional debug aids, and a small set of features compiled in that a release kernel does not have. The driver source does not change. The user-space tools do not change. What changes is that the running kernel now catches more of its own bugs, preserves more context when a panic happens, and offers more ways to inspect itself.

This section is the longest in the chapter because the debug kernel is the ground the rest of the chapter stands on. Every later tool depends on having the right options set. `DTrace` depends on `KDTRACE_HOOKS`. `ddb` depends on `DDB` and `KDB`. `INVARIANTS` catches bugs that would otherwise produce silent memory corruption. `WITNESS`

catches lock order violations that would otherwise produce intermittent panics weeks later. Each option has a cost, a benefit, and a context in which it is appropriate; Section 4 walks through them.

23.5.1 The Core Debug Options

FreeBSD's kernel is configured through a kernel config file, such as `/usr/src/sys/amd64/conf/GENERIC`. The config file is a list of options lines that enable or disable compile-time features. A debug kernel is produced by starting from `GENERIC` (or another baseline config), adding debug options, rebuilding, installing, and booting the result.

The options that matter for driver debugging fall into four groups: the kernel debugger, the consistency checks, the lock debug infrastructure, and the tracing infrastructure.

Kernel debugger group. These enable the in-kernel debugger `ddb` and the infrastructure that supports it.

- `options KDB`: enables the Kernel Debugger framework. This is the umbrella option that allows any backend (DDB, GDB over serial, etc.) to attach.
- `options DDB`: enables the in-kernel DDB debugger. DDB is the backend most driver authors use. It runs inside the kernel and understands kernel data structures.
- `options DDB_CTF`: enables loading of CTF (Compact C Type Format) data into the kernel, which lets `ddb` and `DTrace` print type-aware information about kernel structures. Needs `makeoptions WITH_CTF=1` in the config.
- `options DDB_NUMSYM`: enables symbol-number lookups inside DDB. Useful for poking at specific addresses.
- `options KDB_UNATTENDED`: on a panic, do not pause to enter `ddb`. Instead, the kernel dumps core (if `dumpdev` is configured), reboots, and continues. This is the right setting for lab VMs where the reader does not want to lose the kernel to a debugger session on every panic.
- `options KDB_TRACE`: on any entry into the debugger (panic or otherwise), automatically print a stack trace. Saves a step in almost every diagnostic session.
- `options BREAK_TO_DEBUGGER`: allows a break on the serial or USB console to drop into `ddb`. Useful on headless test systems.
- `options ALT_BREAK_TO_DEBUGGER`: alternative key combination for the same effect.

A typical debug kernel config includes all of these. The `GENERIC` kernel on `amd64` already enables `KDB`, `KDB_TRACE`, and `DDB`; the additional options the reader enables are `KDB_UNATTENDED`, `INVARIANTS`, `INVARIANT_SUPPORT`, `WITNESS`, and friends.

Consistency check group. These enable extra runtime assertions in the kernel code.

- `options INVARIANTS`: enable assertions in the kernel. Almost every `/usr/src/sys/*` file contains `KASSERT` calls, and those `KASSERT` calls only do something when `INVARIANTS` is enabled. Without `INVARIANTS`, a `KASSERT(ptr != NULL, ("oops"))` compiles to nothing. With `INVARIANTS`, it panics the kernel on failure. The cost is measurable (maybe 10% of kernel CPU time on a busy system) but usually negligible for driver development, because the kernel spends less time per operation than the application using it. The benefit is that bugs the driver would otherwise silently corrupt data over are caught at the moment they happen.
- `options INVARIANT_SUPPORT`: the machinery that `INVARIANTS` itself uses. Required if `INVARIANTS` is enabled; also required separately if any loadable module was built with `INVARIANTS`. Forgetting this

produces “unresolved symbol” errors at module load.

- `options DIAGNOSTIC`: light-weight additional checks, less aggressive than `INVARIANTS`. Some subsystems use this for checks that are cheap enough to run in production but too slow for tight loops.

Lock debug group. These enable the infrastructure that verifies the driver’s synchronisation discipline.

- `options WITNESS`: enable the lock-order verifier. `WITNESS` tracks every lock the kernel takes, records the order in which locks are acquired, and complains loudly when a thread acquires locks in an order that could deadlock if another thread held them the opposite way. A driver with a lock-order bug will produce a `WITNESS` report the first time the bad order is exercised, even if no actual deadlock occurs. This is an enormous debugging win, because it catches the bug before the deadlock manifests.
- `options WITNESS_SKIPSPIN`: skip the lock-order check for spin locks. Spin locks have their own constraints that the general checker is not designed to evaluate, and the checker can produce false positives on them. Enabling `WITNESS_SKIPSPIN` keeps the checker useful for the common mutex case.
- `options WITNESS_KDB`: on a lock-order violation, drop immediately into `ddb` instead of just logging. Aggressive; appropriate for a VM where a manual `ddb` session is easy.
- `options DEBUG_LOCKS`: additional debugging of the generic lock API (separate from `WITNESS`). Catches use of uninitialised locks, locks taken in the wrong context, and related issues.
- `options LOCK_PROFILING`: instrumentation that lets `lockstat` measure lock contention. Not strictly a debug option (it is a profiling option), and expensive; use it only when looking for contention, not by default.
- `options DEBUG_VFS_LOCKS`: VFS-specific lock debugging. Useful only for filesystem driver work.

Tracing infrastructure group. These enable the `DTrace` and `KTR` frameworks.

- `options KDTRACE_HOOKS`: the kernel-wide `DTrace` hooks. Without this, `DTrace` has nothing to attach to. Lightweight even when no `DTrace` script is running, because the hooks are mostly no-op function pointer slots.
- `options KDTRACE_FRAME`: unwind frame information that `DTrace` uses for `stack()` actions. Needed for meaningful stack traces inside `DTrace` scripts.
- `makeoptions WITH_CTF=1`: a `makeoptions` (not `options`) line that enables CTF generation for the kernel. CTF is the type-data `DTrace` and `DDB` use to understand kernel structure layouts.
- `options KTR`: the in-kernel event-ring-buffer tracer. Different from `ktrace`; `KTR` is a high-performance in-kernel buffer of kernel events that is mostly useful for low-level kernel hackers. Most driver authors do not need `KTR` and can leave it off; Chapter 23 mentions it only to name it correctly.
- `options KTR_ENTRIES`, `KTR_COMPILE`, `KTR_MASK`: the `KTR` configuration knobs. See `/usr/src/sys/conf/NOTES` for the full list.

For Chapter 23 the reader needs `KDB`, `DDB`, `DDB_CTF`, `KDB_UNATTENDED`, `KDB_TRACE`, `INVARIANTS`, `INVARIANT_SUPPORT`, `WITNESS`, `WITNESS_SKIPSPIN`, `KDTRACE_HOOKS`, and `makeoptions DEBUG=-g` for symbols. If the reader followed Chapter 22’s prerequisites, these are already enabled.

23.5.2 Building a Debug Kernel

Building a custom kernel is a standard FreeBSD operation. The workflow:

1. Copy the `GENERIC` config to a new name:

```
cd /usr/src/sys/amd64/conf
sudo cp GENERIC MYDEBUG
```

2. Edit MYDEBUG to add the debug options. A minimal addition:

```
ident MYDEBUG

options INVARIANTS
options INVARIANT_SUPPORT
options WITNESS
options WITNESS_SKIPSPIN
options KDB_UNATTENDED

makeoptions DEBUG=-g
makeoptions WITH_CTF=1
```

DDB, KDB, KDTRACE_HOOKS, and DDB_CTF are already in GENERIC on amd64 as of FreeBSD 14, so they do not need to be re-added. If you are on a non-x86 architecture, check the corresponding conf/GENERIC file to see what is already included.

A small but frequent point of confusion: /usr/src/sys/amd64/conf/ holds the kernel *configuration source files*. The older /usr/src/sys/amd64/compile/ directory some tutorials mention is a build-output directory that config(8) historically populated; on a modern FreeBSD system the actual build products live under /usr/obj/usr/src/amd64.amd64/sys/<KERNCONF>/, and the compile/ path is no longer the place you edit or look for source.

3. Build the kernel:

```
cd /usr/src
sudo make -j4 buildkernel KERNCONF=MYDEBUG
```

On a reasonable VM this takes ten to twenty minutes. On a slower VM, half an hour. The build is the longest single step in the debug workflow, but it runs once per config change, not once per test.

4. Install the new kernel:

```
sudo make installkernel KERNCONF=MYDEBUG
```

This copies the kernel files into /boot/kernel and backs up the previous kernel to /boot/kernel.old. If the new kernel fails to boot, the reader can select kernel.old from the boot loader's menu and continue.

5. Reboot.
6. Verify:

```
uname -a
```

The output should mention MYDEBUG in the kernel's identification string.

7. Confirm debug options are active:

```
sysctl debug.witness.watch
sysctl kern.witness
```

On a WITNESS-enabled kernel, these produce output. On a non-WITNESS kernel, the sysctls do not exist.

The total time from “want a debug kernel” to “running a debug kernel” is usually under an hour. The time invested is recovered within the first debugging session, because the debug kernel is the kernel that catches bugs.

23.5.3 When to Use INVARIANTS

INVARIANTS is the most important single debug option for a driver developer. It is also the most misunderstood.

When a kernel programmer writes `KASSERT(condition, ("message %d", val))`, the intent is: this condition must be true at this point; if it is ever false, something has gone wrong in a way the code cannot recover from. Without INVARIANTS, KASSERT compiles to nothing; the condition is not checked. With INVARIANTS, KASSERT evaluates the condition and calls `panic` if it fails, printing the message.

Consider a concrete example from the `myfirst` driver. The DMA submission path (Chapter 21) holds a buffer lock, sets an in-flight flag, and calls `bus_dmamap_sync`. A bug that calls the submission path twice without waiting for completion is a double-submit bug, and it can corrupt DMA. A KASSERT at the top of the submission path catches it:

```
static int
myfirst_dma_submit(struct myfirst_softc *sc, ...)
{
    KASSERT(!sc->dma_in_flight,
        ("myfirst_dma_submit: previous transfer still in flight"));
    ...
}
```

Without INVARIANTS, this line does nothing. With INVARIANTS, the first time a bug submits a second transfer before the first completes, the kernel panics with the message. The reader looks at the backtrace, sees the double-submit, and fixes the bug.

This is the shape of INVARIANTS’s value: it turns silent corruption into loud panics. The trade-off is that a driver author who writes no KASSERTs gets no benefit from INVARIANTS; the option is only useful if the code it runs against has assertions. The `myfirst` driver at Stage 1 has a few; Section 7 adds more; Section 8 moves them into the debug header.

Other assertion helpers exist alongside KASSERT:

- `MPASS(expr)`: a compact form of `KASSERT(expr, (...))`. Expands to the same check with a compiler-provided message. Useful when the condition is self-explanatory.
- `VNASSERT(expr, vp, msg)`: assertion specialised for vnodes. Driver authors rarely use this directly.
- `atomic_testandset_int`, `atomic_cmpset_int`: not assertion helpers, but common operations in driver synchronisation that have their own debug-aware variants.

Read KASSERT in its context: it is a documentation aid as much as a debug aid. A reader looking at an unfamiliar function's source finds the KASSERTs at the top and immediately knows the invariants the code expects. A driver that accumulates KASSERTs over time becomes self-documenting in a way plain prose does not match.

23.5.4 When to Use WITNESS and DEBUG_LOCKS

WITNESS is the other half of the driver-debugging pair. Where INVARIANTS catches logic bugs, WITNESS catches synchronisation bugs.

The basic idea: every lock has an identifier, the kernel records the set of locks each thread holds, and when a thread tries to acquire a lock, WITNESS checks whether the order of acquisition is consistent with previous acquisitions in the kernel's history. If thread A takes lock X then lock Y, and later thread B takes lock Y then lock X, the two threads form a potential deadlock: if A holds X waiting for Y while B holds Y waiting for X, neither can proceed. WITNESS flags the inconsistency the second time a thread tries to acquire the locks in the wrong order, even if the actual deadlock does not happen during the test.

A WITNESS complaint looks something like:

```
lock order reversal:
 1st 0xfffff80001a23200 myfirst_sc (myfirst_sc) @
↪ /usr/src/sys/dev/myfirst/myfirst_pci.c:523
 2nd 0xfffff80001a23240 some_other_lock (some_other_lock) @ /some/other/file.c:789
witness_order_list_add: lock order reversal
```

The message identifies both locks, their addresses, the files and line numbers where they were acquired, and the reversal. A driver author reads this and immediately knows where to look.

WITNESS interacts with the other debug options. WITNESS_KDB drops into ddb on a violation instead of just logging. WITNESS_SKIPSPIN skips spin locks, which have different rules (they cannot sleep, they have priority handling, and the ordering discipline is slightly different). DEBUG_LOCKS adds extra checks on top of WITNESS: detection of double-unlock, detection of unlock by the wrong thread, detection of use of uninitialised mutexes. For driver development, enabling all three (WITNESS, WITNESS_SKIPSPIN, DEBUG_LOCKS) is the default recommendation.

The cost of WITNESS is measurable. Every lock acquisition adds a few dozen instructions and one cache line of memory touch. On a driver-heavy workload this can be 20 or 30 percent slower than a non-debug kernel. For development and testing, this is acceptable. For a production kernel, WITNESS is usually off.

Section 7's common-bug walkthrough includes a WITNESS lab: the reader deliberately introduces a lock order violation into the `myfirst` driver, rebuilds, reloads, triggers the path, and watches WITNESS catch it. That lab is the fastest way to internalise how the tool works.

23.5.5 When to Use DEBUG_MEMGUARD and MEMGUARD

MEMGUARD is a heavier memory-debug allocator. It allocates memory with guard pages on either side, detects writes past the allocation boundary, and detects use-after-free. It is enabled with options `DEBUG_MEMGUARD` and configured at runtime through `sysctl vm.memguard.desc`.

MEMGUARD is expensive: it uses far more memory than a normal allocator because of the guard pages, and its allocation path is slower. The right use is targeted: you enable it for a specific malloc type the driver uses heavily, not for the whole kernel. If the `myfirst` driver has a malloc type `M_MYFIRST`, the reader can enable MEMGUARD for just that type:

```
sysctl vm.memguard.desc="M_MYFIRST"
```

After a reboot (MEMGUARD needs to initialise its arena at boot), allocations of `M_MYFIRST` memory will go through MEMGUARD. A write past the end of an allocation panics immediately. A use after free panics immediately. The reader can run tests and have the kernel catch heap bugs in the driver's memory without slowing the rest of the system down.

Chapter 23 does not make heavy use of MEMGUARD, because the `myfirst` driver's memory patterns are simple. Section 7 mentions it as a tool in the kit for drivers that do more complex allocation. For the reader who eventually writes a driver with dynamic queues, buffer pools, or long-lived allocations, MEMGUARD is worth returning to.

23.5.6 Trade-Offs: Performance vs. Debug Visibility

The temptation, when first learning the options, is to enable everything. This is the right choice during development. It is the wrong choice for production.

A kernel with `INVARIANTS`, `WITNESS`, `DEBUG_LOCKS`, `KDTRACE_HOOKS`, `DDB`, and `MEMGUARD` on is noticeably slower than `GENERIC`: maybe 20% slower overall, 50% slower on lock-heavy workloads, depending on what the machine is doing. On a development workstation, 20% slower is fine. On a production server, 20% slower means 20% more hardware to do the same work.

The distinction between debug kernels and production kernels is sharp in practice. Developers run debug kernels, because they care about finding bugs. Users run release kernels, because they care about performance. A driver that is correct under `WITNESS` and `INVARIANTS` is correct, period; those options do not change driver behavior, they only check it. A driver that passes a `WITNESS` lab run under a debug kernel will also pass under a release kernel. This is the invariant that makes the pattern work: you develop against a strict kernel and deploy against a permissive one, and the strict kernel's extra checks are what keep the permissive one correct.

For the reader, the practical advice: run a debug kernel on the VM you use for driver work, all the time. Do not switch to a release kernel for daily development. The feedback on driver mistakes is worth the performance cost a hundred times over. When you are ready to test the driver on a production-style kernel (for timing measurements, for integration tests), boot a release kernel for that specific test, confirm correct behavior, and then go back to the debug kernel.

23.5.7 A Survey of `ddb` Commands

DDB is the in-kernel debugger. When the kernel panics on a system where DDB is compiled in, instead of rebooting immediately, the system drops into a debugger prompt on the console:

```
KDB: enter: panic
[ thread pid 42 tid 100049 ]
Stopped at: kdb_enter+0x3b:  movq    $0, kdb_why
db>
```

At that prompt, the reader can inspect the kernel's state. The commands are terse and direct. A full tour belongs in a separate reference; what follows is the list a driver author uses most.

Inspection commands:

- `bt` or `backtrace`: show the current thread's backtrace. This is the first command the reader runs on every panic.
- `ps`: show the process table. Useful for seeing what was running.
- `show thread <tid>`: show details of a specific thread.
- `show proc <pid>`: show details of a specific process.
- `show pcpu`: show per-CPU state, including the current thread on each CPU.
- `show allpcpu`: show all per-CPU state at once.
- `show lockchain <addr>`: show the chain of threads blocked on a specific lock.
- `show alllocks`: show all locks currently held, across all threads. Useful for diagnosing apparent deadlocks.
- `show mbufs`: show mbuf statistics, if the network stack is involved.
- `show malloc`: show kernel malloc statistics, grouped by type. Useful for finding a driver that leaked memory before the panic.
- `show registers`: show the current CPU's registers.

Navigation commands:

- `x/i <addr>`: disassemble at an address. Useful for looking at the instruction where a page fault happened.
- `x/xu <addr>`: dump memory as hex bytes.
- `x/sz <addr>`: dump memory as a null-terminated string.

Control commands:

- `continue`: resume execution. If the kernel was stable enough to enter `ddb`, this can sometimes let the machine continue normally. Often the safer choice is `panic`.
- `reset`: reboot the machine immediately.
- `panic`: force a panic, which triggers a crash dump if `dumpdev` is set.
- `call <func>`: call a kernel function by name. Advanced; rarely useful in driver debugging.

Thread-walking commands:

- `show all procs`: list all processes.
- `show sleepq`: show sleeping queues.
- `show turnstile`: show turnstile state (used for blocking locks).
- `show sema`: show semaphores.

A driver author does not memorise this list. The pattern is: when you enter `ddb`, run `bt` first, then `ps`, then `show alllocks`, then `show mbufs`, then `show malloc`. Those five commands capture 80% of what most driver panics need.

A useful habit is to print the commands and keep them next to the keyboard for the first few ddb sessions. The first session feels awkward because the muscle memory is not there. By the third session, the commands are fluent.

23.5.8 A Brief Introduction to kgdb

ddb is the live debugger. kgdb is the post-mortem debugger. When a kernel panics with dumpdev configured, the panic routine writes the full kernel memory image to the dump device. On the next boot, savecore(8) copies the dump from the dump device into /var/crash. A reader can then open it with kgdb:

```
sudo kgdb /boot/kernel/kernel.debug /var/crash/vmcore.0
```

(Note: kernel.debug is the debug-symbol version of the kernel, produced when the kernel was built with makeoptions DEBUG=-g. Without DEBUG=-g, kgdb can still open the dump but produces less useful output.)

Inside kgdb, the reader has something close to a normal gdb interface. The commands are the same: bt, frame N, info locals, print variable, list, and so on. The difference is that the machine is not running; the kernel is paused forever at the moment of the panic, and every inspection is against that frozen state. You can walk up the stack, look at data structures, follow pointers, and reason about what led to the panic. You cannot step forward.

A typical kgdb session on a panic looks like this:

```
(kgdb) bt
#0  doadump (...)
#1  kern_reboot (...)
#2  vpanic (...)
#3  panic (...)
#4  witness_checkorder (...)
...

(kgdb) frame 10
#10 myfirst_pci_detach (dev=0xfffff80001a23000) at myfirst_pci.c:789
789          mtx_destroy(&sc->lock);

(kgdb) list
784          mtx_lock(&sc->lock);
785          sc->detaching = true;
786          /* wait for all users */
787          while (sc->refs > 0)
788              cv_wait(&sc->cv, &sc->lock);
789          mtx_destroy(&sc->lock);
790          return (0);
791      }
```

```
(kgdb) print sc->refs
$1 = 0

(kgdb) print sc->lock
$2 = {
    ...
}
```

The reader walks the stack to find the driver frame, looks at what the driver was doing, and identifies the bug. In this example, the driver was destroying a mutex that was still locked, because line 789 calls `mtx_destroy` without unlocking. The bug is immediately visible.

Chapter 23 does not make heavy use of `kgdb`, because the labs are designed to be diagnosable from `ddb` and `dmesg` alone. Section 7 mentions `kgdb` in the common-bug walkthrough for a class of bugs where post-mortem inspection is the right tool. The reader who wants more should read the `kgdb(1)` man page and experiment on a known panic.

23.5.9 Exercise: Build a Debug Kernel and Confirm Symbols Are Present With `kgdb`

A short exercise that grounds the section.

1. If not already done, build and install a MYDEBUG kernel as described above, with `INVARIANTS`, `WITNESS`, `WITNESS_SKIPSPIN`, and `makeoptions DEBUG=-g`. Reboot.
2. Confirm the running kernel is the debug build:

```
uname -v | grep MYDEBUG
sysctl debug.witness.watch
```

3. Configure a dump device. This is usually a swap partition:

```
sudo dumpon /dev/adaNpY
```

where `adaNpY` is your swap partition. On a typical VM this is `ada0p3`.

4. Trigger a deliberate panic to produce a crash dump. The safest way is to run `sysctl debug.kdb.panic=1` as root:

```
sudo sysctl debug.kdb.panic=1
```

The system enters `ddb` (because `KDB_UNATTENDED` was not set for this exercise; if it was set, the system writes a dump and reboots). Run `bt`, `ps`, and `show alllocks`, then type `panic` to let the dump proceed. The system writes to the dump device and reboots.

5. After reboot, verify that `savecore(8)` has copied the dump:

```
ls /var/crash
```

You should see `vmcore.0`, `info.0`, and `kernel.0`.

6. Open the dump with `kgdb`:

```
sudo kgdb /var/crash/kernel.0 /var/crash/vmcore.0
```

7. Inside `kgdb`, run `bt`. You should see the panic stack. Walk up a few frames with `frame N`. Confirm that source lines and function arguments are visible; this is what the debug symbols buy.
8. Quit `kgdb` with `quit`.

The exercise is not about fixing a bug; it is about proving that the debug infrastructure works. From this point on, any real panic the reader causes is fully debuggable: the crash is preserved, the symbols are present, the debugger opens the dump. That is the foundation. Without it, debugging is mostly guessing.

23.5.10 Wrapping Up Section 4

Section 4 turned the reader’s kernel into a debug kernel. The options enable assertions, lock-order checking, DTrace hooks, debug symbols, and the in-kernel debugger. The cost is a slower system; the benefit is a system that catches its own bugs loudly. `ddb` is the live inspection tool; `kgdb` is the post-mortem inspection tool. Both work against the same debug kernel the reader is now running.

With the logging pipeline (Section 3) and the debug kernel (Section 4) in place, the reader has the infrastructure that every later tool assumes. Section 5 introduces DTrace, the tool that turns the debug kernel into a live tracing and measurement platform.

23.6 Section 5: Using DTrace for Live Kernel Inspection

DTrace is the scalpel of the FreeBSD debugging toolkit. Where `printf` is a blunt instrument and `ddb` is a hammer, DTrace is a scalpel: it lets you ask specific, sharp questions about the kernel’s live behavior and get answers without modifying the source, rebuilding the kernel, or stopping the system. A driver author who learns DTrace reaches a new kind of productivity: the ability to say “I wonder how often the driver’s interrupt fires” and get an answer in ten seconds without changing a line of code.

This section introduces DTrace at the level of detail a driver author needs. It does not try to be a DTrace reference; the excellent `dttrace(1)` man page, the DTrace Guide available online, and the book *Illumos DTrace Toolkit* are the deeper references. What Section 5 gives the reader is enough DTrace to instrument the `myfirst` driver, measure its behavior, and understand what DTrace can and cannot do.

23.6.1 What Is DTrace?

DTrace is a **dynamic tracing framework**. The word “dynamic” is important: it does not require the kernel to have been built with knowledge of every trace point; it does not require the running code to be patched; it does not require the reader to rebuild and reboot. You load the DTrace module, ask for a probe, and the framework attaches instrumentation to the running kernel code. When the instrumented code runs, the instrumentation fires, and DTrace records the event. When you are done, you detach, and the instrumentation is removed.

The framework originated in Solaris and was ported to FreeBSD. It lives under `/usr/src/cddl/` (the CDDL-licensed portion of the source tree), because it inherits the Solaris licensing. The user-space tool is `dttrace(1)`; the kernel-side infrastructure is a set of modules that can be loaded and unloaded on demand.

DTrace has three organising concepts: **providers**, **probes**, and **actions**.

A **provider** is a source of trace points. Examples of providers FreeBSD ships with: `fbt` (function boundary tracing, one probe per entry and exit of every kernel function), `syscall` (one probe per system call entry and return), `sched` (scheduler events), `io` (block I/O events), `vfs` (filesystem events), `proc` (process events), `vm` (virtual memory events), `sdt` (statically defined tracing, for probes compiled into the kernel), and `lockstat` (lock operations). Some providers (`fbt`, `syscall`) are always available once the module is loaded; others (`sdt`) only exist if the kernel code explicitly defines probes.

A **probe** is a specific point the reader can trace. Probes have four-part names in the form `provider:module:function:name`. For example, `fbt:kernel:myfirst_pci_attach:entry` is the `fbt` provider's entry probe on the function `myfirst_pci_attach` in the kernel module `kernel` (the main kernel binary, not a loadable module). A wildcard `*` matches any segment: `fbt::myfirst_*:entry` matches every function whose name starts with `myfirst_`, in any module, at function entry.

An **action** is what DTrace does when a probe fires. The simplest action is `{ trace(...) }`, which records the argument. More interesting actions aggregate: `@counts[probefunc] = count()` counts how many times each function's probe fired. The D language (DTrace's scripting language) is C-like but safer: it has no loops, no function calls, no memory allocation, and runs in the kernel under strict safety constraints.

The power of DTrace comes from the combination: you enable a specific probe, you attach a specific action, you observe only the events you care about, and you do it without touching the source.

23.6.2 Enabling DTrace Support

The kernel options needed for DTrace were covered in Section 4: `options KDTRACE_HOOKS`, `options KDTRACE_FRAME`, `options DDB_CTF`, and `makeoptions WITH_CTF=1`. If the reader built the `MYDEBUG` kernel with those options, DTrace is ready to go.

The user-space side needs the DTrace module to be loaded. The umbrella module is `dtraceall`, which pulls in all the providers:

```
sudo kldload dtraceall
```

Alternatively, individual providers can be loaded:

```
sudo kldload dtrace           # core framework
sudo kldload dtraceall        # all providers (recommended for development)
sudo kldload fbt              # just function boundary tracing
sudo kldload systrace         # just syscall tracing
```

For development, `dtraceall` is the simplest choice: all the providers the reader might want are available.

To confirm DTrace is working:

```
sudo dtrace -l | head -20
```

This lists the first twenty probes DTrace knows about. On a default FreeBSD 14 kernel with `dtraceall` loaded, the output includes hundreds of thousands of probes, most from the `fbt` provider (every kernel function has an entry and return probe). A quick count:

```
sudo dtrace -l | wc -l
```

A typical number is 40,000 to 80,000 probes, depending on what modules are loaded.

To list only the probes in the `myfirst` driver's functions:

```
sudo dtrace -l -n 'fbt::myfirst_*:'
```

If the driver is loaded, this produces a list of every `myfirst_*` function's entry and return probes. If the driver is not loaded, the list is empty: `fbt` probes only exist for code the kernel has loaded.

23.6.3 Writing Simple DTrace Scripts

The simplest DTrace invocation traces a single probe and prints when it fires:

```
sudo dtrace -n 'fbt::myfirst_pci_attach:entry'
```

The syntax is `provider:module:function:name`. The double colon in `fbt::myfirst_pci_attach:entry` uses the default empty module (which matches any module). DTrace prints one line per probe firing:

CPU	ID	FUNCTION:NAME
2	28791	myfirst_pci_attach:entry

This confirms the function was called, on CPU 2, with the internal probe ID 28791. For a one-shot “did this function run?” question, that is already enough.

A slightly more useful script traces the entry and return:

```
sudo dtrace -n 'fbt::myfirst_pci_attach:entry,fbt::myfirst_pci_attach:return'
```

Each line shows when entry and return fired. The two together tell the reader that the function ran and returned without panicking.

To trace many functions at once, use a wildcard:

```
sudo dtrace -n 'fbt::myfirst_*:entry'
```

Every `myfirst_*` function's entry produces a line. For the reader debugging an attach path, this is a cheap way to see the call sequence in real time.

To capture arguments, add a D-language action:

```
sudo dtrace -n 'fbt::myfirst_pci_suspend:entry { printf("%d: suspending %s", timestamp,  
↪ stringof(args[0]->name)); }'
```

The `args[0]` refers to the first argument of the probed function. For FBT probes, `args[0]` is the first argument to the function. For `myfirst_pci_suspend`, the first argument is a `device_t`, and `args[0]->name` (if FreeBSD's `device_t` layout has a `name` field accessible to DTrace) shows the device name.

Note: `device_t` is actually a pointer to an opaque structure, and DTrace can only follow type-aware pointer chains if the CTF data is loaded. This is why options `DDB_CTF` and `makeoptions WITH_CTF=1` matter: without them, DTrace sees `args[0]` as an integer and cannot dereference it meaningfully. With them, DTrace knows `args[0]` is a `device_t` (which is a `struct _device *`) and can walk the structure.

A more interesting script counts how many times each `myfirst_*` function fires:

```
sudo dtrace -n 'fbt::myfirst_*.entry { @counts[probfunc] = count(); }'
```

Let this run for a minute while the driver is exercised. Press Ctrl-C. DTrace prints a histogram:

<code>myfirst_pci_attach</code>	1
<code>myfirst_pci_detach</code>	1
<code>myfirst_pci_suspend</code>	42
<code>myfirst_pci_resume</code>	42
<code>myfirst_intr_filter</code>	10342
<code>myfirst_rx_task</code>	5120
<code>myfirst_dma_submit</code>	1024

The reader sees, at a glance, the driver's call distribution: two attach/detach, forty-two suspend/resume cycles, ten thousand interrupt filters, five thousand rx tasks, one thousand DMA submits. Any unexpected number is a signal to investigate. If the reader expected one-to-one between interrupts and tasks, and sees two-to-one, there is a bug.

23.6.4 Measuring Function Latency

One of DTrace's most useful idioms is measuring how long a function takes. The pattern is to timestamp entry, timestamp return, and subtract:

```
sudo dtrace -n '  
fbt::myfirst_pci_suspend:entry { self->start = timestamp; }  
fbt::myfirst_pci_suspend:return /self->start/ {  
    @times = quantize(timestamp - self->start);  
    self->start = 0;  
}'
```

The first clause stores the entry timestamp in a thread-local variable `self->start`. The second clause, predicated on `self->start` being non-zero, computes the delta, adds it to a quantize aggregation (a logarithmic histogram), and clears the variable. Let this run while the reader exercises suspend cycles, press Ctrl-C, and DTrace prints the histogram:

value	----- Distribution -----	count
1024		0
2048	@@@@	4
4096	@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@	28
8192	@@@@@@@@@@	8
16384	@@	2
32768		0

The numbers are in nanoseconds. This says: most suspend calls took between 4,096 and 8,192 nanoseconds (4 to 8 microseconds), with a tail reaching 16,384 nanoseconds (16 microseconds). For a suspend call that should be fast, this is the kind of measurement that tells the reader whether the driver's discipline is working.

For a different kind of insight, use `avg( )` or `max( )` aggregations:

```
sudo dtrace -n '
fbt::myfirst_pci_suspend:entry { self->start = timestamp; }
fbt::myfirst_pci_suspend:return /self->start/ {
    @avg = avg(timestamp - self->start);
    @max = max(timestamp - self->start);
    self->start = 0;
}'
```

This prints the average and maximum suspend time at the end. Useful for summary reports.

23.6.5 Using the syscall Provider

The `syscall` provider shows every system call. This is especially useful for driver work, because user-space calls to the driver (`open`, `read`, `write`, `ioctl`) go through the `syscall` layer first. A script that watches for system calls on a specific file descriptor is a lightweight way to see how user programs use the driver:

```
sudo dtrace -n 'syscall::ioctl:entry /execname == "myftest"/ { printf("%s ioctl on fd
↵ %d", execname, arg0); }'
```

This prints every `ioctl` system call made by a process named `myftest`. The `execname` variable is a built-in DTrace variable that holds the current process's executable name. Running the test program `myftest` (a small user program that exercises the driver) while this script runs gives a live view of every `ioctl` the program issues.

23.6.6 Using the sched Provider

The sched provider tracks scheduler events: thread switches, wake-ups, enqueues. It is useful for understanding concurrency in the kernel, including driver interrupt handling:

```
sudo dtrace -n 'sched:::on-cpu /execname == "myftest"/ { @[cpu] = count(); }'
```

This counts, per CPU, how many times the myftest program was scheduled onto each CPU during the trace. Useful for understanding whether the scheduler is keeping the program on one core or bouncing it across cores, which affects the driver's cache behavior.

23.6.7 Using the io Provider

The io provider shows block I/O events. Not directly relevant to myfirst (a generic PCI driver), but useful to know about. A script to show every disk I/O:

```
sudo dtrace -n 'io:::start { printf("%s %d", execname, args[0]->bio_bcount); }'
```

For storage drivers, io is essential. For myfirst it is peripheral. Chapter 23 does not explore it in depth.

23.6.8 Statically Defined Tracing (SDT)

The fbt provider instruments every function entry and return. That is flexible but blunt: you get entry/return probes for every function, and you often want probes at specific interesting points inside a function.

SDT (Statically Defined Tracing) solves this. SDT lets the driver author add named probes to specific places in the source, and DTrace makes them visible. When DTrace is not watching, the probes are essentially free: they compile to a single no-op instruction. When DTrace attaches, the no-op is replaced with a trap that fires the probe.

The SDT machinery lives in `/usr/src/sys/sdt.h`. The basic macros are:

- `SDT_PROVIDER_DEFINE(name)`: declares a provider. Usually done once per driver, in the main source file.
- `SDT_PROBE_DEFINE(provider, module, function, name, "arg1-type", "arg2-type", ...)`: declares a probe with N typed arguments.
- `SDT_PROBE(provider, module, function, name, arg1, arg2, ...)`: fires the probe with the given arguments.

A minimal example for the myfirst driver. In `myfirst_pci.c`, near the top:

```
#include <sys/sdt.h>

SDT_PROVIDER_DEFINE(myfirst);
SDT_PROBE_DEFINE2(myfirst, , attach, entry,
```

```

    "struct myfirst_softc *", "device_t");
SDT_PROBE_DEFINE2(myfirst, , attach, return,
    "struct myfirst_softc *", "int");
SDT_PROBE_DEFINE3(myfirst, , dma, submit,
    "struct myfirst_softc *", "bus_addr_t", "size_t");
SDT_PROBE_DEFINE2(myfirst, , dma, complete,
    "struct myfirst_softc *", "int");

```

Then in the driver code:

```

int
myfirst_pci_attach(device_t dev)
{
    struct myfirst_softc *sc = device_get_softc(dev);
    int error;

    SDT_PROBE2(myfirst, , attach, entry, sc, dev);
    ...
    error = ...; /* real attach code */
    ...
    SDT_PROBE2(myfirst, , attach, return, sc, error);
    return (error);
}

```

From user space, the reader can now observe myfirst probes through DTrace:

```
sudo dtrace -n 'myfirst:::' | head
```

This shows every myfirst probe. To watch only DMA events:

```

sudo dtrace -n 'myfirst:::dma-submit'
sudo dtrace -n 'myfirst:::dma-complete'

```

To count submissions per DMA size:

```
sudo dtrace -n 'myfirst:::dma-submit { @[args[2]] = count(); }'
```

This prints a histogram of DMA transfer sizes at the end of the trace. For a driver debugging session, that is the kind of data that turns speculation into evidence.

Chapter 23's Stage 2 adds exactly these probes to the myfirst driver. The exercise at the end of Section 5 walks through the addition. The corresponding files live in `examples/part-05/ch23-debug/stage2-sdt/`.

Real drivers that use SDT are worth studying. `/usr/src/sys/dev/virtio/virtqueue.c` defines:

```
SDT_PROVIDER_DEFINE(virtqueue);
SDT_PROBE_DEFINE6(virtqueue, , enqueue_segments, entry, "struct virtqueue *",
    ...);
SDT_PROBE_DEFINE1(virtqueue, , enqueue_segments, return, "uint16_t");
```

and fires them in `virtqueue_enqueue`:

```
SDT_PROBE6(virtqueue, , enqueue_segments, entry, vq, desc, head_idx, ...);
...
SDT_PROBE1(virtqueue, , enqueue_segments, return, idx);
```

The reader can run `sudo dtrace -l -n 'virtqueue:::'` on a system with virtio-backed devices (common in VMs) and see these probes immediately. Tracing them gives a live view of virtqueue activity that would be impossible to reproduce with `printf`.

23.6.9 Useful Providers for Driver Work

A summary of the providers a driver author uses most:

- **fbt**: function boundary tracing. Every kernel function has entry and return probes. Best for “did this function run?” and “what arguments did it see?”.
- **sdt**: statically defined tracing. Probes the driver explicitly adds. Best for observing driver-specific events at specific points.
- **syscall**: system call tracing. Best for seeing what user programs are asking the kernel to do.
- **sched**: scheduler events. Best for understanding concurrency and CPU usage.
- **io**: block I/O events. Best for storage work.
- **vfs**: filesystem events. Best for filesystem driver work.
- **proc**: process lifecycle events. Best for process creation, exit, and signal delivery.
- **lockstat**: lock operations. Best for lock contention analysis; expensive when active.
- **priv**: privilege-related events. Useful for security auditing.

Most driver work uses `fbt`, `sdt`, and `syscall` heavily, and touches the others occasionally.

23.6.10 Common DTrace Pitfalls

A few patterns that trip new DTrace users.

Buffer too small. DTrace has per-CPU buffers; if a script fires probes faster than DTrace can drain them, events are dropped. The reader sees a message like `dtrace: X drops on CPU N`. The fix is usually to make the predicate tighter (fire less often), or to increase buffer size with `-b 16m`.

Forgetting to clear thread-local variables. In the latency-measurement pattern, the script sets `self->start` on entry and checks `self->start != 0` on return. If the reader forgets to clear it (`self->start = 0`; after use), the variable accumulates state and later traces get confused. Always pair `set` with `clear`.

Using `trace()` when `printf()` would be clearer. `trace(x)` prints the value in a terse format that is hard to read. `printf("x=%d", x)` is usually better.

Running as a non-privileged user. DTrace requires root. Running `dtrace` as a regular user fails with a permission error. Always use `sudo dtrace`.

Probes that do not exist. If a probe name is mistyped or the module is not loaded, `dtrace` prints an error. The fix is to confirm the probe exists with `dtrace -l` before trying to use it.

The `stringof()` trap. When printing a kernel string pointer, the reader often needs `stringof(ptr)`, not `ptr`, because DTrace does not automatically follow kernel pointers. Forgetting this produces garbage output.

23.6.11 Exercise: Use DTrace to Measure How Long a Driver Read Operation Takes

A hands-on companion for Section 5.

1. Ensure the MYDEBUG kernel is running and `dtraceall` is loaded.
2. Write a user program that reads from `/dev/myfirst0` a hundred times. The program can be as simple as:

```
#include <fcntl.h>
#include <unistd.h>
int main(void) {
    int fd = open("/dev/myfirst0", O_RDWR);
    char buf[4096];
    for (int i = 0; i < 100; i++)
        read(fd, buf, sizeof(buf));
    close(fd);
    return 0;
}
```

Compile with `cc -o myftest myftest.c`.

3. In one terminal, run the DTrace latency script on the driver's read path. Assuming the driver's read function is `myfirst_read`:

```
sudo dtrace -n '
fbt::myfirst_read:entry { self->start = timestamp; }
fbt::myfirst_read:return /self->start/ {
    @times = quantize(timestamp - self->start);
    self->start = 0;
}'
```

4. In another terminal, run the user program:

```
./myftest
```

5. Press Ctrl-C on the DTrace script. Read the histogram. Most reads should be in the low microseconds (a few thousand nanoseconds).

6. Take a screenshot or save the output. Compare to your expectation. If a read is taking longer than you expect, that is the start of a debugging session.

Optional: run the same script against `fbt::myfirst_intr_filter:entry` and observe the interrupt latency distribution.

23.6.12 Wrapping Up Section 5

Section 5 introduced DTrace as the scalpel of the debugging toolkit. DTrace is dynamic: probes attach to running code without rebuilds. It is safe: the D language has no loops or memory allocation. It is broad: providers expose tens of thousands of observation points across the kernel. For driver work, the combination of `fbt` (automatic probes on every function) and `sdt` (custom probes at specific points) gives the reader the ability to answer almost any “what is the driver doing?” question without modifying the source.

The cost of DTrace when inactive is essentially zero: probes exist in the kernel as no-op instructions or function-pointer slots. The cost when active depends on what the script does: a simple count is cheap, a complex histogram is more expensive, and a script that prints on every probe can flood the buffer. Understanding the trade-off is part of writing effective DTrace scripts.

With logging (Section 2), log pipelines (Section 3), debug kernels (Section 4), and DTrace (Section 5) in place, the reader has four complementary views into kernel behavior. Section 6 adds the last essential view for driver work: `kttrace`, the tool that shows what a user program is doing as it crosses into the kernel through the driver’s interface.

23.7 Section 6: Tracing Kernel Activity With `kttrace` and `kdump`

DTrace traces the kernel from the kernel side: which functions ran, how long they took, what state they touched. `kttrace` traces the same activity from the user side: which system calls a process made, what arguments it passed, what return values it got, what signals it received. Together, the two tools give a complete view. DTrace shows what the kernel did; `kttrace` shows what the program asked for.

For driver work, `kttrace` is the tool of choice when the question is “what is the user program doing?”. A driver that gets strange arguments can be traced back: maybe the program is calling `ioctl` with the wrong command number, or `write` with a zero-length buffer, or `open` with the wrong flags. `kttrace` answers those questions by recording every syscall the program issues.

23.7.1 What Is `kttrace`?

`kttrace(1)` is a user-space tool that tells the kernel to record a trace of one or more processes’ activities into a file. The recorded events include:

- Every system call entry and return, with arguments and return value.
- Every signal delivered to the process.
- Every `namei` (path-name lookup) event.
- Every I/O buffer passed to read or write.
- Context switches, at a coarse level.

The trace is written to a binary file (by default `ktrace.out`). To read it, the reader runs `kdump(1)`, which translates the binary trace into human-readable text.

`ktrace` is a simple tool, much older than DTrace, and it shows its age in some interfaces. But for the core question of “what did this process ask the kernel to do?”, it is ideal.

The implementation is in `/usr/src/sys/kern/kern_ktrace.c` and the user-facing header is in `/usr/src/sys/sys/ktrace.h`. The user-space tools are under `/usr/src/usr.bin/ktrace/` and `/usr/src/usr.bin/kdump/`.

The kernel needs options `KTRACE` to support `ktrace`. This option is enabled in `GENERIC` by default on every architecture; the reader does not need to rebuild a kernel to use `ktrace`.

23.7.2 Starting a Trace

The simplest invocation attaches to a running process by PID:

```
sudo ktrace -p 12345
```

This tells the kernel to start recording the process with PID 12345. The kernel writes events to `ktrace.out` in the current directory. When the reader is done, they stop the trace:

```
sudo ktrace -C
```

Or, more precisely, stop tracing for a specific PID:

```
sudo ktrace -c -p 12345
```

To run a command from scratch with tracing enabled:

```
sudo ktrace /path/to/program arg1 arg2
```

This starts the program and traces it from the first syscall. When the program exits, the trace file remains for the reader to examine.

Useful flags:

- `-d`: trace descendants too. If the program forks, the children are also traced.
- `-i`: inherit the trace across `exec`. If the program execs another program, the trace continues. Without `-i`, the trace stops at `exec`.
- `-t <mask>`: select which event types to record. The mask is a set of letter codes, like `cnios` (calls, namei, I/O, signals). Default is most types.
- `-f <file>`: write to a specific file instead of `ktrace.out`.
- `-C`: clear all tracing (stop all traces).

A common combination:

```
sudo ktrace -di -f mytrace.out -p 12345
```

Trace the process, all descendants, through `exec`, into `mytrace.out`.

23.7.3 Reading Output With `kdump`

The binary trace file is useless on its own. `kdump` translates it:

```
kdump -f mytrace.out
```

Output looks like:

```
12345 myftest  CALL  open(0x8004120a0,0x0)
12345 myftest  NAMI  "/dev/myfirst0"
12345 myftest  RET   open 3
12345 myftest  CALL  ioctl(0x3,0x20006601,0x7fffffff8c0)
12345 myftest  RET   ioctl 0
12345 myftest  CALL  read(0x3,0x7fffffff900,0x1000)
12345 myftest  GIO   fd 3 read 4096 bytes
                  "... data bytes ..."
12345 myftest  RET   read 4096/0x1000
```

Each line has a PID, a process name, an event type, and event-specific data:

- `CALL` lines show a syscall entry, with arguments as hex numbers.
- `NAMI` lines show a path-name lookup, with the resolved string.
- `RET` lines show a syscall return, with the value.
- `GIO` lines show generic I/O buffers, followed by a dump of the buffer contents.
- `PSIG` lines show signal delivery.

The output is dense but readable with practice. A common pattern for driver work:

1. Run the user program under `ktrace`.
2. Look for the `open` of `/dev/myfirst0`. Note the file descriptor the kernel returned.
3. Filter subsequent lines to the operations on that fd: `ioctl`, `read`, `write`, `close`.
4. Note the exact arguments, especially for `ioctl` where the command number encodes which operation the driver was asked to perform.

For example, the `ioctl(0x3,0x20006601,0x7fffffff8c0)` line above shows the program issuing an `ioctl` on fd 3 with command `0x20006601`. The command is the `_IO/_IOR/_IOW/_IOWR` encoding; the low 16 bits are the command number and the high bits are the direction and size. A driver author debugging an `ioctl` bug can look at this command, compare it to the definitions in the driver's header, and confirm the program is sending the command the driver expects.

23.7.4 Useful kdump Filters

kdump takes several flags that filter the output:

- **-t <mask>**: filter by type, same mask as `ktrace -t`.
- **-E**: show elapsed time since the previous event.
- **-R**: show relative timestamps instead of absolute.
- **-l**: show the PID and process name on every line.
- **-d**: hex-dump I/O buffers (on by default for GIO lines).
- **-H**: more readable hex dump.

A common refinement:

```
kdump -f mytrace.out -E -t cnri
```

Elapsed time, syscall entry/return plus namei plus I/O.

Piping to `grep` is also useful:

```
kdump -f mytrace.out | grep -E 'myfirst|ioctl|open|read|write|close' | less
```

This shows only lines relevant to the driver.

23.7.5 Finding Driver-Related Syscalls

A user program talks to a driver through four main syscalls: `open`, `ioctl`, `read`, `write`, and `close`. A driver author tracing a user program watches for these:

- **`open("/dev/myfirst0", ...)`**: the program opens the device. The return value is a file descriptor the subsequent syscalls use.
- **`ioctl(fd, cmd, arg)`**: the program issues a device-specific command. The `cmd` encodes the operation; `arg` is usually a pointer to a structure.
- **`read(fd, buf, size)`**: the program reads `size` bytes into `buf`.
- **`write(fd, buf, size)`**: the program writes `size` bytes from `buf`.
- **`close(fd)`**: the program closes the descriptor.

A trace that shows these in order is the reader's view of the user side of the driver's interface. A bug where the program is passing the wrong size to `read` shows up immediately as `read(fd, buf, 0)` (zero size) in the trace. A bug where the program is forgetting to call `close` shows up as the absence of a `close` call before `exit`. A bug where the program is using an undefined `ioctl` command shows up as an unusual hex value in the `cmd` argument.

23.7.6 Comparing ktrace and DTrace

The two tools overlap in some ways and complement each other in others. A rough guide:

Use `ktrace` when:

- You want to see what a user program is doing, with the program's perspective.
- You want a record you can save and review later.
- You want minimal setup (no kernel rebuild, no script writing).
- You want to catch the arguments to syscalls, including pointer-destination strings and I/O buffers.

Use DTrace when:

- You want to see what is happening inside the kernel.
- You want to measure latency, distribution, or contention.
- You want to aggregate data across many events.
- You need to trace without modifying the program or running it under a wrapper.

For most driver work, the reader uses both. `kttrace` answers “what is the program asking for?”. DTrace answers “what is the driver doing about it?”. The combination is the full picture.

23.7.7 A Note on KTR vs. kttrace

The names confuse new driver authors. `kttrace` (lowercase) is the user-space tracing tool described above. `KTR` (uppercase) is the in-kernel event tracing framework, enabled with options `KTR`. They are different.

`KTR` is a high-performance circular buffer for kernel events, intended for kernel developers debugging the kernel itself. Driver authors can add `CTR` (category-0, category-1, etc.) macros to their code and observe them through `ddb`'s `show ktr` command. In practice, `KTR` is used by a few core kernel subsystems (scheduler, locking, some device subsystems) and rarely by individual drivers. Chapter 23 does not pursue `KTR` further; the reader who is curious can read `/usr/src/sys/sys/ktr.h` for the macro definitions and `/usr/src/sys/conf/NOTES` for the configuration options.

23.7.8 Exercise: Trace a Simple User Program Accessing Your Driver

A hands-on companion for Section 6.

1. Write a small user program `myftest.c` that opens `/dev/myfirst0`, issues one `ioctl`, reads 4 KB, writes 4 KB, and closes. The simplest version:

```
#include <sys/ioctl.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>

int
main(void)
{
    int fd, error;
    char buf[4096];

    fd = open("/dev/myfirst0", O_RDWR);
    if (fd < 0) { perror("open"); return 1; }
```

```
error = ioctl(fd, 0x20006601 /* placeholder */, NULL);
printf("ioctl: %d\n", error);

error = read(fd, buf, sizeof(buf));
printf("read: %d\n", error);

error = write(fd, buf, sizeof(buf));
printf("write: %d\n", error);

close(fd);
return 0;
}
```

Compile: `cc -o myftest myftest.c`.

2. Trace the program:

```
sudo ktrace -di -f mytrace.out ./myftest
```

3. Read the trace:

```
kdump -f mytrace.out | less
```

4. Find the lines relating to `/dev/myfirst0`:

```
kdump -f mytrace.out | grep -E 'myfirst0|CALL|RET|NAMI' | less
```

5. Confirm that each syscall is there, with the expected arguments. For the `ioctl`, decode the command number by hand: `0x20006601` means direction `IOC_OUT` (high byte `0x20`), size 0 (next 14 bits), type 'f' (`0x66`), command 01. A driver that defined `MYFIRST_IOC_RESET = _IO('f', 1)` would see exactly this command.

6. (Optional) Run the program under DTrace in a second terminal, counting the driver functions:

```
sudo dtrace -n 'fbt::myfirst_*:entry { @[probefunc] = count(); }' &
./myftest
```

Compare the DTrace count to the `ktrace` sequence. The reader should see that each user syscall triggered specific driver functions: `open` → `myfirst_open`, `ioctl` → `myfirst_ioctl`, `read` → `myfirst_read`, and so on.

The exercise grounds the tools in a specific, concrete trace. Every driver author has this pattern in their toolkit for the rest of their career: user-side trace with `ktrace`, kernel-side trace with DTrace, correlated by time and PID.

23.7.9 Wrapping Up Section 6

Section 6 introduced `ktrace` and `kdump`, the user-side tracing pair that complements `DTrace`. `ktrace` records the syscalls a process makes; `kdump` translates the record into text; the combination shows the driver's interface from the user program's perspective. For a driver author, this view is essential for debugging bugs that originate on the user side of the driver boundary.

With logging, log analysis, debug kernels, `DTrace`, and `ktrace` in hand, the reader has the five tools that handle most driver-debugging questions. Section 7 turns to the classes of bugs these tools most often find, with the characteristic symptom of each and the tool that best diagnoses it.

23.8 Section 7: Diagnosing Common Driver Bugs

The previous six sections built a toolbox: logging, log inspection, debug kernels, `DTrace`, and `ktrace`. Section 7 uses that toolbox on the bugs readers will actually meet. Every experienced FreeBSD driver author has hit each of these at least once. The purpose of this section is to describe the symptoms, explain the underlying cause, and match each bug class to the tool that diagnoses it most efficiently.

None of the bugs below are exotic. They arise from ordinary coding mistakes, incomplete understanding of the kernel environment, or assumptions that hold on one platform but not another. The reader who learns to recognise the symptoms will save hours of frustration, because the first step toward a fix is always correct classification.

23.8.1 7.1 Memory Leaks: the Quiet Growth

A memory leak in a kernel module is harder to detect than one in a user program. There is no `valgrind` equivalent that runs transparently, and the kernel does not exit when the module unloads, so a leak continues to consume memory even after the driver is gone. For long-running systems, a small leak is fatal: the pool it allocates from grows indefinitely, other subsystems start failing, and eventually the machine runs out of kernel memory and panics.

The characteristic symptom is simple: a `malloc` pool that grows but never shrinks. The tool that exposes the symptom is `vmstat -m`:

```
vmstat -m | head -1
vmstat -m | grep -E 'Type|myfirst'
```

The output looks like:

Type	InUse	MemUse	Requests	Size(s)
myfirst	12	3K	48	256

The three columns that matter are `InUse`, `MemUse`, and `Requests`. `InUse` is the number of allocations from this pool that are currently outstanding. `MemUse` is the total memory those allocations occupy, in kilobytes.

`Requests` is the total number of allocations made since the pool was created, including those that were later freed.

A healthy pool has `InUse` that stays roughly constant under a steady workload. A leaking pool has `InUse` that grows without bound.

To confirm a leak, run the workload that exercises the driver, then take two snapshots of `vmstat -m` several minutes apart:

```
vmstat -m | grep myfirst
# run the workload for 5 minutes
vmstat -m | grep myfirst
```

If the first shows `InUse=12` and the second shows `InUse=4800`, and the workload was not expected to add four thousand allocations, the driver is leaking.

The second step is to identify the offending path. Every `malloc(9)` call pairs with an expected `free(9)`. The leak is in whichever path allocates without freeing. Search the driver for `malloc(..., M_MYFIRST, ...)` and for `free(..., M_MYFIRST)`, and verify that every allocation has a matching free on every exit path.

The most common leak patterns in driver code are:

1. **Error-path leaks.** An allocation succeeds, a later step fails, and the error return skips the `free`. The fix is a single cleanup path with `goto fail;` labels, of the kind explored in Chapter 18.
2. **Conditional leaks.** Memory is freed only when a flag is set, and sometimes the flag is not set. The fix is to free unconditionally or to track ownership more carefully.
3. **Forgotten context-destructor.** An object is allocated per open file descriptor and stored on `si_drv1` or similar, but the `d_close` handler does not free it. The fix is to treat `d_close` (or the `cdev-destroy` callback) as the symmetric counterpart to `d_open`.
4. **Timer or task-queue leak.** A task is scheduled, its cleanup routine allocates memory for deferred processing, but the task is cancelled before it runs and the allocated buffer is never freed.

Once the path is identified, add the missing `free`, reload the module, rerun the workload, and confirm with `vmstat -m` that `InUse` now stabilises. A small bit of added logging is often useful: a `device_printf` in the allocation path and another in the free path quickly exposes the ratio of allocations to frees in `dmesg`. For a driver using a `DPRINTF` macro, a dedicated `MYF_DBG_MEM` class makes memory tracing optional and configurable at runtime, which is exactly what Section 8 will implement for the `myfirst` driver.

DTrace can also help here, especially in kernels with `KDTRACE_HOOKS`. A simple one-liner counts the driver's `malloc` and `free` calls and shows any imbalance:

```
dtrace -n 'fbt::malloc:entry /execname == "kernel"/ { @["malloc"] = count(); }
         fbt::free:entry   /execname == "kernel"/ { @["free"]   = count(); }'
```

A much more driver-specific approach is to add DTrace probes to the driver itself, one at each allocation and one at each free, and let DTrace count the ratio. This is the technique Section 8 demonstrates for the `myfirst` driver.

23.8.2 7.2 Race Conditions: the Rare, Destructive Bug

Race conditions are the hardest driver bugs to reproduce and the easiest to miss. A race happens when two threads access the same state without correct synchronisation, and the resulting behaviour depends on the relative timing of the two threads. Under light load the race may never fire; under heavy load it fires hundreds of times per second.

The symptoms of race conditions vary widely: - sporadic panics at unpredictable places - data corruption (values that should never exist) - occasional lock-related assertion failures such as “mutex myfirst_lock not owned” - impossible-looking stacks where two threads appear to be inside the same mutex-protected region

No single symptom proves a race, but any pattern of bugs that appears only at high load should raise suspicion.

The single most effective FreeBSD tool for finding races is `WITNESS`, as introduced in Section 4. A kernel built with options `WITNESS` examines every lock operation and panics on any violation of the declared lock order, on any attempt to acquire a spin lock while holding a sleep lock, on any call to a sleeping function while holding a spin lock, and on many related mistakes.

When `WITNESS` panics or prints a violation, it produces a stack trace showing: - which lock was being acquired - which locks were already held - the declared order that the acquisition would violate - both stacks (the current lock and the conflicting one)

The fix is usually one of:

1. **Reorder the acquisitions.** If code takes lock A then lock B while another path takes B then A, one of them must change.
2. **Add a missing lock.** If state is accessed without any mutex, add the appropriate `MTX_DEF` mutex around the access.
3. **Remove a redundant lock.** Sometimes two locks protect overlapping state, and one can be eliminated.
4. **Switch lock types.** Spin locks and sleep locks have different rules. A region that needs to sleep must use a sleep lock. A region called from interrupt context must often use a spin lock.

A driver that hits no `WITNESS` violations is not free of races, because `WITNESS` catches only lock-ordering issues. Races over state that is accessed without any lock at all require a different approach: careful reading of the code, `DTrace` probes at the suspect points to confirm timing, and occasionally `mtx_assert(&sc->sc_mtx, MA_OWNED)` assertions sprinkled through the critical sections. `INVARIANTS` enables these assertions, which is one more reason to run a debug kernel during development.

Between `WITNESS`, `INVARIANTS`, `mtx_assert`, and `DTrace` counters on the paths where contention is suspected, most races can be narrowed down within an hour or two. Races that survive this arsenal are rare and almost always involve lock-free data structures, atomic operations, or memory-ordering assumptions that require careful review with a senior engineer.

23.8.3 7.3 bus_space and bus_dma Misuse

A new driver author often runs into bus-access bugs that share a distinct family of symptoms:

- reads return `0xFF`, `0xFFFFFFFF`, or otherwise suspicious constants
- writes appear to succeed but the device does not react
- the machine works for seconds or minutes and then hangs or panics

- behaviour is correct on one architecture (amd64) and wrong on another (arm64, riscv64)

Each of these points to a `bus_space` or `bus_dma` misuse.

The first mistake is skipping the handle. `bus_space_read_4()` takes a `bus_space_tag_t` and a `bus_space_handle_t`, obtained from `rman_get_bustag()` and `rman_get_bushandle()` on the resource allocated during attach. Using the raw physical address of the register, or a pointer obtained by a direct cast of the resource, bypasses the platform's bus abstraction. On amd64 the program may appear to work (the kernel has the MMIO region mapped in a way that tolerates this), but on arm64 the register access fails.

The fix is mechanical: always go through `bus_space_read_N()` / `bus_space_write_N()` or the newer `bus_read_N()` / `bus_write_N()` resource-based helpers, never dereference raw pointers for device memory.

The second mistake is incorrect size. `bus_space_read_4` reads a 32-bit value. If the register is 16-bit and the code reads 4 bytes, it reads into an adjacent register as well, and the adjacent register's value now appears at bits 16 through 31 of the returned value. Worse, some hardware does not tolerate a mismatched size and responds with an error. The fix is to use the correct `read_1`, `read_2`, `read_4`, or `read_8` variant for each register's width, as documented in the device's datasheet.

The third mistake is incorrect offset. The `bus_space` handle refers to the base of the mapped region; the offset is added to compute the register address. A typo in the offset reads a different register. For example, reading offset `0x18` instead of `0x10` yields an unexpected value, and the driver's subsequent logic is based on a false reading. The fix is to define every offset as a named constant in a header file, and refer to the constant rather than the number: `#define MYFIRST_REG_STATUS 0x10`, `#define MYFIRST_REG_CONFIG 0x14`, and so on.

For DMA, the most common misuse is freeing or reusing a buffer while the device is still reading from or writing to it. The characteristic symptom is intermittent data corruption, sometimes only under high load. The cause is missing `bus_dmamap_sync` calls, a freed `bus_dmamem` buffer while the descriptor is still enqueued, or an incorrect direction in `bus_dmamap_sync`.

The diagnostic approach is to log every DMA map, sync, and unmap operation, then cross-check the log against the descriptor ring's state. A DTrace one-liner on the driver's DMA paths is usually enough to spot the mistake:

```
dtrace -n 'fbt::myfirst_dma_sync:entry { printf("%s dir=%d", probefunc, arg1); }'
```

For a full treatment of DMA rules and pitfalls, Chapter 21 is the reference. The debug chapter's role is to help the reader recognise the symptoms and narrow the cause.

23.8.4 7.4 Use-After-Detach

A subtler class of bug arises when the driver's detach path returns, the driver's softc is freed, but some other part of the kernel still holds a reference to the freed memory. Examples include:

- an interrupt arriving after `bus_release_resource` but before the handler is torn down
- a callout that fires after detach, dereferencing a freed softc
- a DTrace probe in the driver that fires from a task still running at unload time
- a character-device node held open by a user process, receiving I/O while the driver is unloading

The symptoms are nearly always fatal: page faults in driver code reached after detach, kernel panics with corrupted stacks, or spurious data in the kernel message buffer just before the panic.

The fix has several components, each of which the detach path must implement in a specific order:

1. First, prevent new entries: set a flag in the softc (`sc->sc_detaching = 1`), or take a write lock that all entry points check, so new calls see the driver as going away.
2. Wait for in-progress callers to finish. `bus_tearardown_intr` drains the interrupt handler. `callout_drain` waits for a pending callout to complete. `taskqueue_drain` drains any deferred tasks. `destroy_dev_sched_cb` waits for open file descriptors to close.
3. Only after every external caller is drained, release the resources the driver allocated in attach: free memory, release IRQs, release memory resources, destroy mutexes.

The principle is simple: detach is the mirror image of attach, and every resource allocated in attach must be released in detach in the reverse order. Violations of this principle produce use-after-detach bugs.

The debug tool of choice is the debug kernel itself. A `DEBUG_MEMGUARD`-enabled kernel can be configured to poison freed memory, so an access to a freed softc produces an immediate page fault with a clear stack, instead of a subtle corruption that may take hours to surface.

23.8.5 7.5 Interrupt Handler Mistakes

Driver authors new to the kernel sometimes treat interrupt handlers as ordinary functions and make common but harmful mistakes:

- calling a sleeping function (`malloc(..., M_WAITOK, ...)`, `tsleep`, `mtx_lock` of a sleep lock, `copyin`, `uiomove`) from a filter-level interrupt handler
- attempting to hold a sleep lock for an extended period
- reading or writing a large data block inside the interrupt handler instead of a taskqueue
- not ack'ing the interrupt, so the handler fires forever in a loop

Each of these has a distinct symptom. Sleeping in interrupt context produces a `WITNESS` violation or, in release kernels, a silent corruption of the scheduler state. Holding a sleep lock too long causes other threads to spin or sleep, and latency balloons. Doing heavy work in the handler blocks subsequent interrupts and degrades the whole system's responsiveness. Not ack'ing the interrupt pins a CPU at 100% on interrupt handling forever.

The cure is always the same: make the interrupt handler short and atomic. It should: 1. Read the status register to determine the cause. 2. Acknowledge the interrupt by writing to the status register. 3. Hand off any substantial work to a taskqueue or `ithread`, as explored in Chapter 19. 4. Return.

Complex processing, memory allocation, and long-running operations all belong in the deferred path, not the handler itself.

DTrace is particularly useful for diagnosing interrupt-handler performance. The `intr` provider, or an `fbt` probe on the handler entry, can measure how long each invocation takes:

```
dtrace -n 'fbt::myfirst_intr:entry { self->t = timestamp; }
          fbt::myfirst_intr:return /self->t/ {
            @ = quantize(timestamp - self->t); self->t = 0; }'
```

A healthy handler returns in a few microseconds. If the quantization shows invocations in the hundreds of microseconds or milliseconds, the handler is doing too much work and should be refactored.

23.8.6 7.6 Lifecycle Sequencing Errors

The driver's lifecycle moves through probe, attach, open, close, detach, and unload. Each method has rules about what it may do and what must have happened before. Breaking the rules produces characteristic bugs:

- calling `bus_alloc_resource_any` during probe (it must happen in attach) yields partial allocations and confusion in the probe-ordering logic
- doing substantial work in probe slows boot significantly, because the same probe runs for every candidate device
- creating the `cdev` before the hardware is initialised lets user processes open the device and receive I/O on uninitialised state
- destroying the `cdev` in detach while other threads still hold it open corrupts the devfs state
- doing significant work in module unload instead of detach leaves resources allocated per attach unreleased

The cure is to keep each lifecycle method focused on its job:

- **probe** only identifies the device and returns `BUS_PROBE_DEFAULT` or an error; no allocations, no registration
- **attach** allocates resources, initialises the softc, sets up interrupts, creates the `cdev`, registers the device
- **detach** reverses attach exactly, in reverse order
- **open** / **close** manage per-file state without touching device-wide resources
- **unload** performs only module-level cleanup; per-device cleanup belongs in detach

When a driver follows this structure, lifecycle bugs are rare. When it deviates, they are common and painful.

23.8.7 7.7 Copy-To/From-User Mistakes

The `copyin(9)`, `copyout(9)`, and related `fueword` / `suword` functions transfer data across the user/kernel boundary. They are protected by the virtual memory system: if the user-space address is invalid, the copy returns `EFAULT` instead of panicking. However, this protection only applies if the copy is done through these functions. A driver that dereferences a user-space pointer directly will panic the moment the user pointer is bad, which in real-world deployments is most of the time.

The symptom is a panic with a user-space address in the faulting instruction, stack pointing at the driver's read/write/ioctl path.

The fix is mandatory: every time user data crosses the boundary, use `copyin/copyout` or a `uiomove` through a `struct uio`. Never cast a user pointer to a kernel pointer and dereference it. This rule is absolute.

For `ioctl` specifically, the `d_ioctl` handler receives a kernel pointer, because the kernel has already copied the fixed-size argument from user space. But if the `ioctl`'s argument is a structure that contains a pointer to a larger user buffer, that embedded pointer is still user-space, and `copyin` is required to safely access the buffer it points to.

23.8.8 7.8 Module-Level Bugs

Some bugs affect the whole module rather than a single device:

- module fails to load: the loader prints an error in `dmesg`; read it and fix the symptom (missing dependency, symbol resolution failure, version mismatch)
- module fails to unload: usually means a device is still attached. Detach all devices (`devctl detach myfirst0`) before `kldunload`
- module unloads but leaves garbage: the unload path (`module_t` event handler, or `DRIVER_MODULE_ORDERED` unload) did not reverse module load. Fix by reading the unload path and making it symmetric to load.

The debug tools are simple: `dmesg | tail` shows the load/unload messages, `kldstat -v` shows the module state, `vmstat -m` shows whether the module's malloc pool was drained on unload.

23.8.9 7.9 A Debug Checklist

Here is a compact checklist that ties tools to symptoms. Readers can keep this at hand during driver development.

Symptom	First tool	Second tool
Bad or missing <code>dmesg</code> line	<code>dmesg</code> , <code>/var/log/messages</code>	<code>add device_printf</code>
<code>InUse</code> grows in <code>vmstat -m</code>	<code>vmstat -m</code>	DTrace malloc count
Panic on lock order	WITNESS trace	<code>ddb> show locks</code>
Sporadic corruption under load	WITNESS, INVARIANTS	DTrace timing
Register reads return <code>0xFFFFFFFF</code>	review <code>bus_space</code> handle	datasheet, offset
Machine hangs on device use	DTrace <code>fbt::myfirst_*</code>	<code>ddb> bt</code>
Panic after detach	<code>DEBUG_MEMGUARD</code>	audit detach order
Slow system under driver load	DTrace <code>intr</code> provider	shorten handler
<code>ioctl</code> with wrong command	<code>ktrace / kdump</code>	decode command
Panic with user-space faulting address	review <code>copyin/copyout</code>	audit <code>ioctl</code> pointer
Module will not unload	<code>kldstat -v</code>	<code>devctl detach</code>

The table is not exhaustive, but it covers the vast majority of real driver bugs. With it and the tools from Sections 1 through 6, the reader has a structured approach to most field diagnostics.

23.8.10 Wrapping Up Section 7

Section 7 turned the toolbox of logging, log analysis, debug kernels, DTrace, and `ktrace` on the bugs the reader is most likely to hit: memory leaks, race conditions, bus-access errors, use-after-detach, interrupt mistakes, lifecycle errors, user/kernel boundary bugs, and module-level issues. For each class, the section described the symptom, the underlying cause, and the tool that diagnoses it most efficiently. The checklist at the end makes

the mapping explicit, and future sessions at the keyboard should let the reader reach for the right tool on the first try rather than the third.

The remaining gap is the driver itself. So far in this chapter, `myfirst` has been used mostly as the target of external tools. Section 8 closes that gap by refactoring the driver to expose trace points and a debug-verbosity knob of its own, so that instrumentation becomes an organic part of the code rather than an afterthought. The version bumps to 1.6-debug and the driver acquires `myfirst_debug.h` and the structure that will stay with it through the remainder of the book.

23.9 Section 8: Refactoring and Versioning With Trace Points

The previous seven sections of this chapter were about learning to use tools that already exist in FreeBSD: `printf`, `dmesg`, debug kernels, DTrace, `ktrace`, and the discipline of reading their output. Section 8 turns inward. The `myfirst` driver that arrived from Chapter 22 as version 1.5-power has no real debug infrastructure of its own. Every logging statement is unconditional. There is no knob the operator can turn, no `sysctl` to request more verbose output when a problem arises, and no static trace points to hook with DTrace. Section 8 fixes that gap.

The work in this section is small in lines of code but large in impact. By the end, the driver will have:

1. A `myfirst_debug.h` header that defines a `DPRINTF` macro, verbosity bits, and SDT trace points.
2. A `sysctl` tree (`dev.myfirst.0.debug`) that sets the verbosity at runtime.
3. An entry-exit-error trace pattern applied consistently across the driver.
4. Three SDT provider probes that expose the open, close, and I/O events.
5. A `DEBUG.md` document in the example tree describing how to configure and read the new output.
6. A version bump to 1.6-debug recorded in `myfirst_version` and `MODULE_VERSION`.

The resulting driver will be the platform for the remaining chapters of the book. Every new subsystem added in Parts 5, 6, and 7 will hook into this framework, so that the trace infrastructure grows with the driver rather than being bolted on at the end.

23.9.1 8.1 Why a Debug Header

Larger FreeBSD drivers have a dedicated header file for debug infrastructure. The pattern is visible in `/usr/src/sys/dev/ath/if_ath_debug.h`, `/usr/src/sys/dev/bwn/if_bwn_debug.h`, `/usr/src/sys/dev/iwn/if_iwn_debug.h`, and many others. Each of these headers:

1. defines a `DPRINTF` macro that checks a verbosity bitmask in the softc
2. declares a set of verbosity classes as `#define MYF_DBG_INIT 0x0001`, `MYF_DBG_OPEN 0x0002`, and so on
3. optionally declares SDT probes that map to the driver's functional boundaries

Putting this in a header has two benefits. First, the debug infrastructure is a separate concern, factored cleanly away from the functional code. Second, when a reader wants to understand the driver's tracing story, the header is the one file to read. The pattern is mature, widely used, and is what `myfirst` will adopt in this section.

The header will be kept under `examples/part-05/ch23-debug/stage3-refactor/myfirst_debug.h`, and the driver source will include it at the top:

```
#include "myfirst_debug.h"
```

The companion changes in the driver source are additive: existing `device_printf` calls can remain, and new calls are added through the `DPRINTF` macro.

23.9.2 8.2 Declaring Debug Classes

The first step is to define the verbosity classes. A class is a single bit in a 32-bit mask. The driver reserves one bit per functional area. For `myfirst`, eight classes are more than enough at this stage:

```
/* myfirst_debug.h */
#ifndef _MYFIRST_DEBUG_H_
#define _MYFIRST_DEBUG_H_

#include <sys/sdt.h>

#define MYF_DBG_INIT      0x00000001 /* probe/attach/detach */
#define MYF_DBG_OPEN      0x00000002 /* open/close lifecycle */
#define MYF_DBG_IO        0x00000004 /* read/write paths */
#define MYF_DBG_IOCTL     0x00000008 /* ioctl handling */
#define MYF_DBG_INTR      0x00000010 /* interrupt handler */
#define MYF_DBG_DMA        0x00000020 /* DMA mapping/sync */
#define MYF_DBG_PWR        0x00000040 /* power-management events */
#define MYF_DBG_MEM        0x00000080 /* alloc/free trace */

#define MYF_DBG_ANY       0xFFFFFFFF
#define MYF_DBG_NONE      0x00000000

#endif /* _MYFIRST_DEBUG_H_ */
```

The value `MYF_DBG_NONE` is the default: no debug output at all. `MYF_DBG_ANY` enables every class, useful during development. A typical operator configuration might enable only `MYF_DBG_INIT | MYF_DBG_OPEN` to get lifecycle events without the per-I/O chatter.

Each bit is declared as a single hex digit or pair, so the operator can set the mask with a simple value: `sysctl dev.myfirst.0.debug=0x3` enables initialisation and open tracing. The commented names make the purpose of each bit clear.

23.9.3 8.3 The DPRINTF Macro

Next, the macro that gates logging on the mask:


```

#ifdef _KERNEL
#define DPRINTF(sc, m, ...) do {                                \
    if ((sc)->sc_debug & (m))                                    \
        device_printf((sc)->sc_dev, __VA_ARGS__);              \
} while (0)
#endif

```

The macro takes three arguments: the softc pointer, the bitmask, and the format string plus variadic arguments. It expands to a test of `sc->sc_debug & m`, followed by a call to `device_printf` if the test succeeds. The `do { ... } while (0)` pattern is the standard idiom for multi-statement macros that must behave like a single statement in `if/else` contexts.

The cost of the `DPRINTF` call when the bit is clear is a single load from `sc->sc_debug`, a bitwise AND, and a branch. The branch is almost always predicted as “not taken”, so the cost in practice is negligible. The driver can sprinkle `DPRINTF` calls freely throughout the code, and the user pays nothing when debug is disabled.

When the bit is set, the call becomes a normal `device_printf`, appearing in `dmesg` like any other kernel log. Nothing in the behaviour differs from the usual logging path, except that the logging is now conditional.

23.9.4 8.4 Adding `sc_debug` to the softc

The softc gains one new field, a `uint32_t sc_debug`. Its value is manipulated by a `sysctl`, which the `attach` function registers. The relevant excerpt from `myfirst_debug.c`:

```

struct myfirst_softc {
    device_t      sc_dev;
    struct mtx     sc_mtx;
    struct cdev    *sc_cdev;
    uint32_t      sc_debug;      /* debug verbosity mask */
    /* other fields as in 1.5-power */
};

```

The `attach` function initialises the field to zero and registers the `sysctl`:

```

sc->sc_debug = 0;
sysctl_ctx_init(&sc->sc_sysctl_ctx);
sc->sc_sysctl_tree = SYSCTL_ADD_NODE(&sc->sc_sysctl_ctx,
    SYSCTL_CHILDREN(device_get_sysctl_tree(sc->sc_dev)),
    OID_AUTO, "debug",
    CTLFLAG_RW, 0, "debug verbosity tree");

SYSCTL_ADD_U32(&sc->sc_sysctl_ctx,
    SYSCTL_CHILDREN(sc->sc_sysctl_tree),
    OID_AUTO, "mask",
    CTLFLAG_RW, &sc->sc_debug, 0, "debug class bitmask");

```

After attach, the sysctl appears as:

```
sysctl dev.myfirst.0.debug.mask
```

and the operator can change it at will:

```
sysctl dev.myfirst.0.debug.mask=0x3      # enable INIT + OPEN
sysctl dev.myfirst.0.debug.mask=0xFFFFFFFF # enable all classes
sysctl dev.myfirst.0.debug.mask=0        # disable
```

The detach routine destroys the sysctl context the same way it does today:

```
sysctl_ctx_free(&sc->sc_sysctl_ctx);
```

One detail matters: placing the sysctl field near the top of the softc puts it in the first cache line, where its access cost during DPRINTF is smallest. This is a small performance point, but the book's emphasis on principled coding should note it.

23.9.5 8.5 The Entry / Exit / Error Pattern

With DPRINTF in place, the driver's functions can use a consistent pattern for tracing. Each substantial function logs entry, exit (or error), and any interesting intermediate state. For a small function like `myfirst_open`, this looks like:

```
static int
myfirst_open(struct cdev *dev, int flags, int devtype, struct thread *td)
{
    struct myfirst_softc *sc = dev->si_drv1;
    int error = 0;

    DPRINTF(sc, MYF_DBG_OPEN, "open: pid=%d uid=%d flags=0x%x\n",
            td->td_proc->p_pid, td->td_ucred->cr_uid, flags);

    mtx_lock(&sc->sc_mtx);
    if (sc->sc_open_count >= MYFIRST_MAX_OPENS) {
        error = EBUSY;
        goto out;
    }
    sc->sc_open_count++;
out:
    mtx_unlock(&sc->sc_mtx);

    if (error != 0)
```

```

        DPRINTF(sc, MYF_DBG_OPEN, "open failed: error=%d\n", error);
    else
        DPRINTF(sc, MYF_DBG_OPEN, "open ok: count=%d\n",
            sc->sc_open_count);

    return (error);
}

```

Three principles guide the pattern:

1. The entry log shows who called and with what arguments. For open, this is pid, uid, and flags.
2. The exit log shows the outcome: either “failed: error=N” or “ok: ...” with any relevant state.
3. Intermediate state is logged when it matters. For open, the new open count is useful.

The pattern is the same for close, read, write, and ioctl. Each function has two or three DPRINTF calls bracketing the work.

This discipline pays off the first time a bug appears. Without the pattern, the developer must add logging retroactively, guessing where the problem is. With the pattern, turning on a single bit (sysctl dev.myfirst.0.debug.mask=0x2 for MYF_DBG_OPEN) produces a complete trace of every open and close, with arguments and outcomes. The diagnostic time drops from hours to minutes.

23.9.6 8.6 Adding SDT Probes

The static DPRINTF macro generates human-readable messages in dmesg. SDT probes produce machine-readable events that DTrace can aggregate, filter, and time. Both have their place. The driver declares SDT probes for the three most interesting events, and a thoughtful user can attach custom scripts at will.

In myfirst_debug.h, the probe declarations look like:

```

#include <sys/sdt.h>

SDT_PROVIDER_DECLARE(myfirst);
SDT_PROBE_DECLARE(myfirst, , , open);
SDT_PROBE_DECLARE(myfirst, , , close);
SDT_PROBE_DECLARE(myfirst, , , io);

```

The matching definitions live in the driver source (myfirst_debug.c):

```

#include <sys/sdt.h>

SDT_PROVIDER_DEFINE(myfirst);
SDT_PROBE_DEFINE2(myfirst, , , open,
    "struct myfirst_softc *", "int");
SDT_PROBE_DEFINE2(myfirst, , , close,

```

```

    "struct myfirst_softc *", "int");
SDT_PROBE_DEFINE4(myfirst, , , io,
    "struct myfirst_softc *", "int", "size_t", "off_t");

```

The `DEFINE` macros register the probes with the kernel's SDT infrastructure. `DEFINEn` takes `n` arguments, each with a C-style type string that describes what the DTrace script will receive as `arg0`, `arg1`, etc.

The driver then fires the probes at the appropriate locations:

```

/* in myfirst_open */
SDT_PROBE2(myfirst, , , open, sc, flags);

/* in myfirst_close */
SDT_PROBE2(myfirst, , , close, sc, flags);

/* in myfirst_read or myfirst_write */
SDT_PROBE4(myfirst, , , io, sc, is_write, (size_t)uio->uio_resid, uio->uio_offset);

```

The cost of a probe that no script has attached to is a single branch around a no-op, the same negligible cost as `DPRINTF` with a clear bit. When a DTrace script does attach, the branch is taken, the arguments are recorded, and the script sees them as `args[0]`, `args[1]`, and so on.

With these probes, a reader can now run scripts like:

```
dtrace -n 'myfirst::: { @[probename] = count(); }'
```

This counts every `myfirst` probe event. To see the bytes transferred per second:

```
dtrace -n 'myfirst:::io { @["bytes"] = sum(arg2); }'
```

Or to trace every open with PID and flags:

```
dtrace -n 'myfirst:::open { printf("open pid=%d flags=0x%x", pid, arg1); }'
```

The driver now exposes its behaviour in two forms: human-readable via `DPRINTF`, and machine-parseable via SDT. An operator can pick the right form for the job.

23.9.7 8.7 Writing `DEBUG.md`

The example tree gains a short document (`examples/part-05/ch23-debug/stage3-refactor/DEBUG.md`) that explains the debug and trace infrastructure to a reader who downloads the files. The document is short but covers:

1. What `DPRINTF` is and how to enable it.

2. The table of class bits.
3. Example `sysctl` commands to enable each class.
4. The list of SDT probes and the argument order.
5. Three example DTrace one-liners.
6. How to combine `DPRINTF` and SDT for end-to-end debugging.

The document is not long: about thirty lines. Its purpose is to make the tooling self-documenting, so a reader who picks up the example in a year still knows how to use it.

23.9.8 8.8 Bumping the Version

Every chapter in the book that changes the driver's behaviour bumps its version. The rule is: the version string in the driver matches the version shown in the example-tree README and in the chapter's text.

In Chapter 22, the driver reached 1.5-power. Section 8 of Chapter 23 takes it to 1.6-debug. The changes in `myfirst_debug.c` are:

```
static const char myfirst_version[] = "myfirst 1.6-debug";

MODULE_VERSION(myfirst, 16);
```

The text above the version constant says explicitly what changed:

```
/*
 * myfirst driver - version 1.6-debug
 *
 * Added in this revision:
 *   - DPRINTF macro with 8 verbosity classes
 *   - sysctl dev.myfirst.0.debug.mask for runtime control
 *   - SDT probes for open, close, io
 *   - Entry/exit/error pattern across all methods
 */
```

The attach function logs the version at the `MYF_DBG_INIT` level, so the operator can confirm the loaded driver:

```
DPRINTF(sc, MYF_DBG_INIT, "attach: %s loaded\n", myfirst_version);
```

With the verbosity enabled, a reader who loads the driver sees:

```
myfirst0: attach: myfirst 1.6-debug loaded
```

which confirms both the module identity and the debug infrastructure.

23.9.9 8.9 Wrapping Section 8

Section 8 refactored the `myfirst` driver to carry its own debug infrastructure. The driver now has a verbosity mask, a `sysctl` to set it at runtime, a consistent entry-exit-error pattern in each function, and three SDT probes at the functional boundaries. The debug header can be included in future chapters, so every subsystem added later in the book can hook into the same framework.

The driver's version advances from 1.5-power to 1.6-debug. The example tree gets a `stage3-refactor` directory with the final source, a `DEBUG.md` document, and a `README.md` that guides the reader through building, loading, and exercising the new infrastructure.

With Section 8 complete, the chapter has covered the full arc: understand why kernel debugging is hard, use every tool FreeBSD provides, recognise the common bugs, and finally build the driver so that it supports its own debugging. The remaining material in the chapter is the lab sequence, the challenges, and the closing material that ties this chapter to Chapter 24.

23.10 Hands-On Labs

The labs in this chapter form a progression. Each one reinforces a specific tool from Sections 1 through 6, and the final lab applies the refactor from Section 8 so the reader walks away with a driver that is instrumented and ready for the rest of the book.

None of these labs is long. All five can be completed in a single evening, though spreading them across two or three sessions gives the reader time to absorb each tool.

23.10.1 Lab 23.1: A First DDB Session

Goal: Enter the kernel debugger deliberately, inspect the driver's state, and exit safely.

Prerequisites: A debug kernel built as described in Section 4, with options `KDB` and options `DDB`. The `myfirst` driver from Chapter 22 (version 1.5-power) loaded.

Steps:

1. Confirm the debug kernel is running:

```
sysctl kern.version
sysctl debug.kdb
```

The second command should show DDB among the available backends.

2. Load the driver and confirm the device node:

```
sudo kldload myfirst
ls /dev/myfirst0
```

3. Enter DDB with a keyboard break. On the system console, press `Ctrl-Alt-Esc` on a VGA console, or send the `BREAK` character on a serial console. The prompt appears:

```
KDB: enter: manual entry
[thread pid 42 tid 100024 ]
Stopped at      kdb_enter+0x37: movq    $0,0x158e4fa(%rip)
db>
```

4. Show the device tree:

```
db> show devmap
```

Locate `myfirst0` in the output.

5. Print the backtrace of the current thread:

```
db> bt
```

6. Print all processes:

```
db> ps
```

Confirm the reader's own shell is visible in the list.

7. Continue the kernel:

```
db> continue
```

The system returns to normal operation.

What to observe:

- The kernel halted cleanly on the break.
- All devices, processes, and kernel threads were inspectable.
- The `continue` command returned the system to normal without any side effect.

Record the time spent and any notes in the lab logbook under “Chapter 23, Lab 1”. The reader should come away understanding that DDB is a safe tool when used deliberately.

23.10.2 Lab 23.2: Measuring the Driver with DTrace

Goal: Use DTrace to measure the `myfirst` driver's open, close, and I/O rates under a simple workload.

Prerequisites: A kernel with options `KDTRACE_HOOKS` and `makeoptions WITH_CTF=1` loaded. The `myfirst` driver loaded at version 1.5-power (SDT probes are not yet present at this stage; the lab uses `fbt` only).

Steps:

1. Confirm DTrace works:

```
sudo dtrace -l | head -5
```

2. Start a DTrace script that counts entries to each `myfirst` function:

```
sudo dtrace -n 'fbt::myfirst_*:entry { @[probefunc] = count(); }'
```

3. In a second terminal, exercise the driver:

```
for i in $(seq 1 100); do cat /dev/myfirst0 > /dev/null; done
```

4. Stop the DTrace script (Ctrl-C in the first terminal) and read the count:

```
myfirst_open          100
myfirst_close         100
myfirst_read          100
```

5. Now measure the time spent in myfirst_read:

```
sudo dtrace -n 'fbt::myfirst_read:entry { self->t = timestamp; }
               fbt::myfirst_read:return /self->t/ {
                 @ = quantize(timestamp - self->t); self->t = 0; }'
```

6. Exercise the driver with 1000 reads, stop DTrace, and read the quantization.

What to observe:

- Every cat produced one open, one read, and one close, matching the count.
- The time per read is small (microseconds) and concentrated in one bucket of the quantization.
- No modification was made to the driver; the measurement is non-invasive.

23.10.3 Lab 23.3: User-Side Tracing with ktrace

Goal: Trace a user program that exercises the myfirst driver and verify the syscall sequence.

Prerequisites: The driver loaded. The test program myftest.c from Section 6.4 compiled.

Steps:

1. Compile the test program (from Section 6.4):

```
cc -o myftest myftest.c
```

2. Run it under ktrace:

```
sudo ktrace -di -f mytrace.out ./myftest
```

3. Dump the trace:

```
kdump -f mytrace.out | less
```

4. Locate the syscalls relevant to the driver:

```
kdump -f mytrace.out | grep -E 'myfirst0|CALL|RET'
```

5. Correlate the trace with the driver's DTrace counts from Lab 23.2. Every user-side syscall should match one or more kernel-side function entries.

What to observe:

- The user program's view of the driver is complete in the trace.
- Arguments, return values, and error codes are all visible.
- User-side tracing complements the DTrace kernel-side view.

23.10.4 Lab 23.4: Finding a Memory Leak with `vmstat -m`

Goal: Introduce a deliberate leak, detect it with `vmstat -m`, and fix it.

Prerequisites: The driver source under the reader's control. A compiled, loadable module.

Steps:

1. Modify `myfirst_open` to allocate a small buffer and store it in `si_drv2`, without freeing it in `myfirst_close`:

```
/* in myfirst_open */  
dev->si_drv2 = malloc(128, M_MYFIRST, M_WAITOK | M_ZERO);
```

2. Build and load the driver.
3. Run a workload that opens and closes the device many times:

```
for i in $(seq 1 1000); do cat /dev/myfirst0 > /dev/null; done
```

4. Check the memory pool:

```
vmstat -m | grep myfirst
```

The `InUse` column should show 1000 (or more), and `MemUse` should show roughly 128 KB.

5. Fix the leak by adding `free(dev->si_drv2, M_MYFIRST);` to `myfirst_close`.
6. Build and reload the driver. Run the workload again.
7. Check the pool:

```
vmstat -m | grep myfirst
```

`InUse` should now stabilise near zero after the workload.

What to observe:

- The `vmstat -m` output exposed the leak immediately.
- Without a specialist tool, the leak would have gone unnoticed until the kernel ran out of memory.
- The fix is mechanical and easy once the symptom is identified.

Do not leave the leak-inducing code in the driver after the lab. Revert to the clean version before continuing.

23.10.5 Lab 23.5: Installing the 1.6-debug Refactor

Goal: Apply the Section 8 refactor to the `myfirst` driver and confirm the new infrastructure works.

Prerequisites: The driver source from Chapter 22 under the reader's control. A development environment able to build kernel modules.

Steps:

1. Create `myfirst_debug.h` exactly as shown in Section 8.2, and the SDT declarations from Section 8.6.
2. Update the `softc` to add `uint32_t sc_debug;`.
3. Register the `sysctl` in `myfirst_attach` as shown in Section 8.4.
4. Replace unconditional `device_printf` calls with `DPRINTF(sc, MYF_DBG_<class>, ...)` calls as appropriate.
5. Define the SDT providers and probes in `myfirst_debug.c` (or `myfirst.c`), and fire them in `open`, `close`, and `read/write`.
6. Bump `myfirst_version` to 1.6-debug and `MODULE_VERSION` to 16.
7. Build:

```
cd /path/to/myfirst-source
make clean && make
```

8. Reload:

```
sudo kldunload myfirst
sudo kldload ./myfirst.ko
```

9. Confirm the module loaded at the new version:

```
kldstat -v | grep myfirst
```

10. Confirm the `sysctl` is present:

```
sysctl dev.myfirst.0.debug.mask
```

11. Enable full verbosity and open the device:

```
sudo sysctl dev.myfirst.0.debug.mask=0xFFFFFFFF
cat /dev/myfirst0 > /dev/null
dmesg | tail
```

Expected output includes lines like:

```
myfirst0: open: pid=1234 uid=1001 flags=0x0
myfirst0: open ok: count=1
myfirst0: read: size=4096 off=0
```

12. Disable verbosity and confirm the messages stop:

```
sudo sysctl dev.myfirst.0.debug.mask=0
cat /dev/myfirst0 > /dev/null
dmesg | tail
```

13. Attach DTrace to the SDT probes:

```
sudo dtrace -n 'myfirst::: { @[probename] = count(); }'
```

In a second terminal, exercise the driver:

```
for i in $(seq 1 100); do cat /dev/myfirst0 > /dev/null; done
```

Stop DTrace and confirm the probes fired.

What to observe:

- Runtime control of verbosity is immediate and responsive.
- The DPRINTF path produces human-readable messages; SDT probes produce machine-parseable events.
- The two forms are complementary and independent.
- Both are turned off by default, so production users pay no runtime cost.

The driver is now at version 1.6-debug, with infrastructure that will support every later chapter. Record the upgrade in the logbook.

23.11 Challenge Exercises

The exercises below build on the chapter material. They are open-ended enough that each reader will arrive at slightly different answers. Take your time; work incrementally; use the tools from the chapter at each step.

23.11.1 Challenge 23.1: An SDT Probe in a Real Driver

Pick a driver from `/usr/src/sys/dev/` that already contains SDT probes. Good candidates include `virtqueue.c`, some `ath` files, or `if_re.c`.

For the chosen driver:

1. List its probes with `dtrace -l -P <provider>`.
2. Write a one-liner that counts events grouped by probe name.
3. Write a second one-liner that aggregates by one of the probe arguments (for example, a packet length or a command number).
4. Explain in three sentences what the driver's author gained by adding the probes.

23.11.2 Challenge 23.2: A Custom DTrace Script

Write a DTrace script that:

1. Attaches to the `myfirst` SDT probes (added in Lab 23.5).
2. Tracks the lifetime of each open file descriptor by pid.
3. Prints a summary when a fd closes, showing: pid, how many reads, how many writes, total bytes, and the elapsed time between open and close.

The script should be at most 50 lines of D code. Test it by running a small user program that opens, reads, writes, and closes `/dev/myfirst0` several times.

23.11.3 Challenge 23.3: A WITNESS Experiment

Modify the driver to contain a deliberate lock-ordering mistake. For example, add two mutexes `sc_mtx_a` and `sc_mtx_b`, and arrange one code path to take A then B while another path takes B then A.

Build with options `WITNESS`, load, and trigger the mistake. Capture the resulting kernel panic or WITNESS output. Describe in three sentences what WITNESS showed, why it caught the issue, and what the fix would be.

Be sure to revert the mistake after the exercise. Do not leave a broken driver loaded.

23.11.4 Challenge 23.4: Extended Error Categorisation

Choose any four error paths in the `myfirst` driver (for example, the `EBUSY` path in `myfirst_open`, an `EINVAL` path in `myfirst_ioctl`, and so on). For each:

1. Identify the underlying cause that would produce this error in the real world.
2. Add a `DPRINTF(sc, MYF_DBG_<class>, ...)` that captures the cause clearly.
3. Write a brief note in `DEBUG.md` explaining what to look for in `dmesg` when the error occurs.

The goal is to make every error in the driver self-explanatory without requiring the reader to cross-reference the source code.

23.11.5 Challenge 23.5: Boot-Time Debug

Modify the driver to log the attach-time version and hardware details only when `bootverbose` is set. The effect should be: a normal boot shows one line, a verbose boot (requested with `boot -v`) shows detailed configuration.

Read `/usr/src/sys/sys/system.h` for the declaration of `bootverbose`, and `/usr/src/sys/kern/subr_boot.c` for examples. Describe in two sentences what FreeBSD uses `bootverbose` for, and why it is better than a driver-specific verbose flag for the particular case of early boot.

23.12 Troubleshooting and Common Mistakes

Every reader will hit at least a few of the issues below. Each one is recorded with the symptom, the likely cause, and the path to a fix.

“My `kldload` fails with `link_elf: symbol X undefined.`”

- Cause: the module depends on a kernel symbol that is not in the running kernel. Usually this means the kernel is older than the module source, or the module was compiled against a different kernel build.
- Fix: rebuild the kernel and module from the same source tree, using the same compile flags. Confirm `uname -a` and the module’s build directory match.

“DTrace says `invalid probe specifier.`”

- Cause: the probe name is mistyped, or the provider/module/function field does not exist at runtime.

- Fix: run `dtrace -l | grep <provider>` to list the available probes, and pick a name from the list. Remember that wildcards must match an existing name.

“WITNESS panics at boot with WITNESS_CHECKORDER.”

- Cause: an early driver (often third-party) violates the declared lock order. Under a stock kernel the violation was silent; under WITNESS it causes an immediate panic.
- Fix: disable the offending driver temporarily, or set `debug.witness.watch=0` at boot time to disable the check, or rebuild the offending driver with a corrected lock order.

“DPRINTF messages do not appear in dmesg.”

- Cause: the corresponding bit in `sc_debug` is not set.
- Fix: confirm the `sysctl` with `sysctl dev.myfirst.0.debug.mask`. Set the bit: `sysctl dev.myfirst.0.debug.mask`. Retry.

“DTrace says probe description myfirst::: matched 0 probes.”

- Cause: either the driver is not loaded, or CTF was not generated (the SDT probes are declared but not visible to DTrace).
- Fix: run `kldstat | grep myfirst` and confirm the module is loaded. If it is, rebuild the kernel with `makeoptions WITH_CTF=1` and reboot.

“Memory leak persists after my fix.”

- Cause: there is more than one leak. The first fix addressed one path, but another path is still leaking.
- Fix: re-examine the `vmstat -m` numbers. If they still grow, trace the allocator: add `device_printf` at every `malloc` and `free` call, reload, exercise the driver, and count the appearances.

“The driver unloads, but vmstat -m still shows nonzero InUse.”

- Cause: the unload path did not release all memory. This is often because something was allocated by `open` but not freed by `close`, and the `close` path was never called before `detach`.
- Fix: verify `destroy_dev_sched_cb` is used to wait for outstanding opens before `detach` proceeds. Every `si_drv1` or `si_drv2` buffer allocated per `open` must be freed in `close` or in the `detach`-ordered callback.

“My fbt probe has a misleading name.”

- Cause: some drivers use inline functions or static helpers that the compiler may have inlined away. The compiled binary shows only the containing function.
- Fix: compile the module with `-O0` or add `__noinline` to the helper. This is useful for debugging only; release builds should use the normal optimisation level.

“ktrace records are too long to read.”

- Cause: default `ktrace` captures many event classes at once.
- Fix: limit to specific classes with `-t` flag: `ktrace -t c ./myftest` captures only syscalls, not I/O or NAMI.

“The kernel boots but prints no driver messages at all.”

- Cause: either the driver is not built into the kernel, or the driver’s probe returned a non-zero error.

- Fix: `kldstat -v | grep myfirst` confirms the module is loaded. `dmesg | grep -i 'probe'` finds the probe messages. If the module is loaded but the driver is not attached, check `device_set_desc` and the probe function's return value.

23.13 Appendix: Real-World Debug Patterns in the FreeBSD Tree

The patterns in Sections 7 and 8 are based on conventions already used by production drivers. A reader who inspects the tree will see variations on these conventions everywhere. This appendix gives a short tour.

if_ath_debug.h (`/usr/src/sys/dev/ath/if_ath_debug.h`) defines approximately 40 verbosity classes covering reset, interrupt, RX, TX, beacon, rate control, channel selection, and many others, each expressed as a bit in a 64-bit mask. The class names follow the `ATH_DEBUG_<subsystem>` pattern (for example `ATH_DEBUG_RECV`, `ATH_DEBUG_XMIT`, `ATH_DEBUG_RESET`, `ATH_DEBUG_BEACON`), and the `DPRINTF` macro follows the standard pattern. The driver uses `ATH_DEBUG_<class>` in hundreds of places throughout the source, giving operators fine-grained control over trace output.

virtqueue.c (`/usr/src/sys/dev/virtio/virtqueue.c`) uses SDT probes heavily. A single `SDT_PROVIDER_DEFINE(virtqueue)` at the top declares the provider, followed by a dozen `SDT_PROBE_DEFINE` calls for each significant event: enqueue, dequeue, notify, and so on. The driver fires the probes inside the hot path, and the probes are no-ops when no script is attached. DTrace scripts like `dtrace -n 'virtqueue::: { @[probename] = count(); }'` give instant visibility into virtio device activity.

if_re.c (`/usr/src/sys/dev/re/if_re.c`) mixes `device_printf`, `printf`, and `if_printf`. The `device_printf` calls appear at attach and detach, where the device identity is useful. `if_printf` appears in the packet paths, where the interface identity matters. `printf` appears in the shared helpers, where neither identity is available. The division of responsibility is pragmatic, not ideological.

uart_core.c (`/usr/src/sys/dev/uart/uart_core.c`) defines `UART_DBG_LEVEL` and a family of `UART_DBG` macros that check the level and log through `printf`. The level is set at compile time via options `UART_POLL_FREQ` and options `UART_DEV_TOLERANCE_PCT`, among others. The design is static: debug decisions are fixed at build time rather than runtime. This is the right choice for a driver that must run in very early boot, when `sysctl` is not yet available.

random_harvestq.c (`/usr/src/sys/dev/random/random_harvestq.c`) uses `WITNESS` extensively. Every lock in the file has a `mtx_assert(&lock, MA_OWNED)` check at the head of each function that relies on the lock. When `WITNESS` is enabled in the build, these assertions catch any caller that forgot to acquire the lock. In release builds, the assertion compiles out and has zero cost.

Across the tree, the pattern is consistent: drivers expose their internals through a combination of static debug logging, SDT probes, and assertion macros. Each of these costs nothing at runtime when disabled, and gives rich visibility when enabled. The reader who adopts this pattern in their own driver writes code that scales gracefully from first-day debugging through production monitoring.

23.14 Appendix: Worked Case Studies in Driver Debugging

The material in this chapter is most useful when it is rehearsed against concrete bugs. This appendix walks through three realistic case studies. Each one begins with a symptom the reader might encounter, follows the

diagnostic steps the tools of this chapter enable, arrives at a root cause, and records the fix. The three cases together cover the three broad categories of driver bugs: a correctness bug in the logging path, a lock-order bug, and a performance bug in the interrupt path.

The cases are written as narrative rather than as a dry sequence of commands because debugging is a narrative process. The experienced developer does not follow a single algorithm; they read a trace, form a hypothesis, test it, refine it, and iterate. These walkthroughs try to preserve that texture while keeping each step reproducible.

23.14.1 Case Study 1: The Missing Message

Symptom. A reader has added a `DPRINTF(sc, MYF_DBG_OPEN, "open ok: count=%d\n", sc->sc_open_count)` line to `myfirst_open`. They set `sysctl dev.myfirst.0.debug.mask=0x2` to enable `MYF_DBG_OPEN`, open the device, and check `dmesg`. Nothing appears.

First hypothesis: the bit is wrong. The reader double-checks the mask. `MYF_DBG_OPEN` is `0x02`, so setting `mask=0x2` should enable it. The hypothesis is wrong.

Second hypothesis: the device is not being opened. The reader runs `cat /dev/myfirst0 > /dev/null` and checks `dmesg`. Still nothing.

Third hypothesis: the sysctl is not actually setting the field. The reader reads the `sysctl` back:

```
sysctl dev.myfirst.0.debug.mask
```

The output shows `0x0`. The write did not take effect. This is the real problem.

Narrowing down. The reader inspects the attach code and finds:

```
SYSCALL_ADD_U32(&sc->sc_sysctl_ctx,  
    SYSCALL_CHILDREN(sc->sc_sysctl_tree),  
    OID_AUTO, "mask",  
    CTLFLAG_RD, &sc->sc_debug, 0, "debug class bitmask");
```

Note: `CTLFLAG_RD` (read-only), not `CTLFLAG_RW`. The `sysctl` was declared read-only, so the `sysctl` command appeared to succeed but actually silently refused the write.

Fix. Change the flag to `CTLFLAG_RW`:

```
SYSCALL_ADD_U32(&sc->sc_sysctl_ctx,  
    SYSCALL_CHILDREN(sc->sc_sysctl_tree),  
    OID_AUTO, "mask",  
    CTLFLAG_RW, &sc->sc_debug, 0, "debug class bitmask");
```

Rebuild, reload, set the mask, open the device, and the message appears.

Lessons learned. Three tools combined to find this bug in under a minute: `dmesg` showed the symptom (missing message), `sysctl` exposed the real state (write not taking effect), and the driver source made the cause visible

(CTLFLAG_RD). Without any one of these, the reader might have wasted ten minutes on a wrong hypothesis. The tools do not replace thinking; they provide the evidence that thinking needs.

The sysctl read-back is a useful discipline in general. Any time a driver value is supposed to be writable, test the write by reading it back. This one habit will catch a class of bugs that have bitten every FreeBSD developer at least once.

23.14.2 Case Study 2: The Intermittent Panic

Symptom. Under light load the driver works perfectly. Under a stress workload (say, one hundred simultaneous readers), the kernel panics with:

```
panic: mutex myfirst_lock recursed on non-recursive mutex myfirst
```

The panic is intermittent: sometimes the workload runs for five minutes before panicking, sometimes it panics immediately. The reader confirms that a debug kernel with options `WITNESS` and options `INVARIANTS` is in use.

First step: read the stack. The panic message includes a stack trace. The relevant lines point at `myfirst_read` as the recursion site, with `myfirst_read` itself already on the stack higher up. Somehow, `myfirst_read` is being called while a previous call to `myfirst_read` is still executing, and both calls are on the same thread.

Hypothesis. Something inside `myfirst_read` is triggering a secondary `myfirst_read` call. The most likely candidate is a `uiomove`, which calls the user program's page-fault path, which in very unusual circumstances can call back into device I/O.

Verification. The reader adds two `DPRINTF` calls, one at entry and one just before `uiomove`, each capturing the current thread:

```
DPRINTF(sc, MYF_DBG_IO, "read entry: tid=%d\n", curthread->td_tid);  
DPRINTF(sc, MYF_DBG_IO, "read about to uiomove: tid=%d\n", curthread->td_tid);
```

Rebuild, reload, run the workload. The kernel log shows two “read entry” messages for the same tid before any “about to uiomove”, which confirms the hypothesis.

Diagnosis. The `mtx_lock(&sc->sc_mtx)` call in `myfirst_read` is held during `uiomove`. The `uiomove` target happens to be in a memory-mapped region of `/dev/myfirst0`, because the test workload was using `mmap` output. The page fault that `uiomove` triggers re-enters `myfirst_read` to service the fault, which attempts to acquire `sc->sc_mtx` a second time.

Fix. The lock should be released before `uiomove`. In general, a sleep lock must never be held across a function that may page-fault.

```
/* buggy version */  
mtx_lock(&sc->sc_mtx);  
error = uiomove(sc->sc_buffer, sc->sc_bufsize, uio);  
mtx_unlock(&sc->sc_mtx);
```



```

/* fixed version */
mtx_lock(&sc->sc_mtx);
/* snapshot the buffer so we can release the lock */
tmp_buffer = sc->sc_buffer;
tmp_bufsize = sc->sc_bufsize;
mtx_unlock(&sc->sc_mtx);
error = uiomove(tmp_buffer, tmp_bufsize, uio);

```

Rebuild, reload, run the stress workload. The panic is gone.

Lessons learned. WITNESS did not catch this directly because it is a recursion, not a lock-order violation. But INVARIANTS turned the recursion into a clean panic instead of silent corruption, and the DPRINTF trace made the chronology visible. The rule “do not hold a sleep lock across uiomove or any function that may sleep” is stated in locking(9) but is easy to forget in practice. Every sleep-lock acquisition in driver code deserves a momentary “could the wrapped call sleep or fault?” check.

23.14.3 Case Study 3: The Slow Interrupt

Symptom. A driver works correctly but the system is sluggish when the device is active. Shell responsiveness drops, interactive applications stutter, and top shows high system CPU.

First step: confirm the driver is the cause. The reader unloads the driver and compares:

```

# before unload
top -aC1 -bn1 | head -20
sudo kldunload myfirst
# after unload
top -aC1 -bn1 | head -20

```

If the sluggishness disappears when the driver unloads, the driver is the source.

Second step: identify the expensive code path. DTrace’s profile provider samples the kernel at a fixed rate and shows where time is being spent:

```

sudo dtrace -n 'profile-1001hz /arg0/ { @[stack(10)] = count(); } tick-10s { exit(0); }'

```

After ten seconds of the workload, the output shows the hottest stacks. If myfirst_intr dominates the sample, the interrupt handler is too heavy.

Third step: measure how long each interrupt takes.

```

sudo dtrace -n 'fbt::myfirst_intr:entry { self->t = timestamp; }
               fbt::myfirst_intr:return /self->t/ {
                   @ = quantize(timestamp - self->t); self->t = 0; }'

```

A healthy handler completes in under 10 microseconds (10 000 nanoseconds). If the quantization shows most invocations in the 100 000 to 1 000 000 nanosecond range (100 microseconds to 1 millisecond), the handler is doing too much work.

Fourth step: read the handler code. The reader inspects `myfirst_intr` and finds:

```
static void
myfirst_intr(void *arg)
{
    struct myfirst_softc *sc = arg;
    struct mbuf *m;

    mtx_lock(&sc->sc_mtx);
    m = m_getcl(M_NOWAIT, MT_DATA, M_PKTHDR);
    if (m != NULL) {
        myfirst_process_data(sc, m);
        m_freem(m);
    }
    mtx_unlock(&sc->sc_mtx);
}
```

The handler does real work: it allocates an mbuf, processes data, and frees the mbuf. `myfirst_process_data` includes several packets of work, each with their own allocation and processing. Under load, each interrupt can hold the CPU for hundreds of microseconds.

Fix. The heavy work moves to a taskqueue. The handler itself does the absolute minimum: acknowledges the interrupt and signals the taskqueue.

```
static void
myfirst_intr(void *arg)
{
    struct myfirst_softc *sc = arg;

    /* Acknowledge the interrupt */
    bus_write_4(sc->sc_res, MYFIRST_REG_INTR_STATUS, 0);
    /* Schedule deferred processing */
    taskqueue_enqueue(sc->sc_tq, &sc->sc_process_task);
}

static void
myfirst_process_task(void *arg, int pending)
{
    struct myfirst_softc *sc = arg;
    struct mbuf *m;

    mtx_lock(&sc->sc_mtx);
```

```

    m = m_getcl(M_NOWAIT, MT_DATA, M_PKTHDR);
    if (m != NULL) {
        myfirst_process_data(sc, m);
        m_freem(m);
    }
    mtx_unlock(&sc->sc_mtx);
}

```

The taskqueue is set up in attach as explored in Chapter 19.

Rebuild, reload, run the workload. Shell responsiveness returns, top shows the driver contributing a normal amount of system CPU, and the latency quantization of myfirst_intr is now concentrated in the 1 to 10 microsecond bucket.

Lessons learned. Three DTrace measurements guided the fix: profile-1001hz located the hot code, fbt::myfirst_intr:entry/return measured the handler duration, and the final post-fix measurement confirmed the improvement. No source code was modified during diagnosis; the modifications were made only after the bug was understood.

The rule that interrupt handlers must be short is familiar to driver authors, but enforcement is not automatic. Debug builds do not panic on a slow handler. The only way to catch this category of bug is to measure, and DTrace is the right tool for the job. A mature driver team runs fbt::<driver>_intr:entry/return profiling as part of every release cycle; their handlers stay lean because the measurements make any drift visible.

23.14.4 Case Study 4: The Vanishing Device

Symptom. After kldunload myfirst, a second kldload myfirst sometimes panics with a page fault in devfs_ioctl, stack pointing at the old myfirst_ioctl. The panic does not happen every time, but under load (for example, if a daemon is keeping /dev/myfirst0 open across the unload) it is consistently fatal.

First step: read the panic. The stack shows devfs_ioctl -> myfirst_ioctl -> (address not found). The “address not found” is the faulting instruction. This is a use-after-free of function code: the kernel is trying to call a function whose memory has been unmapped.

Hypothesis. The driver’s unload path did not wait for a process that had the device open. The process issued an ioctl between the unload and the re-load; the ioctl dispatched to the now-unmapped myfirst_ioctl.

Verification. The reader inspects myfirst_detach and finds:

```

static int
myfirst_detach(device_t dev)
{
    struct myfirst_softc *sc = device_get_softc(dev);

    destroy_dev(sc->sc_cdev);
    bus_release_resource(dev, SYS_RES_MEMORY, 0, sc->sc_mem);
    mtx_destroy(&sc->sc_mtx);
}

```

```

        return (0);
    }

```

The `destroy_dev` call is the issue: it immediately destroys the device node, but in-flight ioctls on the node may still be executing. When `destroy_dev` returns, the driver thinks it is safe to release resources. But the ioctl is still dispatching when those resources go away.

Fix. Use `destroy_dev_sched_cb` (or equivalently, `destroy_dev_sched` and then wait for completion) to defer the actual destruction until no thread is inside the device's methods.

```

static int
myfirst_detach(device_t dev)
{
    struct myfirst_softc *sc = device_get_softc(dev);

    /* Tell devfs to destroy the node after all callers finish */
    destroy_dev_sched_cb(sc->sc_cdev, myfirst_detach_cb, sc);
    /* Return success; cleanup happens in the callback */
    return (0);
}

static void
myfirst_detach_cb(void *arg)
{
    struct myfirst_softc *sc = arg;

    bus_release_resource(sc->sc_dev, SYS_RES_MEMORY, 0, sc->sc_mem);
    mtx_destroy(&sc->sc_mtx);
}

```

The detach returns immediately; the actual resource release happens in the callback, which is scheduled to fire only when every caller inside the `cdev`'s methods has exited.

Rebuild, reload repeatedly under the problematic workload. The panic is gone.

Lessons learned. This is the classic use-after-detach bug described in Section 7.4. The debug kernel's `DEBUG_MEMGUARD` option would have caught it earlier by poisoning the freed memory, turning the latent corruption into an immediate page fault on the next call. The fix uses a FreeBSD primitive (`destroy_dev_sched_cb`) that exists precisely for this case. Reading the `cdev`-management documentation in `cdev(9)` is the exercise that prevents this bug in new code.

23.14.5 Case Study 5: The Corrupted Output

Symptom. A user program reads from `/dev/myfirst0` expecting a specific pattern (say, the sequence `0x01`, `0x02`, `0x03`, ...). Most of the time the pattern is correct. Occasionally, on a heavily loaded system, the

pattern is off: a byte or two is wrong, or the sequence is shifted. There is no panic, no error code, just subtly wrong data.

First step: reproduce reliably. The reader writes a small test program that reads repeatedly and compares against the expected pattern. Under load, the mismatches occur at roughly one in ten thousand reads. The reader confirms that unloading the driver and using a simple `/dev/zero` test produces no mismatches, which rules out the user program as the source of the bug.

Hypothesis 1: a race on the buffer. If two threads are reading concurrently, and the buffer is shared without locking, they might corrupt each other. The reader inspects the driver and finds:

```
static int
myfirst_read(struct cdev *dev, struct uio *uio, int ioflag)
{
    struct myfirst_softc *sc = dev->si_drv1;

    /* Generate the pattern into sc->sc_buffer */
    for (int i = 0; i < sc->sc_bufsize; i++)
        sc->sc_buffer[i] = (uint8_t)(i + 1);

    return (uiomove(sc->sc_buffer, sc->sc_bufsize, uio));
}
```

The buffer is per-softc, shared across all opens. Two threads calling read simultaneously write to the same buffer; the second thread overwrites the first's content before the first has finished copying. The uiomove of the first thread then reads partially from the first thread's pattern and partially from the second thread's. The result is a garbled sequence.

Verification. The reader adds a DPRINTF at the start and end of the read, capturing the thread's tid:

```
DPRINTF(sc, MYF_DBG_IO, "read start: tid=%d\n", curthread->td_tid);
/* ... generate and uiomove ... */
DPRINTF(sc, MYF_DBG_IO, "read end: tid=%d\n", curthread->td_tid);
```

When the log is examined during a corruption event, two start messages appear between a single start and its end. The race is confirmed.

Fix. The buffer must either be per-call (allocated on each read) or the access must be serialised with a mutex:

```
static int
myfirst_read(struct cdev *dev, struct uio *uio, int ioflag)
{
    struct myfirst_softc *sc = dev->si_drv1;
    uint8_t *buf;
    int error;
```

```
    buf = malloc(sc->sc_bufsize, M_MYFIRST, M_WAITOK);
    for (int i = 0; i < sc->sc_bufsize; i++)
        buf[i] = (uint8_t)(i + 1);

    error = uiomove(buf, sc->sc_bufsize, uio);
    free(buf, M_MYFIRST);
    return (error);
}
```

A per-call allocation removes the shared state entirely, and the bug goes away. A lock-based fix is faster for small buffers but allows only one reader at a time; the trade-off depends on the driver's use case.

Lessons learned. The corruption is subtle precisely because it does not produce a panic or an error. WITNESS cannot help here because there is no lock to order. INVARIANTS cannot help because the shared state does not trigger an assertion. The only way to find this is: test the output, observe the corruption, form a hypothesis about races, add instrumentation to confirm the hypothesis, and fix. The DPRINTF pattern is the instrumentation; a disciplined test harness is the reproduction. Together, they solve the problem.

This class of bug is the strongest argument for driver authors to write test harnesses from the start. A small program that reads from the device and compares against an expected output catches many bugs in minutes that would otherwise survive for months in the field.

23.14.6 Summary of the Five Cases

The five case studies covered a deliberately diverse set of bugs. Case 1 was a logging correctness bug, found in seconds once the sysctl was read back. Case 2 was a lock-recursion triggered by a paging path, found by combining a panic trace and DPRINTF. Case 3 was a performance bug in the interrupt handler, diagnosed with DTrace profiling. Case 4 was a use-after-detach in the cdev path, diagnosed with the panic stack and fixed with the correct FreeBSD primitive. Case 5 was a shared-state race in the read path, found with a test harness and DPRINTF.

Three patterns recur across the cases:

1. **Tools find symptoms; reading code finds causes.** No tool pointed directly at the buggy line. Each tool narrowed the search: `vmstat -m`, WITNESS, DTrace, DDB. Then the reader opened the source and found the mistake.
2. **Reproduction is more important than tooling.** A bug that reproduces reliably can be diagnosed with almost any tool. A bug that does not reproduce must be reproduced first, usually by scripting a workload that exercises the suspect path.
3. **Debug infrastructure pays its cost many times over.** The DPRINTF macro, the SDT probes, the `sc_debug` sysctl: each of these took five minutes to add, and each saved hours of debugging later. The reader who writes instrumented code from the start is the reader who avoids the long, painful debug sessions.

With these patterns in mind, the reader is ready to carry the chapter's methods forward. Every new piece of the `myfirst` driver added in later chapters will be developed with DPRINTF calls, SDT probes, and test harnesses in place; the drop in debugging pain will be noticeable from Chapter 24 onward.

23.15 Appendix: Extended DTrace Techniques for Driver Work

This appendix extends Section 5 with a focused tour of DTrace techniques that come up often in driver development. Section 5 introduced the basics; this appendix shows how to combine them productively. Readers who find DTrace useful may want to keep this appendix as a cheat sheet.

23.15.1 Predicates and Selective Tracing

A predicate narrows a probe to events that match a condition. It appears in `/ . . . /` between the probe description and the action:

```
dtrace -n 'fbt::myfirst_read:entry /execname == "myftest"/ {  
    @ = count(); }'
```

Only reads from processes named `myftest` are counted. Predicates can reference `pid`, `tid`, `execname`, `zone-name`, `uid`, and the probe arguments `arg0`, `arg1`, etc.

A common pattern is to filter by `softc` pointer to focus on a single device instance:

```
dtrace -n 'fbt::myfirst_read:entry /args[0] == 0xffffffff8000a0000000/ {  
    @ = count(); }'
```

The literal address of the `softc` comes from `devctl` or from inspecting the driver's `sysctl`. When multiple `myfirst` devices are present, this isolates one at a time.

23.15.2 Aggregations That Tell a Story

Aggregations are DTrace's statistical workhorses. Section 5 introduced `count()` and `quantize()`; other aggregation functions are equally useful:

- `sum(x)`: running total of `x`
- `avg(x)`: running average of `x`
- `min(x)`: minimum of `x` seen
- `max(x)`: maximum of `x` seen
- `lquantize(x, low, high, step)`: linear quantization with bounds
- `llquantize(x, factor, low, high, steps)`: logarithmic quantization with bounds

A typical per-function latency report uses `quantize` because it reveals distribution:

```
dtrace -n 'fbt::myfirst_read:entry { self->t = timestamp; }  
    fbt::myfirst_read:return /self->t/ {  
        @ = quantize((timestamp - self->t) / 1000);  
        self->t = 0; }'
```

Divide by 1000 to convert nanoseconds to microseconds, which makes the buckets meaningful. Add an @buf indexed aggregation to see latency by buffer size:

```
dtrace -n '  
fbt::myfirst_read:entry {  
    self->t = timestamp;  
    self->len = args[1]->uio_resid;  
}  
fbt::myfirst_read:return /self->t/ {  
    @["read", self->len] = quantize((timestamp - self->t) / 1000);  
    self->t = 0;  
}'
```

This aggregates separately for each distinct uio_resid, revealing whether large reads are disproportionately slow.

23.15.3 Thread-Local Variables

The self-> prefix introduces a thread-local variable: each thread sees its own copy. This is how Section 5's latency measurement works: the entry probe stores the timestamp in self->t, and the return probe reads it. Because both probes fire on the same thread, the variable is unambiguous.

Common uses:

```
self->start    /* time of entry for a specific function */  
self->args     /* saved arguments for use in the return probe */  
self->error    /* error code captured from return */
```

Thread-local variables are zeroed by default. Use them freely; their cost is one word of memory per live thread.

23.15.4 Arrays and Associative Access

DTrace arrays are sparse associative arrays, indexed by any expression. Aggregations use them implicitly (the @[key] syntax), but ordinary variables can use them too:

```
dtrace -n 'fbt::myfirst_read:entry {  
    counts[args[0], execname]++; }'
```

Though in practice, aggregations are preferable because they accumulate correctly across all CPUs.

23.15.5 Speculative Tracing

For a high-rate event where most cases are uninteresting, DTrace's speculative tracing buffers output until a decision is made:

```
dtrace -n '  
fbt::myfirst_read:entry {  
    self->spec = speculation();  
    speculate(self->spec);  
    printf("read entry: pid=%d", pid);  
}  
fbt::myfirst_read:return /self->spec && args[1] == 0/ {  
    speculate(self->spec);  
    printf("read return: bytes=%d", arg1);  
    commit(self->spec);  
    self->spec = 0;  
}  
fbt::myfirst_read:return /self->spec && args[1] != 0/ {  
    discard(self->spec);  
    self->spec = 0;  
}'
```

Only reads that return successfully (`args[1] == 0`) have their output committed; failed reads are discarded. This produces a quiet, focused log under heavy load.

Speculative tracing is one of DTrace's most effective features and is rarely needed for small drivers. But for a driver with thousands of events per second, it makes the difference between useful output and an unreadable flood.

23.15.6 Predicate Chains for Narrow Tracing

A common need is to trace one function call tree from a specific entry point. Predicates can chain using thread-local flags:

```
dtrace -n '  
syscall::ioctl:entry /pid == $target/ { self->trace = 1; }  
fbt::myfirst_*: /self->trace/ { printf("%s: %s", probefunc, probename); }  
syscall::ioctl:return /self->trace/ { self->trace = 0; }'
```

Run with `-p <pid>` to target a specific process. The trace flag is set on `syscall` entry and cleared on return, so only driver functions called during the `ioctl` are logged. This pattern is fundamental for isolating a specific user action.

23.15.7 Profiling the Kernel

The profile provider samples the kernel at a regular rate. Useful probes include:

- `profile-97`: samples every 97th of a second, on every CPU
- `profile-1001hz`: samples at 1001 Hz, slightly offset from any whole-second workload
- `profile-5ms`: samples every 5 ms

Each sample records the current stack. Aggregating by stack reveals where time is spent:

```
dtrace -n 'profile-1001hz /arg0/ { @[stack(8)] = count(); }
        tick-10s { exit(0); }'
```

The `arg0` predicate filters out idle CPUs. The `stack(8)` records the top 8 frames. The `tick-10s` ends the run after 10 seconds.

The output looks like:

```
kernel`myfirst_process_data+0x120
kernel`myfirst_intr+0x48
kernel`ithread_loop+0x180
kernel`fork_exit+0x7f
kernel`fork_trampoline+0xe
15823
```

15 823 samples at that stack, out of about 100 000 total (10 seconds * 1001 Hz * a few CPUs). The driver's interrupt path dominates. Combined with the latency measurement above, this is enough to justify a refactor.

23.15.8 Pretty-Printing Structures

A CTF-enabled kernel gives DTrace the types of every kernel structure. This lets `print()` show fields by name instead of raw memory:

```
dtrace -n 'fbt::myfirst_read:entry { print(*args[0]); }'
```

The output shows every field of the struct `myfirst_softc` the read is operating on, by field name, with human-readable values.

The feature depends on CTF having been generated (`makeoptions WITH_CTF=1`). Without CTF, DTrace falls back to address-based printing, which is much less readable.

23.15.9 Ring-Buffered Output

For very long DTrace runs, the default output fills up memory. The `-b` flag sets a ring buffer that discards older records when full:

```
dtrace -b 16m -n 'myfirst::: { trace(timestamp); }'
```

A 16 MB ring holds roughly the last 16 MB of trace output. Older data is overwritten as newer data arrives, which is typically fine for diagnosis: the interesting window is the last few seconds before a problem, not the previous hour.

23.15.10 Exiting Cleanly

DTrace accumulates aggregations in memory until the script exits. For a long run, the reader must exit cleanly to see the aggregations:

```
dtrace -n 'fbt::myfirst_*.entry { @[probefunc] = count(); }  
        tick-60s { printf("%Y\n", walltimestamp); printa(@);  
                  clear(@); }'
```

Every 60 seconds, the aggregate is printed and cleared. The output is a running report, rather than a single report at exit. This pattern is ideal for overnight runs: the output contains per-minute snapshots, and the reader can see how the driver's behaviour evolves.

23.15.11 Summary

These techniques together cover roughly 80% of the DTrace skills a driver author needs. The remaining 20% come from the DTrace Guide and from reading existing scripts in `/usr/share/dtrace/toolkit`.

Two rules of thumb guide advanced use:

1. **Start with the simplest script that might answer the question.** Add predicates only when the script produces too much output. Add aggregations only when counts and times matter.
2. **Validate DTrace's output against a known workload.** Run a simple test case first (say, one open, one read, one close), confirm the DTrace output matches, and only then trust the tool on a complex workload.

With these rules, DTrace is the most capable debugging tool in the FreeBSD arsenal, and the one that rewards every hour of study with months of saved debugging time.

23.16 Appendix: WITNESS Trace Interpretation

WITNESS is often a reader's first exposure to interpreting a kernel diagnostic trace. This appendix walks through the format of a WITNESS violation message and explains what each field means.

23.16.1 The General Format

A WITNESS violation looks like:

```
lock order reversal: (non-sleepable after sleepable)
 1st 0xfffff8000a000000 myfirst_lock (myfirst, sleep mutex) @ myfirst.c:123
 2nd 0xfffff8000a001000 myfirst_intr_lock (myfirst, spin mutex) @ myfirst.c:234
lock order myfirst_intr_lock -> myfirst_lock established at:
#0 0xfffffffff80abcdef at kdb_backtrace+0x1f
#1 0xfffffffff80abcdee at _witness_debugger+0x4f
#2 0xfffffffff80abcdaf at witness_checkorder+0x21f
#3 0xfffffffff80c00000 at _mtx_lock_flags+0x8f
#4 0xfffffffff80ffff00 at myfirst_intr+0x30
#5 ...
panic: witness
```

The message has three sections: the header line, the lock description section, and the backtrace of where the reversed order was first established.

23.16.2 The Header

```
lock order reversal: (non-sleepable after sleepable)
```

This says a spin (non-sleepable) mutex was acquired while a sleep (sleepable) mutex was already held. The reverse order (spin first, then sleep) is allowed; this order (sleep first, then spin) is not.

Common header variants:

- `lock order reversal: (sleepable after non-sleepable)`: means the declared order was later broken by an acquisition that goes the other way
- `spin lock recursion`: attempted to acquire a spin lock that the current thread already holds, which is not allowed except for recursive spin locks
- `spin lock held too long`: a spin lock was held for longer than the threshold (usually multiple seconds)
- `blocking on condition variable with spin lock`: tried to sleep while holding a spin lock

23.16.3 The Lock Descriptions

```
1st 0xfffff8000a000000 myfirst_lock (myfirst, sleep mutex) @ myfirst.c:123
2nd 0xfffff8000a001000 myfirst_intr_lock (myfirst, spin mutex) @ myfirst.c:234
```

- **0xfffff8000a000000**: the address of the mutex in memory
- **myfirst_lock**: the human-readable name passed to `mtx_init`
- **(myfirst, sleep mutex)**: the class (usually the driver name) and the type
- **@ myfirst.c:123**: the source location of the `mtx_init` call (when WITNESS is set up with `DEBUG_LOCKS`)

Both locks appear, first the one that was already held, then the one being acquired.

23.16.4 The Backtrace

The backtrace shows where the reversed order was first recorded. WITNESS remembers the first instance of each (A, B) pair it sees; later violations reference the earlier occurrence. If the first occurrence was valid at the time (for example, both locks were acquired correctly) but a later path reverses them, the backtrace points at the earlier, correct acquisition. The reader may need to search the source for the specific reversal path, which is usually the current stack trace (not shown in this snippet).

Two frames to look for:

1. The function that currently holds the first lock and is trying to acquire the second (or vice versa).
2. The function that first acquired the two locks in the original order.

Comparing the two reveals the conflict.

23.16.5 Reading a Real Example

A common example in new drivers:

```
lock order reversal: (non-sleepable after sleepable)
 1st 0xffff0 myfirst_lock (sleep mutex) @ myfirst.c:100
 2nd 0xffff1 ithread_lock (spin mutex) @ kern_intr.c:234
```

The new driver's code at `myfirst.c:100` acquires `myfirst_lock` (a sleep mutex). Somewhere deeper in the code path, an interrupt scheduling function tries to acquire `ithread_lock` (a spin mutex used by the interrupt subsystem). This is the sleep-then-spin violation, which is always a bug: a spin lock should never be acquired while a sleep lock is held.

The fix: rework the code so that the spin lock (or the subsystem that uses it) is not called while holding `myfirst_lock`. Common approaches include releasing `myfirst_lock` before the call, and then reacquiring it afterward if needed.

23.16.6 Asserting Lock Ownership

Independent of WITNESS, `mtx_assert(&lock, MA_OWNED)` at the top of each function that assumes a lock is held is a strong defensive coding pattern. When `INVARIANTS` is set, the assertion fires on any violation. Combined with WITNESS, these two checks catch the vast majority of lock-related bugs at their first occurrence, long before they corrupt production data.

23.16.7 When WITNESS Is Too Noisy

Occasionally WITNESS produces violations in code that is actually correct, usually because the lock acquisition is guarded by a runtime check that WITNESS cannot see. For those cases, `mtx_lock_flags(&lock, MTX_DUPOK)` or `MTX_NEW` tell WITNESS to allow the acquisition. Use these flags sparingly; most WITNESS violations are real.

23.17 Appendix: Building a Driver-Specific Debug Toolkit

The debug macros and SDT probes added to `myfirst` in Section 8 are a starting point. Over the course of a driver's life, the toolkit grows: more classes, more probes, custom sysctls, perhaps a debug-only ioctl that dumps internal state. This appendix outlines patterns for extending the toolkit.

23.17.1 Debug-Only Sysctls

A sysctl under `dev.myfirst.0.debug.*` is the natural home for runtime debug controls. Add nodes liberally:

- `dev.myfirst.0.debug.mask`: the class bitmask
- `dev.myfirst.0.debug.loglevel`: the syslog priority for `DPRINTF` output
- `dev.myfirst.0.debug.trace_pid`: a PID to focus tracing on
- `dev.myfirst.0.debug.count_io`: a counter of I/O events processed
- `dev.myfirst.0.debug.dump_state`: write-only, triggers a one-shot state dump

Each of these costs very little in size and is enormously useful during field debugging. The rule is: if a piece of state would be useful to inspect during a bug hunt, expose it via sysctl.

23.17.2 A Debug Ioctl

For state too complex or large to expose as a sysctl, a debug-only ioctl is a good choice. Define:

```
#define MYFIRST_IOC_DUMP_STATE _IOR('f', 100, struct myfirst_debug_state)

struct myfirst_debug_state {
    uint64_t    open_count;
    uint64_t    read_count;
    uint64_t    write_count;
    uint64_t    error_count;
    uint32_t    current_mask;
    /* ... */
};
```

The ioctl handler copies the current state into the user buffer. A small user program reads and prints it:

```
struct myfirst_debug_state s;
int fd = open("/dev/myfirst0", O_RDONLY);
ioctl(fd, MYFIRST_IOC_DUMP_STATE, &s);
printf("opens=%llu, reads=%llu, writes=%llu, errors=%llu\n",
       s.open_count, s.read_count, s.write_count, s.error_count);
```

A debug ioctl is especially useful when the state of interest is not a single number but a structure, and when sysctl's type constraints are awkward.

23.17.3 Counters for Key Events

The driver's `softc` gains a set of counters, one per significant event. The counters are simple `uint64_t` fields incremented with `atomic_add_64` or under the `softc` lock. They are exposed via `sysctl` or the `debug ioctl`, and they are cleared at the reader's request:

```
sc->sc_stats.opens++;      /* in open */
sc->sc_stats.reads++;      /* in read */
sc->sc_stats.errors++;     /* on any error path */
```

At ten counters per driver, the memory cost is under 100 bytes, and the visibility into the driver's behaviour is immense. Production monitoring systems can poll the counters periodically and alert on anomalies.

23.17.4 A Self-Test `ioctl`

For complex drivers, a self-test `ioctl` can be invaluable. It runs a sequence of internal tests (each one a function in the driver) and reports which passed and which failed. The results go back as a small structure. An operator debugging a field issue can run the self-test and immediately know whether any of the driver's subsystems are broken.

The self-test `ioctl` is not a substitute for unit testing or integration testing. It is a diagnostic shortcut for the field.

23.17.5 Integration with `rc` Scripts

A debug-aware `rc` script (in `/usr/local/etc/rc.d/myfirst_debug`) can set the debug mask at boot:

```
#!/bin/sh
# PROVIDE: myfirst_debug
# REQUIRE: myfirst
# KEYWORD: shutdown

. /etc/rc.subr

name="myfirst_debug"
rcvar="myfirst_debug_enable"
start_cmd="myfirst_debug_start"
stop_cmd="myfirst_debug_stop"

myfirst_debug_start()
{
    echo "Enabling myfirst debug mask"
    sysctl dev.myfirst.0.debug.mask=${myfirst_debug_mask:-0x0}
}
```

```
myfirst_debug_stop()
{
    echo "Disabling myfirst debug"
    sysctl dev.myfirst.0.debug.mask=0
}

load_rc_config $name
run_rc_command "$1"
```

An operator sets `myfirst_debug_enable=YES` and `myfirst_debug_mask=0x3` in `/etc/rc.conf`, reboots, and the debug infrastructure engages automatically. This pattern is how production systems manage debug verbosity across many devices.

23.17.6 Summary

The infrastructure in Section 8 is enough for the book's running example. Real drivers grow beyond it, acquiring a dozen or more debug-oriented features over time. The pattern is always the same: make instrumentation cheap when off, rich when on, and universally controllable by the operator. With that discipline, a driver remains diagnosable throughout its life.

23.18 Appendix: A Lab Reference for Chapter 23

This last appendix collects the labs into a table with time estimates and dependencies, to help the reader plan their session:

Lab	Topic	Est. time	Dependencies
23.1	First DDB session	15 min	Debug kernel with KDB/DDB
23.2	DTrace measurement of driver	20 min	KDTRACE_HOOKS, CTF, driver loaded
23.3	User-side ktrace of driver access	15 min	ktrace binary, driver loaded
23.4	Memory leak introduction and fix	30 min	Ability to rebuild the module
23.5	Installing the 1.6-debug refactor	60 min	Section 8 material, rebuild setup

Total focused time: about two hours and twenty minutes, plus time to absorb the material. The book's intent is that the reader does each lab at a pace that allows genuine understanding rather than rushing.

Optional follow-ups:

- **23.6 (optional).** Combine `ktrace` and `DTrace` on the same test program, and correlate the user-side trace with the driver-side events.

- **23.7 (optional).** Add a debug ioctl (as described in the Debug Toolkit appendix) and write a small user program that triggers it.

These two extensions solidify the chapter's themes and build practical muscles that every later chapter will exercise.

23.19 Appendix: Complete Annotated Listing of `myfirst_debug.h` and `myfirst_debug.c` Changes

To consolidate Section 8 into a single reference, this appendix collects the full text of the files as they appear after the 1.6-debug refactor, with inline comments explaining each block. A reader who prefers to see the finished code in one place can use this section; the example tree under `examples/part-05/ch23-debug/stage3-refactor/` contains the same text as downloadable files.

23.19.1 `myfirst_debug.h`

```
/*
 * myfirst_debug.h - debug and tracing infrastructure for the myfirst driver
 *
 * This header is included from the driver's source files. It provides:
 * - a bitmask of debug verbosity classes
 * - the DPRINTF macro for conditional device_printf
 * - declarations for SDT probes that the driver fires at key points
 *
 * The matching SDT_PROVIDER_DEFINE and SDT_PROBE_DEFINE calls live in the
 * driver source, which owns the storage for the probe entries.
 */

#ifndef _MYFIRST_DEBUG_H_
#define _MYFIRST_DEBUG_H_

#include <sys/param.h>
#include <sys/systm.h>
#include <sys/sdt.h>

/*
 * Debug verbosity classes. Each class is a single bit in sc->sc_debug.
 * The operator sets sysctl dev.myfirst.0.debug.mask to a combination of
 * these bits to enable the corresponding categories of output.
 *
 * Add new classes here when the driver grows new subsystems. Use the
 * next unused bit and update DEBUG.md accordingly.
 */
```

```

#define MYF_DBG_INIT      0x00000001  /* probe/attach/detach */
#define MYF_DBG_OPEN      0x00000002  /* open/close lifecycle */
#define MYF_DBG_IO        0x00000004  /* read/write paths */
#define MYF_DBG_IOCTL     0x00000008  /* ioctl handling */
#define MYF_DBG_INTR      0x00000010  /* interrupt handler */
#define MYF_DBG_DMA        0x00000020  /* DMA mapping/sync */
#define MYF_DBG_PWR       0x00000040  /* power-management events */
#define MYF_DBG_MEM       0x00000080  /* malloc/free trace */
/* Bits 0x0100..0x8000 reserved for future driver subsystems */

#define MYF_DBG_ANY       0xFFFFFFFF
#define MYF_DBG_NONE      0x00000000

/*
 * DPRINTF - conditionally log a message via device_printf when the
 * given class bit is set in the softc's debug mask.
 *
 * Usage: DPRINTF(sc, MYF_DBG_OPEN, "open: pid=%d\n", pid);
 *
 * When the bit is clear, the cost is one load and one branch, which
 * is negligible in practice. When the bit is set, the cost equals
 * a normal device_printf call.
 */
#ifdef _KERNEL
#define DPRINTF(sc, m, ...) do { \
    if ((sc)->sc_debug & (m)) \
        device_printf((sc)->sc_dev, __VA_ARGS__); \
} while (0)
#endif

/*
 * SDT probe declarations. The matching SDT_PROBE_DEFINE calls are in
 * myfirst_debug.c (or in the main driver source if preferred).
 *
 * Probe argument conventions:
 *   open (softc *, flags)          -- entry, before access check
 *   close (softc *, flags)         -- entry, before state update
 *   io   (softc *, is_write, resid, off) -- entry, into read or write
 */
SDT_PROVIDER_DECLARE(myfirst);
SDT_PROBE_DECLARE(myfirst, , , open);
SDT_PROBE_DECLARE(myfirst, , , close);
SDT_PROBE_DECLARE(myfirst, , , io);

```

```
#endif /* _MYFIRST_DEBUG_H_ */
```

The header has three sections: the class bitmask, the DPRINTF macro, and the SDT probe declarations. Each is small, self-contained, and designed to grow as the driver grows.

23.19.2 The SDT Definitions in `myfirst_debug.c`

```
/*
 * myfirst_debug.c - storage for the SDT probe entries.
 *
 * This file exists to hold the SDT_PROVIDER_DEFINE and SDT_PROBE_DEFINE
 * declarations. By convention in the myfirst driver, these live in a
 * dedicated source file to keep the main driver uncluttered.
 */

#include <sys/param.h>
#include <sys/system.h>
#include <sys/sdt.h>
#include "myfirst_debug.h"

/*
 * The provider "myfirst" exposes all of our static probes to DTrace.
 * Scripts select probes with "myfirst::".
 */
SDT_PROVIDER_DEFINE(myfirst);

/*
 * open: fired on every successful or attempted device open.
 *   arg0 = struct myfirst_softc *
 *   arg1 = int flags from the open call
 */
SDT_PROBE_DEFINE2(myfirst, , , open,
    "struct myfirst_softc *", "int");

/*
 * close: fired on every device close.
 *   arg0 = struct myfirst_softc *
 *   arg1 = int flags
 */
SDT_PROBE_DEFINE2(myfirst, , , close,
    "struct myfirst_softc *", "int");

/*
```

```

* io: fired on every read or write call, at function entry.
*  arg0 = struct myfirst_softc *
*  arg1 = int is_write (0 for read, 1 for write)
*  arg2 = size_t resid (bytes requested)
*  arg3 = off_t offset
*/
SDT_PROBE_DEFINE4(myfirst, , , io,
    "struct myfirst_softc *", "int", "size_t", "off_t");

```

The file is short, intentionally. Its sole purpose is to hold the storage for the probe entries. If the driver grows more probes, the reader adds them here.

23.19.3 Firing the Probes in the Driver

Inside `myfirst.c` (or whichever file implements each method), the probes fire at the appropriate call sites:

```

/*
 * myfirst_open: device open method.
 * Called when a user process opens /dev/myfirst0.
 */
static int
myfirst_open(struct cdev *dev, int flags, int devtype, struct thread *td)
{
    struct myfirst_softc *sc = dev->si_drv1;
    int error = 0;

    DPRINTF(sc, MYF_DBG_OPEN, "open: pid=%d uid=%d flags=0x%x\n",
        td->td_proc->p_pid, td->td_ucred->cr_uid, flags);

    /* Fire the SDT probe at entry, before any state change. */
    SDT_PROBE2(myfirst, , , open, sc, flags);

    mtx_lock(&sc->sc_mtx);
    if (sc->sc_open_count >= MYFIRST_MAX_OPENS) {
        error = EBUSY;
        goto out;
    }
    sc->sc_open_count++;
out:
    mtx_unlock(&sc->sc_mtx);

    if (error != 0)
        DPRINTF(sc, MYF_DBG_OPEN, "open failed: error=%d\n", error);
    else

```

```

        DPRINTF(sc, MYF_DBG_OPEN, "open ok: count=%d\n",
                sc->sc_open_count);

        return (error);
}

/*
 * myfirst_close: device close method.
 * Called when the last reference to the device is released.
 */
static int
myfirst_close(struct cdev *dev, int flags, int devtype, struct thread *td)
{
    struct myfirst_softc *sc = dev->si_drv1;

    DPRINTF(sc, MYF_DBG_OPEN, "close: pid=%d flags=0x%x\n",
            td->td_proc->p_pid, flags);

    SDT_PROBE2(myfirst, , , close, sc, flags);

    mtx_lock(&sc->sc_mtx);
    if (sc->sc_open_count > 0)
        sc->sc_open_count--;
    DPRINTF(sc, MYF_DBG_OPEN, "close ok: count=%d\n", sc->sc_open_count);
    mtx_unlock(&sc->sc_mtx);

    return (0);
}

/*
 * myfirst_read: device read method.
 */
static int
myfirst_read(struct cdev *dev, struct uio *uio, int ioflag)
{
    struct myfirst_softc *sc = dev->si_drv1;
    int error;

    DPRINTF(sc, MYF_DBG_IO, "read: pid=%d resid=%zu off=%jd\n",
            curthread->td_proc->p_pid,
            (size_t)uio->uio_resid, (intmax_t)uio->uio_offset);

    SDT_PROBE4(myfirst, , , io, sc, 0,
            (size_t)uio->uio_resid, uio->uio_offset);

```

```

        error = myfirst_read_impl(sc, uio);

        if (error != 0)
            DPRINTF(sc, MYF_DBG_IO, "read failed: error=%d\n", error);
        return (error);
}

/*
 * myfirst_write: device write method.
 */
static int
myfirst_write(struct cdev *dev, struct uio *uio, int ioflag)
{
    struct myfirst_softc *sc = dev->si_drv1;
    int error;

    DPRINTF(sc, MYF_DBG_IO, "write: pid=%d resid=%zu off=%jd\n",
        curthread->td_proc->p_pid,
        (size_t)uio->uio_resid, (intmax_t)uio->uio_offset);

    SDT_PROBE4(myfirst, , , io, sc, 1,
        (size_t)uio->uio_resid, uio->uio_offset);

    error = myfirst_write_impl(sc, uio);

    if (error != 0)
        DPRINTF(sc, MYF_DBG_IO, "write failed: error=%d\n", error);
    return (error);
}

```

Each method follows the entry-exit-error pattern:

1. A DPRINTF at entry, showing who called and with what arguments.
2. An SDT_PROBE after the entry log, before the actual work begins.
3. An error-path DPRINTF on any non-zero return.
4. A success-path DPRINTF showing the outcome when useful.

23.19.4 Registering the Sysctl in myfirst_attach

The sysctl that controls sc_debug is registered during device attach:

```

/*
 * Build the debug sysctl tree: dev.myfirst.N.debug.*

```

```

*/
sysctl_ctx_init(&sc->sc_sysctl_ctx);

sc->sc_sysctl_tree = SYSCTL_ADD_NODE(&sc->sc_sysctl_ctx,
    SYSCTL_CHILDREN(device_get_sysctl_tree(sc->sc_dev)),
    OID_AUTO, "debug",
    CTLFLAG_RW, 0, "debug and tracing controls");

SYSCTL_ADD_U32(&sc->sc_sysctl_ctx,
    SYSCTL_CHILDREN(sc->sc_sysctl_tree),
    OID_AUTO, "mask",
    CTLFLAG_RW, &sc->sc_debug, 0,
    "debug class bitmask (see myfirst_debug.h for class definitions)");

```

The detach path tears the sysctl tree down:

```
sysctl_ctx_free(&sc->sc_sysctl_ctx);
```

23.19.5 The Version Bump

At the top of the driver source, the version string and the `MODULE_VERSION` are updated:

```

static const char myfirst_version[] = "myfirst 1.6-debug";

/* ...driver methods, attach, detach, etc... */

MODULE_VERSION(myfirst, 16);

```

And near the top of `myfirst_attach`, the version is logged:

```
DPRINTF(sc, MYF_DBG_INIT, "attach: %s loaded\n", myfirst_version);
```

With the mask set to include `MYF_DBG_INIT`, the operator sees a clean version-reporting line on every attach.

23.19.6 A Sample dmesg With All Classes Enabled

When all classes are enabled and the device is exercised, `dmesg` shows a comprehensive log:

```

myfirst0: attach: myfirst 1.6-debug loaded
myfirst0: open: pid=1234 uid=1001 flags=0x0
myfirst0: open ok: count=1
myfirst0: read: pid=1234 resid=4096 off=0

```

```
myfirst0: close: pid=1234 flags=0x0
myfirst0: close ok: count=0
```

Every event the driver cares about is visible, tagged with the device name, the PID, and the relevant arguments. A field engineer receiving this log can reconstruct the sequence of events without needing source access.

With all classes disabled, `dmesg` shows only the attach line (if `MYF_DBG_INIT` is enabled) or nothing (if the mask is 0). The driver runs at full speed with no observability cost.

23.19.7 A Sample DTrace Session With the Probes

The SDT probes allow machine-readable analysis:

```
$ sudo dtrace -n 'myfirst::: { @[probename] = count(); } tick-10s { exit(0); }'
dtrace: description 'myfirst::: ' matched 3 probes
CPU      ID                FUNCTION:NAME
  0  49012                :tick-10s

close                                43
open                                43
io                                  120
```

Over the 10 seconds of sampling, the driver processed 43 open/close cycles and 120 I/O events. The aggregate is generated without any modification to the driver and at essentially no runtime cost, because no script attached means the probes are inert.

23.19.8 Summary of the Refactor

The 1.6-debug refactor adds roughly 100 lines of code distributed as:

- `myfirst_debug.h`: 50 lines (classes, `DPRINTF`, SDT declarations)
- `myfirst_debug.c`: 30 lines (SDT provider and probe definitions)
- Driver source changes: 20 lines (`sysctl` registration, `DPRINTF` calls, probe firings)

The investment is small; the return is enormous. Every subsequent chapter of the book inherits this infrastructure without additional cost, and every field issue in the driver's future gains a clear diagnostic path.

23.20 Appendix: What the Kernel Does When a Driver Panics

A reader who has followed the chapter so far understands the debugging tools well. The final appendix covers what happens when something goes catastrophically wrong: a panic. The behaviour of the kernel during a panic is not optional for a driver author to understand, because any bug severe enough to panic produces a specific sequence of events that the driver must be correct about.

23.20.1 The Panic Sequence

When any kernel code calls `panic()`, the following happens, in order:

1. The panic message is printed to the console, typically via `printf`.
2. The CPU raises its priority to the maximum, stopping most ordinary kernel activity.
3. If the kernel was built with options `KDB`, and `kdb_unattended` is not set, control transfers to the kernel debugger. The operator sees the `db>` prompt.
4. If the operator (or `kdb_unattended=1`) continues the panic, the kernel initiates a crash dump. The dump writes the current memory image to the crash-dump device (typically `swap`).
5. The kernel reboots (unless `panic_reboot_wait_time` is set).

On the next boot, `savecore(8)` extracts the dump from the swap device, writes it to `/var/crash/vmcore.N`, and rennumbers the counter. The reader can then run `kgdb` against the dump to inspect post-mortem state.

23.20.2 Driver Responsibilities

The panic path places specific requirements on drivers:

1. **Do not panic during detach.** A driver that panics during `detach` prevents clean shutdown of the rest of the kernel. Every `detach` path should be panic-free.
2. **Do not allocate memory in the panic handler.** If the driver registers a shutdown hook that runs during panic, the hook must not call `malloc` or any function that may sleep.
3. **Do not rely on user space during panic.** User programs are frozen during panic; the driver cannot send them messages.
4. **Do not wait for interrupts during panic.** Interrupts may be disabled during the panic path; a driver that waits for one will hang.

These rules apply to shutdown hooks and to the rare paths the kernel traverses during panic. In normal operation, drivers do not run in the panic path, but certain callbacks do.

23.20.3 Reading a Crash Dump

The crash dump is a full memory image of the kernel at the moment of panic. `kgdb` opens it with full symbolic information if the kernel has debug info:

```
sudo kgdb /boot/kernel/kernel /var/crash/vmcore.0
```

Inside `kgdb`, the most useful commands are:

```
(kgdb) bt           # backtrace of the thread that panicked
(kgdb) info threads  # list all threads
(kgdb) thread N      # switch to thread N
(kgdb) frame N       # show frame N (in the current backtrace)
```

```
(kgdb) list          # show source at the current frame
(kgdb) info locals   # show local variables
(kgdb) print var     # show a variable
(kgdb) print *sc     # show the softc pointer contents
(kgdb) ptype struct myfirst_softc # show the structure type
```

Common investigation steps:

1. `bt` to see what panicked.
2. `list` to see the source line.
3. `info locals` to see the local state.
4. `print` fields of interest.
5. `thread apply all bt` to see all threads, in case the panic was triggered by one thread's interaction with another.

23.20.4 Combining Kernel and Driver Symbols

If the driver is built as a module, `kgdb` needs to load its symbols too:

```
(kgdb) add-symbol-file /boot/kernel/myfirst.ko <address>
```

The `<address>` is the load address of the module, which `kldstat -v` shows in the running system, or which appears in the panic output. Once loaded, `kgdb` can decode frames inside the module with full source visibility.

23.20.5 Crash Dump Size

A full crash dump is as large as the kernel's memory usage at the time of the panic, typically gigabytes. The dump device (usually swap) must have enough space. Systems tuned for rapid panic recovery often use a dedicated dump device, not swap, to avoid swap-space constraints.

For drivers-only debugging, a mini dump captures only the kernel's own memory, not user memory. Set `dumpdev=none` or `dumpdev=mini` in `loader.conf` to tune this. Mini dumps are small (tens of megabytes) and load quickly into `kgdb`.

23.20.6 Summary

The panic path is rare in production but essential to understand. A driver that handles its errors cleanly rarely triggers a panic; a driver that does panic produces a crash dump the author can analyse with `kgdb`. The combination of `INVAARIANTS`, `WITNESS`, `DEBUG_MEMGUARD`, and a functioning crash-dump pipeline makes panics into valuable diagnostic events rather than catastrophes.

23.21 Appendix: Chapter 23 at a Glance

This single-page summary is intended as a recap the reader can keep open while they work. Each entry names a concept, points to where it was introduced, and gives the minimal command or code to exercise it.

23.21.1 Tools in order of first use

1. **printf / device_printf** (Section 2): Basic kernel logging. Use `device_printf(dev, "msg\n")` inside driver methods.
2. **log(LOG_PRIORITY, ...)** (Section 2): Syslog-routed logging, with a priority level. Use for messages that belong in `/var/log/messages`.
3. **dmesg** (Section 3): Read the kernel message buffer. Combine with `grep` and `tail` for filtered output.
4. **/var/log/messages** (Section 3): Persistent log history. `syslogd(8)` writes to it; `newsyslog(8)` rotates it.
5. **Debug kernel** (Section 4): Rebuild with `KDB`, `DDB`, `INVARIANTS`, `WITNESS`, `KDTRACE_HOOKS`, and `DEBUG=-g`. Slower but diagnostic.
6. **ddb** (Section 4): Interactive kernel debugger. Enter via `sysctl debug.kdb.enter=1` or a console break. Use `bt`, `show locks`, `ps`, `continue`.
7. **kgdb** (Section 4): Post-mortem debugger. Open `/boot/kernel/kernel` and `/var/crash/vmcore.N` together.
8. **DTrace** (Section 5): Live tracing and measurement. `dtrace -n 'fbt::myfirst_*:entry { @[probe-func] = count(); }'`.
9. **SDT probes** (Section 5, Section 8): Static trace points in the driver source. `dtrace -n 'myfirst::: { @[probename] = count(); }'`.
10. **ktrace/kdump** (Section 6): User-side syscall tracing. `sudo ktrace -di -f trace.out ./program; kdump -f trace.out`.
11. **vmstat -m** (Section 7.1): Per-pool kernel memory view. Used to detect leaks.

23.21.2 Reflexes to build

- Add a `device_printf` before you suspect a bug, not after.
- Check `dmesg` before guessing about driver behavior.
- Rebuild with `INVARIANTS` and `WITNESS` during development.
- Write DTrace one-liners instead of compile-reload cycles when the question is “does this fire?”.
- Maintain `vmstat -m` stability as a continuous invariant, not an after-the-fact check.
- Never hold a sleep lock across `uiomove` or any sleeping call.
- Make interrupt handlers short. Push work to `taskqueue`.
- Match every `malloc` with a `free`. Test the match with the tools.

23.21.3 The 1.6-debug refactor

- Adds `myfirst_debug.h` with 8 verbosity classes and the `DPRINTF` macro.
- Adds a `sysctl dev.myfirst.0.debug.mask` for runtime control.
- Adds three SDT probes: `myfirst:::open`, `myfirst:::close`, `myfirst:::io`.
- Follows the `entry/exit/error` pattern across every method.
- Bumps `myfirst_version` to 1.6-debug and `MODULE_VERSION` to 16.

23.21.4 Recommended next steps

- Complete all five labs if time allows.

- Attempt Challenge 23.2, the custom DTrace script.
- Keep a personal lab logbook entry for this chapter.
- Read `/usr/src/sys/dev/ath/if_ath_debug.h` as a real-world reference.
- Before starting Chapter 24, confirm the driver is at version 1.6-debug and the debug mask is responsive.

This concludes the chapter's reference material. The "Wrapping Up" and bridge sections that follow prepare the reader for Chapter 24.

23.22 Wrapping Up

Chapter 23 introduced the working methods and tools of FreeBSD kernel debugging. The chapter's structure followed a deliberate arc: explain why kernel debugging is different, teach the core logging and inspection tools, build a debug kernel, master DTrace and ktrace, recognise the common bug classes, and finally refactor the `myfirst` driver to support its own observability.

Along the way the chapter has earned its place in the book. It sits between the architectural work of Part 4 and the integration work of Part 5 because the remaining chapters will add substantial new code to `myfirst` (devfs hooks, ioctl, sysctl, network interfaces, GEOM interfaces, USB support, virtio support) and each of those chapters depends on the ability to see what the driver is doing and to fix problems as they arise. The tooling and instrumentation are not an afterthought; they are how the rest of the book stays manageable.

The driver itself is now at version 1.6-debug. Its softc carries an `sc_debug` mask; the tree has a `sysctl dev.myfirst.0.debug` subtree; the `DPRINTF` macro gates logging by class; and three SDT probes expose the open, close, and I/O events. Every later chapter in Parts 5, 6, and 7 will extend the same framework: new classes will appear in `myfirst_debug.h`, and new probes will be declared as the driver acquires new functional boundaries.

23.22.1 What the reader can now do

- Use `device_printf`, `log`, and the `DPRINTF` macro pattern to log structured kernel events.
- Read `dmesg` and `/var/log/messages` to locate driver messages and correlate them with events.
- Build a debug kernel with `KDB`, `DDB`, `INVARIANTS`, `WITNESS`, and `KDTRACE_HOOKS`.
- Write DTrace one-liners that count, aggregate, and time kernel events.
- Use `ktrace` and `kdump` to observe user-side interactions with the driver.
- Recognise the most common driver bugs and match them to the right debug tool.
- Declare SDT probes and expose them to DTrace scripts.
- Bump the driver's version and carry the debug infrastructure forward.

23.22.2 What lies ahead

With the debug infrastructure in place, Chapter 24 (Integrating with the Kernel: devfs, ioctl, and sysctl) will extend the driver's interface to user space. The reader will create richer `cdev` entries, define custom ioctls, and register tuning sysctls for the driver's runtime parameters. Every one of those additions can now be instrumented and traced with the framework from this chapter, so the work will be grounded in visible, debuggable behaviour from the first line.

The debug arsenal built in Chapter 23 will see heavy use from here on. In Chapter 24 the reader will add their first real ioctl, and the first time an argument arrives with the wrong value, they will turn on MYF_DBG_IOCTL, read the log, and find the mistake in seconds rather than hours. In Chapter 25 (Writing Character Drivers in Depth), the reader will use DTrace to confirm that the driver's select/poll path is firing correctly. In Chapter 26 and beyond, each new subsystem hook will be logged through DPRINTF and probed with SDT. The debug chapter's return on investment is long and steady.

A final encouragement: the tools of this chapter are not glamorous. They do not produce working drivers on their own. But they are what turns a frustrating six-hour bug into a crisp twenty-minute diagnosis. Every experienced FreeBSD developer has, on at least a dozen occasions, traced a problem through `vmstat -m`, then DTrace, then the source code, and emerged with a precise fix. The reader who masters the toolbox of this chapter joins that group and is now ready for the richer integration work that follows.

23.23 Bridge to Chapter 24

Chapter 24 (Integrating with the Kernel: devfs, ioctl, and sysctl) opens the next arc of the book. Up to Chapter 23 the driver has been a single cdev node with fixed behaviour. Chapter 24 makes the interface expressive: readers will extend the cdev into richer devfs entries, define custom ioctls to configure the device, and register sysctl nodes for runtime parameters.

The bridge is direct. The debug mask introduced in Section 8 is a sysctl. The ioctls in Chapter 24 will be traced through MYF_DBG_IOCTL. The custom devfs nodes will log their creation and destruction through MYF_DBG_INIT. When the reader builds new features in Chapter 24, the debug framework is ready to observe them.

See you in Chapter 24.

23.24 Reference: Chapter 23 Quick-Reference Card

The table below summarises the tools, commands, and patterns introduced in Chapter 23 for quick lookup.

23.24.1 Logging

Call	Use
<code>printf(...)</code>	Very basic logging; no device identity.
<code>device_printf(dev, "...")</code>	Standard driver log; prefixes with device name.
<code>log(LOG_PRI, "...")</code>	Syslog with priority; sent to <code>/var/log/messages</code> .
<code>DPRINTF(sc, CLASS, "...")</code>	Conditional on <code>sc->sc_debug & CLASS</code> ; standard driver macro.

23.24.2 Syslog priority levels

Priority	Meaning
LOG_EMERG	System is unusable.
LOG_ALERT	Action must be taken immediately.
LOG_CRIT	Critical conditions.
LOG_ERR	Error conditions.
LOG_WARNING	Warning conditions.
LOG_NOTICE	Normal but significant condition.
LOG_INFO	Informational.
LOG_DEBUG	Debug-level messages.

23.24.3 Kernel options for debugging

Option	Effect
options KDB	Includes the kernel debugger framework.
options DDB	Interactive debugger UI on console.
options DDB_CTF	CTF support in DDB for type-aware printing.
options KDB_TRACE	Automatic backtrace on entry to KDB.
options KDB_UNATTENDED	Panics reboot instead of dropping into DDB.
options INVARIANTS	Enables KASSERT, MPASS, and other kernel assertions.
options WITNESS	Tracks lock order and panics on violations.
options WITNESS_KDB	Drops to KDB on a WITNESS violation.
options DEBUG_MEMGUARD	Poisons freed memory; catches use-after-free.
options KDTRACE_HOOKS	Enables DTrace probes.
options KDTRACE_FRAME	Produces frame pointers for DTrace stack walks.
makeoptions WITH_CTF=1	Generates CTF in the kernel and modules.
makeoptions DEBUG=-g	Includes full DWARF in the kernel and modules.

23.24.4 Common DTrace one-liners

```
# Count every function entry in the driver
dtrace -n 'fbt::myfirst_*.entry { @[probfunc] = count(); }'

# Measure the time spent in a specific function
dtrace -n 'fbt::myfirst_read.entry { self->t = timestamp; }
```

```

        fbt::myfirst_read:return /self->t/ {
            @ = quantize(timestamp - self->t); self->t = 0; }'

# Count SDT probes fired by the driver
dtrace -n 'myfirst::: { @[probename] = count(); }'

# Show syscall frequency for a named process
dtrace -n 'syscall:::entry /execname == "myftest"/ { @[probefunc] = count(); }'

# Aggregate I/O sizes through the driver
dtrace -n 'myfirst:::io { @ = quantize(arg2); }'

```

23.24.5 ktrace / kdump workflow

```

# Record a process under ktrace
sudo ktrace -di -f trace.out ./myprogram

# Dump in human-readable form
kdump -f trace.out

# Filter for syscalls only
kdump -f trace.out | grep -E 'CALL|RET|NAMI'

```

23.24.6 Debugging checklist (abbreviated)

1. Is dmesg clean? If not, read the last message before the problem.
2. Is vmstat -m stable for the driver's pool? If not, look for a leak.
3. Does WITNESS panic on the workload? If yes, the lock order is wrong.
4. Do the fbt::myfirst_*:entry counts match the expected activity? If not, check the paths.
5. Does the user-side ktrace show the expected syscall sequence? If not, check the ioctl or command setup.
6. Is the driver at the expected version? kldstat -v | grep myfirst and confirm MODULE_VERSION.

23.25 Reference: Glossary of Chapter 23 Terms

CTF: Compact C Type Format. A kernel type-information format that DDB and DTrace use to print typed structures. Requires makeoptions WITH_CTF=1.

DDB: The built-in FreeBSD kernel debugger. Interactive, console-based. Activated by options DDB.

DDB_CTF: A DDB option that lets the debugger print typed values using CTF type information.

device_printf: The FreeBSD-standard logging function for driver messages. Prefixes output with the device's name.

DPRINTF: The conventional driver macro for conditional logging, gated on a verbosity mask.

DTrace: A dynamic tracing framework. Uses `fbt`, `syscall`, `sdt`, `io`, `sched`, `lockstat`, and other providers.

fbt: Function Boundary Tracing. A DTrace provider that traces entry and return of every kernel function.

INVARIANTS: A kernel build option that enables `KASSERT`, `MPASS`, and related assertions. Standard for debug builds.

KASSERT: An assertion macro that evaluates only when `INVARIANTS` is set. Panics if the condition is false.

KDB: The kernel debugger framework. `DDB` is one backend.

KDTRACE_HOOKS: The kernel build option that enables DTrace probes.

kdump: The user-space tool that reads `kttrace` output files and prints them in human-readable form.

kttrace: The user-space tool that records syscalls and other events of a given process.

log: The kernel function for sending messages to `syslog` with a priority level.

MPASS: Similar to `KASSERT` but with a compile-time-only check. Zero cost in release builds.

myfirst_debug.h: The debug header introduced in Section 8, declaring verbosity classes and SDT probe structures.

sc_debug: The `uint32_t` field in the driver's `softc` that controls which `DPRINTF` categories are active.

SDT: Statically Defined Tracing. Compile-time probes that DTrace can attach to.

SDT_PROBE_DEFINE: Macro family that registers a new SDT probe in the kernel or driver.

syslog: The BSD logging subsystem; message priorities map to destinations per `/etc/syslog.conf`.

vmstat -m: A user-space tool that shows per-pool kernel memory statistics.

WITNESS: A kernel lock-order verification system. Panics on lock-order violations when options `WITNESS` is set.

1.6-debug: The driver version after Section 8. Carries the verbosity mask, `sysctl`, `DPRINTF` pattern, and SDT probes.

24 Integrating with the Kernel

24.1 Reader Guidance & Outcomes

Chapter 23 closed with a driver that can finally explain itself. The `myfirst` driver at version 1.6-debug knows how to log structured messages through `device_printf`, how to gate verbose output through a runtime `sysctl` mask, how to expose static probe points to DTrace, and how to leave a paper trail an operator can read back later. Combined with the power discipline added in Chapter 22, the DMA pipeline added in Chapter 21, and the interrupt machinery added in Chapter 19 and Chapter 20, the driver is now a complete unit in itself: it boots, it runs, it talks to a real PCI device, it survives suspend and resume, and it tells the developer what it is doing along the way.

What the driver does not yet do is behave like a citizen of the wider kernel. There is still very little of `myfirst` that an outside program can see, control, or measure. The driver creates exactly one device node when the module loads. There is no way for a user-space tool to ask the driver to reset itself, to flip a configuration knob at runtime, or to read out a counter. There are no metrics in the system's `sysctl` tree that an operator could feed to a monitoring system. There is no way to give the driver several instances cleanly. There is no integration with the network stack, with the storage stack, or with any of the kernel subsystems users routinely reach for from their own programs. In every meaningful sense, the driver is still standing alone in the corner. Chapter 24 is the chapter that brings it into the room.

Chapter 24 teaches kernel integration at the right level for this stage of the book. The reader will spend this chapter learning what integration actually means, why it matters more than it looks at first, and how each integration surface is built. The chapter starts with the conceptual story: the difference between a working driver and an integrated driver, and the cost of leaving integration as an afterthought. It then spends most of its time on the four interfaces a typical FreeBSD driver always integrates with: the device file system `devfs`, the user-controlled `ioctl(2)` channel, the system-wide `sysctl(8)` tree, and the kernel's lifecycle hooks for clean attach, detach, and module unload. After those four come two optional chapters in miniature, one for drivers whose hardware is a network device and one for drivers whose hardware is a storage controller. Both are introduced at the conceptual level so the reader recognises them when they appear later in Part 6 and Part 7, and neither is taught in full because each will eventually deserve its own chapter. The chapter then steps back to discuss the discipline of registration and teardown across all of these surfaces: the order matters, the failure paths matter, the corner cases under stress matter, and a driver that gets the integration interfaces right but the lifecycle wrong is still a fragile driver. Finally the chapter closes with a refactor that splits the new code into its own files, bumps the driver to version 1.7-integration, updates the version banner, and leaves the source tree organised for everything that follows.

The arc of Part 5 continues here. Chapter 22 made the driver survive a change of power state. Chapter 23 made the driver tell you what it is doing. Chapter 24 makes the driver fit naturally into the rest of the system, so that the tools and habits FreeBSD users already know carry over to your driver without surprises. Chapter 25 will continue the arc by teaching the maintenance discipline that keeps a driver readable, tunable, and extensible as

it evolves, and Part 6 will then begin the transport-specific chapters that lean on every quality the Part 5 chapters have built.

24.1.1 Why `devfs`, `ioctl`, and `sysctl` Integration Earn a Chapter of Their Own

One concern that surfaces here is whether wiring up `devfs`, `ioctl`, and `sysctl` really deserves a full chapter. The driver already has a single `cdev` node from a much earlier chapter. Adding an `ioctl` looks small. Adding a `sysctl` looks even smaller. Why spread the work over a long chapter, when each interface looks like a few dozen lines of code?

The answer is that the few-dozen-lines view is the easy part. Each interface has a set of conventions and pitfalls that are not obvious from reading the API once, and the cost of getting them wrong is paid not by the developer but by the operator who tries to monitor the driver, the user who tries to reset the device, the packager who tries to load and unload the module under load, and the next developer who tries to extend the driver six months later. Chapter 24 spends its time on those conventions and pitfalls because that is where the value is.

The first reason this chapter earns its place is that **the integration interfaces are how everything else reaches the driver**. A reader who has followed the book this far has built a driver that does interesting work, but only the kernel itself currently knows how to ask the driver to do that work. Once the driver has an `ioctl` interface, a shell script can drive it. Once the driver has a `sysctl` tree, a monitoring system can watch it. Once the driver creates `/dev` nodes that follow the standard conventions, a packager can ship `udev`-style rules, the system administrator can write `/etc/devfs.rules` for it, and another driver can layer on top of it through `vop_open` or through `ifnet`. None of that depends on what the driver is for; all of it depends on whether the integration is done well.

The second reason is that **integration choices show up in production failure modes**. A driver that calls `make_dev` from the wrong context can deadlock at module load. A driver that omits the `_IO`, `_IOR`, `_IOW`, `_IOWR` discipline forces every caller to invent a private convention for who copies what across the user-kernel boundary, and at least one of those callers will get it wrong. A driver that forgets to call `sysctl_ctx_free` on detach leaks OIDs, and the next module load that uses the same name fails with a confusing message. A driver that destroys its `cdev` before draining its open file handles produces use-after-free panics. Chapter 24 spends paragraphs on each of these because each is a real bug the FreeBSD community has had to chase down across the years, and the right time to learn the discipline is before writing the first line of integration code, not after.

The third reason is that **the integration code is the first place a driver's design becomes visible to someone other than its author**. Until Chapter 24 the driver was a black box with one method table and one device node. From Chapter 24 onward, the driver has a public surface. The names of its `sysctls` show up in monitoring graphs. The numbers of its `ioctls` show up in shell scripts and in user-space libraries. The layout of its `/dev` nodes shows up in package documentation and in administrator runbooks. Once a public surface exists, changing it has a cost. The chapter therefore takes care to teach the conventions that keep the surface stable as the driver evolves. The version bump to 1.7-integration is also the first version of the driver that has a real public face; everything before it was an internal milestone.

Chapter 24 earns its place by teaching those three ideas together, concretely, with the `myfirst` driver as the running example. A reader who finishes Chapter 24 can integrate any FreeBSD driver into the standard system interfaces, knows the conventions and pitfalls of each integration surface, can read another driver's integration code and identify what is normal and what is unusual, and has a `myfirst` driver that other software can finally talk to.

24.1.2 Where Chapter 23 Left the Driver

A short checkpoint before the real work starts. Chapter 24 extends the driver produced at the end of Chapter 23, tagged as version 1.6-debug. If any of the items below is uncertain, return to Chapter 23 and fix it before starting this chapter, because the integration topics assume the debug discipline already exists, and several of the new integration surfaces will use it.

- Your driver compiles cleanly and identifies itself as 1.6-debug in `kldstat -v`.
- The driver still does everything it did at 1.5-power: it attaches to a PCI (or simulated PCI) device, allocates MSI-X vectors, runs a DMA pipeline, and survives `devctl suspend myfirst0` followed by `devctl resume myfirst0`.
- The driver has a `myfirst_debug.c` and `myfirst_debug.h` pair on disk. The header defines `MYF_DBG_INIT`, `MYF_DBG_OPEN`, `MYF_DBG_IO`, `MYF_DBG_IOCTL`, `MYF_DBG_INTR`, `MYF_DBG_DMA`, `MYF_DBG_PWR`, and `MYF_DBG_MEM`. The `DPRINTF(sc, MASK, fmt, ...)` macro is in scope from any source file in the driver.
- The driver has three SDT probes named `myfirst:::open`, `myfirst:::close`, and `myfirst:::io`. The simple DTrace one-liner `dtrace -n 'myfirst::: { @[probenam] = count(); }'` returns counts when the device is exercised.
- The softc carries a `uint32_t sc_debug` field, and `sysctl dev.myfirst.0.debug.mask` reads and writes it.
- The driver has a `DEBUG.md` document beside the source. `HARDWARE.md`, `LOCKING.md`, `SIMULATION.md`, `PCI.md`, `INTERRUPTS.md`, `MSIX.md`, `DMA.md`, and `POWER.md` are also up to date in your working tree from earlier chapters.
- Your test kernel still has `INVARIANTS`, `WITNESS`, `WITNESS_SKIPSPIN`, `DDB`, `KDB`, `KDB_UNATTENDED`, `KDTRACE_HOOKS`, and `DDB_CTF` enabled. Chapter 24's labs use the same kernel.

That driver is what Chapter 24 extends. The additions are bigger than in Chapter 23 in code lines, but smaller in conceptual surface. The new pieces are: a richer `/dev/myfirst0` node that can clone instances on demand, a small set of well-typed ioctls with a public header that user-space programs can include, a per-instance `sysctl` subtree under `dev.myfirst.N` exposing a handful of metrics and one writable knob, a refactor that splits the new code into `myfirst_ioctl.c` and `myfirst_sysctl.c` with matching headers, a small companion user-space program named `myfirstctl` that exercises the new interfaces, an `INTEGRATION.md` document beside the source, an updated regression test, and a version bump to 1.7-integration.

24.1.3 What You Will Learn

Once you have finished this chapter, you will be able to:

- Explain what kernel integration means in concrete FreeBSD terms, distinguish a self-contained driver from an integrated one, and name the specific user-visible benefits each integration surface delivers.
- Describe what `devfs` is, how it differs from older static `/dev` schemes, and how device nodes come into and out of existence under it. Use `make_dev`, `make_dev_s`, `make_dev_credf`, and `destroy_dev` correctly. Choose the right flag set, ownership, and mode for a node.
- Initialise and populate a `struct cdevsw` with the modern `D_VERSION` field, the minimal callback set, and the optional callbacks (`d_kqfilter`, `d_mmap_single`, `d_purge`).

- Use the `cdev->si_drv1` and `si_drv2` fields to attach per-node driver state, and read that state back from inside the `cdevsw` callbacks.
- Create more than one device node from a single driver instance and choose between fixed-name nodes, indexed nodes, and clonable nodes through the `dev_clone` event handler.
- Set per-node permissions and ownership at creation time, and adjust them after creation through `devfs.rules` so administrators can grant access without rebuilding the driver.
- Explain what `ioctl(2)` is, how the kernel encodes `ioctl` commands using `_IO`, `_IOR`, `_IOW`, and `_IOWR`, what each macro means about the direction of data flow, and why getting the encoding right matters for portability between 32-bit and 64-bit user space.
- Define a public `ioctl` header for a driver, choose a free magic letter, and document each command so that user-space callers can rely on the interface across releases.
- Implement a `d_ioctl` handler that dispatches on the command word, performs the per-command logic safely, and returns the right `errno` on each failure path.
- Read and understand the kernel's automatic copyin/copyout layer for `ioctl` data and recognise the cases where the driver still has to copy memory itself: variable-length payloads, embedded user pointers, and structures whose layout requires explicit alignment.
- Explain `sysctl(9)`, distinguish static OIDs from dynamic OIDs, and walk through the `device_get_sysctl_ctx` and `device_get_sysctl_tree` pattern that gives every device its own subtree.
- Add read-only counters using `SYSCTL_ADD_UINT` and `SYSCTL_ADD_QUAD`, add writable knobs with appropriate access flags, and add custom procedural OIDs with `SYSCTL_ADD_PROC` for cases where the value has to be computed at read time.
- Manage tunables that a user can set in `/boot/loader.conf` with `TUNABLE_INT_FETCH`, and combine tunables and sysctls so the same configuration knob can be set at boot or adjusted at runtime.
- Recognise the introductory shape of FreeBSD's networking integration: how `if_alloc`, `if_initname`, `if_attach`, `bpfattach`, `if_detach`, and `if_free` compose; what an `if_t` is at the conceptual level; and what role drivers play in the larger `ifnet` machinery. Understand that Chapter 28 returns to this in depth.
- Recognise the introductory shape of FreeBSD's storage integration: what CAM is, how `cam_sim_alloc`, `xpt_bus_register`, the `sim_action` callback, and `xpt_done` compose; what a CCB is conceptually; and why CAM exists at all. Understand that Chapter 27's storage drivers and Chapter 27's GEOM material return to this in depth.
- Apply a registration and teardown discipline that is robust under repeated `kldload/kldunload`, under attach failure midway through `bringup`, under detach failure when in-flight users still hold open file descriptors, and under genuine surprise removal of the underlying device.
- Refactor a driver that has accumulated several integration surfaces into a maintainable structure: a separate file per integration concern, a public header for user space, a private header for in-driver use, and an updated build system that compiles all of the pieces into a single kernel module.

The list is long because integration touches several subsystems. Each item is narrow and teachable. The chapter's work is putting them together into a single coherent driver.

24.1.4 What This Chapter Does Not Cover

Several adjacent topics are explicitly deferred so Chapter 24 stays focused on integration discipline.

- **The full implementation of an ifnet network driver**, including transmit and receive queues, multi-queue coordination through `iflib(9)`, BPF integration beyond the introductory `bpfattach` call, link-state events, and the full Ethernet driver lifecycle. Chapter 28 is the dedicated network driver chapter and assumes Chapter 24's integration discipline is already in place.
- **The full implementation of a CAM storage driver**, including target mode, the full CCB type set, asynchronous notifications through `xpt_setup_ccb` and `xpt_async`, and the geometry presentation through `disk_create` or `GEOM`. Chapter 27 covers the storage stack in depth.
- **GEOM integration**, including providers, consumers, classes, `g_attach`, `g_detach`, and the GEOM event machinery. GEOM is its own subsystem with its own conventions; Chapter 27 covers it.
- **epoch(9)-based concurrency**, which is the modern locking pattern for ifnet hot paths. Chapter 24 mentions it only in context. Chapter 28 (network drivers) returns to it alongside `iflib(9)` where epoch-style concurrency is needed in practice.
- **mac(9) (Mandatory Access Control) integration**, which adds policy hooks around the integration surfaces. The MAC framework is a specialised topic that does not yet apply to the simple `myfirst` driver.
- **vfs(9) integration**, which is what file systems do. A character driver does not interact with VFS at the layer of `vop_open` or `vop_read`; it interacts with `cdevsw` and `devfs`. The chapter is careful not to confuse the two.
- **Cross-driver interfaces through kobj(9) and custom interfaces declared with the INTERFACE build mechanism**. These are how the network and storage stacks define their internal contracts. They are mentioned in context in Section 7 but the deep treatment belongs in a later, more advanced chapter.
- **The new netlink(9) interface that recent FreeBSD kernels expose for some network management traffic**. Netlink is currently used by the routing subsystem rather than by individual device drivers, and the right place to teach it is alongside the network chapter.
- **Custom protocol modules through pr_protocol_init**, which is for new transport protocols rather than for device drivers.

Staying inside those lines keeps Chapter 24 a chapter about how a driver becomes part of the kernel, not a chapter about every kernel subsystem a driver may eventually touch.

24.1.5 Estimated Time Investment

- **Reading only**: four to five hours. Chapter 24's ideas are mostly conceptual extensions of things the reader has already met. The new vocabulary (`devfs`, `cdev`, `ioctl`, `sysctl`, `ifnet`, `CAM`) is mostly familiar by name from earlier chapters; the chapter's job is to give each of them a concrete shape.
- **Reading plus typing the worked examples**: ten to twelve hours over two or three sessions. The driver evolves through three integration surfaces in turn (`devfs`, `ioctl`, `sysctl`), each with its own short stage. Each stage is short and self-contained; the testing is what takes the time, because the integration surfaces are best tested by writing a small user-space program that drives them.
- **Reading plus all labs and challenges**: fifteen to eighteen hours over three or four sessions. The labs include a clone-aware `devfs` experiment, a full `ioctl` roundtrip with `myfirstctl`, a `sysctl`-driven counter monitoring exercise, a cleanup-discipline lab that intentionally breaks teardown to expose the failure pattern, and a small `ifnet`-stub challenge for readers who want a preview of Chapter 28.

Sections 3 and 4 are the densest in terms of new vocabulary. The `ioctl` macros and the `sysctl` callback signatures are the only truly new APIs in the chapter; the rest is composition. If the macros feel opaque on a first pass, that is normal. Stop, run the matching exercise on the driver, and come back when the shape has settled.

24.1.6 Prerequisites

Before starting this chapter, confirm:

- Your driver source matches Chapter 23 Stage 3 (1.6-debug). The starting point assumes every Chapter 23 primitive: the `DPRINTF` macro, the SDT probes, the debug mask `sysctl`, and the `my-first_debug.c/myfirst_debug.h` file pair. Chapter 24 builds new integration code that uses each of those at appropriate moments.
- Your lab machine runs FreeBSD 14.3 with `/usr/src` on disk and matching the running kernel.
- A debug kernel with `INVARIANTS`, `WITNESS`, `WITNESS_SKIPSPIN`, `DDB`, `KDB`, `KDB_UNATTENDED`, `KDTRACE_HOOKS`, and `DDB_CTF` is built, installed, and booting cleanly.
- `bhyve(8)` or `qemu-system-x86_64` is available, and you have a usable VM snapshot at the 1.6-debug state. Chapter 24's labs include intentional failure scenarios for the cleanup-discipline section, and a snapshot makes recovery cheap.
- The following user-space commands are in your path: `dmesg`, `sysctl`, `kldstat`, `kldload`, `kldunload`, `devctl`, `cc`, `make`, `dtrace`, `dd`, `head`, `cat`, `chown`, `chmod`, and `truss`. Chapter 24's labs make light use of `truss`, the FreeBSD equivalent of Linux `strace`, to verify that user-space programs really do reach the driver through the new `ioctl`s.
- You are comfortable writing a short C program against the FreeBSD `libc` headers. The chapter introduces the user-space side of the new `ioctl`s through a small program named `myfirstctl`.
- A working knowledge of `git` is helpful but not required. The chapter recommends you commit between stages so each version of the driver has a recoverable point.

If any item above is shaky, fix it now. Integration code is, on balance, less dangerous than the kernel-mode work of earlier chapters because most of the failure modes are caught at module load or at user-space invocation rather than producing kernel panics. But the lessons compound: a clone-aware `make_dev` mistake in this chapter will produce ugly diagnostics in Chapter 28 when the network driver also wants its own nodes, and a `sysctl` OID leak in this chapter will produce confusing module-load failures in Chapter 27 when the storage driver tries to register names that already exist.

24.1.7 How to Get the Most Out of This Chapter

Five habits pay off in this chapter more than in any of the previous Part 5 chapters.

First, keep `/usr/src/sys/dev/null/null.c`, `/usr/src/sys/sys/conf.h`, `/usr/src/sys/sys/ioccom.h`, and `/usr/src/sys/sys/sysctl.h` bookmarked. The first is the shortest non-trivial character driver in the FreeBSD source tree and is the canonical example of the `cdevsw/make_dev/destroy_dev` pattern. The second declares the `cdevsw` structure, the `make_dev` family, and the `MAKEDEV_*` flag bits the chapter uses repeatedly. The third defines the `ioctl` encoding macros (`_IO`, `_IOR`, `_IOW`, `_IOWR`) and contains the `IOC_VOID`, `IOC_IN`, `IOC_OUT`, and `IOC_INOUT` constants the kernel uses to decide whether to copy data automatically. The fourth defines the `sysctl` OID macros, the `SYSTCL_HANDLER_ARGS` calling convention, and the static and dynamic OID interfaces. None of these files are long; the longest is a few thousand lines and most of that is comments. Reading each once at the start of the corresponding section is the single most effective thing you can do for fluency.

Second, keep three real driver examples close to hand: `/usr/src/sys/dev/null/null.c`, `/usr/src/sys/net/if_tuntap.c`, and `/usr/src/sys/dev/virtio/block/virtio_blk.c`. The first is the minimal

cdevsw example. The second is the canonical clone-aware `dev_clone` example used by Section 5 to introduce `ifnet`. The third illustrates a full `device_get_sysctl_ctx` based dynamic `sysctl` tree and a `SYSCTL_ADD_PROC` callback that toggles a runtime knob. Chapter 24 points back to each at the right moment. Reading them once now, without trying to memorise, gives the rest of the chapter concrete anchors to hang its ideas on.

A note on line numbers. Pointers into `null.c`, `if_tuntap.c`, and `virtio_blk.c` later in the chapter are anchored on named symbols: a specific `make_dev` call, a `SYSCTL_ADD_PROC` handler, a particular `cdevsw`. Those names carry across future FreeBSD 14.x releases. The specific line each name sits on does not. When the prose quotes a location, open the file and search for the symbol rather than scrolling to the number.

Third, type every code change into the `myfirst` driver by hand. Integration code is the kind of code that is easy to copy and very hard to remember later. Typing out the `cdevsw` table, the `ioctl` command definitions, the `sysctl` tree construction, and the user-space `myfirstctl` program builds the kind of familiarity that copy-paste cannot. The goal is not to have the code; the goal is to be the person who could write it again, from scratch, in twenty minutes when a future bug demands it.

Fourth, build the user-space program at every stage. Many of the chapter's lessons are visible only on the user side. Whether the kernel has copied an `ioctl` payload correctly, whether a `sysctl` is readable but not writable, whether a `/dev` node has the permissions you set, whether a clone produces a usable device node, all of those questions are answered with `cat`, `dd`, `chmod`, `sysctl`, `truss`, and the small `myfirstctl` companion program. A driver tested only by its kernel-side counters has been only half tested.

Fifth, after finishing Section 4 re-read Chapter 23's debug discipline. Each integration surface in Chapter 24 is wrapped in a `DPRINTF` from Chapter 23. Each `ioctl` path fires `MYF_DBG_IOCTL` log lines. Each `sysctl` path can be observed through the SDT machinery. Seeing how Chapter 23's tools serve Chapter 24's interfaces reinforces both chapters and prepares the reader for Chapter 25, where the same pattern continues.

24.1.8 Roadmap Through the Chapter

The sections in order are:

1. **Why Integration Matters.** The conceptual story. From standalone module to system component; the price of leaving integration as an afterthought; the four user-visible interfaces every integrated driver eventually needs; the optional subsystem hooks the chapter introduces but does not finish.
2. **Working With devfs and the Device Tree.** The kernel's view of `/dev`. What `devfs` is and how it differs from the static device tables of older systems; the lifecycle of a `cdev`; `make_dev` and friends in detail; the `cdevsw` structure and its callbacks; the `si_drv1/si_drv2` fields; permissions and ownership; clonable nodes through the `dev_clone` event handler. Stage 1 of the Chapter 24 driver replaces the original ad-hoc node creation with a clean clone-aware pattern.
3. **Implementing ioctl() Support.** The user-driven control interface. What `ioctls` are; the `_IO/_IOR/_IOW/_IOWR` encoding; the kernel's automatic copyin/copyout layer; how to choose a magic letter and number; how to lay out a public `ioctl` header; how to write a `d_ioctl` callback that dispatches on the command word; common pitfalls (variable-length data, embedded pointers, version evolution). Stage 2 adds `myfirst_ioctl.c` and `myfirst_ioctl.h` and a small `myfirstctl` user-space program.

4. **Exposing Metrics Through `sysctl()`.** The monitoring and tuning interface. What `sysctls` are; static versus dynamic OIDs; the `device_get_sysctl_ctx/device_get_sysctl_tree` pattern; counters, knobs, and procedural callbacks; the `SYSCTL_ADD_*` family; tunables in `/boot/loader.conf`; access control and units. Stage 3 adds `myfirst_sysctl.c` and a per-instance metrics subtree.
5. **Integration With the Networking Subsystem (Optional).** A brief, conceptual look at `ifnet`. What `if_t` is; the `if_alloc/if_initname/if_attach/bpfattach/if_detach/if_free` outline; how `tun(4)` and `tap(4)` are structured around it; what role the driver plays inside the larger network stack. The section is short on purpose; Chapter 28 is the network driver chapter.
6. **Integration With the CAM Storage Subsystem (Optional).** A brief, conceptual look at CAM. What CAM is; what a SIM and a CCB are; the `cam_sim_alloc/xpt_bus_register/xpt_action/xpt_done` outline; how a small read-only memory disk could be exposed through it. The section is short on purpose; Chapter 27 is the storage driver chapter.
7. **Registration, Teardown, and Cleanup Discipline.** The cross-cutting topic. Module event handlers (`MOD_LOAD`, `MOD_UNLOAD`, `MOD_SHUTDOWN`); attach failure with partial cleanup; detach failure when users still hold open handles; the cleanup-on-failure pattern; ordering between integration surfaces; what `bus_generic_attach`, `bus_generic_detach`, and `device_delete_children` do for you; `SYSINIT` and `EVENTHANDLER` for cross-cutting registrations.
8. **Refactoring and Versioning an Integrated Driver.** The clean house. The final split into `myfirst.c`, `myfirst_debug.c/.h`, `myfirst_ioctl.c/.h`, and `myfirst_sysctl.c/.h`; the public `myfirst.h` header for user-space callers; the `INTEGRATION.md` document; the version bump to 1.7-integration; the regression test additions; the commit and tag.

After the eight sections come a set of hands-on labs that exercise each integration surface end to end, a set of challenge exercises that stretch the reader without introducing new foundations, a troubleshooting reference for the symptoms most readers will hit, a Wrapping Up that closes Chapter 24's story and opens Chapter 25's, a bridge to the next chapter, and the usual quick-reference card and glossary.

If this is your first pass, read linearly and do the labs in order. If you are revisiting, Sections 2 and 3 stand alone and make good single-sitting reads. Section 5 and Section 6 are short and conceptual and can be skipped on a first pass without losing the chapter's main thread, then returned to before starting Chapter 26 or Chapter 27.

A small note before the technical work begins. The chapter often asks the reader to compile a tiny user-space program, run it against the driver, observe the result, and then return to the kernel side to read what happened. That rhythm is deliberate. Integration is not a property of the driver alone; it is a property of the relationship between the driver and the rest of the system. The user-space programs are short, but they are how the chapter measures whether each integration surface really works.

24.2 Section 1: Why Integration Matters

Before the code, the framing. Section 1 spells out what changes when a driver becomes integrated. The reader who has followed Part 4 and the first chapters of Part 5 has built a working driver. What does it mean to say that this driver is not yet *integrated*, and what specific qualities does integration add?

This section answers that question carefully and at length, because the rest of the chapter is the implementation of these qualities. A reader who knows clearly *why* each integration surface exists will find the implementation work in Sections 2 through 8 much easier. A reader who skips the framing will spend the chapter wondering

why the driver needs an `ioctl` at all when an internal `sysctl` could do the same job; that question has a real answer, and Section 1 is where the answer lives.

24.2.1 From Standalone Module to System Component

A standalone driver is a kernel module that does its job correctly when called by the kernel and stays out of the way otherwise. The current `myfirst` driver, at version 1.6-debug, is exactly that kind of module. It has a single `cdev` node, no public `ioctls`, no published `sysctls` beyond a handful of internal debug knobs, no relationship to any kernel subsystem outside its own little corner, and no expectation that any user-space program will reach in and tell it what to do. It works, and it works in isolation.

A system component, by contrast, is a kernel module whose value depends on its relationships with the rest of the system. The same `myfirst` driver, integrated, presents a `/dev/myfirst0` node with the right permissions for the role of its hardware, exposes a small `ioctl` interface that user programs can use to reset the device or query its status, publishes a per-instance `sysctl` tree that monitoring software can scrape, registers itself with the appropriate kernel subsystem if the hardware is a network or storage device, and cleans up cleanly when unloaded. Each of those interfaces is small. Together they are the difference between a driver that runs on one developer's lab machine and a driver that ships with FreeBSD.

The shift from standalone to integrated is not a one-line change to the driver. It is a series of conscious decisions, each of which expands the driver's public surface and each of which incurs a maintenance cost. Decisions that look small in the moment, like the choice of a magic letter for an `ioctl` or the name of a `sysctl` OID, become long-lived contracts. A driver that picked the letter `M` for its `ioctls` in 2010 still has those numbers in its public header today, because changing them would break every user-space program that ever called them.

A useful way to picture the shift is to imagine two readers approaching the driver. The first reader is the developer who wrote the driver: they know everything about it, they can change anything in it, and they can rebuild it whenever they like. The second reader is the system administrator who installs the module from a package and never reads its source: they only ever see the driver through its `/dev` nodes, its `sysctls`, its `ioctls`, and its log messages. A standalone driver is one designed for the first reader; an integrated driver is one designed for both.

Chapter 24 teaches the reader how to design for the second reader. That is the conceptual work the chapter does, and the code work in Sections 2 through 8 is the practical realisation of it.

24.2.2 Common Integration Targets

A FreeBSD driver typically integrates with four kernel-side surfaces and, depending on the hardware, with one of two subsystems. Naming the targets clearly at the outset helps the reader hold the chapter's structure in mind.

The first target is **devfs**. Every character driver creates one or more `/dev` nodes through `make_dev` (or one of its variants) and removes them through `destroy_dev`. The shape and naming of those nodes is how the rest of the system addresses the driver. A node named `/dev/myfirst0` lets the administrator open it with `cat /dev/myfirst0`, lets a script include it in a `find /dev -name 'myfirst*'`, and lets the kernel itself dispatch open, read, write, and `ioctl` calls into the driver. Section 2 is dedicated to devfs.

The second target is **ioctl(2)**. The `read(2)` and `write(2)` syscalls move bytes; they do not control the driver. Any control operation that does not fit the read/write data-flow model lives in `ioctl(2)`. A user-space program

calls `ioctl(fd, MYF_RESET)` to ask the driver to reset its hardware, or `ioctl(fd, MYF_GET_STATS, &stats)` to read out a counter snapshot. Each `ioctl` is a small, well-typed entry point with a number, a direction, and a payload. Section 3 is dedicated to `ioctls`.

The third target is **`sysctl(8)`**. Counters, statistics, and tunable parameters live in the system-wide `sysctl` tree, accessible from user space with `sysctl(8)` and from C with `sysctlbyname(3)`. A driver places its OIDs under `dev.<driver>.<unit>.<name>` so that `sysctl dev.myfirst.0` lists every metric and knob the device exposes. The `sysctl` interface is the right home for read-only counters and for slow-changing knobs; `ioctl` is the right home for fast actions and for typed data. Section 4 is dedicated to `sysctls`.

The fourth target is **the kernel's lifecycle hooks**. The module event handler (`MOD_LOAD`, `MOD_UNLOAD`, `MOD_SHUTDOWN`), the device tree's attach and detach methods, and the cross-cutting registrations through `EVENTHANDLER(9)` and `SYSINIT(9)` together define what happens when the driver enters and leaves the kernel. A driver that gets the integration interfaces right but the lifecycle wrong leaks resources, deadlocks at unload, or panics on the third `kldload`/`kldunload` cycle. Section 7 is dedicated to lifecycle discipline.

In addition, drivers whose hardware is a network device integrate with the **`ifnet`** subsystem, and drivers whose hardware is a storage device integrate with the **`CAM`** subsystem. Both are conceptually large enough that a deep treatment requires its own chapter (Chapter 28 for `ifnet`, Chapter 27 for `CAM` and `GEOM`). Section 5 and Section 6 of this chapter introduce them at the level needed to recognise their shape and to know what kind of work they involve.

Each target deserves naming for a specific reason. `devfs` is the name a casual user types into a shell. `ioctl` is the entry point for typed actions a program needs to issue against a device. `sysctl` is the place a monitoring tool looks for numbers. The lifecycle hooks are the place a packager and a system administrator hit when they `kldload` and `kldunload`. The four together cover the four faces of the driver that matter to anyone other than its author.

24.2.3 Benefits of Proper Integration

Integration is not an end in itself. It is a means to a small set of practical outcomes that the chapter will keep returning to.

The first outcome is **monitoring**. A driver whose counters are visible through `sysctl` can be scraped by `prometheus-fbsd-exporter`, by Nagios checks, by a small shell script that runs every minute. A driver whose counters are only visible through `device_printf` to `dmesg` can only be inspected by reading the log file by hand. The two operational realities are very different, and the difference is determined entirely by the integration choice the developer made when the driver was written.

The second outcome is **management**. A driver that exposes an `ioctl` named `MYF_RESET` lets the administrator script a reset cycle into a maintenance window. A driver without that interface forces the administrator to `kldunload` and `kldload` the module, which is a much heavier operation that drops every open file descriptor and may not be acceptable while production traffic is flowing through the device.

The third outcome is **automation**. A driver that emits well-formed `/dev` nodes with predictable names lets `devd(8)` react to attach and detach events, run scripts on hotplug, and integrate the driver into the larger system's startup, shutdown, and recovery flows. A driver that emits an opaque single node and never tells anyone about its lifecycle cannot be automated without resorting to `dmesg` log scraping, which is brittle.

The fourth outcome is **reusability**. A driver whose ioctl interface is well documented can be the foundation of higher-level libraries. The `bsnmp` daemon, for example, uses well-defined kernel interfaces to expose driver counters through SNMP without touching the driver source. A driver that designed its interfaces correctly the first time gains those benefits without further work.

These four outcomes (monitoring, management, automation, reusability) are the practical reason for everything that follows. Each section in the chapter delivers a piece of one of these outcomes, and the closing refactor in Section 8 is what makes the whole package presentable to the rest of the system.

24.2.4 A Small Tour of System Tools That Depend on Integration

A useful exercise before the technical work is to look at the user-space tools that exist precisely because drivers integrate with the kernel subsystems above. None of these tools work for a standalone driver. All of them work, automatically, for an integrated one.

`devinfo(8)` walks the kernel's device tree and prints what it finds. It works because every device in the tree was registered through the newbus interface and every device has a name, a unit number, and a parent. The administrator runs `devinfo -v` and sees the entire device hierarchy, including the `myfirst0` instance.

`sysctl(8)` reads from and writes to the kernel `sysctl` tree. It works because every counter and knob in the kernel is reachable through the OID hierarchy, including the OIDs the driver registered through `device_get_sysctl_tree`.

`devctl(8)` lets the administrator manipulate individual devices: `devctl detach`, `devctl attach`, `devctl suspend`, `devctl resume`, `devctl rescan`. It works because every device implements the `kobj` methods the kernel's device tree machinery expects. Chapter 22 already used `devctl suspend myfirst0` and `devctl resume myfirst0`.

`devd(8)` watches the kernel's device-event channel and runs scripts in response to `attach`, `detach`, `hotplug`, and similar events. It works because the kernel emits structured events for each newbus operation. A driver that follows the standard newbus pattern is automatically visible to `devd`.

`ifconfig(8)` configures network interfaces. It works because every network driver registers with the `ifnet` subsystem and accepts a standard set of ioctls (Section 5 introduces this).

`camcontrol(8)` controls SCSI and SATA devices through CAM. It works because every storage driver registers a SIM and processes CCBs (Section 6 introduces this).

`gstat(8)` shows real-time GEOM statistics, `geom(8)` lists the GEOM tree, `top -H` shows per-thread CPU usage. Each of these tools relies on a specific integration surface that drivers register with. The driver that ignores those surfaces gets none of the benefit.

A simple way to confirm the point is to run, on your lab machine, the following exercises:

```
# Tour the device tree
```

```
devinfo -v | head -40
```

```
# Tour the sysctl tree, just the dev branch
```

```
sysctl dev | head -40
```

```
# See what the live network interfaces look like
ifconfig -a

# See what storage looks like through CAM
camcontrol devlist

# See what GEOM sees
geom -t

# Watch the device event channel for a few seconds
sudo devd -d -f /dev/null &
DEVDPID=$!
sleep 5
kill $DEVDPID
```

Each command exists because drivers integrate. Read the output and notice how much of what is visible came from drivers that did the integration work. The `myfirst` driver, at the start of Chapter 24, contributes almost nothing to that output. By the end of Chapter 24, it will contribute its `dev.myfirst.0` subtree to `sysctl`, its `myfirst0` device to `devinfo -v`, and its `/dev/myfirst*` nodes to the file system. Each step is small. The aggregate is the difference between a one-off lab driver and a real piece of FreeBSD.

24.2.5 What “Optional” Means in This Chapter

Section 5 (networking) and Section 6 (storage) are labelled optional. The label deserves a careful definition.

Optional does not mean unimportant. Both sections will become essential reading later in the book, the networking material before Chapter 27 and the storage material before Chapter 26 and Chapter 28. Optional means that for a reader who is following the `myfirst` PCI driver as the running example, the network and storage hooks would not be exercised in this chapter, because `myfirst` is not a network device and is not a storage device. The chapter introduces the conceptual shape of those hooks so that the reader recognises them when they appear, and so that the structural decisions in Section 7 and Section 8 take them into account.

A reader on a first pass who is short on time can skip Sections 5 and 6. The other sections do not depend on them. A reader who plans to follow Chapter 26 or Chapter 27 should read them, because they introduce the vocabulary those chapters will assume.

The chapter is honest about the depth of what it teaches in those sections. The networking subsystem is several thousand lines of source code in `/usr/src/sys/net`. The CAM subsystem is several thousand lines of source code in `/usr/src/sys/cam`. Each took years to design and is still evolving. Chapter 24 introduces them at the level of *here is the shape, here are the calls a driver typically makes, here is one real driver that uses each*. The full mechanics belong elsewhere.

24.2.6 Pitfalls on the Way to Integration

Three pitfalls trip up most first-time integrators.

The first pitfall is **adding integration surfaces ad hoc**. A driver that grew an `ioctl` one Tuesday because the developer needed a quick way to test the device, and a `sysctl` the following week because the developer wanted to see a counter, and another `ioctl` the following month because of a reported bug, ends up with a public surface that is inconsistent in style, inconsistent in naming, inconsistent in error handling, and inconsistent in documentation. The right pattern is to design the public surface deliberately, to use consistent naming, and to document each entry point in a header file before writing the implementation. Section 3 and Section 8 return to this discipline.

The second pitfall is **mixing concerns inside the device methods**. A driver whose `device_attach` does the `kobj` work, the resource allocation, the `devfs` node creation, the `sysctl` tree construction, and the user-space-facing setup in one long function quickly becomes unreadable. The chapter recommends separating these concerns into helper functions at first, and into separate source files in Section 8. The `myfirst_debug.c` and `myfirst_debug.h` pair from Chapter 23 was the first step in that direction; the new `myfirst_ioctl.c` and `myfirst_sysctl.c` files from this chapter continue the pattern.

The third pitfall is **not testing the public surface from user space**. A driver that the developer tested only by exercising it from the kernel side will pass every kernel-side test and still fail the moment a real user-space program calls it, because the developer assumed something about the calling convention that does not hold in practice. The chapter therefore insists on building the small `myfirstctl` companion program as soon as the driver has any `ioctls`, and on testing every `sysctl` through `sysctl(8)` rather than only by reading the in-driver counter directly. The user-space tests are the only ones that confirm integration actually works.

These pitfalls are not unique to FreeBSD. They show up in every operating system that has a kernel-user boundary. FreeBSD's tooling makes it easier than most systems to do the right thing, because the conventions are well documented in the manual pages (`devfs(5)`, `ioctl(2)`, `sysctl(9)`, `style(9)`) and because the kernel itself ships with hundreds of integrated drivers the reader can study. The chapter relies on those conventions and points back to the matching real drivers as it goes.

24.2.7 A Mental Model for the Chapter

Before moving into Section 2, it helps to fix a single picture in mind. The driver, by the end of Chapter 24, will look like this from the outside:

Userland tools		Kernel
+-----+		+-----+
myfirstctl	ioctl(2)	d_ioctl callback
sysctl(8)	+----->	sysctl OID tree
cat /dev/myfirst0	read/write	d_read, d_write
chmod, chown	fileops	devfs node lifecycle
devinfo -v	newbus query	device_t myfirst0
dtrace -n 'myfirst:::'	SDT probes	sc_debug, DPRINTF
+-----+		+-----+

Every entry on the left is a tool a real FreeBSD user already knows. Every entry on the right is a piece of integration the chapter teaches you to write. The arrows in between are what each section of the chapter implements.

Hold that picture in mind as the technical work begins. The point of every section that follows is to add one of those arrows to the driver, with the discipline that lets the arrow stay reliable as the driver grows.

24.2.8 Wrapping Up Section 1

Integration is the discipline of making a driver visible, controllable, and observable from outside its own source code. The four primary integration surfaces in FreeBSD are `devfs`, `ioctl`, `sysctl`, and the kernel lifecycle hooks. Two optional subsystem hooks are `ifnet` for network devices and `CAM` for storage devices. Together, these are how a driver stops being a one-developer project and becomes a piece of FreeBSD. The remaining sections of this chapter implement each surface, with the `myfirst` driver as the running example and the version bumping from 1.6-debug to 1.7-integration as the visible milestone.

In the next section, we turn to the first and most foundational integration surface: `devfs` and the device file system that gives every character driver its `/dev` presence.

24.3 Section 2: Working With `devfs` and the Device Tree

The first integration surface every character driver crosses is `devfs`, the device file system. The reader has been creating `/dev/myfirst0` since the earliest chapters, but the call to `make_dev` was always presented as a single line of boilerplate without much explanation. Section 2 fills that gap. It explains what `devfs` actually is, walks through the lifecycle of a `cdev`, surveys the variants of `make_dev` and the `cdevsw` callback table in detail, shows how to attach per-node state through `si_drv1` and `si_drv2`, teaches the modern clone-aware pattern for drivers that want one node per instance on demand, and ends by showing how an administrator can adjust permissions and ownership without rebuilding the driver.

24.3.1 What `devfs` Is

`devfs` is a virtual file system that exposes the kernel's set of registered character devices as a tree of files under `/dev`. It is virtual in the same sense `procfs` and `tmpfs` are virtual: there is no on-disk storage backing it. Every file under `/dev` is the kernel's projection of a `cdev` structure into the file system namespace. When a user-space program calls `open("/dev/null", O_RDWR)`, the kernel looks up the path in `devfs`, finds the matching `cdev`, follows the pointer to the `cdevsw` table, and dispatches the open through it.

Older UNIX systems used a static device tree. An administrator ran a program like `MAKEDEV` or edited `/dev` directly, and the file system contained device nodes whether the corresponding hardware was present or not. The static approach had two well-known problems. First, the administrator had to know in advance which devices were possible and create the matching nodes by hand, with the right major and minor numbers. Second, the file system contained orphan nodes for hardware the system did not actually have, which was confusing.

FreeBSD's `devfs`, introduced as the default in the early 5.x release series, replaced the static scheme with a dynamic one. The kernel itself decides which device nodes exist, based on which drivers have called `make_dev`. When the driver calls `make_dev`, the node appears under `/dev`. When the driver calls `destroy_dev`, the node disappears. The administrator does not maintain device entries by hand any more, and there are no orphan nodes for hardware that is not present.

The trade-off `devfs` introduces is that the lifecycle of a `/dev` node is now controlled by the kernel rather than by the file system. A driver that fails to remove its node on detach leaves it visible until the kernel itself removes it (which it eventually will, but not as immediately as the driver). A driver that accidentally creates the same node twice gets a panic from the duplicate-name check. A driver that creates a node from the wrong context can deadlock the kernel. The chapter teaches the patterns that avoid each of these.

A useful detail for understanding devfs is that the kernel keeps a single global namespace of device nodes, and `make_dev` registers into that namespace. The administrator can mount additional devfs instances inside jails or chroots; each instance projects a filtered view of the global namespace, controlled through `devfs.rules(8)`. The driver itself does not need to know about these projections. It simply registers its `cdev` once, and the kernel and the rule system together decide which views can see it.

24.3.2 The Lifecycle of a `cdev`

Every `cdev` goes through five stages. Knowing these stages by name makes the rest of the section easier to follow.

The first stage is **registration**. The driver calls `make_dev` (or one of its variants) from `device_attach` (or from a module event handler, depending on whether the device is bus-attached or pseudo-device). The call returns a `struct cdev *` that the driver stores in its `softc`. From this moment on, the node is visible under `/dev`.

The second stage is **use**. User-space programs can open the node and call `read`, `write`, `ioctl`, `mmap`, `poll`, or `kqueue` against it. Each call goes through the matching `cdevsw` callback the driver installed at registration time. A `cdev` may have many open file handles against it at any moment, and the driver's callbacks must be safe to call concurrently with each other and with the driver's own internal work (interrupts, callouts, taskqueues).

The third stage is **destruction request**. The driver calls `destroy_dev` (or `destroy_dev_sched` for the asynchronous variant). The node is unlinked from `/dev` immediately, so no new opens can succeed. Existing opens are not closed at this moment.

The fourth stage is **drain**. `destroy_dev` blocks until every open file handle for the `cdev` has gone through `d_close` and every in-flight call into the driver has returned. The driver's callbacks are guaranteed not to be called any more once `destroy_dev` returns.

The fifth stage is **release**. Once `destroy_dev` returns, the driver can free the `softc`, release any resources the `cdev` callbacks were using, and unload the module. The `struct cdev *` itself is owned by the kernel and is freed by the kernel when its last reference goes away; the driver does not free it.

The drain step is the one that catches first-time drivers most often. A naive driver does the equivalent of “`destroy_dev; free(sc);`” and then a held open file handle calls into the `cdevsw` and dereferences the freed `softc`, which panics. The chapter teaches how to handle this correctly: put the `destroy_dev` call before any state-freeing in the detach path, and trust the kernel to drain the in-flight calls before the destroy returns.

24.3.3 The `make_dev` Family

FreeBSD provides several variants of `make_dev`, each with a different combination of options. They live in `/usr/src/sys/sys/conf.h` and `/usr/src/sys/fs/devfs/devfs_devs.c`. The chapter introduces the most useful four.

The simplest form is **`make_dev`** itself:

```
struct cdev *
make_dev(struct cdevsw *devsw, int unit, uid_t uid, gid_t gid,
         int perms, const char *fmt, ...);
```

This call creates a node owned by `uid:gid` with permissions `perms`, named according to the `printf`-style format. It uses `M_WAITOK` internally and may sleep, so it must be called from a context that can sleep (typically `device_attach` or a module load handler, never an interrupt handler). It cannot fail: if it cannot allocate memory, it sleeps until it can. The reader has been using this form since the early chapters.

The richer form is `make_dev_credf`:

```
struct cdev *
make_dev_credf(int flags, struct cdevsw *devsw, int unit,
               struct ucred *cr, uid_t uid, gid_t gid, int mode,
               const char *fmt, ...);
```

This variant takes an explicit `flags` argument and an explicit credential. The credential is used by the MAC framework when checking whether the device may be created with the given owner. The flags select features such as `MAKEDEV_ETERNAL` (the kernel never destroys this node automatically) and `MAKEDEV_ETERNAL_KLD` (the same, but allowed only inside a loadable module). The `null(4)` driver uses this form, as the chapter quoted in Section 1's reference list.

The form recommended for new drivers is `make_dev_s`:

```
int
make_dev_s(struct make_dev_args *args, struct cdev **cdev,
           const char *fmt, ...);
```

This variant takes a structure of arguments rather than a long argument list. The structure is initialised with `make_dev_args_init(&args)` before its fields are filled in. The advantage of `make_dev_s` is that it can fail rather than sleep, and the failure is reported through a return value rather than through a sleep. It also has an output parameter for the `cdev *`, which means the caller does not need to remember which positional return represents what. New code should prefer `make_dev_s` because the failure path is cleaner.

The argument structure looks like this:

```
struct make_dev_args {
    size_t      mda_size;
    int         mda_flags;
    struct cdevsw *mda_devsw;
    struct ucred *mda_cr;
    uid_t       mda_uid;
    gid_t       mda_gid;
    int         mda_mode;
    int         mda_unit;
    void        *mda_si_drv1;
    void        *mda_si_drv2;
};
```


The `mda_size` field is used by the kernel to detect ABI mismatches; `make_dev_args_init` sets it correctly. The `mda_si_drv1` and `mda_si_drv2` fields let the driver attach two pointers of its own to the `cdev` at creation time; the chapter uses `mda_si_drv1` to attach a pointer to the `softc`.

The `MAKEDEV_*` flag bits relevant to most drivers are:

Flag	Meaning
<code>MAKEDEV_REF</code>	The returned <code>cdev</code> is referenced; balance with <code>dev_rel</code> .
<code>MAKEDEV_NOWAIT</code>	Do not sleep; return failure if the call would have to sleep.
<code>MAKEDEV_WAITOK</code>	The call may sleep (default for <code>make_dev</code>).
<code>MAKEDEV_ETERNAL</code>	The kernel does not destroy this node automatically.
<code>MAKEDEV_ETERNAL_KLD</code>	Same as <code>ETERNAL</code> but allowed inside a loadable module.
<code>MAKEDEV_CHECKNAME</code>	Validate the name against the devfs character set.

The `make_dev_p` variant is similar to `make_dev_s` but takes a positional argument list. It is older, still supported, and used by some drivers in the tree; new drivers can ignore it in favour of `make_dev_s`.

24.3.4 The `cdevsw` Structure

The `cdevsw` table is the dispatch table for character device callbacks. The reader has installed one in earlier chapters, but Section 2 examines it field by field.

A minimal modern `cdevsw` looks like:

```
static struct cdevsw myfirst_cdevsw = {
    .d_version = D_VERSION,
    .d_flags   = D_TRACKCLOSE,
    .d_name    = "myfirst",
    .d_open    = myfirst_open,
    .d_close   = myfirst_close,
    .d_read    = myfirst_read,
    .d_write   = myfirst_write,
    .d_ioctl   = myfirst_ioctl,
};
```

`d_version` must be set to `D_VERSION`. The kernel uses this field to detect drivers built against an older `cdevsw` layout. A missing or wrong `d_version` is a common source of confusing module-load failures; always set it explicitly.

`d_flags` controls a small set of optional behaviours. The most common flags are:

Flag	Meaning
D_TRACKCLOSE	Call <code>d_close</code> on the last close of each fd, not on every close.
D_NEEDGIANT	Take the kernel-wide Giant lock around dispatch (rare in modern code).
D_NEEDMINOR	Allocate a minor number (legacy; rarely needed today).
D_MMAP_ANON	The driver supports anonymous mmap through <code>dev_pager</code> .
D_DISK	The cdev is the entry point for a disk-like device.
D_TTY	The cdev is a terminal device; affects line-discipline routing.

For the `myfirst` driver, `D_TRACKCLOSE` is the only flag worth setting. It causes the kernel to call `d_close` exactly once per file descriptor, on the last close of that descriptor, rather than on every close. Without `D_TRACKCLOSE`, a driver that wants to count open file handles has to handle the same `d_close` being called many times, which is awkward.

`d_name` is the name the kernel uses in some diagnostic messages. It is conventionally the same as the driver's name.

The callback fields are pointers to the functions that implement each operation. A driver only needs to install the callbacks it actually supports; missing callbacks default to safe stubs that return `ENODEV` or `EOPNOTSUPP`. The most common combination for a character driver is `d_open`, `d_close`, `d_read`, `d_write`, and `d_ioctl`. Drivers that support polling add `d_poll`. Drivers that support `kqueue` add `d_kqfilter`. Drivers that map memory into user space add `d_mmap` or `d_mmap_single`. Drivers that emulate a disk add `d_strategy`.

The `d_purge` callback is rare but worth knowing. The kernel calls it when the cdev is being destroyed and the driver should release any pending I/O. Most drivers do not need it because their `d_close` already handles release.

The `d_open` callback signature is:

```
int myfirst_open(struct cdev *dev, int oflags, int devtype, struct thread *td);
```

`dev` is the cdev being opened. `oflags` is the union of the `open(2)` flags (`O_RDWR`, `O_NONBLOCK`, etc.). `devtype` carries the device type and is rarely useful for a character driver. `td` is the thread doing the open. The callback returns 0 on success or an `errno` on failure. A typical pattern stores the softc pointer found through `dev->si_drv1` into a per-open private structure that subsequent calls can recover.

The `d_close` signature is parallel:

```
int myfirst_close(struct cdev *dev, int fflags, int devtype, struct thread *td);
```

The `d_read` and `d_write` signatures use the `uio` machinery the reader met in Chapter 8:

```
int myfirst_read(struct cdev *dev, struct uio *uio, int ioflag);
int myfirst_write(struct cdev *dev, struct uio *uio, int ioflag);
```

The `d_ioctl` signature is:

```
int myfirst_ioctl(struct cdev *dev, u_long cmd, caddr_t data,
                 int fflag, struct thread *td);
```

`cmd` is the `ioctl` command word. `data` points to the `ioctl` data buffer (which the kernel has already copied in for `IOC_IN` commands and will copy out for `IOC_OUT` commands, as Section 3 will discuss in detail). `fflag` is the file flags from the open call. `td` is the calling thread.

24.3.5 Per-Cdev State Through `si_drv1`

The `cdev` is the kernel's handle to the device, and the driver almost always needs a way to find its own `softc` from a `cdev` pointer. The standard mechanism is `cdev->si_drv1`. The driver sets this field when it creates the `cdev` (either at the call to `make_dev` or by writing to the field afterward) and reads it in every `cdevsw` callback.

The pattern looks like this in attach:

```
sc->sc_cdev = make_dev(&myfirst_cdevsw, device_get_unit(dev),
                     UID_ROOT, GID_WHEEL, 0660, "myfirst%d", device_get_unit(dev));
sc->sc_cdev->si_drv1 = sc;
```

And like this in each callback:

```
static int
myfirst_open(struct cdev *dev, int oflags, int devtype, struct thread *td)
{
    struct myfirst_softc *sc = dev->si_drv1;

    DPRINTF(sc, MYF_DBG_OPEN, "open: pid=%d flags=%#x\n",
            td->td_proc->p_pid, oflags);
    /* ... rest of open ... */
    return (0);
}
```

`si_drv2` is a second pointer the driver can use however it likes. Some drivers use it for a per-instance cookie; others use it for nothing and leave it `NULL`. The `myfirst` driver uses only `si_drv1`.

The `make_dev_s` variant is cleaner because it sets `si_drv1` at creation time:

```
struct make_dev_args args;
make_dev_args_init(&args);
args.mda_devsw = &myfirst_cdevsw;
args.mda_uid = UID_ROOT;
args.mda_gid = GID_WHEEL;
```

```

args.mda_mode = 0660;
args.mda_si_drv1 = sc;
args.mda_unit = device_get_unit(dev);
error = make_dev_s(&args, &sc->sc_cdev, "myfirst%d", device_get_unit(dev));
if (error != 0) {
    device_printf(dev, "make_dev_s failed: %d\n", error);
    goto fail;
}

```

The `make_dev_s` form has the additional advantage that `si_drv1` is set before the `cdev` is visible in `/dev`, which closes a small but real race window in which a quick-to-open program could call into a `cdevsw` whose `si_drv1` was still `NULL`. New drivers should prefer this form.

24.3.6 The `null(4)` Reference

The cleanest small example of `cdevsw` and `make_dev_credf` in the tree is `/usr/src/sys/dev/null/null.c`. The relevant excerpts are short enough to read in one pass. The `cdevsw` declarations (a separate one for `/dev/null` and `/dev/zero`):

```

static struct cdevsw null_cdevsw = {
    .d_version = D_VERSION,
    .d_read    = (d_read_t *)nullop,
    .d_write   = null_write,
    .d_ioctl   = null_ioctl,
    .d_name    = "null",
};

```

The module event handler that creates and destroys the nodes:

```

static int
null_modevent(module_t mod, int type, void *data)
{
    switch (type) {
    case MOD_LOAD:
        full_dev = make_dev_credf(MAKEDEV_ETERNAL_KLD, &full_cdevsw, 0,
            NULL, UID_ROOT, GID_WHEEL, 0666, "full");
        null_dev = make_dev_credf(MAKEDEV_ETERNAL_KLD, &null_cdevsw, 0,
            NULL, UID_ROOT, GID_WHEEL, 0666, "null");
        zero_dev = make_dev_credf(MAKEDEV_ETERNAL_KLD, &zero_cdevsw, 0,
            NULL, UID_ROOT, GID_WHEEL, 0666, "zero");
        break;
    case MOD_UNLOAD:
        destroy_dev(full_dev);
    }
}

```

```

        destroy_dev(null_dev);
        destroy_dev(zero_dev);
        break;
    case MOD_SHUTDOWN:
        break;
    default:
        return (EOPNOTSUPP);
    }
    return (0);
}

DEV_MODULE(null, null_modevent, NULL);
MODULE_VERSION(null, 1);

```

A few details are worth pausing on. The `MAKEDEV_ETERNAL_KLD` flag tells the kernel that even if the module is unloaded under unusual circumstances, the `cdev` should not be silently invalidated. The `0666` mode means everyone can read and write; that is correct for `/dev/null`. The unit number is `0` because there is only ever one of each. The `MOD_LOAD` arm runs at module load and creates the nodes; the `MOD_UNLOAD` arm runs at module unload and destroys them; `MOD_SHUTDOWN` runs during the system shutdown sequence and does nothing here, because no shutdown work is needed for these pseudo-devices.

This is the canonical shape of a pseudo-device that lives at module scope rather than at device-tree scope. The `myfirst` driver, by contrast, creates its `cdev` in `device_attach` because the `cdev`'s lifetime is tied to a specific PCI device's lifetime, not to the module's lifetime. The two patterns are different but coexist comfortably.

24.3.7 Multiple Nodes and Clonable Nodes

A single driver can create more than one node. Three patterns are common.

The first is **fixed-name nodes**. The driver knows in advance how many nodes it needs and creates them with fixed names: `/dev/myfirst0`, `/dev/myfirst-status`, `/dev/myfirst-config`. This is the right pattern when each node has a different role and the count is known.

The second is **indexed nodes**. The driver creates one node per unit, named `/dev/myfirstN` where `N` is the unit number. This is the right pattern when each node represents a separate instance of the same kind of object, such as one per attached PCI card.

The third is **clonable nodes**. The driver registers a clone handler that creates a new node on demand whenever a user opens a name matching a pattern. The reader of `tun(4)` opens `/dev/tun` and gets `/dev/tun0`; opening `/dev/tun` again yields `/dev/tun1`; the kernel allocates the next free unit on each open. This is the right pattern for pseudo-devices that the user wants to “create” by simply opening them.

The clone mechanism is the `dev_clone` event handler. It lives in `/usr/src/sys/sys/conf.h`:

```

typedef void (*dev_clone_fn)(void *arg, struct ucred *cred, char *name,
    int namelen, struct cdev **result);

```

```
EVENTHANDLER_DECLARE(dev_clone, dev_clone_fn);
```

The driver registers a handler with `EVENTHANDLER_REGISTER`, and the kernel calls the handler whenever an open path under `/dev` does not match an existing node. The handler decides whether the name belongs to its driver, allocates a new unit if so, calls `make_dev` to create the node with that name, and stores the resulting `cdev` pointer through the `result` argument. The kernel then re-opens the freshly created node and continues with the user's open call.

The `tun(4)` driver shows this pattern. From `/usr/src/sys/net/if_tuntap.c`:

```
static eventhandler_tag clone_tag;

static int
tuntapmodevent(module_t mod, int type, void *data)
{
    switch (type) {
        case MOD_LOAD:
            clone_tag = EVENTHANDLER_REGISTER(dev_clone, tuncclone, 0, 1000);
            if (clone_tag == NULL)
                return (ENOMEM);
            ...
            break;
        case MOD_UNLOAD:
            EVENTHANDLER_DEREGISTER(dev_clone, clone_tag);
            ...
    }
}

static void
tuncclone(void *arg, struct ucred *cred, char *name, int namelen,
          struct cdev **dev)
{
    /* If *dev != NULL, another handler already created the cdev. */
    if (*dev != NULL)
        return;

    /* Examine name; if it matches our pattern, allocate a unit */
    /* and call make_dev to populate *dev. */
    ...
}
```

A few rules apply to clone handlers. The handler must not assume it is the only handler registered; multiple subsystems can register against `dev_clone`, and each handler must check whether `*dev` is already non-NULL

before doing any work. The handler runs in a context that can sleep, so it can call `make_dev` directly. The handler should validate the name carefully because it is supplied by user space.

The `myfirst` driver uses an indexed-node pattern at first (one node per attached PCI device, `/dev/myfirst0`, `/dev/myfirst1`, etc.), and Section 2's lab walks through adding a clone handler so the user can open `/dev/myfirst-clone` and get a new unit on demand. The clone pattern is most useful for pseudo-devices that have no underlying hardware; for hardware-backed drivers it is rarely needed.

24.3.8 Permissions, Ownership, and `devfs.rules`

`make_dev` takes the initial owner UID, group GID, and permissions for the node. These values are baked in at creation time. They are visible from user space through `ls -l /dev/myfirst0` and they determine which user-space programs can open the node.

For a hardware-backed driver, the right defaults depend on the role. A device that should only be accessible to root uses `UID_ROOT`, `GID_WHEEL`, `0600`. A device that any administrative user should be able to access uses `UID_ROOT`, `GID_OPERATOR`, `0660`. A device that any user can read but only root can write uses `UID_ROOT`, `GID_WHEEL`, `0644`. The `myfirst` driver uses `UID_ROOT`, `GID_WHEEL`, `0660` by default; the user is expected to be root or to be given access through `devfs.rules` if they need it.

The administrator can override these defaults at runtime through `devfs.rules(8)`. A typical rule file looks like:

```
[localrules=10]
add path 'myfirst*' mode 0660
add path 'myfirst*' group operator
```

The administrator activates the rules by adding to `/etc/rc.conf`:

```
devfs_system_ruleset="localrules"
```

After service `devfs restart` (or a reboot), the rules apply to new device nodes that appear. This mechanism lets the administrator grant access to the driver without rebuilding the module, which is the right division of responsibility: the developer chooses safe defaults, and the administrator widens them when needed.

A common mistake is to make a device world-writable by default because “it makes testing easier”. A driver that ships with `0666` permissions to a device that controls hardware is a security problem. The chapter's recommendation is `0660` with `GID_WHEEL` as the default, and to instruct readers in `INTEGRATION.md` how to use `devfs.rules` to change it if they need to.

24.3.9 Putting It All Together: The Stage 1 Driver

The Chapter 24 Stage 1 driver replaces the original ad-hoc node creation with the modern `make_dev_s` pattern, sets `si_drv1` at creation time, uses the `D_TRACKCLOSE` flag, and prepares the way for the `ioctl` callbacks to be added in Section 3. Here is the relevant excerpt of the new attach function. Type this in by hand; the whole point of the chapter is the careful change from the old ad-hoc form to the new disciplined form.

```

static int
myfirst_attach(device_t dev)
{
    struct myfirst_softc *sc;
    struct make_dev_args args;
    int error;

    sc = device_get_softc(dev);
    sc->sc_dev = dev;

    /* ... earlier attach work: PCI resources, MSI-X, DMA, sysctl tree
       * stub, debug subtree. See Chapters 18-23 for these. ... */

    /* Build the cdev for /dev/myfirstN. */
    make_dev_args_init(&args);
    args.mda_devsw = &myfirst_cdevsw;
    args.mda_uid = UID_ROOT;
    args.mda_gid = GID_WHEEL;
    args.mda_mode = 0660;
    args.mda_si_drv1 = sc;
    args.mda_unit = device_get_unit(dev);
    error = make_dev_s(&args, &sc->sc_cdev, "myfirst%d",
        device_get_unit(dev));
    if (error != 0) {
        device_printf(dev, "make_dev_s failed: %d\n", error);
        DPRINTF(sc, MYF_DBG_INIT, "cdev creation failed (%d)\n", error);
        goto fail;
    }
    DPRINTF(sc, MYF_DBG_INIT, "cdev created at /dev/myfirst%d\n",
        device_get_unit(dev));

    /* ... rest of attach: register callouts, finalise sysctl OIDs, etc. */

    return (0);

fail:
    /* Unwind earlier resources in reverse order. */
    /* See Section 7 for the discipline that goes here. */
    return (error);
}

```

The corresponding cdevsw with D_TRACKCLOSE:


```

static d_open_t myfirst_open;
static d_close_t myfirst_close;
static d_read_t myfirst_read;
static d_write_t myfirst_write;
static d_ioctl_t myfirst_ioctl;

static struct cdevsw myfirst_cdevsw = {
    .d_version = D_VERSION,
    .d_flags   = D_TRACKCLOSE,
    .d_name    = "myfirst",
    .d_open    = myfirst_open,
    .d_close   = myfirst_close,
    .d_read    = myfirst_read,
    .d_write   = myfirst_write,
    .d_ioctl   = myfirst_ioctl,
};

```

The corresponding open and close, with `si_drv1` lookups and per-open counter updates:

```

static int
myfirst_open(struct cdev *dev, int oflags, int devtype, struct thread *td)
{
    struct myfirst_softc *sc = dev->si_drv1;

    SDT_PROBE2(myfirst, , , open, sc, oflags);

    mtx_lock(&sc->sc_mtx);
    sc->sc_open_count++;
    mtx_unlock(&sc->sc_mtx);

    DPRINTF(sc, MYF_DBG_OPEN,
        "open: pid=%d flags=%#x open_count=%u\n",
        td->td_proc->p_pid, oflags, sc->sc_open_count);
    return (0);
}

static int
myfirst_close(struct cdev *dev, int fflags, int devtype, struct thread *td)
{
    struct myfirst_softc *sc = dev->si_drv1;

    SDT_PROBE2(myfirst, , , close, sc, fflags);

    mtx_lock(&sc->sc_mtx);

```

```

KASSERT(sc->sc_open_count > 0,
    ("myfirst_close: open_count underflow"));
sc->sc_open_count--;
mtx_unlock(&sc->sc_mtx);

DPRINTF(sc, MYF_DBG_OPEN,
    "close: pid=%d flags=0x%x open_count=%u\n",
    td->td_proc->p_pid, fflags, sc->sc_open_count);
return (0);
}

```

And the matching detach update, with `destroy_dev` before any softc state release:

```

static int
myfirst_detach(device_t dev)
{
    struct myfirst_softc *sc = device_get_softc(dev);

    /* Refuse detach while users still hold the device open. */
    mtx_lock(&sc->sc_mtx);
    if (sc->sc_open_count > 0) {
        mtx_unlock(&sc->sc_mtx);
        device_printf(dev, "detach refused: %u open(s) outstanding\n",
            sc->sc_open_count);
        return (EBUSY);
    }
    mtx_unlock(&sc->sc_mtx);

    /* Destroy the cdev first. The kernel drains any in-flight
     * callbacks before destroy_dev returns. */
    if (sc->sc_cdev != NULL) {
        destroy_dev(sc->sc_cdev);
        sc->sc_cdev = NULL;
        DPRINTF(sc, MYF_DBG_INIT, "cdev destroyed\n");
    }

    /* ... rest of detach: tear down DMA, MSI-X, callouts, sysctl ctx,
     * etc., in reverse order of attach. See Section 7. */

    return (0);
}

```

This is the Stage 1 milestone. The driver now has a clean, modern, beginner-friendly devfs entry. The next stages will add ioctls and sysctls on top of this foundation.

24.3.10 A Concrete Walkthrough: Loading and Inspecting the Stage 1 Driver

Build, load, and inspect the Stage 1 driver:

```
cd ~/myfirst-1.7-integration/stage1-devfs
make
sudo kldload ./myfirst.ko

# Confirm the device exists and has the expected attributes.
ls -l /dev/myfirst0

# Read its sysctl debug subtree (still from Chapter 23).
sysctl dev.myfirst.0

# Open it with cat to confirm the cdevsw read and close paths fire.
sudo cat /dev/myfirst0 > /dev/null
dmesg | tail -20
```

The expected output for `ls -l /dev/myfirst0`:

```
crw-rw----  1 root  wheel  0x71 Apr 19 16:30 /dev/myfirst0
```

The `crw-rw----` says character device, owner read+write, group read+write, no permission for others. The owner is root. The group is wheel. The minor number is `0x71` (the kernel's allocation; the value will differ between systems). The size is the unit number that `make_dev_s` was called with.

The expected output for the `dmesg` snippet:

```
myfirst0: cdev created at /dev/myfirst0
myfirst0: open: pid=4321 flags=0x1 open_count=1
myfirst0: close: pid=4321 flags=0x1 open_count=0
```

(Assuming the debug mask has `MYF_DBG_INIT` and `MYF_DBG_OPEN` set. If the mask is zero, the lines are silent; turn them on with `sysctl dev.myfirst.0.debug.mask=0xFFFFFFFF` from Chapter 23's debug discipline.)

If the open and close lines do not appear, check the debug mask. If the open line appears but not the close, you forgot `D_TRACKCLOSE` and the kernel is calling close on every fd close rather than only the last; either turn `D_TRACKCLOSE` on or expect multiple close lines per open. If the open count goes negative, you have a real bug: a close was called without a matching open.

24.3.11 Common Mistakes With devfs

Five mistakes account for most of the devfs problems first-time integrators encounter. Naming them up front saves a debugging session later.

The first mistake is **calling `make_dev` from an unsleepable context**. `make_dev` may sleep. If it is called from an interrupt handler, from a callout, from inside a spin lock, or from inside any context where sleeping is forbidden, the kernel will panic with `WITNESS` or with `INVARIANTS` complaining about a sleepable function in a non-sleepable context. The fix is either to call `make_dev` from `device_attach` or from a module event handler (both safe contexts), or to use `make_dev_s` with `MAKEDEV_NOWAIT` (which then can fail and the caller must handle the failure).

The second mistake is **forgetting to set `si_drv1`**. The `cdevsw` callbacks then dereference a `NULL` pointer and panic the kernel. The fix is to set `si_drv1` immediately after `make_dev` (or, better, to use `make_dev_s` and set `mda_si_drv1` in the args, which closes the race window between creation and assignment).

The third mistake is **calling `destroy_dev` after freeing the `softc`**. The `cdevsw` callbacks may still be in flight when `destroy_dev` is called; the kernel drains them before `destroy_dev` returns. If the `softc` is already freed, the callbacks dereference garbage. The fix is to call `destroy_dev` first, then free the `softc`, in that strict order.

The fourth mistake is **creating two `cdevs` with the same name**. The kernel checks for duplicate names and the second `make_dev` call panics or returns an error depending on the variant. The fix is to compose the name from the unit number, or to use a clone handler.

The fifth mistake is **not handling the case where the device is open at detach**. A naive driver simply calls `destroy_dev` and frees the `softc`, which works only if no user has the device open. The user holding `/dev/my-first0` open with `cat` defeats this. The fix is to refuse detach when `open_count > 0`, or to use the kernel's `dev_ref/dev_rel` machinery to coordinate. The chapter's pattern is the simple refusal (`return (EBUSY)`) because it gives the cleanest user-visible error.

24.3.12 Pitfalls Unique to Multi-Instance Drivers

Drivers that create more than one `cdev` per attached device, or one `cdev` per channel inside a multi-channel device, hit a few additional pitfalls.

The first is **leaking nodes on partial failure**. If the driver creates three `cdevs` and the third call fails, it must destroy the first two before returning. Section 7's cleanup-on-failure pattern is the canonical solution. The chapter's lab in this section walks through the pattern with a deliberate failure injection.

The second is **forgetting that each `cdev` needs its own `si_drv1`**. A driver that creates per-channel nodes typically wants `si_drv1` to point to the channel rather than to the `softc`. The `cdevsw` callbacks then walk back from the channel to the `softc` as needed. Mixing the two leads to channels stomping on each other's state.

The third is **race-free node visibility**. Between `make_dev` (or `make_dev_s`) returning and the driver finishing the rest of attach, a fast user can already open the node. The driver must be ready to handle opens that arrive before attach is fully done. The simplest pattern is to defer `make_dev` to the very end of attach, so the node is only visible after every other piece of state is ready. The `myfirst` driver follows this pattern.

These pitfalls do not arise often for single-`cdev` drivers like `myfirst`, but recognising them now means the reader will not be surprised when they show up in Chapter 27 (storage drivers can have many `cdevs`, one per LUN) or Chapter 28 (network drivers can have many `cdevs`, one per command channel)

24.3.13 Wrapping Up Section 2

Section 2 made the driver's `/dev` presence first-class. The `cdev` is now created with the modern `make_dev_s` pattern, the `cdevsw` is fully populated with `D_TRACKCLOSE` and a debug-friendly callback set, the per-`cdev` state is wired through `si_drv1`, and the detach path drains and destroys the `cdev` cleanly. The driver still does the same work as 1.6-`debug`, but it now exposes that work through a properly-constructed `/dev` node that an administrator can `chmod`, `chown`, monitor through `ls`, and reason about from outside the kernel.

In the next section, we turn to the second integration surface: the user-driven control interface that lets a program tell the driver what to do. The vocabulary is `ioctl(2)`, `_IO`, `_IOR`, `_IOW`, `_IOWR`, and a small public header that user-space programs include.

24.4 Section 3: Implementing `ioctl()` Support

24.4.1 What `ioctl(2)` Is, and Why Drivers Need It

The `read(2)` and `write(2)` system calls are excellent for moving streams of bytes between a user program and a driver. They are not, however, well suited to control. A `read` cannot ask the driver “what is your current state?” without overloading the meaning of the bytes returned. A `write` cannot ask the driver “please reset your statistics” without inventing a private command vocabulary inside the byte stream. The system call that fills this gap is `ioctl(2)`, the input/output control call.

`ioctl` is a side channel for commands. The signature in user space is straightforward: `int ioctl(int fd, unsigned long request, ...)`; The first argument is a file descriptor (in our case, an open `/dev/myfirst0`). The second is a numeric request code that tells the driver what to do. The third is an optional pointer to a structure that carries the parameters of the request, either inbound, outbound, or both. The kernel routes the call into the `cdevsw` of the `cdev` backing that file descriptor, into the function pointed at by `d_ioctl`. The driver looks at the request code, performs the corresponding action, and returns 0 on success or a positive `errno` on failure.

Almost every driver that exposes a control interface to user space uses `ioctl` for it. Disk drivers use `ioctl` to report sector size, partition tables, and dump targets. Tape drivers use `ioctl` for rewind, eject, and tension commands. Network drivers use `ioctl` for media changes (`SIOCIFLAGS`), MAC address updates, and bus probe commands. Sound drivers use `ioctl` for sample rate, channel count, and buffer size negotiation. The vocabulary is so universal that learning it once unlocks every category of driver in the tree.

For the `myfirst` driver, `ioctl` lets us add commands that have no clean expression as bytes. We can let the operator query the in-memory message length without having to read it. We can let the operator reset the message and the open counters without having to write a special sentinel. We can expose the driver's version number so user-space tools can detect the API they are talking to. Each of these is a one-line change for the operator and a half-page change for the driver, and each is a textbook fit for `ioctl`.

This section walks through the entire `ioctl` pipeline: the encoding of request codes, the kernel's automatic copyin and copyout, the design of a public header, the implementation of the dispatcher, the construction of a small user-space companion program, and the most common pitfalls. By the end of the section the driver will be at version 1.7-integration-stage-2 and will support four `ioctl` commands: `MYFIRSTIOC_GETVER`, `MYFIRSTIOC_GETMSG`, `MYFIRSTIOC_SETMSG`, and `MYFIRSTIOC_RESET`.

24.4.2 How `ioctl` Numbers Are Encoded

An `ioctl` request code is not an arbitrary integer. It is a packed 32-bit value that encodes four pieces of information in fixed bit fields, defined in `/usr/src/sys/sys/ioccom.h`. The header begins with a comment showing the layout, which is worth reading before going further.

```
/*
 * Ioctl's have the command encoded in the lower word, and the size of
 * any in or out parameters in the upper word. The high 3 bits of the
 * upper word are used to encode the in/out status of the parameter.
 *
 *      31 29 28                16 15                8 7                0
 *      +-----+-----+-----+-----+
 *      | I/O | Parameter Length | Command Group | Command |
 *      +-----+-----+-----+-----+
 */
```

The four fields are:

The **direction bits** (bits 29 to 31) tell the kernel whether the third argument to `ioctl` is purely outbound (`IOC_OUT`, the kernel will copy the result back to user space), purely inbound (`IOC_IN`, the kernel will copy the user data into the kernel before the dispatcher runs), bidirectional (`IOC_INOUT`, both directions), or absent (`IOC_VOID`, the request takes no data argument). The kernel uses these bits to decide what `copyin` and `copyout` to perform automatically. The driver itself never has to call `copyin` or `copyout` for a properly encoded `ioctl`.

The **parameter length** (bits 16 to 28) encodes the size in bytes of the structure passed as the third argument, capped at `IOCPARM_MAX` = 8192. The kernel uses this size to allocate a temporary kernel buffer, perform the appropriate `copyin` or `copyout`, and present the buffer to the dispatcher as the `caddr_t` data argument. A driver that needs to pass more than 8192 bytes through a single `ioctl` must either embed a pointer in a smaller structure (with the cost of doing its own `copyin`), or use a different mechanism such as `mmap` or `read`.

The **command group** (bits 8 to 15) is a single character that names the family of related `ioctls`. It is conventionally one of the printable ASCII letters and identifies the subsystem. 'd' is used by GEOM disk `ioctls` (`DIOCGMEDIASIZE`, `DIOCGSECTORSIZE`). 'i' is used by `if_ioctl` (`SIOCIFFLAGS`). 't' is used by terminal `ioctls` (`TIOCGPTN`). The reader should choose a letter that is not already taken by something the driver might coexist with. For the `myfirst` driver we will use 'M'.

The **command number** (bits 0 to 7) is a small integer that identifies the specific `ioctl` within the group. Numbering usually starts at 1 and increases monotonically as commands are added. Reusing a number is a backward-compatibility hazard, so a driver that retires a command should leave the number reserved rather than recycle it.

The macros in `ioccom.h` build these encodings for you. They are the only way to construct `ioctl` numbers correctly:

```
#define _IO(g,n)      _IOC(IOC_VOID, (g), (n), 0)
#define _IOR(g,n,t)   _IOC(IOC_OUT,  (g), (n), sizeof(t))
```

```
#define _IOW(g,n,t)      _IOC(IOC_IN,  (g), (n), sizeof(t))
#define _IOWR(g,n,t)     _IOC(IOC_INOUT,(g), (n), sizeof(t))
```

`_IO` declares a command that takes no argument. `_IOR` declares a command that returns a `t`-sized result to user space. `_IOW` declares a command that accepts a `t`-sized argument from user space. `_IOWR` declares a command that accepts a `t`-sized argument and writes a `t`-sized result back through the same buffer. The `t` is a type, not a pointer; the macros use `sizeof(t)` to compute the length field.

A few real-world examples make the pattern concrete. From `/usr/src/sys/sys/disk.h`:

```
#define DIOCGSECTORSIZE _IOR('d', 128, u_int)
#define DIOCGMEDIASIZE _IOR('d', 129, off_t)
```

These commands are read-only requests for the sector size (returned as `u_int`) and the media size (returned as `off_t`). The group letter `'d'` and the numbers 128 and 129 are reserved for the disk subsystem.

The driver itself never has to decode the bit layout. The command code is opaque to the dispatcher, which compares it to named constants:

```
switch (cmd) {
case MYFIRSTIOC_GETVER:
    ...
    break;
case MYFIRSTIOC_RESET:
    ...
    break;
default:
    return (ENOIOCTL);
}
```

The kernel uses the bit layout when it sets up the call (to allocate the buffer and perform copyin/copyout), and user space uses it implicitly through the macros. Between those two points, the request code is simply a label.

24.4.3 Choosing a Group Letter

The choice of a group letter matters because conflicts are silent. Two drivers that pick the same group letter and the same command number will see ioctl requests intended for the other driver if the operator confuses the two. The kernel does not enforce uniqueness across drivers, partly because no central authority assigns letters and partly because most letters are de facto reserved by tradition rather than registration.

A defensive approach is to follow these conventions:

Use **lowercase letters** (`'d'`, `'t'`, `'i'`) only when extending a well-known subsystem whose letter you already know. The lowercase letters are heavily used by base drivers and are easy to collide with.

Use **uppercase letters** (`'M'`, `'X'`, `'Q'`) for fresh drivers that need their own ioctl namespace. There are 26 uppercase letters and far fewer collisions in tree.

Avoid **digits** entirely. They are reserved by historical convention for early subsystems and a new driver that uses one will look out of place to a reviewer.

For the `myfirst` driver we use 'M'. It is the first letter of the driver name, it is uppercase (so it does not collide with any base subsystem), and it makes the request codes self-documenting in stack traces and in `kt race` output: a hex dump of an `ioctl` number with `0x4d` (the ASCII value of 'M') in the group field is unambiguously a `myfirst` command.

24.4.4 The `d_ioctl` Callback Signature

The dispatcher function pointed at by `cdevsw->d_ioctl` has the type `d_ioctl_t`, defined in `/usr/src/sys/sys/conf.h`. The signature is:

```
typedef int d_ioctl_t(struct cdev *dev, u_long cmd, caddr_t data,
                     int fflag, struct thread *td);
```

The five arguments deserve a slow read.

`dev` is the `cdev` that backs the file descriptor on which the user called `ioctl`. The driver uses `dev->si_drv1` to recover its softc. This is the same pattern used by every `cdevsw` callback we have already seen.

`cmd` is the request code, the value the user passed as the second argument to `ioctl`. The driver compares it to the named constants in its public header.

`data` is the kernel's local copy of the third argument. Because the kernel did the copyin and will do the copyout (for `_IOR`, `_IOW`, and `_IOWR` requests), `data` is always a kernel pointer. The driver dereferences it directly without calling `copyin`. For `_IO` requests, `data` is undefined and must not be dereferenced.

`fflag` is the file flags from the open call: `FREAD`, `FWRITE`, or both. The driver can use `fflag` to enforce read-only or write-only access for specific commands. A command that resets state, for example, might require `FWRITE` and return `EBADF` otherwise.

`td` is the calling thread. The driver can use `td` to extract the caller's credentials (`td->td_ucred`), to perform privilege checks (`priv_check_cred(td->td_ucred, PRIV_DRIVER, 0)`), or simply to log the caller's pid. For most commands, `td` is unused.

The return value is 0 on success, or a positive `errno` value on failure. The special value `ENOIOCTL` (defined in `/usr/src/sys/sys/errno.h`) tells the kernel that the driver does not recognise the command, and the kernel will then route the command through the file-system layer's generic `ioctl` handler. Returning `EINVAL` instead of `ENOIOCTL` for an unknown command is a subtle bug: it tells the kernel "I recognised the command but the arguments are wrong", which suppresses the generic fallback. Always use `ENOIOCTL` for the default case.

24.4.5 How the Kernel Performs Copyin and Copyout

Before the dispatcher runs, the kernel inspects the direction bits and parameter length encoded in `cmd`. If `IOC_IN` is set, the kernel reads the parameter length, allocates a temporary kernel buffer of that size, copies the user-space argument into it (`copyin`), and passes the kernel buffer as `data`. If `IOC_OUT` is set, the kernel allocates a buffer, calls the dispatcher with the (uninitialised) buffer as `data`, and on a 0 return, copies the buffer back

to user space (copyout). If both bits are set (IOC_INOUT), the kernel does both copyin and copyout around the dispatcher call. If neither is set (IOC_VOID), no buffer is allocated and data is undefined.

This automation has two consequences worth remembering.

First, the dispatcher writes to and reads from data using normal C dereference. The driver never calls copyin or copyout for a correctly encoded ioctl. This is one of the reasons a properly designed ioctl interface is so much simpler to implement than, say, a write-and-read protocol that fakes a control channel.

Second, the parameter type encoded in _IOW or _IOR must match what the dispatcher actually reads or writes. If the user-space header declares _IOR('M', 1, uint32_t) but the dispatcher writes a uint64_t into *(uint64_t *)data, the dispatcher will overrun the kernel's 4-byte buffer and corrupt adjacent stack memory, panicking the kernel under a WITNESS-enabled build and silently corrupting state under a production build. The header is the contract; the dispatcher must honour it byte for byte.

For ioctls with embedded pointers (a struct that contains a char *buf pointing at a separate buffer), the kernel cannot copyin or copyout the buffer because it is not part of the structure. The driver must do its own copyin and copyout for the buffer contents, while the kernel handles the wrapping struct. This pattern is needed for variable-length data and is covered in the pitfalls subsection below.

24.4.6 Designing a Public Header

A driver that exposes ioctls must publish a header that declares the request codes and the data structures. User-space programs include this header to construct correct calls. The header lives outside the kernel module: it is part of the driver's contract with user space and must be installable on the system (for example, into /usr/local/include/myfirst/myfirst_ioctl.h).

The convention is to place the header in the driver's source directory with a .h suffix that matches the public name. For the myfirst driver the header is myfirst_ioctl.h. Its responsibilities are narrow: declare the ioctl numbers, declare the structures used as ioctl arguments, declare any related constants (such as the maximum length of the message field), and nothing else. It must not include kernel-only headers, must not declare kernel-only types, and must compile cleanly when included by a user-space program.

Here is the full header for the chapter's stage 2 driver:

```
/*
 * myfirst_ioctl.h - public ioctl interface for the myfirst driver.
 *
 * This header is included by both the kernel module and any user-space
 * program that talks to the driver. Keep it self-contained: no kernel
 * headers, no kernel types, no inline functions that pull kernel state.
 */

#ifndef _MYFIRST_IOCTL_H_
#define _MYFIRST_IOCTL_H_

#include <sys/ioccom.h>
#include <sys/types.h>
```

```
/*
 * Maximum length of the in-driver message, including the trailing NUL.
 * The driver enforces this on SETMSG; user-space programs that build
 * larger buffers will see EINVAL.
 */
#define MYFIRST_MSG_MAX 256

/*
 * The interface version. Bumped when this header changes in a way that
 * is not backward-compatible. User-space programs should call
 * MYFIRSTIOC_GETVER first and refuse to operate on an unexpected
 * version.
 */
#define MYFIRST_IOCTL_VERSION 1

/*
 * MYFIRSTIOC_GETVER - return the driver's interface version.
 *
 *   ioctl(fd, MYFIRSTIOC_GETVER, &ver);    // ver = 1, 2, ...
 *
 * No FREAD or FWRITE flag is required.
 */
#define MYFIRSTIOC_GETVER _IOR('M', 1, uint32_t)

/*
 * MYFIRSTIOC_GETMSG - copy the current in-driver message into the
 * caller's buffer. The buffer must be MYFIRST_MSG_MAX bytes; the
 * message is NUL-terminated.
 */
#define MYFIRSTIOC_GETMSG _IOR('M', 2, char[MYFIRST_MSG_MAX])

/*
 * MYFIRSTIOC_SETMSG - replace the in-driver message. The buffer must
 * be MYFIRST_MSG_MAX bytes; the kernel takes the prefix up to the
 * first NUL or to MYFIRST_MSG_MAX - 1 bytes.
 *
 * Requires FWRITE on the file descriptor.
 */
#define MYFIRSTIOC_SETMSG _IOW('M', 3, char[MYFIRST_MSG_MAX])

/*
 * MYFIRSTIOC_RESET - reset all per-instance counters and clear the
 * message. Returns 0 on success.
 */
```

```

*
* Requires FWRITE on the file descriptor.
*/
#define MYFIRSTIOC_RESET    _IO('M', 4)

#endif /* _MYFIRST_IOCTL_H_ */

```

A few details in this header are worth pausing on.

The use of `uint32_t` and `sys/types.h` (rather than `u_int32_t` and `sys/cdefs.h`) keeps the header portable across the FreeBSD base and any program that follows POSIX. The kernel and user-space agree on the size of `uint32_t`, so the encoded length in the request code matches the dispatcher's view of the data.

The maximum message length, `MYFIRST_MSG_MAX = 256`, is well under `IOCPARM_MAX = 8192`, so the kernel will copyin and copyout the message without complaint. A driver that needed to move larger messages would either raise the limit (up to 8192) or switch to the embedded-pointer pattern.

The `MYFIRST_IOCTL_VERSION` constant gives user-space a way to detect API changes. The first ioctl that any program should issue is `MYFIRSTIOC_GETVER`; if the returned version is not what the program was compiled against, the program should refuse to issue further ioctls and print a clear error. This is standard practice for drivers that expect to evolve.

The argument type `char[MYFIRST_MSG_MAX]` is unusual but legal in `_IOR` and `_IOW`. The macro takes `sizeof(t)`, and `sizeof(char[256]) == 256`, so the encoded length is exactly the array size. This is the cleanest way to express a fixed-size buffer in a public ioctl header.

24.4.7 Implementing the Dispatcher

With the header in hand, the dispatcher is a switch statement that reads the command code, performs the action, and either returns 0 (success) or a positive `errno` (failure). The dispatcher lives in `myfirst_ioctl.c`, a new source file added to the driver in stage 2.

The complete dispatcher:

```

/*
 * myfirst_ioctl.c - ioctl dispatcher for the myfirst driver.
 *
 * The d_ioctl callback in myfirst_cdevsw points at myfirst_ioctl.
 * Per-command argument layout is documented in myfirst_ioctl.h, which
 * is shared with user space.
 */

#include <sys/param.h>
#include <sys/systm.h>
#include <sys/conf.h>
#include <sys/file.h>
#include <sys/lock.h>

```

```
#include <sys/mutex.h>
#include <sys/malloc.h>
#include <sys/proc.h>

#include "myfirst.h"
#include "myfirst_debug.h"
#include "myfirst_ioctl.h"

SDT_PROBE_DEFINE3(myfirst, , , ioctl,
    "struct myfirst_softc *", "u_long", "int");

int
myfirst_ioctl(struct cdev *dev, u_long cmd, caddr_t data, int fflag,
    struct thread *td)
{
    struct myfirst_softc *sc = dev->si_drv1;
    int error = 0;

    SDT_PROBE3(myfirst, , , ioctl, sc, cmd, fflag);
    DPRINTF(sc, MYF_DBG_IOCTL, "ioctl: cmd=0x%08lx fflag=0x%x\n",
        cmd, fflag);

    mtx_lock(&sc->sc_mtx);

    switch (cmd) {
    case MYFIRSTIOC_GETVER:
        *(uint32_t *)data = MYFIRST_IOCTL_VERSION;
        break;

    case MYFIRSTIOC_GETMSG:
        /*
         * Copy the current message into the caller's buffer.
         * The buffer is MYFIRST_MSG_MAX bytes; we always emit
         * a NUL-terminated string.
         */
        strncpy((char *)data, sc->sc_msg, MYFIRST_MSG_MAX);
        break;

    case MYFIRSTIOC_SETMSG:
        if ((fflag & FWRITE) == 0) {
            error = EBADF;
            break;
        }
        /*
```

```

        * The kernel has copied MYFIRST_MSG_MAX bytes into
        * data. Take the prefix up to the first NUL.
        */
        strncpy(sc->sc_msg, (const char *)data, MYFIRST_MSG_MAX);
        sc->sc_msglen = strlen(sc->sc_msg);
        DPRINTF(sc, MYF_DBG_IOCTL,
            "SETMSG: new message is %zu bytes\n", sc->sc_msglen);
        break;

case MYFIRSTIOC_RESET:
    if ((fflag & FWRITE) == 0) {
        error = EBADF;
        break;
    }
    sc->sc_open_count = 0;
    sc->sc_total_reads = 0;
    sc->sc_total_writes = 0;
    bzero(sc->sc_msg, sizeof(sc->sc_msg));
    sc->sc_msglen = 0;
    DPRINTF(sc, MYF_DBG_IOCTL,
        "RESET: counters and message cleared\n");
    break;

default:
    error = ENOIOCTL;
    break;
}

mtx_unlock(&sc->sc_mtx);
return (error);
}

```

Several disciplined choices are baked into this dispatcher and they are worth pointing out before the reader writes their own.

The dispatcher takes the softc mutex once at the top and releases it once at the bottom. Every command runs under the mutex. This prevents a read from racing a SETMSG (the read would otherwise see a half-replaced message buffer) and prevents two simultaneous RESET calls from corrupting the counters. The mutex is the same `sc->sc_mtx` introduced earlier in Part IV; we are simply extending its scope to cover ioctl serialisation.

The dispatcher's first action after taking the mutex is a single SDT probe and a single DPRINTF. Both report the command and the file flags. The SDT probe lets a DTrace script trace every ioctl in real time; the DPRINTF lets an operator turn on `MYF_DBG_IOCTL` and watch the same flow through `dmesg`. Both use the debug infrastructure introduced in Chapter 23 with no new machinery.

The `MYFIRSTIOC_SETMSG` and `MYFIRSTIOC_RESET` paths check `fflag & FWRITE` before they mutate state.

Without this check, a program that opened the device read-only could change the driver's state, which is a privilege-escalation pattern in some drivers. The check returns `EBADF` (bad file descriptor for the operation) rather than `EPERM` (no permission), because the failure is about the file's open flags rather than about the user's identity.

The default branch returns `ENOIOCTL`, never `EINVAL`. This is the rule from the previous subsection, repeated here because it is the single most common bug in homemade dispatchers.

The `strncpy` calls in `GETMSG` and `SETMSG` are the safe string copy primitive in the FreeBSD kernel. They guarantee NUL termination and never overrun the destination. The same calls would be `strncpy` in older code; `strncpy` is the modern preferred form and is what `style(9)` recommends.

24.4.8 The Softc Additions

Stage 2 extends the `softc` with two fields and confirms that the existing fields are still in use:

```
struct myfirst_softc {
    device_t      sc_dev;
    struct cdev    *sc_cdev;
    struct mtx     sc_mtx;

    /* From earlier chapters. */
    uint32_t      sc_debug;
    u_int         sc_open_count;
    u_int         sc_total_reads;
    u_int         sc_total_writes;

    /* New for stage 2. */
    char          sc_msg[MYFIRST_MSG_MAX];
    size_t        sc_msglen;
};
```

The message buffer is a fixed-size array sized to match the public header. Storing it inline (rather than as a pointer to a separately allocated buffer) keeps the lifetime simple: the buffer lives exactly as long as the `softc`. There is no `malloc` to track and no `free` to forget.

The initialisation in `myfirst_attach` becomes:

```
strncpy(sc->sc_msg, "Hello from myfirst", sizeof(sc->sc_msg));
sc->sc_msglen = strlen(sc->sc_msg);
```

The driver now has a default greeting that survives until the operator changes it through `SETMSG`, and survives unload/load cycles only by virtue of the new value being re-set on each load. (This is the same lifetime as every other `softc` field; persistence across reboots would require a `sysctl` tunable, which is the subject of Section 4.)

24.4.9 Wiring the Dispatcher Into cdevsw

The cdevsw declared in stage 1 already had a `.d_ioctl` slot waiting to be filled. Stage 2 fills it:

```
static struct cdevsw myfirst_cdevsw = {
    .d_version = D_VERSION,
    .d_flags   = D_TRACKCLOSE,
    .d_name    = "myfirst",
    .d_open    = myfirst_open,
    .d_close   = myfirst_close,
    .d_read    = myfirst_read,
    .d_write   = myfirst_write,
    .d_ioctl   = myfirst_ioctl,    /* new */
};
```

The kernel reads this table once, when the module loads. There is no runtime registration step; the cdevsw is part of the driver's static state.

24.4.10 Building the Stage 2 Driver

The Makefile for stage 2 must include the new source file:

```
KMOD=    myfirst
SRCS=    myfirst.c myfirst_debug.c myfirst_ioctl.c

CFLAGS+= -I${.CURDIR}

SYSDIR?= /usr/src/sys

.include <bsd.kmod.mk>
```

The build commands are unchanged from stage 1:

```
$ make
$ sudo kldload ./myfirst.ko
$ ls -l /dev/myfirst0
crw-rw----  1 root wheel 0x... <date> /dev/myfirst0
```

If the build fails because `myfirst_ioctl.h` is not found, check that the `CFLAGS` line includes `-I${.CURDIR}`. If the load fails because of an unresolved symbol such as `myfirst_ioctl`, check that `myfirst_ioctl.c` is listed in `SRCS` and that the function name matches the cdevsw entry.

24.4.11 The myfirstctl User-Space Companion

A driver with an ioctl interface needs a small companion program that exercises it. Without one, the operator has no way to call the ioctls except through a hand-written test or through `devctl(8)`'s ioctl pass-through, which is awkward for routine use.

The companion is `myfirstctl`, a single-file C program that takes a subcommand on the command line and calls the corresponding ioctl. It is intentionally small (under 200 lines) and depends only on the public header.

```
/*
 * myfirstctl.c - command-line front end to the myfirst driver's ioctls.
 *
 * Build:  cc -o myfirstctl myfirstctl.c
 * Usage:  myfirstctl get-version
 *         myfirstctl get-message
 *         myfirstctl set-message "<text>"
 *         myfirstctl reset
 */

#include <sys/types.h>
#include <sys/ioctl.h>

#include <err.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#include "myfirst_ioctl.h"

#define DEVPATH "/dev/myfirst0"

static void
usage(void)
{
    fprintf(stderr,
        "usage: myfirstctl get-version\n"
        "       myfirstctl get-message\n"
        "       myfirstctl set-message <text>\n"
        "       myfirstctl reset\n");
    exit(EX_USAGE);
}

int
main(int argc, char **argv)
```



```
{

    int fd, flags;
    const char *cmd;

    if (argc < 2)
        usage();
    cmd = argv[1];

    /*
     * SETMSG and RESET need write access; the others only need
     * read. Open the device with the right flags so the dispatcher
     * does not return EBADF.
     */
    if (strcmp(cmd, "set-message") == 0 ||
        strcmp(cmd, "reset") == 0)
        flags = O_RDWR;
    else
        flags = O_RDONLY;

    fd = open(DEVPATH, flags);
    if (fd < 0)
        err(EX_OSERR, "open %s", DEVPATH);

    if (strcmp(cmd, "get-version") == 0) {
        uint32_t ver;
        if (ioctl(fd, MYFIRSTIOC_GETVER, &ver) < 0)
            err(EX_OSERR, "MYFIRSTIOC_GETVER");
        printf("driver ioctl version: %u\n", ver);
    } else if (strcmp(cmd, "get-message") == 0) {
        char buf[MYFIRST_MSG_MAX];
        if (ioctl(fd, MYFIRSTIOC_GETMSG, buf) < 0)
            err(EX_OSERR, "MYFIRSTIOC_GETMSG");
        printf("%s\n", buf);
    } else if (strcmp(cmd, "set-message") == 0) {
        char buf[MYFIRST_MSG_MAX];
        if (argc < 3)
            usage();
        strlcpy(buf, argv[2], sizeof(buf));
        if (ioctl(fd, MYFIRSTIOC_SETMSG, buf) < 0)
            err(EX_OSERR, "MYFIRSTIOC_SETMSG");
    } else if (strcmp(cmd, "reset") == 0) {
        if (ioctl(fd, MYFIRSTIOC_RESET) < 0)
            err(EX_OSERR, "MYFIRSTIOC_RESET");
    } else {
```

```
        usage();  
    }  
  
    close(fd);  
    return (0);  
}
```

Two details deserve note.

The program opens the device with the minimum flags required for the requested operation. `MYFIRSTIOC_GETVER` and `MYFIRSTIOC_GETMSG` work fine with `O_RDONLY`, but `MYFIRSTIOC_SETMSG` and `MYFIRSTIOC_RESET` require `O_RDWR` because the dispatcher checks `fflag & FWRITE`. A user who runs `myfirstctl set-message foo` without the appropriate group membership will see `Permission denied` from `open`; one who has the membership but the dispatcher still rejects them would see `Bad file descriptor` from `ioctl`. Both errors are intelligible.

The `MYFIRSTIOC_RESET` call passes no third argument because the macro `_IO` (with no `R` or `W`) declares a void-data `ioctl`. The library's `ioctl(2)` is variadic, so calling it with two arguments is legal, but care is needed because an extra argument would be passed but ignored. The convention in this book is to call `_IO` `ioctls` with exactly two arguments to make the void-data nature clear in the source.

A typical session looks like this:

```
$ myfirstctl get-version  
driver ioctl version: 1  
$ myfirstctl get-message  
Hello from myfirst  
$ myfirstctl set-message "drivers are fun"  
$ myfirstctl get-message  
drivers are fun  
$ myfirstctl reset  
$ myfirstctl get-message  
  
$
```

After the reset, the message buffer is empty and `myfirstctl get-message` prints a blank line. The counters are also reset, which the next section's `sysctl` interface will let us verify directly.

24.4.12 Common Pitfalls With `ioctl`

The first pitfall is **type-size mismatch between header and dispatcher**. If the header declares `_IOR('M', 1, uint32_t)` but the dispatcher writes a `uint64_t` into `*(uint64_t *)data`, the kernel allocated a 4-byte buffer and the dispatcher writes 8 bytes into it. The 4 extra bytes corrupt whatever sat next to the buffer (often other `ioctl` arguments or the dispatcher's local stack frame). Under `WITNESS` and `INVARIANTS`, the kernel may catch the overrun and panic; under a production build, the result is silent corruption. The fix is to keep the header

and the dispatcher in lockstep, ideally by including the same header in both places (which the chapter's pattern does).

The second pitfall is **embedded pointers**. A struct that contains `char *buf; size_t len;` cannot be safely transferred through a single `_IOW`. The kernel will copy in the struct (the pointer and the length), but the buffer that the pointer points at is in user space and the dispatcher cannot dereference it directly. The dispatcher must call `copyin(uap->buf, kbuf, uap->len)` to transfer the buffer contents itself. Forgetting this step causes the dispatcher to read user-space memory through a kernel pointer, which the kernel's address-space protection will trap as a fault. The fix is either to inline the buffer in the struct (the pattern this chapter uses for the message field) or to add explicit `copyin/copyout` calls inside the dispatcher.

The third pitfall is **forgetting to handle `ENOIOCTL` correctly**. A driver that returns `EINVAL` for an unknown command suppresses the kernel's generic `ioctl` fallback. The user might see `Invalid argument` from a command that should have been silently passed up to the file system layer (such as `FIONBIO` for non-blocking I/O hint). The fix is to use `ENOIOCTL` as the default return.

The fourth pitfall is **changing the wire format of an existing `ioctl`**. Once a program is compiled against `MYFIRSTIOC_SETMSG` declared with a 256-byte buffer, recompiling the driver with a 512-byte buffer breaks the program: the encoded length in the request code changes, the kernel detects the mismatch (because the user passed a 256-byte buffer with the new 512-byte command), and the `ioctl` call returns `ENOTTY` ("Inappropriate `ioctl` for device"). The fix is to leave existing `ioctls` alone and to define new commands with new numbers when the format must change. The `MYFIRST_IOCTL_VERSION` constant lets user-space programs detect such evolution before they issue the affected calls.

The fifth pitfall is **doing slow work in the dispatcher while holding a mutex**. The dispatcher in this section holds `sc->sc_mtx` for the entire switch statement, which is fine because every command is fast (a `memcpy`, a counter reset, a `strncpy`). A real driver that needs to perform a hardware operation that may take milliseconds must drop the mutex first and re-acquire it after, or use a sleepable lock. Holding a non-sleepable mutex across a `tsleep` or `msleep` would panic the kernel.

24.4.13 Wrapping Up Section 3

Section 3 completed the second integration surface: a properly designed `ioctl` interface. The driver now exposes four commands through `MYFIRSTIOC_GETVER`, `MYFIRSTIOC_GETMSG`, `MYFIRSTIOC_SETMSG`, and `MYFIRSTIOC_RESET`. The interface is self-describing (any user-space program can call `MYFIRSTIOC_GETVER` to detect the API version), the encoding is explicit (the `_IOR/_IOW/_IO` macros from `/usr/src/sys/sys/ioc-com.h`), and the kernel handles the `copyin` and `copyout` automatically based on the bit layout. The companion `myfirstctl` program demonstrates how a user-space tool exercises the interface without ever touching the bytes of the request code itself.

The driver milestone for stage 2 is the addition of `myfirst_ioctl.c` and `myfirst_ioctl.h`, both of which integrate cleanly with the debug infrastructure from Chapter 23 (the `MYF_DBG_IOCTL` mask bit and the `myfirst:::ioctl` SDT probe). The `Makefile` grew one entry in `SRCS` and the `cdevsw` grew one filled-in callback. Everything else in the driver is unchanged.

In Section 4 we turn to the third integration surface: read-only and read-write knobs that an administrator can query and tune from the shell using `sysctl(8)`. Where `ioctl` is the right channel for a program that already has the device open, `sysctl` is the right channel for a script or an operator who wants to inspect or adjust driver state without opening anything. The two interfaces complement each other; most production drivers offer both.

24.5 Section 4: Exposing Metrics Through `sysctl()`

24.5.1 What `sysctl` Is, and Why Drivers Use It

`sysctl(8)` is the FreeBSD kernel's hierarchical name service. Every name in the tree maps to a piece of kernel state: a constant, a counter, a tunable variable, or a function pointer that produces a value on demand. The tree is rooted at `kern.`, `vm.`, `hw.`, `net.`, `dev.`, and a handful of other top-level prefixes. Any program with the appropriate privileges can read and (for writable nodes) modify these values through the `sysctl(3)` library, the `sysctl(8)` command-line tool, or the `sysctlbyname(3)` convenience interface.

For drivers, the relevant subtree is `dev.<driver_name>.<unit>.*`. The Newbus subsystem creates this prefix automatically for every attached device. A driver named `myfirst` with unit 0 attached gets the prefix `dev.myfirst.0` for free, with no driver code required. The driver's only job is to populate the prefix with named OIDs (object identifiers) for the values it wants to expose.

Why expose state through `sysctl` rather than `ioctl`? The two mechanisms answer different questions. `ioctl` is the right channel for a program that has already opened the device and wants to issue a command. `Sysctl` is the right channel for an operator at a shell prompt who wants to inspect or adjust state without opening anything. Most production drivers offer both: the `ioctl` interface for programs, and the `sysctl` interface for humans, scripts, and monitoring tools.

The common pattern is that `sysctl` exposes:

- **counters** that summarise the driver's activity since attach
- **read-only state** such as version numbers, hardware identifiers, and link state
- **read-write tunables** such as debug masks, queue depths, and timeout values
- **boot-time tunables** that are read from `/boot/loader.conf` before the driver attaches

By the end of this section, the `myfirst` driver will expose all four categories under `dev.myfirst.0` and will read its initial debug mask from `/boot/loader.conf`. The driver milestone for stage 3 is the addition of `myfirst_sysctl.c` and a small tree of OIDs.

24.5.2 The `Sysctl` Namespace

A complete `sysctl` name looks like a dotted path. The default Newbus prefix for our driver is:

```
dev.myfirst.0
```

Underneath this prefix the driver may add anything it likes. The `myfirst` driver in stage 3 will add:

```
dev.myfirst.0.%desc      "myfirst pseudo-device, integration version 1.7"
dev.myfirst.0.%driver    "myfirst"
dev.myfirst.0.%location  ""
dev.myfirst.0.%pnpinfo   ""
dev.myfirst.0.%parent    "nexus0"
dev.myfirst.0.version    "1.7-integration"
dev.myfirst.0.open_count 0
```

```

dev.myfirst.0.total_reads      0
dev.myfirst.0.total_writes    0
dev.myfirst.0.message         "Hello from myfirst"
dev.myfirst.0.debug.mask      0
dev.myfirst.0.debug.classes    "INIT(0x1) OPEN(0x2) IO(0x4) IOCTL(0x8) ..."

```

The first five names (the ones starting with %) are added by Newbus automatically and describe the device-tree relationship. The remaining names are the driver's contribution. Of these, `version`, `open_count`, `total_reads`, `total_writes`, and `debug.classes` are read-only; `message` and `debug.mask` are read-write. The `debug` subtree is itself a node, which means it can hold further OIDs as the driver grows.

The reader can already see the result on a system with `myfirst` loaded:

```

$ sysctl dev.myfirst.0
dev.myfirst.0.debug.classes: INIT(0x1) OPEN(0x2) IO(0x4) IOCTL(0x8) INTR(0x10)
↪ DMA(0x20) PWR(0x40) MEM(0x80)
dev.myfirst.0.debug.mask: 0
dev.myfirst.0.message: Hello from myfirst
dev.myfirst.0.total_writes: 0
dev.myfirst.0.total_reads: 0
dev.myfirst.0.open_count: 0
dev.myfirst.0.version: 1.7-integration
dev.myfirst.0.%parent: nexus0
dev.myfirst.0.%pnpinfo:
dev.myfirst.0.%location:
dev.myfirst.0.%driver: myfirst
dev.myfirst.0.%desc: myfirst pseudo-device, integration version 1.7

```

The order of the lines is the order of OID creation, reversed. (Newbus adds the %-prefixed names last, so they print first when `sysctl` walks the list backward. This is cosmetic and has no semantic meaning.)

24.5.3 Static OIDs Versus Dynamic OIDs

`Sysctl` OIDs come in two flavours.

A **static OID** is declared at compile time with one of the `SYSCTL_*` macros (`SYSCTL_INT`, `SYSCTL_STRING`, `SYSCTL_ULONG`, and so on). The macro generates a constant data structure that the linker glues into a special section, and the kernel assembles the section into the global tree at boot. Static OIDs are appropriate for system-wide values that exist for the lifetime of the kernel: timer ticks, scheduler statistics, and similar.

A **dynamic OID** is created at runtime with one of the `SYSCTL_ADD_*` functions (`SYSCTL_ADD_INT`, `SYSCTL_ADD_STRING`, `SYSCTL_ADD_PROC`, and so on). The function takes a context, a parent, a name, and a pointer to the underlying data, and inserts a new node into the tree. Dynamic OIDs are appropriate for per-instance values that come and go with a device: a driver creates them in `attach` and tears them down in `detach`.

Driver code uses dynamic OIDs almost exclusively. A driver does not exist at compile time of the kernel; it appears when the module loads, and any sysctl subtree it owns must be built up at attach time and disposed of at detach. The Newbus framework gives every driver a per-device sysctl context and a parent OID specifically for this purpose:

```
struct sysctl_ctx_list *ctx;
struct sysctl_oid *tree;
struct sysctl_oid_list *child;

ctx = device_get_sysctl_ctx(dev);
tree = device_get_sysctl_tree(dev);
child = SYSCTL_CHILDREN(tree);
```

`device_get_sysctl_ctx` returns the per-device context. The context tracks every OID the driver creates so the framework can free them all in one call when the driver detaches. The driver does not have to track them itself.

`device_get_sysctl_tree` returns the per-device tree node, which is the OID corresponding to `dev.<driver>.<unit>`. The tree was created by Newbus when the device was added.

`SYSCTL_CHILDREN(tree)` extracts the child list from the tree node. This is what the driver passes as the parent argument to subsequent `SYSCTL_ADD_*` calls.

With these three handles in hand, the driver can add any number of OIDs to its subtree:

```
SYSCTL_ADD_UINT(ctx, child, OID_AUTO, "open_count",
    CTLFLAG_RD, &sc->sc_open_count, 0,
    "Number of times the device has been opened");
```

The `SYSCTL_ADD_UINT` call adds an unsigned-int OID under the parent named `open_count`, with `CTLFLAG_RD` (read-only), backed by `&sc->sc_open_count`, with no special initial value, and with a description. The description is what `sysctl -d dev.myfirst.0.open_count` will print. Always write a useful description; an empty one is a documentation hole.

The matching call for a read-write integer is identical except for the flag:

```
SYSCTL_ADD_UINT(ctx, child, OID_AUTO, "debug_mask_simple",
    CTLFLAG_RW, &sc->sc_debug, 0,
    "Simple writable debug mask");
```

The `CTLFLAG_RW` flag tells the kernel to allow writes from privileged users (root, or processes with `PRIV_DRIVER`).

For a string, the macro is `SYSCTL_ADD_STRING`:

```
SYSCALL_ADD_STRING(ctx, child, OID_AUTO, "version",
    CTLFLAG_RD, sc->sc_version, 0,
    "Driver version string");
```

The third-from-last argument is a pointer to the buffer that holds the string, and the second-from-last is the buffer's size (zero means unbounded for read-only strings).

24.5.4 Handler-Backed OIDs

Some OIDs need more logic than a plain memory access. Reading the OID may need to compute the value from several softc fields; writing the OID may need to validate the new value and update related state. These OIDs use a handler function and the macro `SYSCALL_ADD_PROC`.

A handler has the signature:

```
static int handler(SYSCALL_HANDLER_ARGS);
```

`SYSCALL_HANDLER_ARGS` is a macro that expands to:

```
struct sysctl_oid *oidp, void *arg1, intptr_t arg2,
struct sysctl_req *req
```

`oidp` identifies the OID being accessed. `arg1` and `arg2` are the user-supplied arguments registered when the OID was created (typically `arg1` points at the softc and `arg2` is unused or holds a small constant). `req` carries the read/write context: `req->newptr` is non-NULL for a write (and points at the new value the user is supplying), and the handler must call `SYSCALL_OUT(req, value, sizeof(value))` to return a value on a read.

A typical handler that exposes a computed value:

```
static int
myfirst_sysctl_message_len(SYSCALL_HANDLER_ARGS)
{
    struct myfirst_softc *sc = arg1;
    u_int len;

    mtx_lock(&sc->sc_mtx);
    len = (u_int)sc->sc_msglen;
    mtx_unlock(&sc->sc_mtx);

    return (sysctl_handle_int(oidp, &len, 0, req));
}
```

The handler computes the value (here, by copying the message length under the mutex), then defers to `sysctl_handle_int`, which does the bookkeeping for the read or write and (for a write) calls back into the

handler with the new value already in `*ptr`. The handler-of-handlers pattern is idiomatic; using it correctly avoids reimplementing copyin and copyout for every typed handler.

The handler is registered with `SYSCTL_ADD_PROC`:

```
SYSCTL_ADD_PROC(ctx, child, OID_AUTO, "message_len",
    CTLTYPE_UINT | CTLFLAG_RD | CTLFLAG_MPSAFE,
    sc, 0, myfirst_sysctl_message_len, "IU",
    "Current length of the in-driver message");
```

Three arguments deserve attention. `CTLTYPE_UINT | CTLFLAG_RD | CTLFLAG_MPSAFE` is the type-and-flag word. `CTLTYPE_UINT` declares the OID's external type (unsigned int); `CTLFLAG_RD` declares it read-only; `CTLFLAG_MPSAFE` declares that the handler is safe to call without the giant lock. The `CTLFLAG_MPSAFE` flag is mandatory for new code; without it, the kernel still works but takes the giant lock around every read, which serialises the entire system on any sysctl access.

The seventh argument is the format string. "IU" declares an unsigned int (I for integer, U for unsigned). The full set is documented in `/usr/src/sys/sys/sysctl.h`: "I" for int, "IU" for uint, "L" for long, "LU" for ulong, "Q" for int64, "QU" for uint64, "A" for string, "S, structname" for an opaque struct. The `sysctl(8)` command uses the format string to decide how to print the value when invoked without `-x` (the raw hex flag).

24.5.5 The myfirst Sysctl Tree

The full sysctl tree for stage 3 is built in a single function, `myfirst_sysctl_attach`, called from `myfirst_attach` after the cdev has been created. The function is short enough to read end to end:

```
/*
 * myfirst_sysctl.c - sysctl tree for the myfirst driver.
 *
 * Builds dev.myfirst.<unit>.* with version, counters, message, and a
 * debug subtree (debug.mask, debug.classes).
 */

#include <sys/param.h>
#include <sys/system.h>
#include <sys/conf.h>
#include <sys/kernel.h>
#include <sys/lock.h>
#include <sys/malloc.h>
#include <sys/mutex.h>
#include <sys/sysctl.h>

#include "myfirst.h"
#include "myfirst_debug.h"
```



```
#define MYFIRST_VERSION "1.7-integration"

static int
myfirst_sysctl_message_len(SYSCTL_HANDLER_ARGS)
{
    struct myfirst_softc *sc = arg1;
    u_int len;

    mtx_lock(&sc->sc_mtx);
    len = (u_int)sc->sc_msglen;
    mtx_unlock(&sc->sc_mtx);

    return (sysctl_handle_int(oidp, &len, 0, req));
}

void
myfirst_sysctl_attach(struct myfirst_softc *sc)
{
    device_t dev = sc->sc_dev;
    struct sysctl_ctx_list *ctx;
    struct sysctl_oid *tree;
    struct sysctl_oid_list *child;
    struct sysctl_oid *debug_node;
    struct sysctl_oid_list *debug_child;

    ctx = device_get_sysctl_ctx(dev);
    tree = device_get_sysctl_tree(dev);
    child = SYSCTL_CHILDREN(tree);

    /* Read-only: driver version. */
    SYSCTL_ADD_STRING(ctx, child, OID_AUTO, "version",
        CTLFLAG_RD, MYFIRST_VERSION, 0,
        "Driver version string");

    /* Read-only: counters. */
    SYSCTL_ADD_UINT(ctx, child, OID_AUTO, "open_count",
        CTLFLAG_RD, &sc->sc_open_count, 0,
        "Number of currently open file descriptors");

    SYSCTL_ADD_UINT(ctx, child, OID_AUTO, "total_reads",
        CTLFLAG_RD, &sc->sc_total_reads, 0,
        "Total read() calls since attach");

    SYSCTL_ADD_UINT(ctx, child, OID_AUTO, "total_writes",
```

```

        CTLFLAG_RD, &sc->sc_total_writes, 0,
        "Total write() calls since attach");

/* Read-only: message buffer (no copy through user) */
SYSCTL_ADD_STRING(ctx, child, OID_AUTO, "message",
        CTLFLAG_RD, sc->sc_msg, sizeof(sc->sc_msg),
        "Current in-driver message");

/* Read-only handler: message length, computed. */
SYSCTL_ADD_PROC(ctx, child, OID_AUTO, "message_len",
        CTLTYPE_UINT | CTLFLAG_RD | CTLFLAG_MPSAFE,
        sc, 0, myfirst_sysctl_message_len, "IU",
        "Current length of the in-driver message in bytes");

/* Subtree: debug. */
debug_node = SYSCTL_ADD_NODE(ctx, child, OID_AUTO, "debug",
        CTLFLAG_RD | CTLFLAG_MPSAFE, NULL,
        "Debug controls and class enumeration");
debug_child = SYSCTL_CHILDREN(debug_node);

SYSCTL_ADD_UINT(ctx, debug_child, OID_AUTO, "mask",
        CTLFLAG_RW | CTLFLAG_TUN, &sc->sc_debug, 0,
        "Bitmask of enabled debug classes");

SYSCTL_ADD_STRING(ctx, debug_child, OID_AUTO, "classes",
        CTLFLAG_RD,
        "INIT(0x1) OPEN(0x2) IO(0x4) IOCTL(0x8) "
        "INTR(0x10) DMA(0x20) PWR(0x40) MEM(0x80)",
        0, "Names and bit values of debug classes");
}

```

Three details are worth a closer look.

The version OID is backed by a string constant (MYFIRST_VERSION), not a softc field. A read-only string OID can point at any stable buffer; the kernel never writes through the pointer. This is safer and simpler than carrying a per-softc copy of the version, and it lets the version be visible through `sysctl` even if the driver fails attach partway through.

The message OID points directly into the softc's `sc_msg` field with `CTLFLAG_RD`. A reader that calls `sysctl dev.myfirst.0.message` will get the current value. Because the OID is read-only, `sysctl` will not write to the buffer, so we do not need a write handler. (A read-write version of this OID would need a handler to validate the input; the read-write path runs through the `ioctl` interface in stage 2.)

The `debug.mask` OID has `CTLFLAG_RW | CTLFLAG_TUN`. The `RW` flag allows writes from a privileged user. The `TUN` flag tells the kernel to look for a matching tunable in `/boot/loader.conf` and apply it before the OID becomes accessible. (We will set up the `loader.conf` hook in the next subsection.)

24.5.6 Wiring Sysctl Into Attach and Detach

The attach path now calls the sysctl builder after the cdev is created:

```
static int
myfirst_attach(device_t dev)
{
    struct myfirst_softc *sc = device_get_softc(dev);
    struct make_dev_args args;
    int error;

    sc->sc_dev = dev;
    mtx_init(&sc->sc_mtx, "myfirst", NULL, MTX_DEF);
    strncpy(sc->sc_msg, "Hello from myfirst", sizeof(sc->sc_msg));
    sc->sc_msglen = strlen(sc->sc_msg);

    make_dev_args_init(&args);
    args.mda_devsw = &myfirst_cdevsw;
    args.mda_uid = UID_ROOT;
    args.mda_gid = GID_WHEEL;
    args.mda_mode = 0660;
    args.mda_si_drv1 = sc;
    args.mda_unit = device_get_unit(dev);

    error = make_dev_s(&args, &sc->sc_cdev,
        "myfirst%d", device_get_unit(dev));
    if (error != 0) {
        mtx_destroy(&sc->sc_mtx);
        return (error);
    }

    myfirst_sysctl_attach(sc);

    DPRINTF(sc, MYF_DBG_INIT, "attach: cdev created and sysctl tree built\n");
    return (0);
}
```

The detach path is unchanged: it does not have to call `myfirst_sysctl_detach`. The Newbus framework owns the per-device sysctl context and tears it down automatically when the device detaches. The driver only needs to clean up resources that it allocated outside the framework (the cdev and the mutex). This is one of the small but real reasons to prefer the per-device context over a private context.

24.5.7 Boot-Time Tunables Through `/boot/loader.conf`

A driver can let the operator configure its initial behaviour at boot time by reading values from the loader environment. The loader (`/boot/loader.efi` or `/boot/loader`) parses `/boot/loader.conf` before the kernel starts and exports the variables into a small environment that the kernel can query.

The simplest way to read a loader variable is `TUNABLE_INT_FETCH`:

```
TUNABLE_INT_FETCH("hw.myfirst.debug_mask_default", &sc->sc_debug);
```

The first argument is the loader variable name. The second is a pointer to the destination, which is also the default if the variable is absent. The call is silent if the variable is absent and writes the parsed value otherwise.

The call goes in `myfirst_attach` before `myfirst_sysctl_attach`. By the time the `sysctl` tree is built, `sc->sc_debug` already has the loader-supplied value (or the compile-time default), and the `dev.myfirst.0.debug.mask` OID reflects it.

A representative `/boot/loader.conf` entry for the driver looks like:

```
myfirst_load="YES"  
hw.myfirst.debug_mask_default="0x06"
```

The first line tells the loader to load `myfirst.ko` automatically. The second sets the default debug mask to `MYF_DBG_OPEN | MYF_DBG_IO`. After boot, `sysctl dev.myfirst.0.debug.mask` reports 6 and the operator can modify it at runtime without rebooting.

The naming convention is loose but follows a few practices. Keep loader variables under `hw.<driver>.<knob>` because the `hw.` namespace is conventionally read-only at runtime and is not subject to surprise renames. Use `default` in the variable name when the value is the initial value of a runtime-modifiable OID, to make the relationship clear. Document every loader variable in the driver's man page or in the chapter's reference card (the chapter has one at the end).

24.5.8 Combining Sysctl With the Debug Mask

The reader will recall from Chapter 23 that the driver already has a `sc->sc_debug` field and a `DPRINTF` macro that consults it. With stage 3 in place, the operator can now manipulate the mask from the shell:

```
$ sysctl dev.myfirst.0.debug.mask  
dev.myfirst.0.debug.mask: 0  
$ sysctl dev.myfirst.0.debug.classes  
dev.myfirst.0.debug.classes: INIT(0x1) OPEN(0x2) IO(0x4) IOCTL(0x8) ...  
$ sudo sysctl dev.myfirst.0.debug.mask=0xff  
dev.myfirst.0.debug.mask: 0 -> 255  
$ # now every DPRINTF call inside the driver will print
```

The `classes` OID exists precisely to spare the operator from having to memorise the bit values. `sysctl` prints both names together and the operator can copy a hex value off the screen and paste it into the next command.

The same mechanism extends to any other knob the driver may want to expose. A driver that has a tunable timeout would add:

```
SYSCTL_ADD_UINT(ctx, child, OID_AUTO, "timeout_ms",
    CTLFLAG_RW | CTLFLAG_TUN, &sc->sc_timeout_ms, 0,
    "Operation timeout in milliseconds");
```

A driver that wants per-instance enable/disable of a feature would add a `bool` (declared with `SYSCTL_ADD_BOOL`, which is the modern preferred type for boolean flags) or an `int` with two valid values (0 and 1).

24.5.9 Building the Stage 3 Driver

The Makefile for stage 3 lists the new source file:

```
KMOD=    myfirst
SRCS=    myfirst.c myfirst_debug.c myfirst_ioctl.c myfirst_sysctl.c

CFLAGS+= -I${.CURDIR}

SYSDIR?= /usr/src/sys

.include <bsd.kmod.mk>
```

After make and `kldload`, the operator can immediately walk the tree:

```
$ sudo kldload ./myfirst.ko
$ sysctl -a dev.myfirst.0
dev.myfirst.0.debug.classes: INIT(0x1) OPEN(0x2) IO(0x4) IOCTL(0x8) ...
dev.myfirst.0.debug.mask: 0
dev.myfirst.0.message_len: 18
dev.myfirst.0.message: Hello from myfirst
dev.myfirst.0.total_writes: 0
dev.myfirst.0.total_reads: 0
dev.myfirst.0.open_count: 0
dev.myfirst.0.version: 1.7-integration
dev.myfirst.0.%parent: nexus0
dev.myfirst.0.%pnpinfo:
dev.myfirst.0.%location:
dev.myfirst.0.%driver: myfirst
dev.myfirst.0.%desc: myfirst pseudo-device, integration version 1.7
```

Opening and reading the device should immediately bump the counters:

```
$ cat /dev/myfirst0
Hello from myfirst
$ sysctl dev.myfirst.0.total_reads
dev.myfirst.0.total_reads: 1
$ sysctl dev.myfirst.0.open_count
dev.myfirst.0.open_count: 0
```

The `open_count` shows zero because `cat` opens the device, reads, and immediately closes; by the time `sysctl` runs, the count has returned to zero. To see a non-zero value, hold the device open in another terminal:

```
# terminal 1
$ exec 3< /dev/myfirst0

# terminal 2
$ sysctl dev.myfirst.0.open_count
dev.myfirst.0.open_count: 1

# terminal 1
$ exec 3<&-

# terminal 2
$ sysctl dev.myfirst.0.open_count
dev.myfirst.0.open_count: 0
```

The shell's `exec 3< /dev/myfirst0` opens the device on file descriptor 3 and leaves it open until `exec 3<&-` closes it. This is a useful technique for inspecting any driver's open-count metric without writing a program.

24.5.10 Common Pitfalls With Sysctl

The first pitfall is **forgetting `CTLFLAG_MPSAFE`**. Without the flag, the kernel takes the giant lock around the OID's handler. For a read-only integer this is harmless; for a heavily accessed OID it serialises the entire kernel and is a latency disaster. Modern kernel code uses `CTLFLAG_MPSAFE` everywhere; the absence of the flag is a sign that the code predates the move to fine-grained locking and should be reviewed for correctness anyway.

The second pitfall is **using a static OID in driver code**. The `SYSCTL_INT` and `SYSCTL_STRING` macros (without the `_ADD_` prefix) declare static OIDs and place them in a special linker section that is processed at kernel boot. A loadable module that uses these macros will install the OIDs when the module loads, but the OIDs will reference per-instance fields that did not exist at compile time, leading to crashes the moment the operator reads them. The fix is to use the `SYSCTL_ADD_*` family for all driver OIDs.

The third pitfall is **leaking the per-driver context**. A driver that uses its own `sysctl_ctx_init` and `sysctl_ctx_free` (rather than the per-device context returned by `device_get_sysctl_ctx`) must remember to call `sysctl_ctx_free` in detach. Forgetting this leaks every OID the driver created and panics the kernel the next time the operator reads one. The fix is to use the per-device context (which the framework cleans up automatically) wherever possible.

The fourth pitfall is **putting per-instance state in a process-shared OID**. A driver that wants a tunable shared across all its instances might be tempted to put it under `kern.myfirst.foo` or under `dev.myfirst.foo`. The latter looks innocent but breaks: when the second instance attaches, Newbus tries to create `dev.myfirst.0.foo` and `dev.myfirst.1.foo`, and the existing `dev.myfirst.foo` (without the unit) is no longer in scope. The fix is to use either `hw.myfirst.<knob>` for shared tunables or per-instance OIDs for per-instance state, but not both with the same name.

The fifth pitfall is **changing the type of an OID**. An OID declared as `CTLTYPE_UINT` cannot have its type changed without invalidating any user-space program that called `sysctlbyname` against it. The kernel returns `EINVAL` if the user passes a buffer of the wrong size. The fix is to keep the type stable across releases; if a different type is needed, define a new OID name and deprecate the old one.

24.5.11 Wrapping Up Section 4

Section 4 added the third integration surface: the `sysctl` tree under `dev.myfirst.0`. The driver now exposes its version, counters, current message, debug mask, and class enumeration, all with descriptive help text and all built with the per-device `sysctl` context provided by Newbus. The debug mask can be set at boot time through `/boot/loader.conf` and tuned at runtime through `sysctl(8)`. A short stanza in `attach` builds the entire tree; `detach` does nothing because the framework cleans up automatically.

The driver milestone for stage 3 is the addition of `myfirst_sysctl.c` and a small extension to `myfirst_attach`. The `cdevsw`, the `ioctl` dispatcher, the debug infrastructure, and the rest of the driver are unchanged. The `sysctl` tree is purely additive.

In Section 5 we look at an optional but illustrative integration target: the network stack. Most drivers will never become network drivers, but understanding how a driver registers an `ifnet` and participates in the `if_*` API gives the reader an example of the pattern the kernel uses for every “subsystem with a registration interface.” If the driver is not a network driver, the reader can read Section 5 for context and skip directly to Section 7.

24.6 Section 5: Networking Integration (Optional)

24.6.1 Why This Section Is Optional

The `myfirst` driver is not a network driver and will not become one in this chapter. The `cdevsw`, `ioctl`, and `sysctl` interfaces we have built are sufficient for it. The reader who is following the chapter to integrate a non-network driver may safely skip ahead to Section 7 and lose nothing essential.

However, networking integration is a perfect illustration of a more general principle: many FreeBSD subsystems offer a registration interface that turns a driver into a participant in a larger framework. The pattern is the same whether the framework is networking, storage, USB, or sound: the driver allocates a framework-defined object, fills in callbacks, calls a registration function, and from that moment forward receives callbacks from the framework. Reading this section, even without writing a network driver, builds the intuition for every other framework integration in the book.

The chapter uses the network stack as its example for two reasons. First, it is the most widely understood framework, so the vocabulary (`ifnet`, `if_attach`, `bpf`) connects to user-visible commands such as `ifconfig(8)` and `tcpdump(8)`. Second, the network registration interface is small enough to walk through end to end without losing the reader. Section 6 then shows the same pattern applied to the CAM storage stack.

24.6.2 What an ifnet Is

ifnet is the network stack's per-interface object. It is the network counterpart to the cdev we worked with in Section 2. Just as a cdev represents one device node under /dev, an ifnet represents one network interface under ifconfig. Every line of ifconfig -a corresponds to one ifnet.

The ifnet is opaque from outside the network stack. Drivers see it through the if_t typedef and manipulate it through accessor functions (if_setflags, if_getmtu, if_settransmitfn). The opacity is deliberate: it lets the network stack evolve the ifnet internals without breaking every driver every release. New drivers should use the if_t API exclusively.

The lifecycle of an ifnet in a driver is:

1. **allocate** with if_alloc(IFT_<type>)
2. **name** with if_initname(ifp, "myif", unit)
3. **fill in callbacks** for ioctl, transmit, init, and similar
4. **attach** with if_attach(ifp), which makes the interface visible
5. **attach to BPF** with bpfattach(ifp, ...) so tcpdump can see traffic
6. ... interface lives, sees traffic, runs ioctls ...
7. **detach from BPF** with bpfdetach(ifp)
8. **detach** with if_detach(ifp), which removes it from the visible list
9. **free** with if_free(ifp)

The lifecycle mirrors the cdev lifecycle (allocate, name, attach, detach, free) almost exactly, which is not a coincidence; both the network stack and devfs evolved from the same registration-interface pattern.

24.6.3 A Walkthrough Using disc(4)

The simplest in-tree example of an ifnet driver is disc(4), the discard interface. disc(4) accepts packets and silently drops them; its driver code is therefore mostly the integration scaffolding, with no protocol logic to distract the reader. The full driver lives in /usr/src/sys/net/if_disc.c.

The relevant function is disc_clone_create, which is called whenever the operator runs ifconfig disc create:

```
static int
disc_clone_create(struct if_clone *ifc, int unit, caddr_t params)
{
    struct ifnet      *ifp;
    struct disc_softc *sc;

    sc = malloc(sizeof(struct disc_softc), M_DISC, M_WAITOK | M_ZERO);
    ifp = sc->sc_ifp = if_alloc(IFT_LOOP);
    ifp->if_softc = sc;
    if_initname(ifp, discname, unit);
    ifp->if_mtu = DSMTU;
    ifp->if_flags = IFF_LOOPBACK | IFF_MULTICAST;
```



```

    ifp->if_drv_flags = IFF_DRV_RUNNING;
    ifp->if_ioctl = discioctl;
    ifp->if_output = discoutput;
    ifp->if_hdrlen = 0;
    ifp->if_addrlen = 0;
    ifp->if_snd.ifq_maxlen = 20;
    if_attach(ifp);
    bpfattach(ifp, DLT_NULL, sizeof(u_int32_t));

    return (0);
}

```

Step by step:

`malloc` allocates the driver's softc with `M_WAITOK | M_ZERO`. The `waitok` flag is allowed because `clone-create` runs in a sleepable context. The zero flag initialises the structure to zero, which lets the driver assume any field it does not explicitly set is zero or `NULL`.

`if_alloc(IFT_LOOP)` allocates the `ifnet` from the network stack's pool. The `IFT_LOOP` argument identifies the interface type, which the stack uses for SNMP-style reporting and for some default behaviours. Other common types are `IFT_ETHER` (for Ethernet drivers) and `IFT_TUNNEL` (for tunnel pseudo-devices).

`if_initname` sets the user-visible name. `discname` is the string "disc", and `unit` is the unit number passed in by the cloning framework. Together they form `disc0`, `disc1`, and so on.

The next several lines fill in callbacks and per-interface data: the MTU, the flags, the `ioctl` handler (`discioctl`), the output function (`discoutput`), the maximum send queue length, and so on. This is the network equivalent of the `cdevsw` table from Section 2; the difference is that it is filled into a per-interface object rather than into a static table.

`if_attach(ifp)` makes the interface visible to user space. After this call returns, `ifconfig disc0` works, the interface shows up in `netstat -i`, and protocols can bind to it.

`bpfattach(ifp, DLT_NULL, ...)` attaches the interface to the BPF (Berkeley Packet Filter) machinery, which is what `tcpdump` reads. `DLT_NULL` declares the link-layer type as "no link layer", appropriate for a loopback. An Ethernet driver would call `bpfattach(ifp, DLT_EN10MB, ETHER_HDR_LEN)`. Without `bpfattach`, `tcpdump` cannot see the interface's traffic, even though the interface itself works.

The destruction path mirrors the creation path, in reverse:

```

static void
disc_clone_destroy(struct ifnet *ifp)
{
    struct disc_softc *sc;

    sc = ifp->if_softc;

    bpfdetach(ifp);
}

```

```
    if_detach(ifp);
    if_free(ifp);

    free(sc, M_DISC);
}
```

bpfdetach first, because tcpdump may have a reference. if_detach next, because the network stack may still be queuing traffic to the interface; if_detach drains the queue and removes the interface from the visible list. if_free last, because the ifnet may still be referenced by sockets that the upper layers have not yet finished cleaning up; if_free defers the actual free until the last reference goes away.

The disc(4) driver is roughly 200 lines. A real Ethernet driver is closer to 5000, but the integration boilerplate (allocate, initname, attach, bpfattach, detach, bpfdetach, free) is identical. The 4800 extra lines are protocol-specific details: descriptor rings, interrupt handlers, MAC address management, multicast filters, statistics, link-state polling, and so on. Each is its own pattern, and Chapter 28 covers them in detail. The integration framing covered here is the foundation under all of them.

24.6.4 How an Operator Sees the Result

Once disc_clone_create returns successfully, the operator can manipulate the interface from the shell:

```
$ sudo ifconfig disc create
$ ifconfig disc0
disc0: flags=8049<UP,LOOPBACK,RUNNING,MULTICAST> metric 0 mtu 1500
$ sudo ifconfig disc0 inet 169.254.99.99/32
$ sudo tcpdump -i disc0 &
$ ping -c1 169.254.99.99
... ping output ...
$ sudo ifconfig disc destroy
```

Each of these commands hits a different part of the integration:

- ifconfig disc create calls disc_clone_create, which builds the ifnet and attaches it.
- ifconfig disc0 reads the ifnet's flags and MTU through the if_t accessors.
- ifconfig disc0 inet 169.254.99.99/32 calls into discioctl with SIOCAIFADDR, the ioctl that adds an address.
- tcpdump -i disc0 opens the BPF tap that bpfattach created.
- ping -c1 sends a packet, which routes through discoutput, gets dropped, and never returns.
- ifconfig disc destroy calls disc_clone_destroy, which detaches and frees.

The whole integration is visible at the user-space level. None of the underlying protocol machinery had to change to accommodate the new driver; the network stack's framework already had a slot for it.

24.6.5 What This Pattern Generalises To

The same registration pattern applies to many other subsystems:

- The **sound stack** (`sys/dev/sound`) uses `pcm_register` and `pcm_unregister` to make a sound device visible. The driver fills in callbacks for buffer playback, mixer access, and channel configuration.
- The **USB stack** (`sys/dev/usb`) uses `usb_attach` and `usb_detach` to register USB device drivers. The driver fills in callbacks for transfer setup, control requests, and disconnection.
- The **GEOM I/O framework** (`sys/geom`) uses `g_attach` and `g_detach` to register storage providers and consumers. The driver fills in callbacks for I/O start, completion, and orphaning.
- The **CAM SIM framework** (`sys/cam`) uses `cam_sim_alloc` and `xpt_bus_register` to register storage adapters. Section 6 walks through this in more detail.
- The **kobj method dispatch system** (which we have already seen behind `device_method_t`) is itself a registration framework: the driver declares a methods table and the kobj subsystem dispatches calls through it.

In every case the steps are the same: allocate the framework's object, fill in callbacks, call the registration function, take traffic, and unregister cleanly. The vocabulary changes, but the rhythm does not.

24.6.6 Wrapping Up Section 5

Section 5 used the network stack to illustrate a registration-style integration. The driver allocates an `ifnet`, names it, fills in callbacks, attaches it to the stack, attaches it to BPF, takes traffic, and tears down in reverse order at destroy time. The pattern is small and well bounded; the protocol machinery sits behind it and is the subject of Chapter 27.

The reader who is not writing a network driver gets a useful payoff from this section even without applying it to `myfirst`: every other registration-style integration in the FreeBSD kernel follows the same shape. Once the rhythm of allocate-name-fill-attach-trafficdetach-free is internalised, the storage stack in Section 6 will look familiar at a glance.

In Section 6 we apply the same lens to the CAM storage stack. The vocabulary changes (`cam_sim`, `xpt_bus_register`, `xpt_action`, CCBs), but the registration shape is the same.

24.7 Section 6: CAM Storage Integration (Optional)

24.7.1 Why This Section Is Optional

`myfirst` is not a storage adapter and will not become one. The reader integrating a non-storage driver should skim this section for vocabulary, take note of the registration shape that mirrors Section 5, and continue to Section 7.

The reader who is integrating a storage adapter (a SCSI host bus adapter, an NVMe controller, an emulated virtual storage controller) will find here the bare bones of how CAM expects a driver to talk to it. The complete protocol surface is large enough to fill a chapter on its own and is the topic of Chapter 27; what we cover here is just the integration framing, identical in spirit to the `if_alloc / if_attach` framing we used for networking.

24.7.2 What CAM Is

CAM (Common Access Method) is FreeBSD's storage subsystem above the device-driver layer. It owns the queue of pending I/O requests, the abstract notion of a target and a logical unit number (LUN), the path-routing

logic that sends a request to the right adapter, and the set of generic peripheral drivers (da(4) for disks, cd(4) for optical, sa(4) for tape) that turn block I/O into protocol-specific commands. The driver sits underneath CAM and is responsible only for the adapter-specific work of sending commands to the hardware and reporting completions.

The vocabulary CAM uses is small but specific:

- A **SIM** (SCSI Interface Module) is the framework's view of a storage adapter. The driver allocates one with `cam_sim_alloc`, fills in a callback (the action function), and registers it with `xpt_bus_register`. The SIM is the analogue of an `ifnet` for the storage stack.
- A **CCB** (CAM Control Block) is a single I/O request. CAM hands a CCB to the driver through the action callback; the driver inspects the CCB's `func_code`, performs the requested action, fills in the result, and returns the CCB to CAM with `xpt_done`. CCBs are the analogue of `mbufs` for the storage stack, with the difference that a CCB carries both the request and the response.
- A **path** identifies a destination as a (bus, target, LUN) triple. The driver calls `xpt_create_path` to build a path it can use for asynchronous events.
- The **XPT** (Transport Layer) is the central CAM dispatch mechanism. The driver calls `xpt_action` to send a CCB to CAM (or to itself, for self-targeted actions); CAM eventually calls back into the driver's action function for I/O CCBs targeted at the driver's bus.

24.7.3 The Registration Lifecycle

For a single-channel adapter the registration steps are:

1. Allocate a CAM device queue with `cam_simq_alloc(maxq)`.
2. Allocate a SIM with `cam_sim_alloc(action, poll, "name", softc, unit, mtx, max_tagged, max_dev_transactions, devq)`.
3. Lock the driver's mutex.
4. Register the SIM with `xpt_bus_register(sim, dev, 0)`.
5. Create a path that the driver can use for events: `xpt_create_path(&path, NULL, cam_sim_path(sim), CAM_TARGET_WILDCARD, CAM_LUN_WILDCARD)`.
6. Unlock the mutex.

The cleanup runs in reverse order:

1. Lock the driver's mutex.
2. Free the path with `xpt_free_path(path)`.
3. Deregister the SIM with `xpt_bus_deregister(cam_sim_path(sim))`.
4. Free the SIM with `cam_sim_free(sim, TRUE)`. The `TRUE` argument tells CAM to free the underlying `devq` as well; pass `FALSE` if the driver wants to retain the `devq` for reuse.
5. Unlock the mutex.

The `ahci(4)` driver in `/usr/src/sys/dev/ahci/ahci.c` is a good real-world example. Its channel attach path includes the canonical sequence:

```
ch->sim = cam_sim_alloc(ahciaction, ahcipoll, "ahcich", ch,
    device_get_unit(dev), (struct mtx *)&ch->mtx,
```

```

    (ch->quirks & AHCI_Q_NOCCS) ? 1 : min(2, ch->numslots),
    (ch->caps & AHCI_CAP_SNCQ) ? ch->numslots : 0,
    devq);
if (ch->sim == NULL) {
    cam_simq_free(devq);
    device_printf(dev, "unable to allocate sim\n");
    error = ENOMEM;
    goto err1;
}
if (xpt_bus_register(ch->sim, dev, 0) != CAM_SUCCESS) {
    device_printf(dev, "unable to register xpt bus\n");
    error = ENXIO;
    goto err2;
}
if (xpt_create_path(&ch->path, NULL, cam_sim_path(ch->sim),
    CAM_TARGET_WILDCARD, CAM_LUN_WILDCARD) != CAM_REQ_CMP) {
    device_printf(dev, "unable to create path\n");
    error = ENXIO;
    goto err3;
}

```

The goto labels (err1, err2, err3) feed into a single cleanup section that unwinds whatever has been allocated so far. This is the standard FreeBSD-driver pattern for failure handling and is exactly the discipline Section 7 will codify.

24.7.4 The Action Callback

The action callback is the heart of a CAM driver. Its signature is `void action(struct cam_sim *sim, union ccb *ccb)`. The driver inspects `ccb->ccb_h.func_code` and dispatches:

```

static void
mydriver_action(struct cam_sim *sim, union ccb *ccb)
{
    struct mydriver_softc *sc;

    sc = cam_sim_softc(sim);

    switch (ccb->ccb_h.func_code) {
    case XPT SCSI IO:
        mydriver_start_io(sc, ccb);
        /* completion is asynchronous; xpt_done called later */
        return;

    case XPT_RESET_BUS:

```

```

        mydriver_reset_bus(sc);
        ccb->ccb_h.status = CAM_REQ_CMP;
        break;

    case XPT_PATH_INQ: {
        struct ccb_pathinq *cpi = &ccb->cpi;

        cpi->version_num = 1;
        cpi->hba_inquiry = PI_SDTR_ABLE | PI_TAG_ABLE;
        cpi->target_sprt = 0;
        cpi->hba_misc = PIM_NOBUSRESET | PIM_SEQSCAN;
        cpi->hba_eng_cnt = 0;
        cpi->max_target = 0;
        cpi->max_lun = 7;
        cpi->initiator_id = 7;
        strncpy(cpi->sim_vid, "FreeBSD", SIM_IDLEN);
        strncpy(cpi->hba_vid, "MyDriver", HBA_IDLEN);
        strncpy(cpi->dev_name, cam_sim_name(sim), DEV_IDLEN);
        cpi->unit_number = cam_sim_unit(sim);
        cpi->bus_id = cam_sim_bus(sim);
        cpi->ccb_h.status = CAM_REQ_CMP;
        break;
    }

    default:
        ccb->ccb_h.status = CAM_REQ_INVALID;
        break;
}

xpt_done(ccb);
}

```

Three branches illustrate the patterns:

`XPT SCSI IO` is the data path. The driver kicks off an asynchronous I/O (writing descriptors to the hardware, programming DMA, and so on) and returns immediately without calling `xpt_done`. The hardware completes the I/O some milliseconds later, raises an interrupt, the interrupt handler computes the result, fills in the CCB's status, and only then calls `xpt_done`. CAM does not require synchronous completion; the driver may take as long as the hardware takes.

`XPT RESET BUS` is a synchronous control. The driver performs the reset, sets `CAM_REQ_CMP`, and falls through to `xpt_done`. There is no asynchronous component.

`XPT PATH INQ` is the SIM's self-description. The first time CAM probes the SIM it issues `XPT_PATH_INQ` and reads back the bus characteristics: maximum LUN, supported flags, vendor identifiers, and so on. The driver fills in the structure and returns. Without a correct `XPT_PATH_INQ` response, CAM cannot probe targets behind

the SIM and the driver appears registered but inert.

The default branch returns `CAM_REQ_INVALID` for any function code the driver does not implement. CAM is forgiving about this; it simply treats the request as unsupported and either falls back to a generic implementation or surfaces the error to the peripheral driver.

24.7.5 How an Operator Sees the Result

Once a CAM-bearing driver has called `xpt_bus_register`, CAM probes the bus and the user-visible result is one or more entries in `camcontrol devlist`:

```
$ camcontrol devlist
<MyDriver Volume 1.0>          at scbus0 target 0 lun 0 (pass0,da0)
$ ls /dev/da0
/dev/da0
$ diskinfo /dev/da0
/dev/da0    512 ... ..
```

The `da0` device under `/dev` is a CAM peripheral driver (`da(4)`) wrapping the LUN that CAM discovered behind the SIM. The operator never deals with the SIM directly; they see only the standard `/dev/daN` interface that every block device uses. This is what makes CAM such a productive integration target: write a SIM, get full disk-style I/O for free.

24.7.6 Pattern Recognition

By now the reader should see the same shape we saw in Section 5:

Step	Networking	CAM
Allocate object	<code>if_alloc</code>	<code>cam_sim_alloc</code>
Name and configure	<code>if_initname</code> , set callbacks	implicit in <code>cam_sim_alloc</code> arguments
Attach to framework	<code>if_attach</code>	<code>xpt_bus_register</code>
Make discoverable	<code>bpfattach</code>	<code>xpt_create_path</code>
Take traffic	<code>if_output</code> callback	action callback
Complete an op	(synchronous)	<code>xpt_done(ccb)</code>
Detach	<code>bpfdetach</code> , <code>if_detach</code>	<code>xpt_free_path</code> , <code>xpt_bus_deregister</code>
Free	<code>if_free</code>	<code>cam_sim_free</code>

Other registration interfaces (`pcm_register` for sound, `usb_attach` for USB, `g_attach` for GEOM) follow the same column structure with their own vocabulary. Once the reader sees this table once, every subsequent integration is a matter of looking up the names.

24.7.7 Wrapping Up Section 6

Section 6 sketched the registration interface for a CAM SIM. The driver allocates a SIM with `cam_sim_alloc`, registers it with `xpt_bus_register`, creates a path for events, takes I/O through the action callback, completes I/O with `xpt_done`, and unregisters in reverse order at detach. The same registration-style integration pattern that we saw with `ifnet` applies, with the obvious change of vocabulary.

The reader has now seen three integration surfaces (`devfs`, `ioctl`, `sysctl`) that almost every driver needs and two registration-style surfaces (network, CAM) that some drivers need. In Section 7 we step back and codify the lifecycle discipline that holds it all together: the order of registration in attach, the order of teardown in detach, and the small set of patterns that distinguish a driver that loads, runs, and unloads cleanly from one that leaks resources or panics on detach.

24.8 Section 7: Registration, Teardown, and Cleanup Discipline

24.8.1 The Cardinal Rule

A driver that integrates with the kernel through several frameworks (`devfs` for `/dev`, `sysctl` for tunables, `ifnet` for networking, CAM for storage, callouts for timers, taskqueues for deferred work, and so on) accumulates a small zoo of allocated objects and registered callbacks. Every one of these has the same property: it must be released in the reverse order in which it was created. Forgetting this turns clean detach into a kernel panic, leaks resources at module unload, and scatters dangling pointers through subsystems the driver no longer owns.

The cardinal rule of integration is therefore very simple to state, even if applying it cleanly requires care:

Every successful registration must be paired with a deregistration. The order of deregistration is the reverse of the order of registration. A failed registration must trigger the deregistration of every preceding successful one before the function returns the failure.

That single sentence describes the entire lifecycle discipline. The rest of this section is a guided tour of how to apply it.

24.8.2 Why Reverse Order

The reverse-order rule sounds arbitrary; it is not. Each registration is a promise to the framework that “from now until I call deregister, you may call back into me, depend on my state, or hand me work.” A framework that has a callback into the driver and that holds work for it cannot safely be torn down while another framework still has access to the same state.

For example, suppose the driver registers a callout, then a `cdev`, then a `sysctl` OID. The `cdev`’s read callback may consult a value that the callout updates; the callout, in turn, may read state that the `sysctl` OID exposes. If detach tears down the callout first, then while the `cdev` is being torn down, a read from user space could try to consult a value that the callout was supposed to keep refreshed; the value is now stale and the read returns nonsense. If detach tears down the `cdev` first, then there is no way for read to come in any more, and the callout can be safely cancelled. The order matters.

The general rule is: tear down whatever can call into you, before tearing down what it depends on.

For most drivers the dependency chain is the same as the creation order:

- The cdev depends on the softc (the cdev's callbacks dereference `si_drv1`).
- The sysctl OIDs depend on the softc (they point at softc fields).
- The callouts and taskqueues depend on the softc (they receive a softc pointer as their argument).
- The interrupt handlers depend on the softc, the locks, and any DMA tags.
- The DMA tags and bus resources depend on the device.

If the driver creates these in this order, it should destroy them in the exact reverse order: interrupts first (they can fire at any time), then callouts and taskqueues (they execute at any time), then cdevs (they take user-space calls), then sysctl OIDs (the framework cleans these up automatically), then DMA, then bus resources, then locks. The softc itself is the last thing to free.

24.8.3 The `goto err1` Pattern in Attach

The hardest place to apply the rule is in attach, when a partial failure can leave the driver halfway initialised. The canonical FreeBSD pattern is the chain of `goto` labels, each of which represents the cleanup needed up to that point:

```
static int
myfirst_attach(device_t dev)
{
    struct myfirst_softc *sc = device_get_softc(dev);
    struct make_dev_args args;
    int error;

    sc->sc_dev = dev;
    mtx_init(&sc->sc_mtx, "myfirst", NULL, MTX_DEF);
    strncpy(sc->sc_msg, "Hello from myfirst", sizeof(sc->sc_msg));
    sc->sc_msglen = strlen(sc->sc_msg);

    TUNABLE_INT_FETCH("hw.myfirst.debug_mask_default", &sc->sc_debug);

    make_dev_args_init(&args);
    args.mda_devsw = &myfirst_cdevsw;
    args.mda_uid = UID_ROOT;
    args.mda_gid = GID_WHEEL;
    args.mda_mode = 0660;
    args.mda_si_drv1 = sc;
    args.mda_unit = device_get_unit(dev);

    error = make_dev_s(&args, &sc->sc_cdev,
        "myfirst%d", device_get_unit(dev));
    if (error != 0)
        goto fail_mtx;
```

```

    myfirst_sysctl_attach(sc);

    DPRINTF(sc, MYF_DBG_INIT, "attach: stage 3 complete\n");
    return (0);

fail_mtx:
    mtx_destroy(&sc->sc_mtx);
    return (error);
}

```

The single error label here exists because there is only one point at which a real failure can occur (the `make_dev_s` call). A more elaborate driver would have one label per registration step. By convention each label is named for the step that failed (`fail_mtx`, `fail_cdev`, `fail_sysctl`), and each label runs the cleanup for every step **above** it in the function. The label that handles the last possible failure is the longest cleanup; the label that handles the first failure is the shortest.

A four-stage attach for a hypothetical hardware driver would look like:

```

static int
mydriver_attach(device_t dev)
{
    struct mydriver_softc *sc = device_get_softc(dev);
    int error;

    mtx_init(&sc->sc_mtx, "mydriver", NULL, MTX_DEF);

    error = bus_alloc_resource_any(...);
    if (error != 0)
        goto fail_mtx;

    error = bus_setup_intr(...);
    if (error != 0)
        goto fail_resource;

    error = make_dev_s(...);
    if (error != 0)
        goto fail_intr;

    return (0);

fail_intr:
    bus_teardown_intr(...);
fail_resource:
    bus_release_resource(...);
}

```

```
fail_mtx:
    mtx_destroy(&sc->sc_mtx);
    return (error);
}
```

The labels read top-to-bottom in the same order as the cleanup actions execute. A failure at any step jumps to the matching label and falls through the cleanup labels for every preceding successful step. The pattern is so common that reading driver code without it is jarring; reviewers expect to see it.

24.8.4 The Detach Mirror

Detach should be the exact mirror of a successful attach. Every registration done in attach must have a matching deregistration in detach, in reverse order:

```
static int
myfirst_detach(device_t dev)
{
    struct myfirst_softc *sc = device_get_softc(dev);

    mtx_lock(&sc->sc_mtx);
    if (sc->sc_open_count > 0) {
        mtx_unlock(&sc->sc_mtx);
        return (EBUSY);
    }
    mtx_unlock(&sc->sc_mtx);

    DPRINTF(sc, MYF_DBG_INIT, "detach: tearing down\n");

    /*
     * destroy_dev drains any in-flight cdevsw callbacks. After
     * this call returns, no new open/close/read/write/ioctl can
     * arrive, and no in-flight callback is still running.
     */
    destroy_dev(sc->sc_cdev);

    /*
     * The per-device sysctl context is torn down automatically by
     * the framework after detach returns successfully. Nothing to
     * do here.
     */

    mtx_destroy(&sc->sc_mtx);
    return (0);
}
```

The detach starts by checking `open_count` under the mutex; if anyone holds the device open, detach refuses (returning `EBUSY`) so the operator gets a clear error rather than a panic. After the check, the function tears down whatever attach allocated, in reverse order: `cdev` first, then `sysctl` (automatic), then mutex.

The early `EBUSY` return is the “soft” detach pattern. It puts the responsibility for closing the device on the operator: `kldunload myfirst` will fail until the operator runs `kill cat` (or whatever else is holding the device open). The alternative is the “hard” pattern of refusing detach only if a critical resource is in use and accepting that ordinary file descriptors are a kernel responsibility to drain. The hard pattern is more involved (it usually requires `dev_ref` and `dev_rel`) and is left as a topic for Chapter 27’s CAM-driver section.

24.8.5 Module Event Handlers

Up to this point we have discussed attach and detach, the per-device lifecycle hooks Newbus calls when a driver instance is added or removed. There is also a per-module lifecycle, controlled by the function registered through `DRIVER_MODULE` (or `MODULE_VERSION` plus a `DECLARE_MODULE`). The kernel calls this function on `MOD_LOAD`, `MOD_UNLOAD`, and `MOD_SHUTDOWN`.

For most drivers the per-module hook is unused. `DRIVER_MODULE` takes a `NULL` event handler by default, and the kernel does the right thing: on `MOD_LOAD` it adds the driver to the bus’s driver list, and on `MOD_UNLOAD` it walks the bus and detaches every instance. The driver author writes only attach and detach.

Some drivers do need a module-level hook, however. The classic case is a driver that has to set up a global resource (a global hash table, a global mutex, a global event handler) that is shared across all instances. The hook for that is:

```
static int
myfirst_modevent(module_t mod, int what, void *arg)
{
    switch (what) {
        case MOD_LOAD:
            /* allocate global state */
            return (0);
        case MOD_UNLOAD:
            /* free global state */
            return (0);
        case MOD_SHUTDOWN:
            /* about to power off; flush anything important */
            return (0);
        default:
            return (EOPNOTSUPP);
    }
}

static moduledata_t myfirst_mod = {
    "myfirst", myfirst_modevent, NULL
};
```

```
DECLARE_MODULE(myfirst, myfirst_mod, SI_SUB_DRIVERS, SI_ORDER_ANY);  
MODULE_VERSION(myfirst, 1);
```

The `myfirst` driver in this chapter does not have global state and therefore does not need a `modevent`. The default `DRIVER_MODULE` machinery is sufficient. We mention the hook here so the reader can recognise it in larger drivers.

24.8.6 EVENTHANDLER for System Events

Some drivers care about events that happen elsewhere in the kernel: a process is forking, the system is shutting down, the network is changing state, and so on. The `EVENTHANDLER` mechanism lets a driver register a callback for a named event:

```
static eventhandler_tag myfirst_eh_tag;  
  
static void  
myfirst_shutdown_handler(void *arg, int howto)  
{  
    /* called when the system is shutting down */  
}  
  
/* In attach: */  
myfirst_eh_tag = EVENTHANDLER_REGISTER(shutdown_pre_sync,  
    myfirst_shutdown_handler, sc, EVENTHANDLER_PRI_ANY);  
  
/* In detach: */  
EVENTHANDLER_DEREGISTER(shutdown_pre_sync, myfirst_eh_tag);
```

The `shutdown_pre_sync`, `shutdown_post_sync`, `shutdown_final`, and `vm_lowmem` event names are the most commonly used in drivers. Each is a documented hook point and each has its own semantics about what the driver may do (sleep, allocate memory, take locks, talk to hardware) inside the callback.

The cardinal rule applies to event handlers exactly as it does to everything else: every successful `EVENTHANDLER_REGISTER` must be paired with an `EVENTHANDLER_DEREGISTER` in the reverse order. Forgetting the deregister leaves a dangling function pointer in the event-handler table; the next time the event fires after module unload, the kernel will jump into freed memory and panic.

24.8.7 SYSINIT for One-Shot Kernel Init

One last machinery worth knowing about is `SYSINIT(9)`, the kernel's compile-time-registered one-shot init mechanism. A `SYSINIT` declaration in driver code:

```
static void
myfirst_sysinit(void *arg __unused)
{
    /* runs once, very early at kernel boot */
}
SYSINIT(myfirst_init, SI_SUB_DRIVERS, SI_ORDER_FIRST,
        myfirst_sysinit, NULL);
```

declares a function that runs at a specific point during kernel init, before any user-space process exists. `SYSINIT` is rarely needed in driver code; the function does not get re-run when the module reloads, so it does not give the driver a chance to set up per-load state. Most drivers that think they want `SYSINIT` actually want a `MOD_LOAD` event handler instead.

The matching `SYSUNINIT(9)` declaration:

```
SYSUNINIT(myfirst_uninit, SI_SUB_DRIVERS, SI_ORDER_FIRST,
        myfirst_sysuninit, NULL);
```

declares a function that runs at the corresponding teardown point. The order of declaration matters: `SI_SUB_DRIVERS` runs after `SI_SUB_VFS` but before `SI_SUB_KICK_SCHEDULER`, so a `SYSINIT` at this level can already use the file system but cannot yet schedule processes.

24.8.8 bus_generic_detach and device_delete_children

Drivers that are themselves buses (a PCI-to-PCI bridge driver, a USB hub driver, a bus-style virtual driver) have child devices attached to them. Detaching the parent must first detach all the children, in the right order. The framework provides two helpers:

`bus_generic_detach(dev)` walks the device's children and calls `device_detach` on each one. It returns 0 if every child detached successfully, or the first non-zero return code if any child refused.

`device_delete_children(dev)` calls `bus_generic_detach` and then `device_delete_child` for every child, freeing the child device structures.

A bus-style driver's detach should always begin with one of these two:

```
static int
mybus_detach(device_t dev)
{
    int error;

    error = bus_generic_detach(dev);
    if (error != 0)
        return (error);

    /* now safe to tear down per-bus state */
```

```
...  
return (0);  
}
```

If the driver tears down its bus state before detaching the children, the children find their parent's resources freed out from under them and crash. The order is therefore: detach children first (`bus_generic_detach`), then tear down per-bus state.

24.8.9 Putting It All Together

The lifecycle discipline can be summarised in a small checklist that every driver should pass:

1. **Every allocation has a corresponding free.** Track this with a `goto err` chain in `attach` and a mirror order in `detach`.
2. **Every registration has a corresponding deregistration.** This applies equally to `cdevs`, `sysctls`, `callouts`, `taskqueues`, `event handlers`, `interrupt handlers`, `DMA tags`, and `bus resources`.
3. **The order of teardown is the reverse of the order of setup.** A driver that violates this will leak, panic, or both.
4. **The detach function refuses the operation if any externally-visible resource is still in use.** `EBUSY` is the right return code.
5. **The detach function never frees the softc; the framework does that automatically after detach returns successfully.**
6. **The cdev is destroyed (not freed) with `destroy_dev`, and `destroy_dev` blocks until in-flight callbacks return.**
7. **The per-device `sysctl` context is torn down automatically; the driver does not call `sysctl_ctx_free` for it.**
8. **Bus-style drivers detach their children first with `bus_generic_detach` or `device_delete_children`, then tear down per-bus state.**
9. **A failed attach unwinds every preceding step before returning the failure code.**
10. **The kernel never sees a half-attached driver: `attach` either succeeds completely or fails completely.**

The `myfirst` driver of stage 3 passes every item on this checklist; the lab in Section 9 has the reader inject a deliberate failure to see the unwinding in action.

24.8.10 Wrapping Up Section 7

Section 7 codified the lifecycle discipline that holds every previous section together. The `goto err` chain in `attach` and the reverse-order teardown in `detach` are the two patterns the reader will use in every driver they write from now on. Module-level hooks (`MOD_LOAD`, `MOD_UNLOAD`), event-handler registration (`EVENTHANDLER_REGISTER`), and bus-style detach (`bus_generic_detach`) are the variations that some drivers need; for a single-instance pseudo-driver such as `myfirst`, the basic `attach/detach` pair plus the `goto err` chain is sufficient.

In Section 8 we step back to the chapter's other meta-topic: how the driver evolved from version 1.0 in Part II through 1.5-`channels` in Part III, 1.6-`debug` in Chapter 23, and now 1.7-`integration` here, and how that

evolution should be visible in source comments, in the `MODULE_VERSION` declaration, and in user-visible places such as the `sysctl` version OID. The reader will leave Chapter 24 not just with a fully integrated driver but with a discipline for how a driver's version number tells a reader what to expect.

24.9 Section 8: Refactoring and Versioning

24.9.1 A Driver Has a History

The `myfirst` driver did not appear fully formed. It started in Part II as a single-file demonstration of how `DRIVER_MODULE` works, grew in Part III to support multiple instances and per-channel state, gained debug and tracing infrastructure in Chapter 23, and acquires its full integration surface in this chapter. Each step left the source larger and more capable.

Drivers in the FreeBSD tree have similarly long histories. `null(4)` dates back to 1982; its `cdevsw` has been refactored at least three times to accommodate kernel evolution, but its user-visible behaviour is unchanged. `if_ethersubr.c` predates IPv6 and its API has grown new functions every release while the legacy ones stayed in place. The art of driver maintenance is partly about knowing how to extend a driver without breaking what came before.

This section is a short pause to talk about three closely related disciplines: how to refactor a driver as it grows, how to express the version it is at, and how to decide what counts as a breaking change. The chapter's working example is the transition from 1.6-debug (the end of Chapter 23) to 1.7-integration (the end of this chapter), but the patterns generalise to any driver project.

24.9.2 From One File to Several

The `myfirst` driver in Chapter 23 was a small but real source tree:

```
myfirst.c      /* probe, attach, detach, cdevsw, read, write */
myfirst.h      /* softc, function declarations */
myfirst_debug.c /* SDT provider definition */
myfirst_debug.h /* DPRINTF, debug class bits */
Makefile
```

Stage 3 of this chapter adds two new source files:

```
myfirst_ioctl.c /* ioctl dispatcher */
myfirst_ioctl.h /* PUBLIC ioctl interface for user space */
myfirst_sysctl.c /* sysctl OID construction */
```

The decision to split each new concern into its own pair of files is deliberate. A single 2000-line `myfirst.c` would compile and load and work, but it would also be harder to read, harder to test, and harder for a co-maintainer to navigate. Splitting along concern lines (open/close vs ioctl vs sysctl vs debug) lets each file fit on a screen and lets the reader understand one concern at a time.

The pattern is roughly:

- `<driver>.c` holds probe, attach, detach, the `cdevsw` struct, and the small set of `cdevsw` callbacks (open, close, read, write).
- `<driver>.h` holds the `softc`, function declarations shared across files, and any private constants. **Not** included by user space.
- `<driver>_debug.c` and `<driver>_debug.h` hold the SDT provider, the `DPRINTF` macro, the debug class enumeration. **Not** included by user space.
- `<driver>_ioctl.c` holds the `ioctl` dispatcher. `<driver>_ioctl.h` is the **public** header, includes only `sys/types.h` and `sys/ioccom.h`, and is safe to include from user-space code.
- `<driver>_sysctl.c` holds the `sysctl` OID construction. **Not** included by user space.

The split between public and private headers matters for two reasons. First, public headers must compile cleanly without kernel context (`_KERNEL` is not defined when user space includes them); a header that pulls in `sys/lock.h` and `sys/mutex.h` will fail to compile from a user-space build. Second, public headers are part of the driver's contract with user space and must be installable into a system-wide location such as `/usr/local/include/myfirst/myfirst_ioctl.h`. A private header that accidentally becomes public is a maintenance trap: every user-space program that includes it pins the driver's internal layout, and any future refactor breaks them.

The `myfirst_ioctl.h` header in this chapter is the driver's only public header. It is small, self-contained, and uses only stable types.

24.9.3 Version Strings, Version Numbers, and the API Version

A driver carries three different versions, each meaning something different.

The **release version** is the human-readable string printed in `dmesg`, exposed through `dev.<driver>.0.version`, and used in conversation and documentation. The `myfirst` driver uses dotted strings such as `1.6-debug` and `1.7-integration`. The format is convention; what matters is that the string is short, descriptive, and unique per release.

The **module version** is an integer declared with `MODULE_VERSION(<name>, <integer>)`. It is used by the kernel to enforce dependencies between modules. A module that depends on `myfirst` declares `MODULE_DEPEND(other, myfirst, 1, 1, 1)`, where the three integers are the minimum, preferred, and maximum acceptable versions. Bumping the module version signals "I broke compatibility with previous versions; modules that depend on me must be rebuilt."

The **API version** is the integer exposed through `MYFIRSTIOC_GETVER` and stored in the `MYFIRST_IOCTL_VERSION` constant. It is used by user-space programs to detect API drift before they issue `ioctls` that might fail. Bumping the API version signals "the user-space-visible interface changed in a way that older programs will not handle."

The three versions are independent. The same release may bump only the API version (because a new `ioctl` was added) without bumping the module version (because in-kernel dependents are unaffected). Conversely, a refactor that changes the layout of an exported in-kernel data structure may bump the module version without bumping the API version, because user space sees no change.

For `myfirst`, the chapter uses these values:

```
/* myfirst_sysctl.c */
#define MYFIRST_VERSION "1.7-integration"

/* myfirst.c */
MODULE_VERSION(myfirst, 1);

/* myfirst_ioctl.h */
#define MYFIRST_IOCTL_VERSION 1
```

The release is 1.7-integration because we just landed the integration work. The module version remains 1 because no in-kernel dependents exist. The API version is 1 because this is the first chapter that exposes ioctls; the chapter's stage 2 introduced the interface, and any future change to the ioctl layout would have to bump it.

24.9.4 When to Bump Each

The rule for bumping the **release version** is “every time the driver changes in a way the operator might care about.” Adding a feature, changing default behaviour, fixing a notable bug all qualify. The release version is for humans; it should change often enough that the field is informative.

The rule for bumping the **module version** is “when in-kernel users of the driver would need a recompile to keep working.” Adding a new in-kernel function is not a bump (old dependents still work). Removing a function or changing its signature is a bump. Renaming a struct field that other modules read is a bump. A driver that exports nothing in-kernel may keep its module version at 1 forever.

The rule for bumping the **API version** is “when an existing user-space program would misinterpret the driver's responses or fail in a non-obvious way.” Adding a new ioctl is not a bump (old programs do not use it). Changing the layout of an existing ioctl's argument structure is a bump. Renumbering an existing ioctl is a bump. A driver that has not yet released to users may freely change the API version while the interface is still being designed; once the first user has shipped against it, every change is a public event.

24.9.5 Compatibility Shims

A driver that has shipped widely accumulates compatibility shims. The classic shape is a “version 1” ioctl that the driver supports forever next to a “version 2” ioctl that supersedes it. User-space programs that use the v1 interface keep working, programs that use v2 get the new behaviour, and the driver carries both code paths.

The cost of shims is real. Every shim is code that has to be tested, documented, and maintained. Every shim is also a covered API that constrains future refactors. A driver with five shims is harder to evolve than a driver with one.

The discipline is therefore to design carefully up front so that shims are rare. Three habits help:

- **Use named constants, not literal numbers.** A program that says MYFIRSTIOC_SETMSG rather than 0x802004d3 will keep working when the driver renumbers the ioctl, because both header and program rebuild against the new header.

- **Prefer additive change over mutating change.** When the driver needs to expose a new field, add a new `ioctl` rather than extending an existing structure. The old `ioctl` keeps its layout; the new one carries the extra information.
- **Version every public structure.** A struct `myfirst_v1_args` paired with `MYFIRSTIOC_SETMSG_V1` is a small annotation now and a large compatibility win later.

`myfirst` in this chapter is so small that it does not yet have any shims. The chapter's sole concession to versioning is the `MYFIRSTIOC_GETVER` `ioctl`, which gives a future maintainer a clean place to add shim logic when the time comes.

24.9.6 A Worked Refactor: Splitting `myfirst.c`

The transition from Chapter 23's stage 3 (debug) to this chapter's stage 3 (`sysctl`) is itself a small refactor. The starting source had a single 1000-line `myfirst.c` and a small `myfirst_debug.c`. The ending source has the same `myfirst.c` shrunk by about 100 lines, plus two new files (`myfirst_ioctl.c` and `myfirst_sysctl.c`) that absorb the new logic.

The refactor steps were:

1. Add the two new files with the new logic.
2. Add the new function declarations to `myfirst.h` so the `cdevsw` can reference `myfirst_ioctl`.
3. Update `myfirst.c` to call `myfirst_sysctl_attach(sc)` from `attach`.
4. Update the `Makefile` to list the new files in `SRCS`.
5. Build, load, exercise, and verify that the driver still passes every Chapter 23 lab.
6. Bump the release version to 1.7-integration.
7. Add the `MYFIRSTIOC_GETVER` test to the chapter's verification scripts.

Each step is small enough to review on its own. None of them touches the existing logic, which means the refactor is unlikely to introduce regressions in code that worked before. This is the discipline of additive refactoring: grow the driver outward by adding new files and new declarations, leave the existing code in place, and bump the version when the dust settles.

A more aggressive refactor (renaming a function, rearranging a structure, changing the `cdevsw`'s flag set) would need a different discipline: a single commit per change, regression tests run after each, and a clear note in the version bump about what was rearranged. Drivers that ship widely use this discipline for every release; the in-tree `if_em` driver, for example, has a multi-commit refactor in nearly every minor release of FreeBSD, each commit rolled out independently and tested in isolation.

24.9.7 Three In-Tree Drivers Compared

Three drivers in the FreeBSD source tree illustrate the source-layout discipline at three points along the complexity spectrum. Reading them as a triplet makes the patterns visible.

`/usr/src/sys/dev/null/null.c` is the smallest. It is a single 200-line source file with one `cdevsw` table, one set of callbacks, no separate header, and no debug or `sysctl` machinery. The whole driver fits on three printed pages. This is the layout for a driver whose entire job is to be present and to absorb (or generate) bytes; integration is only at the `cdev` layer.

`/usr/src/sys/net/if_disc.c` is a two-file network driver: `if_disc.c` for the driver code and the implicit `if.h` for the framework. The driver registers with the network stack but otherwise has no `sysctl` tree, no debug subtree, and no public `ioctl` header (it uses the standard `if_ioctl` set defined by the framework). This is the layout for a driver that is an instance of a framework rather than its own thing; the framework defines the surface, the driver fills in the slots.

`/usr/src/sys/dev/ahci/ahci.c` is a multi-file driver with separate files for the AHCI core, the PCI attach glue, the device-tree FDT attach glue, the enclosure-management code, and the bus-specific logic. Each file is dedicated to one concern; the central file is over 5000 lines but the per-file size is manageable. This is the layout that scales to a real production driver: split by concern, glue by header, and use the file boundary as the unit of refactoring.

The `myfirst` driver in this chapter sits in the middle. Stage 3 has five source files: `myfirst.c` (open/close/read/write and the `cdevsw`), `myfirst.h` (softc, declarations), `myfirst_debug.c` and `myfirst_debug.h` (debug and SDT), `myfirst_ioctl.c` and `myfirst_ioctl.h` (ioctl, with the latter being public), and `myfirst_sysctl.c` (`sysctl` tree). This is enough to demonstrate the split-by-concern pattern without the cognitive overhead of a fifty-file driver. A reader who needs to grow `myfirst` further has a clear template: add a new pair of files for the new concern, add the source file to `SRCS`, add the public header to the install set if user space needs it, and update `MYFIRST_VERSION`.

24.9.8 Wrapping Up Section 8

Section 8 closed the loop on the chapter's other theme: how a driver's source layout, version numbers, and refactor discipline track its evolution. The driver milestone for this chapter is 1.7-integration, expressed simultaneously as the release string in `MYFIRST_VERSION`, the module version 1 (unchanged because no in-kernel dependents exist), and the API version 1 (set for the first time because this is the first chapter that exposes a stable `ioctl` interface). Refactoring was kept additive, so no shim was needed.

The reader has now seen the full integration surface: Sections 2 through 4 covered the three universals (`devfs`, `ioctl`, `sysctl`), Sections 5 and 6 sketched the registration-style integrations (network, CAM), Section 7 codified the lifecycle discipline, and Section 8 framed the whole as an evolution that the driver's version numbers should track. The remaining sections of the chapter give the reader hands-on practice with the same material.

24.9.9 Putting It All Together: The Final Attach and Detach

Before the chapter moves to the labs, it is worth seeing the chapter's complete attach and detach functions in one place. They tie every previous section together: the `cdev` construction from Section 2, the `ioctl` wiring from Section 3, the `sysctl` tree from Section 4, the lifecycle discipline from Section 7, and the version handling from Section 8.

The complete attach for stage 3:

```
static int
myfirst_attach(device_t dev)
{
    struct myfirst_softc *sc = device_get_softc(dev);
    struct make_dev_args args;
```

```
int error;

/* 1. Stash the device pointer and initialise the lock. */
sc->sc_dev = dev;
mtx_init(&sc->sc_mtx, "myfirst", NULL, MTX_DEF);

/* 2. Initialise the in-driver state to its defaults. */
strncpy(sc->sc_msg, "Hello from myfirst", sizeof(sc->sc_msg));
sc->sc_msglen = strlen(sc->sc_msg);
sc->sc_open_count = 0;
sc->sc_total_reads = 0;
sc->sc_total_writes = 0;
sc->sc_debug = 0;

/* 3. Read the boot-time tunable for the debug mask. If the
 * operator set hw.myfirst.debug_mask_default in
 * /boot/loader.conf, sc_debug now holds that value;
 * otherwise sc_debug remains zero.
 */
TUNABLE_INT_FETCH("hw.myfirst.debug_mask_default", &sc->sc_debug);

/* 4. Construct the cdev. The args struct gives us a typed,
 * versionable interface; mda_si_drv1 wires the per-cdev
 * pointer to the softc atomically, closing the race window
 * between creation and assignment.
 */
make_dev_args_init(&args);
args.mda_devsw = &myfirst_cdevsw;
args.mda_uid = UID_ROOT;
args.mda_gid = GID_WHEEL;
args.mda_mode = 0660;
args.mda_si_drv1 = sc;
args.mda_unit = device_get_unit(dev);

error = make_dev_s(&args, &sc->sc_cdev,
    "myfirst%d", device_get_unit(dev));
if (error != 0)
    goto fail_mtx;

/* 5. Build the sysctl tree. The framework owns the per-device
 * context, so we do not need to track or destroy it
 * ourselves; detach below does not call sysctl_ctx_free.
 */
myfirst_sysctl_attach(sc);
```

```

DPRINTF(sc, MYF_DBG_INIT,
        "attach: stage 3 complete, version " MYFIRST_VERSION "\n");
return (0);

fail_mtx:
    mtx_destroy(&sc->sc_mtx);
    return (error);
}

```

The complete detach for stage 3:

```

static int
myfirst_detach(device_t dev)
{
    struct myfirst_softc *sc = device_get_softc(dev);

    /* 1. Refuse detach if anyone holds the device open. The
     *    chapter's pattern is the simple soft refusal; Challenge 3
     *    walks through the more elaborate dev_ref/dev_rel pattern
     *    that drains in-flight references rather than refusing.
     */
    mtx_lock(&sc->sc_mtx);
    if (sc->sc_open_count > 0) {
        mtx_unlock(&sc->sc_mtx);
        return (EBUSY);
    }
    mtx_unlock(&sc->sc_mtx);

    DPRINTF(sc, MYF_DBG_INIT, "detach: tearing down\n");

    /* 2. Destroy the cdev. destroy_dev blocks until every
     *    in-flight cdevsw callback returns; after this call,
     *    no new open/close/read/write/ioctl can arrive.
     */
    destroy_dev(sc->sc_cdev);

    /* 3. The per-device sysctl context is torn down automatically
     *    by the framework after detach returns successfully.
     *    Nothing to do here.
     */

    /* 4. Destroy the lock. Safe now because the cdev is gone and
     *    no other code path can take it.
     */
}

```

```
    */  
    mtx_destroy(&sc->sc_mtx);  
  
    return (0);  
}
```

Two things deserve a final note.

The order of operations is the strict reverse of attach: lock first in attach, lock destroyed last in detach; cdev created near the end of attach, cdev destroyed near the start of detach; sysctl tree created last in attach, sysctl tree torn down first (by the framework, automatically) in detach. This is the cardinal rule of Section 7 in concrete form.

The refusal pattern in detach (the `if (open_count > 0)` check) is the chapter's choice for simplicity. A real driver may need the more elaborate `dev_ref/dev_rel` machinery to implement a draining detach; Challenge 3 walks through that variant. For `myfirst`, the simple refusal gives the operator a clear error and is sufficient.

In Section 9 we move from explanation to practice. The labs walk the reader through building stage 1, stage 2, and stage 3 of the integration in turn, with verification commands and expected output for each. After the labs come the challenge exercises (Section 10), the troubleshooting catalogue (Section 11), and the wrap-up and bridge that close the chapter.

24.10 Hands-on Labs

The labs in this section take the reader from a freshly cloned working tree through every integration surface added in the chapter. Each lab is small enough to complete in one sitting and is paired with a verification command that confirms the change. Run the labs in order; later labs build on earlier ones.

The companion files under `examples/part-05/ch24-integration/` contain three staged reference drivers (`stage1-devfs/`, `stage2-ioctl/`, `stage3-sysctl/`) that match the milestones of this chapter. The labs assume the reader either starts from their own end-of-Chapter-23 driver (version 1.6-debug) or copies the appropriate stage directory into a working location, makes changes there, and consults the matching staged directory if they get stuck.

Each lab uses a real FreeBSD 14.3 system. A virtual machine is fine; do not run these labs on a production host because module load and unload can hang or panic the system if the driver has bugs.

24.10.1 Lab 1: Build and Load the Stage 1 Driver

Goal: bring the driver from the Chapter 23 baseline (1.6-debug) up to the stage 1 milestone of this chapter (a properly constructed cdev under `/dev/myfirst0`).

Setup:

Start from your own end-of-Chapter-23 working tree (the driver at version 1.6-debug) or copy the reference tree of Chapter 23's last stage into a lab directory:

```
$ cp -r ~/myfirst-1.6-debug ~/myfirst-lab1
$ cd ~/myfirst-lab1
$ ls
Makefile  myfirst.c  myfirst.h  myfirst_debug.c  myfirst_debug.h
```

If you want to compare against the already-migrated starting point for the chapter's stage 1 (with `make_dev_s` already applied), consult `examples/part-05/ch24-integration/stage1-devfs/` as a reference solution rather than as a starting directory.

Step 1: Open `myfirst.c` and find the existing call to `make_dev`. The Chapter 23 code uses the older single-call form. Replace it with the `make_dev_args` form from Section 2:

```
struct make_dev_args args;
int error;

make_dev_args_init(&args);
args.mda_devsw = &myfirst_cdevsw;
args.mda_uid = UID_ROOT;
args.mda_gid = GID_WHEEL;
args.mda_mode = 0660;
args.mda_si_drv1 = sc;
args.mda_unit = device_get_unit(dev);

error = make_dev_s(&args, &sc->sc_cdev,
    "myfirst%d", device_get_unit(dev));
if (error != 0) {
    mtx_destroy(&sc->sc_mtx);
    return (error);
}
```

Step 2: Add `D_TRACKCLOSE` to the `cdevsw` flags (it should already have `D_VERSION`):

```
static struct cdevsw myfirst_cdevsw = {
    .d_version = D_VERSION,
    .d_flags   = D_TRACKCLOSE,
    .d_name    = "myfirst",
    .d_open    = myfirst_open,
    .d_close   = myfirst_close,
    .d_read    = myfirst_read,
    .d_write   = myfirst_write,
};
```

Step 3: Confirm `myfirst_open` and `myfirst_close` use `dev->si_drv1` to recover the softc:


```
static int
myfirst_open(struct cdev *dev, int oflags, int devtype, struct thread *td)
{
    struct myfirst_softc *sc = dev->si_drv1;
    ...
}
```

Step 4: Build and load:

```
$ make
$ sudo kldload ./myfirst.ko
```

Verification:

```
$ ls -l /dev/myfirst0
crw-rw---- 1 root wheel 0x... <date> /dev/myfirst0
$ sudo cat /dev/myfirst0
Hello from myfirst
$ sudo kldstat | grep myfirst
N    1 0xffff... 1...    myfirst.ko
$ sudo dmesg | tail
... (debug messages from MYF_DBG_INIT)
```

If the `ls` shows the wrong owner, group, or mode, re-check the `mda_uid`, `mda_gid`, and `mda_mode` values. If `cat` returns an empty string, check that `myfirst_read` is filling the user buffer from `sc->sc_msg`. If the load succeeds but the device does not appear, check that the `cdevsw` is referenced from `make_dev_args`.

Cleanup:

```
$ sudo kldunload myfirst
$ ls -l /dev/myfirst0
ls: /dev/myfirst0: No such file or directory
```

A successful unload removes the device node. If the unload fails with `Device busy`, check that no shell or program has the device open (`fstat | grep myfirst0`).

24.10.2 Lab 2: Add the `ioctl` Interface

Goal: extend the driver to stage 2 by adding the four `ioctl` commands from Section 3.

Setup:

```
$ cp -r examples/part-05/ch24-integration/stage1-devfs ~/myfirst-lab2
$ cd ~/myfirst-lab2
```

Step 1: Create `myfirst_ioctl.h` from the template in Section 3. Place it in the same directory as the other source files. Include `sys/ioccom.h` and `sys/types.h`. Define `MYFIRST_MSG_MAX = 256` and the four `ioctl` numbers. Do not include any kernel-only header.

Step 2: Create `myfirst_ioctl.c` from the template in Section 3. The dispatcher is a single function `myfirst_ioctl` with the standard `d_ioctl_t` signature.

Step 3: Add `myfirst_ioctl.c` to `SRCS` in the `Makefile`:

```
SRCS=    myfirst.c myfirst_debug.c myfirst_ioctl.c
```

Step 4: Update the `cdevsw` to point `.d_ioctl` at the new dispatcher:

```
.d_ioctl = myfirst_ioctl,
```

Step 5: Add `sc_msg` and `sc_msglen` to the `softc` and initialise them in `attach`:

```
strcpy(sc->sc_msg, "Hello from myfirst", sizeof(sc->sc_msg));
sc->sc_msglen = strlen(sc->sc_msg);
```

Step 6: Build the user-space companion. Place `myfirstctl.c` in the same directory and create a small `Makefile.user`:

```
CC?= cc
CFLAGS+= -Wall -Werror -I.

myfirstctl: myfirstctl.c myfirst_ioctl.h
    ${CC} ${CFLAGS} -o myfirstctl myfirstctl.c
```

(Note that the indentation must be a tab, not spaces, for `make` to parse the rule.)

Build the kernel module and the companion:

```
$ make
$ make -f Makefile.user
$ sudo kldload ./myfirst.ko
```

Verification:

```
$ ./myfirstctl get-version
driver ioctl version: 1
$ ./myfirstctl get-message
Hello from myfirst
$ sudo ./myfirstctl set-message "drivers are fun"
$ ./myfirstctl get-message
drivers are fun
$ sudo ./myfirstctl reset
$ ./myfirstctl get-message

$
```

If set-message returns `Permission denied`, the issue is that the device is mode `0660` and the user is not in `wheel`. Either run with `sudo` (as the verification commands above do) or change the device group to one the user belongs to with `mda_gid` and reload the module.

If set-message returns `Bad file descriptor`, the issue is that `myfirstctl` opened the device read-only. Check that the program selects `O_RDWR` for set-message and reset.

If any ioctl returns `Inappropriate ioctl for device`, the issue is a mismatch between the encoded length in `myfirst_ioctl.h` and the dispatcher's view of the data. Re-check the `_IOR/_IOW` macros and the size of the structures they declare.

Cleanup:

```
$ sudo kldunload myfirst
```

24.10.3 Lab 3: Add the Sysctl Tree

Goal: extend the driver to stage 3 by adding the sysctl OIDs from Section 4 and reading a tunable from `/boot/loader.conf`.

Setup:

```
$ cp -r examples/part-05/ch24-integration/stage2-ioctl ~/myfirst-lab3
$ cd ~/myfirst-lab3
```

Step 1: Create `myfirst_sysctl.c` from the template in Section 4. The function `myfirst_sysctl_attach(sc)` builds the entire tree.

Step 2: Add `myfirst_sysctl.c` to `SRCS` in the Makefile.

Step 3: Update `myfirst_attach` to call `TUNABLE_INT_FETCH` and `myfirst_sysctl_attach`:

```
TUNABLE_INT_FETCH("hw.myfirst.debug_mask_default", &sc->sc_debug);
```

```
/* ... after make_dev_s succeeds: */  
myfirst_sysctl_attach(sc);
```

Step 4: Build and load:

```
$ make  
$ sudo kldload ./myfirst.ko
```

Verification:

```
$ sysctl -a dev.myfirst.0  
dev.myfirst.0.debug.classes: INIT(0x1) OPEN(0x2) IO(0x4) IOCTL(0x8) ...  
dev.myfirst.0.debug.mask: 0  
dev.myfirst.0.message_len: 18  
dev.myfirst.0.message: Hello from myfirst  
dev.myfirst.0.total_writes: 0  
dev.myfirst.0.total_reads: 0  
dev.myfirst.0.open_count: 0  
dev.myfirst.0.version: 1.7-integration
```

Open the device once and re-check the counters:

```
$ cat /dev/myfirst0  
Hello from myfirst  
$ sysctl dev.myfirst.0.total_reads  
dev.myfirst.0.total_reads: 1
```

Test the loader-time tunable. Edit `/boot/loader.conf` (back it up first):

```
$ sudo cp /boot/loader.conf /boot/loader.conf.backup  
$ sudo sh -c 'echo hw.myfirst.debug_mask_default="\0x06\" >> /boot/loader.conf'
```

Note that this only takes effect on the next reboot, and only if the module is loaded by the loader (not by `kldload` after boot). For an interactive test without rebooting, set the value before loading:

```
$ sudo kenv hw.myfirst.debug_mask_default=0x06  
$ sudo kldload ./myfirst.ko  
$ sysctl dev.myfirst.0.debug.mask  
dev.myfirst.0.debug.mask: 6
```

If the value is 0 instead of 6, check that the `TUNABLE_INT_FETCH` call uses the same string as the `kenv` command. The call must run before `myfirst_sysctl_attach` so the value is in place when the OID is created.

Cleanup:

```
$ sudo kldunload myfirst
$ sudo cp /boot/loader.conf.backup /boot/loader.conf
```

24.10.4 Lab 4: Walk the Lifecycle by Injecting a Failure

Goal: see the goto err chain in attach actually unwind by deliberately failing one of the steps.

Setup:

```
$ cp -r examples/part-05/ch24-integration/stage3-sysctl ~/myfirst-lab4
$ cd ~/myfirst-lab4
```

Step 1: Open myfirst.c and find myfirst_attach. Insert a deliberate failure right after make_dev_s succeeds:

```
error = make_dev_s(&args, &sc->sc_cdev,
    "myfirst%d", device_get_unit(dev));
if (error != 0)
    goto fail_mtx;

/* DELIBERATE FAILURE for Lab 4 */
device_printf(dev, "Lab 4: injected failure after make_dev_s\n");
error = ENXIO;
goto fail_cdev;

myfirst_sysctl_attach(sc);
return (0);

fail_cdev:
    destroy_dev(sc->sc_cdev);
fail_mtx:
    mtx_destroy(&sc->sc_mtx);
    return (error);
```

Step 2: Build and try to load:

```
$ make
$ sudo kldload ./myfirst.ko
kldload: an error occurred while loading module myfirst. Please check dmesg(8) for more
↪ details.
$ sudo dmesg | tail
myfirst0: Lab 4: injected failure after make_dev_s
```

Verification:

```
$ ls /dev/myfirst0
ls: /dev/myfirst0: No such file or directory
$ kldstat | grep myfirst
$
```

The cdev is gone (the goto `fail_cdev` cleanup destroyed it), the module is not loaded, and no resources leaked. If the cdev remains after the failure, the cleanup is missing a `destroy_dev` call. If the kernel panics on the next module load attempt, the cleanup is freeing or destroying something twice.

Bonus: change the failure injection to occur **before** `make_dev_s`. The cleanup chain should now skip the `fail_cdev` label and only run `fail_mtx`. Verify that the cdev was never created and that the mutex is destroyed:

```
$ sudo kldload ./myfirst.ko
$ sudo dmesg | tail
... no Lab 4 message because it now runs before make_dev_s ...
```

Cleanup:

Remove the deliberate failure block before continuing.

24.10.5 Lab 5: Trace the Integration Surfaces with DTrace

Goal: use the SDT probes from Chapter 23 to trace `ioctl`, `open`, `close`, and read traffic in real time.

Setup: stage 3 driver loaded as in Lab 3.

Step 1: Verify the probes are visible to DTrace:

```
$ sudo dtrace -l -P myfirst
  ID PROVIDER  MODULE  FUNCTION  NAME
... id myfirst  kernel  -         open
... id myfirst  kernel  -         close
... id myfirst  kernel  -         io
... id myfirst  kernel  -         ioctl
```

If the list is empty, the SDT probes are not registered. Check that `myfirst_debug.c` is in `SRCS` and that `SDT_PROBE_DEFINE*` is called from there.

Step 2: Open a long-running trace in one terminal:

```
$ sudo dtrace -n 'myfirst:::ioctl { printf("ioctl cmd=0x%x flags=0x%x", arg1, arg2); }'
dtrace: description 'myfirst:::ioctl' matched 1 probe
```

Step 3: Exercise the driver in another terminal:

```
$ ./myfirstctl get-version
$ ./myfirstctl get-message
$ sudo ./myfirstctl set-message "Lab 5"
$ sudo ./myfirstctl reset
```

The DTrace terminal should show one line per ioctl with the command code and the file flags.

Step 4: Combine multiple probes into one script:

```
$ sudo dtrace -n '
    myfirst:::open { printf("open  pid=%d", pid); }
    myfirst:::close { printf("close pid=%d", pid); }
    myfirst:::io    { printf("io    pid=%d write=%d resid=%d", pid, arg1, arg2); }
    myfirst:::ioctl { printf("ioctl pid=%d cmd=0x%x", pid, arg1); }
'
```

In another terminal:

```
$ cat /dev/myfirst0
$ ./myfirstctl get-version
$ echo "hello" | sudo tee /dev/myfirst0
```

The DTrace output now shows the complete traffic pattern, with the open and close around each operation, the read or write inside, and any ioctls. This is the value of having SDT probes integrated with the cdevsw callbacks: every integration surface that the driver exposes is also a probe surface for DTrace.

Cleanup:

```
^C
$ sudo kldunload myfirst
```

24.10.6 Lab 6: Integration Smoke Test

Goal: build a single shell script that exercises every integration surface in one run and produces a green/red summary that the reader can paste into a bug report or a release-readiness checklist.

A smoke test is a small, fast, end-to-end check that the driver is alive and that every surface responds. It does not replace careful unit testing; it gives the reader a five-second confirmation that nothing is obviously broken before they invest more time. Real drivers have smoke tests; the chapter recommends adding one to every new driver from day one.

Setup: stage 3 driver loaded.

Step 1: Create `smoke.sh` in the working directory:

```
#!/bin/sh
# smoke.sh - end-to-end smoke test for the myfirst driver.

set -u
fail=0

check() {
    if eval "$1"; then
        printf "  PASS  %s\n" "$2"
    else
        printf "  FAIL  %s\n" "$2"
        fail=$((fail + 1))
    fi
}

echo "=== myfirst integration smoke test ==="

# 1. Module is loaded.
check "kldstat | grep -q myfirst" "module is loaded"

# 2. /dev node exists with the right mode.
check "test -c /dev/myfirst0" "/dev/myfirst0 exists as a character device"
check "test \"\$(stat -f %Lp /dev/myfirst0)\\" = \"660\" " "/dev/myfirst0 is mode 0660"

# 3. Sysctl tree is present.
check "sysctl -N dev.myfirst.0.version >/dev/null 2>&1" "version OID is present"
check "sysctl -N dev.myfirst.0.debug.mask >/dev/null 2>&1" "debug.mask OID is present"
check "sysctl -N dev.myfirst.0.open_count >/dev/null 2>&1" "open_count OID is present"

# 4. Ioctls work (requires myfirstctl built).
check "./myfirstctl get-version >/dev/null" "MYFIRSTIOC_GETVER returns success"
check "./myfirstctl get-message >/dev/null" "MYFIRSTIOC_GETMSG returns success"
check "sudo ./myfirstctl set-message smoke && [ \"\$(./myfirstctl get-message)\\" =
↪ smoke ]" "MYFIRSTIOC_SETMSG round-trip works"
check "sudo ./myfirstctl reset && [ -z \"\$(./myfirstctl get-message)\\" ]"
↪ "MYFIRSTIOC_RESET clears state"

# 5. Read/write basic path.
check "echo hello | sudo tee /dev/myfirst0 >/dev/null" "write to /dev/myfirst0
↪ succeeds"
check "[ \"\$(cat /dev/myfirst0)\\" = hello ]" "read returns the previously written
↪ message"

# 6. Counters update.
```



```

sudo ./myfirstctl reset >/dev/null
cat /dev/myfirst0 >/dev/null
check "[ \"\$(sysctl -n dev.myfirst.0.total_reads)\\" = 1 ]" "total_reads incremented
↳ after one read"

# 7. SDT probes are registered.
check "sudo dtrace -l -P myfirst | grep -q open" "myfirst:::open SDT probe is visible"

echo "=== summary ==="
if [ $fail -eq 0 ]; then
    echo "ALL PASS"
    exit 0
else
    printf "%d FAIL\n" "$fail"
    exit 1
fi

```

Step 2: Make it executable and run it:

```

$ chmod +x smoke.sh
$ ./smoke.sh
=== myfirst integration smoke test ===
PASS module is loaded
PASS /dev/myfirst0 exists as a character device
PASS /dev/myfirst0 is mode 0660
PASS version OID is present
PASS debug.mask OID is present
PASS open_count OID is present
PASS MYFIRSTIOC_GETVER returns success
PASS MYFIRSTIOC_GETMSG returns success
PASS MYFIRSTIOC_SETMSG round-trip works
PASS MYFIRSTIOC_RESET clears state
PASS write to /dev/myfirst0 succeeds
PASS read returns the previously written message
PASS total_reads incremented after one read
PASS myfirst:::open SDT probe is visible
=== summary ===
ALL PASS

```

If any check fails, the script's output points directly at the broken integration surface. A failed `version OID` is present means the `sysctl` construction did not run; a failed `MYFIRSTIOC_GETVER` means the `ioctl` dispatcher is not wired correctly; a failed `total_reads incremented` means the `read` callback is not bumping the counter under the mutex.

Verification: re-run after every change to the driver. A passing smoke test before a commit is the cheapest possible insurance against a regression that breaks the basic flow.

24.10.7 Lab 7: Reload Without Restarting User-Space Programs

Goal: confirm that the driver can be unloaded and reloaded while a user-space program holds an open file descriptor open in another terminal.

This test reveals lifecycle bugs that the chapter's "soft detach" pattern is supposed to prevent. A driver that returns `EBUSY` from `detach` when a user holds the device open is correctly defending itself; a driver that lets `detach` succeed and then panics when the user issues an `ioctl` is broken.

Setup: stage 3 driver loaded.

Step 1 (terminal 1): hold the device open with a long-running command:

```
$ sleep 3600 < /dev/myfirst0 &
$ jobs
[1]+  Running                  sleep 3600 < /dev/myfirst0 &
```

Step 2 (terminal 2): try to unload:

```
$ sudo kldunload myfirst
kldunload: can't unload file: Device busy
```

This is the expected behaviour. The chapter's `myfirst_detach` checks `open_count > 0` and returns `EBUSY` rather than tearing down the `cdev` under an open file descriptor.

Step 3 (terminal 2): verify that the device is still functional from a different shell:

```
$ ./myfirstctl get-version
driver ioctl version: 1
$ sysctl dev.myfirst.0.open_count
dev.myfirst.0.open_count: 1
```

The open count reflects the held file descriptor.

Step 4 (terminal 1): release the file descriptor:

```
$ kill %1
$ wait
```

Step 5 (terminal 2): unload now succeeds:

```
$ sudo kldunload myfirst
$ sysctl dev.myfirst.0
sysctl: unknown oid 'dev.myfirst.0'
```

The OID is gone because Newbus tore down the per-device sysctl context after detach returned successfully.

Verification: the unload should succeed every time without a panic. If the kernel panics during step 5, the cause is almost always that the cdev's callbacks are still in flight when `destroy_dev` returns; check that the cdevsw's `d_close` correctly releases anything it acquired in `d_open`, and check that no callout or taskqueue is still scheduled.

A bonus extension is to write a small program that opens the device, immediately calls `MYFIRSTIOC_RESET`, and then loops on `MYFIRSTIOC_GETVER` for several seconds. While the loop is running, try to unload from another terminal. The unload should still fail with `EBUSY`; the in-flight ioctls should not corrupt anything.

24.10.8 Wrapping Up the Labs

The seven labs walked the reader through the full integration surface, the lifecycle discipline, the smoke test, and the soft-detach contract. Stage 1 added the cdev; stage 2 added the ioctl interface; stage 3 added the sysctl tree; the lab on lifecycle (Lab 4) confirmed the unwind; the DTrace lab (Lab 5) confirmed integration with Chapter 23's debug infrastructure; the smoke test (Lab 6) gave the reader a reusable verification script; and the reload lab (Lab 7) confirmed the soft-detach contract.

A driver that passes all seven labs is at the chapter's milestone version, `1.7-integration`, and is ready for the next chapter's topic. The challenge exercises in Section 10 give the reader optional follow-on work that extends the driver beyond what the chapter covers.

24.11 Challenge Exercises

The challenges below are optional and are intended for the reader who wants to push the driver beyond the chapter's milestone. Each challenge has a stated goal, a few hints about the approach, and a note about which sections of the chapter contain the relevant material. None of the challenges has a single correct answer; readers are encouraged to compare their solution with a reviewer or with the in-tree drivers cited as references.

24.11.1 Challenge 1: Add a Variable-Length Ioctl

Goal: extend the ioctl interface so that a user-space program can transfer a buffer larger than the fixed 256 bytes used by `MYFIRSTIOC_SETMSG`.

The chapter's pattern is fixed-size: `MYFIRSTIOC_SETMSG` declares `_IOW('M', 3, char[256])` and the kernel handles the entire copyin. For larger buffers (say, up to 1 MB), the embedded-pointer pattern is needed:

```
struct myfirst_blob {
    size_t  len;
    char   *buf;    /* user-space pointer */
}
```

```
};
#define MYFIRSTIOC_SETBLOB _IOW('M', 5, struct myfirst_blob)
```

The dispatcher must call `copyin` to transfer the bytes the pointer references; the structure itself comes through automatic `copyin` as before. Hints: enforce a maximum length (1 MB is reasonable). Allocate a temporary kernel buffer with `malloc(M_TEMP, len, M_WAITOK)`; do not allocate it inside the softc mutex. Free it before returning. Reference: Section 3, “Common Pitfalls With `ioctl`”, second pitfall.

A bonus extension is to add `MYFIRSTIOC_GETBLOB` that copies the current message in the same variable-length format; pay attention to the case where the user-supplied buffer is shorter than the message and decide whether to truncate, return `ENOMEM`, or write back the required length. Real drivers (`SIOCGIFCAP`, `KIOCGRPC`) use the latter pattern.

24.11.2 Challenge 2: Add a Per-Open Counter

Goal: maintain a per-file-descriptor counter (one number for each open of `/dev/myfirst0`) instead of just the per-instance counter we have now.

The chapter’s `sc_open_count` aggregates across all opens. A per-open counter would let a program know how much it has read from its own descriptor. Hints: use `cdevsw->d_priv` to attach a per-fd structure (a `struct myfirst_fdpriv` containing a counter). Allocate the structure in `myfirst_open` and free it in `myfirst_close`. The framework gives each `cdev_priv` a unique pointer in the file’s `f_data` field; the read and write callbacks can then look up the per-fd structure through `devfs_get_cdevpriv()`.

Reference: `/usr/src/sys/kern/kern_conf.c` for `devfs_set_cdevpriv` and `devfs_get_cdevpriv`. The pattern is also used by `/usr/src/sys/dev/random/random_harvestq.c`.

A bonus extension is to add a `sysctl` OID that reports the sum of per-fd counters and verify that it equals the existing aggregate counter at all times. Discrepancies indicate a missing increment somewhere.

24.11.3 Challenge 3: Implement Soft Detach With `dev_ref`

Goal: replace the chapter’s “refuse detach if open” pattern with the cleaner “drain to last close, then detach” pattern.

The chapter’s `detach` returns `EBUSY` if any user holds the device open. A more elegant pattern uses `dev_ref/dev_rel` to count outstanding references and waits for the count to reach zero before completing detach. Hints: take a `dev_ref` in `myfirst_open` and release it in `myfirst_close`. In `detach`, set a “going-away” flag, then call `destroy_dev_drain` (or write a small loop that calls `tsleep` while `dev_refs > 0`) before calling `destroy_dev`. Once the count reaches zero and the `cdev` is destroyed, complete detach normally.

Reference: `/usr/src/sys/kern/kern_conf.c` for the `dev_ref` machinery; `/usr/src/sys/fs/cuse` is a real driver that uses the drain pattern for sleeping detach.

The bonus extension is to add a `sysctl` OID that reports the current reference count and verify that it matches the open count.

24.11.4 Challenge 4: Replace the Static Magic Letter

Goal: replace the hard-coded 'M' magic letter in `myfirst_ioctl.h` with a name that does not collide with anything else in the tree.

The chapter chose 'M' arbitrarily and warned about the risk of collisions. A more defensive driver uses a longer magic identifier and constructs the ioctl numbers from it. Hints: define `MYFIRST_IOC_GROUP = 0x83` (or any byte not used by another driver). The `_IOC` macro then takes that constant rather than a character literal. Document the choice with a comment in the header explaining how it was chosen.

A bonus is to `grep /usr/src/sys` for `_IO[RW]?\\(.\\? 'M'` and produce a list of every existing use of 'M'. (There are several, including MIDI ioctls and others; the survey itself is educational.)

24.11.5 Challenge 5: Add an EVENTHANDLER for Shutdown

Goal: make the driver behave gracefully when the system is shutting down.

The chapter's driver has no shutdown handler; if the system shuts down with `myfirst` loaded, the framework eventually calls `detach`. A more polished driver registers an `EVENTHANDLER` for `shutdown_pre_sync` so it can flush any in-flight state before the file system goes read-only.

Hints: register the handler in `attach` with `EVENTHANDLER_REGISTER(shutdown_pre_sync, ...)`. The handler is called at the corresponding shutdown stage. Deregister in `detach` with `EVENTHANDLER_DEREGISTER`. Inside the handler, set the driver to a quiescent state (clear the message, zero the counters); at this point the file system is still writable, so any user feedback through `printf` will land in `/var/log/messages` after the next boot.

Reference: Section 7, "EVENTHANDLER for System Events" and `/usr/src/sys/sys/eventhandler.h` for the full list of named events.

24.11.6 Challenge 6: A Second Per-Driver Sysctl Subtree

Goal: add a second subtree under `dev.myfirst.0` that exposes per-thread statistics.

The chapter's tree has a `debug.` subtree. A complete driver might also have a `stats.` subtree (for read/write statistics broken down by file descriptor) or an `errors.` subtree (for error counters). Hints: use `SYSCTL_ADD_NODE` to create a new node, then `SYSCTL_ADD_*` to populate it under the new node's `SYSCTL_CHILDREN`. The pattern is identical to the `debug.` subtree but rooted under a different name.

Reference: Section 4, "The `myfirst` Sysctl Tree" for the existing `debug.` subtree as a model; `/usr/src/sys/dev/iicbus` for several drivers that use multi-subtree sysctl layouts.

24.11.7 Challenge 7: Cross-Module Dependency

Goal: build a second small module (`myfirst_logger`) that depends on `myfirst` and uses its in-kernel API.

The chapter's `myfirst` driver does not export any symbols for in-kernel users. Adding a second module that calls into `myfirst` exercises the `MODULE_DEPEND` machinery. Hints: declare a symbol-bearing function in `myfirst.h` (perhaps `int myfirst_get_message(int unit, char *buf, size_t len)`) and implement it in `myfirst.c`. Build the second module with `MODULE_DEPEND(myfirst_logger, myfirst, 1, 1, 1)` so the kernel loads `myfirst` automatically when `myfirst_logger` loads.

A bonus is to bump the module version of `myfirst` to 2, change the in-kernel API in a non-backward-compatible way, and observe that the second module fails to load until rebuilt against the new version. Reference: Section 8, “Version Strings, Version Numbers, and the API Version”.

24.11.8 Closing the Challenges

The seven challenges range from short (Challenge 4 is mostly a rename and a comment) to substantial (Challenge 3 requires reading and understanding `dev_ref`). The reader who completes all seven will have hands-on familiarity with every integration corner the chapter only sketches. The reader who completes any one will have a deeper feel for the integration discipline than the chapter alone can provide.

24.12 Troubleshooting

The integration surfaces in this chapter sit at the seam between the kernel and the rest of the system. Problems at the seam often look like driver bugs but are really symptoms of a missing flag, a typo in a header, or a misunderstanding about who owns what. The catalogue below collects the most common symptoms, their likely causes, and the fix for each.

24.12.1 `/dev/myfirst0` Does Not Appear After `kldload`

The first thing to check is whether the module loaded successfully:

```
$ kldstat | grep myfirst
```

If the module is not listed, the load failed; consult `dmesg` for a more specific message. The most common reason is an unresolved symbol (often because the new source file is not in `SRCS`).

If the module is listed but the device node is missing, the `make_dev_s` call inside `myfirst_attach` likely failed. Add a `device_printf(dev, "make_dev_s returned %d\n", error)` next to the call and re-try. The most common reason for a non-zero return is that another driver already created `/dev/myfirst0` (the kernel will not silently overwrite an existing node) or that `make_dev_s` was called from an unsleepable context with `MAKEDEV_NOWAIT`.

A subtler reason is that `cdevsw->d_version` does not equal `D_VERSION`. The kernel checks this and refuses to register a `cdevsw` with a version mismatch. The fix is `static struct cdevsw myfirst_cdevsw = { .d_version = D_VERSION, ... };` exactly.

24.12.2 `cat /dev/myfirst0` Returns “Permission denied”

The device exists but the user cannot open it. The default mode in this chapter is `0660` and the default group is `wheel`. Either run with `sudo`, change `mda_gid` to the user’s group, or change `mda_mode` to `0666` (the latter is fine for a teaching module but a poor choice for a production driver because any local user could open the device).

24.12.3 `ioctl` Returns “Inappropriate ioctl for device”

The kernel returned `ENOTTY`, which means it could not match the request code to any `cdevsw`. The two common causes are:

- The driver’s dispatcher returned `ENOIOCTL` for the command. The kernel translates `ENOIOCTL` to `ENOTTY` for user space. The fix is to add a case for the command in the dispatcher’s switch statement.
- The encoded length in the request code does not match the actual buffer size used by the program. This happens after a header refactor where the `_IOR` line was edited but the user-space program was not recompiled against the new header. The fix is to recompile the program against the current header and rebuild the module against the same source.

24.12.4 `ioctl` Returns “Bad file descriptor”

The dispatcher returned `EBADF`, which is the chapter’s pattern for “the file is not opened with the correct flags for this command”. The fix is to open the device with `O_RDWR` rather than `O_RDONLY` for any command that mutates state. The `myfirstctl` companion program already does this; a custom program may not.

24.12.5 `sysctl dev.myfirst.0` Shows the Tree but Reads Return “operation not supported”

This usually means the `sysctl` OID was added with a stale or invalid handler pointer. If the read returns immediately with `EOPNOTSUPP` (95), the cause is almost always that the OID was registered with `CTL_TYPE_OPAQUE` and a handler that does not call `SYSCTL_OUT`. The fix is to use one of the typed `SYSCTL_ADD_*` helpers (`SYSCTL_ADD_UINT`, `SYSCTL_ADD_STRING`, `SYSCTL_ADD_PROC` with the correct format string) so the framework knows what to do on a read.

24.12.6 `sysctl -w dev.myfirst.0.foo=value` Fails With “permission denied”

The OID was probably created with `CTLFLAG_RD` (read-only) when the writable variant `CTLFLAG_RW` was intended. Re-check the flag word in the `SYSCTL_ADD_*` call and rebuild.

If the flag is correct and the failure persists, the user may not be running as root. `Sysctl` writes require the `PRIV_SYSCTL` privilege by default; use `sudo` for the write.

24.12.7 `sysctl` Hangs or Causes a Deadlock

The OID handler is taking the giant lock (because `CTLFLAG_MPSAFE` is missing) at the same time as another thread is holding the giant lock and calling into the driver. The fix is to add `CTLFLAG_MPSAFE` to every OID’s flag word. Modern kernels assume `MPSAFE` everywhere; the absence of the flag is a code-review issue.

A subtler cause is a handler that takes the softc mutex while another thread is holding the softc mutex and reading from `sysctl`. Audit the handler: it should compute the value under the mutex but call `sysctl_handle_*` outside the mutex. The chapter’s `myfirst_sysctl_message_len` follows this pattern.

24.12.8 `kldunload myfirst` Fails With “Device busy”

The detach refused because some user holds the device open. Find them with `fstat | grep myfirst0` and either ask them to close it or kill the process. After they release the device, the unload will succeed.

If `fstat` shows nothing and the unload still fails, the cause is most likely a leaked `dev_ref`. Re-check that every code path in the driver that takes a `dev_ref` also calls `dev_rel`; in particular, any error path inside `myfirst_open` must release any reference taken before the failure.

24.12.9 `kldunload myfirst` Causes a Kernel Panic

The driver’s detach is destroying or freeing something that the kernel is still using. The two most common causes are:

- The detach freed the `softc` before destroying the `cdev`. The `cdev`’s callbacks may still be in flight; they dereference `si_drv1`, get garbage, and panic. The fix is the strict order: `destroy_dev` (which drains in-flight callbacks) first, then `mutex_destroy`, then `return`; the framework frees the `softc`.
- The detach forgot to deregister an event handler. The next event fires after unload and jumps into freed memory. The fix is to call `EVENTHANDLER_DEREGISTER` for every `EVENTHANDLER_REGISTER` done in `attach`.

The `lock_order reversal` and `WITNESS` messages in `dmesg` are useful diagnostics for both cases. A panic with `page fault while in kernel mode` and a corrupted `%rip` value is the second pattern; a panic with `lock_order reversal` and a stack trace through both subsystems is the first.

24.12.10 DTrace Probes Are Not Visible

`dtrace -l -P myfirst` returns nothing even though the module is loaded. The cause is almost always that the SDT probes are declared in a header but not defined anywhere. Probes need both `SDT_PROBE_DECLARE` (in the header, where consumers see them) and `SDT_PROBE_DEFINE*` (in exactly one source file, which owns the probe storage). The chapter’s pattern places the defines in `myfirst_debug.c`. If that file is not in `SRCS`, the probes will not be defined and DTrace will see nothing.

A subtler cause is the SDT probe was renamed in the header but the matching `SDT_PROBE_DEFINE*` was not updated. The build still succeeds because the two declarations refer to different symbols, but DTrace sees only the defined name. Audit the header and source for the same probe name.

24.12.11 `sysctl` Tree Survives Unload But Hangs the Next `Sysctl`

This happens when the driver uses its own `sysctl` context (rather than the per-device one) and forgets to call `sysctl_ctx_free` in `detach`. The `OIDs` reference fields in the now-freed `softc`; the next `sysctl` walk dereferences the freed memory and the kernel either panics or returns garbage. The fix is to switch to `device_get_sysctl_ctx`, which the framework cleans up automatically.

24.12.12 General Diagnostic Checklist

When something goes wrong and the cause is not obvious, walk through this short list before reaching for `kgdb`:

1. `kldstat | grep <driver>`: is the module actually loaded?
2. `dmesg | tail`: are there any kernel messages mentioning the driver?
3. `ls -l /dev/<driver>0`: does the device node exist with the expected mode?
4. `sysctl dev.<driver>.0.%driver`: is Newbus aware of the device?
5. `fstat | grep <driver>0`: is anyone holding the device open?
6. `dtrace -l -P <driver>`: are the SDT probes registered?
7. Re-read the attach function and check that every step has a matching cleanup in detach.

The first six commands take ten seconds and rule out the bulk of the common problems. The seventh is the slow one but is almost always the eventual answer for any bug that the first six did not surface.

24.12.13 Frequently Asked Questions

The questions below come up often enough during integration work that the chapter ends with a short FAQ. Each answer is intentionally compact; the relevant section of the chapter has the full discussion.

Q1. Why use both `ioctl` and `sysctl` when they seem to overlap?

They answer different questions. `ioctl` is for a program that has already opened the device and wants to issue a command (request a state, push new state, trigger an action). `sysctl` is for an operator at a shell prompt or a script that wants to inspect or adjust state without opening anything. The same value can be exposed through both interfaces, and many production drivers do exactly that: a `MYFIRSTIOC_GETMSG` for programs and a `dev.myfirst.0.message` for humans. Each user picks the channel that fits their context.

Q2. When should I use `mmap` instead of `read/write/ioctl`?

Use `mmap` when the data is large, accessed at random, and naturally lives at a memory address (a frame buffer, a DMA descriptor ring, a memory-mapped register space). Use `read/write` when the data is sequential, byte-oriented, and small per call. Use `ioctl` for control commands. The three are not in opposition; many drivers expose all three (as `vt(4)` does for the console).

Q3. Why does the chapter use `make_dev_s` rather than `make_dev`?

`make_dev_s` is the modern preferred form. It returns an explicit error rather than panicking on a duplicate name; it accepts an `args` structure so new options can be added without churn; and it is what most current drivers use. The older `make_dev` still works but is discouraged for new code.

Q4. Do I need to declare `D_TRACKCLOSE`?

You need it if your driver's `d_close` should be called only on the last close of a file descriptor (the natural meaning of “close”). Without it, the kernel calls `d_close` for every close of every duplicated descriptor, which surprises most drivers. Set it in any new `cdevsw` unless you have a specific reason not to.

Q5. When should I bump `MODULE_VERSION`?

When something in the driver's in-kernel API changes incompatibly. Adding new exported symbols is fine; renaming or removing them is a bump. Changing the layout of a publicly visible structure is a bump. Bumping the module version forces dependents (`MODULE_DEPEND` consumers) to be rebuilt.

Q6. When should I bump the API version constant in my public header?

When something in the user-visible interface changes incompatibly. Adding a new ioctl is fine; changing the layout of an existing ioctl's argument structure is a bump. Renumbering an existing ioctl is a bump. Bumping the API version lets user-space programs detect incompatibility before they issue calls.

Q7. Should I detach my OIDs in myfirst_detach?

No, not if you used `device_get_sysctl_ctx` (the per-device context). The framework cleans the per-device context up automatically after a successful detach. You only need explicit cleanup if you used `sysctl_ctx_init` to create your own context.

Q8. Why does my detach panic with “invalid memory access”?

Almost always because the cdev's callbacks are still in flight when the driver freed something they reference. The fix is to call `destroy_dev(sc->sc_cdev)` first; `destroy_dev` blocks until every in-flight callback returns. After it returns, the cdev is gone and no new callbacks can arrive. Only then is it safe to free the softc, free the locks, and so on. The strict order is non-negotiable.

Q9. What is the difference between dev_ref / dev_rel and D_TRACKCLOSE?

`D_TRACKCLOSE` is a cdevsw flag that controls when the kernel calls `d_close`: with it, only on the last close; without it, on every close. `dev_ref/dev_rel` is a reference-counting mechanism that lets the driver delay detach until outstanding references are released. They are unrelated and complementary. The chapter uses `D_TRACKCLOSE` in stage 1; Challenge 3 demonstrates `dev_ref/dev_rel`.

Q10. Why does my sysctl write return EPERM even though I am root?

Three possible causes. (a) The OID was created with `CTLFLAG_RD` only; add `CTLFLAG_RW`. (b) The OID has `CTLFLAG_SECURE` and the system is at `securelevel > 0`; lower the `securelevel` or remove the flag. (c) The user is not actually root but is in a jail without `allow.sysvipc` or similar; root inside a jail does not have `PRIV_SYSCTL` for arbitrary OIDs.

Q11. My sysctl handler takes the giant lock when it should not. What did I forget?

`CTLFLAG_MPSAFE` in the flag word. Without it, the kernel takes the giant lock around every call into the handler. Add it everywhere; modern kernels assume `MPSAFE` everywhere.

Q12. Should I name my ioctl group letter uppercase or lowercase?

Uppercase for new drivers. Lowercase letters are heavily used by base subsystems ('d' for disk, 'i' for `if_ioctl`, 't' for terminal) and the chance of collision is real. Uppercase letters are mostly free, and a fresh driver should choose one of them.

Q13. My ioctl returns inappropriate ioctl for device and I do not understand why.

The kernel returned `ENOTTY` because either (a) the dispatcher returned `ENOIOCTL` for the command (add a case for it), or (b) the encoded length in the request code does not match the buffer the user passed (recompile both sides against the same header).

Q14. Should I use strncpy or strlcpy in the kernel?

`strlcpy`. It guarantees NUL termination and never overruns the destination. `strncpy` does neither and is a frequent source of subtle bugs. The FreeBSD `style(9)` man page recommends `strlcpy` for all new code.

Q15. My module loads but dmesg shows no messages from my driver. What is wrong?

The driver's debug mask is zero. The chapter's `DPRINTF` macro prints only when the mask bit is set. Either set the mask before loading (`kenv hw.myfirst.debug_mask_default=0xff`), or set it after loading (`sysctl dev.myfirst.0.debug.mask=0xff`).

Q16. Why does the chapter mention DTrace so often?

Because it is the most productive debugging tool in the FreeBSD kernel and because the Chapter 23 debug infrastructure is designed to integrate with it. SDT probes give the operator a runtime tap into every integration surface without rebuilding the driver. A driver that exposes well-named SDT probes is much easier to debug than one that does not.

Q17. Can I use this driver as a template for a real hardware driver?

The integration surface (`cdev`, `ioctl`, `sysctl`) translates directly. The hardware-specific parts (resource allocation, interrupt handling, DMA setup) come in Chapters 18 to 22 of Part IV. A real PCI driver typically combines the structural patterns from Part IV with the integration patterns from this chapter to arrive at a shipping driver.

Q18. How do I grant a non-root user access to `/dev/myfirst0` without rebuilding the driver?

Use `devfs.rules(5)`. Add a rule file under `/etc/devfs.rules` that matches the device name and sets the owner, group, or mode at runtime. For example, to let group operator read and write `/dev/myfirst*`:

```
[myfirst_rules=10]
add path 'myfirst*' mode 0660 group operator
```

Enable the ruleset with `devfs_system_ruleset="myfirst_rules"` in `/etc/rc.conf` and service `devfs` restart. The driver's `mda_uid`, `mda_gid`, and `mda_mode` still set the defaults at creation time; `devfs.rules` lets the administrator override them without touching the source.

Q19. My SRCS list keeps growing. Is that a problem?

Not by itself. The `SRCS` line in the kernel-module `Makefile` lists every source file that compiles into the module; growing the list as new responsibilities get their own file is normal and expected. The chapter's Stage 3 driver already has four source files (`myfirst.c`, `myfirst_debug.c`, `myfirst_ioctl.c`, `myfirst_sysctl.c`), and Chapter 25 will add more. The warning sign is not the number of entries but the lack of structure: if `SRCS` contains unrelated files that were merged together without a naming scheme, the driver has outgrown its layout and deserves a small refactor. Chapter 25 treats that refactor as a first-class habit.

Q20. What should I do next?

Read Chapter 25 (advanced topics and practical tips) to turn this integrated driver into a *maintainable* one, work through the chapter's challenges if you want hands-on practice, and look at one of the in-tree drivers cited in the reference card for a fully worked example. The `null(4)` driver is the gentlest entry; the `if_em` Ethernet driver is the most complete; the `ahci(4)` storage driver shows the CAM patterns. Pick the one closest to what you want to build and read it end to end.

24.13 Wrapping Up

This chapter brought `myfirst` from a working but isolated module into a fully integrated FreeBSD driver. The arc was deliberate: each section added one concrete integration surface and ended with the driver more useful

and more discoverable than it began. The reader who walked the labs in Section 9 now has, on disk, a driver that exposes a properly constructed cdev under `/dev`, four well-designed ioctls under a public header, a self-describing sysctl tree under `dev.myfirst.0`, a boot-time tunable through `/boot/loader.conf`, and a clean lifecycle that survives load/unload cycles without leaking resources.

The technical milestones along the way were:

- Stage 1 (Section 2) replaced the older `make_dev` call with the modern `make_dev_args` form, populated `D_TRACKCLOSE`, wired `si_drv1` for per-cdev state, and walked the cdev lifecycle from creation through drain to destruction. The driver's `/dev` presence became first-class.
- Stage 2 (Section 3) added the `MYFIRSTIOC_GETVER`, `MYFIRSTIOC_GETMSG`, `MYFIRSTIOC_SETMSG`, and `MYFIRSTIOC_RESET` ioctls and the matching `myfirst_ioctl.h` public header. The dispatcher reuses the debug infrastructure from Chapter 23 (`MYF_DBG_IOCTL` and the `myfirst::ioctl` SDT probe). The companion `myfirstctl` user-space program demonstrated how a small command-line tool exercises every ioctl without ever decoding a request code by hand.
- Stage 3 (Section 4) added the `dev.myfirst.0.*` sysctl tree, including a `debug.` subtree that lets the operator inspect and modify the debug mask at runtime, a `version` OID that reports the integration release, counters for read and write activity, and a string OID for the current message. The boot-time tunable `hw.myfirst.debug_mask_default` lets the operator pre-load the debug mask before attach.
- Sections 5 and 6 sketched the same registration-style integration applied to the network stack (`if_alloc`, `if_attach`, `bpfattach`) and the CAM storage stack (`cam_sim_alloc`, `xpt_bus_register`, `xpt_action`). The reader who is not building a network or storage driver still gained a useful pattern: every framework registration in FreeBSD uses the same allocate-name-fill-attach-traffic-detach-free shape.
- Section 7 codified the lifecycle discipline that holds it all together: every successful registration must be paired with a deregistration in reverse order, and a failed attach must unwind every prior step before returning. The `goto err` chain is the canonical encoding of this rule.
- Section 8 framed the chapter as one step in a longer arc: `myfirst` started as a one-file demonstration, grew into a multi-file driver across Parts II to IV, gained debug and tracing in Chapter 23, and gained the integration surface here. The release version, the module version, and the API version each track a different aspect of that evolution; bumping each at the right time is the version discipline of a long-lived driver.

The chapter's labs (Section 9) walked the reader through every milestone, the challenges (Section 10) gave the motivated reader follow-on work, and the troubleshooting catalogue (Section 11) gathered the most common symptoms and fixes for quick reference.

The result is a driver milestone (1.7-integration) that the reader can take into the next chapter without any unfinished integration work waiting to bite them. The patterns from this chapter (cdev construction, ioctl design, sysctl trees, lifecycle discipline) are also the patterns that the rest of Part V and most of Parts VI and VII will assume the reader knows.

24.14 Bridge to Chapter 25

Chapter 25 (Advanced Topics and Practical Tips) closes Part 5 by turning the integrated driver of this chapter into a *maintainable* one. Where Chapter 24 added the interfaces that let the driver speak to the rest of the system, Chapter 25 teaches the engineering habits that keep those interfaces stable and readable as the driver absorbs the next year of bug fixes, portability changes, and feature requests. The driver grows from 1.7-integration to 1.8-maintenance; the visible additions are modest but the discipline behind them is what separates a driver that survives one development cycle from one that survives a decade.

The bridge from Chapter 24 to Chapter 25 has four concrete parts.

First, the rate-limited logging that Chapter 25 introduces sits directly on top of the `DPRINTF` macro from Chapter 23 and the integration surfaces this chapter added. A new `DLOG_RL` macro built around `ppsratecheck(9)` lets the driver keep the same debug classes it already uses but without flooding `dmesg` during an event storm. The discipline is small: choose a per-second limit, fold it into the existing debug call sites, and audit the handful of places where an unrestricted `device_printf` could run in a loop.

Second, the `ioctl` and `sysctl` paths this chapter built will be audited in Chapter 25 for a consistent `errno` vocabulary. The chapter distinguishes `EINVAL` from `ENXIO`, `ENOIOCTL` from `ENOTTY`, `EBUSY` from `EAGAIN`, and `EPERM` from `EACCES`, so that every integration surface returns the right code on every failure. The reader walks the dispatcher written in Section 3 and the `sysctl` handlers written in Section 4, and adjusts them where the wrong error was returned.

Third, the boot-time tunable `hw.myfirst.debug_mask_default` introduced in Section 4 will be generalised in Chapter 25 into a small but disciplined tunable vocabulary through `TUNABLE_INT_FETCH`, `TUNABLE_LONG_FETCH`, `TUNABLE_BOOL_FETCH`, and `TUNABLE_STR_FETCH`, cooperating with writable `sysctls` under `CTLFLAG_TUN`. The same `MYFIRST_VERSION`, `MODULE_VERSION`, and `MYFIRST_IOCTL_VERSION` triple this chapter settled on will be extended with a `MYFIRSTIOC_GETCAPS` `ioctl` so user-space tools can detect features at runtime without trial and error.

Fourth, the `goto_err` chain introduced in Section 7 will be promoted from a lab exercise into the driver's production cleanup pattern, and the chapter's refactor will move the Newbus attach logic and the `cdev` callbacks into separate files (`myfirst_bus.c` and `myfirst_cdev.c`) alongside a `myfirst_log.c` for the new logging macros. Chapter 25 also introduces `SYSINIT(9)` and `SYSUNINIT(9)` for driver-wide initialisation and a `shutdown_pre_sync` event handler through `EVENTHANDLER(9)`, adding two more registration-style surfaces to the ones this chapter already taught.

Read on with the confidence that the integration vocabulary is now in place. Chapter 25 takes this driver and makes it ready for the long haul; Part 6 then begins the transport-specific chapters that lean on every habit Part 5 has built.

24.15 Reference Card and Glossary

The remaining pages of the chapter are a compact reference. They are designed to be read straight through the first time and then jumped into when the reader needs to look something up. The order is: a reference card of the important macros, structures, and flags; a glossary of integration vocabulary; and a short directory of the companion files that ship with the chapter.

24.15.1 Quick Reference: cdev Construction

Function	When to use
<code>make_dev_args_init(args)</code>	Always before <code>make_dev_s</code> ; zeroes the args struct safely.
<code>make_dev_s(args, &cdev, fmt, ...)</code>	The modern preferred form. Returns 0 or <code>errno</code> .
<code>make_dev(devsw, unit, uid, gid, mode, fmt, ...)</code>	Older single-call form. Discouraged for new code.
<code>make_dev_credf(flags, ...)</code>	When you need <code>MAKEDEV_*</code> flag bits.
<code>destroy_dev(cdev)</code>	Always in detach; drains in-flight callbacks.
<code>destroy_dev_drain(cdev)</code>	When detach must wait for outstanding refs.

24.15.2 Quick Reference: cdevsw Flags

Flag	Meaning
<code>D_VERSION</code>	Required; identifies the cdevsw layout version.
<code>D_TRACKCLOSE</code>	Call <code>d_close</code> only on the last close. Recommended.
<code>D_NEEDGIANT</code>	Take the giant lock around every callback. Discouraged.
<code>D_DISK</code>	Cdev represents a disk; uses bio rather than uio for I/O.
<code>D_TTY</code>	Cdev is a terminal; affects line discipline routing.
<code>D_MMAP_ANON</code>	Cdev supports anonymous mmap.
<code>D_MEM</code>	Cdev is <code>/dev/mem</code> -like; raw memory access.

24.15.3 Quick Reference: make_dev Flags (MAKEDEV_*)

Flag	Meaning
<code>MAKEDEV_REF</code>	Take an extra reference; caller must <code>dev_ref</code> later.
<code>MAKEDEV_NOWAIT</code>	Do not sleep; return <code>ENOMEM</code> if no memory.
<code>MAKEDEV_WAITOK</code>	Sleeping is allowed (default for most callers).
<code>MAKEDEV_ETERNAL</code>	Cdev never goes away; certain optimisations apply.
<code>MAKEDEV_ETERNAL_KLD</code>	Like <code>ETERNAL</code> , but only for the lifetime of the kld.
<code>MAKEDEV_CHECKNAME</code>	Validate the name; <code>ENAMETOOLONG</code> if too long.

24.15.4 Quick Reference: ioctl Encoding Macros

All in `/usr/src/sys/sys/ioccom.h`.

Macro	Direction	Argument
<code>_IO(g, n)</code>	none	none
<code>_IOR(g, n, t)</code>	out	type <code>t</code> , size <code>sizeof(t)</code>
<code>_IOW(g, n, t)</code>	in	type <code>t</code> , size <code>sizeof(t)</code>
<code>_IOWR(g, n, t)</code>	in and out	type <code>t</code> , size <code>sizeof(t)</code>
<code>_IOWINT(g, n)</code>	none, but value of <code>int</code> passed	<code>int</code>

The arguments mean:

- `g`: group letter, conventionally 'M' for `myfirst` and similar.
- `n`: command number, monotonically increasing within the group.
- `t`: argument type, used only for its `sizeof`.

The maximum size is `IOCPARM_MAX` = 8192 bytes. For larger transfers, use the embedded-pointer pattern (Challenge 1) or a different mechanism such as `mmap` or `read/write`.

24.15.5 Quick Reference: `d_ioctl_t` Signature

```
int d_ioctl(struct cdev *dev, u_long cmd, caddr_t data,
            int fflag, struct thread *td);
```

Argument	Meaning
<code>dev</code>	The <code>cdev</code> backing the file descriptor. Use <code>dev->si_drv1</code> to get the softc.
<code>cmd</code>	The request code from user space. Compared to named constants.
<code>data</code>	Kernel-side buffer with the user data. Direct dereference; no copyin needed.
<code>fflag</code>	File flags from the open call (<code>FREAD</code> , <code>FWRITE</code>). Check before mutating.
<code>td</code>	Calling thread. Use <code>td->td_ucred</code> for credentials.

Return 0 on success, a positive `errno` on failure, or `ENOIOCTL` for unknown commands.

24.15.6 Quick Reference: sysctl OID Macros

All in `/usr/src/sys/sys/sysctl.h`.

Macro	Adds
<code>SYSCTL_ADD_INT(ctx, parent, nbr, name, flags, ptr, val, descr)</code>	Signed int, backed by <code>*ptr</code> .
<code>SYSCTL_ADD_UINT</code>	Unsigned int.
<code>SYSCTL_ADD_LONG / SYSCTL_ADD_ULONG</code>	Long / unsigned long.
<code>SYSCTL_ADD_S64 / SYSCTL_ADD_U64</code>	64-bit signed / unsigned.
<code>SYSCTL_ADD_BOOL</code>	Boolean (preferred over int 0/1).
<code>SYSCTL_ADD_STRING(ctx, parent, nbr, name, flags, ptr, len, descr)</code>	NUL-terminated string.
<code>SYSCTL_ADD_NODE(ctx, parent, nbr, name, flags, handler, descr)</code>	Subtree node.
<code>SYSCTL_ADD_PROC(ctx, parent, nbr, name, flags, arg1, arg2, handler, fmt, descr)</code>	Handler-backed OID.

24.15.7 Quick Reference: sysctl Flag Bits

Flag	Meaning
<code>CTLFLAG_RD</code>	Read-only.
<code>CTLFLAG_WR</code>	Write-only (rare).
<code>CTLFLAG_RW</code>	Read-write.
<code>CTLFLAG_TUN</code>	Loader tunable; read at boot from <code>/boot/loader.conf</code> .
<code>CTLFLAG_MPSAFE</code>	Handler is safe without giant lock. Always set for new code.
<code>CTLFLAG_PRISON</code>	Visible inside jails.
<code>CTLFLAG_VNET</code>	Per-VNET (virtualised network stack).
<code>CTLFLAG_DYN</code>	Dynamic OID; set automatically by <code>SYSCTL_ADD_*</code> .
<code>CTLFLAG_SECURE</code>	Read-only when <code>securelevel > 0</code> .

24.15.8 Quick Reference: sysctl Type Bits

OR'd into the flag word for `SYSCTL_ADD_PROC` and similar.

Flag	Meaning
CTLTYPE_INT / CTLTYPE_UINT	Signed / unsigned int.
CTLTYPE_LONG / CTLTYPE_ULONG	Long / unsigned long.
CTLTYPE_S64 / CTLTYPE_U64	64-bit signed / unsigned.
CTLTYPE_STRING	NUL-terminated string.
CTLTYPE_OPAQUE	Opaque blob; rarely used in new code.
CTLTYPE_NODE	Subtree node.

24.15.9 Quick Reference: sysctl Handler Format Strings

Used by SYSCTL_ADD_PROC to tell sysctl(8) how to print the value.

Format	Type
"I"	int
"IU"	unsigned int
"L"	long
"LU"	unsigned long
"Q"	int64
"QU"	uint64
"A"	NUL-terminated string
"S, structname"	opaque struct (rare)

24.15.10 Quick Reference: sysctl Handler Boilerplate

```
static int
my_handler(SYSCTL_HANDLER_ARGS)
{
    struct my_softc *sc = arg1;
    u_int val;

    /* Read the current value into val under the mutex. */
    mtx_lock(&sc->sc_mtx);
    val = sc->sc_field;
    mtx_unlock(&sc->sc_mtx);

    /* Let the framework do the read or write. */
}
```

```

        return (sysctl_handle_int(oidp, &val, 0, req));
    }

```

24.15.11 Quick Reference: per-Device Sysctl Context

```

struct sysctl_ctx_list *ctx = device_get_sysctl_ctx(dev);
struct sysctl_oid      *tree = device_get_sysctl_tree(dev);
struct sysctl_oid_list *child = SYSCTL_CHILDREN(tree);

```

The framework owns the ctx; the driver does not call `sysctl_ctx_free` on it. The framework cleans up automatically after a successful detach.

24.15.12 Quick Reference: Loader Tunables

```

TUNABLE_INT_FETCH("hw.driver.knob", &sc->sc_knob);
TUNABLE_LONG_FETCH("hw.driver.knob", &sc->sc_knob);
TUNABLE_STR_FETCH("hw.driver.knob", buf, sizeof(buf));

```

The first argument is the loader variable name. The second is a pointer to the destination, which is also the default value if the variable is not set.

24.15.13 Quick Reference: ifnet Lifecycle

Function	When
<code>if_alloc(IFT_<type>)</code>	Allocate the ifnet.
<code>if_initname(ifp, name, unit)</code>	Set the user-visible name (ifconfig shows it).
<code>if_setflags(ifp, flags)</code>	Set <code>IFF_*</code> flags.
<code>if_setsoftc(ifp, sc)</code>	Attach the driver's softc.
<code>if_setioctlfn(ifp, fn)</code>	Set the ioctl handler.
<code>if_settransmitfn(ifp, fn)</code>	Set the transmit function.
<code>if_attach(ifp)</code>	Make the interface visible.
<code>bpfattach(ifp, dlt, hdrlen)</code>	Make traffic visible to BPF.
<code>bpfdetach(ifp)</code>	Reverse <code>bpfattach</code> .
<code>if_detach(ifp)</code>	Reverse <code>if_attach</code> .

Function	When
<code>if_free(ifp)</code>	Free the ifnet.

24.15.14 Quick Reference: CAM SIM Lifecycle

Function	When
<code>cam_simq_alloc(maxq)</code>	Allocate the device queue.
<code>cam_sim_alloc(action, poll, name, sc, unit, mtx, max_tagged, max_dev_tx, devq)</code>	Allocate the SIM.
<code>xpt_bus_register(sim, dev, 0)</code>	Register the bus with CAM.
<code>xpt_create_path(&path, NULL, cam_sim_path(sim), targ, lun)</code>	Create a path for events.
<code>xpt_action(ccb)</code>	Send a CCB to CAM.
<code>xpt_done(ccb)</code>	Tell CAM that the driver finished a CCB.
<code>xpt_free_path(path)</code>	Reverse <code>xpt_create_path</code> .
<code>xpt_bus_deregister(cam_sim_path(sim))</code>	Reverse <code>xpt_bus_register</code> .
<code>cam_sim_free(sim, free_devq)</code>	Reverse <code>cam_sim_alloc</code> . Pass TRUE to also free the devq.

24.15.15 Quick Reference: Module Lifecycle

```
static moduledata_t mymod = {
    "myfirst",          /* name */
    myfirst_modevent, /* event handler, can be NULL */
    NULL                /* extra data, rarely used */
};
DECLARE_MODULE(myfirst, mymod, SI_SUB_DRIVERS, SI_ORDER_ANY);
MODULE_VERSION(myfirst, 1);
MODULE_DEPEND(myfirst, otherdriver, 1, 1, 1);
```

The event handler signature is `int (*)(module_t mod, int what, void *arg)`. The `what` argument is one of `MOD_LOAD`, `MOD_UNLOAD`, `MOD_QUIESCE`, or `MOD_SHUTDOWN`. Return 0 on success or a positive errno.

24.15.16 Quick Reference: Event Handler

```
eventhandler_tag tag;

tag = EVENTHANDLER_REGISTER(event_name, callback,
    arg, EVENTHANDLER_PRI_ANY);

EVENTHANDLER_DEREGISTER(event_name, tag);
```

Common event names: shutdown_pre_sync, shutdown_post_sync, shutdown_final, vm_lowmem, power_suspend_early, power_resume.

24.15.17 Quick Reference: Errno Conventions

Errno	When to return
0	Success.
EINVAL	The arguments are recognised but invalid.
EBADF	The file descriptor is not opened correctly for the operation.
EBUSY	The resource is in use (often returned from detach).
ENOIOCTL	The ioctl command is unknown. Use this for the default case in d_ioctl.
ENOTTY	The kernel translation of ENOIOCTL for user space.
ENOMEM	Allocation failed.
EAGAIN	Try again later (often returned from non-blocking I/O).
EPERM	The caller lacks the necessary privilege.
EOPNOTSUPP	The operation is not supported by this driver.
EFAULT	A user pointer is invalid (returned by failed copyin/copyout).
ETIMEDOUT	A wait timed out.
EIO	Generic I/O error from hardware.

24.15.18 Quick Reference: Debug Class Bits (from Chapter 23)

Bit	Name	Used for
0x01	MYF_DBG_INIT	probe / attach / detach
0x02	MYF_DBG_OPEN	open / close lifecycle
0x04	MYF_DBG_IO	read / write paths

Bit	Name	Used for
0x08	MYF_DBG_IOCTL	ioctl handling
0x10	MYF_DBG_INTR	interrupt handler
0x20	MYF_DBG_DMA	DMA mapping/sync
0x40	MYF_DBG_PWR	power-management events
0x80	MYF_DBG_MEM	malloc/free trace
0xFFFFFFFF	MYF_DBG_ANY	all classes
0	MYF_DBG_NONE	no logging

The mask is set per instance via `dev.<driver>.<unit>.debug.mask`, or globally at boot via `hw.<driver>.debug_mask_de` in `/boot/loader.conf`.

24.15.19 Glossary of Integration Vocabulary

API version: An integer exposed through a driver’s `ioctl` interface (typically through a `GETVER` `ioctl`) that user-space programs can query to detect changes in the driver’s public interface. Bumped only when the user-visible interface changes incompatibly.

bpfattach: The function that hooks an `ifnet` into the BPF (Berkeley Packet Filter) machinery so `tcpdump` and similar tools can observe its traffic. Must be paired with `bpfdetach`.

bus_generic_detach: A helper that detaches every child of a bus-style driver. Used as the first step in a bus driver’s detach to release child devices before the parent tears down its own state.

CAM: The Common Access Method, FreeBSD’s storage subsystem above device drivers. Owns the I/O queue, target/LUN abstraction, and peripheral drivers (`da`, `cd`, `sa`).

CCB: CAM Control Block. A single I/O request structured as a tagged union; the driver inspects `ccb->ccb_h.func_code` and dispatches accordingly. Completed via `xpt_done`.

cdev: Character device. The kernel’s per-device-node object that backs an entry under `/dev`. Created with `make_dev_s`, destroyed with `destroy_dev`.

cdevsw: Character device switch. The static table of callbacks (`d_open`, `d_close`, `d_read`, `d_write`, `d_ioctl`, ...) that the kernel invokes on operations against a `cdev`.

copyin / copyout: Functions that transfer bytes between user-space and kernel-space addresses. The kernel performs them automatically for properly encoded `ioctls`; the driver calls them explicitly only for embedded-pointer patterns.

CTLFLAG_MPSAFE: A `sysctl` flag that declares the `OID`’s handler safe to call without the giant lock. Mandatory for new code; without it the kernel takes the giant lock around every access.

d_ioctl_t: The function-pointer type for the `cdevsw`’s `ioctl` callback. Signature: `int (*)(struct cdev *, u_long, caddr_t, int, struct thread *)`.

d_priv: A per-file-descriptor private pointer attached via `devfs_set_cdevpriv`. Used for state that must be associated with one open rather than with the driver instance as a whole.

dev_ref / dev_rel: A pair of functions that increment and decrement a cdev's reference count. Used to coordinate detach with in-flight callbacks; see Challenge 3.

devfs: The kernel-managed file system that backs /dev. The driver creates cdevs and devfs makes them visible.

devfs.rules(8): A configuration mechanism for runtime devfs permissions. Applied by service devfs restart after editing /etc/devfs.rules.

DTrace: The dynamic tracing framework. Drivers expose probe points through SDT macros; DTrace scripts attach to them at runtime.

EVENTHANDLER_REGISTER: The mechanism for registering a callback for a named system-wide event (shutdown_pre_sync, vm_lowmem, etc.). Must be paired with EVENTHANDLER_DEREGISTER.

ifnet: The network stack's per-interface object. The network counterpart to the cdev.

if_t: The opaque typedef the network stack uses for ifnet. Drivers manipulate the interface through accessor functions rather than direct field access.

IOC_VOID / IOC_IN / IOC_OUT / IOC_INOUT: The four direction bits encoded in an ioctl request code. Used by the kernel to decide what copyin/copyout to perform.

IOCPARM_MAX: The maximum size (8192 bytes) of an ioctl's argument structure as encoded in the request code. Larger transfers require an embedded-pointer pattern.

kldload / kldunload: The user-space tools that load and unload kernel modules. Both invoke the corresponding module event handler (MOD_LOAD and MOD_UNLOAD).

make_dev_args: The structure passed to make_dev_s to describe a new cdev. Initialised with make_dev_args_init.

make_dev_s: The modern preferred function for creating a cdev. Returns 0 or a positive errno; sets *cdev on success.

MAKEDEV_*: Flag bits passed to make_dev_credf and similar. Common bits: MAKEDEV_REF, MAKEDEV_NOWAIT, MAKEDEV_ETERNAL_KLD.

MOD_LOAD / MOD_UNLOAD / MOD_SHUTDOWN: The events delivered to a module's event handler. Returns 0 to acknowledge or non-zero to reject.

MODULE_DEPEND: The macro that declares a module's dependency on another. The kernel uses the version arguments (min, pref, max) to enforce compatibility.

MODULE_VERSION: The macro that declares a module's version number. Bumped when in-kernel users would need a recompile.

Newbus: FreeBSD's device-tree framework. Owns the device_t, the per-device softc, the per-device sysctl context, and the probe/attach/detach lifecycle.

OID: Object identifier. A node in the sysctl tree. Static OIDs are declared at compile time; dynamic OIDs are added at runtime with SYSCTL_ADD_*.

Path (CAM): A (bus, target, LUN) triple identifying a CAM destination. Created with xpt_create_path.

Public header: A header that user-space programs include to talk to the driver. Must compile cleanly without _KERNEL defined; uses only stable types.

Registration framework: A FreeBSD subsystem that exposes an “allocate-name-fill-attach-traffic-detach-free” interface for drivers. Examples: networking (ifnet), storage (CAM), sound, USB, GEOM.

Release version: The human-readable string identifying a driver’s release. Exposed through sysctl as `dev.<driver>.<unit>.version`.

SDT: Statically defined tracing. The kernel’s mechanism for compile-time probe points consumable by DTrace.

si_drv1 / si_drv2: Two private pointer fields in `struct cdev` available for the driver’s use. Conventionally `si_drv1` points at the `softc`.

SIM: SCSI Interface Module. CAM’s view of a storage adapter. Allocated with `cam_sim_alloc`, registered with `xpt_bus_register`.

Soft detach: A detach pattern where the driver waits for outstanding references to drop to zero rather than refusing detach immediately. See Challenge 3.

Softc: Software context. The driver’s per-instance state. Allocated by Newbus and accessed via `device_get_softc(dev)`.

SYSINIT: A compile-time-registered one-shot kernel init function. Runs at a specific stage during boot. Rarely needed in driver code.

SYSCTL_HANDLER_ARGS: A macro that expands to the standard argument list for a sysctl handler: `oidp, arg1, arg2, req`.

TUNABLE_INT_FETCH: A function that reads a value from the loader environment and writes it into a kernel variable. The variable retains its previous value if the loader variable is absent.

XPT: CAM Transport Layer. The central CAM dispatch mechanism. Driver calls `xpt_action` to send a CCB; CAM calls back via the SIM’s action function for I/O CCBs.

24.15.20 Companion File Inventory

The companion files for this chapter live under `examples/part-05/ch24-integration/` in the book’s repository. The directory layout is:

```
examples/part-05/ch24-integration/
├─ README.md
├─ INTEGRATION.md
├─ stage1-devfs/
│   └─ Makefile
│   └─ myfirst.c           (with make_dev_args)
│   └─ README.md
├─ stage2-ioctl/
│   └─ Makefile
│   └─ Makefile.user      (for myfirstctl)
│   └─ myfirst_ioctl.c
│   └─ myfirst_ioctl.h    (PUBLIC)
│   └─ myfirstctl.c
```

```

|   └─ README.md
└─ stage3-sysctl/
|   └─ Makefile
|   └─ myfirst.c
|   └─ myfirst_sysctl.c
|   └─ README.md
└─ labs/
    └─ lab24_1_stage1.sh      (verification commands for Lab 1)
    └─ lab24_2_stage2.sh
    └─ lab24_3_stage3.sh
    └─ lab24_4_failure.sh
    └─ lab24_5_dtrace.sh
    └─ lab24_6_smoke.sh
    └─ lab24_7_reload.sh
    └─ loader.conf.example

```

The starting point for Lab 1 is the reader's own end-of-Chapter-23 driver (1.6-debug); `stage1-devfs/`, `stage2-ioctl/`, and `stage3-sysctl/` are reference solutions the reader can consult after completing each lab. The `labs` directory has small shell scripts that perform the verification commands and that the reader can adapt for their own tests.

The `README.md` at the chapter root describes how to use the directory, the order of the stages, and the relationship between the staged trees. The `INTEGRATION.md` is a longer document that maps every concept in the chapter to the file it appears in.

24.15.21 Where the Chapter's Source Tree Sits in Real FreeBSD

For a reader who wants to look up the in-tree implementations referenced throughout the chapter, here is a short index of the most important files:

Concept	In-tree file
<code>ioctl</code> encoding	<code>/usr/src/sys/sys/ioccom.h</code>
<code>cdevsw</code> definition	<code>/usr/src/sys/sys/conf.h</code>
<code>make_dev</code> family	<code>/usr/src/sys/kern/kern_conf.c</code>
<code>sysctl</code> framework	<code>/usr/src/sys/sys/sysctl.h,</code> <code>/usr/src/sys/kern/kern_sysctl.c</code>
<code>ifnet</code> API	<code>/usr/src/sys/net/if.h,</code> <code>/usr/src/sys/net/if.c</code>
<code>ifnet</code> example	<code>/usr/src/sys/net/if_disc.c</code>
CAM SIM API	<code>/usr/src/sys/cam/cam_xpt.h,</code> <code>/usr/src/sys/cam/cam_sim.h</code>
CAM example	<code>/usr/src/sys/dev/ahci/ahci.c</code>

Concept	In-tree file
EVENTHANDLER	<code>/usr/src/sys/sys/eventhandler.h</code>
MODULE machinery	<code>/usr/src/sys/sys/module.h,</code> <code>/usr/src/sys/kern/kern_module.c</code>
TUNABLE	<code>/usr/src/sys/sys/sysctl.h</code> (search for <code>TUNABLE_INT_FETCH</code>)
SDT probes	<code>/usr/src/sys/sys/sdt.h,</code> <code>/usr/src/sys/cddl/dev/sdt/sdt.c</code>
The <code>null(4)</code> reference	<code>/usr/src/sys/dev/null/null.c</code>

Reading these files alongside the chapter is the next step for any reader who wants to deepen their integration knowledge. The `null(4)` driver in particular is worth reading in full; it is small enough to absorb in one sitting and demonstrates almost every pattern this chapter covered.

24.15.22 What Did Not Make It Into the Chapter

A handful of integration topics belong to FreeBSD's wider toolbox but did not earn a section here, either because they are subsystem-specific in a way that would distract a beginner or because they are covered more thoroughly in a later chapter. Naming them here keeps the chapter honest about its scope and gives the reader a forward-looking map.

The first omission is `geom(4)`. A driver that exposes a block device hooks into GEOM rather than CAM. The registration pattern is similar to the `cdev` pattern (allocate a `g_geom`, fill in `g_class` callbacks, call `g_attach`), but the vocabulary is different enough that mixing it into the chapter would have blurred the storage-versus-character distinction. Drivers for raw disks and pseudo-disk targets live in this neighbourhood; the canonical reference is `/usr/src/sys/geom/geom_disk.c`.

The second omission is `usb(4)`. A USB driver registers with the USB stack through `usb_attach` and a USB-specific method table rather than through Newbus directly. The integration surfaces (`devfs`, `sysctl`) are the same once the device is attached, but the upper edge is owned by the USB stack. The canonical references are under `/usr/src/sys/dev/usb/`.

The third omission is `iicbus(4)` and `spibus(4)`. Drivers that talk to I2C or SPI peripherals attach as children of a bus driver and use bus-specific transfer routines. The integration surfaces remain the same, but the device-tree and FDT integration that drives modern Arm SoCs adds vocabulary that warrants its own chapter. Part VI covers these surfaces in their proper context.

The fourth omission is the `kqueue(2)` and `poll(2)` integration. A character driver that wants to wake user-space programs blocked on `select`, `poll`, or `kqueue` must implement `d_kqfilter` (and optionally `d_poll`), wire `selwakeup` and `KNOTE` into the data path, and supply a small set of filter operations. The mechanism is not difficult, but it is conceptually a layer above the basic `cdev` contract; we will return to it in Chapter 26.

A reader who needs any of these surfaces today should treat the chapter's pattern as the foundation and reach for the in-tree references named above. The discipline (register at attach, drain at detach, hold a single mutex across mutating callbacks, version the public surface) is the same.

24.15.23 Reader Self-Check

Before turning the page, the reader who has worked through the chapter should be able to answer the following questions without consulting the text. Each question maps to a section that introduced the underlying material. If a question is unfamiliar, the chapter section listed in parentheses is the right place to revisit before moving on.

1. What does `D_TRACKCLOSE` change about how `d_close` is invoked? (Section 2)
2. Why is `mda_si_drv1` preferable to assigning `si_drv1` after `make_dev` returns? (Section 2)
3. What does the `_IOR('M', 1, uint32_t)` macro encode in the resulting request code? (Section 3)
4. Why must the dispatcher's default branch return `ENOIOCTL` rather than `EINVAL`? (Section 3)
5. Which kernel function tears down the per-device `sysctl` context, and when does it run? (Section 4)
6. How does `CTLFLAG_TUN` cooperate with `TUNABLE_INT_FETCH` to apply a boot-time value? (Section 4)
7. What is the difference between the `MYFIRST_VERSION` string, the `MODULE_VERSION` integer, and the `MYFIRST_IOCTL_VERSION` integer? (Section 8)
8. Why does the cleanup chain in `attach` use labelled `gotos` in reverse order rather than nested `if` statements? (Section 7)
9. How does the chapter's soft-detach pattern differ from the `dev_ref/dev_rel` pattern that Challenge 3 sketches? (Sections 7 and 10)
10. Which two integration surfaces are required for almost every driver, and which two are required only for drivers that join a specific subsystem? (Sections 1, 2, 3, 4, 5, 6)

A reader who answers most of these without hesitation has internalised the chapter and is ready for what comes next. A reader who hesitates on more than two should revisit the relevant sections before tackling Chapter 25's maintenance discipline.

24.15.24 Final Word

Integration is what turns a working module into a usable driver. The patterns in this chapter are not optional polish; they are the difference between a driver that an operator can adopt and one that an operator has to fight. Master them once, and every subsequent driver gets easier to build, easier to maintain, and easier to ship.

The next chapter takes the discipline introduced here and generalises it into a set of maintenance habits: rate-limited logging, consistent `errno` vocabulary, tunables and versioning, production-grade cleanup, and the `SYSINIT/SYUNINIT/EVENTHANDLER` mechanisms that extend a driver's lifecycle beyond simple load and unload. The vocabulary changes, but the rhythm is the same: register, take traffic, deregister cleanly. With the foundation of Chapter 24 in place, Chapter 25 will feel like a natural extension rather than a new world.

One last thought before turning the page. The integration surfaces in this chapter are deliberately small in number. There are `devfs`, `ioctl`, and `sysctl`. There are the optional registrations to subsystems like `ifnet` and `CAM`. There is the lifecycle discipline that ties them together. Five concepts, in total.

Once these are familiar, the rest of the book is the application of the same patterns to ever more interesting hardware. A reader who has finished this chapter has finished the kernel-API half of the book; what remains is the systems and the discipline to wield it well.

25 Advanced Topics and Practical Tips

25.1 Reader Guidance & Outcomes

Chapter 24 closed with a driver that the rest of the system can talk to. The `myfirst` driver at version 1.7-integration has a clean `/dev/myfirst0` node created through `make_dev_s`, a public `ioctl` header shared between the kernel and user space, a per-instance `sysctl` subtree under `dev.myfirst.0`, a boot-time tunable for the debug mask, and an `ioctl` dispatcher that obeys the `ENOIOCTL` fallback rule so that kernel helpers like `FIONBIO` still reach the `cdev` layer correctly. The driver compiles, loads, runs under stress, and survives repeated `kldload` and `kldunload` cycles without leaking OIDs or `cdev` nodes. In every observable sense, the driver works.

Chapter 25 is about the difference between a driver that *works* and a driver that is *maintainable*. Those two qualities are not the same, and the difference shows up slowly. A working driver passes its first round of testing, attaches cleanly to its hardware, and goes into use. A maintainable driver also does that, and then absorbs the next year of bug fixes, feature additions, portability changes, new hardware revisions, and kernel API churn without slowly collapsing under its own weight. The first gives a developer a good day. The second gives the driver a good decade.

Chapter 25 is the closing chapter of Part 5. Where Chapter 23 taught observability and Chapter 24 taught integration, Chapter 25 teaches the engineering habits that preserve both of those qualities over time. Part 6 begins immediately after with transport-specific drivers (USB in Chapter 26, storage in Chapter 27, networking in Chapter 28 and beyond), and each of those chapters will assume the discipline introduced here. Without rate-limited logging, a USB hotplug storm fills the message buffer. Without a consistent error convention, a storage driver and its peripheral in CAM disagree about what `EBUSY` means. Without loader tunables, a network driver with a suboptimal default queue depth cannot be tuned on a production box without a rebuild. Without versioning discipline, a user-space tool written for this version of the driver silently misinterprets a new field added two months later. Each habit is small. Together, they are what make a driver a long-lived piece of FreeBSD rather than a short-lived lab experiment.

The chapter's running example remains the `myfirst` driver. At the start of the chapter it is at version 1.7-integration. By the end of the chapter it is at version 1.8-maintenance, split into more files than before, logging without flooding the message buffer, returning errors from a consistent vocabulary, configurable from `/boot/loader.conf`, shipping with a `MAINTENANCE.md` document that explains the ongoing maintenance contract, announcing its events through the `devctl` channel, and hooked into the kernel's shutdown and low-memory events through `EVENTHANDLER(9)`. None of those additions require new hardware knowledge. All of them require sharper discipline.

Part 5 closes here with the habits that keep the driver coherent as it grows. Chapter 22 made the driver survive a change of power state. Chapter 23 made the driver tell you what it is doing. Chapter 24 made the driver fit into the rest of the system. Chapter 25 makes the driver keep all of those qualities as it evolves. Chapter 26 will then open Part 6 by putting these qualities to work against a real transport, the Universal Serial Bus, where every shortcut in logging or in failure-path handling is exposed by the speed and variety of USB traffic.

25.1.1 Why Maintenance Discipline Earns a Chapter of Its Own

Before we go further, it is worth pausing on whether rate-limited logging, errno vocabulary, and loader tunables really deserve a full chapter. The earlier chapters have already taught so much. Adding a `ppsratecheck(9)`-backed log macro looks small. Standardising error codes looks even smaller. Why spread the work across a long chapter when each habit looks like a handful of lines?

The answer is that each habit is small, and the absence of each habit is large. A driver that logs without rate limiting is fine in the lab and catastrophic in production the first time a flaky cable triggers ten thousand re-enumerations per second. A driver that returns `EINVAL` when it should return `ENXIO`, and `ENXIO` when it should return `ENOIOCTL`, is fine when the author is the only caller and a bug report waiting to happen when a second developer writes the first user-space helper. A driver that lets every configuration default be a compile-time constant is fine for one person and unworkable for a team maintaining the same module across several production boxes with different workloads. Chapter 25 spends time on each of these habits because the value is measured not in the lab but in the two-year maintenance cost that each habit reduces.

The first reason the chapter earns its place is that **these habits shape what the driver's code base looks like as it grows**. A reader who has followed Chapters 23 and 24 has already seen the driver split into multiple files: `myfirst.c`, `myfirst_debug.c`, `myfirst_ioctl.c`, `myfirst_sysctl.c`. That was modularisation done in the small, one surface at a time. Chapter 25 revisits modularisation with the question most readers have not yet asked: *what does a maintainable source layout look like once the driver has a dozen files and three developers?* The chapter answers that question with a conscious split between the Newbus attach layer, the `cdev` layer, the `ioctl` layer, the `sysctl` layer, and the logging layer, and then uses that split to support every other habit the chapter teaches.

The second reason is that **these habits determine whether the driver can be debugged in production**. A driver that logs judiciously and returns informative errors gives an operator enough information to file a useful bug report. A driver that logs too much or too little, or that invents its own errno conventions, forces the operator to reason from symptoms alone, and the developer ends up chasing intermittent problems blind. Chapter 23's debugging toolkit is effective, but it depends on the driver cooperating. The cooperation is built here.

The third reason is that **these habits make the driver extensible without breaking its callers**. The `myfirst_ioctl.h` header from Chapter 24 is already a contract between the driver and user space. Chapter 25 teaches the reader how to evolve that contract, add a new `ioctl` that older user-space programs can ignore safely, retire a deprecated `sysctl` without breaking administrators' scripts, and bump the driver's version in a way that external consumers can check at run time. Without those habits, the first v2 of the driver forces every caller to be rewritten. With them, the driver can add features for a decade and still run the user-space helpers that were compiled the first week the driver shipped.

Chapter 25 earns its place by teaching those three ideas together, concretely, with the `myfirst` driver as the running example. A reader who finishes Chapter 25 can make any FreeBSD driver ready for long-term maintenance, can read another driver's production-hardening patterns and recognise which are principled and which are ad hoc, can negotiate compatibility with existing user-space tools, and has a `myfirst` driver at version 1.8-maintenance that is visibly ready to begin Part 6.

25.1.2 Where Chapter 24 Left the Driver

A brief recap of where you should stand. Chapter 25 extends the driver produced at the end of Chapter 24 Stage 3, tagged as version 1.7-integration. If any of the items below is uncertain, return to Chapter 24 and fix it before starting this chapter, because the new material assumes every Chapter 24 primitive is working.

- Your driver compiles cleanly and identifies itself as 1.7-integration in the output of `kldstat -v`.
- A `/dev/myfirst0` node exists after `kldload`, has ownership `root:wheel` and mode `0660`, and disappears cleanly on `kldunload`.
- The module exports four ioctls: `MYFIRSTIOC_GETVER`, `MYFIRSTIOC_GETMSG`, `MYFIRSTIOC_SETMSG`, and `MYFIRSTIOC_RESET`. The small `myfirstctl` user-space program from Chapter 24 exercises each one and returns success on all four.
- The `sysctl` subtree `dev.myfirst.0` lists at least `version`, `open_count`, `total_reads`, `total_writes`, `message`, `message_len`, `debug.mask`, and `debug.classes`.
- `sysctl dev.myfirst.0.debug.mask=0xff` enables every debug class, and the driver's subsequent log output shows the expected tags.
- The boot-time tunable `hw.myfirst.debug_mask_default`, placed in `/boot/loader.conf`, applies before attach and sets the initial value of the `sysctl`.
- Repeated `kldload` and `kldunload` in a loop for a minute leave no residual OIDs, no orphan `cdev`, and no leaked memory as reported by `vmstat -m | grep myfirst`.
- Your working tree contains `HARDWARE.md`, `LOCKING.md`, `SIMULATION.md`, `PCI.md`, `INTERRUPTS.md`, `MSIX.md`, `DMA.md`, `POWER.md`, `DEBUG.md`, and `INTEGRATION.md` from the earlier chapters.
- Your test kernel has `INVARIANTS`, `WITNESS`, `WITNESS_SKIPSPIN`, `DDB`, `KDB`, `KDB_UNATTENDED`, `KDTRACE_HOOKS`, and `DDB_CTF` enabled. Chapter 25's labs rely on `WITNESS` and on `INVARIANTS` just as strongly as Chapter 24's did.

That driver is what Chapter 25 extends. The additions are smaller in code lines than in any previous Part 5 chapter, but larger in conceptual surface. The new pieces are: a `myfirst_log.c` and `myfirst_log.h` pair built around `ppsratecheck(9)`, a labelled-goto cleanup chain in `myfirst_attach`, a refined error vocabulary throughout the dispatcher, a pair of `SYSINIT/SYUNINIT` hooks for driver-wide initialisation, a `shutdown_pre_sync` event handler, a new `MYFIRSTIOC_GETCAPS` ioctl that lets user space query feature bits, a modest refactor that splits the Newbus attach out of `myfirst.c` into `myfirst_bus.c` and the `cdev` callbacks out into `myfirst_cdev.c`, a `MAINTENANCE.md` document that explains the version-bumping policy, an updated regression script, and a version bump to 1.8-maintenance.

25.1.3 What You Will Learn

By the end of this chapter you will be able to:

- Explain why unrestricted kernel logging is a production hazard, describe how `ppsratecheck(9)` limits events per second, write a rate-limited logging macro that cooperates with the debug mask from Chapter 23, and recognise the three classes of log messages that deserve different throttling strategies.
- Audit a driver's `read`, `write`, `ioctl`, `open`, `close`, `attach`, `detach`, and `sysctl`-handler paths for correct `errno` usage. Distinguish `EINVAL` from `ENXIO`, `ENOIOCTL` from `ENOTTY`, `EBUSY` from `EAGAIN`, `EPERM` from `EACCES`, and `EIO` from `EFAULT`, and know when each is the right return.

- Add loader tunables through `TUNABLE_INT_FETCH`, `TUNABLE_LONG_FETCH`, `TUNABLE_BOOL_FETCH`, and `TUNABLE_STR_FETCH`, and combine them with writable sysctls so one knob can be set at boot or adjusted at runtime. Understand how `CTLFLAG_TUN` cooperates with the tunable fetchers.
- Expose configuration as a small, well-documented surface rather than as a pile of ad hoc environment variables. Choose between per-driver and per-instance tunables with discipline. Document every tunable's unit, range, and default.
- Version a driver's user-visible interface with a stable three-way split: the human-readable release string in `dev.myfirst.0.version`, the integer `MODULE_VERSION` used by the kernel's module dependency machinery, and the wire-format integer `MYFIRST_IOCTL_VERSION` baked into the public header.
- Add a new ioctl to an existing public header without breaking older callers, retire a deprecated ioctl with a correct deprecation period, and provide a capability bitmask through `MYFIRST_IOC_GETCAPS` so user-space programs can detect feature availability without trial and error.
- Structure a driver's attach and detach paths with the `goto fail;` pattern so that every allocation has exactly one cleanup site, every cleanup runs in reverse order of the allocation, and a partial attach never leaves behind a resource the detach path will not free.
- Split a driver into logical source files along the lines of responsibility rather than along the lines of file size. Choose between a single large file, a small collection of topic-focused files, and a full subsystem tree, and know when each is appropriate.
- Prepare a driver for production use with `MODULE_DEPEND`, `MODULE_PNP_INFO`, a well-behaved modevent handler that accepts `MOD_QUIESCE` when the driver can cleanly pause, a small build system that ships both the module and its documentation, and a `devd(8)`-ready pattern for announcing driver events through `devctl_notify`.
- Use `SYSINIT(9)` and `SYSUNINIT(9)` to hook driver-wide setup and teardown at specific kernel subsystem stages, and understand the difference between module event handlers and subsystem-level init hooks.
- Register and deregister callbacks on well-known kernel events through `EVENTHANDLER(9)`: `shutdown_pre_sync`, `shutdown_post_sync`, `shutdown_final`, `vm_lowmem`, `power_suspend_early`, and `power_resume`. Know how to choose a priority and how to guarantee deregistration on detach.

The list is long because maintenance discipline touches many small surfaces at once. Each item is narrow and teachable. The chapter's work is making them a habit.

25.1.4 What This Chapter Does Not Cover

Several adjacent topics are explicitly deferred so Chapter 25 stays focused on maintenance discipline at the right level for a reader finishing Part 5.

- **Transport-specific production patterns** such as USB hotplug storms, SATA link-state events, and Ethernet media-change handling belong in Part 6 where each transport is taught in full. Chapter 25 teaches the *general* habits; Chapter 26 and later apply them to USB specifically.
- **Full test-framework design**, including regression harnesses that run across several kernel configurations and fault-injection scenarios, belongs in the hardware-free testing sections of Chapters 26, 27, and 28. Chapter 25 adds one more line to the existing regression script; it does not introduce a whole harness.
- **`fail(9)` and `fail_point(9)`**, the kernel's error-injection facilities, are deferred to Chapter 28 alongside the storage driver work where they are most frequently used.
- **Continuous integration, package signing, and distribution** are operational concerns for the project shipping the driver rather than for the driver's source code. The chapter says just enough about packaging

to make the driver reproducible.

- **MAC(9) (Mandatory Access Control) hooks** are a specialised framework and are best introduced in a later security-focused chapter.
- **kbi(9) stability and ABI freezing** are release-engineering decisions made by the FreeBSD project, not by the driver author. The chapter notes the ABI implications of kernel-exported functions but does not cover release engineering in depth.
- **capsicum(4)** capability-mode integration for user-space helpers is a topic for user-space security, not for the driver itself. The chapter's `myfirstctl` remains a traditional UNIX tool.
- **Advanced concurrency patterns** such as epoch(9), read-mostly locks, and lock-free queues. These are mentioned only in passing; the driver's single softc mutex continues to be sufficient at this stage.

Staying inside those lines keeps Chapter 25 a chapter about *maintenance discipline*, not a chapter about every technique a senior kernel developer might use on a senior kernel problem.

25.1.5 Estimated Time Investment

- **Reading only:** three to four hours. Chapter 25's ideas are conceptually lighter than Chapter 24's, and much of the vocabulary is by now familiar. The chapter's job is to turn familiar primitives into discipline.
- **Reading plus typing the worked examples:** eight to ten hours over two or three sessions. The driver evolves through four short stages (rate-limited logging, error audit, tunable and version discipline, SYSINIT and EVENTHANDLER), each smaller than a single Chapter 24 stage. The refactor in Section 6 touches several files but changes little code; most of the work is moving existing code into its new home.
- **Reading plus all labs and challenges:** twelve to fifteen hours over three or four sessions. The labs include a log-flood reproduction and repair, an errno audit with truss, a tunable lab that boots a VM twice with different `/boot/loader.conf` values, a deliberate attach-failure lab that exercises every label in the `goto fail; chain`, a `shutdown_pre_sync` lab that confirms the callback really runs at the right moment, and a regression-script walkthrough that ties everything together.

Section 5 (failure-path management) is the densest in new discipline rather than new vocabulary. The `goto fail; pattern` itself is mechanical; the trick is reading a real FreeBSD attach function and seeing every allocation as a candidate for a new label. If the pattern feels mechanical on the first pass, that is the signal that it has become a habit.

25.1.6 Prerequisites

Before starting this chapter, confirm:

- Your driver source matches Chapter 24 Stage 3 (1.7-integration). Every Chapter 24 primitive is assumed: the `make_dev_s`-based cdev creation, the `myfirst_ioctl.c` dispatcher, the `myfirst_sysctl.c` tree construction, the `MYFIRST_VERSION`, `MODULE_VERSION`, and `MYFIRST_IOCTL_VERSION` triple, and the per-device `sysctl_ctx_free-free` pattern.
- Your lab machine runs FreeBSD 14.3 with `/usr/src` on disk and matching the running kernel.
- A debug kernel with `INVARIANTS`, `WITNESS`, `WITNESS_SKIPSPIN`, `DDB`, `KDB`, `KDB_UNATTENDED`, `KDTRACE_HOOKS`, and `DDB_CTF` is built, installed, and booting cleanly.

- A snapshot of the 1.7-integration state is saved in your VM. Chapter 25's labs include intentional attach-failure scenarios, and a snapshot makes recovery cheap.
- The following user-space commands are in your path: `dmesg`, `sysctl`, `kldstat`, `kldload`, `kldunload`, `devctl`, `devd`, `cc`, `make`, `dtrace`, `truss`, `ktrace`, `kdump`, and `procstat`.
- You are comfortable editing `/boot/loader.conf` and rebooting a VM to pick up new tunables.
- You have the `myfirstctl` companion program from Chapter 24 built and working.

If any item above is shaky, fix it now. Maintenance discipline is easier to learn on a driver that already obeys the earlier chapters' rules than on one that still has unresolved issues from earlier stages.

25.1.7 How to Get the Most Out of This Chapter

Five habits pay off in this chapter more than in any of the previous Part 5 chapters.

First, keep four short manual-page files open in a browser tab or a terminal pane: `ppsratecheck(9)`, `style(9)`, `sysctl(9)`, and `module(9)`. The first is the canonical documentation of the rate-check API. The second is FreeBSD's coding style. The third explains the `sysctl` framework. The fourth is the module event handler contract. None of them is long; each is worth skimming once at the start of the chapter and referring back to when the prose says "consult the manual page for details."

Second, keep three real drivers close to hand. `/usr/src/sys/dev/mmc/mmcsd.c` shows `ppsratecheck` used to throttle a device-`printf` in production. `/usr/src/sys/dev/virtio/block/virtio_blk.c` shows a clean `goto fail;` chain in its attach path and a production-quality set of tunables. `/usr/src/sys/dev/e1000/em_txrx.c` shows how a complex driver splits logging, tunables, and dispatch across several files. Chapter 25 points back to each at the right moment; reading them once now gives the rest of the chapter concrete anchors.

A note on line numbers. When the chapter points at a particular place in `mmcsd.c`, `virtio_blk.c`, or `em_txrx.c`, the pointer is a named symbol, not a numeric line. `ppsratecheck`, the `goto fail;` labels in `virtio_blk_attach`, and the `TUNABLE_*_FETCH` calls stay findable under those names in future tree revisions even as the lines around them move. The audit examples you will see later in the chapter use `file:line` notation purely as a sample tool transcript and carry the same health warning.

Third, type every change into the `myfirst` driver by hand. The additions in Chapter 25 are the kind of changes a developer makes by reflex after a year of maintenance work. Typing them out now builds the reflex; pasting them skips the lesson.

Fourth, after the tunable material in Section 3, reboot your VM at least once with a new `/boot/loader.conf` setting and watch the driver pick it up during attach. Tunables are one of those features that feels abstract until you see a real value flow from the bootloader through the kernel into your softc. Two reboots and one `sysctl` command is all it takes.

Fifth, when the section on `goto fail;` asks you to introduce a deliberate failure into `myfirst_attach`, actually do it. Injecting a single `return (ENOMEM);` into the middle of attach and watching the cleanup chain unwind correctly is the single best way to internalise the pattern. The chapter suggests a specific place to inject, and the regression script confirms the cleanup really ran.

25.1.8 Roadmap Through the Chapter

The sections in order are:

1. **Rate Limiting and Logging Etiquette.** Why uncontrolled kernel logging is a production hazard; the three classes of driver log messages (lifecycle, error, debug); `ppsratecheck(9)` and `ratecheck(9)` as the FreeBSD answer to log-flood; a rate-limited `DLOG_RL` macro that cooperates with the Chapter 23 debug mask; `log(9)` priority levels and their relationship to `device_printf` and `printf`; what the kernel message buffer actually costs and how not to spend it carelessly.
2. **Error Reporting and Return Conventions.** Why `errno` discipline is a contract with every caller; the small vocabulary of kernel `errno`s a driver routinely uses; when each is appropriate; `ENOIOCTL` versus `ENOTTY` and why the driver must never return `EINVAL` from the `ioctl` default; `sysctl` handler return codes; module event handler return codes; a checklist the reader can apply to every driver they write from now on.
3. **Driver Configuration via Loader Tunables and `sysctl`.** The difference between `/boot/loader.conf` tunables and runtime `sysctls`; the `TUNABLE_*_FETCH` family and the `CTLFLAG_TUN` flag; per-driver versus per-instance tunables; how to document a tunable so operators can trust it; a worked lab that boots a VM with a tunable in three different positions and observes the effect in the driver's `sysctl` tree.
4. **Versioning and Compatibility Strategies.** The three-way version split (`MODULE_VERSION` integer, `MYFIRST_VERSION` human string, `MYFIRST_IOCTL_VERSION` wire-format integer); how each one is used; how to add a new `ioctl` without breaking older callers; how to retire a deprecated `ioctl`; `MYFIRST_IOCTL_GETCAPS` and the capability-bitmask idea; how a driver can deprecate a `sysctl` OID gracefully in the absence of a dedicated kernel flag; how `MODULE_DEPEND` enforces a minimum version of a dependency module.
5. **Managing Resources in Failure Paths.** The cleanup-on-failure problem in `myfirst_attach`; the `goto fail`; pattern and why linear unwinding beats nested `if` chains; label naming conventions (`fail_mtx`, `fail_cdev`, `fail_sysctl`); common mistakes (falling through after success, missing a label, adding a resource without adding its cleanup); a small helper-function discipline that reduces duplication; a deliberate-failure lab that tests the whole chain.
6. **Modularisation and Separation of Concerns.** Splitting a driver into files along the axes of responsibility; the canonical split for a character driver (`myfirst.c`, `myfirst_bus.c`, `myfirst_cdev.c`, `myfirst_ioctl.c`, `myfirst_sysctl.c`, `myfirst_debug.c`, `myfirst_log.c`); public versus private headers; how to organise the `Makefile` so all of these files build into one `.ko`; when modularisation helps and when it gets in the way; how a team of developers uses the split to reduce merge conflict.
7. **Preparing for Production Use.** `MODULE_DEPEND` and dependency enforcement; `MODULE_PNP_INFO` for auto-loading; `MOD_QUIESCE` and the pause-before-unload contract; a build-system pattern that installs the module and its documentation; a `devd(8)` rule that reacts to the driver's events; a small `MAINTENANCE.md` document that states the driver's maintenance contract in writing.
8. **SYSINIT, SYSUNINIT, and EVENTHANDLER.** The kernel's broader lifecycle machinery beyond `MOD_LOAD` and `MOD_UNLOAD`; `SYSINIT(9)` and `SYSUNINIT(9)` with subsystem IDs and order constants; real FreeBSD examples of each; `EVENTHANDLER(9)` for cross-cutting notifications (`shutdown_pre_sync`, `vm_lowmem`, `power_resume`); how to register and deregister cleanly; how a driver uses all three mechanisms without drifting into overengineering.

After the eight sections come a set of hands-on labs that exercise each discipline, a set of challenge exercises that stretch the reader without introducing new foundations, a troubleshooting reference for the symptoms most readers will hit, a Wrapping Up that closes Chapter 25's story and opens Chapter 26's, a bridge to the next chapter, a quick-reference card, and a glossary.

If this is your first pass, read linearly and do the labs in order. If you are revisiting, Sections 1 and 5 stand alone

and make good single-sitting reads. Section 8 is a short conceptual payoff at the end of the chapter; it depends lightly on Section 7's production-use material and is easy to save for a second sitting.

A small note before the technical work begins. Chapter 25 is the last chapter of Part 5. Its additions are smaller than Chapter 24's, but they touch almost every file in the driver. Expect to spend more time re-reading your own earlier code than writing new code. That, too, is maintenance discipline. A driver you re-read patiently is a driver you can change confidently; a driver you change confidently is a driver you can keep alive.

25.2 Section 1: Rate Limiting and Logging Etiquette

The first discipline this chapter teaches is the discipline of not talking too much. The `myfirst` driver at the end of Chapter 24 logs when it attaches, when it detaches, when a client opens or closes the device, when a read or write crosses the boundary, when an `ioctl` is dispatched, and when the debug mask is adjusted. Every one of those log lines was introduced for a good reason, and every one of them is useful when a single event happens. What none of the Chapter 24 log lines accounts for is what happens when the same event fires a hundred thousand times a second.

This section explains why that question matters more than it looks, introduces the three categories of driver log message that behave differently under pressure, teaches the FreeBSD rate-check primitives (`ratecheck(9)` and `ppsratecheck(9)`), and shows how to build a small, disciplined macro on top of them that cooperates with the existing debug-mask machinery from Chapter 23. By the end of the section the `myfirst` driver has a `myfirst_log.c` and `myfirst_log.h` pair, and its message buffer no longer turns into noise under stress.

25.2.1 The Problem With Unrestricted Logging

A kernel message is cheap to write and expensive to carry. `device_printf(dev, "something happened\n")` is a single function call, tens of nanoseconds on a modern CPU, and it returns almost immediately. The cost is not in the call; the cost is in everything that happens to the bytes afterwards. The formatted string is copied into the kernel message buffer, a circular area of kernel memory whose size is fixed at boot. It is delivered to the console device if the console is attached (often a serial port in a VM, with a finite bit rate). It is sent to the syslog daemon running in user space through the `log(9)` path if the driver uses that path, and then through `newsyslog(8)` into `/var/log/messages` on disk. Each of those steps has a cost, and every one of them is synchronous at the moment the driver writes the line.

When the driver writes one line, none of this matters. When the driver writes a million lines in a second, all of it matters. The kernel message buffer fills, and the oldest messages are overwritten before anyone reads them. The console, typically running at 115200 baud, falls behind and cannot catch up, which in turn backs pressure into the kernel path that wrote the line, which is your driver's fast path. The syslog daemon wakes up, does work, and goes back to sleep many times per second, stealing cycles from other processes. The disk where `/var/log/messages` lives fills at a predictable rate, and a driver that logs at ten thousand lines per second can fill a reasonably sized partition in an afternoon.

None of these symptoms is caused by a bug in the driver's logic. They are caused by the driver's *logging volume*, which in turn is caused by the driver firing a reasonable log line on every event. Reasonable log lines are fine as long as events are rare. They become a hazard when events are common. The whole craft of logging etiquette is knowing, at the moment you write a log line, whether the event behind it is rare or common, and writing the code so that uncontrolled repetition cannot turn a rare-event log into a common-event log.

A concrete example from a real driver illustrates the point. Consider a PCIe SSD controller that notifies its driver of a recoverable queue full condition. On a healthy system that condition is rare enough that logging each occurrence is useful. On a sick system it may happen hundreds of times per second until someone replaces the hardware. If the driver writes a line every time, the message buffer fills with nearly identical lines, all the earlier messages from that boot are overwritten and lost, and the operator who tries to diagnose the problem by reading `dmesg` sees only the last page of the flood. The hardware's actual behaviour is obscured by the driver's reaction to it. A rate-limited log line would have shown the first few occurrences, the rate, and then a periodic reminder; the earlier context in `dmesg` would have survived; the operator would have had something to work with.

The lesson generalises. The right log discipline is not “log less” and not “log more” but “log at a rate that remains useful regardless of how often the underlying event fires.” The rest of this section teaches that discipline concretely.

25.2.2 Three Categories of Driver Log Message

Before picking the right throttling policy, it helps to name the three categories of log message a driver typically emits. Each category has a different throttling story.

The first category is **lifecycle events**. These are the messages that mark attach, detach, suspend, resume, module load, and module unload. They occur once per lifecycle transition, typically a few times in the lifetime of a module. No throttling is needed; the volume is naturally low. Rate-limiting lifecycle messages would be a mistake because it would hide important state transitions.

The second category is **error and warning messages**. These are messages that report something the driver thinks is wrong. By construction each of these should be rare; if a warning is firing a hundred times a second, the warning is telling you something about the rate of underlying events, and that information is worth preserving even when the event repeats. Error and warning messages benefit strongly from rate limiting, but the rate limit should preserve at least one message per burst and should make the *rate* itself visible.

The third category is **debug and trace messages**. These are the messages under the `DPRINTF` macros from Chapter 23. They are intentionally verbose when the debug mask is on and silent when the mask is off. Throttling them at the emit site adds noise to what is already a low-signal path; the better discipline is to avoid emitting them when the mask is off, which is already what the existing `DPRINTF` does. Debug and trace messages do not need additional rate limiting, but they need the user to be able to turn them off entirely with a single `sysctl` command. The existing Chapter 23 plumbing already provides that.

With the three categories named, the rest of the section focuses on the second category. Lifecycle messages are fine as they are. Debug messages are handled by the existing mask. Error and warning messages are where the real discipline goes.

25.2.3 Introducing `ratecheck` and `ppsratecheck`

FreeBSD's kernel provides two closely related primitives for rate-limited output. Both live in `/usr/src/sys/kern/kern_time.c` and are declared in `/usr/src/sys/sys/time.h`.

`ratecheck(struct timeval *lasttime, const struct timeval *mininterval)` is the simpler of the two. The caller holds a `struct timeval` remembering when the event last fired, along with a minimum interval

between allowed prints. On each call, `ratecheck` compares the current time to `*lasttime`, and if `mininterval` has passed, it updates `*lasttime` and returns 1. Otherwise it returns 0. The calling code prints only when the return is 1. The result is a simple floor on the rate of prints: at most one print per `mininterval`.

`ppsratecheck(struct timeval *lasttime, int *curpps, int maxpps)` is the more commonly used form in drivers. Its name is a legacy of the pulse-per-second telemetry use case it was originally written for. The kernel source exposes it through a `#define` in `/usr/src/sys/sys/time.h`:

```
int    eventratecheck(struct timeval *, int *, int);
#define ppsratecheck(t, c, m) eventratecheck(t, c, m)
```

The call accepts a pointer to a timestamp, a pointer to a counter of events in the current one-second window, and the maximum allowed events per second. On each call, if the second has not yet rolled over, the counter is incremented. If the counter exceeds `maxpps`, the function returns 0 and the caller suppresses its output. When a new second begins, the counter is reset to 1 and the function returns 1, allowing one print for the new second. A special value of `maxpps == -1` disables rate limiting entirely (useful for debug paths).

Both primitives are cheap: a comparison and an arithmetic update, no locks. Both are safe to call from any context where the driver currently calls `device_printf`, including interrupt handlers, as long as the storage they access is stable in that context. In practice drivers keep the `struct timeval` and the counter inside their `softc`, protected by the same lock that protects the log site, or they use per-CPU state where that is convenient.

A short example from the FreeBSD tree shows the pattern in real use. The MMC SD card driver, `/usr/src/sys/dev/mmc/mmc.c`, rate-limits a complaint about write errors so that a bad card does not flood the log:

```
if (ppsratecheck(&sc->log_time, &sc->log_count, LOG_PPS))
    device_printf(dev, "Error indicated: %d %s\n",
        err, mmc_errmsg(err));
```

The driver stores `log_time` and `log_count` in its `softc`, picks a reasonable `LOG_PPS` (typically 5 to 10), and wraps the `device_printf` call in the rate check. The first few errors in any second produce log lines; the next several hundred in the same second produce nothing.

That is the whole idea. Everything that follows in this section is about doing the same thing with more structure, more discipline, and less repetition.

25.2.4 A Simple Rate-Limited Log Macro

The goal is a macro the driver can use instead of bare `device_printf` in any error or warning path where the event might repeat. The macro should:

1. Silently discard output when the rate limit is exceeded.
2. Allow a different rate per call site, or at least per category.
3. Cooperate with the existing debug-mask machinery from Chapter 23 so that debug output remains controlled by the mask rather than by the rate limiter.
4. Compile out of non-debug builds if the driver chooses, with no runtime cost.

A minimal implementation looks like this. In a new `myfirst_log.h`:

```
#ifndef _MYFIRST_LOG_H_
#define _MYFIRST_LOG_H_

#include <sys/time.h>

struct myfirst_ratelimit {
    struct timeval rl_lasttime;
    int           rl_curpps;
};

/*
 * Default rate for warning messages: at most 10 per second per call
 * site. Chosen to keep the log readable under a burst while still
 * showing the rate itself.
 */
#define MYF_RL_DEFAULT_PPS 10

/*
 * DLOG_RL - rate-limited device_printf.
 *
 * rlp must point at a per-call-site struct myfirst_ratelimit stored in
 * the driver (typically in the softc). pps is the maximum allowed
 * prints per second. The remaining arguments match device_printf.
 */
#define DLOG_RL(sc, rlp, pps, fmt, ...) do { \
    if (ppsratecheck(&(rlp)->rl_lasttime, &(rlp)->rl_curpps, pps)) \
        device_printf((sc)->sc_dev, fmt, ##__VA_ARGS__); \
} while (0)

#endif /* _MYFIRST_LOG_H_ */
```

In the `softc`, reserve one or more rate-limit structures:

```
struct myfirst_softc {
    /* ... existing fields ... */
    struct myfirst_ratelimit sc_rl_ioerr;
    struct myfirst_ratelimit sc_rl_short;
};
```

At each error site, replace the bare `device_printf` with `DLOG_RL`:

```

/* old:
 * device_printf(sc->sc_dev, "I/O error on read, ENXIO\n");
 */
DLOG_RL(sc, &sc->sc_rl_ioerr, MYF_RL_DEFAULT_PPS,
        "I/O error on read, ENXIO\n");

```

The macro uses a comma operator inside a `do { ... } while (0)` block so it fits anywhere a statement fits, including inside `if` and `else` bodies without braces. The `ppsratecheck` call is inexpensive; when the rate limit is exceeded, the `device_printf` is simply not called. When the rate limit is not exceeded, the behaviour is identical to a direct `device_printf`.

A small but important point: each call site should have its own struct `myfirst_ratelimit`. Sharing one structure across multiple unrelated call sites means the first path that fires in each second suppresses every other path for the rest of that second. In a driver with a handful of rare-but-possible errors, reserve one rate-limit structure per category, name it after the category, and use it consistently.

25.2.5 Cooperating With the Chapter 23 Debug Mask

The rate-limited macro solves the error-and-warning case. The debug case already has its own mechanism from Chapter 23:

```
DPRINTF(sc, MYF_DBG_IO, "read: %zu bytes requested\n", uio->uio_resid);
```

The `DPRINTF` macro expands to nothing when the corresponding bit in `sc_debug` is clear, so debug output at the quiet mask (`mask = 0`) has no runtime cost. There is no need to rate-limit debug output: the operator turns it on when they want to see it and turns it off when they do not. If the operator turns on `MYF_DBG_IO` on a busy device and sees a flood of output, that is the intended behaviour; they wanted the flood. The rate-limit macro and the debug macro serve different purposes and should not be combined.

Where the two do meet is in the occasional log line that is conceptually a warning but that the developer wants the ability to silence entirely. For those, the right pattern is to gate the `DLOG_RL` call on a debug bit:

```

if ((sc->sc_debug & MYF_DBG_IO) != 0)
    DLOG_RL(sc, &sc->sc_rl_short, MYF_RL_DEFAULT_PPS,
            "short read: %d bytes\n", n);

```

The rate-limit fires under debug mask, and the output is both opt-in and bounded. This is a minority pattern; most warnings should fire unconditionally at a rate limit, and most debug prints should be mask-gated without a rate limit.

25.2.6 `log(9)` Priority Levels

A third logging primitive deserves mention here: `log(9)`. Unlike `device_printf`, which always routes through the kernel message buffer, `log` routes through the syslog path with a syslog priority. The function lives in `/usr/src/sys/kern/subr_prf.c` and takes a priority from `/usr/src/sys/sys/syslog.h`:

```
void log(int level, const char *fmt, ...);
```

Common priorities: `LOG_EMERG` (0) for system-unusable conditions, `LOG_ALERT` (1) for immediate action, `LOG_CRIT` (2) for critical conditions, `LOG_ERR` (3) for error conditions, `LOG_WARNING` (4) for warnings, `LOG_NOTICE` (5) for notable but normal conditions, `LOG_INFO` (6) for informational messages, `LOG_DEBUG` (7) for debug-level messages. A driver that uses `log(LOG_WARNING, ...)` for its warning path rather than `device_printf` gains the ability to be filtered by `syslog.conf(5)` into a separate log file without the driver author having to do anything else.

The tradeoff is that `log(9)` does not prepend the device name. A driver using `log` has to format the device name into the message manually, which is verbose. Most FreeBSD drivers therefore prefer `device_printf` for driver-specific messages and reserve `log` for cross-cutting notifications. The `myfirst` driver follows the same convention: `device_printf` for everything the operator is meant to read with `dmesg`, `log` for nothing at this stage.

A pragmatic guideline: use `device_printf` when the message is *about this device*. Use `log(9)` when the message is *about a cross-cutting condition* that the syslog infrastructure is the right place to see, such as an authentication event or a policy violation. Driver code rarely needs the second kind.

25.2.7 The Kernel Message Buffer and What It Costs

One more technical detail before the section closes. The kernel message buffer (`msgbuf`) is a fixed-size circular buffer inside the kernel, allocated at boot. Its size is controlled by the `kern.msgbufsize` tunable, which defaults to 96 KiB on amd64 and can be raised in `/boot/loader.conf`. Every `printf`, `device_printf`, and `log` call routes through the buffer. When the buffer fills, the oldest messages are overwritten. The buffer's contents are what `dmesg` prints.

Two practical consequences follow. First, a flood of short messages can evict the earlier messages an operator needs. A line that says “hello” uses a few dozen bytes; a buffer of 96 KiB holds maybe three thousand such lines; a loop that prints at ten thousand lines per second evicts the entire boot log in less than half a second. Second, a formatted message is not free to produce. `printf`-style formatting costs CPU, and inside an interrupt handler or a hot path that cost shows up directly in latency numbers. The rate-limited macro helps with the first consequence. The second is why debug messages are mask-gated: a `DPRINTF` at mask zero compiles down to an empty statement at runtime, skipping both the formatting and the storage.

Increasing `kern.msgbufsize` is a reasonable response to a machine that repeatedly loses boot logs, but it is not a substitute for rate limiting. A bigger buffer simply buys more room before the flood evicts the older messages; rate limiting reduces the flood itself. Both are worth doing. `kern.msgbufsize=262144` in `/boot/loader.conf` is a common operator choice on production machines. It is not a Chapter 25 action, because the driver cannot change the buffer size at run time.

25.2.8 A Worked Example: The `myfirst` Read Path

Putting the pieces together, consider the existing `myfirst_read` callback. A simplified version from Chapter 24 looked like this:

```
static int
myfirst_read(struct cdev *cdev, struct uio *uio, int ioflag)
{
    struct myfirst_softc *sc = cdev->si_drv1;
    int error = 0;

    mtx_lock(&sc->sc_mtx);
    if (uio->uio_resid == 0) {
        device_printf(sc->sc_dev, "read: empty request\n");
        goto out;
    }
    /* copy bytes into user space, update counters ... */
out:
    mtx_unlock(&sc->sc_mtx);
    return (error);
}
```

That code has a latent rate-flood problem. Under stress, a buggy or malicious user-space program can call `read(fd, buf, 0)` in a tight loop and fill the message buffer with “empty request” lines. The event is not an error in the driver; it is a strange-but-legal syscall pattern. Logging it at all is marginal, but if the driver logs it, the log line must be rate-limited.

After the refactor, the same path looks like this:

```
static int
myfirst_read(struct cdev *cdev, struct uio *uio, int ioflag)
{
    struct myfirst_softc *sc = cdev->si_drv1;
    int error = 0;

    mtx_lock(&sc->sc_mtx);
    if (uio->uio_resid == 0) {
        DLOG_RL(sc, &sc->sc_rl_short, MYF_RL_DEFAULT_PPS,
            "read: empty request\n");
        goto out;
    }
    /* copy bytes into user space, update counters ... */
out:
    mtx_unlock(&sc->sc_mtx);
    return (error);
}
```

The change is three lines. The effect is that the log can no longer flood, and the first occurrence in any second still produces a line for the operator to notice. The softc gains one `struct myfirst_ratelimit` `sc_rl_short` field; no other code moves.

Apply the same transformation to every `device_printf` in every error or warning path, reserve one struct `myfirst_ratelimit` per category, and the driver is rate-limited. The diff is mechanical; the discipline is what makes the diff possible in the first place.

25.2.9 Common Mistakes and How to Avoid Them

Three mistakes are common when first applying rate limiting. Each is easy to spot once you know to look.

The first mistake is **sharing a single rate-limit structure across unrelated call sites**. If site A and site B both use `sc->sc_rl_generic`, a burst at site A silences site B for the rest of the second, and the operator sees only one category. The correct discipline is one rate-limit structure per logical category. Two or three categories per driver is usual; ten is a sign the driver is logging too many kinds of events.

The second mistake is **rate-limiting lifecycle messages**. The driver loads and prints a banner. That banner fires once. Wrapping it in `ppsratecheck` adds noise for no gain, and on a second load during an unfortunate second-boundary it can skip the banner entirely. Reserve rate limiting for the messages that can actually repeat.

The third mistake is **forgetting that the rate-limit counter lives in the softc**. A call site that fires before attach completes, or after detach starts, can touch a softc whose rate-limit structure is not yet initialised (or has been zeroed by `bzero(sc, sizeof(*sc))`). The struct `timeval` and `int` are both value types; a zero-initialised structure is fine for the first call, because `ppsratecheck` handles the `lasttime == 0` case correctly. But an uninitialised heap allocation that later holds garbage is not fine, because the `lasttime` field may hold a large value that makes the code think the last event was in the distant future, and every subsequent call returns 0 until the kernel's clock passes that future time, which may be never. The fix is to ensure the softc is zero-initialised, which in `myfirst` it already is (`newbus` allocates softc with `MALLOC(... M_ZERO)`). A driver that allocates its own state with `M_NOWAIT` without `M_ZERO` must call `bzero` explicitly.

25.2.10 When Not to Rate-Limit

Rate limiting is a discipline for paths that can fire often. Some paths cannot. A `KASSERT` failure panics the kernel, so rate-limiting the pre-panic message is wasted effort. An error that aborts a module load ends the load, so only one copy of the message can ever appear. A `device_printf` at attach time fires at most once per instance. For all of these, bare `device_printf` is correct and the extra wrapper is clutter.

A useful rule of thumb: if the call site runs after attach has completed and before detach has run, and if the event can be caused by something external to the driver (a misbehaving user-space program, a flaky device, a stressed kernel), then rate-limit. Otherwise, do not.

25.2.11 What the myfirst Driver Now Contains

After this section the `myfirst` driver's working tree has two new files:

```
myfirst_log.h   - the DLOG_RL macro and struct myfirst_ratelimit definition
myfirst_log.c   - any non-trivial rate-check helpers (empty for now)
```

The `myfirst.h` header still holds the `softc`. The `softc` gains two or three `struct myfirst_ratelimit` fields, named for the call-site categories that use them. The `read`, `write`, and `ioctl` paths replace their bare `device_printf` calls at error sites with `DLOG_RL`. The `attach`, `detach`, `open`, and `close` paths keep their bare `device_printf` calls, because those are lifecycle messages and do not repeat.

The `Makefile` gains one line:

```
SRCS= myfirst.c myfirst_debug.c myfirst_ioctl.c myfirst_sysctl.c \  
      myfirst_log.c
```

The module builds, loads, and behaves exactly as before in the common case. Under stress the driver no longer floods the message buffer. That is the whole contribution of Section 1.

25.2.12 Wrapping Up Section 1

A driver that logs without discipline is a driver that fails gracefully in the lab and fails loudly in production. The FreeBSD rate-check primitives `ratecheck(9)` and `ppsratecheck(9)` are small enough to understand in an hour and effective enough to pay back their cost for the rest of the driver's life. Combined with the existing debug-mask machinery from Chapter 23, they give the `myfirst` driver a clean three-way logging story: lifecycle messages go through plain `device_printf`, error and warning messages go through `DLOG_RL`, and debug messages go through `DPRINTF` under the mask.

In the next section, we turn from what the driver says to what it returns. A log line is for the operator; an `errno` is for the caller. A driver that says the right thing to the operator but the wrong thing to the caller is still a broken driver.

25.3 Section 2: Error Reporting and Return Conventions

The second discipline the chapter teaches is the discipline of returning the right `errno`. An `errno` is a small number. The set of possible `errno`s is defined in `/usr/src/sys/sys/errno.h`, and at time of writing FreeBSD defines fewer than a hundred of them. A driver that is careless about which `errno` it returns looks fine in the moment, because the caller mostly only checks whether the return value was non-zero, and any non-zero value passes that test. It looks much less fine a few months later when the first user-space helper tries to distinguish *why* a call failed, and the driver's `errno` choices turn out to be inconsistent. This section teaches the small vocabulary of `errno`s a driver routinely uses, shows how to choose between them, and walks through an audit of the `myfirst` driver's existing paths.

25.3.1 Why Errno Discipline Matters

A driver's `errno` return is a contract with every caller. User-space programs use `errno`s through `strerror(3)` and through direct comparison (`if (errno == EBUSY)`). Kernel code that invokes driver callbacks uses the return value to decide what to do next: a `d_open` that returns `EBUSY` causes the kernel to fail the `open(2)` syscall with `EBUSY`; a `d_ioctl` that returns `ENOIOCTL` causes the kernel to fall through to the generic `ioctl` layer; a `device_attach` that returns a non-zero value causes Newbus to roll back the attach and disown the device.

Each of those consumers expects a specific value to mean a specific thing. A driver that returns `EINVAL` where `ENXIO` was expected does not necessarily fail; often it just misleads, and the misleading `errno` shows up as a mystifying diagnostic somewhere the driver author will never see.

The discipline is cheap. The costs of ignoring it compound over time. A driver that chose its `errno`s well from the start is a driver that produces accurate manual pages, accurate user-space helper error messages, and accurate bug reports. A driver that was sloppy about `errno`s starts producing `strerror` output that is slightly wrong in many places, and the user-space side of the ecosystem inherits the sloppiness.

25.3.2 The Small Vocabulary

The full list of `errno`s is long. The subset a typical character driver uses is short. The table below is the vocabulary most often needed in a FreeBSD driver, grouped by when each is appropriate.

Errno	Numeric value	When to return
<code>0</code>	0	Success. The only non-error return.
<code>EPERM</code>	1	The caller lacks privilege for the requested operation, even though the call itself is well-formed. Example: a non-root user requesting a privileged <code>ioctl</code> .
<code>ENOENT</code>	2	The requested object does not exist. Example: a lookup by name or ID that finds nothing.
<code>EIO</code>	5	Generic I/O error from hardware. Use when the hardware returned a failure and there is no more specific <code>errno</code> .
<code>ENXIO</code>	6	The device is gone, detached, or otherwise unreachable. Example: an <code>ioctl</code> on a file descriptor whose underlying device was removed. Different from <code>ENOENT</code> : the object was there and is now gone.
<code>EBADF</code>	9	The file descriptor is not opened correctly for the operation. Example: a <code>MYFIRSTIOC_SETMSG</code> call made on a file descriptor opened read-only.
<code>ENOMEM</code>	12	Allocation failed. Use for <code>malloc(M_NOWAIT)</code> failures and similar.
<code>EACCES</code>	13	The caller lacks permission at the file-system level. Different from <code>EPERM</code> : <code>EACCES</code> is about file permissions, <code>EPERM</code> is about privilege.

Errno	Numeric value	When to return
EFAULT	14	A user pointer is invalid. Returned by failed copyin or copyout. Drivers should forward copyin/copyout failures unchanged.
EBUSY	16	The resource is in use. Use for detach that cannot proceed because a client still holds the device open, or for mutex-like acquire attempts that cannot wait.
EINVAL	22	The arguments are recognised but invalid. Use when the driver understood the request but the inputs are malformed.
EAGAIN	35	Try again later. Returned from non-blocking I/O when the operation would block, or from allocation failures that may succeed on retry.
EOPNOTSUPP	45	The operation is not supported by this driver. Use when the call is well-formed but the driver has no code to handle it.
ETIMEDOUT	60	A wait timed out. Use for hardware commands that did not complete within the driver's timeout budget.
ENOIOCTL	-3	The ioctl command is unknown to this driver. Use this for the default case in <code>d_ioctl</code>; the kernel translates it to ENOTTY for user space.
ENOSPC	28	No space left, whether on the device, in a buffer, or in an internal table.

Three pairs in this table are famously easy to confuse: EPERM versus EACCES, ENOENT versus ENXIO, and EINVAL versus EOPNOTSUPP. Each is worth looking at in turn.

EPERM versus EACCES. EPERM is about privilege: the caller is not privileged enough to perform the operation. EACCES is about permission: the file-system ACL or mode bits forbid the access. A non-root user trying to write to `/dev/myfirst0` when the node mode is `0600 root:wheel` gets EACCES from the kernel before the driver is consulted. A root user trying to call a privileged ioctl that the driver rejects because the caller is not in a particular jail gets EPERM from the driver. The distinction matters because the administrator's remedy differs: EACCES asks the administrator to adjust device permissions, while EPERM asks the administrator to adjust the caller's privileges.

ENOENT versus ENXIO. ENOENT is *there is no such object*. ENXIO is *the object is gone, or the device is unreachable*. On a lookup into a driver's internal table, ENOENT is the right answer when the requested key is not present. On an operation against a device that has been detached or that has signalled a surprise-removal condition, ENXIO is the right answer. The distinction matters because operational tools treat them differently: ENOENT suggests

the caller gave the wrong key; `ENXIO` suggests the device needs reattachment.

`EINVAL` versus `EOPNOTSUPP`. `EINVAL` is *I understood what you asked but the arguments are wrong*. `EOPNOTSUPP` is *I do not support what you asked*. A `MYFIRSTIOC_SETMSG` call with a buffer that is too long is `EINVAL`. A `MYFIRSTIOC_SETMODE` call for a mode the driver never implements is `EOPNOTSUPP`. The distinction matters because `EOPNOTSUPP` tells the caller to use a different approach, while `EINVAL` tells the caller to fix the arguments and retry.

A fourth confusion deserves a paragraph of its own: `ENOIOCTL` versus `ENOTTY`. `ENOIOCTL` is a negative value (-3) defined for the `ioctl` code path inside the kernel. A driver's `d_ioctl` default case returns `ENOIOCTL` to tell the kernel "I do not recognise this command; please fall through to the generic layer." The generic layer handles `FIONBIO`, `FIOASYNC`, `FIOGETOWN`, `FIOSETOWN`, and similar cross-device `ioctl`s. If the generic layer also does not recognise the command, it translates `ENOIOCTL` into `ENOTTY` (positive 25) for delivery to user space. The common mistake is to return `EINVAL` from the default case of a `d_ioctl` switch, which suppresses the generic fallback entirely. The Chapter 24 driver already returns `ENOIOCTL` correctly; Chapter 25's audit confirms it and checks every other `errno` in the driver for similar issues.

25.3.3 The `ioctl` Dispatcher Audit

The first audit pass targets `myfirst_ioctl.c`. Each case in the switch statement produces at most one non-zero return. The audit looks at each and asks whether the returned `errno` is correct.

Case `MYFIRSTIOC_GETVER`: returns 0 on success, never fails. Nothing to audit.

Case `MYFIRSTIOC_GETMSG`: returns 0 on success. The current code does not reject on `fflag` because the message is public. That is a design choice, not a bug. If the driver wanted to restrict `GETMSG` to readers (i.e., require `FREAD`), it would return `EBADF` on the `fflag` check, consistent with the `SETMSG` and `RESET` cases.

Case `MYFIRSTIOC_SETMSG`: returns `EBADF` when the file descriptor lacks `FWRITE`, which is correct. The second audit question is what happens when the input is not NUL-terminated: `strncpy` in the kernel tolerates it (copies up to `MYFIRST_MSG_MAX - 1` and terminates), so the driver does not need to check. The third question is whether the length should be validated before copy. The kernel's automatic copyin already enforced the fixed length baked into the `ioctl` encoding, so there is no user-space buffer to validate; the value is in `data` and has been copied already.

Case `MYFIRSTIOC_RESET`: returns `EBADF` when the file descriptor lacks `FWRITE`. The Chapter 25 audit raises a second question: should the reset be privileged? A driver that lets any writer call `RESET` and zero out statistics is exposing a minor denial-of-service surface. The simple fix is to check `priv_check(td, PRIV_DRIVER)` before executing the reset:

```
case MYFIRSTIOC_RESET:
    if ((fflag & FWRITE) == 0) {
        error = EBADF;
        break;
    }
    error = priv_check(td, PRIV_DRIVER);
    if (error != 0)
        break;
```

```
/* ... existing reset body ... */  
break;
```

If `priv_check` fails, the `errno` is `EPERM` (the kernel returns `EPERM` rather than `EACCES` because the check is about privilege, not file-system permissions). The `myfirstctl` program running as root sees 0; a non-root program running as the `_myfirst` user sees `EPERM`.

Default case: returns `ENOIOCTL`, which is correct. Leave it alone.

25.3.4 The Read and Write Path Audit

The second audit pass targets the read and write callbacks.

For `myfirst_read`, the current code returns 0 on success, `EFAULT` when `uiomove` fails, and 0 when `uio_resid == 0`. The 0 return on an empty request is the standard UNIX behaviour (a read of zero bytes is allowed and returns 0 bytes) and is correct. No `errno` change is needed.

For `myfirst_write`, similarly, 0 on success, `EFAULT` on `uiomove` failure, 0 on zero-byte write. Correct.

Neither callback needs `EIO`: the driver does not do hardware I/O at this point, so there is no hardware failure to propagate. A future version of the driver that drives real hardware would return `EIO` from the read or write callback when the hardware indicated a transport-level failure. Adding that return now would be premature; it is the kind of thing Chapter 28's storage work will treat concretely.

25.3.5 The Open and Close Path Audit

The open callback currently returns 0 unconditionally. The audit question is whether it should ever fail. Three failure modes are conventionally possible: the device is exclusive-open and already has a user (`EBUSY`), the device is powered down and not currently acceptable to open (`ENXIO`), or the driver is being detached right now (`ENXIO`). The simple `myfirst` driver does not enforce exclusive open, and it always accepts opens except during detach. During detach the kernel destroys the `cdev` before the detach returns, so any open arriving after `destroy_dev` begins is rejected by the kernel itself before the driver's `d_open` is called. The `myfirst` driver therefore does not need explicit `ENXIO` logic. Leaving the open callback returning 0 is correct.

The close callback returns 0 unconditionally. That is correct. The only conceivable reason for `d_close` to return non-zero is a hardware operation during close that failed; since the `myfirst` driver does no such operation, 0 is the right return.

25.3.6 The Attach and Detach Path Audit

Attach and detach are the callbacks Newbus invokes. Their return values tell Newbus whether to roll back or proceed.

`myfirst_attach`'s non-zero return means "attach failed; please roll back." Every error path in attach must return a positive `errno`. The current code returns the error value from `make_dev_s`, which is positive on failure; that is correct. The additions in Section 5 of this chapter will introduce more error paths with labelled `gotos`;

each of them will use the correct `errno` for the failed step (`ENOMEM` for allocation failure, `ENXIO` for resource allocation failure, etc.).

`myfirst_detach`'s non-zero return means "cannot detach right now; please leave the device attached." The current code returns `EBUSY` when `sc_open_count > 0`, which is correct. Newbus translates `EBUSY` from detach into a `devctl detach` failure with the same `errno`, which is the right user-visible behaviour.

The module event handler (`myfirst_modevent`) returns non-zero to reject the event. `MOD_UNLOAD` that cannot proceed because some device instance is still in use returns `EBUSY`. `MOD_LOAD` that cannot proceed because of a sanity-check failure returns an appropriate `errno` (`ENOMEM`, `EINVAL`, etc.). The current code is correct.

25.3.7 The Sysctl Handler Audit

Sysctl handlers have their own `errno` conventions. The Chapter 24 driver has one custom handler, `myfirst_sysctl_message_len`. Its body is:

```
static int
myfirst_sysctl_message_len(SYSCTL_HANDLER_ARGS)
{
    struct myfirst_softc *sc = arg1;
    u_int len;

    mtx_lock(&sc->sc_mtx);
    len = (u_int)sc->sc_msglen;
    mtx_unlock(&sc->sc_mtx);

    return (sysctl_handle_int(oidp, &len, 0, req));
}
```

The handler reads its input with `sysctl_handle_int`, which returns 0 on success and a positive `errno` on failure. The handler forwards that `errno` unchanged, which is correct. No audit change is needed.

A sysctl handler that writes (rather than only reads) should check `req->newptr` to distinguish a read from a write, and should return `EPERM` if the write is attempted on a read-only OID. The existing `debug.mask` OID is declared with `CTLFLAG_RW`, so the kernel allows writes automatically; the handler does not need a privilege check because the OID is already restricted to root by the sysctl MIB permissions. The Chapter 25 driver does not add more custom sysctl handlers at this stage.

25.3.8 Error Path Messages

Returning the right `errno` is half the contract. Emitting the right log message is the other half. The discipline combines the rate-limited logging of Section 1 with the `errno` vocabulary of Section 2. A warning path looks like this:

```
if (input_too_large) {
    DLOG_RL(sc, &sc->sc_rl_inval, MYF_RL_DEFAULT_PPS,
        "ioctl: SETMSG buffer too large (%zu > %d)\n",
        length, MYFIRST_MSG_MAX);
    error = EINVAL;
    break;
}
```

Three properties make this a good error path. First, the log message names the call (“ioctl: SETMSG”), the reason (“buffer too large”), and the numeric values involved. Second, the errno returned is `EINVAL`, which is the right value for “I understood but the argument is wrong.” Third, the whole path is rate-limited so that a buggy user-space program calling the `ioctl` in a tight loop cannot flood the message buffer.

A bad error path looks like this:

```
if (input_too_large) {
    device_printf(sc->sc_dev, "ioctl failed\n");
    return (-1);
}
```

Three properties make this a bad error path. The log message is uninformative: “ioctl failed” says nothing the caller did not already know. The return value is `-1`, which is not a valid kernel errno. And the log line is not rate-limited, so a misbehaving caller can fill the message buffer with noise.

The good path takes nine lines and the bad path takes three, which is a good trade. An error log line is only printed when something is wrong; spending a few extra seconds to make it informative when it does print is worth the time.

25.3.9 Module Event Handler Conventions

The module event handler has its own errno convention. The handler signature is:

```
static int
myfirst_modevent(module_t mod, int what, void *arg)
{
    switch (what) {
    case MOD_LOAD:
        /* Driver-wide init. */
        return (0);
    case MOD_UNLOAD:
        /* Driver-wide teardown. */
        return (0);
    case MOD_QUIESCE:
        /* Pause and prepare for unload. */

```



```

        return (0);
    case MOD_SHUTDOWN:
        /* System shutting down. */
        return (0);
    default:
        return (EOPNOTSUPP);
    }
}

```

Each case returns 0 on success, or a positive errno to reject the event. The specific errnos by case:

- MOD_LOAD returns ENOMEM if a global allocation failed, ENXIO if the driver is not compatible with the current kernel, or EINVAL if a tunable's value is out of range.
- MOD_UNLOAD returns EBUSY if the driver cannot be unloaded right now because some instance is still in use. The kernel respects this and leaves the module loaded.
- MOD_QUIESCE returns EBUSY if the driver cannot pause. A driver that does not support quiescence simply returns 0 from this case, because quiescence is an optional feature and returning success says "I am paused" in the trivial sense of having no work in flight.
- MOD_SHUTDOWN rarely fails; it returns 0 unless the driver has a specific reason to object to shutdown. A driver that wants to flush persistent state uses an EVENTHANDLER on shutdown_pre_sync rather than rejecting MOD_SHUTDOWN.
- The default case returns EOPNOTSUPP to indicate the driver does not recognise the event type. This is not an error; it is the standard way to say "I do not implement this event."

25.3.10 An Errno Checklist

To close the section, a checklist the reader can run against any driver they write. Each item is a question whose answer should be yes.

1. Every non-zero return in a callback is a positive errno from `errno.h`, except for `d_ioctl` which may return `ENOIOCTL` (a negative value).
2. `copyin` and `copyout` failures propagate their errno unchanged (typically `EFAULT`).
3. The default case of `d_ioctl` returns `ENOIOCTL`, not `EINVAL`.
4. `d_detach` returns `EBUSY` if the device is still in use, not `ENXIO` or some other value.
5. `d_open` returns `ENXIO` if the underlying hardware is gone or if the driver is being detached, not `EIO`.
6. `d_write` returns `EBADF` if the file descriptor lacks `FWRITE`, not `EPERM`.
7. Every error path logs a message that names the call, the reason, and the relevant values, using the rate-limited macro.
8. No error path logs *and* returns a generic errno. If the driver has enough context to log the specific reason, it has enough context to return a specific errno.
9. The driver distinguishes `EINVAL` (arguments wrong) from `EOPNOTSUPP` (feature absent) consistently.
10. The driver distinguishes `ENOENT` (no such key) from `ENXIO` (device unreachable) consistently.

A driver that passes this checklist has a consistent errno surface, and the surface is small enough that the manual page can list every errno the driver returns and say exactly when each occurs.

25.3.11 Wrapping Up Section 2

Ernos are a small vocabulary and a contract. Sloppiness about either shows up as mystifying behaviour in user space; discipline about both shows up as accurate diagnostics and shorter bug reports. Combined with Section 1's rate-limited logging, the `myfirst` driver now talks carefully to both the operator (through log lines) and the caller (through ernos).

In the next section, we look at the third audience the driver owes a contract to: the administrator who configures the driver through `/boot/loader.conf` and `sysctl`. Configuration is a third kind of contract, and discipline about it is how the driver stays useful across workloads without needing a rebuild.

25.4 Section 3: Driver Configuration via Loader Tunables and `sysctl`

The third discipline is the discipline of externalising decisions. Any driver has values that someone might reasonably want to change without rebuilding the module: a timeout, a retry count, an internal buffer size, a verbosity level, a feature toggle. A driver that bakes those values into the source forces every change to go through a full compile, install, and reboot cycle. A driver that exposes them as loader tunables and `sysctls` lets an operator adjust behaviour at boot or at run time with a single edit or a single command. The cost of offering the knobs is small; the cost of not offering them is paid by the operator.

This section teaches the two FreeBSD mechanisms for externalising configuration: loader tunables (read from `/boot/loader.conf` and applied before the kernel reaches `attach`) and `sysctls` (read and written at run time through `sysctl(8)`). It explains how they cooperate through the `CTLFLAG_TUN` flag, shows how to choose between per-driver and per-instance tunables, walks through the `TUNABLE_*_FETCH` family, and ends with a short lab in which the `myfirst` driver gains three new tunables and the reader boots the VM with each.

25.4.1 The Difference Between a Tunable and a `Sysctl`

A tunable and a `sysctl` look similar to an operator. Both are strings in a namespace like `hw.myfirst.debug_mask_default` or `dev.myfirst.0.debug.mask`. Both take values the operator sets. Both end up in kernel memory. They differ in when and how.

A **tunable** is a variable set in the bootloader environment. The bootloader (`loader(8)`) reads `/boot/loader.conf`, collects its `key=value` pairs into an environment, and hands that environment to the kernel when the kernel starts. The kernel exposes this environment through the `getenv(9)` family and through the `TUNABLE_*_FETCH` macros. Tunables are read during boot, usually before the corresponding driver attaches. They cannot be changed at run time (changing `/boot/loader.conf` requires a reboot for the change to take effect). They are appropriate for values that must be known before `attach` runs: the size of a statically-allocated table, a feature flag that controls which code paths are compiled into the attach path, the initial value of a debug mask.

A **`sysctl`** is a variable in the kernel's hierarchical configuration tree, accessible at run time through the `sysctl(2)` syscall and the `sysctl(8)` tool. `sysctls` can be read-only (`CTLFLAG_RD`), read-write (`CTLFLAG_RW`), or read-only-root-writable (various combinations of flags). They are appropriate for values that make sense to change after the driver has attached: a verbosity level, a throttle rate, a counter reset command, a writable status knob.

The useful feature is that the two mechanisms can share a variable. A `sysctl` declared with `CTLFLAG_TUN` tells the kernel to read a tunable of the same name at boot and use its value as the initial value of the `sysctl`. The

operator can then adjust the sysctl at run time, and the tunable persists across reboots as the default. The `myfirst` driver already uses this pattern for its debug mask: `debug.mask` is a `CTLFLAG_RW | CTLFLAG_TUN` sysctl, and `hw.myfirst.debug_mask_default` is the matching tunable in `/boot/loader.conf`. Section 3 generalises that pattern to every configuration knob the driver wants to expose.

25.4.2 The `TUNABLE_*_FETCH` Family

FreeBSD provides a family of macros for reading tunables from the bootloader environment. Each macro reads the named tunable, parses it into the right C type, and stores the result. If the tunable is not set, the variable keeps its existing value; the caller must therefore initialise the variable to the right default before calling the fetch macro.

The macros, declared in `/usr/src/sys/sys/kernel.h`:

```
TUNABLE_INT_FETCH(path, pval)      /* int */
TUNABLE_LONG_FETCH(path, pval)     /* long */
TUNABLE_ULONG_FETCH(path, pval)    /* unsigned long */
TUNABLE_INT64_FETCH(path, pval)    /* int64_t */
TUNABLE_UINT64_FETCH(path, pval)   /* uint64_t */
TUNABLE_BOOL_FETCH(path, pval)     /* bool */
TUNABLE_STR_FETCH(path, pval, size) /* char buffer of given size */
```

Each expands to the matching `getenv_*` call. For `TUNABLE_INT_FETCH`, for example, the expansion is `getenv_int(path, pval)`, which reads the bootloader environment and parses the value as an integer.

The path is a string, conventionally of the form `hw.<driver>.<knob>` for per-driver tunables and `hw.<driver>.<unit>.<knob>` for per-instance tunables. The `hw.` prefix is a convention for hardware-related tunables; other prefixes (`kern.`, `net.`) exist for different subsystems but are less common in driver code.

A worked example from the `myfirst` driver shows the pattern:

```
static int
myfirst_attach(device_t dev)
{
    struct myfirst_softc *sc = device_get_softc(dev);
    int error;

    /* Initialise defaults. */
    sc->sc_debug = 0;
    sc->sc_timeout_sec = 30;
    sc->sc_max_retries = 3;

    /* Read tunables. The variables keep their default values if
     * the tunables are not set. */
    TUNABLE_INT_FETCH("hw.myfirst.debug_mask_default", &sc->sc_debug);
```

```
TUNABLE_INT_FETCH("hw.myfirst.timeout_sec", &sc->sc_timeout_sec);
TUNABLE_INT_FETCH("hw.myfirst.max_retries", &sc->sc_max_retries);

/* ... rest of attach ... */
}
```

The operator sets the tunables in `/boot/loader.conf`:

```
hw.myfirst.debug_mask_default="0xff"
hw.myfirst.timeout_sec="15"
hw.myfirst.max_retries="5"
```

After reboot, every instance of `myfirst` attaches with `sc_debug=0xff`, `sc_timeout_sec=15`, `sc_max_retries=5`. No rebuild was required; the values live outside the driver source.

25.4.3 Per-Driver Versus Per-Instance Tunables

A driver that might attach more than once needs a decision: should its tunables apply to every instance, or should each instance have its own?

The per-driver form uses a path of the form `hw.myfirst.debug_mask_default`. Every instance of `myfirst` at attach time reads this single variable, so all instances start with the same default. This is the simpler form and is correct when the tunable has the same meaning on every instance.

The per-instance form uses a path of the form `hw.myfirst.0.debug_mask_default`, where `0` is the unit number. Each instance reads its own variable, so instance `0` and instance `1` can have different defaults. This is the right form when the hardware behind each instance can reasonably need different configuration, for example two PCI adapters on the same system with different workloads.

The decision is a design choice, not a correctness question. Most drivers use the per-driver form for most tunables, with per-instance forms reserved for the few cases where per-instance configuration actually matters. For `myfirst`, a fictional pseudo-device, per-driver is the right default for every tunable. The Chapter 25 driver therefore adds three per-driver tunables (`timeout_sec`, `max_retries`, `log_ratelimit_pps`) and keeps the existing per-driver `debug_mask_default`.

A pattern that combines both forms, if the driver needs it, is to read the per-driver tunable first as a baseline and then read the per-instance tunable as an override:

```
int defval = 30;

TUNABLE_INT_FETCH("hw.myfirst.timeout_sec", &defval);
sc->sc_timeout_sec = defval;
TUNABLE_INT_FETCH_UNIT("hw.myfirst", unit, "timeout_sec",
    &sc->sc_timeout_sec);
```

FreeBSD does not have a `TUNABLE_INT_FETCH_UNIT` macro out of the box; a driver that needs this has to compose the path with `snprintf` and then call `getenv_int` by hand. The effort is small but the need is rare, so `myfirst` does not go there.

25.4.4 The `CTLFLAG_TUN` Flag

The second half of the externalisation story is that a tunable by itself is read only at boot. To make the same value adjustable at run time, the driver declares the matching sysctl with `CTLFLAG_TUN`:

```
SYSCALL_ADD_UINT(ctx, child, OID_AUTO, "debug.mask",
    CTLFLAG_RW | CTLFLAG_TUN,
    &sc->sc_debug, 0,
    "Bitmask of enabled debug classes");
```

`CTLFLAG_TUN` tells the kernel that the initial value of this sysctl should be taken from the bootloader environment variable of the same name, using the OID name as the key. The match is textual and automatic; the driver does not need to call `TUNABLE_INT_FETCH` separately.

There is a subtle rule about when `CTLFLAG_TUN` is honoured. The flag applies to the OID's *initial* value, which is read from the environment when the sysctl is created. If the driver calls `TUNABLE_INT_FETCH` explicitly before creating the sysctl, the explicit fetch wins and `CTLFLAG_TUN` is effectively redundant. If the driver does not call `TUNABLE_INT_FETCH` and relies on `CTLFLAG_TUN` alone, the sysctl's initial value comes from the environment automatically.

In practice, the `myfirst` driver uses both mechanisms for clarity. The explicit `TUNABLE_INT_FETCH` in `attach` makes the driver's intent visible in the source; the `CTLFLAG_TUN` on the sysctl gives the operator a clear hint in the sysctl documentation that the OID respects a loader tunable. Either mechanism alone would work; using both is a small duplication that pays off in readability.

25.4.5 Declaring a Tunable as a Static Sysctl

For driver-wide sysctls that do not belong to a specific instance, FreeBSD offers compile-time macros that bind a sysctl to a static variable and read its default from the environment in one declaration. The canonical form:

```
SYSCALL_NODE(_hw, OID_AUTO, myfirst, CTLFLAG_RW, NULL,
    "myfirst pseudo-driver");

static int myfirst_verbose = 0;
SYSCALL_INT(_hw_myfirst, OID_AUTO, verbose,
    CTLFLAG_RWTUN, &myfirst_verbose, 0,
    "Enable verbose driver logging");
```

The `SYSCALL_NODE` declares a new parent node `hw.myfirst`. The `SYSCALL_INT` declares an integer OID `hw.myfirst.verbose` with `CTLFLAG_RWTUN` (which combines `CTLFLAG_RW` and `CTLFLAG_TUN`). The `myfirst_verbose` variable is the driver's global verbosity level. The operator sets `hw.myfirst.verbose=1` in

`/boot/loader.conf` to enable verbose output at boot, or runs `sysctl hw.myfirst.verbose=1` to toggle it at run time.

The static declaration is appropriate for driver-wide state. The per-instance state (`sc_debug`, counters) continues to live under `dev.myfirst.<unit>.*` and is declared dynamically through `device_get_sysctl_ctx`.

25.4.6 A Small Side Note on `SYSCCTL_INT` Versus `SYSCCTL_ADD_INT`

The static form `SYSCCTL_INT(parent, OID_AUTO, ...)` is a compile-time declaration. The dynamic form `SYSCCTL_ADD_INT(ctx, list, OID_AUTO, ...)` is a runtime call. Both produce a `sysctl` OID. The static form is appropriate for driver-wide `sysctls` whose existence does not depend on attaching to hardware. The dynamic form is appropriate for per-instance `sysctls` created in `attach` and destroyed on `detach`.

A common beginner mistake is to use the dynamic form for driver-wide `sysctls`, which works but requires a driver-wide `sysctl_ctx_list` that must be initialised at `MOD_LOAD` and freed at `MOD_UNLOAD`. The static form avoids all of that: the `sysctl` exists from the moment the module loads until the moment it unloads, and the kernel handles the registration and deregistration automatically.

25.4.7 Documenting a Tunable

A tunable that the operator does not know about is a tunable that does not get used. The discipline is to document every tunable the driver exposes, in three places.

First, the tunable's declaration in the source should include a one-line description string. For `SYSCCTL_ADD_UINT` and friends, the last argument is the description:

```
SYSCCTL_ADD_UINT(ctx, child, OID_AUTO, "timeout_sec",
    CTLFLAG_RW | CTLFLAG_TUN,
    &sc->sc_timeout_sec, 0,
    "Timeout in seconds for hardware commands (default 30, min 1, max 3600)");
```

The description string is what `sysctl -d` prints when an operator asks for documentation. A good description names the unit, the default, and the acceptable range.

Second, the driver's `MAINTENANCE.md` (introduced in Section 7) should list every tunable with a paragraph each. The paragraph explains what the tunable does, when to change it, what the default is, and what side effects setting it has.

Third, the driver's manual page (typically `myfirst(4)`) should list every tunable under a `LOADER TUNABLES` section and every `sysctl` under a `SYSCCTL VARIABLES` section. The `myfirst` driver does not yet have a manual page; the chapter treats the manual page as a later concern. The `MAINTENANCE.md` document carries the full documentation in the meantime.

25.4.8 A Worked Example: `hw.myfirst.timeout_sec`

The `myfirst` driver does not have real hardware at this stage, but the chapter introduces a fictional `timeout_sec` knob that future chapters will use. The complete mini-workflow is:

1. In `myfirst.h`, add the field to the `softc`:

```
struct myfirst_softc {
    /* ... existing fields ... */
    int    sc_timeout_sec;
};
```

2. In `myfirst_bus.c` (the new file introduced in Section 6, which holds attach and detach), initialise the default and read the tunable:

```
sc->sc_timeout_sec = 30;
TUNABLE_INT_FETCH("hw.myfirst.timeout_sec", &sc->sc_timeout_sec);
```

3. In `myfirst_sysctl.c`, expose the knob as a runtime sysctl:

```
SYSCTL_ADD_INT(ctx, child, OID_AUTO, "timeout_sec",
    CTLFLAG_RW | CTLFLAG_TUN,
    &sc->sc_timeout_sec, 0,
    "Timeout in seconds for hardware commands");
```

4. In `MAINTENANCE.md`, document the tunable:

```
hw.myfirst.timeout_sec
    Timeout in seconds for hardware commands.  Default 30.
    Acceptable range 1 through 3600.  Values below 1 are
    clamped to 1; values above 3600 are clamped to 3600.
    Adjustable at run time via sysctl dev.myfirst.<unit>.
    timeout_sec.
```

5. In the regression script, add a line that verifies the tunable picks up its default:

```
[ "$(sysctl -n dev.myfirst.0.timeout_sec)" = "30" ] || fail
```

The driver now has a timeout knob that the operator can set at boot through `/boot/loader.conf`, can adjust at run time through `sysctl`, and can find documented in `MAINTENANCE.md`. Every future chapter that introduces a new configurable value will follow the same five-step workflow.

25.4.9 Range Checks and Validation

A tunable that the operator can set to any value is a tunable that can be set to an out-of-range value, either by accident (a typo in `/boot/loader.conf`) or by a misguided attempt at tuning. The driver must validate the value it reads and clamp or reject it.

For tunables read at boot with `TUNABLE_INT_FETCH`, the validation happens inline:

```
sc->sc_timeout_sec = 30;
TUNABLE_INT_FETCH("hw.myfirst.timeout_sec", &sc->sc_timeout_sec);
if (sc->sc_timeout_sec < 1 || sc->sc_timeout_sec > 3600) {
    device_printf(dev,
```

```

        "tunable hw.myfirst.timeout_sec out of range (%d), "
        "clamping to default 30\n",
        sc->sc_timeout_sec);
    sc->sc_timeout_sec = 30;
}

```

For sysctls with runtime write support, the validation happens in the handler. A simple CTLFLAG_RW sysctl on an int variable accepts any int; to reject out-of-range writes, the driver declares a custom handler:

```

static int
myfirst_sysctl_timeout(SYSCTL_HANDLER_ARGS)
{
    struct myfirst_softc *sc = arg1;
    int v;
    int error;

    v = sc->sc_timeout_sec;
    error = sysctl_handle_int(oidp, &v, 0, req);
    if (error != 0 || req->newptr == NULL)
        return (error);
    if (v < 1 || v > 3600)
        return (EINVAL);
    sc->sc_timeout_sec = v;
    return (0);
}

```

The handler reads the current value, calls `sysctl_handle_int` to perform the actual I/O, and applies the new value only if it is in range. A write of 0 or 7200 returns `EINVAL` to the operator without changing the sysctl's value. This is the right behaviour: the operator gets clear feedback that the write was rejected.

The `myfirst` driver at this stage does not validate its integer sysctls because none of them can meaningfully go out of range (the debug mask is a bitmask, and any 32-bit value is a legal mask). Future drivers that introduce timeouts, retry counts, and buffer sizes will use the custom-handler pattern consistently.

25.4.10 When to Expose a Tunable and When to Leave It Internal

Exposing a tunable is a commitment. Once the operator sets `hw.myfirst.timeout_sec=15` in `/boot/loader.conf`, the driver has made a promise that the meaning of that knob will not change in a later version. Removing the tunable breaks production deployments. Changing its interpretation silently breaks them worse.

The right discipline is to expose a value as a tunable only when all three of these are true:

1. The value has an operational use case. Someone might reasonably need to change it on a real deployment.
2. The range of reasonable values is known. The driver can document it in `MAINTENANCE.md`.
3. The cost of supporting the knob for the life of the driver is worth the operational value it provides.

A driver that asks itself those three questions and answers yes to all three exposes a small, purposeful set of tunables. A driver that exposes every internal constant as a tunable because “operators might want to tune it” ends up with a sprawling configuration surface that nobody can document and nobody can test across its full range.

For `myfirst`, the initial set of tunables is deliberately small: `debug_mask_default`, `timeout_sec`, `max_retries`, `log_ratelimit_pps`. Each has a clear operational case, a clear default, and a clear range. The driver is not trying to expose every int field in the `softc` as a tunable; it is trying to expose the ones that an operator might actually want to touch.

25.4.11 A Cautionary Note on `CTLFLAG_RWTUN` for Strings

The `TUNABLE_STR_FETCH` macro reads a string from the bootloader environment into a fixed-size buffer. The matching sysctl flag, `CTLFLAG_RWTUN` on a `SYSCTL_STRING`, works, but it has a trap: the string’s storage must be a static buffer, not a per-instance `char[]` field in the `softc`. A string sysctl that writes into a `softc` field can outlive the `softc` if the sysctl framework does not deregister the OID before the `softc` is freed, which leads to use-after-free bugs.

The safer pattern is to expose strings as read-only and handle writes through a custom handler that copies the new value into the `softc` under a lock. The `myfirst` driver follows this pattern: `dev.myfirst.0.message` is exposed with `CTLFLAG_RD` only, and writes go through the `MYFIRSTIOC_SETMSG` ioctl instead. The ioctl path takes the `softc` mutex, copies the new value, and unlocks; there is no lifetime issue with sysctl OIDs.

String tunables and sysctls are useful enough for some drivers to be worth the care, but the Chapter 25 driver does not need them. The principle is worth naming because the trap surfaces later in real drivers.

25.4.12 Tunables Versus Kernel Modules: Where They Live

Two small but important details about the loader environment are worth naming.

First, a tunable in `/boot/loader.conf` applies from the moment the kernel starts. It is available to any module that calls `TUNABLE_*_FETCH` or has a `CTLFLAG_TUN` sysctl, even if the module is not loaded at boot. A module loaded later with `kldload` still sees the tunable’s value. This is convenient: the operator sets the tunable once and forgets about it until the module loads.

Second, a tunable is read from the environment but cannot be written back. Changing `hw.myfirst.timeout_sec` at run time (with `kenv`) does not affect any driver that has already read it; the variable in the `softc` is what matters, not the environment. To change a value at run time, the operator uses the matching sysctl.

These two details together explain why `CTLFLAG_TUN` is the right shape for most configuration knobs: the tunable sets the boot default, the sysctl handles run-time adjustment, and the operator’s toolkit (`/boot/loader.conf` plus `sysctl(8)`) works as expected.

25.4.13 Wrapping Up Section 3

Configuration is a conversation with the operator. A driver that externalises the right values through tunables and sysctls can be tuned without rebuilding; a driver that hides every value inside the source forces a rebuild for every change. The `TUNABLE_*_FETCH` family and the `CTLFLAG_TUN` flag together cover boot-time and run-time

adjustment, and the per-driver versus per-instance choice fits the driver to its operational reality. The `myfirst` driver now has three new tunables in addition to the existing `debug_mask_default`, each with a documented range and each with a matching `sysctl`.

In the next section, we turn from what the driver exposes to how the driver evolves. A configuration knob that works today must still work tomorrow when the driver changes. Versioning discipline is what keeps that promise.

25.5 Section 4: Versioning and Compatibility Strategies

The fourth discipline is the discipline of evolving without breaking. Every public surface the `myfirst` driver offers, the `/dev/myfirst0` node, the `ioctl` interface, the `sysctl` tree, the tunable set, is a contract with someone. A change that silently alters the meaning of any of those is a breaking change, and breaking changes that slip past the developer's notice are the source of a disproportionate number of real-world driver bugs. This section teaches how to version a driver's public surface deliberately, so that changes are visible to callers and so that older callers keep working when the driver adds new features.

The chapter uses three distinct version numbers for the `myfirst` driver. Each has a specific purpose. Conflating them is a source of confusion worth avoiding before it takes root.

25.5.1 The Three Version Numbers

The `myfirst` driver has three version identifiers, introduced across Chapters 23, 24, and 25. Each lives in a different place and changes for a different reason.

The first is the **human-readable release string**. For `myfirst`, this is `MYFIRST_VERSION`, defined in `myfirst_sysctl.c` and exposed through the `dev.myfirst.0.version` `sysctl`. Its current value is "1.8-maintenance". The release string is for humans: an operator who runs `sysctl dev.myfirst.0.version` sees a short label that identifies this particular checkpoint of the driver's history. The release string is not parsed by programs; it is read by people. It changes whenever the driver reaches a new milestone the author wants to mark, which in this book is at the end of each chapter.

The second is the **kernel module version integer**. This is `MODULE_VERSION(myfirst, N)`, where `N` is an integer used by the kernel's dependency machinery. Another module that declares `MODULE_DEPEND(other, myfirst, 1, 18, 18)` requires `myfirst` to be present at version 18 or above (and below or equal to 18, which in this declaration means exactly 18). The module version integer changes only when an in-kernel caller of the module would need a recompile, for example when a shared symbol's signature changes. For a driver that exports no public kernel symbols (like `myfirst`), the module version number is mostly symbolic; the chapter bumps it at each milestone to keep the reader's mental model aligned across the three version identifiers.

The third is the **ioctl interface version integer**. For `myfirst`, this is `MYFIRST_IOCTL_VERSION` in `myfirst_ioctl.h`. Its current value is 1. The `ioctl` interface version changes when the `ioctl` header changes in a way that an older user-space program compiled against the previous version would misinterpret. A renumbered `ioctl` command, a changed payload layout, a changed semantics of an existing `ioctl`: each of these is a breaking change to the `ioctl` interface and must bump the version. Adding a new `ioctl` command, extending a payload with a field at the end without reinterpreting existing fields, adding a feature that does not affect older commands: these are compatible changes and do not require a bump.

A simple rule of thumb keeps the three straight. The release string is what the operator reads. The module version integer is what other modules check. The ioctl version integer is what user-space programs check. Each moves on its own schedule.

25.5.2 Why Users Need to Query the Version

A user-space program that talks to the driver through ioctls has a problem. The header `myfirst_ioctl.h` defines a set of commands, layouts, and version-1 semantics. A new version of the driver may add commands, change layouts, or change semantics. When the user-space program runs on a system with a newer or older driver than the one it was compiled against, it has no way to know the driver's actual version unless it asks.

The solution is an ioctl whose only purpose is to return the driver's ioctl version. The `myfirst` driver already has one: `MYFIRSTIOC_GETVER`, defined as `_IOR('M', 1, uint32_t)`. A user-space program calls this ioctl immediately after opening the device, compares the returned version to the version it was compiled against, and decides whether it can proceed safely.

The pattern in user space:

```
#include "myfirst_ioctl.h"

int fd = open("/dev/myfirst0", O_RDWR);
uint32_t ver;
if (ioctl(fd, MYFIRSTIOC_GETVER, &ver) < 0)
    err(1, "getver");
if (ver != MYFIRST_IOCTL_VERSION)
    errx(1, "driver version %u, tool expects %u",
        ver, MYFIRST_IOCTL_VERSION);
```

The tool refuses to run if the versions do not match. That is one possible policy. A more forgiving policy would allow the tool to run against a newer driver if the driver's new ioctls are supersets of the old ones, and would allow the tool to run against an older driver if the tool can fall back to the older command set. A more rigid policy would require exact match. The tool's author chooses among these based on how much effort is worth spending on backward compatibility.

25.5.3 Adding a New Ioctl Without Breaking Older Callers

The common case is adding a new feature to the driver, which usually means adding a new ioctl. The discipline is straightforward as long as two rules are followed.

First, **do not reuse an existing ioctl number**. Each ioctl command has a unique (magic, number) pair encoded by `_IO`, `_IOR`, `_IOW`, or `_IOWR`. The current assignments in `myfirst_ioctl.h`:

```
#define MYFIRSTIOC_GETVER    _IOR('M', 1, uint32_t)
#define MYFIRSTIOC_GETMSG    _IOR('M', 2, char[MYFIRST_MSG_MAX])
#define MYFIRSTIOC_SETMSG    _IOW('M', 3, char[MYFIRST_MSG_MAX])
#define MYFIRSTIOC_RESET    _IO('M', 4)
```

A new ioctl takes the next available number under the same magic letter: `MYFIRSTIOC_GETCAPS = _IOR('M', 5, uint32_t)`. The number 5 has not been used before and cannot conflict with an older program's compiled binary. An older program compiled against a version without `GETCAPS` simply never sends that ioctl, so the older program is unaffected by the addition.

Second, **do not bump `MYFIRST_IOCTL_VERSION` for a pure addition**. A new ioctl that does not change the meaning of the old ones is a compatible change. An older user-space program that never heard of the new ioctl still speaks the same language; the version integer should stay the same. Bumping the version for every addition would force every caller to rebuild whenever the driver gains a new command, which defeats the purpose of versioning.

A new ioctl that replaces an existing one with different semantics does require a bump. If the driver adds `MYFIRSTIOC_SETMSG_V2` with a new layout and retires `MYFIRSTIOC_SETMSG`, older programs that call the retired command see a changed behaviour (the driver may return `ENOIOCTL` or may behave differently). That is a breaking change, and the bump signals it.

25.5.4 Retiring a Deprecated Ioctl

Retirement is the politely-managed form of removal. When a command is to be removed, the driver announces the intention, keeps the command working for a transition period, and removes it in a later version. A typical deprecation sequence:

- Version N: announce deprecation in `MAINTENANCE.md`. The command still works.
- Version N+1: the command works but logs a rate-limited warning each time it is used. Users see the warning and know to migrate.
- Version N+2: the command returns `EOPNOTSUPP` and logs a rate-limited error. Most users have migrated by now; the few who have not are forced to.
- Version N+3: the command is removed from the header. Programs that still reference it no longer compile.

The transition period should be measured in releases (typically one or two major versions) rather than in calendar time. A driver that keeps its deprecation contract predictable gives consumers a stable target to aim at.

For `myfirst` in this chapter, no command is yet deprecated. The chapter introduces the pattern for the future. The same discipline applies to the `sysctl` tree: a rate-limited warning in the `OID`'s handler tells operators that the name is on its way out, and a note in `MAINTENANCE.md` records the planned removal date.

25.5.5 The Capability Bitmask Pattern

For drivers that evolve over several releases, a single version integer tells callers which version they are talking to but not which specific features that version supports. A feature-rich driver benefits from a more fine-grained mechanism: a capability bitmask.

The idea is simple. The driver defines a set of capability bits in `myfirst_ioctl.h`:

```
#define MYF_CAP_RESET      (1U << 0)
#define MYF_CAP_GETMSG     (1U << 1)
#define MYF_CAP_SETMSG     (1U << 2)
```

```
#define MYF_CAP_TIMEOUT      (1U << 3)
#define MYF_CAP_MAXRETRIES  (1U << 4)
```

A new ioctl, MYFIRSTIOC_GETCAPS, returns a uint32_t with the bits set for the features this driver actually supports:

```
#define MYFIRSTIOC_GETCAPS  _IOR('M', 5, uint32_t)
```

In the kernel:

```
case MYFIRSTIOC_GETCAPS:
    *(uint32_t *)data = MYF_CAP_RESET | MYF_CAP_GETMSG |
        MYF_CAP_SETMSG;
    break;
```

In user space:

```
uint32_t caps;
ioctl(fd, MYFIRSTIOC_GETCAPS, &caps);
if (caps & MYF_CAP_TIMEOUT)
    set_timeout(fd, 60);
else
    warnx("driver does not support timeout configuration");
```

The capability bitmask allows a user-space program to discover features without trial and error. If the caller wants to know whether a feature exists, it checks the bit; if the bit is set, the caller knows the driver supports the feature and the relevant ioctls. An older driver that does not define the bit does not pretend to support a feature it never heard of.

The pattern scales well as the driver grows. Each release adds new bits for new features. Retired features keep their bit reserved as unused; recycling a bit for a new meaning would be a breaking change. The bitmask itself is a uint32_t, giving the driver 32 features before it needs to add a second word. If the driver reaches 32 features, adding a second word is a compatible change (the new bits are in a new field, so older programs that read only the first word see the same bits).

Chapter 25 adds MYFIRSTIOC_GETCAPS to the myfirst driver with three bits set: MYF_CAP_RESET, MYF_CAP_GETMSG, and MYF_CAP_SETMSG. The myfirstctl user-space program is extended to query the capabilities at startup and to refuse to invoke an unsupported feature.

25.5.6 Sysctl Deprecation

FreeBSD does not offer a dedicated CTLFLAG_DEPRECATED flag on the sysctl tree. The related flag CTLFLAG_SKIP, defined in /usr/src/sys/sys/sysctl.h, hides an OID from default listings (it is still readable if named explicitly), but it is primarily used for purposes other than announcing retirement. The polite

way to retire a sysctl OID is therefore to replace its handler with one that does the intended work *and* logs a rate-limited warning the first few times the OID is touched.

```
static int
myfirst_sysctl_old_counter(SYSCTL_HANDLER_ARGS)
{
    struct myfirst_softc *sc = arg1;

    DLOG_RL(sc, &sc->sc_rl_deprecated, MYF_RL_DEFAULT_PPS,
        "sysctl dev.myfirst.%d.old_counter is deprecated; "
        "use new_counter instead\n",
        device_get_unit(sc->sc_dev));
    return (sysctl_handle_int(oidp, &sc->sc_old_counter, 0, req));
}
```

The operator sees the warning in `dmesg` the first few times the OID is read, which is a strong hint to migrate. The sysctl still works, so scripts that reference it explicitly do not break during the transition. After a release or two, the OID itself is removed. A note in `MAINTENANCE.md` records the intention and the target release.

For `myfirst`, no sysctl is yet deprecated. The Chapter 25 driver introduces the pattern in documentation and keeps it ready for future use.

25.5.7 User-Visible Behaviour Changes

Not every breaking change is a rename or a renumbering. Sometimes the driver keeps the same ioctl, the same sysctl, the same tunable, and quietly changes what the operation does. A `MYFIRSTIOC_RESET` that used to zero counters but now also clears the message is a behaviour change. A sysctl that used to report total bytes written but now reports kilobytes is a behaviour change. A tunable that used to be an absolute value and is now a multiplier is a behaviour change.

Behaviour changes are the hardest breaking changes to catch because they do not show up in diffs of header files or sysctl listings. The discipline is to document every behaviour change in `MAINTENANCE.md` under a “Change Log” section, to bump the ioctl interface version integer when an ioctl’s semantics change, and to announce sysctl semantics changes in the description string itself.

A good pattern for behaviour changes is to introduce a new named command or a new sysctl rather than redefining an existing one. `MYFIRSTIOC_RESET` keeps the old semantics. `MYFIRSTIOC_RESET_ALL` is a new command with the new semantics. The old command is eventually deprecated. The cost is a slightly larger public surface for the transition period; the benefit is that no caller is broken by a silent behaviour change.

25.5.8 MODULE_DEPEND and Inter-Module Compatibility

The `MODULE_DEPEND` macro declares that one module depends on another and requires a specific version range:

```
MODULE_DEPEND(myfirst, dependency, 1, 2, 3);
```

The three integers are the minimum, preferred, and maximum version of dependency that `myfirst` is compatible with. The kernel refuses to load `myfirst` if dependency is not present or is outside the range.

For drivers that do not publish in-kernel symbols, `MODULE_DEPEND` is most often used to depend on a standard subsystem module:

```
MODULE_DEPEND(myfirst_usb, usb, 1, 1, 1);
```

This declares that the USB version of `myfirst` needs the USB stack at version 1 exactly. The version numbers for subsystem modules are managed by the subsystem authors; a driver author finds the current values in the subsystem's header (for USB, `/usr/src/sys/dev/usb/usbd.h`) or in another driver that already depends on it.

For `myfirst` at the end of Chapter 25, no `MODULE_DEPEND` is needed because the pseudo-driver does not require a subsystem. Chapter 26's USB chapter will add the first real `MODULE_DEPEND` when the driver is turned into a USB-attached version.

25.5.9 A Worked Example: The 1.7 to 1.8 Transition

The Chapter 25 driver bumps three version identifiers at the end of the chapter:

- `MYFIRST_VERSION`: from "1.7-integration" to "1.8-maintenance".
- `MODULE_VERSION(myfirst, N)`: from 17 to 18.
- `MYFIRST_IOCTL_VERSION`: stays at 1, because the ioctl additions in this chapter are pure additions (new commands, no removal, no semantic changes).

The `GETCAPS` ioctl is added with command number 5, which was previously unused. Older `myfirstctl` binaries, compiled against the Chapter 24 version of the header, do not know about `GETCAPS` and do not send it; they continue to work unchanged. New `myfirstctl` binaries, compiled against the Chapter 25 header, query `GETCAPS` at startup and behave accordingly.

The `MAINTENANCE.md` document gains a Change Log entry for 1.8:

```
## 1.8-maintenance
```

- Added `MYFIRSTIOC_GETCAPS` (command 5) returning a capability bitmask. Compatible with all earlier user-space programs.
- Added tunables `hw.myfirst.timeout_sec`, `hw.myfirst.max_retries`, `hw.myfirst.log_ratelimit_pps`. Each has a matching writable sysctl under `dev.myfirst.<unit>`.
- Added rate-limited logging through `ppsratecheck(9)`.
- No breaking changes from 1.7.

A user of the driver reading `MAINTENANCE.md` sees at a glance what changed and can evaluate whether they need to update their tools. A user who does not read `MAINTENANCE.md` can still query the capabilities at run time and discover the new features programmatically.

25.5.10 Common Mistakes in Versioning

Three mistakes are common when first applying version discipline. Each is worth naming.

The first mistake is **reusing an ioctl number**. A number that was once assigned and later retired stays retired. A new command gets the next available number, not the number of a retired command. Reusing a number silently breaks older callers that had compiled in the old meaning; the compiler has no way to detect the conflict because the retired command's header was removed.

The second mistake is **bumping the version integer for every change**. If every patch bumps `MY-FIRST_IOCTL_VERSION`, user-space tools have to rebuild constantly or the version check fails. The integer should change only for genuine breaking changes. Pure additions leave it alone.

The third mistake is **treating the release string as a semantic version**. The release string is for humans; it can be anything. The module version integer and the ioctl version integer are parsed by programs and should follow a discipline (monotonically increasing, bumped only for specific reasons). Confusing the two leads to confusing version numbers.

25.5.11 Wrapping Up Section 4

Versioning is the discipline of evolving without breaking. A driver that keeps its three version identifiers distinct, its ioctl additions compatible, its deprecations announced, and its capability bits accurate gives its callers a stable target over the long life of the driver. The `myfirst` driver now has a working `GETCAPS` ioctl, a documented deprecation policy in `MAINTENANCE.md`, and three version identifiers that each change for their own reason. Everything a future developer needs to add a feature or retire a command is already in place.

In the next section, we turn from the driver's public surface to its private resource discipline. A driver that crashes on attach failure is a driver that cannot recover from any error. The labelled-goto pattern is how FreeBSD drivers make every allocation reversible.

25.6 Section 5: Managing Resources in Failure Paths

Every attach routine is an ordered sequence of acquisitions. It allocates a lock, creates a `cdev`, hangs a `sysctl` tree on the device, perhaps registers an event handler or a timer, and on more complex drivers it allocates bus resources, maps I/O windows, attaches an interrupt, and sets up DMA. Each acquisition can fail. And each acquisition that succeeded before the failure must be released in the reverse order, or the kernel leaks memory, leaks a lock, leaks a `cdev`, and in the worst case keeps a device node alive with a stale pointer inside it.

The `myfirst` driver has been growing its attach path one section at a time since Chapter 17. Attach started small: a lock and a `cdev`. Chapter 24 added the `sysctl` tree. Chapter 25 is about to add rate-limit state, tunable-fetched defaults, and a counter or two. The order in which those resources are acquired now matters to the cleanup path. Every new acquisition has to know where in the unwind ordering it belongs, and the unwind itself has to be structured so that adding a new resource next week does not force a rewrite of the attach function.

Chapter 20 introduced the pattern informally; this section gives it a name, a vocabulary, and a discipline strong enough to survive the full Chapter 25 shape of `myfirst_attach`.

25.6.1 The Problem: Nested if Paths Do Not Scale

The naive shape of an attach routine is a ladder of nested if statements. Each success condition contains the next step. Each failure returns. The problem is that every failure has to unwind whatever the previous steps already did, and the unwind code is duplicated at every level of the ladder:

```
/*
 * Naive attach. DO NOT WRITE DRIVERS THIS WAY. This example shows
 * how the nested-if pattern forces duplicated cleanup at every level
 * and why it becomes unmaintainable as soon as a fourth resource is
 * added to the chain.
 */
static int
myfirst_attach_bad(device_t dev)
{
    struct myfirst_softc *sc = device_get_softc(dev);
    struct make_dev_args args;
    int error;

    sc->sc_dev = dev;
    mtx_init(&sc->sc_mtx, "myfirst", NULL, MTX_DEF);

    make_dev_args_init(&args);
    args.mda_devsw = &myfirst_cdevsw;
    args.mda_uid = UID_ROOT;
    args.mda_gid = GID_WHEEL;
    args.mda_mode = 0660;
    args.mda_si_drv1 = sc;
    args.mda_unit = device_get_unit(dev);

    error = make_dev_s(&args, &sc->sc_cdev, "myfirst%d",
        device_get_unit(dev));
    if (error == 0) {
        myfirst_sysctl_attach(sc);
        if (myfirst_log_attach(sc) == 0) {
            /* all resources held; we succeeded */
            return (0);
        } else {
            /* log allocation failed: undo sysctl and cdev */
            /* but wait, sysctl is owned by Newbus, so skip it */
            destroy_dev(sc->sc_cdev);
            mtx_destroy(&sc->sc_mtx);
            return (ENOMEM);
        }
    } else {
```

```

        mtx_destroy(&sc->sc_mtx);
        return (error);
    }
}

```

Even in this small example the unwinding logic appears in two different places, the reader has to read a branch to know which resources have been acquired at each point, and adding a fourth resource forces another level of nesting and another duplicated cleanup block. Real drivers have seven or eight resources. A driver like `if_em` at `/usr/src/sys/dev/e1000/if_em.c` has over a dozen. Nested `if` is not an option there.

The failure modes of the nested pattern are not theoretical. A common bug pattern in older FreeBSD drivers was a missing `mtx_destroy` or a missing `bus_release_resource` in one of the cleanup branches: one branch destroyed the lock, another forgot. Every branch was a chance to make a mistake, and the bug only showed up when that specific failure fired, which meant it often did not show up until a customer reported a panic on a device that failed to attach.

25.6.2 The `goto fail;` Pattern

FreeBSD's answer to the nested-cleanup problem is the labelled-goto pattern. The attach function is written as a linear sequence of acquisitions. Each acquisition that can fail is followed by a test that either falls through on success or jumps to a cleanup label on failure. The cleanup labels are ordered from most-acquired back to least-acquired. Each label releases the resources that were held at that point and then falls through to the next label. The function ends with a single `return (0)` on success and a single `return (error)` at the bottom of the cleanup chain:

```

static int
myfirst_attach(device_t dev)
{
    struct myfirst_softc *sc = device_get_softc(dev);
    struct make_dev_args args;
    int error;

    /* Resource 1: softc basics. Cannot fail. */
    sc->sc_dev = dev;

    /* Resource 2: the lock. Cannot fail on DEF mutex. */
    mtx_init(&sc->sc_mtx, "myfirst", NULL, MTX_DEF);

    /* Resource 3: the cdev. Can fail. */
    make_dev_args_init(&args);
    args.mda_devsw = &myfirst_cdevsw;
    args.mda_uid = UID_ROOT;
    args.mda_gid = GID_WHEEL;
    args.mda_mode = 0660;
}

```

```

args.mda_si_drv1 = sc;
args.mda_unit    = device_get_unit(dev);

error = make_dev_s(&args, &sc->sc_cdev, "myfirst%d",
    device_get_unit(dev));
if (error != 0)
    goto fail_mtx;

/* Resource 4: the sysctl tree. Cannot fail (Newbus owns it). */
myfirst_sysctl_attach(sc);

/* Resource 5: the log state. Can fail. */
error = myfirst_log_attach(sc);
if (error != 0)
    goto fail_cdev;

/* All resources held. Announce and return. */
DPRINTF(sc, MYF_DBG_INIT,
    "attach: version 1.8-maintenance ready\n");
return (0);

fail_cdev:
    destroy_dev(sc->sc_cdev);
fail_mtx:
    mtx_destroy(&sc->sc_mtx);
    return (error);
}

```

Read the function top to bottom. Each step is a resource acquisition. Each failure check is a two-line block: if the acquisition failed, jump to the label named after the previously acquired resource. The labels at the bottom release the resources in reverse order and fall through to the next label. The final `return (error)` returns the `errno` from whichever acquisition failed.

This shape scales. Adding a sixth resource means adding one acquisition block at the top, one `goto` target at the bottom, and one line of cleanup code. No nesting, no duplication, no branch tree. The same rule that governs the attach path governs every future addition to the attach path: acquire, test, go-to-the-previous-label, release in the reverse order.

25.6.3 Why Linear Unwinding Is the Right Shape

The value of the labelled-goto pattern is not purely stylistic. It maps directly onto the structural property that the attach sequence is a stack of resources, and cleanup is the pop operation on that stack.

A stack has three properties that are easy to state and easy to violate. First, resources are released in the reverse order of acquisition. Second, a failed acquisition does not add a resource to the stack, so cleanup starts from the

previously acquired resource, not from the one that just failed. Third, every resource on the stack is released exactly once: not zero times, not twice.

Each of these properties has a visible correlate in the `goto fail;` pattern. The cleanup labels appear in the file in the reverse order of the acquisitions: the last acquisition's cleanup label is at the top of the cleanup chain. A failed acquisition jumps to the label named after the previous acquisition, not after itself; the name of the label is literally the name of the resource that now has to be undone. And because each label falls through to the next, and each resource appears in exactly one label, every resource is freed exactly once on every failure path.

The stack discipline is what makes the pattern robust. If a reader wants to audit the cleanup path for correctness, they do not have to read branches. They just count the labels, count the acquisitions, and compare.

25.6.4 Label Naming Conventions

Labels in FreeBSD drivers traditionally begin with `fail_` followed by the name of the resource that is about to be undone. The name of the resource matches the name of the field in the `softc` or the name of the function called to acquire it. Common patterns seen across the tree:

- `fail_mtx` undoes `mtx_init`
- `fail_sx` undoes `sx_init`
- `fail_cdev` undoes `make_dev_s`
- `fail_ires` undoes `bus_alloc_resource` for an IRQ
- `fail_mres` undoes `bus_alloc_resource` for a memory window
- `fail_intr` undoes `bus_setup_intr`
- `fail_dma_tag` undoes `bus_dma_tag_create`
- `fail_log` undoes a driver-private allocation (the rate-limit block in `myfirst`)

Some older drivers use numbered labels (`fail1`, `fail2`, `fail3`). Numbered labels are legal but inferior: adding a resource in the middle of the sequence forces renumbering every label after the insertion point, and the label numbers do not tell the reader which resource is being cleaned up. Named labels survive insertions gracefully and document themselves.

Whatever convention a driver picks, it should be consistent across all of its files. `myfirst` uses the `fail_<resource>` convention for every attach function from this chapter forward.

25.6.5 The Rule of Falling Through

The single rule that every cleanup chain must obey is that each cleanup label falls through to the next. A stray return in the middle of the chain, or a missing label, skips cleanup for the resources that should have been released. The compiler does not warn about either mistake.

Consider what happens if a developer edits the cleanup chain and accidentally writes this:

```
fail_cdev:
    destroy_dev(sc->sc_cdev);
    return (error);          /* BUG: skips mtx_destroy. */
fail_mtx:
```

```
mtx_destroy(&sc->sc_mtx);  
return (error);
```

The first return prevents the `mtx_destroy` from running on the `fail_cdev` path. The lock is leaked. The kernel's witness code will not complain, because the leaked lock is never acquired again. The leak persists until the machine reboots. It is invisible in normal operation and only shows up as a slow memory bloat on a system where the driver attaches and fails repeatedly (a hot-pluggable device, for example).

The way to prevent this kind of bug is to write the cleanup chain with a single return at the bottom and no intermediate returns. The labels in the middle contain only the cleanup call for their resource. Fall-through is the default and intended behaviour:

```
fail_cdev:  
    destroy_dev(sc->sc_cdev);  
fail_mtx:  
    mtx_destroy(&sc->sc_mtx);  
    return (error);
```

A reader auditing the chain reads it as a simple list: destroy cdev, destroy lock, return. There is no branching to follow, and adding a label means adding a single line of cleanup code and optionally a single new target.

25.6.6 What the Success Path Looks Like

The attach function succeeds with a single return (0), placed immediately before the first cleanup label. This is the point at which every acquisition has succeeded and no cleanup is needed. The return (0) separates the acquisition chain from the cleanup chain visually: everything above it is acquisition, everything below it is cleanup.

Some drivers forget this separation and fall through from the last acquisition into the first cleanup label, freeing resources they just acquired. A stray missing return (0) is the simplest way to produce this bug:

```
/* Resource N: the final acquisition. */  
...  
  
/* Forgot to put a return here. */  
  
fail_cdev:  
    destroy_dev(sc->sc_cdev);
```

Without the return (0), control falls through into `fail_cdev` after every successful attach, destroying the cdev on the success path. The driver then reports attach as failed because `error` is zero and the kernel sees the successful return, but the cdev it just created is gone. The result is a device node that disappears seconds after it appears. Debugging this requires noticing that the attach message prints but the device does not respond; not an easy bug to find in a busy log.

The defence is discipline. Every attach function ends its acquisition chain with `return (0);` on its own line, followed by a blank line, followed by the cleanup labels. No exceptions. A linter like `igor` or a reviewer's eye catches violations quickly when the shape is always the same.

25.6.7 When an Acquisition Cannot Fail

Some acquisitions cannot fail. `mtx_init` for a default-style mutex cannot return an error. `sx_init` cannot. `callout_init_mtx` cannot. `SYSCTL_ADD_*` calls cannot return an error that a driver is expected to check (a failure there is a kernel-internal problem, not a driver problem).

For acquisitions that cannot fail, there is no `goto`. The acquisition is followed by the next step without a test. The cleanup label for the acquisition is still required, because the cleanup chain has to release the resource if a later acquisition fails:

```
mtx_init(&sc->sc_mtx, "myfirst", NULL, MTX_DEF);

error = make_dev_s(&args, &sc->sc_cdev, ...);
if (error != 0)
    goto fail_mtx;          /* undoes the lock. */
```

`fail_mtx` exists even though `mtx_init` itself had no failure path, because the lock still needs to be destroyed if anything below it fails.

The pattern holds: every acquired resource has a label, whether or not its acquisition can fail.

25.6.8 Helpers to Reduce Duplication

When several acquisitions share the same shape (allocate, check, `goto` on error), it is tempting to hide them behind a helper function. The helper's job is to consolidate the acquisition and the check; the caller only sees a single `if (error != 0) goto fail_x;` line. This is fine as long as the helper follows the same discipline: on failure, it releases nothing that it acquired partially, and it returns a meaningful `errno` so the caller's `goto` target can rely on it.

In `myfirst`, Section 5's companion examples introduce a helper called `myfirst_log_attach` that allocates the rate-limit state, initialises its fields, and returns 0 on success or a non-zero `errno` on failure. The attach function calls it with a single line:

```
error = myfirst_log_attach(sc);
if (error != 0)
    goto fail_cdev;
```

The helper itself follows the same pattern internally. If it allocates two resources and the second fails, the helper unwinds the first before returning. The caller sees the helper as a single atomic acquisition: it either fully succeeded or fully failed, and the caller never has to worry about the helper's intermediate state.

Helpers that are too eager to simplify, however, break the pattern. A helper that allocates a resource and stores it into the `softc` is fine. A helper that allocates a resource, stores it into the `softc`, and also releases it on error is

not fine: the caller's cleanup label will also try to release it, leading to a double-free. The rule is that acquisition helpers either succeed and leave the resource in the softc, or they fail and leave the softc unchanged. They do not half-succeed.

25.6.9 Detach as the Mirror of Attach

The detach routine is the cleanup chain of a successful attach. It has to release exactly the resources that attach acquired, in the reverse order. The shape of the detach function is the shape of the cleanup chain with the labels removed and the acquisitions deleted:

```
static int
myfirst_detach(device_t dev)
{
    struct myfirst_softc *sc = device_get_softc(dev);

    /* Check for busy first. */
    mtx_lock(&sc->sc_mtx);
    if (sc->sc_open_count > 0) {
        mtx_unlock(&sc->sc_mtx);
        return (EBUSY);
    }
    mtx_unlock(&sc->sc_mtx);

    /* Release resources in the reverse order of attach. */
    myfirst_log_detach(sc);
    destroy_dev(sc->sc_cdev);
    /* Sysctl is cleaned up by Newbus after detach returns. */
    mtx_destroy(&sc->sc_mtx);

    return (0);
}
```

Read side by side with the attach function, the correspondence is exact. Every resource named in attach has a release in detach. Every new acquisition added to attach has a matching addition in detach. A reviewer auditing a patch that adds a new resource to the driver should be able to find both additions in the diff, one in the attach chain and one in the detach chain; a diff that adds only to attach is incomplete.

A useful discipline when touching the attach chain is to open the detach function in an adjacent editor buffer and add the release immediately after adding the acquisition. This is the simplest way to make sure the two functions stay in sync: they are edited together as a single operation.

25.6.10 Deliberate Failure Injection for Testing

A cleanup chain is correct only if every label is reachable. The only way to be sure is to fire each failure path on purpose and observe that the driver unloads cleanly afterwards. Waiting for real hardware failures to exercise

the paths is not a strategy: most paths are never exercised in real life.

The tool for this kind of testing is deliberate failure injection. The developer adds a temporary `goto` or a temporary early return in the middle of the attach chain and confirms that the driver's resources are all released when the injected failure fires.

A minimal pattern for `myfirst`:

```
#ifdef MYFIRST_DEBUG_INJECT_FAIL_CDEV
    error = ENOMEM;
    goto fail_cdev;
#endif
```

Compile the driver with `-DMYFIRST_DEBUG_INJECT_FAIL_CDEV` and load it. Attach returns `ENOMEM`. `kldstat` shows no residue. `dmesg` shows the attach failure and no kernel complaints about leaked locks or leaked resources. Unload the module, remove the define, recompile, and the driver is back to normal.

Do this once per label, in turn:

1. Inject a failure just after the lock is initialised. Confirm that only the lock is released.
2. Inject a failure just after the cdev is created. Confirm that the cdev and lock are released.
3. Inject a failure just after the sysctl tree is built. Confirm that the cdev and lock are released, and the sysctl OIDs disappear.
4. Inject a failure just after the log state is initialised. Confirm that every resource acquired to that point is released.

If any injection leaves a residue, the cleanup chain has a bug. Fix the bug, rerun the injection, and move on.

This is uncomfortable work the first time, and reassuring afterwards. A driver whose every failure path has been exercised once is a driver whose failure paths will keep working as the code evolves. A driver whose failure paths have never been exercised is a driver with latent bugs that will appear at the worst possible moment.

The companion example `ex05-failure-injection/` under `examples/part-05/ch25-advanced/` contains a version of `myfirst_attach` with every failure injection site marked by a commented `#define`. The lab at the end of the chapter walks through each injection in turn.

25.6.11 A Complete `myfirst_attach` for Chapter 25

Putting all of Section 5 together with the Chapter 25 additions (log state, tunable fetches, capability bitmask), the final attach function looks like this:

```
static int
myfirst_attach(device_t dev)
{
    struct myfirst_softc *sc = device_get_softc(dev);
    struct make_dev_args args;
    int error;
```



```
/*
 * Stage 1: softc basics. Cannot fail. Recorded for consistency;
 * no cleanup label is needed because no resource is held yet.
 */
sc->sc_dev = dev;

/*
 * Stage 2: lock. Cannot fail on MTX_DEF, but needs a label
 * because anything below this line can fail and must release it.
 */
mtx_init(&sc->sc_mtx, "myfirst", NULL, MTX_DEF);

/*
 * Stage 3: pre-populate the softc with defaults, then allow
 * boot-time tunables to override. No allocations here, so no
 * cleanup is needed. Defaults come from the Section 3 tunable
 * set.
 */
strcpy(sc->sc_msg, "Hello from myfirst", sizeof(sc->sc_msg));
sc->sc_msglen = strlen(sc->sc_msg);
sc->sc_open_count = 0;
sc->sc_total_reads = 0;
sc->sc_total_writes = 0;
sc->sc_debug = 0;
sc->sc_timeout_sec = 5;
sc->sc_max_retries = 3;
sc->sc_log_pps = MYF_RL_DEFAULT_PPS;

TUNABLE_INT_FETCH("hw.myfirst.debug_mask_default",
    &sc->sc_debug);
TUNABLE_INT_FETCH("hw.myfirst.timeout_sec",
    &sc->sc_timeout_sec);
TUNABLE_INT_FETCH("hw.myfirst.max_retries",
    &sc->sc_max_retries);
TUNABLE_INT_FETCH("hw.myfirst.log_ratelimit_pps",
    &sc->sc_log_pps);

/*
 * Stage 4: cdev. Can fail. On failure, release the lock and
 * return the error from make_dev_s.
 */
make_dev_args_init(&args);
args.mda_devsw = &myfirst_cdevsw;
args.mda_uid = UID_ROOT;
```

```

args.mda_gid      = GID_WHEEL;
args.mda_mode     = 0660;
args.mda_si_drv1  = sc;
args.mda_unit     = device_get_unit(dev);

error = make_dev_s(&args, &sc->sc_cdev, "myfirst%d",
    device_get_unit(dev));
if (error != 0)
    goto fail_mtx;

/*
 * Stage 5: sysctl tree. Cannot fail. The framework owns the
 * context, so no cleanup label is required specifically for it.
 */
myfirst_sysctl_attach(sc);

/*
 * Stage 6: rate-limit and counter state. Can fail if memory
 * allocation fails. On failure, release the cdev and the lock.
 */
error = myfirst_log_attach(sc);
if (error != 0)
    goto fail_cdev;

DPRINTF(sc, MYF_DBG_INIT,
    "attach: version 1.8-maintenance complete\n");
return (0);

fail_cdev:
    destroy_dev(sc->sc_cdev);
fail_mtx:
    mtx_destroy(&sc->sc_mtx);
    return (error);
}

```

Every resource is accounted for. Every failure path is linear. The function has a single success return at the transition from acquisitions to cleanup, and a single failure return at the bottom of the cleanup chain. Adding a seventh resource next chapter is a three-line operation: one new acquisition block, one new label, one new cleanup line.

25.6.12 Common Mistakes in Failure Paths

A few failure-path mistakes are worth naming once so they can be recognised when they appear in someone else's code or in a review.

The first mistake is **a missing label**. A developer adds a new resource acquisition but forgets to add its cleanup label. The compiler does not warn; the chain looks fine from the outside; but on failure after the new acquisition, the cleanup of everything below is skipped. The rule is that every acquisition has a label. Even if the acquisition cannot fail, it still needs a label so later acquisitions can reach it.

The second mistake is **freeing a resource twice**. A developer adds a local cleanup inside a helper and forgets that the caller's cleanup label also frees the resource. The helper frees once, the caller frees again, and either the kernel panics (for memory) or the witness code complains (for locks). The rule is that only one party owns each resource's cleanup. If the helper acquires the resource and stores it into the softc, the helper does not clean it up on the caller's behalf; it either succeeds or leaves the softc untouched.

The third mistake is **relying on NULL tests**. A developer writes a cleanup chain like this:

```
fail_cdev:
    if (sc->sc_cdev != NULL)
        destroy_dev(sc->sc_cdev);
fail_mtx:
    if (sc->sc_mtx_initialised)
        mtx_destroy(&sc->sc_mtx);
```

The logic is: skip cleanup if the resource was not actually acquired. The intent is defensive; the effect is to hide bugs. If the NULL check is there because the cleanup might be reached in a state where the resource is not acquired, the chain is wrong: the goto target should be a different label. The correct behaviour is to make the cleanup label unreachable unless the resource was actually acquired. Labels that can be reached in either state are a symptom of a muddled acquisition order, and the NULL check only papers over it.

The fourth mistake is **using goto for non-error flow**. goto in the attach function is strictly for failure paths. A goto that skips a section of the acquisition chain on some non-error condition is a violation of the linear-cleanup invariant: the cleanup chain assumes that every label corresponds to a resource that has been acquired, and a goto that bypasses an acquisition breaks that assumption. If conditional acquisition is needed, use an if around the acquisition itself, not a goto around it.

25.6.13 Wrapping Up Section 5

Attach and detach are the seams that hold a driver to the kernel. A correct attach is a linear stack of acquisitions; a correct detach is the stack popped in reverse. The labelled-goto pattern is how FreeBSD drivers encode that stack in C without buying into the kernels of other operating systems (C++ destructors, Go defer, Rust Drop). It is unglamorous and it scales: a driver with a dozen resources reads exactly like a driver with two, and the rules for adding a new resource are always the same.

The myfirst attach function now has four failure labels and a clean separation of acquisition, success return, and cleanup. Every new resource that Chapter 26 adds will fit into this shape.

In the next section, we step back from any single function and look at how a growing driver is spread across files. One myfirst.c with every function in it has carried us for eight chapters; it is time to split it into focused units so that the structure of the driver is visible at the file level.

25.7 Section 6: Modularization and Separation of Concerns

By the end of Chapter 24 the `myfirst` driver has grown beyond what one source file can comfortably hold. The file shape was `myfirst.c` plus `myfirst_debug.c`, `myfirst_ioctl.c`, and `myfirst_sysctl.c`; `myfirst.c` still carried the `cdevsw`, the read/write callbacks, the open/close callbacks, the attach and detach routines, and the module glue. That was fine for teaching, because every addition landed in a file small enough for a reader to hold in their head. It is no longer fine for a driver that has an `ioctl` surface, a `sysctl` tree, a debug framework, a rate-limited logging helper, a capability bitmask, a versioning discipline, and a labelled-cleanup attach routine. A file with that much in it becomes painful to read, painful to diff, and painful to hand to a new contributor.

Section 6 is about the other direction. It does not introduce new behaviour; every function that exists at the end of Section 5 is still here at the end of Section 6. What changes is the file layout and the boundary lines between pieces. The goal is a driver whose structure you can understand from `ls`, and whose individual files each answer a single question.

25.7.1 Why Split Files

The temptation with a self-contained driver is to keep everything in one file. A single `myfirst.c` is easy to locate, easy to `grep`, easy to copy into a tarball. Splitting feels like bureaucracy. The argument for splitting appears when the driver crosses one of three thresholds.

The first threshold is **comprehension**. A reader who opens `myfirst.c` should be able to find what they are looking for in a few seconds. A 1200-line file with eight unrelated responsibilities is hard to navigate; the reader has to scroll past the `cdevsw` to find the `sysctl`, past the `sysctl` to find the `ioctl`, past the `ioctl` to find the attach routine. Each time they switch subject, they have to reload their mental context. With separate files, the subject is the filename: `myfirst_ioctl.c` is about `ioctls`, `myfirst_sysctl.c` is about `sysctls`, `myfirst.c` is about lifecycle.

The second threshold is **independence**. Two unrelated changes should not modify the same file. When a developer adds a `sysctl` and another developer adds an `ioctl`, their patches should not compete for the same lines of `myfirst.c`. Small, focused files let two changes land in parallel with no merge conflict and no risk that a bug in one change accidentally touches the other.

The third threshold is **testability and reuse**. A driver's logging infrastructure, its `ioctl` dispatch, and its `sysctl` tree are often useful to more than one driver inside the same project. Keeping them in separate files with clean interfaces makes them candidates for sharing later. A driver that lives in a single file cannot easily share anything; extraction means copying and manually renaming, which is an error-prone operation.

`myfirst` at the end of Chapter 25 has crossed all three thresholds. Splitting the file is the maintenance act that keeps the driver healthy for the next ten chapters.

25.7.2 A File Layout for `myfirst`

The proposed layout is the one the Makefile in the final Chapter 25 examples directory uses:

<code>myfirst.h</code>	- public types and constants (<code>softc</code> , <code>SRB</code> , status bits).
<code>myfirst.c</code>	- module glue, <code>cdevsw</code> , <code>devclass</code> , module events.

<code>myfirst_bus.c</code>	- Newbus methods and <code>device_identify</code> .
<code>myfirst_cdev.c</code>	- open/close/read/write callbacks; no <code>ioctl</code> .
<code>myfirst_ioctl.h</code>	- <code>ioctl</code> command numbers and payload structures.
<code>myfirst_ioctl.c</code>	- <code>myfirst_ioctl</code> switch and helpers.
<code>myfirst_sysctl.c</code>	- <code>myfirst_sysctl_attach</code> and handlers.
<code>myfirst_debug.h</code>	- <code>DPRINTF/DLOG/DLOG_RL</code> macros and class bits.
<code>myfirst_debug.c</code>	- debug-class enumeration (if any out-of-line).
<code>myfirst_log.h</code>	- rate-limit state structure.
<code>myfirst_log.c</code>	- <code>myfirst_log_attach/detach</code> and helpers.

Seven `.c` files and four `.h` files. Each `.c` file has a subject matter named by its filename. The headers declare the interfaces that cross file boundaries. No file imports internals of another file; every cross-file reference goes through a header.

At first sight this looks like more files than the driver needs. It is not. Each file has a specific responsibility, and the header that goes with it is one to three dozen lines of declarations. The cumulative size is the same as the single-file version; the structure is dramatically clearer.

25.7.3 The Single-Responsibility Rule

The rule that governs the split is a single-responsibility rule: each file answers one question about the driver.

- `myfirst.c` answers: how does this module attach to the kernel and wire its pieces together?
- `myfirst_bus.c` answers: how does Newbus discover and instantiate my driver?
- `myfirst_cdev.c` answers: how does the driver serve open/close/read/write?
- `myfirst_ioctl.c` answers: how does the driver handle the commands its header declares?
- `myfirst_sysctl.c` answers: how does the driver expose its state to `sysctl(8)`?
- `myfirst_debug.c` answers: how are debug messages classified and rate-limited?
- `myfirst_log.c` answers: how is rate-limit state initialised and released?

The test for whether a change belongs in a given file is the answer test. If the change does not answer the file's question, it belongs elsewhere. A new `sysctl` does not belong in `myfirst_ioctl.c`; a new `ioctl` does not belong in `myfirst_sysctl.c`; a new read callback variant does not belong in `myfirst.c`. The rule is explicit, and a reviewer applying it rejects patches that put things in the wrong file.

Applying the rule to the existing Chapter 24 shape gives the Chapter 25 shape.

25.7.4 Public vs Private Headers

Headers carry the interface between files. A driver that splits its `.c` files has to decide, for each declaration, whether it belongs in a public header or a private one.

Public headers contain types and constants that are visible to more than one `.c` file. `myfirst.h` is the main public header for the driver. It declares:

- The `struct myfirst_softc` definition (every `.c` file needs it).
- Constants that appear in more than one file (debug-class bits, `softc` field sizes).

- Prototypes for functions that are called across file boundaries (`myfirst_sysctl_attach`, `myfirst_log_attach`, `myfirst_log_ratelimited_printf`, `myfirst_ioctl`).

Private headers carry declarations that are only needed by one `.c` file. `myfirst_ioctl.h` is the canonical example. It declares the command numbers and payload structures; they are needed by `myfirst_ioctl.c` and by user-space callers, but no other in-kernel file needs them. Putting them in `myfirst.h` would leak the wire format into every translation unit.

The distinction matters because every public declaration is a contract the driver has to honour. A type in `myfirst.h` that changes size breaks every file that includes `myfirst.h`. A type in `myfirst_ioctl.h` that changes size breaks only `myfirst_ioctl.c` and the user-space tools that compiled against it.

For `myfirst` at the end of Chapter 25, the public header `myfirst.h` looks like this (trimmed to the declarations relevant to this section):

```
/*
 * myfirst.h - public types and constants for the myfirst driver.
 *
 * Types and prototypes declared here are visible to every .c file in
 * the driver. Keep this header small. Wire-format declarations live
 * in myfirst_ioctl.h. Debug macros live in myfirst_debug.h. Rate-
 * limit state lives in myfirst_log.h.
 */

#ifndef _MYFIRST_H_
#define _MYFIRST_H_

#include <sys/types.h>
#include <sys/lock.h>
#include <sys/mutex.h>
#include <sys/conf.h>

#include "myfirst_log.h"

struct myfirst_softc {
    device_t      sc_dev;
    struct mtx     sc_mtx;
    struct cdev    *sc_cdev;

    char          sc_msg[MYFIRST_MSG_MAX];
    size_t        sc_msglen;

    u_int         sc_open_count;
    u_int         sc_total_reads;
    u_int         sc_total_writes;

    u_int         sc_debug;
};
```

```

    u_int          sc_timeout_sec;
    u_int          sc_max_retries;
    u_int          sc_log_pps;

    struct myfirst_ratelimit sc_rl_generic;
    struct myfirst_ratelimit sc_rl_io;
    struct myfirst_ratelimit sc_rl_intr;
};

#define MYFIRST_MSG_MAX 256

/* Sysctl tree. */
void myfirst_sysctl_attach(struct myfirst_softc *);

/* Rate-limit state. */
int myfirst_log_attach(struct myfirst_softc *);
void myfirst_log_detach(struct myfirst_softc *);

/* Ioctl dispatch. */
struct thread;
int myfirst_ioctl(struct cdev *, u_long, caddr_t, int, struct thread *);

#endif /* _MYFIRST_H */

```

Nothing in `myfirst.h` references a wire-format constant, a debug-class bit, or a rate-limit structure internals. The `softc` includes three rate-limit fields by value, so `myfirst.h` has to include `myfirst_log.h`, but the internals of `struct myfirst_ratelimit` live in `myfirst_log.h` and are not exposed here.

25.7.5 The Anatomy of `myfirst.c` After the Split

`myfirst.c` after the split is the shortest `.c` file in the driver. It contains the `cdevsw` table, the module event handler, the device class declaration, and the attach/detach routines. Every other responsibility has moved elsewhere:

```

/*
 * myfirst.c - module glue and cdev wiring for the myfirst driver.
 *
 * This file owns the cdevsw table, the devclass, the attach and
 * detach routines, and the MODULE_VERSION declaration. The cdev
 * callbacks themselves live in myfirst_cdev.c. The ioctl dispatch
 * lives in myfirst_ioctl.c. The sysctl tree lives in
 * myfirst_sysctl.c. The rate-limit infrastructure lives in
 * myfirst_log.c.
 */

```

```
#include <sys/param.h>
#include <sys/system.h>
#include <sys/conf.h>
#include <sys/kernel.h>
#include <sys/lock.h>
#include <sys/module.h>
#include <sys/mutex.h>

#include "myfirst.h"
#include "myfirst_debug.h"
#include "myfirst_ioctl.h"

MODULE_VERSION(myfirst, 18);

extern d_open_t    myfirst_open;
extern d_close_t   myfirst_close;
extern d_read_t    myfirst_read;
extern d_write_t   myfirst_write;

struct cdevsw myfirst_cdevsw = {
    .d_version = D_VERSION,
    .d_name    = "myfirst",
    .d_open    = myfirst_open,
    .d_close   = myfirst_close,
    .d_read    = myfirst_read,
    .d_write   = myfirst_write,
    .d_ioctl   = myfirst_ioctl,
};

static int
myfirst_attach(device_t dev)
{
    /* Section 5's labelled-cleanup attach goes here. */
    ...
}

static int
myfirst_detach(device_t dev)
{
    /* Section 5's mirror-of-attach detach goes here. */
    ...
}
```



```
static device_method_t myfirst_methods[] = {
    DEVMETHOD(device_probe,    myfirst_probe),
    DEVMETHOD(device_attach,   myfirst_attach),
    DEVMETHOD(device_detach,   myfirst_detach),
    DEVMETHOD_END
};

static driver_t myfirst_driver = {
    "myfirst",
    myfirst_methods,
    sizeof(struct myfirst_softc),
};

DRIVER_MODULE(myfirst, nexus, myfirst_driver, 0, 0);
```

The file has one job: wire the driver's pieces together at the kernel level. It is a few hundred lines; every other file in the driver is smaller.

25.7.6 myfirst_cdev.c: The Character Device Callbacks

The open, close, read, and write callbacks were the first code we wrote back in Chapter 18. They have grown since then. Extracting them to myfirst_cdev.c keeps them with each other and out of myfirst.c:

```
/*
 * myfirst_cdev.c - character-device callbacks for the myfirst driver.
 *
 * The open/close/read/write callbacks all operate on the softc that
 * make_dev_s installed as si_drv1. The ioctl dispatch is in
 * myfirst_ioctl.c; this file intentionally does not handle ioctls.
 */

#include <sys/param.h>
#include <sys/system.h>
#include <sys/conf.h>
#include <sys/uio.h>
#include <sys/lock.h>
#include <sys/mutex.h>

#include "myfirst.h"
#include "myfirst_debug.h"

int
myfirst_open(struct cdev *dev, int oflags, int devtype, struct thread *td)
{
```

```

    struct myfirst_softc *sc = dev->si_drv1;

    mtx_lock(&sc->sc_mtx);
    sc->sc_open_count++;
    mtx_unlock(&sc->sc_mtx);

    DPRINTF(sc, MYF_DBG_OPEN, "open: count %u\n", sc->sc_open_count);
    return (0);
}

/* close, read, write follow the same pattern. */

```

Each callback begins with `sc = dev->si_drv1` (the per-cdev pointer that `make_dev_args` set) and operates on the `_softc`. No cross-file coupling beyond the public header.

25.7.7 myfirst_ioctl.c: The Command Switch

`myfirst_ioctl.c` has been in its own file since Chapter 22. The Chapter 25 addition is the `MYFIRSTIOC_GETCAPS` handler:

```

int
myfirst_ioctl(struct cdev *dev, u_long cmd, caddr_t data, int flag,
              struct thread *td)
{
    struct myfirst_softc *sc = dev->si_drv1;
    int error = 0;

    switch (cmd) {
    case MYFIRSTIOC_GETVER:
        *(int *)data = MYFIRST_IOCTL_VERSION;
        break;
    case MYFIRSTIOC_RESET:
        mtx_lock(&sc->sc_mtx);
        sc->sc_total_reads = 0;
        sc->sc_total_writes = 0;
        mtx_unlock(&sc->sc_mtx);
        break;
    case MYFIRSTIOC_GETMSG:
        mtx_lock(&sc->sc_mtx);
        strncpy((char *)data, sc->sc_msg, MYFIRST_MSG_MAX);
        mtx_unlock(&sc->sc_mtx);
        break;
    case MYFIRSTIOC_SETMSG:
        mtx_lock(&sc->sc_mtx);

```

```

        strlcpy(sc->sc_msg, (const char *)data, MYFIRST_MSG_MAX);
        sc->sc_msglen = strlen(sc->sc_msg);
        mtx_unlock(&sc->sc_mtx);
        break;
    case MYFIRSTIOC_GETCAPS:
        *(uint32_t *)data = MYF_CAP_RESET | MYF_CAP_GETMSG |
                            MYF_CAP_SETMSG;

        break;
    default:
        error = ENOIOCTL;
        break;
    }
    return (error);
}

```

The switch is the entire public ioctl surface of the driver. Adding a command means adding a case; retiring one means deleting a case and deprecating the constant in `myfirst_ioctl.h`.

25.7.8 `myfirst_log.h` and `myfirst_log.c`: Rate-Limited Logging

Section 1 introduced the rate-limited logging macro `DLOG_RL` and the struct `myfirst_ratelimit` state it tracks. The rate-limit state was left embedded in the softc in Section 1 because the abstraction had not yet been factored out. Section 6 is the right moment to factor it out: the rate-limit code is small enough to be worth collecting in one place and general enough that other drivers might want it.

`myfirst_log.h` contains the state definition:

```

#ifndef _MYFIRST_LOG_H_
#define _MYFIRST_LOG_H_

#include <sys/time.h>

struct myfirst_ratelimit {
    struct timeval rl_lasttime;
    int           rl_curpps;
};

#define MYF_RL_DEFAULT_PPS 10

#endif /* _MYFIRST_LOG_H_ */

```

`myfirst_log.c` contains the attach and detach helpers:

```
#include <sys/param.h>
#include <sys/system.h>
#include <sys/lock.h>
#include <sys/mutex.h>

#include "myfirst.h"
#include "myfirst_debug.h"

int
myfirst_log_attach(struct myfirst_softc *sc)
{
    /*
     * The rate-limit state is embedded by value in the softc, so
     * there is no allocation to do. This function exists so that
     * the attach chain has a named label for logging in case a
     * future version needs per-class allocations.
     */
    bzero(&sc->sc_rl_generic, sizeof(sc->sc_rl_generic));
    bzero(&sc->sc_rl_io,      sizeof(sc->sc_rl_io));
    bzero(&sc->sc_rl_intr,   sizeof(sc->sc_rl_intr));

    return (0);
}

void
myfirst_log_detach(struct myfirst_softc *sc)
{
    /* Nothing to release; the state is embedded in the softc. */
    (void)sc;
}
```

Today `myfirst_log_attach` does no allocation; it zeroes the rate-limit fields and returns. Tomorrow, if the driver needs a dynamic array of per-class counters, the allocation fits here and the attach chain does not have to change. This is the value of extracting the helper before it is strictly necessary: the shape is ready for growth.

The header file size matters here. `myfirst_log.h` is under 20 lines. A header of 20 lines is cheap to include everywhere, cheap to read, and cheap to keep in sync. If `myfirst_log.h` grew to 200 lines, the cost of including it from every `.c` file would start to show up in compile times and in review friction; at that point the next step is to split it again.

25.7.9 The Updated Makefile

The Makefile for the split driver lists every `.c` file:

```
# Makefile for the myfirst driver - Chapter 25 (1.8-maintenance).
#
# Chapter 25 splits the driver into subject-matter files. Each file
# answers a single question; the Makefile lists them in alphabetical
# order after myfirst.c (which carries the module glue) so the
# reader sees the main file first.

KMOD=    myfirst
SRCS=    myfirst.c myfirst_cdev.c myfirst_debug.c myfirst_ioctl.c \
        myfirst_log.c myfirst_sysctl.c

CFLAGS+=    -I${.CURDIR}

SYSDIR?=    /usr/src/sys

.include <bsd.kmod.mk>
```

SRCS lists six .c files, one per subject. Adding a seventh is a one-line change. The kernel build system picks up every file in SRCS automatically; there is no manual linking step, no makefile dependency tree to maintain.

25.7.10 Where to Draw Each File Boundary

The hardest part of splitting a driver is not deciding to split; it is deciding where the boundaries go. Most splits go through three phases, and the phases apply to any driver, not just myfirst.

Phase one is the flat file. Everything is in driver.c. This is the right shape for the first 300 lines of a driver. Splitting sooner creates more friction than it saves.

Phase two is the subject split. The ioctl dispatch goes into driver_ioctl.c, the sysctl tree goes into driver_sysctl.c, the debug infrastructure goes into driver_debug.c. Each file is named after the subject it handles. This is where myfirst has been since Chapter 24.

Phase three is the subsystem split. As the driver grows, one subject grows larger than a single file. The ioctl file splits into driver_ioctl.c (the dispatch) and driver_ioctl_rw.c (the read/write payload helpers). The sysctl file splits similarly. This is where a full-featured driver ends up, often in its third or fourth major version.

myfirst at the end of Chapter 25 is solidly in phase two. Phase three is not warranted yet, and Chapter 26 will reset the clock when it splits the pseudo-driver into a USB-attached variant and leaves myfirst_core.c as the subject-agnostic core. There is no value in pre-emptively splitting to phase three today.

The rule of thumb for when to move from phase two to phase three is: when a single subject file crosses 1,000 lines, or when two unrelated changes to the same subject file cause a merge conflict, the subject is ready to split.

25.7.11 The Include Graph and Build Ordering

Once a driver is split into several files, the include graph matters. A circular include is not a hard error in C, but it is a sign of a muddled dependency structure that will confuse readers. The right shape is a directed acyclic graph of headers, rooted at `myfirst.h`, with leaf headers like `myfirst_ioctl.h` and `myfirst_log.h`.

`myfirst.h` is the widest header. It declares the `softc` and prototypes that every other file uses. It includes `myfirst_log.h` because the `softc` has rate-limit fields by value.

`myfirst_debug.h` is a leaf. It declares the `DPRINTF` macro family and the class bits. It is included by every `.c` file, directly or indirectly. It is not included by `myfirst.h`, because `myfirst.h` should not force the debug macros into any caller that does not want them.

`myfirst_ioctl.h` is a leaf. It declares command numbers, payload structures, and the wire-format version integer. It is included by `myfirst_ioctl.c` (and its user-space counterpart `myfirstctl.c`).

No header includes a file other than public kernel headers and the driver's own headers. No `.c` file includes another `.c` file. The include graph is shallow and easy to diagram.

25.7.12 The Cost of Splitting

Splitting files has a real cost. Every split adds a header file, and every header file has to be maintained. A function whose signature changes has to be updated in the `.c` and in the `.h`, and the change has to propagate to every other `.c` file that includes the header. A driver with twelve files is slightly slower to compile than a driver with one, because every `.c` has to include several headers and the preprocessor has to parse them.

These costs are real but small. They are much smaller than the cost of a monolithic file that nobody wants to touch. The rule is to split when the cost of not splitting exceeds the cost of maintaining the boundary. For `myfirst` at the end of Chapter 25, the threshold has been crossed.

25.7.13 A Practical Procedure for Splitting a Real Driver

Splitting a file is a routine refactor, but a routine refactor can still introduce bugs if done carelessly. A practical procedure for splitting an in-tree driver is:

1. **Identify the subjects.** Read the monolithic file top to bottom and group its functions by subject (`cdev`, `ioctl`, `sysctl`, `debug`, `lifecycle`). Write the grouping on a piece of paper or in a comment block.
2. **Create the empty files.** Add the new `.c` files and their headers to the source tree. Compile once to make sure the build system sees them.
3. **Move one subject at a time.** Move the `ioctl` functions into `driver_ioctl.c`. Move their declarations into `driver_ioctl.h`. Update `driver.c` to `#include "driver_ioctl.h"`. Compile. Run the driver through its test matrix.
4. **Commit each subject split.** Each subject move is a single commit. The commit log reads: “`myfirst: split ioctl dispatch into driver_ioctl.c`”. A reviewer can see the move clearly; a `git blame` shows the same line in the new file on the same commit.

5. **Verify the include graph.** After all subjects are moved, compile with `-Wunused-variable` and `-Wmissing-prototypes` to catch functions that should have had prototypes but did not. Use `nm` on the built module to confirm that no symbols that should be static are being exported.
6. **Retest.** Run the full driver test matrix. Splitting a file should not change behaviour; if a test starts failing, the split has introduced a bug.

The procedure for `myfirst` at the end of Chapter 25 follows exactly these steps. The final directory under `examples/part-05/ch25-advanced/` is the result.

25.7.14 Common Mistakes When Splitting

A few mistakes are common the first time a driver is split. Watching for them shortens the learning curve.

The first mistake is **putting declarations in the wrong header**. If `myfirst.h` declares a function that is only called from `myfirst_ioctl.c`, every other translation unit is paying to parse a declaration it does not need. If `myfirst_ioctl.h` declares a function that is called from both `myfirst_ioctl.c` and `myfirst_cdev.c`, the two consumers are coupled through the `ioctl` header and any change in the `ioctl` header rebuilds both files. The fix is to put cross-cutting declarations in `myfirst.h` and subject-specific declarations in subject-specific headers.

The second mistake is **forgetting static on functions that should be file-scope**. A function that is used only inside `myfirst_sysctl.c` should be declared `static`. Without `static`, the function is exported from the object file, which means another file could accidentally call it, and any later renaming in the original file becomes an ABI change. A `static` discipline prevents this entire class of problem.

The third mistake is **circular includes**. If `myfirst_ioctl.h` includes `myfirst.h`, and `myfirst.h` includes `myfirst_ioctl.h`, the driver compiles (thanks to include guards) but the dependency graph is wrong. Every edit to either file now triggers a rebuild of everything that includes either one. The fix is to decide which header sits higher in the graph and remove the back-reference.

The fourth mistake is **re-introducing a subject into the wrong file**. Six months after the split, someone adds a new `ioctl` by editing `myfirst.c` because that was where `ioctls` used to live. The single-responsibility rule has to be enforced by reviewers. A patch that puts a new `ioctl` in `myfirst.c` is rejected with a comment pointing at `myfirst_ioctl.c`.

25.7.15 Wrapping Up Section 6

A driver whose files each answer one question is a driver that you can hand to a new contributor on their first day. They read the filenames, they pick a subject, and they start editing exactly one file. The `myfirst` driver has crossed that threshold. Six `.c` files plus their headers hold every function the driver has grown since Chapter 17, with each file named for what it does.

In the next section, we turn from internal organisation to external preparation. A driver that is ready for production has a short list of properties that it must satisfy before it can be installed on a machine the developer does not own. Chapter 25's production-readiness checklist names those properties and walks `myfirst` through each one.

25.8 Section 7: Preparing for Production Use

A driver that works on your workstation is not a driver that is ready for production. Production is the set of conditions your code faces when it is installed on hardware you do not own, booted by operators you will never meet, and expected to behave predictably for months or years between reboots. The distance between “it works for me” and “it is ready to ship” is measured in habits, not in features. This section names the habits.

`myfirst` in its Chapter 25 shape is as feature-complete as the pseudo-driver is going to get. The remaining work is not to add functionality, but to harden the edges so the driver survives the environments it cannot control.

25.8.1 The Production Readiness Mindset

The mindset shift is this: every decision the driver makes implicitly at development time has to be made explicitly at production time. Where a tunable has a default, the default has to be the right default. Where a `sysctl` is writable, the consequences of a write by a scared operator at 3 a.m. have to be safe. Where a log message might fire, the message has to be useful without help from the developer. Where a module depends on another module, the dependency has to be declared so the loader does not load them in the wrong order.

Production-readiness is not a one-shot action; it is an attitude that runs through every decision. A driver that is nearly ready for production often has one or two specific gaps: a tunable with no documentation, a log message that fires every microsecond, a detach path that assumes nobody is using the device. The discipline of production-readiness is to find those specific gaps and close them, one by one, until the driver’s behaviour is predictable on a machine the developer is not standing in front of.

25.8.2 Declaring Module Dependencies

The first production habit is to be explicit about what the module needs. If `myfirst` calls a function that lives in another kernel module, the kernel’s module loader needs to know about the dependency before the call, or the kernel loads `myfirst` and panics the first time the dependency is used.

The mechanism is `MODULE_DEPEND`. Section 4 introduced it as a compatibility tool; in production, it is also a correctness tool. A driver without `MODULE_DEPEND` on its real dependencies works by accident in most boot orderings and fails mysteriously in others. A driver with `MODULE_DEPEND` on every real dependency either loads correctly or refuses to load with a clear error message.

For the pseudo-driver `myfirst`, there are no real dependencies yet; the driver only uses symbols from the kernel core, which is always present. The Chapter 26 USB variant will add the first real `MODULE_DEPEND`:

```
MODULE_DEPEND(myfirst_usb, usb, 1, 1, 1);
```

The three version numbers are the minimum, preferred, and maximum of the USB stack version that `myfirst_usb` is compatible with. At load time, the kernel checks the installed USB stack version against this range and refuses to load `myfirst_usb` if the USB stack is missing or outside the range.

The production habit is: before shipping, `grep` the driver for every symbol it calls and confirm that every symbol lives either in the kernel core or in a module that the driver declares a dependency on. A missing `MODULE_DEPEND` works until the boot ordering changes, and then the driver panics on production hardware.

25.8.3 Publishing PNP Information

For hardware drivers, the kernel's module loader consults each module's PNP metadata to decide which driver handles which device. A USB driver that does not publish PNP information works when loaded manually and fails when the boot loader tries to auto-load the driver for a newly-plugged device. The fix is `MODULE_PNP_INFO`, which the driver uses to declare the vendor/product identifiers it handles:

```
MODULE_PNP_INFO("U16:vendor;U16:product", uhub, myfirst_usb,  
    myfirst_pnp_table, nitems(myfirst_pnp_table));
```

The first string describes the format of the PNP table entries. `uhub` is the bus name; `myfirst_usb` is the driver name; `myfirst_pnp_table` is a static array of structures, one per device the driver handles.

`myfirst` at Chapter 25 is still a pseudo-driver and has no hardware to match. `MODULE_PNP_INFO` comes into play in Chapter 26 with the first real hardware attach. For Chapter 25, the production habit is simply to know the macro exists and to plan for it when hardware arrives.

25.8.4 The `MOD_QUIESCE` Event

The kernel module event handler is called with one of four events: `MOD_LOAD`, `MOD_UNLOAD`, `MOD_SHUTDOWN`, `MOD_QUIESCE`. Most drivers handle `MOD_LOAD` and `MOD_UNLOAD` explicitly, and the kernel synthesises the other two. For production drivers, `MOD_QUIESCE` deserves attention.

`MOD_QUIESCE` is the kernel's question "can you be unloaded right now?" It fires before `MOD_UNLOAD` and gives the driver a chance to refuse cleanly. A driver that is mid-operation (an outstanding DMA transfer, an open file descriptor, a pending timer) can return a non-zero `errno` from `MOD_QUIESCE` to refuse the unload; the kernel then does not proceed to `MOD_UNLOAD`.

For `myfirst`, the quiesce check is already built into `myfirst_detach`: if `sc_open_count > 0`, `detach` returns `EBUSY`. The kernel's module loader propagates that `EBUSY` back to `kldunload(8)`, and the operator sees "module `myfirst` is busy". The check is in the right place, but the discipline of thinking about `MOD_QUIESCE` separately from `MOD_UNLOAD` is worth naming: `MOD_QUIESCE` is the "are you safe to unload?" question, and `MOD_UNLOAD` is the "go ahead and unload" command. Some drivers have state that is safe to check in `MOD_QUIESCE` but not safe to acquire in `MOD_UNLOAD`; splitting them lets the driver answer the question without side effects.

25.8.5 Emitting `devctl_notify` Events

Long-running production systems are monitored by daemons like `devd(8)`, which watch for device arrivals, departures, and state changes. The mechanism the kernel uses to notify `devd` is `devctl_notify(9)`: a driver emits a structured event, `devd` reads the event, and `devd` takes a configured action (run a script, log a message, notify an operator).

The prototype is:

```
void devctl_notify(const char *system, const char *subsystem,  
    const char *type, const char *data);
```

- `system` is a top-level category like "DEVFS", "ACPI", or a driver-specific tag.
- `subsystem` is the driver or subsystem name.
- `type` is a short event name.
- `data` is optional structured data (key=value pairs) for the daemon to parse.

For `myfirst`, a useful production event is “the in-driver message was rewritten”:

```
devctl_notify("myfirst", device_get_nameunit(sc->sc_dev),
    "MSG_CHANGED", NULL);
```

After the operator writes a new message through `ioctl(fd, MYFIRSTIOC_SETMSG, buf)`, the driver emits a `MSG_CHANGED` event. A `devd` rule can match the event and, for example, send a syslog entry or notify a monitoring daemon:

```
notify 0 {
    match "system"      "myfirst";
    match "type"        "MSG_CHANGED";
    action "logger -t myfirst 'message changed on $subsystem'";
};
```

The production habit here is to ask, for each interesting event in the driver, whether an operator might want to react to it. If the answer is yes, emit a `devctl_notify` with a well-chosen name. Downstream tooling can then build on the event, and the driver does not have to know what those tools are.

25.8.6 Writing a `MAINTENANCE.md`

Every production driver should have a maintenance file that describes, in plain English, what the driver does, what tunables it accepts, what `sysctls` it exposes, what `ioctls` it handles, what events it emits, and what the version history is. The file lives alongside the source code in the repository; it is read by operators, by new developers, by security reviewers, and by the author six months later.

A concrete skeleton for `MAINTENANCE.md`:

```
# myfirst

A demonstration character driver that carries the book's running
example. This file is the operator-facing reference.

## Overview

myfirst registers a pseudo-device at /dev/myfirst0 and serves a
read-write message buffer, a set of ioctls, a sysctl tree, and a
configurable debug-class logger.

## Tunables
```

- hw.myfirst.debug_mask_default (int, default 0)
Initial value of dev.myfirst.<unit>.debug.mask.
- hw.myfirst.timeout_sec (int, default 5)
Initial value of dev.myfirst.<unit>.timeout_sec.
- hw.myfirst.max_retries (int, default 3)
Initial value of dev.myfirst.<unit>.max_retries.
- hw.myfirst.log_ratelimit_pps (int, default 10)
Initial rate-limit ceiling (prints per second per class).

Sysctls

All sysctls live under dev.myfirst.<unit>.

Read-only: version, open_count, total_reads, total_writes,
message, message_len.

Read-write: debug.mask (mirror of debug_mask_default), timeout_sec,
max_retries, log_ratelimit_pps.

Ioctls

Defined in myfirst_ioctl.h. Command magic 'M'.

- MYFIRSTIOC_GETVER (0): returns MYFIRST_IOCTL_VERSION.
- MYFIRSTIOC_RESET (1): zeros read/write counters.
- MYFIRSTIOC_GETMSG (2): reads the in-driver message.
- MYFIRSTIOC_SETMSG (3): writes the in-driver message.
- MYFIRSTIOC_GETCAPS (5): returns MYF_CAP_* bitmask.

Command 4 was reserved during Chapter 23 draft work and retired
before release. Do not reuse the number.

Events

Emitted through devctl_notify(9).

- system=myfirst subsystem=<unit> type=MSG_CHANGED
The operator-visible message was rewritten.

Version History

See Change Log below.

Change Log

1.8-maintenance

- Added MYFIRSTIOC_GETCAPS (command 5).
- Added tunables for timeout_sec, max_retries, log_ratelimit_pps.
- Added rate-limited logging via ppsratecheck(9).
- Added devctl_notify for MSG_CHANGED.
- No breaking changes from 1.7.

1.7-integration

- First end-to-end integration of ioctl, sysctl, debug.
- Introduced MYFIRSTIOC_{GETVER,RESET,GETMSG,SETMSG}.

1.6-debug

- Added DPRINTF framework and SDT probes.

The file is nothing glamorous. It is a reference that is kept up to date with every version bump and that serves as the single source of truth for operators.

The production habit is: every change to the driver's visible surface (a new tunable, a new sysctl, a new ioctl, a new event, a behaviour change) has a corresponding entry in MAINTENANCE.md. The file never falls behind the code. A driver whose MAINTENANCE.md is out of date is a driver whose users are guessing; a driver whose MAINTENANCE.md is current is a driver whose users can self-serve.

25.8.7 A devd Rule Set

devd(8) rules tell the daemon how to react to kernel events. For a production deployment of myfirst, a minimal rule set would ensure that important events reach the operator:

```
# /etc/devd/myfirst.conf
#
# devd rules for the myfirst driver. Drop this file into
# /etc/devd/ and restart devd(8) for the rules to take effect.

notify 0 {
    match "system"      "myfirst";
    match "type"        "MSG_CHANGED";
    action "logger -t myfirst 'message changed on $subsystem'";
};

# Future: match attach/detach events once Chapter 26's USB variant
# starts emitting them.
```

The file is short. It declares one rule, matches a specific event, and takes a specific action. In production, such

files grow to match more events, trigger more actions, and in some deployments notify a monitoring system that watches for driver anomalies.

Including a draft `devd.conf` in the driver's repository makes it easy for an operator to adopt. They copy the file, adjust the actions, and the driver's events are integrated into the site's monitoring on the first day.

25.8.8 Logs: Friend of the Support Engineer

A production driver's log messages are read by support engineers who do not have access to the source code and cannot reproduce the problem on demand. The rules that make a log message useful to a support engineer are different from the rules that make a log message useful to a developer.

A developer reading their own log message can rely on context that a support engineer does not have. The support engineer cannot ask "which attach?" or "which device?" or "what was error when this fired?" The answer has to already be in the message.

The production habit is to audit every log message in the driver and ask three questions:

1. **Does the message name its device?** `device_printf(dev, ...)` prefixes the output with the device nameunit; bare `printf` does not. Every message that is not from `MOD_LOAD` (where there is no device yet) should be `device_printf`.
2. **Does the message include the relevant numeric context?** "Failed to allocate" is not useful. "Failed to allocate: error 12 (ENOMEM)" is. "Failed to allocate a timer: error 12" is better.
3. **Does the message appear at the appropriate rate?** Section 1 covered rate-limiting. The final pass is to ensure that every message that can fire in a loop is either rate-limited or is demonstrably one-shot.

A log message that satisfies these three questions reaches a support engineer with enough information to file a useful bug report. A log message that fails any of them wastes the operator's time and the developer's.

25.8.9 Handling Bus Attach/Detach Gracefully

Production drivers, especially hot-pluggable ones, have to handle repeated attach and detach cycles without leaks. The discipline of Section 5's labelled-cleanup pattern is part of the answer; the other part is to confirm that repeated attach/detach actually works. The lab at the end of this chapter walks through a regression script that loads, unloads, and reloads the driver 100 times in a row and verifies that the module's memory footprint does not grow.

A driver that passes the 100-cycle test is a driver that will survive a month of hot-plug events on production hardware. A driver that fails the 100-cycle test has a leak that will manifest, over time, as slow memory growth or as the kernel running out of some bounded resource (`sysctl` OIDs, `cdev` minor numbers, `devclass` entries).

The test is simple to run and is disproportionately valuable. Make it part of the driver's pre-release checklist.

25.8.10 Handling Unexpected Operator Actions

Operators make mistakes. They run `kldunload myfirst` while a test program is reading from `/dev/myfirst0`. They set `dev.myfirst.0.debug.mask` to a value that enables every class at once. They copy `MAINTENANCE.md`

and skip the section on tunables. The production driver has to tolerate these actions without crashing, corrupting state, or leaving the system in a broken configuration.

For each exposed interface, the production habit is to ask: what is the worst sequence of operator actions I can imagine, and does the driver survive it?

- `kldunload` while a file descriptor is open: `myfirst_detach` returns `EBUSY`. Operator sees “module busy”. Driver is unchanged.
- A writable `sysctl` being set to an out-of-range value: the `sysctl` handler clamps the value or returns `EINVAL`. Driver’s internal state is unchanged.
- A `MYFIRSTIOC_SETMSG` with a message longer than the buffer: `strncpy` truncates. The copy is correct; the truncation is visible in `message_len`.
- A concurrent pair of `MYFIRSTIOC_SETMSG` calls: the softc mutex serialises them. Whichever runs second wins; both succeed.

If any of these actions produces a crash, a corruption, or an inconsistent state, the driver is not production-ready. The fix is always the same: add the missing guard, restart the test, and add a comment documenting the invariant.

25.8.11 A Production-Readiness Checklist

The habits in this section fit into a short checklist that a developer can walk through before shipping:

`myfirst` production readiness

- [] `MODULE_DEPEND` declared for every real dependency.
- [] `MODULE_PNP_INFO` declared if the driver binds to hardware.
- [] `MOD_QUIESCE` answers “can you unload?” without side effects.
- [] `devctl_notify` emitted for operator-relevant events.
- [] `MAINTENANCE.md` current: tunables, sysctls, ioctls, events.
- [] `devd.conf` snippet included with the driver.
- [] Every log message is `device_printf`, includes `errno`, and is rate-limited if it can fire in a loop.
- [] `attach/detach` survives 100 load/unload cycles.
- [] `sysctls` reject out-of-range values.
- [] `ioctl` payload is bounds-checked.
- [] Failure paths exercised via deliberate injection.
- [] Versioning discipline: `MYFIRST_VERSION`, `MODULE_VERSION`, `MYFIRST_IOCTL_VERSION` each bumped for their own reason.

The list is short on purpose. Twelve items, most of them already addressed by the habits introduced in earlier sections. A driver that checks every box is ready to be installed by someone who will never meet you.

25.8.12 What the myfirst Driver Covers

Running `myfirst` through the checklist at the end of Chapter 25 gives the following status.

`MODULE_DEPEND` is not required because the driver has no subsystem dependencies; this is noted explicitly in `MAINTENANCE.md`.

`MODULE_PNP_INFO` is not required because the driver does not bind to hardware; this is also noted in `MAINTENANCE.md`.

`MOD_QUIESCE` is answered by the `sc_open_count` check in `myfirst_detach`; a dedicated `MOD_QUIESCE` handler is not added for this version because the semantics are identical.

`devctl_notify` is emitted on `MYFIRSTIOC_SETMSG` with event type `MSG_CHANGED`.

`MAINTENANCE.md` is shipped in the examples directory and contains tunables, sysctls, ioctls, events, and a Change Log entry for 1.8-maintenance.

The `devd.conf` snippet is shipped alongside `MAINTENANCE.md` and demonstrates the single `MSG_CHANGED` rule.

Every log message is emitted through `device_printf` (or `DPRINTF`, which wraps `device_printf`); every message that fires in a hot path is wrapped in `DLOG_RL`.

The attach/detach regression script (see Labs) runs 100 cycles without growing the kernel's memory footprint.

The sysctls for `timeout_sec`, `max_retries`, and `log_ratelimit_pps` each reject out-of-range values in their handlers.

The ioctl payloads are bounds-checked at the structure level by the kernel's ioctl framework (`_IOR`, `_IOW`, `_IOWR` declare exact sizes) and inside the driver where string length matters.

Failure injection points are marked by conditional `#ifdef` in the examples; every label has been reached at least once in development.

Version identifiers each have their own rule: string bumped, module integer bumped, ioctl integer unchanged because additions are backwards compatible.

Twelve checks, twelve outcomes. The driver is ready for the next chapter.

25.8.13 Wrapping Up Section 7

Production is the quiet standard that separates interesting code from shippable code. The disciplines named here are not glamorous; they are the specific things that keep a driver working when it is deployed far from the developer who wrote it. `myfirst` has grown through five chapters of instructional content and now wears the harness that lets it survive outside the book.

In the next section, we turn to the two kernel infrastructures that let a driver run code at specific lifecycle points without manual wiring: `SYSINIT(9)` for boot-time initialisation and `EVENTHANDLER(9)` for runtime notifications. These are the last two pieces of the FreeBSD toolkit the book will introduce before Chapter 26 applies everything to a real bus.

25.9 Section 8: SYSINIT, SYSUNINIT, and EVENTHANDLER

A driver's attach and detach routines handle everything that happens between instantiation and teardown, but there are things a driver may need to do that fall outside that window. Some code has to run before any device is instantiated at all: loading boot-time tunables, initialising a subsystem-wide lock, setting up a pool that the first attach will consume. Other code has to run in response to system-wide events that are not device-specific: a system-wide suspend, a low-memory condition, a shutdown that is about to sync filesystems and power down.

The FreeBSD kernel provides two mechanisms for these cases. `SYSINIT(9)` registers a function to run at a specific boot stage, and its companion `SYSUNINIT(9)` registers a cleanup function to run at module unload. `EVENTHANDLER(9)` registers a callback to run whenever the kernel fires a named event.

Both mechanisms have been available since FreeBSD's earliest releases. They are boring infrastructure; that is their value. A driver that uses them correctly can react to the full kernel lifecycle without writing a single line of manual registration code. A driver that ignores them either misses its cue or reinvents a worse version of the same facility.

25.9.1 Why the Kernel Needs Boot-Time Ordering

The FreeBSD kernel boots in a precise order. Memory management comes up before any allocator is usable. Tunables are parsed before drivers can read them. Locks are initialised before anything is allowed to acquire them. Filesystems are mounted only after the devices they live on are probed. Each of these dependencies has to be honoured, or the kernel panics before `init(8)` starts.

The mechanism that enforces the ordering is `SYSINIT(9)`. A `SYSINIT` macro declares that a given function should run at a given subsystem ID with a given order constant. The kernel's boot sequence gathers every `SYSINIT` in the running configuration, sorts them by (subsystem, order), and calls them in that sequence. Modules loaded after the kernel has booted still honour their `SYSINIT` declarations: the loader calls them at module attach time, in the same sorted order.

From the driver's point of view, a `SYSINIT` is a way to say "do this thing at that point in the boot sequence, and I do not care which other code is also registering at that point". The kernel handles the sorting; the driver writes the callback.

25.9.2 The Subsystem ID Space

Subsystem IDs are defined in `/usr/src/sys/sys/kernel.h`. The constants have descriptive names and numeric values that reflect their ordering. A driver picks the subsystem that corresponds to its callback's purpose:

- `SI_SUB_TUNABLES (0x0700000)`: evaluate boot-time tunables. This is where `TUNABLE_INT_FETCH` and its siblings run. Code that consumes tunables has to run after this point.
- `SI_SUB_KLD (0x2000000)`: loadable kernel module setup. Early module infrastructure runs here.
- `SI_SUB_SMP (0x2900000)`: bring up the application processors.
- `SI_SUB_DRIVERS (0x3100000)`: let drivers initialise. This is the subsystem most user-land drivers register against if they need early code that runs before any device attaches.
- `SI_SUB_CONFIGURE (0x3800000)`: configure devices. By the end of this subsystem, every compiled-in driver has had a chance to attach.

There are more than a hundred subsystem IDs in `kernel.h`. The ones above are the ones a character-device driver most often interacts with. The numeric values are sorted so that “smaller number” means “earlier in boot”.

Within a subsystem, the order constant gives fine-grained ordering:

- `SI_ORDER_FIRST` (0x0): run before most other code in the same subsystem.
- `SI_ORDER_SECOND`, `SI_ORDER_THIRD`: explicit step-by-step ordering.
- `SI_ORDER_MIDDLE` (0x1000000): run in the middle. Most driver-level `SYSINIT`s use this or the one below.
- `SI_ORDER_ANY` (0xffffffff): run last. The kernel does not promise any specific order among `SI_ORDER_ANY` entries.

The driver author picks the lowest order that makes the callback run after its prerequisites and before its dependents. For most purposes, `SI_ORDER_MIDDLE` is right.

25.9.3 When a Driver Needs `SYSINIT`

Most character-device drivers do not need `SYSINIT` at all. `DRIVER_MODULE` already registers the driver with Newbus; the driver’s `device_attach` method runs when a matching device appears. That is enough for any work that is per-instance.

`SYSINIT` is for work that is not per-instance. A list of reasons a driver might register a `SYSINIT`:

- **Initialise a global pool** that every instance of the driver will draw from. The pool exists once; it does not belong to any one softc.
- **Register with a kernel subsystem** that expects callers to register before they use it. For example, a driver that wants to receive `vm_lowmem` events registers early so that the first low-memory event does not miss it.
- **Parse a complex tunable** that requires more work than a single `TUNABLE_INT_FETCH`. The tunable-parsing code runs during `SI_SUB_TUNABLES` and populates a global structure that per-instance code consults later.
- **Self-test** a cryptographic primitive or a subsystem initialiser before the first caller can use it.

For `myfirst`, none of these applies today. The driver is per-instance, its tunables are simple, and it uses no subsystem that requires pre-registration. Chapter 25 introduces `SYSINIT` not because `myfirst` needs one, but because the reader should be familiar with the macro and understand when a future change would call for it.

25.9.4 The Shape of a `SYSINIT` Declaration

The macro signature is:

```
SYSINIT(uniquifier, subsystem, order, func, ident);
```

- `uniquifier` is a C identifier that ties the `SYSINIT` symbol to this declaration. It appears nowhere else. Convention is to use a short name that matches the subsystem or function.
- `subsystem` is the `SI_SUB_*` constant.

- `order` is the `SI_ORDER_*` constant.
- `func` is a function pointer with signature `void (*)(void *)`.
- `ident` is a single argument passed to `func`. For most uses, it is `NULL`.

The matching cleanup macro is:

```
SYSUNINIT(uniquifier, subsystem, order, func, ident);
```

`SYSUNINIT` registers a cleanup function. It runs at module unload in the reverse order of `SYSINIT` declarations. For code that is compiled into the kernel (not a module), `SYSUNINIT` never fires because the kernel never unloads; but the declaration is still useful because compiling the driver as a module exercises the cleanup path.

25.9.5 A Worked `SYSINIT` Example for `myfirst`

Consider a hypothetical enhancement to `myfirst`: a global, driver-wide pool of pre-allocated log buffers that every instance can draw from. The pool is initialised once per module load and destroyed once per module unload. Per-instance attach and detach do not touch the pool directly; they only take and return buffers from it.

The `SYSINIT` declaration looks like this:

```
#include <sys/kernel.h>

static struct myfirst_log_pool {
    struct mtx      lp_mtx;
    /* ... per-pool state ... */
} myfirst_log_pool;

static void
myfirst_log_pool_init(void *unused __unused)
{
    mtx_init(&myfirst_log_pool.lp_mtx, "myfirst log pool",
            NULL, MTX_DEF);
    /* Allocate pool entries. */
}

static void
myfirst_log_pool_fini(void *unused __unused)
{
    /* Release pool entries. */
    mtx_destroy(&myfirst_log_pool.lp_mtx);
}

SYSINIT(myfirst_log_pool, SI_SUB_DRIVERS, SI_ORDER_MIDDLE,
        myfirst_log_pool_init, NULL);
```

```
SYSUNINIT(myfirst_log_pool, SI_SUB_DRIVERS, SI_ORDER_MIDDLE,  
          myfirst_log_pool_fini, NULL);
```

When `myfirst` is loaded, the kernel sorts the `SYSINIT` entries and calls `myfirst_log_pool_init` during the `SI_SUB_DRIVERS` phase. The first `myfirst_attach` that runs afterwards finds the pool ready. When the module is unloaded, `myfirst_log_pool_fini` runs after every instance has been detached, giving the pool a chance to release its resources.

This is a sketch for instructional purposes; `myfirst` does not actually use a global pool in the shipped Chapter 25 code. The reader who eventually writes a driver that does need one will find the pattern here.

25.9.6 The Ordering Between `SYSINIT` and `DRIVER_MODULE`

`DRIVER_MODULE` itself is implemented as a `SYSINIT` under the hood. It registers the driver with Newbus during a specific subsystem phase, and Newbus's own `SYSINIT`s probe and attach devices afterwards. A driver's custom `SYSINIT` can therefore be ordered relative to `DRIVER_MODULE` by choosing the right subsystem and order.

A rule of thumb:

- `SYSINIT` at `SI_SUB_DRIVERS` with `SI_ORDER_FIRST` runs before `DRIVER_MODULE`'s registration.
- `SYSINIT` at `SI_SUB_CONFIGURE` with `SI_ORDER_MIDDLE` runs after most device attaches but before the final configuration step.

For a global pool that attach depends on, `SI_SUB_DRIVERS` with `SI_ORDER_MIDDLE` is usually correct: the pool is initialised before `DRIVER_MODULE`'s devices start attaching (because `SI_SUB_DRIVERS` is earlier than `SI_SUB_CONFIGURE`), and the order constant keeps it away from the earliest hooks.

25.9.7 EVENTHANDLER: Reacting to Runtime Events

A `SYSINIT` fires once, at a known boot phase. An `EVENTHANDLER` fires zero or more times, whenever a specific system event occurs. The mechanisms are cousins; they complement each other.

The kernel defines a number of named events. Each event has a fixed callback signature and a fixed set of circumstances in which it fires. A driver that cares about an event registers a callback; the kernel invokes the callback every time the event fires; the driver deregisters the callback at detach.

Some commonly useful events:

- `shutdown_pre_sync`: the system is about to sync filesystems. Drivers with in-memory caches flush them here.
- `shutdown_post_sync`: the system has finished syncing filesystems. Drivers that need to know “the filesystem is quiet” hook here.
- `shutdown_final`: the system is about to halt or reboot. Drivers with hardware state that must be saved do it here.
- `vm_lowmem`: the virtual memory subsystem is under pressure. Drivers with caches of their own should release some memory back.
- `power_suspend_early`, `power_suspend`, `power_resume`: suspend/resume lifecycle.

- `dev_clone`: a device cloning event, used by pseudo-devices that appear on demand.

The list is not fixed; new events are added as the kernel grows. The ones above are the ones a general driver most often considers.

25.9.8 The Shape of an EVENTHANDLER Registration

The pattern has three parts: declare a handler function with the right signature, register it at attach time, deregister it at detach time. The registration returns an opaque tag; the deregistration needs that tag.

For `shutdown_pre_sync`, the handler signature is:

```
void (*handler)(void *arg, int howto);
```

`arg` is whatever pointer the driver passed to the registration; typically it is the `softc`. `howto` is the shutdown flags (`RB_HALT`, `RB_REBOOT`, etc.).

A minimal shutdown handler for `myfirst`:

```
#include <sys/eventhandler.h>

static eventhandler_tag myfirst_shutdown_tag;

static void
myfirst_shutdown(void *arg, int howto)
{
    struct myfirst_softc *sc = arg;

    mtx_lock(&sc->sc_mtx);
    DPRINTF(sc, MYF_DBG_INIT, "shutdown: howto=0x%x\n", howto);
    /* Flush any pending state here. */
    mtx_unlock(&sc->sc_mtx);
}
```

The registration happens inside `myfirst_attach` (or in a helper called from it):

```
myfirst_shutdown_tag = EVENTHANDLER_REGISTER(shutdown_pre_sync,
    myfirst_shutdown, sc, SHUTDOWN_PRI_DEFAULT);
```

The deregistration happens inside `myfirst_detach`:

```
EVENTHANDLER_DEREGISTER(shutdown_pre_sync, myfirst_shutdown_tag);
```

The deregistration is mandatory. A driver that detaches without deregistering leaves a dangling callback pointer in the kernel's event list. When the kernel next fires the event, it calls into a memory region that is no longer mapped, and the system panics.

The tag stored in `myfirst_shutdown_tag` is what binds the registration to the deregistration. For a driver with a single instance, a static variable like the one above works. For a driver with multiple instances, the tag should live in the softc so that each instance's deregistration references its own tag.

25.9.9 EVENTHANDLER in the Attach Chain

Because registration and deregistration are symmetric, they slot cleanly into the labelled-cleanup pattern from Section 5. Registration becomes an acquisition; its failure mode is “did the registration return an error?” (it can fail under low-memory conditions); its cleanup is `EVENTHANDLER_DEREGISTER`.

The updated attach fragment for an `EVENTHANDLER`-aware `myfirst`:

```
/* Stage 7: shutdown handler. */
sc->sc_shutdown_tag = EVENTHANDLER_REGISTER(shutdown_pre_sync,
    myfirst_shutdown, sc, SHUTDOWN_PRI_DEFAULT);
if (sc->sc_shutdown_tag == NULL) {
    error = ENOMEM;
    goto fail_log;
}

return (0);

fail_log:
    myfirst_log_detach(sc);
fail_cdev:
    destroy_dev(sc->sc_cdev);
fail_mtx:
    mtx_destroy(&sc->sc_mtx);
return (error);
```

And the matching detach, with the deregistration first (reverse order of acquisition):

```
EVENTHANDLER_DEREGISTER(shutdown_pre_sync, sc->sc_shutdown_tag);
myfirst_log_detach(sc);
destroy_dev(sc->sc_cdev);
mtx_destroy(&sc->sc_mtx);
```

`sc->sc_shutdown_tag` lives in the softc. Storing it there is important: the deregistration needs to know which specific registration to remove, and per-softc storage keeps the two instances of the driver independent.

25.9.10 Priority: `SHUTDOWN_PRI_*`

Within a single event, callbacks are called in priority order. The priority is the fourth argument to `EVENTHANDLER_REGISTER`. For shutdown events, the common constants are:

- SHUTDOWN_PRI_FIRST: run before most other handlers.
- SHUTDOWN_PRI_DEFAULT: run in the default order.
- SHUTDOWN_PRI_LAST: run after other handlers.

A driver with hardware that needs to be quiesced before filesystems flush might register with SHUTDOWN_PRI_FIRST. A driver whose state depends on the filesystems already being flushed (unlikely in practice) might register with SHUTDOWN_PRI_LAST. Most drivers use SHUTDOWN_PRI_DEFAULT and do not think about priority.

Similar priority constants exist for other events (EVENTHANDLER_PRI_FIRST, EVENTHANDLER_PRI_ANY, EVENTHANDLER_PRI_LAST).

25.9.11 When to Use `vm_lowmem`

`vm_lowmem` is the event the VM subsystem fires when free memory drops below a threshold. A driver that maintains a cache of its own (a pool of pre-allocated blocks, for example) can release some of them back to the kernel in response.

The handler is called with a single argument (the subsystem ID that triggered the event). A minimal handler for a driver with a cache:

```
static void
myfirst_lowmem(void *arg, int unused __unused)
{
    struct myfirst_softc *sc = arg;

    mtx_lock(&sc->sc_mtx);
    /* Release some entries from the cache. */
    mtx_unlock(&sc->sc_mtx);
}
```

The registration looks the same as the shutdown one but for the event name:

```
sc->sc_lowmem_tag = EVENTHANDLER_REGISTER(vm_lowmem,
    myfirst_lowmem, sc, EVENTHANDLER_PRI_ANY);
```

A driver that does not maintain a cache should not register for `vm_lowmem`. The cost of doing so is not zero: the kernel calls every registered handler on every low-memory event, and a no-op handler adds latency to that call chain.

For `myfirst`, there is no cache, so `vm_lowmem` is not used. The pattern is introduced for the reader who is about to write a driver that needs it.

25.9.12 `power_suspend_early` and `power_resume`

Suspend/resume is a sensitive lifecycle. Between `power_suspend_early` and `power_resume`, the driver's devices are expected to be quiescent: no I/O, no interrupts, no state transitions. A driver with hardware state

that must be saved before suspend and restored after resume registers handlers for both events.

For character-device drivers that do not manage hardware, these events usually do not apply. For bus-attached drivers (PCI, USB, SPI), the bus layer handles most of the suspend/resume bookkeeping, and the driver just has to provide `device_suspend` and `device_resume` methods in its `device_method_t` table. The `EVENTHANDLER` approach is for drivers that want to react to a system-wide suspend without being bus-attached.

Chapter 26 will revisit suspend/resume when `myfirst` becomes a USB driver; at that point the bus layer's mechanism is the preferred one.

25.9.13 The Module Event Handler

Related to `SYSINIT` and `EVENTHANDLER` is the module event handler: the callback the kernel invokes for `MOD_LOAD`, `MOD_UNLOAD`, `MOD_QUIESCE`, and `MOD_SHUTDOWN`. Most drivers do not override it; `DRIVER_MODULE` supplies a default implementation that calls `device_probe` and `device_attach` appropriately.

A driver that needs custom behaviour at module load (beyond what `SYSINIT` can do) can supply its own handler:

```
static int
myfirst_modevent(module_t mod, int what, void *arg)
{
    switch (what) {
    case MOD_LOAD:
        /* Custom load behaviour. */
        return (0);
    case MOD_UNLOAD:
        /* Custom unload behaviour. */
        return (0);
    case MOD_QUIESCE:
        /* Can we be unloaded? Return errno if not. */
        return (0);
    case MOD_SHUTDOWN:
        /* Shutdown notification; usually no-op. */
        return (0);
    default:
        return (EOPNOTSUPP);
    }
}
```

The handler is wired up through a `moduledata_t` structure rather than through `DRIVER_MODULE`. The two approaches are mutually exclusive for a given module name; a driver picks one or the other.

For most drivers, `DRIVER_MODULE`'s default is correct, and the module-event handler is not customised. `myfirst` uses `DRIVER_MODULE` throughout.

25.9.14 Deregistration Discipline

The single most important rule when using EVENTHANDLER is: register once, deregister once, in attach and detach respectively. Two failure modes show up when the rule is broken.

The first failure mode is the **missed deregistration**. Detach runs, the tag is not deregistered, the kernel's event list still points at the softc's handler, and the next event fires into freed memory. The panic happens far from the cause, because the next event might fire minutes or hours after detach.

The fix is mechanical: every EVENTHANDLER_REGISTER in attach gets a matching EVENTHANDLER_DEREGISTER in detach. The labelled-cleanup pattern from Section 5 makes this easy: registration is an acquisition with a label, and the cleanup chain deregisters in reverse order.

The second failure mode is the **double registration**. A driver that registers the same handler twice has two entries in the kernel's event list; detaching once removes only one of them. The kernel then has a stale entry pointing at the softc that just went away.

The fix is also mechanical: register exactly once per attach. Do not register in a helper that is called from multiple places; do not register lazily in response to a first event.

25.9.15 A Complete Lifecycle Example

Putting SYSINIT, EVENTHANDLER, and the labelled-cleanup attach together, the full lifecycle of a `myfirst` driver with a global pool and a shutdown handler runs as follows:

At kernel boot or module load: - `SI_SUB_TUNABLES` fires. `TUNABLE_*_FETCH` calls in attach will see their values. - `SI_SUB_DRIVERS` fires. `myfirst_log_pool_init` runs (via `SYSINIT`). The global pool is ready. - `SI_SUB_CONFIGURE` fires. `DRIVER_MODULE` registers the driver. Newbus probes; `myfirst_probe` and `myfirst_attach` run for each instance. - Inside `myfirst_attach`: lock, cdev, sysctl, log state, shutdown handler registered.

At runtime: - `ioctl(fd, MYFIRSTIOC_SETMSG, buf)` updates the message. - `devctl_notify` emits `MSG_CHANGED`; `devd` logs it.

At shutdown: - The kernel fires `shutdown_pre_sync`. `myfirst_shutdown` runs for each registered handler. - Filesystems sync. - `shutdown_final` fires. The machine halts.

At module unload (before shutdown): - `MOD_QUIESCE` fires. `myfirst_detach` returns `EBUSY` if any device is in use. - `MOD_UNLOAD` fires. `myfirst_detach` runs for each instance: deregister handler, release log state, destroy cdev, destroy lock. - `SYSUNINIT` fires. `myfirst_log_pool_fini` runs. The global pool is released. - The module unmaps.

Every step is in a well-defined place. Every acquisition has a matching release. A driver that follows the pattern closely is a driver that the FreeBSD kernel can load, run, and unload any number of times without accumulating state.

25.9.16 Deciding What to Register For

A driver author deciding whether to register for an event should ask three questions.

First, **does the event actually matter to this driver?** `vm_lowmem` matters to a driver with a cache; it is noise to a driver without one. `shutdown_pre_sync` matters to a driver whose hardware needs to be quiesced; it is noise to a pseudo-driver. A handler that does nothing useful is still called on every event, slowing the system down slightly on every trigger.

Second, **is the event the right one?** FreeBSD has several shutdown events. `shutdown_pre_sync` fires before filesystem syncs; `shutdown_post_sync` fires after; `shutdown_final` fires just before halt. A driver registering for the wrong phase might flush its cache too early (before data that should be flushed) or too late (after filesystems are already dying).

Third, **has the event been stable across kernel versions?** `shutdown_pre_sync` has been stable for a long time and is safe to use. Newer or more specialised events may change signatures between releases. A driver that targets a specific FreeBSD release (this book is aligned with 14.3) can rely on the events in that release; a driver that targets a range of releases has to be more careful.

For my first, the shipped Chapter 25 registers `shutdown_pre_sync` as a demonstration. The handler is a no-op: it just logs that shutdown is starting. The registration, deregistration, and labelled cleanup are the point of the example, not the handler body.

25.9.17 Common Mistakes with SYSINIT and EVENTHANDLER

A handful of mistakes recur when these mechanisms are used for the first time.

The first mistake is **running heavy code in a SYSINIT**. Boot-time code runs in a context where many kernel subsystems are still initialising. A `SYSINIT` that calls into a complex subsystem may race with that subsystem's own initialisation. The rule is: `SYSINIT` code should be minimal and self-contained. Complex initialisation belongs in the driver's attach routine, which runs after every subsystem has come up.

The second mistake is **using SYSINIT instead of device_attach**. A `SYSINIT` runs once per module load, but `device_attach` runs once per device. A driver that initialises per-device state in a `SYSINIT` is making a category error; the per-device state does not exist yet at `SYSINIT` time.

The third mistake is **forgetting the priority argument on EVENTHANDLER_REGISTER**. The function takes four arguments: event name, callback, argument, priority. Some drivers forget the priority and pass the wrong number of arguments; the compiler catches this with an error, but a driver that happens to pass 0 by accident registers with the lowest possible priority, which may be wrong.

The fourth mistake is **not zeroing the tag field**. If `sc->sc_shutdown_tag` is uninitialised when `EVENTHANDLER_DEREGISTER` is called on a failure path, the deregistration tries to remove a tag that was never registered. The kernel detects this (the tag does not exist in its event list) and the deregistration is a no-op, but the pattern is fragile. The cleaner discipline is to zero the `softc` at allocation time (Newbus does this automatically via `device_get_softc`, but drivers that allocate their own `softcs` have to do it manually) and to never reach a deregistration for a tag that was not registered.

25.9.18 Wrapping Up Section 8

`SYSINIT` and `EVENTHANDLER` are the kernel's way of letting a driver participate in lifecycles beyond its own attach/detach window. `SYSINIT` runs code at a specific boot phase; `EVENTHANDLER` runs code in response to a

named kernel event. Together they cover the cases where per-device code is not enough and the driver has to engage with the system as a whole.

`myfirst` at the end of Chapter 25 uses `EVENTHANDLER_REGISTER` for a demonstration `shutdown_pre_sync` handler; the registration, deregistration, and labelled-cleanup shape are all in place. `SYSINIT` is introduced but not used, because `myfirst` has no global pool today. The patterns are planted; when a future chapter's driver does need them, the reader will recognise them immediately.

With Section 8 complete, every mechanism the chapter set out to teach is in the driver. The remaining material in the chapter applies the mechanisms through hands-on labs, challenge exercises, and a troubleshooting reference for when things go wrong.

25.10 Hands-on Labs

The labs in this section exercise the chapter's mechanisms on a real FreeBSD 14.3 system. Each lab has a specific measurable outcome; after running the lab, you should be able to state what you saw and what it means. The labs assume you have the `examples/part-05/ch25-advanced/` companion directory at hand.

Before starting, build the driver as shipped at the top of `ch25-advanced/`:

```
# cd examples/part-05/ch25-advanced
# make clean
# make
# kldload ./myfirst.ko
# ls /dev/myfirst*
/dev/myfirst0
```

If any of these steps fails, fix the toolchain or the source before continuing. The labs assume a working baseline.

25.10.1 Lab 1: Reproducing a Log Flood

Purpose: see the difference between an unrated `device_printf` and the rate-limited `DLOG_RL` when fired in a hot loop.

Source: `examples/part-05/ch25-advanced/lab01-log-flood/` contains a small user-space program that calls `read()` on `/dev/myfirst0` 10,000 times as fast as the kernel will let it.

Step 1. Temporarily set the debug mask to enable the I/O class and the `printf` on the read path:

```
# sysctl dev.myfirst.0.debug.mask=0x4
```

The mask bit `0x4` enables `MYF_DBG_IO`, which the read callback uses.

Step 2. Run the flood with the naive `DPRINTF` version of the driver first. Build and load `myfirst-flood-unlimited.ko` from `lab01-log-flood/unlimited/`:

```
# make -C lab01-log-flood/unlimited
# kldunload myfirst
# kldload lab01-log-flood/unlimited/myfirst.ko
# dmesg -c > /dev/null
# ./lab01-log-flood/flood 10000
# dmesg | wc -l
```

Expected outcome: approximately 10,000 lines in `dmesg`. The console may also be filled; the system's log buffer wraps, and earlier messages are lost.

Step 3. Unload and reload the rate-limited version from `lab01-log-flood/limited/`, which uses `DLOG_RL` with a 10 pps cap:

```
# kldunload myfirst
# kldload lab01-log-flood/limited/myfirst.ko
# dmesg -c > /dev/null
# ./lab01-log-flood/flood 10000
# sleep 5
# dmesg | wc -l
```

Expected outcome: approximately 50 lines in `dmesg`. The flood now emits at most 10 messages per second; the 10-second test window produces roughly 50 messages (the first second's burst token plus subsequent seconds' allowances, so count exactly may vary but should be within ten).

Step 4. Compare the two outputs side by side. The rate-limited version is readable; the unlimited version is not. Both drivers had identical read behaviour; only the logging discipline differs.

Record: the wall-clock time it took the flood to complete in both cases. The unlimited version is noticeably slower because the console output itself is a bottleneck. Rate-limiting has a visible performance benefit as well as a clarity benefit.

25.10.2 Lab 2: `errno` Audit with `truss`

Purpose: see what `truss(1)` reports when the driver returns different `errno` values, and calibrate your intuition about which `errno` to return from which code path.

Source: `examples/part-05/ch25-advanced/lab02-errno-audit/` contains a user program that makes a series of deliberately invalid calls and a script that runs it under `truss`.

Step 1. Load the stock `myfirst.ko` if not already loaded:

```
# kldload ./myfirst.ko
```

Step 2. Run the audit program under `truss`:

```
# truss -f -o /tmp/audit.truss ./lab02-errno-audit/audit
# less /tmp/audit.truss
```

The program performs these operations in sequence: 1. Opens `/dev/myfirst0`. 2. Issues an unknown ioctl command (command number 99). 3. Issues `MYFIRSTIOC_SETMSG` with a NULL argument. 4. Writes a zero-length buffer. 5. Writes a buffer larger than the driver accepts. 6. Sets `dev.myfirst.0.timeout_sec` to a value larger than allowed. 7. Closes.

Step 3. In the truss output, find each operation and note its `errno`. The expected outcomes:

1. `open`: returns a file descriptor. No `errno`.
2. `ioctl(_IOC=0x99)`: returns `ENOTTY` (the kernel's translation of the driver's `ENOIOCTL`).
3. `ioctl(MYFIRSTIOC_SETMSG, NULL)`: returns `EFAULT` (the kernel catches the NULL before the handler runs).
4. `write(0 bytes)`: returns 0 (no error, just no bytes written).
5. `write(oversize)`: returns `EINVAL` (the driver rejects lengths above its buffer size).
6. `sysctl write out-of-range`: returns `EINVAL` (the `sysctl` handler rejects the value).
7. `close`: returns 0. No `errno`.

Step 4. For each observed `errno`, locate the driver code that returned it. Walk the call chain from `truss` to the kernel source and confirm that the `errno` you see in `truss` is the one the driver returned. This exercise calibrates your mental map between “what the user sees” and “what the driver says”.

25.10.3 Lab 3: Tunable Reboot Behaviour

Purpose: verify that a loader tunable actually changes the driver's initial state when the module is first loaded.

Source: `examples/part-05/ch25-advanced/lab03-tunable-reboot/` contains a helper script `apply_tunable.sh`.

Step 1. With the stock module loaded and no tunable set, confirm the initial value of the timeout:

```
# kldload ./myfirst.ko
# sysctl dev.myfirst.0.timeout_sec
dev.myfirst.0.timeout_sec: 5
```

The default is 5, set in the `attach` routine.

Step 2. Unload the module, set the loader tunable, reload, and confirm the new initial value:

```
# kldunload myfirst
# kenv hw.myfirst.timeout_sec=12
# kldload ./myfirst.ko
# sysctl dev.myfirst.0.timeout_sec
dev.myfirst.0.timeout_sec: 12
```

The tunable set via `kenv(1)` took effect because `TUNABLE_INT_FETCH` in `attach` read it before the `sysctl` was published.

Step 3. Change the `sysctl` at runtime and confirm that the change is accepted but does not propagate back to the tunable:

```
# sysctl dev.myfirst.0.timeout_sec=25
dev.myfirst.0.timeout_sec: 12 -> 25
# kenv hw.myfirst.timeout_sec
hw.myfirst.timeout_sec="12"
```

The tunable still reads 12; the `sysctl` reads 25. The tunable is the initial value; the `sysctl` is the runtime value. They diverge the moment the `sysctl` is written.

Step 4. Unload and reload. The tunable value is still 12 (because it is in the kernel environment), so the new `sysctl` starts at 12, not 25. This is the lifecycle: tunable sets initial, `sysctl` sets runtime, unload loses runtime, tunable survives.

Step 5. Clear the tunable and reload:

```
# kldunload myfirst
# kenv -u hw.myfirst.timeout_sec
# kldload ./myfirst.ko
# sysctl dev.myfirst.0.timeout_sec
dev.myfirst.0.timeout_sec: 5
```

Back to the attach-time default. The lifecycle is consistent end to end.

25.10.4 Lab 4: Deliberate Attach Failure Injection

Purpose: verify that every label in the cleanup chain is reached and that no resources leak when a failure is injected in the middle of `attach`.

Source: `examples/part-05/ch25-advanced/lab04-failure-injection/` contains four build variants of the module, each compiling in a different failure injection point:

- `inject-mtx/`: fails just after the lock is initialised.
- `inject-cdev/`: fails just after the `cdev` is created.
- `inject-sysctl/`: fails just after the `sysctl` tree is built.
- `inject-log/`: fails just after the log state is initialised.

Each variant defines exactly one of the `MYFIRST_DEBUG_INJECT_FAIL_*` macros from Section 5.

Step 1. Build and load the first variant. The load should fail:

```
# make -C lab04-failure-injection/inject-mtx
# kldload lab04-failure-injection/inject-mtx/myfirst.ko
kldload: an error occurred while loading module myfirst. Please check dmesg(8) for more
↪ details.
```

```
# dmesg | tail -3
myfirst0: attach: stage 1 complete
myfirst0: attach: injected failure after mtx_init
device_attach: myfirst0 attach returned 12
```

The attach function returned ENOMEM (errno 12) at the injected failure point. The module is not loaded:

```
# kldstat -n myfirst
kldstat: can't find file: myfirst
```

Step 2. Repeat for the other three variants. Each should fail at the specific stage the name suggests, and each should leave the kernel in a clean state. To confirm the clean state, check for leftover sysctl OIDs, leftover cdevs, and leftover locks:

```
# sysctl dev.myfirst 2>&1 | head
sysctl: unknown oid 'dev.myfirst'
# ls /dev/myfirst* 2>&1
ls: No match.
# dmesg | grep -i "witness\|leak"
```

Expected: no matches. No sysctl, no cdev, no witness complaints. The cleanup chain is working.

Step 3. Run the combined regression script that builds every variant and checks the results automatically:

```
# ./lab04-failure-injection/run.sh
```

The script builds each variant, loads it, confirms that the load fails, confirms that the state is clean, and reports a one-line summary per variant. A pass on all four variants means every label in the cleanup chain has been exercised on a real kernel and released every resource that was held at that label.

25.10.5 Lab 5: shutdown_pre_sync Handler

Purpose: confirm that the registered shutdown handler actually fires during a real shutdown, and observe its ordering relative to filesystem sync.

Source: `examples/part-05/ch25-advanced/lab05-shutdown-handler/` contains a version of `myfirst.ko` whose `shutdown_pre_sync` handler prints a distinctive message to the console.

Step 1. Load the module and verify the handler is registered by reading the log on attach:

```
# kldload lab05-shutdown-handler/myfirst.ko
# dmesg | tail -1
myfirst0: attach: shutdown_pre_sync handler registered
```

Step 2. Issue a reboot. On a test machine (not a production machine), the simplest way is:

```
# shutdown -r +1 "testing myfirst shutdown handler"
```

Watch the console as the machine shuts down. The expected sequence:

```
myfirst0: shutdown: howto=0x4
Syncing disks, buffers remaining... 0 0 0
Uptime: ...
```

The `myfirst0: shutdown: howto=0x4` line appears **before** “Syncing disks”, because `shutdown_pre_sync` fires before filesystems sync. If the handler message appears after the sync message, the registration was at the wrong event (`shutdown_post_sync` or `shutdown_final`). If the message never appears, the handler was never registered or never deregistered (double-free would cause a panic, but silent missing would suggest a registration bug).

Step 3. After the machine reboots, confirm that unloading before shutdown still removes the handler cleanly:

```
# kldload lab05-shutdown-handler/myfirst.ko
# kldunload myfirst
# dmesg | tail -2
myfirst0: detach: shutdown handler deregistered
myfirst0: detach: complete
```

The deregistration message confirms the cleanup path in `detach` ran. The `attach/detach` pair is symmetric; no event list entries leak.

25.10.6 Lab 6: The 100-Cycle Regression Script

Purpose: run a sustained load/unload cycle to catch leaks that only appear under repeated `attach/detach`. This is the test from Section 7’s production checklist.

Source: `examples/part-05/ch25-advanced/lab06-100-cycles/` contains `run.sh`, which performs 100 cycles of `kldload / sleep / kldunload` and records the kernel’s memory footprint before and after.

Step 1. Record the kernel’s initial memory footprint:

```
# vmstat -m | awk '$1=="Solaris" || $1=="kernel"' > /tmp/before.txt
# cat /tmp/before.txt
```

Step 2. Run the cycle script:

```
# ./lab06-100-cycles/run.sh
cycle 1/100: ok
cycle 2/100: ok
...
cycle 100/100: ok
done: 100 cycles, 0 failures, 0 leaks detected.
```

Step 3. Record the final memory footprint:

```
# vmstat -m | awk '$1=="Solaris" || $1=="kernel"' > /tmp/after.txt
# diff /tmp/before.txt /tmp/after.txt
```

Expected: no significant difference. If there is a difference of more than a few kilobytes (the kernel's own bookkeeping fluctuates), the driver has a leak.

Step 4. If the script reports any failure, examine `/tmp/myfirst-cycles.log` (which `run.sh` populates) for the first failing cycle. The failure is usually at the deregistration step: a missing `EVENTHANDLER_DEREGISTER` or a missing `mtx_destroy`.

A clean 100-cycle run is one of the simplest ways to gain confidence in a driver's lifecycle discipline. Repeat it after every material change to the attach or detach chain.

25.10.7 Lab 7: Capability Discovery in User Space

Purpose: confirm that a user-space program can discover the driver's capabilities at run time and behave accordingly, as designed in Section 4.

Source: `examples/part-05/ch25-advanced/lab07-getcaps/` contains `mfctl25.c`, an updated version of `myfirstctl` that issues `MYFIRSTIOC_GETCAPS` before each operation and skips unsupported ones.

Step 1. Build `mfctl25`:

```
# make -C lab07-getcaps
```

Step 2. Run against the stock Chapter 25 driver and observe the capability report:

```
# ./lab07-getcaps/mfctl25 caps
Driver reports capabilities:
  MYF_CAP_RESET
  MYF_CAP_GETMSG
  MYF_CAP_SETMSG
```

The driver reports three capabilities. The `MYF_CAP_TIMEOUT` bit is defined but not set because the timeout behaviour is a `sysctl`, not an `ioctl`.

Step 3. Run each operation and confirm that the program only attempts supported ones:


```
# ./lab07-getcaps/mfctl25 reset
# ./lab07-getcaps/mfctl25 getmsg
Current message: Hello from myfirst
# ./lab07-getcaps/mfctl25 setmsg "new message"
# ./lab07-getcaps/mfctl25 timeout
Timeout ioctl not supported; use sysctl dev.myfirst.0.timeout_sec instead.
```

The last line is the capability check firing: the program asked for `MYF_CAP_TIMEOUT`, the driver did not advertise it, and the program printed a helpful message instead of issuing an `ioctl` that would return `ENOTTY`.

Step 4. Load an older build (Chapter 24's `myfirst.ko` in `lab07-getcaps/ch24/`) and rerun:

```
# kldunload myfirst
# kldload lab07-getcaps/ch24/myfirst.ko
# ./lab07-getcaps/mfctl25 caps
GETCAPS ioctl not supported. Falling back to default feature set:
  MYF_CAP_RESET
  MYF_CAP_GETMSG
  MYF_CAP_SETMSG
```

When `GETCAPS` itself returns `ENOTTY`, the program falls back to a safe default set that matches Chapter 24's known behaviour. This is the forward-compatibility pattern in action.

Step 5. Reload the Chapter 25 driver to restore the test state:

```
# kldunload myfirst
# kldload ./myfirst.ko
```

The exercise demonstrates that capability discovery lets one user-space program work correctly across two driver versions, which is the whole point of the pattern.

25.10.8 Lab 8: Sysctl Range Validation

Purpose: confirm that every writable sysctl the driver exposes rejects out-of-range values and leaves the internal state untouched on rejection.

Source: `examples/part-05/ch25-advanced/lab08-sysctl-validation/` contains the driver built with range checks and a test script `run.sh` that drives every sysctl to its limits.

Step 1. Load the driver and list its writable sysctls:

```
# kldload ./myfirst.ko
# sysctl -W dev.myfirst.0 | grep -v "^dev.myfirst.0.debug.classes"
dev.myfirst.0.timeout_sec: 5
dev.myfirst.0.max_retries: 3
```

```
dev.myfirst.0.log_ratelimit_pps: 10
dev.myfirst.0.debug.mask: 0
```

Four writable sysctls. Each has a specific valid range.

Step 2. Try to set each sysctl to zero, to its maximum allowed, and to one above its maximum:

```
# sysctl dev.myfirst.0.timeout_sec=0
sysctl: dev.myfirst.0.timeout_sec: Invalid argument
# sysctl dev.myfirst.0.timeout_sec=60
dev.myfirst.0.timeout_sec: 5 -> 60
# sysctl dev.myfirst.0.timeout_sec=61
sysctl: dev.myfirst.0.timeout_sec: Invalid argument
# sysctl dev.myfirst.0.timeout_sec
dev.myfirst.0.timeout_sec: 60
```

The out-of-range attempts are rejected with `EINVAL`; the internal value is unchanged. The valid assignment to 60 succeeds.

Step 3. Repeat for the other sysctls:

- `max_retries`: valid range 1-100. Try 0, 100, 101.
- `log_ratelimit_pps`: valid range 1-10000. Try 0, 10000, 10001.
- `debug.mask`: valid range 0-0xff (the defined bits). Try 0, 0xff, 0x100.

For each, the script reports pass or fail. A driver that passes every case has correct handler-level validation.

Step 4. Examine the sysctl handlers in `examples/part-05/ch25-advanced/myfirst_sysctl.c` and note the pattern:

```
static int
myfirst_sysctl_timeout_sec(SYSCTL_HANDLER_ARGS)
{
    struct myfirst_softc *sc = arg1;
    u_int new_val;
    int error;

    mtx_lock(&sc->sc_mtx);
    new_val = sc->sc_timeout_sec;
    mtx_unlock(&sc->sc_mtx);

    error = sysctl_handle_int(oidp, &new_val, 0, req);
    if (error != 0 || req->newptr == NULL)
        return (error);

    if (new_val < 1 || new_val > 60)
```

```
    return (EINVAL);

    mtx_lock(&sc->sc_mtx);
    sc->sc_timeout_sec = new_val;
    mtx_unlock(&sc->sc_mtx);
    return (0);
}
```

Note the order of operations: copy the current value out for the read side, call `sysctl_handle_int` to handle the copy, validate on write, commit under the lock only after validation succeeds. A handler that commits before validation exposes inconsistent state to concurrent readers.

Step 5. Confirm that the `sysctl` description is helpful (`sysctl -d`):

```
# sysctl -d dev.myfirst.0.timeout_sec
dev.myfirst.0.timeout_sec: Operation timeout in seconds (range 1-60)
```

The description states the unit and the range. A user reading the `sysctl` without consulting any documentation can still set it correctly.

25.10.9 Lab 9: Log Message Audit Across the Driver

Purpose: inventory every log message in the driver and confirm that each one follows the Section 1 and Section 2 disciplines (`device_printf`, includes `errno` where relevant, rate-limited when on a hot path).

Source: `examples/part-05/ch25-advanced/lab09-log-audit/` contains an audit script `audit.sh` and a `grep`-based checker.

Step 1. Run the audit script against the driver's source:

```
# cd examples/part-05/ch25-advanced
# ./lab09-log-audit/audit.sh
```

The script greps for every `printf`, `device_printf`, `log`, `DPRINTF`, and `DLOG_RL` call in the source tree and categorises each into:

- PASS: uses `device_printf` or `DPRINTF`, device name is implicit.
- PASS: uses `DLOG_RL` on a hot path.
- WARN: uses `printf` without device context (may be legitimate at `MOD_LOAD`).
- FAIL: uses `device_printf` on a hot path without rate limiting.

Expected output (for the Chapter 25 stock driver):

```
myfirst.c:    15 log messages - 15 PASS
myfirst_cdev.c: 6 log messages - 6 PASS
```

```
myfirst_ioctl.c: 4 log messages - 4 PASS
myfirst_sysctl.c: 0 log messages
myfirst_log.c: 2 log messages - 2 PASS
Total: 27 log messages - 0 WARN, 0 FAIL
```

Step 2. Intentionally break one message (for example, change a `DPRINTF(sc, MYF_DBG_IO, ...)` in the read callback to a bare `device_printf(sc->sc_dev, ...)`) and rerun:

```
myfirst_cdev.c: 6 log messages - 5 PASS, 1 FAIL
  myfirst_cdev.c:83: device_printf on hot path not rate-limited
Total: 27 log messages - 0 WARN, 1 FAIL
```

The audit caught the regression. Revert the change and rerun to confirm the count returns to zero failures.

Step 3. Add a new log message to a non-hot path (for example, a one-time initialisation message at attach). Confirm the audit accepts it as a PASS:

```
device_printf(dev, "initialised with timeout %u\n",
  sc->sc_timeout_sec);
```

One-shot messages at attach do not need rate limiting because they fire exactly once per instance per load.

Step 4. For each message the audit categorises as a PASS, confirm that the message includes meaningful context. A message like “error” is a PASS for the audit tool but a FAIL for the human reader. A second manual pass over the grep output is required to confirm the messages are actually useful.

The lab demonstrates two points. First, a mechanical audit catches the categorical rules (rate limiting on hot paths, `device_printf` over bare `printf`) but cannot judge message quality. Second, the human pass is what confirms that the messages contain enough context to be diagnosed from. Both passes together give the driver a log surface that will actually help a future support engineer.

25.10.10 Lab 10: Multi-Version Compatibility Matrix

Purpose: confirm that the capability-discovery pattern introduced in Section 4 actually allows a single user-space program to work with three different versions of the driver.

Source: `examples/part-05/ch25-advanced/lab10-compat-matrix/` contains three pre-built `.ko` files corresponding to driver versions 1.6-debug, 1.7-integration, and 1.8-maintenance, plus a single user-space program `mfctl-universal` that uses `MYFIRSTIOC_GETCAPS` (or a fallback) to decide which operations to attempt.

Step 1. Load each driver version in turn and run `mfctl-universal --caps` against it:

```
# kldload lab10-compat-matrix/v1.6/myfirst.ko
# ./lab10-compat-matrix/mfctl-universal --caps
Driver: version 1.6-debug
GETCAPS ioctl: not supported
```

```
Using fallback capability set:
    MYF_CAP_GETMSG
    MYF_CAP_SETMSG

# kldunload myfirst
# kldload lab10-compat-matrix/v1.7/myfirst.ko
# ./lab10-compat-matrix/mfctl-universal --caps
Driver: version 1.7-integration
GETCAPS ioctl: not supported
Using fallback capability set:
    MYF_CAP_RESET
    MYF_CAP_GETMSG
    MYF_CAP_SETMSG

# kldunload myfirst
# kldload lab10-compat-matrix/v1.8/myfirst.ko
# ./lab10-compat-matrix/mfctl-universal --caps
Driver: version 1.8-maintenance
GETCAPS ioctl: supported
Driver reports capabilities:
    MYF_CAP_RESET
    MYF_CAP_GETMSG
    MYF_CAP_SETMSG
```

Three driver versions, one user-space program, three different capability decisions. The program works with each version.

Step 2. Exercise each capability in turn and confirm the program skips unsupported ones:

```
# kldunload myfirst
# kldload lab10-compat-matrix/v1.6/myfirst.ko
# ./lab10-compat-matrix/mfctl-universal reset
reset: not supported on this driver version (1.6-debug)
# ./lab10-compat-matrix/mfctl-universal getmsg
Current message: Hello from myfirst
```

The reset operation, which was added in 1.7, is skipped cleanly on 1.6. The program prints a helpful message rather than issuing an ioctl that would return ENOTTY.

Step 3. Read the `mfctl-universal` source and note the three-tier fallback:

```
uint32_t
driver_caps(int fd, const char *version)
{
    uint32_t caps;
```

```

if (ioctl(fd, MYFIRSTIOC_GETCAPS, &caps) == 0)
    return (caps);
if (errno != ENOTTY)
    err(1, "GETCAPS ioctl");

/* Fallback by version string. */
if (strstr(version, "1.8-") != NULL)
    return (MYF_CAP_RESET | MYF_CAP_GETMSG |
            MYF_CAP_SETMSG);
if (strstr(version, "1.7-") != NULL)
    return (MYF_CAP_RESET | MYF_CAP_GETMSG |
            MYF_CAP_SETMSG);
if (strstr(version, "1.6-") != NULL)
    return (MYF_CAP_GETMSG | MYF_CAP_SETMSG);

/* Unknown version: use the minimal safe set. */
return (MYF_CAP_GETMSG);
}

```

The first tier asks the driver directly. The second tier matches known version strings. The third tier falls back to a minimal set that every version of the driver has supported.

Step 4. Reflect on what happens when version 1.9 ships with a new capability bit. The program does not need to be updated: MYFIRSTIOC_GETCAPS on 1.9 will report the new bit, the program will see it, and if the program knows the corresponding operation it will use it. If the program does not know the operation, the bit is ignored. Either way, the program continues to work.

The lab demonstrates that capability discovery is not an abstract pattern; it is the specific mechanism that lets one user-space program span three driver versions without modification.

25.11 Challenge Exercises

The challenges in this section go beyond the labs. Each one asks you to extend the driver in a direction the chapter pointed at but did not complete. Work through them when you are ready; none of them requires new kernel knowledge beyond what the chapter covered, but each one requires a careful walk through the existing code.

25.11.1 Challenge 1: Per-Class Rate Limits

Section 1 sketched three rate-limit slots in the softc (`sc_rl_generic`, `sc_rl_io`, `sc_rl_intr`), but the `DLOG_RL` macro uses a single pps value (`sc_log_pps`). Extend the driver so that each class has its own sysctl-configurable pps cap:

- Add `sc_log_pps_io` and `sc_log_pps_intr` fields to the `softc` alongside `sc_log_pps` (which stays as the generic cap).
- Add matching `sysctls` under `dev.myfirst.<unit>.log.pps_*` and matching `tunables` under `hw.myfirst.log_pps_*`.
- Update `DLOG_RL_IO` and `DLOG_RL_INTR` helpers (or a generic helper that takes both the class and the pps value) to honour the per-class cap.

Write a short test program that fires a burst of messages in each class and confirm from `dmesg` that each class is rate-limited independently. The generic bucket should not starve the I/O bucket, and vice versa.

Hint: the most reusable shape is a helper function `myfirst_log_ratelimited(sc, class, fmt, ...)` that looks up the right rate-limit state and the right pps cap given the class bit. The `DLOG_RL_*` macros become thin wrappers over the helper.

25.11.2 Challenge 2: A Writable String `sysctl`

Section 3 warned about the complication of writable string `sysctls`. Implement one correctly. The `sysctl` should be `dev.myfirst.<unit>.message`, with `CTLFLAG_RW`, and should let an operator rewrite the in-driver message with a single `sysctl(8)` call.

Requirements:

1. The handler must take the `softc` mutex around the update.
2. The handler must validate the length against `sizeof(sc->sc_msg)` and reject oversized strings with `EINVAL`.
3. The handler must use `sysctl_handle_string` for the copy; do not reimplement user-space access.
4. On successful update, the handler must emit a `devctl_notify` for `MSG_CHANGED`, just like the `ioctl` does.

Test with:

```
# sysctl dev.myfirst.0.message="hello from sysctl"
# sysctl dev.myfirst.0.message
dev.myfirst.0.message: hello from sysctl
# sysctl dev.myfirst.0.message="$(printf 'A%.0s' {1..1000})"
sysctl: dev.myfirst.0.message: Invalid argument
```

The second `sysctl` should fail (oversize), and the driver's message should be unchanged.

Consider: should the `ioctl` and the `sysctl` emit the same `MSG_CHANGED` event, or different ones? Both sides are updating the same underlying state; a single event type is probably correct. Document your decision in `MAINTENANCE.md`.

25.11.3 Challenge 3: A `MOD_QUIESCE` Handler Separate from `Detach`

Section 7 noted that `MOD_QUIESCE` and `MOD_UNLOAD` are conceptually distinct but that `myfirst` handles both through `myfirst_detach`. Split them so the quiesce question can be answered without side effects.

Requirements:

1. Add an explicit `MOD_QUIESCE` check to a module event handler. The handler returns `EBUSY` if any device is open, `0` otherwise.
2. The handler does not call `destroy_dev`, does not destroy locks, does not alter state. It only reads `sc_open_count`.
3. For each attached instance, iterate via `devclass` and check every `softc`. Use the `myfirst_devclass` symbol that `DRIVER_MODULE` exports.

Hint: look at `/usr/src/sys/kern/subr_bus.c` for `devclass_get_softc` and related helpers. They are the way to enumerate `softcs` from a module-level function that does not have a `device_t`.

Test: open `/dev/myfirst0`, try `kldunload myfirst`, confirm that it reports “module busy” and that the driver is unchanged. Close the fd, retry the unload, confirm that it succeeds.

25.11.4 Challenge 4: A Drain-Based Detach Instead of `EBUSY`

The chapter’s detach pattern refuses to unload if the driver is in use. A more elaborate pattern drains in-flight references rather than refusing. Implement it.

Requirements:

1. Add an `is_dying` boolean to the `softc`, protected by `sc_mtx`.
2. In `myfirst_open`, check `is_dying` under the lock and return `ENXIO` if true.
3. In `myfirst_detach`, set `is_dying` under the lock. Wait for `sc_open_count` to reach zero, using `mtx_sleep` with a condition variable or a simple polling loop with a timeout.
4. After `sc_open_count` reaches zero, proceed with `destroy_dev` and the rest of the detach chain.

Add a timeout: if `sc_open_count` does not reach zero within (say) 30 seconds, return `EBUSY` from detach. The operator gets a clear signal that the driver is not draining; they can kill the offending process and retry.

Test: open `/dev/myfirst0` in a loop from one shell, call `kldunload myfirst` from another shell, and observe the drain behaviour.

25.11.5 Challenge 5: A Sysctl-Driven Version Bump Check

Write a small user-space program that reads `dev.myfirst.<unit>.version`, parses the version string, and compares it against a minimum version the program requires. The program should print “ok” if the driver is new enough and “driver too old, please update” if not.

Requirements:

1. Parse the string `X.Y-tag` into integers. Reject malformed strings with a clear error.
2. Compare against a minimum of “1.8”. A driver reporting “1.7-integration” should fail the check; a driver reporting “1.8-maintenance” should pass; a driver reporting “2.0-something” should pass.
3. Exit with status `0` on success, non-zero on failure, so the check can be used in shell scripts.

Reflect: could a well-designed program rely on the version string for compatibility checking, or is the capability bitmask from Section 4 the better signal? There is no single right answer; the exercise is to think about the trade-off.

25.11.6 Challenge 6: Add a Sysctl That Enumerates Open File Descriptors

Add a new sysctl `dev.myfirst.<unit>.open_fds` that returns, as a string, the PIDs of the processes that currently have the device open. This is harder than it sounds: the driver does not normally track which process opened each fd.

Hint: in `myfirst_open`, store the calling thread's PID in a linked list under the `softc`. In `myfirst_close`, remove the corresponding entry. In the sysctl handler, walk the list under the `softc` mutex and build a comma-separated string of PIDs.

Edge cases:

1. A process that is open multiple times (multiple fds, forked children) should appear once or multiple times? Decide and document.
2. The list must be bounded in length (attackers could open the device millions of times).
3. The sysctl value is read-only; the handler must not modify the list.

Reflect: is this information actually useful, or is `fstat(1)` a better tool for the same job? The answer depends on whether the driver can provide information that user-space tools cannot derive themselves.

25.11.7 Challenge 7: A Second EVENTHANDLER for `vm_lowmem`

`myfirst` does not have a cache today, but imagine it did: a pool of pre-allocated 4 KB buffers used for read/write operations. Under low memory, the driver should release some of the buffers back.

Implement a synthetic cache: allocate an array of 64 `malloc(M_TEMP, 4096)` pointers at attach. Register a `vm_lowmem` handler that, when fired, releases half of the cached buffers. Reattach reallocates them.

Requirements:

1. The cache allocations happen under the `softc` mutex.
2. The `vm_lowmem` handler takes the mutex, scans the array, and `free()`s the first 32 buffers.
3. A sysctl `dev.myfirst.<unit>.cache_free` reports the current number of free (NULL) slots; an operator can confirm that the handler fired.

Test: use a `stress -m 10 --vm-bytes 512M` loop to drive the system into low-memory pressure, and watch the `cache_free` sysctl. Over time, it should grow as `vm_lowmem` fires repeatedly.

Reflect: is this what the event is meant to be used for? Many drivers that register for `vm_lowmem` have caches that are much larger than 64 buffers; the cost/benefit is different. This is a teaching exercise; a real driver would think harder about whether its cache is worth the complexity.

25.11.8 Challenge 8: A `MYFIRSTIOC_GETSTATS` Ioctl Returning a Structured Payload

So far every ioctl the driver handles returns a scalar: an integer, a `uint32`, or a fixed-size string. Add a `MYFIRSTIOC_GETSTATS` ioctl that returns a structured payload with every counter the driver maintains.

Requirements:

1. Define `struct myfirst_stats` in `myfirst_ioctl.h` with fields for `open_count`, `total_reads`, `total_writes`, `log_drops` (a new counter you add), and `last_error_errno` (another new counter).

2. Add `MYFIRSTIOC_GETSTATS` with command number 6, declared as `_IOR('M', 6, struct my-first_stats)`.
3. The handler copies the softc's counters into the payload under `sc_mtx` and returns.
4. Advertise a new capability bit `MYF_CAP_STATS` in the `GETCAPS` response.
5. Update `MAINTENANCE.md` to document the new `ioctl` and the new capability.

Edge cases:

1. What happens if the struct size changes later? The `_IOR` macro bakes the size into the command number. Adding a field bumps the command number, which breaks old callers. The fix is to include a version and a reserved space in the struct from day one; any future addition reuses the reserved space.
2. Is it safe to return all counters atomically, or do they need separate locking? Holding `sc_mtx` for the duration of the copy is the simplest discipline.

Reflect: this is where `ioctl` design starts to feel complex. For a simple counter snapshot, a `sysctl` with a string-format output might be easier than an `ioctl` with a versioned struct. Which would you pick, and why?

25.11.9 Challenge 9: Devctl-Based Live Monitoring

Add a second `devctl_notify` event that fires every time the rate-limit bucket drops a message. The event should include the class name and the current bucket state as `key=value` data.

Requirements:

1. When `ppsratecheck` returns zero (message dropped), increment a per-class drop counter and emit a `devctl_notify` with `system="myfirst"`, `type="LOG_DROPPED"`, and data `"class=io drops=42"`.
2. The `devctl` event itself must be rate-limited; otherwise the act of reporting drops becomes another flood. Use a second `ppsratecheck` with a slow cap (e.g. 1 pps) for the `devctl` emissions.
3. Write a `devd` rule that matches the event and logs a summary every time it fires.

Test: run Lab 1's flood program and confirm that `devctl` emits the drop report without flooding itself.

25.12 Troubleshooting Guide

When the mechanisms in this chapter misbehave, the symptoms are often indirect: a silent missing log line, a driver that refuses to load for the wrong reason, a `sysctl` that does not appear, a reboot that does not call your handler. This reference maps common symptoms to the mechanism that is most likely responsible, with the first places to look in the driver source.

25.12.1 Symptom: `kldload` returns “Exec format error”

The module was built against a kernel ABI that does not match the running kernel. The typical cause is a mismatch between the running kernel version and the `SYSDIR` used at compile time.

Check: `uname -r` and the value of `SYSDIR` in the Makefile. If the kernel is 14.3-RELEASE but the build picked up headers from a newer 15.0-CURRENT tree, the ABI is different.

Fix: point `SYSDIR` at the source tree that matches the running kernel. The Chapter 25 Makefile uses `/usr/src/sys` by default; on a 14.3 system with matching `/usr/src`, this is correct.

25.12.2 Symptom: `kldload` returns “No such file or directory” for an obvious-looking file

The file is present but the kernel’s module loader cannot parse it. Common causes: the file is a stale build artefact from a different machine, or the file is corrupt.

Check: file `myfirst.ko` should report it as an ELF 64-bit LSB shared object. If it reports anything else, rebuild from source.

25.12.3 Symptom: `kldload` succeeds but `kldstat` does not show the module

The loader decided to auto-unload the module. This happens when `MOD_LOAD` returned zero but `DRIVER_MODULE`’s `device_identify` did not find any device. For `myfirst`, which uses `nexus` as its parent, this should not happen; the pseudo-driver always finds `nexus`.

Check: `dmesg | tail -20` for any line like `module "myfirst" failed to register`. The message points at what went wrong.

25.12.4 Symptom: `kldload` reports “module busy”

A previous instance of the driver is still loaded and has an open file descriptor somewhere. The `MOD_QUIESCE` path in the old instance returned `EBUSY`.

Check: `fstat | grep myfirst` should show the process holding the fd. Kill the process or close the fd, then retry `kldunload`.

25.12.5 Symptom: `sysctl dev.myfirst.0.debug.mask=0x4` returns “Operation not permitted”

The caller is not root. `sysctls` with `CTLFLAG_RW` typically require root privilege unless they are explicitly marked otherwise.

Check: are you running as root? `sudo sysctl ...` or `su - first`.

25.12.6 Symptom: A new `sysctl` does not appear in the tree

`SYSCTL_ADD_*` either was not called, or was called with the wrong context/tree pointer. The most common bug is to use `SYSCTL_STATIC_CHILDREN` for a per-device OID instead of `device_get_sysctl_tree`.

Check: inside `myfirst_sysctl_attach`, confirm that `ctx = device_get_sysctl_ctx(dev)` and `tree = device_get_sysctl_tree(dev)` are used, and that every `SYSCTL_ADD_*` call passes `ctx` as the first argument.

25.12.7 Symptom: A tunable seems to be ignored

TUNABLE_*_FETCH runs at attach time, but only if the tunable is in the kernel environment at that moment. The common mistakes are (a) setting the tunable after the module is loaded, (b) typing the name wrong, (c) forgetting that kenv is not persistent.

Check: - kenv hw.myfirst.timeout_sec before reloading the module. The value should be what you expect. - The string passed to TUNABLE_INT_FETCH must match kenv exactly. A typo in one side is silent. - Setting via /boot/loader.conf requires a reboot (or a kldunload followed by kldload, which rereads loader.conf for the specific module's tunables).

25.12.8 Symptom: A log message should fire but does not appear in dmesg

Three common causes:

1. The debug-class bit is not set. Check `sysctl dev.myfirst.0.debug.mask`; the class bit must be enabled.
2. The rate-limit bucket is empty. If the message is emitted through `DLOG_RL`, the first few fire and the rest are suppressed silently. Set the pps cap higher via the `sysctl` or wait a second for the bucket to refill.
3. The message is emitted but filtered by the system's `sysctl kern.msgbuf_show_timestamp` setting or the `dmesg` buffer size (`sysctl kern.msgbuf_size`).

Check: `dmesg -c > /dev/null` to clear the buffer, reproduce the action, and re-read the buffer. The emptied buffer should contain only the driver's output.

25.12.9 Symptom: A log message appears once and is then silent forever

The rate-limit bucket permanently denies the message. This happens if `rl_curpps` becomes very large and the pps cap is very low. Check that `ppsratecheck` is being called with a stable `struct timeval` and a stable `int *` (both members of the `softc`); a per-call stack variable would be reset to zero every call and the algorithm would fire every time.

Check: the rate-limit state must be in the `softc` or another persistent location, not in local variables.

25.12.10 Symptom: Attach fails and the cleanup chain does not release a resource

The labelled-goto is missing a step or has a stray `return` that short-circuits the chain. Add a `device_printf(dev, "reached label fail_X\n")` at the top of each label and rerun the failure-injection lab. The label that does not print is the label that was skipped.

Common cause: an intermediate `return (error)` inserted for debugging and never removed. The compiler does not warn because the chain is still syntactically valid; the behaviour is wrong.

25.12.11 Symptom: Detach panics with a “witness” warning

A lock was destroyed while still held, or a lock was acquired after its owner was destroyed. The witness subsystem catches both. The backtrace points at the lock name, which maps to the `softc` field.

Check: the detach chain should be the exact reverse of attach. A common mistake is `mtx_destroy(&sc->sc_mtx)` called before `destroy_dev(sc->sc_cdev)`: the `cdev`'s callbacks can still be running, they try to take the lock, and the lock is gone. Fix: destroy the `cdev` first, then the lock.

25.12.12 Symptom: The driver panics at module unload with a dangling pointer

The `EVENTHANDLER_DEREGISTER` was not called, the kernel fired the event, and the callback pointer pointed at freed memory.

Check: for every `EVENTHANDLER_REGISTER` in attach, search detach for `EVENTHANDLER_DEREGISTER`. The count must match. If the count matches but the panic still happens, the tag stored in the `softc` was corrupted; audit the code path between registration and deregistration for memory scribbles.

25.12.13 Symptom: MYFIRSTIOC_GETVER returns an unexpected value

The `ioctl` version integer in `myfirst_ioctl.h` does not match what `myfirst_ioctl.c` writes into the buffer. This happens if the header is updated but the handler still returns a hard-coded constant.

Check: the handler should write `MYFIRST_IOCTL_VERSION` (the constant from the header), not a literal integer.

25.12.14 Symptom: devctl_notify events never appear in devd.log

`devd(8)` is not running, or its configuration does not match the event.

Check: `- service devd status` confirms the daemon is running. `- grep myfirst /etc/devd/*.conf` should find the rule. `- devd -Df` in the foreground prints every event as it arrives; reproduce the action and watch the output.

25.12.15 Symptom: The 100-cycle regression script grows the kernel's memory footprint

A resource is leaking per load/unload cycle. Common culprits: a `malloc` without a matching `free`, an `EVENTHANDLER_REGISTER` without a matching `deregister`, a `sysctl_ctx_init` that the driver calls manually without calling `sysctl_ctx_free` at detach (Chapter 25 uses `device_get_sysctl_ctx`, which is managed by `Newbus`; a driver that allocates its own context must free it).

Check: `vmstat -m | grep myfirst` before and after to see the driver's own memory consumption, and `vmstat -m | grep solaris` for kernel-level structures that the driver might be allocating indirectly.

25.12.16 Symptom: Two concurrent MYFIRSTIOC_SETMSG calls interleave their writes

The `softc` mutex is not being held around the update. The two threads are writing into `sc->sc_msg` simultaneously, producing a corrupt result.

Check: every access to `sc->sc_msg` and `sc->sc_msglen` in the `ioctl` handler must be inside `mtx_lock(&sc->sc_mtx) ... mtx_unlock(&sc->sc_mtx)`.

25.12.17 Symptom: A sysctl value resets on every module load

This is the intended behaviour, not a bug. The attach-time default is whatever the tunable evaluates to, which is either the `TUNABLE_INT_FETCH` default or the value set via `kenv`. The runtime `sysctl` write is lost at unload. If you want the value to persist, set it via `kenv` or `/boot/loader.conf`.

25.12.18 Symptom: MYFIRSTIOC_GETCAPS returns a value without the bits you just added

The `myfirst_ioctl.c` file was updated but not recompiled, or the wrong build was loaded. Also check that the handler in the switch statement uses the `|=` operator or a single assignment that includes every bit.

Check: `make clean && make` from the example directory. `kldstat -v | grep myfirst` to confirm the loaded module's path matches what you built.

25.12.19 Symptom: A SYSINIT fires before the kernel's allocator is ready

The `SYSINIT` is registered at too early a subsystem ID. Many subsystems (tunables, locks, early initcalls) are not allowed to call `malloc` with `M_NOWAIT` let alone `M_WAITOK`. If your callback calls `malloc` and the kernel panics at boot, look at the subsystem ID.

Check: the subsystem ID in `SYSINIT(...)`. For a callback that allocates memory, use `SI_SUB_DRIVERS` or later; do not use `SI_SUB_TUNABLES` or earlier.

25.12.20 Symptom: A handler registered with EVENTHANDLER_PRI_FIRST still runs late

`EVENTHANDLER_PRI_FIRST` is not a hard guarantee; it is a priority in a sorted queue. If another handler is also registered with `EVENTHANDLER_PRI_FIRST`, the order between them is undefined. The documented priorities are coarse; fine-grained ordering is not supported.

Check: accept that the priority is a hint, not a contract. If the driver absolutely requires running before or after a specific other handler, the design is wrong; restructure the driver so the ordering does not matter.

25.12.21 Symptom: dmesg does not show any output from the driver

The driver is using `printf` (the libc-like one) rather than `device_printf` or `DPRINTF`. Kernel `printf` still works but does not carry the device name, which makes messages hard to filter.

Check: every message in the driver should go through `device_printf(dev, ...)` or `DPRINTF(sc, class, fmt, ...)`. Bare `printf` is usually a mistake.

25.13 Quick Reference

The Quick Reference is a single-page summary of the macros, flags, and functions Chapter 25 introduced. Use it at the keyboard once the material is familiar.

25.13.1 Rate-limited logging

```

struct myfirst_ratelimit {
    struct timeval rl_lasttime;
    int          rl_curpps;
};

#define DLOG_RL(sc, rlp, pps, fmt, ...) do {                                \
    if (ppsratecheck(&(rlp)->rl_lasttime, &(rlp)->rl_curpps, pps)) \
        device_printf((sc)->sc_dev, fmt, ##__VA_ARGS__);          \
} while (0)

```

Use DLOG_RL for any message that can fire in a loop. Place the struct myfirst_ratelimit in the softc (not on the stack).

25.13.2 Errno vocabulary

Errno	Value	Use
0	0	success
EPERM	1	operation not permitted (root-only)
ENOENT	2	no such file
EBADF	9	bad file descriptor
ENOMEM	12	cannot allocate memory
EACCES	13	permission denied
EFAULT	14	bad address (user pointer)
EBUSY	16	resource busy
ENODEV	19	no such device
EINVAL	22	invalid argument
ENOTTY	25	inappropriate ioctl for device
ENOTSUP / EOPNOTSUPP	45	operation not supported
ENOIOCTL	-3	ioctl not handled by this driver (internal; kernel maps to ENOTTY)

25.13.3 Tunable families

```
TUNABLE_INT_FETCH("hw.myfirst.name",    &sc->sc_int_var);
TUNABLE_LONG_FETCH("hw.myfirst.name",   &sc->sc_long_var);
TUNABLE_BOOL_FETCH("hw.myfirst.name",   &sc->sc_bool_var);
TUNABLE_STR_FETCH("hw.myfirst.name",    sc->sc_str_var,
                                     sizeof(sc->sc_str_var));
```

Call each fetch once in attach after the default has been populated. The fetch updates the variable only if the tunable is present.

25.13.4 Sysctl flag summary

Flag	Meaning
CTLFLAG_RD	read-only
CTLFLAG_RW	read-write
CTLFLAG_TUN	cooperate with a loader tunable at attach time
CTLFLAG_RDTUN	shorthand for read-only + tunable
CTLFLAG_RWTUN	shorthand for read-write + tunable
CTLFLAG_MPSAFE	handler is MPSAFE
CTLFLAG_SKIP	hide the OID from default sysctl(8) listings

25.13.5 Version identifiers

- MYFIRST_VERSION: human-readable release string, e.g. "1.8-maintenance".
- MODULE_VERSION(myfirst, N): integer used by MODULE_DEPEND.
- MYFIRST_IOCTL_VERSION: integer returned by MYFIRSTIOC_GETVER; bumped only for wire-format breaks.

25.13.6 Capability bits

```
#define MYF_CAP_RESET      (1U << 0)
#define MYF_CAP_GETMSG     (1U << 1)
#define MYF_CAP_SETMSG     (1U << 2)
#define MYF_CAP_TIMEOUT    (1U << 3)

#define MYFIRSTIOC_GETCAPS _IOR('M', 5, uint32_t)
```


25.13.7 Labelled-cleanup skeleton

```
static int
myfirst_attach(device_t dev)
{
    struct myfirst_softc *sc = device_get_softc(dev);
    int error;

    /* acquire resources in order */
    mtx_init(&sc->sc_mtx, "myfirst", NULL, MTX_DEF);

    error = make_dev_s(...);
    if (error != 0)
        goto fail_mtx;

    myfirst_sysctl_attach(sc);

    error = myfirst_log_attach(sc);
    if (error != 0)
        goto fail_cdev;

    sc->sc_shutdown_tag = EVENTHANDLER_REGISTER(shutdown_pre_sync,
        myfirst_shutdown, sc, SHUTDOWN_PRI_DEFAULT);
    if (sc->sc_shutdown_tag == NULL) {
        error = ENOMEM;
        goto fail_log;
    }

    return (0);

fail_log:
    myfirst_log_detach(sc);
fail_cdev:
    destroy_dev(sc->sc_cdev);
fail_mtx:
    mtx_destroy(&sc->sc_mtx);
    return (error);
}
```

25.13.8 File layout for a modular driver

<code>driver.h</code>	public types
<code>driver.c</code>	module glue, <code>cdevsw</code> , <code>attach/detach</code>
<code>driver_cdev.c</code>	<code>open/close/read/write</code>
<code>driver_ioctl.h</code>	<code>ioctl</code> command numbers
<code>driver_ioctl.c</code>	<code>ioctl</code> dispatch
<code>driver_sysctl.c</code>	<code>sysctl</code> tree
<code>driver_debug.h</code>	<code>DPRINTF</code> macros
<code>driver_log.h</code>	rate-limit structures
<code>driver_log.c</code>	rate-limit helpers

25.13.9 Production checklist

- [] `MODULE_DEPEND` declared for every real dependency.
- [] `MODULE_PNP_INFO` declared if the driver binds to hardware.
- [] `MOD_QUIESCE` answers "can you unload?" without side effects.
- [] `devctl_notify` emitted for operator-relevant events.
- [] `MAINTENANCE.md` current.
- [] `devd.conf` snippet included.
- [] Every log message is `device_printf`, includes `errno`, and is rate-limited if it can fire in a loop.
- [] `attach/detach` survives 100 load/unload cycles.
- [] `sysctls` reject out-of-range values.
- [] `ioctl` payload is bounds-checked.
- [] Failure paths exercised via deliberate injection.
- [] Versioning discipline: three independent version identifiers, each bumped for its own reason.

25.13.10 SYSINIT subsystem IDs

Constant	Value	Use
<code>SI_SUB_TUNABLES</code>	<code>0x0700000</code>	establish tunable values
<code>SI_SUB_KLD</code>	<code>0x2000000</code>	KLD and module setup
<code>SI_SUB_SMP</code>	<code>0x2900000</code>	start the APs
<code>SI_SUB_DRIVERS</code>	<code>0x3100000</code>	let drivers initialise
<code>SI_SUB_CONFIGURE</code>	<code>0x3800000</code>	configure devices

Within a subsystem: - `SI_ORDER_FIRST` = `0x0` - `SI_ORDER_SECOND` = `0x1` - `SI_ORDER_MIDDLE` = `0x1000000` - `SI_ORDER_ANY` = `0xffffffff`

25.13.11 Shutdown event priorities

- SHUTDOWN_PRI_FIRST: run early.
- SHUTDOWN_PRI_DEFAULT: default.
- SHUTDOWN_PRI_LAST: run late.

25.13.12 EVENTHANDLER skeleton

```
sc->sc_tag = EVENTHANDLER_REGISTER(shutdown_pre_sync,
    my_handler, sc, SHUTDOWN_PRI_DEFAULT);
/* ... in detach ... */
EVENTHANDLER_DEREGISTER(shutdown_pre_sync, sc->sc_tag);
```

25.13.13 Tunable name ladders

Tunables and sysctls follow a hierarchical naming convention. The table below lists the nodes this chapter introduces:

Name	Kind	Purpose
hw.myfirst.debug_mask_default	tunable	initial debug mask for every instance
hw.myfirst.timeout_sec	tunable	initial operation timeout in seconds
hw.myfirst.max_retries	tunable	initial retry count
hw.myfirst.log_ratelimit_pps	tunable	initial messages-per-second cap
dev.myfirst.<unit>.version	sysctl (RD)	release string
dev.myfirst.<unit>.open_count	sysctl (RD)	active fd count
dev.myfirst.<unit>.total_read	sysctl (RD)	lifetime read calls
dev.myfirst.<unit>.total_write	sysctl (RD)	lifetime write calls
dev.myfirst.<unit>.messages	sysctl (RD)	current buffer contents
dev.myfirst.<unit>.messagesize	sysctl (RD)	current buffer length
dev.myfirst.<unit>.timeout	sysctl (RWTUN)	runtime timeout
dev.myfirst.<unit>.max_retries	sysctl (RWTUN)	runtime retry count
dev.myfirst.<unit>.log_ratelimit	sysctl (RWTUN)	runtime pps cap
dev.myfirst.<unit>.debug_mask	sysctl (RWTUN)	runtime debug mask
dev.myfirst.<unit>.debug_classes	sysctl (RD)	class names and bit values

Read the table as the interface contract. The `hw.myfirst.*` family is set at boot; the `dev.myfirst.*` family is adjusted at run time. Every writable entry has a matching read-only counterpart the operator can use to confirm the current value.

25.13.14 ioctl command ladder

Chapter 25's `ioctl` header defines these commands under magic 'M':

Command	Number	Direction	Purpose
MYFIRSTIOC_GETVER	0	read	returns MYFIRST_IOCTL_VERSION
MYFIRSTIOC_RESET	1	no-data	zero read/write counters
MYFIRSTIOC_GETMSG	2	read	copy current message out
MYFIRSTIOC_SETMSG	3	write	copy new message in
(retired)	4	n/a	reserved; do not reuse
MYFIRSTIOC_GETCAPS	5	read	capability bitmask

Adding a new command means picking the next unused number. Retiring a command means leaving its number in the ladder with (retired) next to it, not reusing the number.

25.14 Real-World Driver Walkthroughs

The chapter so far has built its discipline on the `myfirst` pseudo-driver. This section turns the lens around and looks at how the same disciplines appear in drivers that ship as part of FreeBSD 14.3. Each walkthrough starts from a real source file in `/usr/src`, names the Chapter 25 pattern that is in play, and points at the lines where the pattern is visible. The goal is not to document the specific drivers (their own documentation does that), but to show that Chapter 25's habits are not invented: they are the habits that working in-tree drivers already practise.

Reading real drivers with a pattern vocabulary is the single fastest way to accelerate your own judgement. Once you recognise `ppsratecheck` on sight, every driver that uses it becomes faster to read.

25.14.1 `mmc`sd(4): Rate-Limited Error Logging on a Hot Path

The `mmc`sd driver at `/usr/src/sys/dev/mmc/mmc.c` serves MMC and SD card storage. A filesystem above it generates a continuous stream of block I/O, and each block that fails the underlying MMC request produces a potential error log line. Without rate limiting, a slow or flaky card would flood `dmesg` in seconds.

The driver declares its rate-limit state `per_softc`, as the chapter recommends:

```
struct mmc_softc {
    ...
    struct timeval log_time;
    int          log_count;
    ...
};
```

`log_time` and `log_count` are the `ppsratecheck` state. Every hot path that emits a log message wraps the `device_printf` the same way:

```
#define LOG_PPS 5 /* Log no more than 5 errors per second. */

...

if (req.cmd->error != MMC_ERR_NONE) {
    if (ppsratecheck(&sc->log_time, &sc->log_count, LOG_PPS))
        device_printf(dev, "Error indicated: %d %s\n",
            req.cmd->error,
            mmcsc_errmsg(req.cmd->error));
    ...
}
```

The pattern is exactly the `DLOG_RL` shape the chapter introduced, with the macro expanded in place. `LOG_PPS` is set to 5 messages per second, and the state lives in the `softc` so repeated calls into the hot path share the same bucket.

Three observations worth taking away. First, this pattern is not theoretical: a shipping FreeBSD driver uses it on a hot path that can fire thousands of times per second. Second, the macro-versus-inline choice is a matter of taste; `mmcsc.c` open-codes the call, and the pattern is just as readable. Third, the `LOG_PPS` constant is conservative (5 per second); the author preferred fewer messages over more. A driver author can tune the pps cap to match the expected error rate and the operator's tolerance.

25.14.2 `uftdi(4)`: Module Dependencies and PNP Metadata

The `uftdi` driver at `/usr/src/sys/dev/usb/serial/uftdi.c` attaches to USB-serial adapters based on FTDI chips. It is a textbook example of a driver that depends on another kernel module: it cannot work without the USB stack.

Near the bottom of the file:

```
MODULE_DEPEND(uftdi, ucom, 1, 1, 1);
MODULE_DEPEND(uftdi, usb, 1, 1, 1);
MODULE_VERSION(uftdi, 1);
```

Two dependencies are declared. The first is on `ucom`, the generic USB-serial framework that `uftdi` builds on. The second is on `usb`, the USB core. Both are bounded to version 1 exactly. Loading `uftdi.ko` on a kernel without `usb` or `ucom` fails with a clear error; loading it on a kernel where the subsystem version has bumped past 1 also fails until `uftdi`'s own declaration is updated.

The PNP metadata is published through a macro that expands to `MODULE_PNP_INFO`:

```
USB_PNP_HOST_INFO(uftdi_devs);
```

USB_PNP_HOST_INFO is the USB-specific helper defined in `/usr/src/sys/dev/usb/usbd.h`. It expands to `MODULE_PNP_INFO` with the correct format string for USB vendor/product tuples. `uftdi_devs` is a static array of `struct usb_device_id` entries, one per (vendor, product, interface) triple the driver handles.

This is the production-readiness pattern from Chapter 25 Section 7 applied to a real hardware driver: dependencies declared, metadata published, version integer present. A new USB serial adapter appearing on a system causes `devd(8)` to consult the PNP metadata, identify `uftdi` as the driver, and load it if it is not already loaded. The mechanism is fully automatic once the metadata is right.

Chapter 26's version of `myfirst` will use the same pattern and the same helper.

25.14.3 `iscsi(4)`: Shutdown Handler Registered in `attach`

The iSCSI initiator at `/usr/src/sys/dev/iscsi/iscsi.c` holds open connections to remote storage targets. When the system shuts down, the initiator must close those connections gracefully before the network layer is torn down; otherwise the remote end is left with stale sessions.

The shutdown handler is registered at `attach`:

```
sc->sc_shutdown_pre_eh = EVENTHANDLER_REGISTER(shutdown_pre_sync,  
        iscsi_shutdown_pre, sc, SHUTDOWN_PRI_FIRST);
```

Two details are important. First, the registration's tag is stored in the `softc` (`sc->sc_shutdown_pre_eh`), so the later deregistration can reference it. Second, the priority is `SHUTDOWN_PRI_FIRST`, not `SHUTDOWN_PRI_DEFAULT`: the iSCSI driver wants to close connections before anyone else begins shutdown work, because storage connections take time to close cleanly.

The deregistration happens in the `detach` path:

```
EVENTHANDLER_DEREGISTER(shutdown_pre_sync, sc->sc_shutdown_pre_eh);
```

One registration, one deregistration. The tag in the `softc` keeps them bound.

For `myfirst`, the chapter's demonstration used `SHUTDOWN_PRI_DEFAULT` because there is no good reason for the pseudo-driver to run early. Real drivers pick a priority based on what depends on what: drivers that must be quiet before other drivers pick `SHUTDOWN_PRI_FIRST`; drivers that depend on filesystems being intact pick `SHUTDOWN_PRI_LAST`. The priority is a design decision, and `iscsi` shows one way to make it.

25.14.4 `ufs_dirhash`: Cache Eviction on `vm_lowmem`

The UFS directory hash cache at `/usr/src/sys/ufs/ufs/ufs_dirhash.c` is a per-filesystem in-memory accelerator for directory lookups. Under normal operation the cache is beneficial; under memory pressure it becomes a liability, so the subsystem registers a `vm_lowmem` handler that discards cache entries:

```
EVENTHANDLER_REGISTER(vm_lowmem, ufsdirhash_lowmem, NULL,
    EVENTHANDLER_PRI_FIRST);
```

The fourth argument is `EVENTHANDLER_PRI_FIRST`, which asks to run early in the list of registered `vm_lowmem` handlers. The `dirhash` author picked early execution because the cache is pure reclaimable memory: freeing it promptly gives other handlers (which may hold dirtier or less easily released state) a better chance of succeeding.

The callback itself does the real work: walk the hash tables, release entries that can be released, account for the memory freed. The essential design point is that the callback does not panic if there is nothing to release; it simply returns having done nothing.

This is the Chapter 25 `vm_lowmem` pattern in a subsystem that is not a device driver but that shares the discipline. The lessons transfer: if `myfirst` ever gets a cache, the shape is already here.

25.14.5 tcp_subr: vm_lowmem Without a Softc

The TCP subsystem at `/usr/src/sys/netinet/tcp_subr.c` also registers for `vm_lowmem`, but its registration is different in an instructive way:

```
EVENTHANDLER_REGISTER(vm_lowmem, tcp_drain, NULL, LOWMEM_PRI_DEFAULT);
```

The third argument (the callback data) is `NULL`, not a `softc` pointer. The TCP subsystem does not have a single `softc`; its state is scattered across many structures. The callback has to find its state by other means (globals, per-CPU variables, hash table lookups).

This raises a question the chapter hinted at: when is it acceptable to pass `NULL` as the callback argument? The answer is: when the callback has another way to find its state. For a driver with a per-device `softc`, passing `sc` is almost always the right choice. For a subsystem with global state, passing `NULL` and letting the callback use its known globals is fine.

`myfirst` will always pass `sc` because `myfirst` is a per-device driver. A reader who finds themselves writing a subsystem-level callback should recognise that the pattern changes in subtle ways when the subject is global.

25.14.6 if_vtnet: Rate-Limited Logging Around a Specific Fault

The VirtIO network driver at `/usr/src/sys/dev/virtio/network/if_vtnet.c` provides a narrower but instructive counterpoint to `mmcsd`. Where `mmcsd` wraps every hot-path error in rate-limited logging, `if_vtnet` reaches for `ppsratecheck` only around a specific misbehaviour: a TSO packet with ECN bits set that the VirtIO host has not negotiated. The call site is small and self-contained:

```
static struct timeval lastecn;
static int curecn;
...
if (ppsratecheck(&lastecn, &curecn, 1))
```

```
if_printf(sc->vtnet_ifp,  
        "TSO with ECN not negotiated with host\n");
```

Two details are worth pointing out. First, the rate-limit state is declared `static` at file scope, not `per-softc`. The author decided that “the VirtIO host disagrees with the guest about ECN” is a system-level misconfiguration rather than a per-interface fault, so one shared bucket is enough; `per-softc` state would let a single virtual interface’s flood starve another’s. Second, the cap is 1 pps, which is deliberately aggressive: the warning is informational and should appear no more than once per second across the entire system. A driver designer who expected the warning to fire often could raise the cap.

FreeBSD also provides `ratecheck(9)`, the event-count sibling of `ppsratecheck`. `ratecheck` fires when the time since the last allowed event exceeds a threshold; `ppsratecheck` fires when the recent burst rate is below a cap. They are complementary: `ratecheck` is better when you want a minimum interval between messages, while `ppsratecheck` is better when you want a maximum burst rate.

The takeaway from `if_vtnet` is that rate-limit state can be global or per-instance, and the choice follows the expected shape of the failure. Chapter 25 put the state in the `softc` because `myfirst`’s errors are per-instance; a different driver might reasonably make a different choice.

25.14.7 vt_core: Multiple EVENTHANDLER Registrations in Sequence

The virtual terminal subsystem at `/usr/src/sys/dev/vt/vt_core.c` registers multiple event handlers at different points in its lifecycle. The shutdown handler for the console window is one:

```
EVENTHANDLER_REGISTER(shutdown_pre_sync, vt_window_switch,  
        vw, SHUTDOWN_PRI_DEFAULT);
```

`SHUTDOWN_PRI_DEFAULT` is the neutral priority: the VT switch runs after anything that asked for `SHUTDOWN_PRI_FIRST` and before anything that asked for `SHUTDOWN_PRI_LAST`. The choice is deliberate: the terminal switch has no ordering requirement against other subsystems, so the author picked the default rather than claiming a priority the driver did not need.

The point for our purposes is that `vt_core` registers this handler once, at boot time, and never deregisters it. A driver whose lifetime is the kernel’s lifetime does not need to deregister; the kernel never calls the handler after it is gone. A driver like `myfirst` that can be loaded and unloaded as a module does need to deregister, because unloading the module destroys the handler’s code. The rule is: deregister if your code can disappear while the kernel keeps running.

This distinction is important for module authors. Built-in code paths often look like they are missing their deregistration, but they are not; they just do not need it. A module that follows the built-in pattern verbatim will panic on unload.

25.14.8 FFS (`ffs_alloc.c`): Per-Condition Rate-Limit Buckets

The FFS allocator at `/usr/src/sys/ufs/ffs/ffs_alloc.c` is a filesystem rather than a device driver, but it faces exactly the log-flood problem Chapter 25 is about. A disk that repeatedly runs out of blocks or inodes can

emit one error per failed `write(2)` call, which is unbounded in practice. The allocator reaches for `ppsrate-check` at four distinct sites, and the way it scopes the rate-limit state is a good lesson in bucket design.

Each mounted filesystem carries rate-limit state in its mount structure. Two separate buckets handle two separate kinds of error:

```
/* "Filesystem full" reports: blocks or inodes exhausted. */
um->um_last_fullmsg
um->um_secs_fullmsg

/* Cylinder-group integrity reports. */
um->um_last_integritymsg
um->um_secs_integritymsg
```

The “filesystem full” bucket is shared across two code paths (`ffs_alloc` and `ffs_realloccg`, both writing a message like “write failed, filesystem is full”) plus inode exhaustion. The integrity bucket is shared across two different integrity failures (cylinder checkhash mismatch and magic-number mismatch). At each call site the shape is identical:

```
if (ppsratecheck(&ump->um_last_fullmsg,
    &ump->um_secs_fullmsg, 1)) {
    UFS_UNLOCK(ump);
    ffs_fserr(fs, ip->i_number, "filesystem full");
    uprintf("\n%s: write failed, filesystem is full\n",
        fs->fs_fsmnt);
    ...
}
```

Three design decisions are visible. First, the rate-limit state is per mount point, not global. A heavy-write filesystem should not suppress messages from a different filesystem that is also filling up. Second, the cap is 1 pps: at most one message per second per mount per bucket. Third, related messages share a bucket (all “fullness” messages; all “integrity” messages), while unrelated ones have their own. The author of `ffs_alloc.c` decided that an operator who is flooded with “filesystem full” should not also be flooded with “out of inodes” for the same mount point; the two are symptoms of the same condition and one message per second is enough.

A pseudo-driver like `myfirst` can borrow the pattern directly. If `myfirst` someday grows a class of errors related to capacity (say, “buffer full” for the write path and “no free slots” for the `ioctl` path), those belong in the same bucket. A completely different failure (say, “command version mismatch”) deserves its own. The discipline that `ffs_alloc.c` applies to a filesystem carries over unchanged to a device driver.

25.14.9 Reading More Drivers

Every FreeBSD driver is a case study in how a specific team solved the problems Chapter 25 names. A few areas of `/usr/src/sys` are particularly rich for pattern-hunting:

- `/usr/src/sys/dev/usb/` for USB drivers: `MODULE_DEPEND` and `MODULE_PNP_INFO` everywhere.

- `/usr/src/sys/dev/pci/` for PCI drivers: labelled-cleanup attach routines at industrial scale.
- `/usr/src/sys/dev/cxgbe/` for a complex modern driver: rate-limited logging, sysctl trees with hundreds of OIDs, versioning via module ABI.
- `/usr/src/sys/netinet/` for subsystem-level EVENTHANDLER usage.
- `/usr/src/sys/kern/subr_*.c` for examples of SYSINIT at many different subsystem IDs.

When you read a new driver, start by finding its attach function. Count the acquisitions and the labels; they should match. Find the detach function. Confirm it releases the resources in reverse order. Find the error paths and see which `errno` each one returns. Look for `ppsratecheck` or `ratecheck` near any log message that fires on a hot path. Look for `MODULE_DEPEND` declarations. Look for `EVENTHANDLER_REGISTER` and confirm there is a matching `deregister`.

Every one of these inspections takes seconds once the patterns are familiar. Each one strengthens your own instinct for when a pattern is or is not being applied correctly.

25.14.10 What You Will Not See

A few patterns that Chapter 25 recommends do not appear in every driver, and their absence is not always a bug. Knowing which patterns are optional keeps you from expecting them everywhere.

`MAINTENANCE.md` is a habit recommended by this book, not a FreeBSD requirement. Most in-tree drivers do not ship a per-driver maintenance file; instead, the manual page is the operator-facing reference, and the release notes carry the change log. Both solutions work; the choice between them is a project convention.

`devctl_notify` is optional. Many drivers do not emit any events, and there is no rule that they must. The pattern is valuable when there are events that operators would actually want to react to; for drivers with quiet behaviour and no operator-visible state changes, emitting events is unnecessary.

`SYSINIT` outside of `DRIVER_MODULE` is uncommon in modern drivers. Most driver-level work happens in `device_attach`, which runs per-instance. Explicit `SYSINIT` registrations are most common in subsystems and core kernel code; individual drivers rarely need one. Chapter 25 introduced `SYSINIT` because the reader will meet it eventually, not because most drivers use it.

The explicit module event handler (a driver-supplied `MOD_LOAD/MOD_UNLOAD` function instead of `DRIVER_MODULE`) is also uncommon. It is there when a driver needs custom behaviour at load time that does not fit the Newbus model, but most drivers happily use the default.

When you read a driver that omits one of these patterns, the absence usually reflects a design decision specific to that driver, not a failure of discipline. The patterns are tools; not every tool is needed on every job.

25.14.11 Wrapping Up the Walkthroughs

Every pattern the chapter introduced is visible somewhere in the FreeBSD source tree. The `mmc` driver rate-limits its hot-path logs. The `uftdi` driver declares its module dependencies and PNP metadata. The `iscsi` driver registers a prioritised `shutdown_pre_sync` handler. The UFS dirhash cache releases memory on `vm_lowmem`. Each of these is a real, shipping, tested application of a Chapter 25 discipline.

Reading these drivers with a pattern vocabulary accelerates your own intuition faster than any single-author textbook can. The patterns repeat; the drivers differ. Once you can name the pattern, every new driver you read reinforces it.

The `myfirst` driver at the end of Chapter 25 wears every discipline. You have just seen the same disciplines worn by eight other drivers. The next step is to wear them yourself, on a driver that is not in this book. That is the work of a career, and the foundations are now in your hands.

25.15 Wrapping Up

Chapter 25 is the longest chapter in Part 5, and its length is deliberate. Every section introduced a discipline that turns a working driver into a maintainable one. None of the disciplines are glamorous; each one is the specific habit that keeps a driver working when it is used by people other than its author.

The chapter began with rate-limited logging. A driver that cannot spam the console is a driver whose log messages are worth reading. `ppsratecheck(9)` and the `DLOG_RL` macro make the discipline mechanical: place the rate-limit state in the softc, wrap hot-path messages with the macro, and let the kernel do the per-second bookkeeping. Everything downstream of logging benefits, because a log you can read is a log you can debug from.

The second section named `errno` values and told them apart. `ENOTTY` is not `EINVAL`; `EPERM` is not `EACCES`; `ENOIOCTL` is the kernel's special signal that a driver did not recognise an `ioctl` command and wants the kernel to try another handler before giving up. Knowing the vocabulary turns vague bug reports into precise ones, and precise bug reports reach root causes faster.

Section 3 treated configuration as a first-class concern. A tunable is a boot-time initial value; a `sysctl` is a runtime handle. The two cooperate through `TUNABLE_*_FETCH` and `CTLFLAG_TUN`. A driver that exposes exactly the right tunables and `sysctls` gives operators enough control to solve their own problems without having to modify the source. The `hw.myfirst.timeout_sec` and `dev.myfirst.0.timeout_sec` pair is now the template every future tunable will follow.

Section 4 named the three version identifiers a driver needs (release string, module integer, `ioctl` wire integer) and pointed out that they change for different reasons. The `GETCAPS` `ioctl` gives user-space a run-time discovery mechanism that survives the addition and removal of capabilities over time, and the `MODULE_DEPEND` macro makes inter-module compatibility explicit. The `myfirst` driver's own jump from 1.7 to 1.8 is a small but complete case study.

Section 5 named the labelled-goto pattern and made it the unconditional discipline for attach and detach. Every resource gets a label; every label falls through to the next; the single `return (0)` separates the acquisition chain from the cleanup chain. The pattern scales from two resources to a dozen without changing shape. Deliberate failure injection confirms that every label is reachable.

Section 6 split `myfirst` into subject-matter files. One file per concern, a single responsibility rule for deciding where to put new code, and a small public header that carries only the declarations every file needs. The driver is now at phase two of the typical multi-file split, and the rules for reaching phase three are named explicitly.

Section 7 turned the focus outward. The production-readiness checklist covered module dependencies, PNP metadata, `MOD_QUIESCE`, `devctl_notify`, `MAINTENANCE.md`, `devd` rules, log message quality, the 100-cycle regression test, input validation, failure-path exercise, and versioning discipline. Every item on the checklist is a specific habit that catches a specific failure mode. `myfirst` at the end of Chapter 25 passes every item.

Section 8 closed the loop with `SYSINIT(9)` and `EVENTHANDLER(9)`. A driver that needs to participate in the kernel lifecycle beyond its own attach/detach window has a clean mechanism for doing so. `myfirst` registers

for `shutdown_pre_sync` as a demonstration; other drivers register for `vm_lowmem`, `suspend/resume`, or custom events.

The driver has grown from a single file with one message buffer into a modular, observable, version-disciplined, rate-limited, and production-ready pseudo-device. The mechanics of FreeBSD character-driver authorship are now in the reader's hands.

What has not been covered yet is the world of hardware. Every chapter so far has used a pseudo-driver; the character device backs a software buffer and a set of counters. A driver that actually talks to hardware has to allocate bus resources, map I/O windows, attach interrupt handlers, program DMA engines, and survive the specific failure modes of real devices. Chapter 26 starts that work.

25.16 Part 5 Checkpoint

Part 5 covered debugging, tracing, and the engineering practices that separate a driver that merely compiles from a driver that a team can maintain. Before Part 6 changes the terrain by attaching `myfirst` to real transports, confirm that the Part 5 habits are in your fingers and not just in your notes.

By the end of Part 5 you should be comfortable doing each of the following:

- Investigating driver behaviour with the right tool for the question: `ktrace` and `kdump` for per-process system calls, `ddb` for live or post-mortem kernel inspection, `kgdb` with a core dump for crashed kernels, `dtrace` and SDT probes for in-kernel tracing without source changes, and `procstat` together with `lockstat` for state and contention views.
- Reading a panic message and extracting the right breadcrumbs, turning a stack trace into a reproducible test case rather than a guessing game.
- Building and running the Chapter 24 integration stack `myfirst`: a driver that exposes its surfaces through `ioctl`, `sysctl`, DTrace probes, and `devctl_notify` while exercising a full lifecycle injection path.
- Applying the Chapter 25 production-readiness checklist item by item: `MODULE_DEPEND`, PNP metadata, `MOD_QUIESCE` as a separate handler, `devctl_notify`, `MAINTENANCE.md`, `devd` rules, log message quality, the 100-cycle regression test, input validation, deliberate failure-path exercise, and version discipline.
- Using `SYSINIT(9)` and `EVENTHANDLER(9)` when a driver must participate in the kernel lifecycle beyond its own attach and detach window, including graceful registration and deregistration.

If any of these still feels soft, the labs to revisit are:

- Debugging tools in practice: Lab 23.1 (A First DDB Session), Lab 23.2 (Measuring the Driver with DTrace), Lab 23.4 (Finding a Memory Leak with `vmstat -m`), and Lab 23.5 (Installing the 1.6-debug Refactor).
- Integration and observability: Lab 1 (Build and Load the Stage 1 Driver), Lab 4 (Walk the Lifecycle by Injecting a Failure), Lab 5 (Trace the Integration Surfaces with DTrace), and Lab 6 (Integration Smoke Test) in Chapter 24.
- Production discipline: Lab 3 (Tunable Reboot Behaviour), Lab 4 (Deliberate Attach Failure Injection), Lab 6 (The 100-Cycle Regression Script), and Lab 10 (Multi-Version Compatibility Matrix) in Chapter 25.

Part 6 expects the following as a baseline:

- Confidence reading a panic and following it to a root cause, since three new transports will introduce three new ways for things to go wrong.
- A `myfirst` that passes the Chapter 25 production checklist, so that Chapter 26 can add USB without the earlier discipline wobbling under new load.
- Awareness that Part 6 changes the running-example pattern: the `myfirst` thread will extend through Chapter 26 as `myfirst_usb`, then pause while Chapters 27 and 28 use fresh, transport-shaped demos. The discipline continues; only the code artefact changes. Chapter 29 returns to the cumulative shape.

If those hold, you are ready for Part 6. If one still looks unsteady, the fix is a single well-chosen lab, not a rush forward.

25.17 Bridge to Chapter 26

Chapter 26 turns the `myfirst` driver outward. Instead of serving a buffer in RAM, the driver will attach to a real bus and service a real device. The book's first hardware target is the USB subsystem: USB is ubiquitous, well-documented, and has a clean kernel interface that is easier to start with than PCI or ISA.

The habits you have built in Chapter 25 carry over unchanged. The labelled-goto pattern in `attach` scales to bus resource allocations: `bus_alloc_resource`, `bus_setup_intr`, `bus_dma_tag_create`, each becomes another acquisition in the chain, each with its own label. The modular file layout extends naturally: `myfirst_usb.c` joins `myfirst.c`, `myfirst_cdev.c`, and the rest. The production checklist adds two items: `MODULE_DEPEND(myfirst_usb, usb, ...)` and `MODULE_PNP_INFO` for the vendor/product identifiers the driver handles. The versioning discipline continues; the Chapter 26 driver will be `1.9-usb`.

What is genuinely new in Chapter 26 is not the driver's structure; it is the interfaces between the driver and the USB subsystem. You will meet `usb_request_methods`, transfer setup callbacks, the separation between control, bulk, interrupt, and isochronous transfers, and the USB-specific lifecycle (`attach` is per-device, just like Newbus; `detach` is per-device, just like Newbus; hot-plug is the normal operating condition, unlike most other buses).

Before you start Chapter 26, pause and confirm that the Chapter 25 material is solid. The labs are meant to be run; running them is the single best way to find the parts of the chapter that did not land. If you have run Labs 1 through 7 and the challenge exercises are clear, you are ready for hardware.

Chapter 26 begins with a simple USB device (a FTDI serial adapter, most likely) and walks through probe, attach, bulk transfer setup, read/write dispatch, and the detach/hot-unplug paths. By the end of Chapter 26, `myfirst_usb.ko` will serve real bytes from a real cable, and the discipline from Chapter 25 will keep it maintainable.

25.18 Glossary

The terms in this glossary all appear in Chapter 25 and are worth knowing by name. Definitions are short on purpose; the chapter body is where each concept is developed.

Attach chain. The ordered sequence of resource acquisitions in a driver's `device_attach` method. Each acquisition that can fail is followed by a goto to the label that undoes the previously acquired resources.

Capability bitmask. A 32-bit (or 64-bit) integer returned by MYFIRSTIOC_GETCAPS, with one bit per optional feature. Lets user-space query the driver at run time for which features it supports.

Cleanup chain. The ordered sequence of labels at the bottom of a driver's device_attach method. Each label releases one resource and falls through to the next. Reverse of the acquisition order.

CTLFLAG_SKIP. A sysctl flag that hides an OID from default sysctl(8) listings. The OID is still readable when its full name is given explicitly. Defined in /usr/src/sys/sys/sysctl.h.

CTLFLAG_RDTUN / CTLFLAG_RWTUN. Shorthand for CTLFLAG_RD | CTLFLAG_TUN and CTLFLAG_RW | CTLFLAG_TUN respectively. Declares a sysctl that cooperates with a loader tunable.

devctl_notify. Kernel function that emits a structured event readable by devd(8). Lets a driver notify user-space daemons of interesting state changes.

DLOG_RL. Macro wrapper over ppsratecheck and device_printf that caps a log message at a given messages-per-second rate.

Errno. A small positive integer representing a specific failure mode. FreeBSD's errno table is in /usr/src/sys/sys/errno.h.

Event handler. A callback registered with EVENTHANDLER_REGISTER that runs whenever the kernel fires a named event. Deregistered with EVENTHANDLER_DEREGISTER.

EVENTHANDLER(9). FreeBSD's generic event-notification framework. Defines events, lets subsystems publish them, lets drivers subscribe.

Failure injection. A deliberate test technique that causes a code path to fail in order to exercise its cleanup. Usually implemented as a conditional return guarded by an #ifdef.

Labelled-goto pattern. See "attach chain" and "cleanup chain". The idiomatic FreeBSD shape for attach and detach that uses goto label; for linear unwinding rather than nested if.

MOD_QUIESCE. The module event that asks "can you be unloaded right now?". A driver returns EBUSY to refuse; 0 to accept.

MODULE_DEPEND. Macro that declares a dependency on another kernel module. The kernel enforces load order and version compatibility.

MODULE_PNP_INFO. Macro that publishes the vendor/product identifiers a driver handles. The kernel uses the metadata to auto-load drivers when matching hardware appears.

MODULE_VERSION. Macro that declares a module's version integer. Used by MODULE_DEPEND for compatibility checks.

myfirst_ratelimit. The struct holding the per-class rate-limit state (lasttime and curpps). Must live in the softc, not on the stack.

MYFIRST_VERSION. The driver's human-readable release string, e.g. "1.8-maintenance". Exposed via dev.myfirst.<unit>.version.

MYFIRST_IOCTL_VERSION. The driver's ioctl wire-format version integer. Returned by MYFIRSTIOC_GETVER. Bumped only for breaking changes.

Pps. Events per second. A rate-limit cap expressed in pps (e.g. 10 pps = 10 messages per second).

ppsratecheck(9). The FreeBSD rate-limit primitive. Takes a struct `timeval`, an `int *`, and a pps cap; returns nonzero to allow the event.

Production-ready. A driver that satisfies the Section 7 checklist: declared dependencies, documented surface, rate-limited logging, exercised failure paths, passed the 100-cycle test.

SHUTDOWN_PRI*. Priority constants passed to `EVENTHANDLER_REGISTER` for shutdown events. `FIRST` runs early; `LAST` runs late; `DEFAULT` runs in the middle.

SI_SUB*. Subsystem identifiers for `SYSINIT`. Numeric values sort by boot order. Common constants: `SI_SUB_TUNABLES`, `SI_SUB_DRIVERS`, `SI_SUB_CONFIGURE`.

SI_ORDER*. Order constants for `SYSINIT` within a subsystem. `FIRST` runs first; `MIDDLE` runs in the middle; `ANY` runs last (no guaranteed order among `ANY` entries).

Single-responsibility rule. Each source file answers one question about the driver. Violations mean new `ioctl`s creeping into `myfirst_sysctl.c` or new `sysctls` creeping into `myfirst_ioctl.c`.

Softc. The per-device state structure. Newbus allocates it, zeroes it, and hands it to `device_attach` via `device_get_softc`.

sysctl(8). The user-space command and kernel interface for runtime parameters. Node names live under a fixed hierarchy (`kern.*`, `hw.*`, `net.*`, `dev.*`, `vm.*`, `debug.*`, etc.).

SYSINIT(9). FreeBSD's boot-time initialisation macro. Registers a function to run at a specific subsystem and order during kernel startup or module load.

SYSUNINIT(9). Companion to `SYSINIT`. Registers a cleanup function to run at module unload in the reverse of the `SYSINIT` order.

Tag (event handler). The opaque value returned by `EVENTHANDLER_REGISTER` and consumed by `EVENTHANDLER_DEREGISTER`. Must be stored (usually in the `softc`) so the deregistration can locate the registration.

Tunable. A value parsed from the kernel environment at boot or module load and consumed via `TUNABLE_*_FETCH`. Sets initial values; lives at the `hw.`, `kern.`, or `debug.` level.

TUNABLE_INT_FETCH, _LONG_FETCH, _BOOL_FETCH, _STR_FETCH. The family of macros that read a tunable from the kernel environment and populate a variable. Silent if the tunable is absent.

Version split. The practice of using three independent version identifiers (release string, module integer, `ioctl` integer) that change for different reasons.

vm_lowmem event. An `EVENTHANDLER` event fired when the virtual memory subsystem is under pressure. Drivers with caches can release some memory back.

Wire format. The layout of data that crosses a user/kernel boundary. The `ioctl` wire format is determined by the `_IOR`, `_IOW`, `_IOWR` declarations and the payload structure. A wire-format change is a breaking change and requires a `MYFIRST_IOCTL_VERSION` bump.

26 USB and Serial Drivers

26.1 Introduction

Chapter 25 closed Part 5 with a driver that the rest of the system could talk to. The `myfirst` driver at version 1.8-maintenance had a rate-limited logging macro, a careful `errno` vocabulary, loader tunables and writable sysctls, a three-way version split, a labelled-goto cleanup chain in `attach` and `detach`, a clean modular source layout, `MODULE_DEPEND` and `MODULE_PNP_INFO` metadata, a `MAINTENANCE.md` document, a `shutdown_pre_sync` event handler, and a regression script that could load and unload the driver a hundred times without leaking a single resource. What the driver did not have was any contact with real hardware. The character device backed a buffer in kernel memory. The sysctl counters tracked operations against that buffer. The `MYFIRSTIOC_GETCAPS` ioctl announced capabilities that were implemented entirely in software. Everything the driver did, it did without ever reading a byte off a wire.

Chapter 26 begins the step outward. Instead of serving a buffer in RAM, the driver will attach to a real bus and service a real device. The bus will be the Universal Serial Bus, because USB is the most approachable transport in FreeBSD: it is ubiquitous, its subsystem is extremely well organised, the kernel interface is designed around a small handful of structures and macros, and every FreeBSD developer already has a dozen USB devices on their desk. After USB, the chapter pivots to the subject that historically preceded USB and still lives alongside it everywhere from debug consoles to GPS modules: the serial port, in its classical form as a UART-driven RS-232 interface and in its modern form as a USB-to-serial bridge. By the end of the chapter, the `myfirst` driver family has grown a new transport-specific sibling, `myfirst_usb`, at version 1.9-usb. That sibling knows how to attach to a real USB device, how to set up a bulk-in and a bulk-out transfer, how to echo received bytes through a `/dev` node, and how to survive the device being yanked out of the port while the driver is in use.

Chapter 26 is the opening chapter of Part 6. Part 6 is organised around the observation that up to this point the book has been teaching the parts of FreeBSD driver development that are *transport-neutral*: the Newbus model, the character device interface, synchronisation, interrupts, DMA, power management, debugging, integration, and maintenance. All of those disciplines apply to every driver regardless of how the device is attached. Part 6 shifts the focus. USB, storage, and networking each have their own bus, their own lifecycle, their own data-flow pattern, and their own idiomatic way of integrating with the rest of the kernel. The disciplines you have built in Parts 1 through 5 carry over unchanged; what is new is the shape of the interface between your driver and the specific subsystem it plugs into. Chapter 26 teaches that shape for USB and for serial devices. Chapter 27 teaches it for storage devices and the VFS layer. Chapter 28 teaches it for network interfaces. Each of the three chapters is structurally parallel: the transport is introduced, the subsystem is mapped, a minimal driver is built, and the reader is shown how to test without the specific hardware everyone happens not to have.

There is a deliberate pairing of USB and serial in this chapter. The two topics sit together because they are both first-class citizens of the same larger mental model: a transport is a *protocol plus a lifecycle*, and the driver is the piece of code that carries data across the protocol boundary and keeps the lifecycle consistent with the kernel's view of the device. USB is a protocol with a rich four-transfer-type vocabulary and a hot-plug lifecycle. A

UART is a protocol with a much simpler byte-framing vocabulary and a statically-attached lifecycle. Studying them together makes the pattern visible. A student who has seen the USB callback state machine and the UART interrupt handler side by side understands that “FreeBSD’s driver model” is not a single shape but a family of shapes, each one adapted to the demands of its own transport.

The second reason to pair USB and serial is historical and practical. A very large number of what the operating system calls “USB devices” are in fact serial ports in disguise. The FTDI FT232R chip, the Prolific PL2303, the Silicon Labs CP210x, and the WCH CH340 all expose a standard serial-port API to user space, but physically they sit on the USB bus. FreeBSD handles that with the `ucom(4)` framework: a USB driver registers callbacks with `ucom`, and `ucom` produces the user-visible `/dev/ttyU0` and `/dev/cuaU0` device nodes and arranges for the `termios`-aware line discipline to operate correctly on top of a USB bulk-in and bulk-out pair. The reader who is about to write a USB driver is likely, sooner or later, to write a USB-to-serial driver, and that driver will be an intersection of the two worlds the chapter introduces. Putting the material into a single chapter makes the intersection visible.

A third reason is pedagogical. The `myfirst` driver so far has been a pseudo-device. The transition to real hardware is a conceptual step, not just a coding step. Many readers will find their first attempt at a hardware-backed driver unsettling: interrupts arrive without asking, transfers can stall or time out, the device can be unplugged mid-operation, and the kernel has opinions about how fast you are allowed to respond. USB is the friendliest possible introduction to that world because the USB subsystem does an unusually large amount of work on the driver’s behalf. Setting up a bulk transfer in USB is not the same kind of problem as setting up a DMA ring on a PCI Express NIC. The USB core manages the low-level DMA bookkeeping; your driver works at the level of “tell me when this transfer completes”. Learning the USB pattern first makes the later hardware chapters (storage, networking, embedded buses in Part 7) less intimidating because by then the basic shape of a transport-specific driver is familiar.

The `myfirst` driver’s path through this chapter is concrete. It picks up at version 1.8-maintenance from the end of Chapter 25. It adds a new source file, `myfirst_usb.c`, compiled into a new kernel module, `myfirst_usb.ko`. The new module declares itself dependent on `usb`, lists a single vendor and product identifier in its match table, probes and attaches on hot-plug, allocates one bulk-in and one bulk-out transfer, exposes a `/dev/myfirst_usb0` node, echoes incoming bytes to the kernel log and copies them back out on a read, handles detach cleanly when the cable is pulled, and carries forward every Chapter 25 discipline without exception. The labs exercise each piece in turn. By the end of the chapter, there is a second driver in the family, a new source layout to accommodate it, and a working example of a FreeBSD USB driver that the reader has typed themselves.

Because this is also a chapter about serial devices, the chapter spends time on the serial half of its scope even though `myfirst_usb` itself is not a serial driver. The serial material teaches how `uart(4)` is laid out, how `ucom(4)` fits in, how `termios` carries baud rate and parity and flow control from user space down to hardware, and how to test serial interfaces without physical hardware using `nmdm(4)`. The serial material does not build a new UART hardware driver from scratch. Writing a UART hardware driver is a specialised undertaking that is almost never the right choice in a modern environment: the existing `ns8250` driver in the base system already handles every PC-compatible serial port, every common PCI serial card, and the ARM PL011 that most virtualised platforms present. The chapter teaches the serial subsystem at the level the reader actually needs: how it is organised, how to read existing drivers, how `termios` reaches a driver’s `param` method, how to use the subsystem from user space, and what to do when the goal is a USB-to-serial driver (the common case) rather than a new hardware driver (the rare case).

The rhythm of Chapter 26 is the rhythm of pattern recognition. The reader will leave the chapter knowing what a USB driver looks like, what a serial driver looks like, where the two overlap, where they differ from the pseudo-drivers of Parts 2 through 5, and how to test both without a lab full of adapters. Those are the foundations of transport-specific driver development. Chapter 27 will then apply the same discipline to storage, and Chapter 28 will apply it to networking, each time taking the same general pattern and bending it to a new transport's rules.

26.1.1 Where Chapter 25 Left the Driver

A short checkpoint before the new work starts. Chapter 26 extends the driver family produced at the end of Chapter 25, tagged as version 1.8-maintenance. If any of the items below is uncertain, return to Chapter 25 and resolve it before starting this chapter, because the new material assumes every Chapter 25 primitive is working and every habit is in place.

- Your driver source matches Chapter 25 Stage 4. `myfirst.ko` compiles cleanly, identifies itself as 1.8-maintenance in `kldstat -v`, and carries the full `MYFIRST_VERSION`, `MODULE_VERSION`, and `MYFIRST_IOCTL_VERSION` triple.
- The source layout is split: `myfirst_bus.c`, `myfirst_cdev.c`, `myfirst_ioctl.c`, `myfirst_sysctl.c`, `myfirst_debug.c`, `myfirst_log.c`, with `myfirst.h` as the shared private header.
- The rate-limited log macro `DLOG_RL` is in place and tied to a struct `myfirst_ratelimit` inside the `softc`.
- The `goto fail`; cleanup chain in `myfirst_attach` is working and exercised by a deliberate failure lab.
- The regression script passes a hundred consecutive `kldload/kldunload` cycles with no residual OIDs, no orphaned `cdev` nodes, and no leaked memory.
- Your lab machine runs FreeBSD 14.3 with `/usr/src` on disk, a debug kernel with `INVARIANTS`, `WITNESS`, `WITNESS_SKIPSPIN`, `DDB`, `KDB`, `KDB_UNATTENDED`, `KDTRACE_HOOKS`, and `DDB_CTF`, and a VM snapshot at the 1.8-maintenance state you can revert to.

That driver, those files, and those habits are what Chapter 26 extends. The additions introduced in this chapter live almost entirely in a new file, `myfirst_usb.c`, which becomes a second kernel module sharing the same conceptual family as `myfirst.ko` but building a separate `myfirst_usb.ko`. The chapter's labs exercise each stage of the new module: probe, attach, transfer setup, callback handling, `/dev` exposure, and detach. The chapter does not modify `myfirst.ko` itself; the existing driver remains a reference implementation of Parts 2 through 5, and the new driver is its USB-transport sibling.

26.1.2 What You Will Learn

By the end of this chapter you will be able to:

- Explain what makes a transport-specific driver different from the pseudo-drivers built in Parts 2 through 5, and name the three broad categories of work a transport-specific driver has to add to its Newbus foundation: matching rules, transfer mechanics, and lifecycle hot-plug handling.
- Describe the USB mental model at the level needed to write a driver: host versus device roles, hubs and ports, device classes (CDC, HID, Mass Storage, Vendor), the descriptor hierarchy (device, configuration, interface, endpoint), the four transfer types (control, bulk, interrupt, isochronous), and the hot-plug lifecycle.

- Read the output of `usbconfig` and `dmesg` for a USB device and identify its vendor identifier, product identifier, interface class, endpoint addresses, endpoint types, and packet sizes.
- Describe the serial mental model at the level needed to write a driver: the UART as a shift register with a baud generator, RS-232 framing, start and stop bits, parity, hardware flow control via RTS and CTS, software flow control via XON and XOFF, and the relationship between `struct termios`, `tty(9)`, and the driver's `param` callback.
- Explain the difference between FreeBSD's `uart(4)` subsystem for real UART hardware and the `ucom(4)` framework for USB-to-serial bridges, and name the two worlds a serial-driver author must never confuse.
- Write a USB device driver that attaches to `uhub`, declares a `STRUCT_USB_HOST_ID` match table, implements `probe` and `attach` and `detach` methods, uses `usbd_transfer_setup` to configure a bulk-in and a bulk-out transfer, and unwinds cleanly through a labelled-goto chain.
- Write a USB transfer callback that follows the `USB_GET_STATE` state machine, handles `USB_ST_SETUP` and `USB_ST_TRANSFERRED` correctly, distinguishes `USB_ERR_CANCELLED` from other errors, and responds to a stalled endpoint with `usbd_xfer_set_stall`.
- Expose a USB device to user space through the `usb_fifo` framework or through a custom `make_dev_s`-registered `cdevsw`, and know when each is the right choice.
- Read an existing UART driver in `/usr/src/sys/dev/uart/` with a pattern vocabulary that makes the code's intent clear on first pass, including the `uart_class/uart_ops` split, the method dispatch, the baud-divisor calculation, and the `tty`-side wakeup machinery.
- Translate a `struct termios` into the four arguments of a UART `param` method (baud, databits, stopbits, parity), and know which `termios` flags belong to the hardware layer and which belong to the line discipline.
- Test a USB driver against a simulated device using QEMU USB redirection or `usb_template(4)`, and test a serial driver against a `nmdm(4)` null-modem pair without any hardware at all.
- Use `cu(1)`, `tip(1)`, `stty(1)`, `comcontrol(8)`, and `usbconfig(8)` to drive, configure, and inspect serial and USB devices from user space.
- Handle hot-unplug cleanly in a USB driver's `detach` path: cancel outstanding transfers, drain callouts and taskqueues, release bus resources, and destroy `cdev` nodes, all while knowing that the device may already be gone by the time the `detach` method runs.

The list is long because transport-specific drivers touch many surfaces at once. Each item is narrow and teachable. The chapter's work is making the set of items into a coherent, reusable mental picture.

26.1.3 What This Chapter Does Not Cover

Several adjacent topics are explicitly deferred to later chapters so Chapter 26 stays focused on the foundations of USB and serial driver development.

- **USB isochronous transfers and high-bandwidth video/audio streaming** are mentioned at a conceptual level in Section 1 but not developed. Isochronous transfers are the most complex of the four transfer types and are almost always used through higher-level frameworks (audio, video capture) that deserve their own treatment. Chapter 26 focuses on control, bulk, and interrupt transfers, which together cover the vast majority of USB driver work.
- **USB device-mode and gadget programming** through `usb_template(4)` is introduced briefly for testing purposes but not built out. Writing a custom USB gadget is a specialised project outside the scope of a first transport-specific chapter.

- **The internals of the USB host controller drivers** (`xhci`, `ehci`, `ohci`, `uhci`) are outside scope. These drivers implement the low-level protocol machinery that `usbd_transfer_setup` eventually calls; a driver author almost never has to modify them. The chapter treats them as a stable platform.
- **Writing a new UART hardware driver from scratch** is outside scope. The existing `ns8250` driver handles every common PC serial port, the `pl011` driver handles most ARM platforms, and the embedded SoC drivers handle the rest. Writing a new UART driver is the specialised work of porting FreeBSD to a new system-on-chip, which is its own topic (touched on in Part 7 alongside Device Tree and ACPI). Chapter 26 teaches the reader how to *read* and *understand* a UART driver rather than how to write one.
- **Storage drivers** (GEOM providers, block devices, VFS integration) are the subject of Chapter 27. USB mass storage is touched on only as an example of a USB device class, not as a driver target.
- **Network drivers** (`ifnet(9)`, `mbufs`, RX/TX ring bookkeeping) are the subject of Chapter 28. USB network adapters are mentioned as an example of CDC Ethernet, not as a driver target.
- **USB/IP for remote USB device testing over a network** is mentioned as an option for readers who truly cannot obtain any USB pass-through, but is not developed. The standard testing pathway in this chapter is a local VM with device redirection.
- **Quirks and vendor-specific workarounds** through `usb_quirk(4)` are mentioned but not developed. A driver author who needs quirks is already past the level this chapter teaches.
- **Bluetooth, Wi-Fi, and other wireless transports that happen to use USB** as their physical bus are outside scope. Those stacks involve protocols well beyond USB itself and are their own bodies of work.
- **Transport-agnostic abstraction for multi-bus drivers** (the same driver logic plugging into PCI, USB, and serial via a common interface) is deferred to Part 7's portability chapter.

Staying inside those lines keeps Chapter 26 a chapter about *the USB and serial transports*, not a chapter about every technique a senior transport-specific kernel developer might use on a senior transport-specific kernel problem.

26.1.4 Where We Are in Part 6

Part 6 has three chapters. Chapter 26 is the opening chapter and teaches transport-specific driver development through USB and serial devices. Chapter 27 teaches transport-specific driver development through storage devices and the VFS layer. Chapter 28 teaches it through network interfaces. The three chapters are structurally parallel in the sense that each introduces a transport, maps the subsystem, builds a minimal driver, and teaches hardware-free testing.

Chapter 26 is the right place to start Part 6 for three reasons. The first is that USB is the most gentle introduction to hardware-backed drivers: its core abstractions are smaller than storage's (no GEOM graph, no VFS), smaller than networking's (no `mbuf` chains, no ring buffers with head/tail pointers and interrupts mitigation), and the subsystem handles a large share of the hard parts on the driver's behalf. The second is that USB appears everywhere. Even a reader who will never write a storage or network driver will probably write a USB driver at some point: a thermometer, a data logger, a custom serial adapter, a factory test fixture. The third is pedagogical. The pattern USB teaches, a subsystem with probe-and-attach lifecycles, transfer setup through a config array, callback-based completion, and a clean detach on unplug, is the same pattern (with different specifics) that storage and networking teach. Seeing it first in USB makes the next two chapters recognisable.

Chapter 26 bridges forward to Chapter 27 by closing on a note about lifecycle: the USB detach path is a dress rehearsal for the storage-device hot-removal path, and the patterns the reader has just practised will come back

the moment an external USB disk is pulled in Chapter 27. It also bridges backward to Chapter 25 by carrying every Chapter 25 discipline forward: `MODULE_DEPEND`, `MODULE_PNP_INFO`, the labelled-goto pattern, the `errno` vocabulary, rate-limited logging, version discipline, and a regression script that exercises the new module as rigorously as Chapter 25 exercised the old one.

26.1.5 A Small Note on Difficulty

If the transition from pseudo-drivers to real hardware looks daunting on the first reading, that feeling is entirely normal. Every experienced FreeBSD developer had a first USB driver that did not attach, a first serial session where `cu` refused to talk, and a first debug session where `dmesg` stayed silent. The chapter is structured to ease you into each of those moments with labs, troubleshooting notes, and exit points. If a section starts to feel overwhelming, the right move is not to push through but to stop, read the corresponding real driver in `/usr/src`, and return when the real code makes the concept visible. The existing FreeBSD drivers are the single best teaching resource this chapter can point to, and the chapter will point to them often.

26.2 Reader Guidance: How to Use This Chapter

Chapter 26 has three layers of engagement, and you can pick the layer that fits your current situation. The layers are independent enough that you can read for understanding now and return for hands-on practice later without losing continuity.

Reading only. Three to four hours. Reading gives you the USB and serial mental models, the shape of the FreeBSD subsystems, and pattern recognition for reading existing drivers. If you are not yet in a position to load kernel modules (because your lab VM is unavailable, you are reading on a commute, or you have a planning meeting in thirty minutes), a reading-only pass is a worthwhile investment. The chapter is written so that the prose carries the teaching load; the code blocks are there to anchor the prose, not to replace it.

Reading plus the hands-on labs. Eight to twelve hours over two or three sessions. The labs guide you through building `myfirst_usb.ko`, exploring real USB devices with `usbconfig`, setting up a simulated serial link with `nmdm(4)`, talking to it with `cu`, and running a hot-unplug stress test. The labs are where the chapter turns from explanation into reflex. If you can spare eight to twelve hours across two or three sessions, do the labs. The cost of skipping them is that the patterns stay abstract instead of becoming habit.

Reading plus the labs plus the challenge exercises. Fifteen to twenty hours over three or four sessions. The challenge exercises push beyond the chapter's worked example into the territory where you have to adapt the pattern to a new requirement: add a control-transfer ioctl, port the driver to the `usb_fifo` framework, read an unfamiliar USB driver end-to-end, simulate a flaky cable with failure injection, or extend the regression script to cover the new module. The challenge material does not introduce new foundations; it stretches the ones the chapter has just taught. Spend time on the challenges in proportion to how much autonomy you expect to have on your next driver project.

Do not rush. This is the first chapter in the book whose material depends on real hardware or convincing simulation. Set aside a block of time when you can watch `dmesg` after `kldload` and read it slowly. A USB driver that attaches without errors is usually right; a USB driver whose attach messages you have not actually read is often wrong in a way that will cost you an hour of debugging two days later. The small discipline of reading the attach output as it happens, rather than assuming it, is a habit worth forming in Chapter 26 because every subsequent transport-specific chapter depends on it.

26.2.1 Recommended Pacing

Three sitting structures work well for this chapter.

- **Two long sittings of four to six hours each.** First sitting: Introduction, Reader Guidance, How to Get the Most Out of This Chapter, Section 1, Section 2, and Lab 1. Second sitting: Section 3, Section 4, Section 5, Labs 2 through 5, and the Wrapping Up. The advantage of long sittings is that you stay in the mental model long enough to connect Section 1's vocabulary to Section 3's callback code.
- **Four medium sittings of two to three hours each.** Sitting 1: Introduction through Section 1 and Lab 1. Sitting 2: Section 2 and Lab 2. Sitting 3: Section 3 and Labs 3 and 4. Sitting 4: Section 4, Section 5, Lab 5, and Wrapping Up. The advantage is that each sitting has a crisp milestone.
- **A linear reading pass followed by a hands-on pass.** Day one: read the entire chapter start to finish without running any code, to get the full mental model in place. Day two or day three: return to the chapter with a kernel source tree and a lab VM open and work through the labs in sequence. The advantage of this approach is that the mental model is fully loaded before you touch code, which catches concept-level mistakes early.

Do not attempt the whole chapter in a single marathon session. The material is dense, and the USB callback state machine in particular does not reward tired reading.

26.2.2 What a Good Study Session Looks Like

A good study session for this chapter has five elements visible at once. Put the book chapter on one side of your screen. Put the relevant FreeBSD source files in a second pane: `/usr/src/sys/dev/usb/usbdi.h`, `/usr/src/sys/dev/usb/misc/uled.c`, and `/usr/src/sys/dev/uart/uart_bus.h` are the three most useful to keep open. Put a terminal on your lab VM in a third pane. Put `man 4 usb`, `man 4 uart`, and `man 4 ucom` in a fourth pane for quick reference. Finally, keep a small note file open for questions you will want to answer later. If a term comes up you cannot define, write it in the note file and keep reading; if the same term comes up twice, look it up before continuing. This is the study posture that gets the most out of a long technical chapter.

26.2.3 If You Do Not Have a USB Device to Test With

Many readers will not have a spare USB device that matches the vendor/product identifiers in the worked example. That is fine. Section 5 teaches three ways to proceed: QEMU USB device redirection from a host to a guest, `usb_template(4)` for FreeBSD-as-USB-device, and the simulated-device approach that tests driver logic without a real bus at all. The chapter's worked example is written so that the driver's match table can be swapped for one matching any USB device you do have on your desk. A USB flash drive will do. A mouse will do. A keyboard will do. The chapter explains how to point the driver at whatever device you happen to have, at the cost of temporarily stealing that device from the kernel's built-in driver, which the chapter also covers.

26.3 How to Get the Most Out of This Chapter

Five habits pay off in this chapter more than in any of the earlier chapters.

First, **keep four short manual-page files open in a browser tab or a terminal pane**: `usb(4)`, `usb_quirk(4)`, `uart(4)`, and `ucom(4)`. These four pages together are the tightest overview the FreeBSD project has of the subsystems this chapter introduces. None of them is long. `usb(4)` describes the subsystem from the user's perspective and lists the `/dev` entries that appear. `usb_quirk(4)` lists the quirks table and explains what a quirk is, which will save you puzzlement later when you see quirk code in real drivers. `uart(4)` describes the serial subsystem from the user's perspective. `ucom(4)` describes the USB-to-serial framework. Skim each once at the start of the chapter. When the prose refers to "consult the manual page for details," return to the appropriate page. The manual pages are authoritative; this book is commentary.

Second, **keep three real drivers close to hand**. `/usr/src/sys/dev/usb/misc/uled.c` is a very small USB driver that talks to a USB-attached LED. It uses the `usb_fifo` framework, which is one of the two user-visible patterns the chapter teaches, and its entire `attach` function is smaller than a page. `/usr/src/sys/dev/usb/misc/ugold.c` is a slightly larger USB driver that reads temperature data from a TEMPer thermometer through interrupt transfers. It demonstrates the other common transfer type and shows how a driver uses a callout to pace its reads. `/usr/src/sys/dev/uart/uart_dev_ns8250.c` is the canonical 16550 UART driver; every PC serial port in the world uses it. Read each of these three files once at the start of the chapter and once more at the end. The first read will feel largely opaque; the second will feel almost obvious. That change is the measure of progress this chapter offers.

Third, **type every code addition by hand**. The `myfirst_usb.c` file grows through the chapter in roughly a dozen small increments. Each increment corresponds to a paragraph or two of prose. Typing the code by hand is what turns the prose into muscle memory. Pasting the code skips the lesson. If that sounds pedantic, notice that every working USB driver author has written a USB driver's `attach` function at least a dozen times; typing this one is the first of that dozen.

Fourth, **read `dmesg` after every `kldload`**. A USB driver produces a predictable pattern of attach messages: the device is detected on a port, the driver probes, the match succeeds, the driver attaches, the `/dev` node appears. If any of those steps is missing, something is wrong, and the sooner you notice the missing step, the sooner you fix it. The smallest discipline this chapter can give you is the habit of running `dmesg | tail -30` immediately after `kldload` and reading every line. If the output is boring, the driver probably works. If the output surprises you, investigate before proceeding.

Fifth, **after every section, ask yourself what would happen if you pulled the cable**. The question sounds silly; it is central. A well-written transport-specific driver is always one that handles being removed while in use. A USB driver in particular runs in a world where hot-unplug is the normal operating condition. If you find yourself writing a section of code and cannot answer "what if the cable were pulled right here," the section is not yet finished. Chapter 26 returns to this question often, not as rhetoric but as a discipline.

26.3.1 What to Do When Something Does Not Work

It will not all work the first time. USB drivers have a few common failure modes, and the chapter documents each of them in the troubleshooting section at the end. A short preview of the most common ones:

- The driver compiles but does not attach when the device is plugged in. Usually the match table has the wrong vendor or product identifier. The fix is to verify the identifier with `usbconfig dump_device_desc`.
- The driver attaches but the `/dev` node does not appear. Usually the `usb_fifo_attach` call failed because the name conflicts with an existing device. The fix is to change the basename or to detach the conflicting

driver first.

- The driver attaches but the first transfer never completes. Usually `usbd_transfer_start` was not called, or the transfer was submitted with a zero-length frame. The fix is to trace through `USB_ST_SETUP` and confirm that `usbd_xfer_set_frame_len` was called before `usbd_transfer_submit`.
- The driver attaches but the kernel panics on unplug. Usually the detach path is missing a `usbd_transfer_unsetup` call or a `usb_fifo_detach` call. The fix is to run the detach sequence under `INVARIANTS` and follow the `WITNESS` output back to the first dropped cleanup.

The troubleshooting section at the end of the chapter develops each of these cases in full, with diagnostic commands and expected output. The goal of this chapter is not to have everything work on the first try; the goal is to have a systematic debugging posture that turns every failure into a teachable moment.

26.3.2 Roadmap Through the Chapter

The sections in order are:

1. **Understanding USB and Serial Device Fundamentals.** The USB mental model at the level needed to write a driver: host and device, hubs and ports, classes, descriptors, transfer types, hot-plug lifecycle. The serial mental model: UART hardware, RS-232 framing, baud rate, parity, flow control, the tty discipline. The FreeBSD-specific split between `uart(4)` and `ucom(4)`. A first exercise with `usbconfig` and `dmesg` that grounds the vocabulary in a device you can see.
2. **Writing a USB Device Driver.** The FreeBSD USB subsystem layout. The Newbus shape of a USB driver. `STRUCT_USB_HOST_ID` and the match table. `DRIVER_MODULE` with `uhub` as the parent. `MODULE_DEPEND` on `usb`. `USB_PNP_HOST_INFO` for auto-load. The probe method using `usbd_lookup_id_by_uaa`. The attach method, the softc layout, the labelled-goto cleanup chain in attach and detach.
3. **Performing USB Data Transfers.** The struct `usb_config` array. `usbd_transfer_setup` and the lifetime of a struct `usb_xfer`. Control, bulk, and interrupt transfer shapes. The `usb_callback_t` state machine and `USB_GET_STATE`. Stall handling with `usbd_xfer_set_stall`. Frame-level operations (`usbd_xfer_set_frame_len`, `usbd_copy_in`, `usbd_copy_out`). Creating a `/dev` entry for the USB device through the `usb_fifo` framework. A worked example that sends bytes down a bulk-out endpoint and reads bytes back from a bulk-in endpoint.
4. **Writing a Serial (UART) Driver.** The `uart(4)` subsystem at the level needed to read a real driver. The `uart_class/uart_ops` split. The method table dispatched through `kobj`. The relationship between `uart_bus_attach` and `uart_tty_attach`. Baud rate, data bits, stop bits, parity, and the `param` method. RTS/CTS hardware flow control. struct `termios` and how it reaches the driver. `/dev/ttyu*` versus `/dev/cuau*` in FreeBSD. The `ucom(4)` framework for USB-to-serial bridges. A guided reading of the `ns8250` driver as a canonical example.
5. **Testing USB and Serial Drivers Without Real Hardware.** `nmdm(4)` virtual null-modem pairs for serial testing. `cu(1)` and `tip(1)` for terminal access. `stty(1)` and `comcontrol(8)` for configuration. QEMU USB device redirection for host-to-guest pass-through. `usb_template(4)` for FreeBSD-as-gadget testing. Software loopback patterns that exercise driver logic without any device at all. A reproducible test harness that runs a regression without human intervention.

After the five sections come a set of hands-on labs, a set of challenge exercises, a troubleshooting reference, a Wrapping Up that closes Chapter 26's story, a bridge to Chapter 27, and a glossary. Read linearly on a first

pass.

26.4 Section 1: Understanding USB and Serial Device Fundamentals

The first section teaches the mental models the rest of the chapter relies on. USB and serial devices share a surprising amount of machinery at the `tty/cdevsw` layer, and at the same time they differ dramatically at the transport layer. A reader who is clear on both the similarities and the differences will find Sections 2 through 5 straightforward. A reader who is not will find the subsequent code confusingly non-obvious. This section is the single best place to spend an extra thirty minutes if you want the rest of the chapter to feel easier.

The section is organised in three arcs. The first arc establishes what a *transport* is, and why transport-specific drivers look different from the pseudo-drivers of Parts 2 through 5. The second arc teaches the USB model: host and device, hubs and ports, classes, descriptors, endpoints, transfer types, and the hot-plug lifecycle. The third arc teaches the serial model: the UART, RS-232 framing, baud rate, parity, flow control, and the FreeBSD-specific split between `uart(4)` and `ucom(4)`. A first exercise at the end grounds the vocabulary in a device you can see with `usbconfig`.

26.4.1 What a Transport Is, and Why It Matters Here

A *transport* is the protocol and the lifecycle by which a device is connected to the rest of the system. Up to this point in the book, the `myfirst` driver has had no transport. Its device existed entirely in the Newbus tree, connected to the nexus through the pseudo parent, and its data flowed into a buffer in kernel memory. That makes `myfirst` a *pseudo-device*: a device whose existence is entirely a software fiction. Pseudo-devices are essential teaching tools. They let the reader learn Newbus, softc management, character device interfaces, `ioctl` handling, locking, and the rest, without also learning the specifics of a bus. By now, those topics are covered.

A transport-specific driver, by contrast, is one that attaches to a *real* bus. The bus has its own rules. It has its own way of saying “a new device has appeared.” It has its own way of delivering data. It has its own way of saying “a device has been removed.” A transport-specific driver is still a Newbus driver (that never changes in FreeBSD), but its parent is no longer the abstract pseudo bus. Its parent is `uhub` if it is a USB driver, `pci` if it is a PCI driver, `acpi` or `fdt` if it is on an embedded platform, and so on. The driver’s attach method receives arguments specific to that bus. Its cleanup responsibilities include bus-specific resources in addition to the ones it already had. Its lifecycle is the bus’s lifecycle, not the module’s lifecycle.

Three broad categories of work distinguish a transport-specific driver from the pseudo-drivers of Parts 2 through 5. They are worth naming explicitly because they recur in every transport chapter in Part 6.

The first is *matching*. A pseudo-driver attaches on module load; there is nothing to match because there is no real device. A transport-specific driver has to declare which devices it handles. On USB, this means a match table of vendor and product identifiers. On PCI, it means a match table of vendor and device identifiers. On ACPI or FDT, it means a match table of string identifiers. The kernel’s bus code enumerates devices as they appear and offers each one to every registered driver in turn; the driver’s probe method decides whether to claim the device. Getting the match table right is the first obstacle every transport-specific driver faces.

The second is *transfer mechanics*. A pseudo-driver’s read and write methods touch a buffer in RAM. A transport-specific driver’s read and write methods have to arrange for data to move across the bus. On USB, this means setting up one or more transfers using `usb_transfer_setup`, submitting them with

`usbd_transfer_submit`, and handling completion in a callback. On PCI, this means programming a DMA engine. On storage, this means translating block requests into bus transactions. The transfer mechanism is bus-specific and is where most of a transport-specific driver's new code lives.

The third is *hot-plug lifecycle*. A pseudo-driver is loaded when the module is loaded and detached when the module is unloaded. That is a simple lifecycle; `kldload` and `kldunload` are the only events it has to respond to. A transport-specific driver has to deal with *hot-plug*: the device can appear and disappear independently of the module's lifecycle. A USB device can be unplugged in the middle of a read. A SATA disk can be yanked out while the filesystem on it is mounted. An Ethernet cable can be pulled while a TCP connection is open. The driver's attach method runs when a device is physically inserted; its detach method runs when a device is physically removed. The detach may happen while the driver is still in use. Handling this correctly is the third big obstacle every transport-specific driver faces.

The rest of Part 6 is about those three categories of work in three different transports. Chapter 26 teaches USB and serial. Chapter 27 teaches storage. Chapter 28 teaches networking. The matching, the transfer mechanics, and the hot-plug lifecycle look different in each transport, but the three-category structure repeats. That structure is what makes it possible to learn one transport well and then learn the next one quickly.

A useful shorthand: in Parts 2 through 5, you learned how to *be* a Newbus driver. In Part 6, you learn how to *attach* to a bus that has its own ideas about when and how you exist.

26.4.2 The USB Mental Model

USB, the Universal Serial Bus, is a tree-structured, host-controlled, hot-pluggable serial bus. Every one of those adjectives matters, and understanding each of them is the foundation of writing a USB driver.

Tree-structured means that USB devices do not sit on a shared wire like devices on an I2C bus or an old ISA bus. Every USB device has exactly one upstream connection, to a parent hub. The root of the tree is the *root hub*, which is exposed by the USB host controller. Downstream of the root hub are other hubs and devices. A hub has a fixed number of downstream ports; each port can either be empty or connect to exactly one device. The tree is rebuilt on boot and updated every time a device is connected or disconnected. On FreeBSD, `usbconfig` shows this tree; on a fresh boot of a typical desktop you will see something like:

```
ugen0.1: <Intel EHCI root HUB> at usb0, cfg=0 md=HOST spd=HIGH
ugen1.1: <AMD OHCI root HUB> at usb1, cfg=0 md=HOST spd=FULL
ugen0.2: <Some Vendor Hub> at usb0, cfg=0 md=HOST spd=HIGH
ugen0.3: <Some Vendor Mouse> at usb0, cfg=0 md=HOST spd=LOW
```

The tree structure matters to a driver author for two reasons. First, it tells you that when you write a USB driver, your driver's *parent* in the Newbus tree is `uhub`. Every USB device sits under a hub. When you write `DRIVER_MODULE(myfirst_usb, uhub, ...)`, you are telling the kernel “my driver attaches to children of `uhub`,” which is the FreeBSD way of saying “my driver attaches to USB devices.” Second, the tree structure means that enumeration is dynamic. The kernel does not know what devices are on the tree until the tree is walked; a driver is offered each device as it appears, and has to decide whether to claim it.

Host-controlled means that one side of the bus is the master, the *host*, and all other sides are slaves, the *devices*. The host initiates every transfer; devices respond. A USB keyboard does not push keystrokes to the host

whenever a key is pressed; the host polls the keyboard on an *interrupt endpoint* many times per second, and the keyboard replies with “no new keys” or “key ‘A’ has been pressed” in response to each poll. This polling-and-response model has important consequences for a driver. Your driver, running on the host, has to initiate every transfer. A device cannot spontaneously send data; it can only respond when the host asks. What looks from user space like “the driver received data” is always, underneath, “the driver had a pending receive transfer and the host controller notified us that the transfer completed.”

For most of this chapter’s purposes, you are writing *host-mode* drivers: drivers that run on the host side. A FreeBSD system can also be configured as a USB *device*, through the `usb_template(4)` subsystem, and present itself as a keyboard or mass storage device or CDC Ethernet interface to another host. Device-mode drivers are a specialised topic touched on only briefly in Section 5 for testing purposes.

Hot-pluggable means that devices can appear and disappear while the system is running, and the subsystem has to cope. The USB host controller notices when a device is plugged in, a hub’s port status registers tell it so, enumerates the new device by asking it for its descriptors, assigns it an address on the bus, and then offers it to any driver whose match table applies. When a device is unplugged, the host controller notices the port status change and tells the subsystem, which in turn calls the driver’s detach method. The detach method may run at any time, including while the driver is holding a transfer that will now never complete, while user space has the driver’s `/dev` node open, or while the system is under load. Writing a correct detach method is the single hardest part of USB driver development. The chapter returns to this repeatedly.

Serial means that USB is a wire-level serial protocol: bytes flow one after another on a differential pair. The speed of the bus has evolved over the years: low-speed (1.5 Mbps), full-speed (12 Mbps), high-speed (480 Mbps), SuperSpeed (5 Gbps), and faster variants above that. From a driver author’s perspective, the speed is mostly transparent: the host controller and the USB core handle the electrical layer and the packet framing, and your driver works at the level of “here is a buffer, please send it” or “here is a buffer, please fill it.” The speed determines how fast data can move, but the driver code is the same.

With those four adjectives in place, the rest of the USB model falls into shape.

26.4.2.1 Device Classes and What They Mean to a Driver

Every USB device belongs to one or more *classes*, and the class tells the host (and the driver) what kind of device it is. Classes are numerical, defined by the USB Implementers Forum, and the values appear in descriptors. The ones a FreeBSD driver author will see most often include:

- **HID (Human Interface Device)**, class 0x03. Keyboards, mice, joysticks, game controllers, and a long tail of programmable devices that pretend to be keyboards or mice. HID devices present reports through interrupt endpoints; FreeBSD’s HID subsystem handles them mostly generically, though a vendor-specific driver can override.
- **Mass Storage**, class 0x08. USB flash drives, external disks, card readers. These attach through `umass(4)` to the CAM storage framework.
- **Communications (CDC)**, class 0x02, with subclasses for ACM (modem-like serial), ECM (Ethernet), NCM (Ethernet with multi-packet aggregation), and others. CDC ACM devices appear through `ucom(4)` as serial ports. CDC ECM and NCM devices appear through `cdce(4)` as network interfaces.
- **Audio**, class 0x01. Microphones, speakers, audio interfaces. FreeBSD’s audio stack handles these through `uaudio(4)`.
- **Printer**, class 0x07. USB printers. Handled through `ulpt(4)`.

- **Hub**, class 0x09. USB hubs themselves. Handled by the core `uhub(4)` driver.
- **Vendor-specific**, class 0xff. Any device whose functionality does not fit a standard class. Almost every interesting hobby USB device (USB-to-serial bridges, thermometers, relay controllers, programmers, loggers) is in this class.

When you write a USB driver, you often write for a vendor-specific device (class 0xff) and match on vendor/product identifiers. Occasionally you write for a standard-class device that FreeBSD does not yet handle, or for a standard-class device that has quirks requiring a dedicated driver. The class is not usually the match criterion; the vendor/product pair is. But the class tells you what framework, if any, you should integrate with. If the device's class is CDC ACM, the right framework is `ucom`. If the class is HID, the right framework is `hidbus` (new in FreeBSD 14). If the class is 0xff, there is no framework; you write a bespoke driver.

26.4.2.2 Descriptors: The Device's Self-Description

When the host enumerates a new USB device, it asks the device to describe itself. The device responds with a hierarchy of *descriptors*. Descriptors are the single most important USB concept to get clear: they are the USB equivalent of the PCI configuration space, but richer and nested.

The hierarchy is:

Device descriptor

Configuration descriptor [1..N]

Interface descriptor [1..M] (optionally with alternate settings)

Endpoint descriptor [0..E]

A *device descriptor* (`struct usb_device_descriptor` in `/usr/src/sys/dev/usb/usb.h`) describes the device as a whole: its vendor identifier (`idVendor`), its product identifier (`idProduct`), its device class, subclass, and protocol, its maximum packet size on endpoint zero, its release number, and the number of configurations it supports. Most devices have one configuration, but the USB spec allows more (a camera that can run in high-bandwidth or low-bandwidth modes, for example).

A *configuration descriptor* (`struct usb_config_descriptor`) describes one mode of operation: the number of interfaces it contains, whether the device is self-powered or bus-powered, its maximum power draw. When a driver selects a configuration (by calling `usbdevreq_set_config`, though in practice the USB core does this for you), the device's endpoints are activated.

An *interface descriptor* (`struct usb_interface_descriptor`) describes one logical function of the device. A composite device, such as a USB printer with a built-in scanner, has one interface per function. Each interface has its own class, subclass, and protocol. A driver can match on interface class rather than device class; this is common when a device's overall class is "Miscellaneous" or "Composite" but one of its interfaces has a specific class. An interface can have multiple *alternate settings*, which select different endpoint layouts; audio streaming interfaces use alternate settings to offer different bandwidths.

An *endpoint descriptor* (`struct usb_endpoint_descriptor`) describes one data channel. Endpoints have:

- An *address*, which is the endpoint number (0 through 15) combined with a direction bit (IN, meaning from device to host, or OUT, meaning from host to device).
- A *type*, which is one of control, bulk, interrupt, or isochronous.

- A *maximum packet size*, which is the largest single packet the endpoint can handle.
- An *interval*, which for interrupt and isochronous endpoints tells the host how often to poll.

Endpoint zero is special: every device has it, it is always a control endpoint, and it is always bidirectional (one IN half and one OUT half). The USB core uses endpoint zero for enumeration (asking the device for descriptors, setting its address, selecting its configuration). A driver can also use endpoint zero for vendor-specific control requests, though drivers usually access it through helper functions rather than setting up a transfer directly.

The descriptor hierarchy matters to a driver because the driver's probe method has access to the descriptors through the struct `usb_attach_arg` it receives, and its match logic often reads fields from them. The struct `usbdev_lookup_info` inside struct `usb_attach_arg` carries the device's identifiers, its class, subclass, and protocol, the current interface's class, subclass, and protocol, and a few other fields. The match table filters on some subset of those; the helper macros `USB_VP(v, p)`, `USB_VPI(v, p, info)`, `USB_IFACE_CLASS(c)`, and similar build entries that match different combinations of fields.

26.4.2.3 The Four Transfer Types

USB defines four transfer types, each suited to a different kind of data movement. A driver picks one or more types for its endpoints, and the choice affects everything about how the driver is structured.

Control transfers are for setup, configuration, and command exchange. Every device supports them on endpoint zero. They have a small, structured format: an eight-byte setup packet (the struct `usb_device_request`) followed by an optional data stage and a status stage. The setup packet specifies what the request is doing: its direction (IN or OUT), its type (standard, class, or vendor), its recipient (device, interface, or endpoint), and four fields (`bRequest`, `wValue`, `wIndex`, `wLength`) whose meaning depends on the request. Standard requests include `GET_DESCRIPTOR`, `SET_CONFIGURATION`, and so on; class and vendor requests are defined by the class specification or the vendor. Control transfers are reliable: the bus protocol guarantees delivery or returns a specific error. They are also relatively slow, because the bus allocates only a small share of its bandwidth to them.

Bulk transfers are for large, reliable, non-time-critical data. A USB flash drive uses bulk transfers for the actual data. A printer uses bulk OUT for the print stream. A USB-to-serial bridge uses bulk IN and bulk OUT for the two directions of the serial stream. Bulk transfers are reliable (errors are retried by the bus hardware), but they have no guaranteed timing: they use whatever bandwidth is left after control, interrupt, and isochronous traffic has been scheduled. In practice, on a lightly-loaded bus, bulk transfers are very fast. On a heavily-loaded bus, they can stall for milliseconds at a time. Bulk endpoints are the most common endpoint type for device-to-host or host-to-device streaming of data where latency is not critical.

Interrupt transfers are for small, time-sensitive data. The name is misleading: there are no hardware interrupts here. The “interrupt” refers to the fact that the device needs to get the host's attention periodically, and the host polls the endpoint at a configurable interval to see whether there is new data. A USB keyboard uses interrupt transfers to deliver keystrokes; a USB mouse uses them for movement reports; a thermometer uses them to deliver periodic readings. Interrupt endpoints have an `interval` field that tells the host how often to poll (in milliseconds for low- and full-speed devices, in microframes for high-speed). A driver that wants to know about input as it happens sets up an interrupt-IN transfer, submits it, and the USB core arranges the polling. When data arrives, the driver's callback fires.

Isochronous transfers are for streaming data with guaranteed bandwidth but no error recovery. USB audio and USB video use isochronous endpoints. The bus reserves a fixed share of each frame for isochronous traffic, so

the bandwidth is predictable, but transfers are not retried on error; if a packet is corrupted, it is lost. This trade-off makes sense for audio and video, where a dropped sample is better than a stall. Isochronous transfers are the most complex to program because they typically operate on many small frames per transfer; the `struct usb_xfer` machinery supports up to thousands of frames per transfer. Chapter 26 introduces isochronous transfers at the conceptual level and does not develop them further; real isochronous drivers (audio, video) are beyond the chapter's scope.

A typical vendor-specific USB device that a hobbyist or a driver-development learner will write code for looks like this: a vendor/product identifier, one vendor-specific interface, one bulk-IN endpoint, one bulk-OUT endpoint, and possibly an interrupt-IN endpoint for status events. That is the shape of the worked example in Sections 2 and 3.

26.4.2.4 The USB Hot-Plug Lifecycle

The hot-plug lifecycle is the sequence of events that happens when a USB device is inserted, in use, and removed. Writing a driver that handles this lifecycle correctly is the most important single discipline in USB driver development.

When a device is inserted, the host controller notices a port status change. It waits for the device to stabilise, then resets the port and assigns the device a temporary address of zero. It sends `GET_DESCRIPTOR` to endpoint zero on address zero, retrieves the device descriptor, and then assigns the device a unique address with `SET_ADDRESS`. All subsequent communication uses the new address. The host sends `GET_DESCRIPTOR` for the full configuration descriptor (including interfaces and endpoints), chooses a configuration, and sends `SET_CONFIGURATION`. At that point the device's endpoints are active and the USB subsystem offers the device to every registered driver in turn by calling each driver's probe method. The first driver to claim the device by returning a non-error code from probe wins; the subsystem then calls that driver's `attach` method.

During normal operation, the driver submits transfers to its endpoints, the host controller schedules them on the bus, and the callbacks fire on completion. This is the steady state Chapter 26's code examples operate in.

When the device is removed, the host controller notices another port status change. It does not wait; the electrical signal is gone immediately. The subsystem cancels any outstanding transfers on the device's endpoints (they complete in the callback with `USB_ERR_CANCELLED`), and then it calls the driver's `detach` method. The `detach` method has to release every resource the `attach` method acquired, including any `/dev` nodes it created, any locks, any buffers, and any transfers. It has to do this in the face of the fact that other threads may be in the middle of calling into the driver through those resources. A read in progress must be woken up and returned with an error. An `ioctl` in progress must be allowed to finish or interrupted. A callback that has just fired with `USB_ERR_CANCELLED` must not try to re-submit.

The hot-plug lifecycle is why USB drivers cannot be written the way pseudo-drivers are written. In a pseudo-driver, the module lifecycle (`kldload/kldunload`) is the only lifecycle; nothing unexpected happens. In a USB driver, the device lifecycle is separate from the module lifecycle and is driven by physical events. A user can unplug the device while a user-space process is blocked in `read()` on the driver's `/dev` node. The driver must wake that process up and return an error. A well-written USB driver treats this as the normal case, not the edge case.

Section 2 will walk through the structure of a USB driver that handles this correctly. For now, keep the lifecycle in mind: probe, attach, steady-state transfers, detach. Every USB driver has that sequence.

26.4.2.5 USB Speeds and What They Imply

USB has gone through several speed generations, and each matters to a driver writer in different ways. Low-speed (1.5 Mbps) was the original USB 1.0 speed, mostly used by keyboards and mice. Full-speed (12 Mbps) was USB 1.1, used by printers, early cameras, and mass-storage devices. High-speed (480 Mbps) was USB 2.0, which became the dominant speed for most devices in the 2000s. SuperSpeed (5 Gbps) was USB 3.0, which added a separate physical layer for high-throughput applications. SuperSpeed+ (10 Gbps and 20 Gbps) came with USB 3.1 and 3.2. USB 4.0 reuses the Thunderbolt physical layer and supports 40 Gbps.

For most driver writing, only three differences between these speeds matter:

Maximum packet size. Low-speed endpoints have a maximum packet size of 8 bytes. Full-speed goes up to 64 bytes. High-speed bulk endpoints go up to 512 bytes. SuperSpeed bulk endpoints go up to 1024 bytes with burst support. Your buffer sizes in the transfer configuration should match the endpoint's speed; using a 512-byte buffer on a full-speed bulk endpoint wastes memory because only 64 bytes fit in each packet.

Isochronous bandwidth. Isochronous transfers reserve bandwidth at a specific speed. A device that asks for 1 MB/s of isochronous bandwidth can only be supported on a host controller that can provide it; on slower hosts, the device must negotiate a lower rate or fail. This is why some USB audio devices work on one port but not another.

Endpoint polling interval. Interrupt endpoints are polled at a specific interval encoded in the descriptor's `bInterval` field. The units are milliseconds at low/full speed and "125 microsecond microframes" at high-/SuperSpeed. The framework handles the math; your driver just declares the logical polling interval via the endpoint descriptor and the framework does the right thing.

For the drivers we write in this chapter (`myfirst_usb` and the UART bridges like FTDI), speed does not affect the code structure. A bulk-IN channel's callback is the same whether it runs at 12 Mbps or 5 Gbps. The differences are in the numbers, not the flow.

26.4.2.6 Endpoints, FIFOs, and Flow Control

A USB endpoint is logically an I/O queue at one end of a pipe. On the device side, an endpoint corresponds to a hardware FIFO in the chip. On the host side, the endpoint is a framework abstraction. Between them, USB packets flow under the control of the USB protocol itself, which handles retransmission, sequencing, and error detection.

The host cannot be told "the device is full" the way you might expect on a traditional serial link. Instead, when a device cannot accept more data (because its FIFO is full), it returns a NAK handshake. NAK means "try again later." The host will keep retrying, at the protocol level, until either the device accepts the data (returns ACK) or some higher-level timeout fires. This is called NAK-limiting or bus throttling, and it happens invisibly to the driver: the framework sees the final ACK and delivers a successful completion.

Similarly, when the device has no data to send (for a bulk-IN or interrupt-IN transfer), it returns NAK to the IN token, and the host polls again. From the driver's perspective, the transfer is simply "pending" until the device has something to say.

This NAK mechanism is how USB handles flow control at the protocol level. Your driver does not need to implement its own throttling logic for bulk and interrupt channels; the USB protocol does it. Where flow control

does come into play is in higher-level protocols, where the device might want to signal a logical end-of-message or a temporary unavailability. Those signals are protocol-specific and not part of USB itself.

26.4.2.7 Descriptors In Depth

USB descriptors are the self-describing mechanism by which a device tells the host what it is and how to talk to it. We introduced them briefly earlier; here is a more complete picture.

The device descriptor is the root. It contains the device's vendor ID, product ID, USB specification version, device class/subclass/protocol (for devices that declare themselves at the device level rather than the interface level), maximum packet size for endpoint zero, and the number of configurations.

Configuration descriptors describe complete configurations. A configuration is a set of interfaces that work together. Most devices have one configuration; some have multiple to support different modes of operation (e.g., a device that can be either a printer or a scanner, selected by configuration).

Interface descriptors describe functional subsets of the device. Each interface has a class, subclass, and protocol that tells the host what kind of driver to use. A multi-function device has multiple interface descriptors in the same configuration. Additionally, an interface can have alternate settings: different sets of endpoints selectable on the fly for things like “low bandwidth mode” vs “high bandwidth mode”.

Endpoint descriptors describe individual endpoints within an interface. Each has an address (with direction bit), a transfer type, a maximum packet size, and an interval (for interrupt and isochronous endpoints).

String descriptors hold human-readable strings: the manufacturer name, the product name, the serial number. These are optional; their presence is indicated by nonzero string indices in the other descriptors.

Class-specific descriptors extend the standard descriptors with class-specific metadata. HID devices have a report descriptor that describes the format of the reports they send. Audio devices have descriptors for audio controls. Mass-storage devices have descriptors for interface subclasses.

The USB framework parses all of this at enumeration time and exposes the parsed data to drivers through the `struct usb_attach_arg`. Your driver does not have to read descriptors itself; it queries the framework for the information it needs. When the chapter says “the interface's `bInterfaceClass`”, what is meant is “the `bInterfaceClass` field of the interface descriptor the framework parsed and cached for us.”

`usbconfig -d ugenN.M dump_all_config_desc` is how you see the parsed descriptors from userland. Run that command on a few devices you own and note how the descriptors look. You will see that even simple devices like a mouse have a nontrivial descriptor tree: typically one device descriptor, one configuration descriptor, one interface descriptor (with `class=HID`), and one or two endpoint descriptors (for the HID report input and maybe an output).

26.4.2.8 Request-Response Over USB

The USB control transfer type supports a request-response pattern between host and device. A control transfer consists of three phases: a setup stage where the host sends an 8-byte setup packet describing the request, an optional data stage where either the host sends data or the device returns data, and a status stage where the recipient acknowledges the operation.

The setup packet has five fields:

- `bmRequestType`: describes the direction (in or out), the type of request (standard, class, or vendor), and the recipient (device, interface, endpoint, or other).
- `bRequest`: the request number. Standard requests have well-known numbers (`GET_DESCRIPTOR` = 6, `SET_ADDRESS` = 5, and so on). Class and vendor requests have class-specific or vendor-specific meanings.
- `wValue`: a 16-bit parameter, often used to specify a descriptor index or a value to set.
- `wIndex`: another 16-bit parameter, often used to specify an interface or endpoint.
- `wLength`: the number of bytes in the data stage (zero if there is no data stage).

Every USB device must support a small set of standard requests: `GET_DESCRIPTOR`, `SET_ADDRESS`, `SET_CONFIGURATION`, and a few others. The framework handles all of these at enumeration time. Your driver may also issue vendor-specific requests to configure the device in ways the standard does not define.

For example, the FTDI driver issues vendor-specific requests like `FTDI_SIO_SET_BAUD_RATE`, `FTDI_SIO_SET_LINE_CTRL`, and `FTDI_SIO_MODEM_CTRL` to program the chip. These requests are documented in FTDI's application notes; they are not part of USB itself, but they work over the USB control-transfer mechanism.

When your driver needs to issue a vendor-specific control request, the pattern is the one we showed in Section 3: construct the setup packet, copy it into frame zero of a control transfer, copy any data into frame one (for data-stage requests), and submit. The framework handles the three phases and calls your callback when the transfer completes.

26.4.3 The Serial Mental Model

The serial side of Chapter 26 is about a much older and much simpler protocol than USB. Serial communication over a UART is one of the oldest ways two computers can talk to each other, and its simplicity is both its strength and its limitation. A reader coming to UART after USB will find the protocol almost trivially small. But the integration with the rest of the operating system, the tty discipline, baud rate management, parity, flow control, and the two-worlds split between `uart(4)` and `ucom(4)`, is where most of the actual work lives.

26.4.3.1 The UART as a Piece of Hardware

A UART is a *Universal Asynchronous Receiver/Transmitter*: a chip that converts bytes into a serial bit stream on a wire and back again. The classical UART has two pins for data (TX and RX), two pins for flow control (RTS and CTS), four pins for modem status (DTR, DSR, DCD, RI), a ground pin, and occasionally a pin for a second “ring” signal that most modern equipment ignores. On a classic PC, the serial port has a nine-pin or twenty-five-pin D-subminiature connector and operates at RS-232 voltage levels (typically +/- 12 V). Modern embedded UARTs usually operate at 3.3 V or 1.8 V logic levels; a level converter chip sits between the UART and the RS-232 connector if a compatible port is needed.

Inside the UART, the core is a shift register. When the driver writes a byte to the UART's transmit register, the UART adds a start bit, shifts the byte out bit by bit at the configured baud rate, adds an optional parity bit, and then adds one or two stop bits. When a receiving UART detects a falling edge (the start bit), it samples the line at the middle of each bit time, assembles the bits into a byte, checks the parity, verifies the stop bit, and then stores the byte in its receive register. If any of those steps fails (the parity does not match, the stop bit is wrong, the framing is off), the UART notes a framing error, a parity error, or a break condition in its status register.

On most modern UARTs, the single receive and transmit registers are backed by small first-in-first-out buffers (FIFOs). The 16550A UART, still the de facto standard, has a 16-byte FIFO on each side. A driver that programs the FIFO with an appropriate “trigger level” can let the hardware buffer incoming bytes and raise an interrupt only when the FIFO passes the trigger level. This is the difference between “one interrupt per byte” (slow) and “one interrupt per trigger level” (fast). The 16550A’s FIFO is a big part of why this chip became the universal PC standard.

The UART’s speed is controlled by a *baud rate divisor*: the UART has an input clock (often 1.8432 MHz on classic PC hardware), and the baud rate is the clock divided by 16 times the divisor. A divisor of 1 with a 1.8432 MHz clock gives 115200 baud. A divisor of 12 gives 9600 baud. The FreeBSD ns8250 driver computes the divisor from the requested baud rate and programs it into the UART’s divisor-latch registers. Section 4 walks through this code.

RS-232 framing is the full protocol: start bit (one), data bits (five, six, seven, or eight), optional parity bit (none, odd, even, mark, or space), stop bit (one or two). A typical modern configuration is “8N1”: eight data bits, no parity, one stop bit. An older configuration sometimes seen on industrial equipment is “7E1”: seven data bits, even parity, one stop bit. The driver programs the UART’s line control register to select the framing; `struct termios` carries the configuration from user space.

26.4.3.2 Flow Control

The UART can transmit faster than the receiver can read if the receiver’s code is slow or is doing other work. *Flow control* is how the receiver tells the transmitter to pause. Two mechanisms exist.

Hardware flow control uses two extra wires: *RTS* (Request To Send) from the receiver, and *CTS* (Clear To Send) from the transmitter’s perspective (it is the wire the other side drives). When the receiver’s buffer is filling up, it deasserts RTS. The transmitter, seeing CTS deasserted, stops transmitting. When the buffer empties, the receiver reasserts RTS, CTS asserts on the other side, and transmission resumes. Hardware flow control is reliable and requires no software overhead on either side; it is the default choice when the hardware supports it.

Software flow control, also called XON/XOFF, uses two in-band bytes: XOFF (traditionally ASCII DC3, 0x13) to pause transmission, and XON (ASCII DC1, 0x11) to resume. The receiver sends XOFF when it is almost full and XON when it has room again. This mechanism works over a three-wire connection (TX, RX, ground) with no extra pins, at the cost of reserving two byte values for control use. If you are sending binary data that may contain 0x11 or 0x13, you cannot use software flow control; hardware flow control is the only option.

FreeBSD’s tty discipline handles software flow control entirely in software, at the line-discipline layer, with no involvement from the UART driver. Hardware flow control is partly in the driver (the driver programs the UART’s automatic RTS/CTS feature if the chip supports it) and partly in the tty layer. A driver author should know which flow control method the tty layer has selected; the `CRTSCTS` flag in `struct termios` signals hardware flow control.

26.4.3.3 /dev/ttyuN and /dev/cuauN: A FreeBSD-Specific Quirk

The FreeBSD tty layer creates two device nodes per serial port. The *callin* node is `/dev/ttyuN` (where N is the port number, 0 for the first port). The *callout* node is `/dev/cuauN`. The distinction is historical, from the days of dial-up modems, and remains useful.

A process opening `/dev/ttyuN` is saying “I want to answer an incoming call”: the open blocks until the modem raises DCD (Data Carrier Detect). Once DCD is up, the open completes. When DCD drops, the open process receives SIGHUP. The node is for incoming connections.

A process opening `/dev/cuauN` is saying “I want to make an outgoing call”: the open succeeds immediately, without blocking on DCD. The process can then dial out or, on non-modem uses, simply talk to the serial port. The node is for outgoing connections, and more generally for any use that does not require modem semantics.

In modern use, when a serial port is connected to something that is not a modem (a microcontroller, a console, a GPS receiver), the right node to open is almost always `/dev/cuau0`. Opening `/dev/ttyu0` on a non-modem port will usually hang, because DCD is never asserted. The distinction is FreeBSD-specific; Linux has no callout nodes and uses `/dev/ttyS0` or `/dev/ttyUSB0` for everything.

The chapter’s labs will use `/dev/cuau0` and the simulated pair `/dev/nmdm0A`/`/dev/nmdm0B` for serial exercises. The callin nodes are not used.

26.4.3.4 Two Worlds: `uart(4)` and `ucom(4)`

FreeBSD separates real UART hardware from USB-to-serial bridges into two distinct subsystems. The separation is not visible from user space (a USB serial adapter and a built-in serial port both appear as tty devices), but it is very visible from inside the kernel, and a driver author must not confuse the two.

`uart(4)` is the subsystem for real UARTs. Its scope includes the built-in serial port on a PC motherboard, PCI serial cards, the PrimeCell PL011 found on ARM embedded boards, the embedded SoC UARTs on i.MX, Marvell, Qualcomm, Broadcom, and Allwinner platforms, and so on. The `uart` subsystem lives in `/usr/src/sys/dev/uart/`. Its core code is in `uart_core.c` and `uart_tty.c`. Its canonical hardware driver is `uart_dev_ns8250.c`. A driver that attaches to a real UART writes a `uart_class` and a small set of `uart_ops`, and the subsystem handles everything else. The `/dev` nodes that `uart(4)` creates are called `ttyu0`, `ttyu1`, and so on (callin) and `cuau0`, `cuau1`, and so on (callout).

`ucom(4)` is the framework for USB-to-serial bridges: FTDI, Prolific, Silicon Labs, WCH, and similar. Its scope is *not* a UART at all; it is a USB device whose endpoints happen to behave like a serial port. The `ucom` framework lives in `/usr/src/sys/dev/usb/serial/`. Its header is `usb_serial.h`. Its body is `usb_serial.c`. A USB-to-serial driver writes USB probe, attach, and detach methods as in any other USB driver, and then registers a `struct ucom_callback` with the framework. The callback has entries for “open”, “close”, “set line parameters”, “start reading”, “stop reading”, “start writing”, and so on. The framework creates the `/dev` node (called `ttyu0`, `ttyu1` for callin, `cuau0`, `cuau1` for callout, note the capital U) and runs the tty discipline on top of the driver’s USB transfers.

The two worlds never mix. `uart(4)` is for hardware that is physically a UART. `ucom(4)` is for USB devices that behave like a UART. A USB-to-serial adapter is a `ucom` driver, not a `uart` driver. A PCI serial card is a `uart` driver (specifically, a shim in `uart_bus_pci.c`), not a `ucom` driver. The user-space interface is similar (both produce `cu*` device nodes), but the kernel code is entirely disjoint.

A historical note that sometimes confuses readers: FreeBSD once had a separate `sio(4)` driver for 16550-family UARTs. `sio(4)` was retired years ago and is not present in FreeBSD 14.3. If you see references to `sio` in older documentation, translate them mentally to `uart(4)`. Do not try to find or extend `sio`; it is gone.

26.4.3.5 What termios Carries, and Where It Goes

`struct termios` is the user-space structure that configures a tty. It has five fields: `c_iflag` (input flags), `c_oflag` (output flags), `c_cflag` (control flags), `c_lflag` (local flags), `c_cc` (control characters), and two speed fields `c_ispeed` and `c_ospeed`. The fields are manipulated with `tcgetattr(3)`, `tcsetattr(3)`, and the shell command `stty(1)`.

A UART driver cares almost exclusively about `c_cflag` and the speed fields. `c_cflag` carries:

- `CSIZE`: the character size (CS5, CS6, CS7, CS8).
- `CSTOPB`: if set, two stop bits; if clear, one.
- `PARENB`: if set, parity is enabled; the type depends on `PARODD`.
- `PARODD`: if set with `PARENB`, odd parity; if clear with `PARENB`, even parity.
- `CRTSCTS`: hardware flow control.
- `CLOCAL`: ignore modem status lines; treat the link as local.
- `CREAD`: enable the receiver.

When user space calls `tcsetattr`, the tty layer checks the request, invokes the driver's `param` method (via the `tsw_param` callback in `ttydevsw`), and the driver translates the `termios` fields into hardware register settings. The `uart_tty.c` bridge code walks through this in full and is the best place to see the translation happen.

`c_iflag`, `c_oflag`, and `c_lflag` are mostly handled by the tty line discipline, not by the driver. They control things like whether the line discipline maps CR to LF, whether echo is enabled, whether canonical mode is active, and so on. A UART driver does not need to know any of that; the tty layer handles it.

26.4.3.6 Flow Control at the Multiple Layers of a TTY

Flow control sounds like a single concept, but in practice there are several independent layers that can each throttle the data flow. Understanding the layers helps debug situations where data is mysteriously not flowing.

The lowest layer is electrical. On a real RS-232 line, flow control signals (RTS, CTS, DTR, DSR) are physical pins on the connector. The remote side's transmitter only sends data when its CTS pin is asserted. The local side asserts the RTS pin to tell the remote it is ready to receive. For this to work, the cable must pass RTS and CTS through correctly, and both ends must have flow control configured consistently.

The next layer is in the UART chip itself. Some 16650 and later UARTs have automatic flow control: if configured, the chip itself monitors CTS and pauses the transmitter without driver involvement. The `CRTSCTS` flag in `c_cflag` enables this.

The next layer is in the UART framework's ring buffers. When the RX ring fills past a high-water mark, the framework deasserts RTS (if flow control is enabled) to tell the remote side to pause. When it drains below a low-water mark, RTS is reasserted.

The next layer is the tty line discipline, which has its own input and output queues. The line discipline can also generate XON/XOFF bytes (0x11 and 0x13) if `IXON` and `IXOFF` are set in `c_iflag`. These are software flow control signals.

The highest layer is the userland program's read loop. If the program is slow at consuming data, bytes accumulate at every layer below it.

When debugging flow-control issues, check each layer. Use `stty -a -f /dev/cuau0` to see what `c_cflag` and `c_iflag` have active. Use `comcontrol /dev/cuau0` to see the current modem signals. Use a multimeter or oscilloscope on the physical signals if you can. Work down the layers until you find the one that is actually blocking the flow.

26.4.3.7 Why Baud Rate Errors Are Insidious

A common class of serial bug is a baud-rate mismatch that almost works. Suppose one side is running at 115200 and the other at 114400 (which is what you get from a slightly-off crystal). Most bytes will come through, but a few will be corrupted. The exact error rate depends on the bit pattern. Long runs of one polarity drift further than alternating patterns.

Even worse, the error rate depends on the byte being sent. ASCII printable characters are in the range 0x20 to 0x7e, where the bits are well-distributed. Non-printable characters like 0xff or 0x00 are more likely to suffer bit errors because they present long runs of one polarity.

If you find your serial driver “mostly works” but drops or corrupts a few bytes out of thousands, suspect a baud-rate mismatch before suspecting a logic bug in your driver. Compare the actual divisor the chip is using against the expected divisor. If they differ, the baud rate is not what you asked for.

The 16550 uses a clock source (usually 1.8432 MHz) divided by a 16-bit divisor to produce 16 times the baud rate. For 115200, the divisor is $(1843200 / (115200 * 16)) = 1$. For 9600, it is 12. For arbitrary rates, the divisor may not be an integer, and the closest integer produces a rounded rate. A rate of 115200 requested from a 24 MHz clock would produce a divisor of $(24000000 / (115200 * 16)) = 13.02$, rounding to 13, giving an actual rate of $(24000000 / (13 * 16)) = 115384$, which is 0.16% off. Standard tolerance for serial communication is 2-3%, so 0.16% is fine.

When you configure a UART for a nonstandard baud rate, check whether the rate can be represented exactly. If not, test with actual data exchange, not just a loopback check.

26.4.3.8 Historical Note on Minor Numbers

Older FreeBSD versions encoded a lot of information into the minor numbers of serial device files. Different minor numbers for the “callin” side vs the “callout” side, for hardware-flow vs software-flow, and for various lock states. This encoding is largely gone in modern FreeBSD; the distinctions are now handled by separate device nodes with different names (`ttyu` vs `cuau`, with suffixes for lock and init states). If you see odd minor-number manipulation in old code, know that modern code does not need it.

26.4.3.9 Wrapping Up Section 1

Section 1 has established the two mental models Chapter 26 depends on. The USB model is a tree-structured, host-controlled, hot-pluggable serial bus with four transfer types, a rich descriptor hierarchy, and a lifecycle in which physical events drive kernel events. The serial model is a simple shift-register hardware protocol with baud rate, parity, stop bits, and optional flow control, integrated into FreeBSD through a subsystem split between `uart(4)` for real UARTs and `ucom(4)` for USB-to-serial bridges, and exposed to user space through the `tty` discipline and device nodes like `/dev/cuau0`.

Before moving on, spend a few minutes with `usbconfig` on a real system. The vocabulary you have just learned is easier to keep straight once you have seen a real USB device's descriptors with your own eyes.

26.4.4 Exercise: Use `usbconfig` and `dmesg` to Explore USB Devices on Your System

This exercise is a short hands-on checkpoint that grounds Section 1's vocabulary in a device you can see. Perform it on your lab VM (or on any FreeBSD 14.3 system with at least one USB device connected). It takes about fifteen minutes.

Step 1. Inventory. Run `usbconfig` with no arguments:

```
$ usbconfig
ugen0.1: <Intel EHCI root HUB> at usb0, cfg=0 md=HOST spd=HIGH (0mA)
ugen0.2: <Generic Storage> at usb0, cfg=0 md=HOST spd=HIGH (500mA)
ugen0.3: <Logitech USB Mouse> at usb0, cfg=0 md=HOST spd=LOW (98mA)
```

The first line is the root hub. Each other line is a device. Read the format: `ugenN.M` where `N` is the bus number and `M` is the device number; the description in angle brackets is the device's string; `cfg` is the active configuration; `md` is the mode (HOST or DEVICE); `spd` is the bus speed (LOW, FULL, HIGH, SUPER); the parenthesised current is the maximum bus-supplied power draw.

Step 2. Dump a device's descriptors. Pick one of the non-root-hub devices and dump its device descriptor:

```
$ usbconfig -d ugen0.2 dump_device_desc

ugen0.2: <Generic Storage> at usb0, cfg=0 md=HOST spd=HIGH (500mA)

bLength = 0x0012
bDescriptorType = 0x0001
bcdUSB = 0x0200
bDeviceClass = 0x0000 <Probed by interface class>
bDeviceSubClass = 0x0000
bDeviceProtocol = 0x0000
bMaxPacketSize0 = 0x0040
idVendor = 0x13fe
idProduct = 0x6300
bcdDevice = 0x0112
iManufacturer = 0x0001 <Generic>
iProduct = 0x0002 <Storage>
iSerialNumber = 0x0003 <0123456789ABCDE>
bNumConfigurations = 0x0001
```

Read each field. Notice that `bDeviceClass` is zero: that is the USB convention for “the class is defined per interface, not at the device level.” For this device, the interface class will be Mass Storage (0x08).

Step 3. Dump the active configuration. Now dump the configuration descriptor, which includes the interfaces and endpoints:

```
$ usbconfig -d ugen0.2 dump_curr_config_desc

ugen0.2: <Generic Storage> at usb0, cfg=0 md=HOST spd=HIGH (500mA)

Configuration index 0

    bLength = 0x0009
    bDescriptorType = 0x0002
    wTotalLength = 0x0020
    bNumInterface = 0x0001
    bConfigurationValue = 0x0001
    iConfiguration = 0x0000 <no string>
    bmAttributes = 0x0080
    bMaxPower = 0x00fa

Interface 0
    bLength = 0x0009
    bDescriptorType = 0x0004
    bInterfaceNumber = 0x0000
    bAlternateSetting = 0x0000
    bNumEndpoints = 0x0002
    bInterfaceClass = 0x0008 <Mass storage>
    bInterfaceSubClass = 0x0006 <SCSI>
    bInterfaceProtocol = 0x0050 <Bulk only>
    iInterface = 0x0000 <no string>

Endpoint 0
    bLength = 0x0007
    bDescriptorType = 0x0005
    bEndpointAddress = 0x0081 <IN>
    bmAttributes = 0x0002 <BULK>
    wMaxPacketSize = 0x0200
    bInterval = 0x0000

Endpoint 1
    bLength = 0x0007
    bDescriptorType = 0x0005
    bEndpointAddress = 0x0002 <OUT>
    bmAttributes = 0x0002 <BULK>
    wMaxPacketSize = 0x0200
    bInterval = 0x0000
```

Every field in Section 1's vocabulary is right there. The interface class is 0x08 (Mass Storage). The subclass is 0x06 (SCSI). The protocol is 0x50 (Bulk-only Transport). There are two endpoints. Endpoint 0 has address

0x81 (the high bit indicates IN direction, the low five bits are the endpoint number, 1). Endpoint 1 has address 0x02 (the high bit is clear, meaning OUT; the endpoint number is 2). Both endpoints are bulk. Both have a maximum packet size of $0x0200 = 512$ bytes. The interval is zero because bulk endpoints do not use it.

Step 4. Match this against dmesg. Run `dmesg | grep -A 3 ugen0.2` (or look at the last boot's output for the matching device). You should see a line like:

```
ugen0.2: <Generic Storage> at usb0
umass0 on uhub0
umass0: <Generic Storage, class 0/0, rev 2.00/1.12, addr 2> on usb0
```

This is the same information, formatted by the kernel's own logging. The driver that attached is `umass`, which is FreeBSD's USB mass storage driver, and it attached to the Mass Storage interface class.

Step 5. Try `usbconfig -d ugen0.3 dump_all_config_desc` on another device. A mouse, a keyboard, or a flash drive will all work. Compare the endpoint types: a mouse has one interrupt-IN endpoint; a flash drive has one bulk-IN and one bulk-OUT; a keyboard has one interrupt-IN. The pattern holds.

If you want a small additional exercise, write down the vendor and product identifiers of one of your devices. In Section 2 you will be asked to put vendor and product identifiers into a match table; using ones you can see now is concrete.

26.4.5 Wrapping Up Section 1

Section 1 has done four things. It established the mental model of a transport: the protocol plus the lifecycle, plus the three broad categories of work (matching, transfer mechanics, hot-plug lifecycle) that a transport-specific driver has to add to its Newbus foundation. It built the USB model: host and device, hubs and ports, classes, descriptors with their nested structure, the four transfer types, and the hot-plug lifecycle. It built the serial model: the UART as a shift register with a baud generator, RS-232 framing, baud rate and parity and stop bits, hardware and software flow control, the FreeBSD-specific callin and callout node distinction, the two-worlds split between `uart(4)` and `ucom(4)`, and the role of `struct termios`. And it anchored the vocabulary in a concrete exercise that reads descriptors off a real USB device with `usbconfig`.

From here, the chapter turns to code. Section 2 builds a USB driver skeleton: probe, attach, detach, match table, registration macros. Section 3 makes that driver do real work by adding transfers. Section 4 turns to the serial side, walks through the `uart(4)` subsystem with a real driver as the guide, and explains where `ucom(4)` fits in. Section 5 brings the material back to the lab and teaches how to test USB and serial drivers without physical hardware. Each section builds on the mental models just established. If a later paragraph refers to a descriptor or a transfer type and the term does not feel immediate, return to Section 1 for a quick refresher before continuing.

26.5 Section 2: Writing a USB Device Driver

26.5.1 Moving from Concepts to Code

Section 1 built a mental picture of USB: a host that talks to devices through a tree of hubs, devices that describe themselves with nested descriptors, four transfer types that cover every conceivable traffic pattern, and a hot-

plug lifecycle that drivers must respect because USB devices appear and disappear at any moment. Section 2 turns those concepts into a real driver skeleton. By the end of this section, you will have a USB driver that compiles, loads, attaches to a matching device, and detaches cleanly when the device is unplugged. It will not yet perform data transfers; that is the job of Section 3. But the scaffolding you build here is the same scaffolding every FreeBSD USB driver uses, from the tiniest notification LED to the most complex mass-storage controller.

The discipline you learned in Chapter 25 carries forward unchanged. Every resource must have an owner. Every successful allocation in `attach` must be paired with an explicit release in `detach`. Every failure path must leave the system in a clean state. The labelled-goto cleanup chain, the `errno`-returning helper functions, the softc-based resource tracking, the rate-limited logging: all of it still applies. What changes is the set of resources you manage. Instead of bus resources allocated through Newbus and a character device created through `make_dev`, you will manage USB transfer objects allocated through the USB stack and, optionally, a `/dev` entry created through the `usb_fifo` framework. The shape of the code stays the same. Only the specific calls change.

This section moves from the outside in. It begins by explaining where a USB driver sits inside the FreeBSD USB subsystem, because placing the driver in its correct environment is a prerequisite for understanding every call that follows. It then covers the match table, which is how a USB driver declares which devices it wants. It walks through `probe` and `attach`, the two halves of the driver's entry point into the world. It covers the softc layout, which is where the driver keeps its per-device state. It presents the cleanup chain, which is how the driver unwinds its own work when `detach` is called. And it ends with the registration macros that bind the driver to the kernel module system.

Along the way, the chapter uses `uled.c` as a recurring reference. That is a real FreeBSD driver, about three hundred lines long, located at `/usr/src/sys/dev/usb/misc/uled.c`. It is short enough to read end to end in a single sitting and rich enough to show every piece of machinery a USB driver needs. If you want to ground every idea in this section against real code, open that file now in another window and keep it open. Every time the chapter references a pattern, you will be able to see the pattern in a working driver.

26.5.2 Where a USB Driver Sits in the FreeBSD Tree

FreeBSD's USB subsystem lives under `/usr/src/sys/dev/usb/`. That directory contains everything from the host controller drivers at the bottom (`controller/ehci.c`, `controller/xhci.c`, and so on) to the class drivers higher up (`net/if_cdce.c`, `wlan/if_rum.c`, `input/ukbd.c`), to serial drivers (`serial/uftdi.c`, `serial/uplcom.c`), to generic framework code (`usb_device.c`, `usb_transfer.c`, `usb_request.c`). When a new driver is added to the tree, it goes into one of these subdirectories according to its role. A driver for a blinking-LED gadget belongs under `misc/`. A driver for a network adapter belongs under `net/`. A driver for a serial adapter belongs under `serial/`. For your own work, you will not add files to `/usr/src/sys/dev/usb/` directly; you will build out-of-tree modules in your own workshop directory, the same way Chapter 25 did. The directory layout matters for reading the source, not for writing it.

Every FreeBSD USB driver sits somewhere in a small vertical stack. At the bottom is the host controller driver, which actually talks to the silicon. Above that is the USB framework, which handles descriptor parsing, device enumeration, transfer scheduling, hub routing, and the generic machinery every device needs. Above the framework are the class drivers, which you will write. A class driver attaches to a USB interface, not to the bus directly. This is the most important architectural point in the chapter.

In the Newbus tree, the attachment relationship looks like this:

```
nexus0
└─ pci0
    └─ ehci0 (or xhci0, depending on the host controller)
        └─ usb0
            └─ uhub0 (the root hub)
                └─ uhub1 (a downstream hub, if present)
                    └─ [class driver]
```

The driver you will write attaches to `uhub`, not to `usb0`, not to `ehci`, and not to `pci`. The USB framework walks the device descriptors, creates a child for each interface, and offers those children to class drivers through the newbus probe mechanism. When your driver's probe routine is called, it is being asked: "here is an interface; is it yours?" The match table in your driver is how you answer that question.

There is one subtle point to absorb. A USB device can expose multiple interfaces simultaneously. A multi-function peripheral (say, a USB audio device with a headset and a microphone on the same silicon) exposes one interface for playback and another for capture. FreeBSD gives each interface its own newbus child, and each child can be claimed by a different driver. This is why USB drivers attach at the interface level: it lets the framework route interfaces independently. Your driver should not assume the device has only one interface. When you write the match table, you write it against a specific interface, identified by its class, subclass, protocol, or by its vendor/product pair plus an optional interface number.

26.5.3 The Match Table: Telling the Kernel Which Devices Are Yours

A USB driver advertises which devices it will accept through an array of `STRUCT_USB_HOST_ID` entries. This is analogous to the PCI match table from Chapter 23, but with USB-specific fields. The authoritative definition lives in `/usr/src/sys/dev/usb/usbdi.h`. Each entry specifies one or more of the following: a vendor ID, a product ID, a device class/subclass/protocol triple, an interface class/subclass/protocol triple, or a manufacturer-defined `bcdDevice` range. You can match broadly (any device that advertises interface class `0x03`, which is HID) or narrowly (the single device with vendor `0x0403` and product `0x6001`, which is an FTDI FT232). Most drivers match narrowly, because most real devices have driver-specific quirks that apply only to particular hardware revisions.

The framework provides convenience macros to build match entries without having to initialize each field by hand. The most common are `USB_VPI(vendor, product, info)` for vendor/product pairs with an optional driver-specific information field, and the more verbose form where you fill in `mf_l`, `pf_l`, `dc_l`, `dcs_l`, `dcp_l`, `ic_l`, `ics_l`, `icpl` flags to indicate which fields are significant. For clarity and maintainability, drivers written today tend to use the compact macros whenever they are applicable.

Here is how `uled.c` declares its match table. The source is in `/usr/src/sys/dev/usb/misc/uled.c`:

```
static const STRUCT_USB_HOST_ID uled_devs[] = {
    {USB_VPI(USB_VENDOR_DREAMCHEEKY, USB_PRODUCT_DREAMCHEEKY_WEBMAIL_NOTIFIER, 0)},
    {USB_VPI(USB_VENDOR_RISO_KAGAKU, USB_PRODUCT_RISO_KAGAKU_WEBMAIL_NOTIFIER, 0)},
};
```

Two entries, each naming a specific vendor/product pair. The third argument to `USB_VPI` is an unsigned integer that the driver can use to distinguish variants at probe time; `uled` sets it to zero because both devices behave the same way. The vendor and product symbolic names resolve to numeric identifiers defined in `/usr/src/sys/dev/usb/usbdevs.h`, which is a large table generated from `/usr/src/sys/dev/usb/usbdevs`. Adding a new match entry for your own development hardware often means adding a line to `usbdevs` and regenerating the header, or bypassing the symbolic names entirely and writing the hexadecimal values directly in the match table.

For your own out-of-tree driver, you do not need to touch `usbdevs` at all. You can write:

```
static const STRUCT_USB_HOST_ID myfirst_usb_devs[] = {
    {USB_VPI(0x16c0, 0x05dc, 0)}, /* VOTI / generic test VID/PID */
};
```

The numeric form is perfectly acceptable. Use it when you are prototyping against a specific device and do not yet want to propose additions to the upstream `usbdevs` file.

One important detail about match tables: the `STRUCT_USB_HOST_ID` type includes a flag byte that records which fields are meaningful. When you use `USB_VPI`, the macro fills in those flags for you. If you hand-build an entry with literal braces, you must also fill in the flags yourself, because a zero flag byte means “match anything,” and you rarely want that. Prefer the macros.

The match table is plain data. It does not allocate memory, it does not touch hardware, and it does not depend on any per-device state. It is loaded into the kernel along with the module and used by the framework every time a new USB device is enumerated.

26.5.4 The probe Method

The USB framework calls a driver’s probe method once per interface when a matching-like candidate is presented. The goal of probe is to answer a single question: “Should this driver attach to this interface?” The method must not touch hardware. It must not allocate resources. It must not sleep. All it does is look at the USB attach argument, compare it against the match table, and return either a bus-probe value (indicating a match, with an associated priority) or `ENXIO` (indicating that this driver does not want this interface).

The attach argument lives in a structure called `struct usb_attach_arg`, defined in `/usr/src/sys/dev/usb/usbdi.h`. It carries the vendor ID, the product ID, the device descriptor, the interface descriptor, and a handful of helper fields. Newbus lets a driver retrieve it through `device_get_ivars(dev)`. For USB drivers, the framework provides a wrapper called `usb_lookup_id_by_uaa` that takes a match table and an attach argument and returns zero on a match or a nonzero `errno` on a miss. This wrapper encapsulates every case the driver needs to handle: vendor/product matching, class/subclass/protocol matching, the flag-byte logic, and the interface-level dispatch.

A complete probe method for our running example looks like this:

```
static int
myfirst_usb_probe(device_t dev)
{
```

```
struct usb_attach_arg *uaa = device_get_ivars(dev);

if (uaa->usb_mode != USB_MODE_HOST)
    return (ENXIO);

if (uaa->info.bConfigIndex != 0)
    return (ENXIO);

if (uaa->info.bInterfaceIndex != 0)
    return (ENXIO);

return (usbd_lookup_id_by_uaa(myfirst_usb_devs,
    sizeof(myfirst_usb_devs), uaa));
}
```

The three guard clauses at the top of the function are worth explaining in detail, because they reflect standard USB-driver hygiene.

The first guard rejects the case where the USB stack is acting as a device rather than a host. FreeBSD's USB stack can operate in USB-on-the-Go device mode, where the machine itself appears as a USB peripheral to some other host. Most drivers are host-side drivers and have no meaningful behavior in device mode, so they reject it immediately.

The second guard rejects configurations other than index zero. USB devices can expose multiple configurations, and a driver usually targets one specific configuration. Restricting probe to configuration index zero keeps the logic simple for the common case.

The third guard rejects interfaces other than index zero. If the device has multiple interfaces and you are writing a driver for the first one, this clause is what ensures the framework does not offer you the other interfaces by mistake.

After the guards, the call to `usbd_lookup_id_by_uaa` does the real matching work. If the device's vendor, product, class, subclass, or protocol matches any entry in the table, the function returns zero, and the probe method returns zero, which the USB framework interprets as "this driver wants this device." Returning `ENXIO` tells the framework to try another candidate driver. If no candidate wants the device, it ends up attached to `ugen`, the generic USB driver, which exposes raw descriptors and transfers through `/dev/ugenN.M` nodes but provides no device-specific behavior.

A subtle point worth noting: `probe` returns zero for a match rather than a positive bus-probe value. Other FreeBSD bus frameworks use positive values like `BUS_PROBE_DEFAULT` to indicate a priority, but for USB the convention is zero for match and a nonzero `errno` for non-match. The framework handles priority through the dispatch order rather than through probe return values.

26.5.5 The attach Method

Once `probe` reports a match, the framework calls `attach`. This is where the driver does real work: allocate its `softc`, record the parent device pointer, lock the interface, set up transfer channels (covered in Section 3),

create a /dev entry if the driver is user-facing, and log a short informational message. Every allocation and registration in attach has to be paired with a symmetric release in detach, and because any step can fail, the function must have a clear cleanup path from every failure point.

A minimal attach method looks like this:

```
static int
myfirst_usb_attach(device_t dev)
{
    struct usb_attach_arg *uaa = device_get_ivars(dev);
    struct myfirst_usb_softc *sc = device_get_softc(dev);
    int error;

    device_set_usb_desc(dev);

    mtx_init(&sc->sc_mtx, "myfirst_usb", NULL, MTX_DEF);

    sc->sc_udev = uaa->device;
    sc->sc_iface_index = uaa->info.bIfaceIndex;

    error = usbd_transfer_setup(uaa->device, &sc->sc_iface_index,
        sc->sc_xfer, myfirst_usb_config, MYFIRST_USB_N_XFER,
        sc, &sc->sc_mtx);
    if (error != 0) {
        device_printf(dev, "usbd_transfer_setup failed: %d\n", error);
        goto fail_mtx;
    }

    sc->sc_dev = make_dev(&myfirst_usb_cdevsw, device_get_unit(dev),
        UID_ROOT, GID_WHEEL, 0644, "myfirst_usb%d", device_get_unit(dev));
    if (sc->sc_dev == NULL) {
        device_printf(dev, "make_dev failed\n");
        error = ENOMEM;
        goto fail_xfer;
    }
    sc->sc_dev->si_drv1 = sc;

    device_printf(dev, "attached\n");
    return (0);

fail_xfer:
    usbd_transfer_unsetup(sc->sc_xfer, MYFIRST_USB_N_XFER);
fail_mtx:
    mtx_destroy(&sc->sc_mtx);
    return (error);
}
```

```
}

```

Read through this function top to bottom. Each block does one thing.

The call to `device_set_usb_desc` fills in the Newbus device description string from the USB descriptors. After this call, `device_printf` messages will include the manufacturer and product strings read from the device itself, which makes logs much more informative.

The call to `mtx_init` creates a mutex that will protect the per-device state. Every USB transfer callback runs under this mutex (the framework takes it for you around the callback), so everything the callback touches must be serialised by it. Chapter 25 introduced mutexes; the usage here is the same.

The two `sc->sc_` assignments cache two pointers that the rest of the driver will need. `sc->sc_udev` is the `struct usb_device *` that the driver uses when issuing USB requests. `sc->sc_iface_index` identifies the interface index this driver attached to, so later transfer-setup calls target the right interface.

The call to `usbd_transfer_setup` is the biggest single operation in `attach`. It allocates and configures all the transfer objects the driver will use, based on a configuration array (`myfirst_usb_config`) that Section 3 will examine in detail. If this call fails, the driver has not yet allocated anything except the mutex, so the cleanup path goes to `fail_mtx` and destroys the mutex.

The call to `make_dev` creates the user-visible `/dev` node. The Chapter 25 pattern applies here: set `si_drv1` on the `cdev` so that the `cdevsw` handlers can retrieve the `softc` through `dev->si_drv1`. If this call fails, the cleanup path goes to `fail_xfer`, which also runs the `unsetup` for the transfers before destroying the mutex.

The return (0) on the happy path is the contract with the framework: a zero return means the device is attached and the driver is ready.

The two labels at the bottom implement the labelled-goto cleanup chain from Chapter 25. Each label corresponds to the state the driver has reached at the time the failure happened, and the cleanup fall-through runs exactly the teardown steps needed to undo the work done so far. When you read a FreeBSD driver and see this pattern, you are looking at the same discipline you practised in Chapter 25 applied to a new set of resources.

One important detail about the USB framework that Chapter 25 did not need to cover: if you look at `uled.c` or any other real USB driver, you will sometimes see `usbd_transfer_setup` accept a pointer to the interface index rather than an integer. The framework can modify that pointer in the case of virtual or multiplexed interfaces; pass it by address, not by value. The skeleton above does this correctly.

26.5.6 The Softc: Per-Device State

A USB driver's `softc` is a plain C structure stored as the Newbus driver data for each attached device. It is allocated automatically by the framework based on the size declared in the driver descriptor, and it is the place where all per-device mutable state lives. For our running example, the `softc` looks like this:

```
struct myfirst_usb_softc {
    struct usb_device *sc_udev;
    struct mtx        sc_mtx;
    struct usb_xfer    *sc_xfer[MYFIRST_USB_N_XFER];
    struct cdev        *sc_dev;
};

```

```
uint8_t      sc_iface_index;
uint8_t      sc_flags;
#define MYFIRST_USB_FLAG_OPEN      0x01
#define MYFIRST_USB_FLAG_DETACHING 0x02
};
```

Let us walk through each member.

`sc_udev` is the opaque pointer the USB framework uses to identify the device. Every USB call that acts on the device takes this pointer.

`sc_mtx` is the per-device mutex that protects the softc itself and any shared state the driver cares about. The mutex must be acquired before touching any field that a transfer callback might also touch, and the transfer callback always runs with this mutex held (the framework handles the locking for you when it invokes the callback).

`sc_xfer[]` is an array of transfer objects, one per channel the driver uses. Its size is a compile-time constant. Section 3 will discuss how each entry in this array is set up by the configuration array passed to `usbd_transfer_setup`.

`sc_dev` is the character device entry, if the driver exposes a user-facing node. For drivers that do not expose a `/dev` node (some drivers only export data through `sysctl` or `devctl` events), this field can be omitted.

`sc_iface_index` records which interface on the USB device this driver attached to. It is used by transfer setup and, in multi-interface drivers, as a discriminator in logging.

`sc_flags` is a bit vector for driver-private state. Two flags are declared here: `MYFIRST_USB_FLAG_OPEN` is set while a userland process holds the device open, and `MYFIRST_USB_FLAG_DETACHING` is set at the start of detach so that any concurrent I/O path can see that it must abort quickly. This is an application of a standard pattern: setting a flag under the mutex at the start of detach, so anyone else who wakes up sees it and bails out.

Real drivers often have many more fields: per-transfer buffers, request queues, callback-to-callback state machines, timers, and so on. You add to the softc as the driver grows. The guiding principle is that any state that persists between function calls, and is not global to the module, belongs in the softc.

26.5.7 The detach Method

When a device is unplugged, when the module is unloaded, or when userspace uses `devctl detach`, the framework calls the driver's detach method. The driver's job is to release every resource it allocated in `attach`, cancel any in-flight work, make sure no callback is running, and return zero. If `detach` returns an error, the framework treats the device as still attached, which can create problems if the hardware has already physically vanished. Most drivers return zero unconditionally, or only return an error in very specific "device busy" cases where the driver implements its own reference counting for userspace handles.

The detach method for our running example is the symmetric cleanup of the attach method:

```
static int
myfirst_usb_detach(device_t dev)
{
    struct myfirst_usb_softc *sc = device_get_softc(dev);

    mtx_lock(&sc->sc_mtx);
    sc->sc_flags |= MYFIRST_USB_FLAG_DETACHING;
    mtx_unlock(&sc->sc_mtx);

    if (sc->sc_dev != NULL) {
        destroy_dev(sc->sc_dev);
        sc->sc_dev = NULL;
    }

    usbd_transfer_unsetup(sc->sc_xfer, MYFIRST_USB_N_XFER);

    mtx_destroy(&sc->sc_mtx);

    return (0);
}
```

The first block sets the detaching flag under the mutex. If another thread is about to take the mutex and start a new transfer, it will see the flag and refuse. The `destroy_dev` call removes the `/dev` entry; after it returns, no new open calls can arrive. The `usbd_transfer_unsetup` call cancels any in-flight transfers and waits for their callbacks to complete; after it returns, no transfer callback can still be running. With no new openers and no running callbacks, it is safe to destroy the mutex.

There is a subtlety here that new kernel programmers sometimes stumble over: the order matters. Destroying the `/dev` entry before unwinding the transfers ensures that no new user operation can start, but it does not stop the transfers that were already running when detach was called. That is `usbd_transfer_unsetup`'s job. Both steps are necessary, and the order (cdev first, then transfers, then mutex) is the right one because each later step depends on no new work arriving during it.

One further point about detach and concurrency. The framework guarantees that no probe, attach, or detach runs concurrently with another probe, attach, or detach on the same device. But transfer callbacks run on their own path, and they can be in progress at the exact moment detach is called. The combination of the detaching flag and `usbd_transfer_unsetup` is what makes this safe. If you add new resources to your driver, you must add symmetric cleanup that accounts for this concurrency.

26.5.8 Registration Macros

Every FreeBSD driver needs to register itself with the kernel so that the kernel knows when to call its probe, attach, and detach routines. USB drivers use a small set of macros that bind everything together into a kernel module. The macros go at the bottom of the driver file and look intimidating at first but are entirely mechanical once you know what each line does.


```
static device_method_t myfirst_usb_methods[] = {
    DEVMETHOD(device_probe, myfirst_usb_probe),
    DEVMETHOD(device_attach, myfirst_usb_attach),
    DEVMETHOD(device_detach, myfirst_usb_detach),
    DEVMETHOD_END
};

static driver_t myfirst_usb_driver = {
    .name      = "myfirst_usb",
    .methods   = myfirst_usb_methods,
    .size      = sizeof(struct myfirst_usb_softc),
};

DRIVER_MODULE(myfirst_usb, uhub, myfirst_usb_driver, NULL, NULL);
MODULE_DEPEND(myfirst_usb, usb, 1, 1, 1);
MODULE_VERSION(myfirst_usb, 1);
USB_PNP_HOST_INFO(myfirst_usb_devs);
```

Let us read each block.

The `device_method_t` array lists the methods the driver supplies. For a USB driver that does not implement extra newbus children, the three entries shown are sufficient: `probe`, `attach`, `detach`. More complex drivers might add `device_suspend`, `device_resume`, or `device_shutdown`, but for the vast majority of USB drivers the three basic entries are all that is needed. `DEVMETHOD_END` terminates the array; the framework requires it.

The `driver_t` structure binds the methods array to a human-readable name and declares the `softc` size. The name is used in kernel logs and by `devctl`. The `softc` size tells Newbus how much memory to allocate per device.

The `DRIVER_MODULE` macro registers the driver with the kernel. The arguments are, in order: the module name, the parent bus name (always `uhub` for USB class drivers), the driver structure, and two optional hooks for events. The event hooks are rarely needed and are usually `NULL`.

The `MODULE_DEPEND` macro declares that this module needs `usb` to be loaded first. The three numbers are the minimum, preferred, and maximum compatible versions of the `usb` module. For most drivers, `1, 1, 1` is correct: the USB framework has versioned its interface at 1 for a long time, and it would be unusual to require anything else.

The `MODULE_VERSION` macro declares this module's own version number. Other modules that want to depend on `myfirst_usb` would reference the number you declare here.

The `USB_PNP_HOST_INFO` macro is the last piece. It exports the match table into a format the `devd(8)` daemon can read, so that when a matching USB device is plugged in, userspace can auto-load the module. This macro is a relatively recent addition to FreeBSD; older drivers may not have it. Including it is strongly recommended for any driver that wants to participate in FreeBSD's USB plug-and-play system.

Together, these five declarations turn your driver file into a loadable kernel module. Once the file is compiled with a `Makefile` that uses `bsd.kmod.mk`, running `kldload myfirst_usb.ko` will bind the driver to the kernel, and any matching device plugged in afterwards will trigger your `probe` and `attach` routines.

26.5.9 The Hot-Plug Lifecycle, Revisited in Code

Section 1 introduced the hot-plug lifecycle at the level of mental model: a device appears, the framework enumerates it, your driver attaches, userland interacts with it, the device disappears, the framework calls detach, your driver cleans up. With the code in front of you, that narrative now has a concrete sequence:

1. The user plugs in a matching device.
2. The USB framework enumerates the device, reads all its descriptors, and decides which interfaces to offer to which drivers.
3. For each interface that matches your driver's match table, the framework creates a Newbus child and calls your probe method.
4. Your probe method returns zero.
5. The framework calls your attach method. You initialise the softc, set up transfers, create the /dev node, and return zero.
6. Userland opens the /dev node and begins issuing I/O. The transfer callbacks from Section 3 start running.
7. The user unplugs the device.
8. The framework calls your detach method. You set the detaching flag, destroy the /dev node, call `usbd_transfer_unsetup` to cancel all in-flight transfers and wait for callbacks to finish, destroy the mutex, and return zero.
9. The framework deallocates the softc and removes the Newbus child.

At every step, the framework handles the parts you do not have to write yourself. Your responsibility is narrow: react correctly to probe, attach, and detach, and run transfer callbacks that respect the state machine. The machinery around you handles enumeration, bus arbitration, transfer scheduling, hub routing, and the dozens of corner cases that USB layer imposes.

The lifecycle has one more subtle quirk that is worth naming. Between the user unplugging the device and the framework calling detach, there is a brief window in which any in-flight transfer sees a special error: `USB_ERR_CANCELLED`. The transfer framework itself generates this error when it tears down the transfers in response to the disconnect. Section 3 will explain how to handle this error in the callback state machine. For now, know that it exists and that it is the driver's normal signal that the device is going away.

26.5.10 Wrapping Up Section 2

Section 2 has given you a complete USB driver skeleton. The skeleton does not yet move data; that is Section 3's topic. But every other part of the driver is in place: the match table, the probe method, the attach method, the softc, the detach method, and the registration macros. You have seen how the USB framework routes a newly enumerated device through your probe routine, how your attach routine takes ownership and sets up state, how the driver integrates with Newbus through `device_get_ivars` and `device_get_softc`, and how the detach routine walks the allocation steps in reverse to leave the system clean.

Two themes from Chapter 25 have extended naturally into USB territory. First, the labelled-goto cleanup chain. Every resource you acquire has its own label, and every failure path falls through exactly the right sequence of teardown calls. When you compare `myfirst_usb_attach` above with the attach functions in `uled.c`, `ugold.c`, or `uftdi.c`, you will see the same pattern repeated. Second, the discipline of single-source-of-truth state in the softc. Every field has one owner, one lifecycle, and one clear place where it is initialised and destroyed. These habits are what make a driver readable, portable, and maintainable.

Section 3 will now give this skeleton a voice. Transfer channels will be declared in a configuration array. The USB framework will allocate the underlying buffers and schedule the transactions. A callback will wake up each time a transfer completes or needs more data, and it will use a three-state state machine to decide what to do. The same discipline you just learned will apply, but the new concern is the data pipeline itself: how bytes move between the driver and the device.

26.5.11 Reading `uled.c` As a Complete Example

Before moving into transfers, it is worth pausing to read the canonical small-driver example end to end. The file `/usr/src/sys/dev/usb/misc/uled.c` is approximately three hundred lines of C that implements a driver for the Dream Cheeky and Riso Kagaku USB webmail notifier LEDs: small USB gadgets with three coloured LEDs that a host program can light up. The driver is short enough to hold in your head, self-contained, and it exercises every pattern we have discussed.

When you open the file, the first block you encounter is the standard set of header includes. A USB driver pulls in headers from several layers: `sys/param.h`, `sys/systm.h`, `sys/bus.h` for the fundamentals; `sys/module.h` for `MODULE_VERSION` and `MODULE_DEPEND`; the USB headers under `dev/usb/` for the framework; and `usbdevs.h` for the symbolic vendor and product constants. Note that `usbdevs.h` is not a hand-maintained header: it is build-generated from the text file `/usr/src/sys/dev/usb/usbdevs` when the kernel or module is compiled, so the constants it exposes reflect whatever entries the in-tree `usbdevs` file currently lists. `uled.c` also pulls in `sys/conf.h` and friends because it creates a character device.

The second block is the `softc` declaration. `uled` keeps its state in a structure that has the device pointer, a mutex, an array of two transfer pointers (one for control, one for data), a character device pointer, a callback state pointer, and a small “color” byte that records the current LED colour. The `softc` is straightforward: every field is private, every allocation has one place where it is made and one place where it is freed.

The third block is the match table. `uled` supports two vendors (Dream Cheeky and Riso Kagaku) with one product ID each. The `USB_VPI` macro fills in the flag byte for a vendor-plus-product match. The table is two entries, flat and simple.

The fourth block is the transfer configuration array. `uled` declares two channels: a control-out channel used to send `SET_REPORT` requests to the device (which is how the LED colour is actually programmed), and an interrupt-in channel that reads status packets from the LED. The control channel has `type = UE_CONTROL` and a buffer size big enough to hold the setup packet plus the payload. The interrupt channel has `type = UE_INTERRUPT`, `direction = UE_DIR_IN`, and a buffer size that matches the LED’s report size.

The fifth block is the callback functions. The control callback follows the three-state machine you saw in Section 3: in `USB_ST_SETUP`, it constructs a setup packet and an eight-byte HID report payload, submits the transfer, and returns. In `USB_ST_TRANSFERRED`, it wakes any userland writer that was waiting for the colour change to complete. In the default case (errors), it handles cancellation gracefully and retries on other errors.

The interrupt callback is similar but without the setup-packet complication. It reads an eight-byte status report, checks whether it indicates a button press (the Riso Kagaku devices have an optional button), and rearms.

The sixth block is the character-device methods. `uled` exposes a `/dev/uled0` entry that accepts `write(2)` calls with a three-byte payload (red, green, blue). The `d_write` handler copies the three bytes into the `softc`, starts the control transfer, and returns. When the transfer completes, the colour is actually programmed. The `d_read` handler is not implemented (LEDs do not have meaningful state to read), so reads return zero.

The seventh block is the Newbus methods: `probe`, `attach`, `detach`. The `probe` uses `usbd_lookup_id_by_uaa` exactly as shown in Section 2. The `attach` calls `device_set_usb_desc`, initialises the mutex, calls `usbd_transfer_setup` with the configuration array, and creates the character device. The `detach` runs these in reverse.

The eighth block is the registration macros. `DRIVER_MODULE(uled, uhub, ...)`, `MODULE_DEPEND(uled, usb, 1, 1, 1)`, `MODULE_VERSION(uled, 1)`, and `USB_PNP_HOST_INFO(uled_devs)`. Exactly the sequence you learned.

Reading through `uled.c` with the Section 2 vocabulary in hand, the whole file legibly maps onto the patterns you now understand. Every structural choice the driver makes has a name. Every line of code is an instance of a general pattern. This is the kind of clarity that makes FreeBSD drivers readable.

Before continuing to Section 3, we recommend you actually open `uled.c` now and read it. Even if some lines are still obscure, the overall structure will match the mental model you have built. The details will make more sense as you progress through the rest of the chapter, and revisiting this file after finishing the chapter is an excellent way to consolidate the material.

26.6 Section 3: Performing USB Data Transfers

26.6.1 The Transfer Configuration Array

A USB driver declares its transfers up front, at compile time, through a small array of `struct usb_config` entries. Each entry describes one transfer channel: its type (control, bulk, interrupt, or isochronous), its direction (in or out), which endpoint it targets, how big its buffer is, which flags apply, and which callback function to invoke when the transfer completes. The framework reads this array once, during `attach`, when the driver calls `usbd_transfer_setup`. From that point on, each channel behaves like a small state machine that the driver drives through its callback.

The configuration array is declarative. You are not programming the sequence of hardware operations; you are telling the framework what channels your driver will use, and the framework builds the infrastructure to support them. This is an effective abstraction, and it is one of the reasons USB drivers in FreeBSD are usually much shorter than equivalent drivers for buses like PCI that demand direct register manipulation.

For our running example, we will declare three channels. A bulk-IN channel for reading data from the device, a bulk-OUT channel for writing data to the device, and an interrupt-IN channel for receiving asynchronous status events. A real driver for a serial adapter or an LED notifier might use one or two of these; we use three to show the pattern applied to different transfer types.

```
enum {
    MYFIRST_USB_BULK_DT_RD,
    MYFIRST_USB_BULK_DT_WR,
    MYFIRST_USB_INTR_DT_RD,
    MYFIRST_USB_N_XFER,
};

static const struct usb_config myfirst_usb_config[MYFIRST_USB_N_XFER] = {
```

```

[MYFIRST_USB_BULK_DT_RD] = {
    .type      = UE_BULK,
    .endpoint  = UE_ADDR_ANY,
    .direction = UE_DIR_IN,
    .bufsize   = 512,
    .flags     = { .pipe_bof = 1, .short_xfer_ok = 1 },
    .callback  = &myfirst_usb_bulk_read_callback,
},
[MYFIRST_USB_BULK_DT_WR] = {
    .type      = UE_BULK,
    .endpoint  = UE_ADDR_ANY,
    .direction = UE_DIR_OUT,
    .bufsize   = 512,
    .flags     = { .pipe_bof = 1, .force_short_xfer = 0 },
    .callback  = &myfirst_usb_bulk_write_callback,
    .timeout   = 5000,
},
[MYFIRST_USB_INTR_DT_RD] = {
    .type      = UE_INTERRUPT,
    .endpoint  = UE_ADDR_ANY,
    .direction = UE_DIR_IN,
    .bufsize   = 16,
    .flags     = { .pipe_bof = 1, .short_xfer_ok = 1 },
    .callback  = &myfirst_usb_intr_callback,
},
};

```

The enumeration at the top gives each channel a name and defines `MYFIRST_USB_N_XFER` as the total count. This is a common idiom; it keeps the channels symbolically accessible and makes it easy to add a new channel later. `MYFIRST_USB_N_XFER` is what you pass to `usbd_transfer_setup`, to `usbd_transfer_unsetup`, and to the softc's `sc_xfer[]` array declaration.

The array itself uses designated initialisers, which keeps the assignment of each channel to its enumeration index explicit. Let us walk through the fields.

`type` is one of `UE_CONTROL`, `UE_BULK`, `UE_INTERRUPT`, or `UE_ISOCHRONOUS`, from `/usr/src/sys/dev/usb/usb.h`. It has to match the endpoint's type as declared in the USB descriptors. If you say `UE_BULK` but the device has an interrupt endpoint, `usbd_transfer_setup` will fail.

`endpoint` identifies the endpoint number, but in most drivers the special value `UE_ADDR_ANY` is used, which tells the framework to pick any endpoint whose type and direction match. This works because most USB interfaces have only one endpoint of each (type, direction) pair, so “any” is unambiguous. A device with multiple bulk-in endpoints would require explicit endpoint addresses.

`direction` is `UE_DIR_IN` or `UE_DIR_OUT`. Again, this must match the descriptors.

`bufsize` is the size of the buffer the framework allocates for this channel. For bulk transfers, 512 bytes is a common choice because that is the maximum packet size for high-speed bulk endpoints, so a single 512-byte

buffer can hold exactly one packet. Larger buffers are supported, but for most purposes 512 or a small multiple is correct. For interrupt endpoints, the buffer can be smaller because interrupt packets are typically eight, sixteen, or sixty-four bytes.

flags is a bitfield struct (each flag is a one-bit integer). The flags affect how the framework handles short transfers, stalls, timeouts, and pipe behaviour.

- `pipe_bof` (pipe blocked on failure): if the transfer fails, block further transfers on the same pipe until the driver explicitly restarts it. This is usually set for both read and write endpoints.
- `short_xfer_ok`: for incoming transfers, treat a transfer that completed with less data than requested as success rather than error. Setting this is what allows a bulk-IN channel to read responses of variable length from a device.
- `force_short_xfer`: for outgoing transfers, finish the transfer with a short packet even when the data is aligned to a full packet boundary. This is used by some protocols to signal the end of a message.
- Several other flags control more advanced behaviour; for most drivers, `pipe_bof` plus `short_xfer_ok` (on reads) plus possibly `force_short_xfer` (on writes, protocol-dependent) is all that is needed.

callback is the function the framework calls whenever this channel needs attention. The callback is a `usb_callback_t`, which takes a pointer to the struct `usb_xfer` and returns void. All of the channel's state-machine logic lives inside the callback.

timeout (in milliseconds) sets an upper bound on how long a transfer can wait before being forcibly completed with an error. Setting a timeout is useful for write channels, because it prevents a hung device from stalling the driver indefinitely. For read channels, leaving the timeout at zero (meaning "no timeout") is common, because reads are often expected to block waiting for the device to have something to say.

This array, combined with `usbd_transfer_setup`, is all the driver needs to declare its data pipeline. The framework allocates the underlying DMA buffers, sets up the scheduling, and watches the pipes. The driver never has to call into a register or schedule a transaction by hand. It just writes callbacks.

26.6.2 Setting Up and Tearing Down Transfers

In the attach method shown in Section 2, the call to `usbd_transfer_setup` creates the channels from the configuration array:

```
error = usbd_transfer_setup(uaa->device, &sc->sc_iface_index,
    sc->sc_xfer, myfirst_usb_config, MYFIRST_USB_N_XFER,
    sc, &sc->sc_mtx);
```

The arguments are, in order: the USB device pointer, a pointer to the interface index (the framework can update it in certain multi-interface scenarios), the destination array for the created transfer objects, the configuration array, the number of channels, the softc pointer (which is passed into callbacks via `usb_xfer_softc`), and the mutex the framework will hold around each callback.

If this call succeeds, `sc->sc_xfer[]` is populated with pointers to struct `usb_xfer` objects. Each object encapsulates a channel's state. From this point, the driver can submit a transfer on a channel with `usbd_transfer_submit(sc->sc_xfer[i])`, and the framework will, in the fullness of time, call the corresponding callback.

The symmetric teardown, shown in the detach method, is `usbd_transfer_unsetup`:

```
usbd_transfer_unsetup(sc->sc_xfer, MYFIRST_USB_N_XFER);
```

This call does three things, in order. It cancels any in-flight transfer on each channel. It waits for the corresponding callback to run with `USB_ST_ERROR` or `USB_ST_CANCELLED`, so the driver has a chance to clean up any per-transfer state. It frees the framework's internal state for the channel. After `usbd_transfer_unsetup` returns, the `sc_xfer[]` entries are no longer valid, and the associated callback will not be invoked again.

This is the piece of machinery that makes detach safe in the presence of ongoing I/O. You do not need to implement your own “wait for outstanding transfers” logic. The framework provides it, atomically, through this single call.

26.6.3 The Callback State Machine

Every transfer callback follows the same three-state state machine. When the framework invokes the callback, you ask `USB_GET_STATE(xfer)` for the current state, and then you handle it. The three possible states are declared in `/usr/src/sys/dev/usb/usbd.h`:

- `USB_ST_SETUP`: the framework is ready to submit a new transfer on this channel. You should prepare the transfer (set its length, copy data into its buffer, and so on) and call `usbd_transfer_submit`. If you have no work for this channel right now, simply return; the framework will leave the channel idle until something else triggers a submit.
- `USB_ST_TRANSFERRED`: the most recent transfer completed successfully. You should read out the results (copy received data out, decide what to do next) and either return (if the channel should go idle) or fall through to `USB_ST_SETUP` to start another transfer.
- `USB_ST_ERROR`: the most recent transfer failed. You should inspect `usbd_xfer_get_error(xfer)` to see why, handle the error (for most errors, you fall through to `USB_ST_SETUP` to retry after a short delay; for stalls, you issue a clear-stall), and decide whether to continue.

The typical shape of a bulk-read callback looks like this:

```
static void
myfirst_usb_bulk_read_callback(struct usb_xfer *xfer, usb_error_t error)
{
    struct myfirst_usb_softc *sc = usbd_xfer_softc(xfer);
    struct usb_page_cache *pc;
    int actlen;

    usbd_xfer_status(xfer, &actlen, NULL, NULL, NULL);

    switch (USB_GET_STATE(xfer)) {
    case USB_ST_TRANSFERRED:
        pc = usbd_xfer_get_frame(xfer, 0);
        /*
         * Copy actlen bytes from pc into the driver's receive buffer.
         */
    }
```

```

        * This is where you hand the data to userland, to a queue,
        * or to another callback.
        */
        myfirst_usb_deliver_received(sc, pc, actlen);
        /* FALLTHROUGH */
    case USB_ST_SETUP:
tr_setup:
        /*
        * Arm a read for 512 bytes. The actual amount received may
        * be less, because we enabled short_xfer_ok in the config.
        */
        usbd_xfer_set_frame_len(xfer, 0, usbd_xfer_max_len(xfer));
        usbd_transfer_submit(xfer);
        break;

    default: /* USB_ST_ERROR */
        if (error == USB_ERR_CANCELLED) {
            /* The device is going away. Do nothing. */
            break;
        }
        if (error == USB_ERR_STALLED) {
            /* Arm a clear-stall on the control pipe; the framework
            * will call us back in USB_ST_SETUP after the clear
            * completes. */
            usbd_xfer_set_stall(xfer);
        }
        goto tr_setup;
    }
}

```

Let us walk through every piece.

The first line retrieves the softc pointer from the transfer object. This is how the callback gets at the per-device state. It works because the softc was passed to `usbd_transfer_setup`, which stored it inside the transfer object.

The call to `usbd_xfer_status` fills in `actlen`, the number of bytes actually transferred on frame zero. For a read, this is how much data arrived. For a write, it is how much data was sent. The other three parameters (which this example does not use) give the total transfer length, the timeout, and a status flags pointer; most callbacks only need `actlen`.

The switch on `USB_GET_STATE(xfer)` is the state machine. In `USB_ST_TRANSFERRED`, the callback copies the received data out of the USB frame into the driver's own buffer. The helper function `myfirst_usb_deliver_received` (which you would write) could push the data onto a queue, wake a sleeping `read()` on the `/dev` node, or feed a higher-level protocol parser.

The `FALLTHROUGH` after processing the transferred data takes the callback into the `USB_ST_SETUP` branch. This

is the idiomatic pattern for channels that run continuously: every time a read finishes, immediately start another read. If the driver wanted to stop reading after one transfer (say, a one-shot control request), it would return; at the end of `USB_ST_TRANSFERRED` instead of falling through.

In `USB_ST_SETUP`, `usbd_xfer_set_frame_len` sets the length of frame zero to the maximum the channel can handle, and `usbd_transfer_submit` hands the transfer to the framework. The framework will start the actual hardware operation and, when complete, call the callback again with either `USB_ST_TRANSFERRED` or `USB_ST_ERROR`.

The default case is where error handling happens. Two errors get special treatment. `USB_ERR_CANCELLED` is the signal that the transfer is being torn down, typically because the device was unplugged or `usbd_transfer_unsetup` was called. The callback must not resubmit the transfer in this case; if it did, it could race with the teardown and potentially touch memory that is about to be freed. Breaking out of the switch without calling `usbd_transfer_submit` is the correct behaviour.

`USB_ERR_STALLED` is the signal that the endpoint returned a STALL handshake, meaning the device is refusing to accept more data until the host clears the stall. The call to `usbd_xfer_set_stall` schedules a clear-stall operation on the control endpoint. After the clear-stall completes, the framework will call the callback again with `USB_ST_SETUP`, at which point the driver can reissue the transfer. This logic is built into the framework so that every driver gets the same correct behaviour with minimal code.

For any other error, the callback falls through to `tr_setup` and attempts to resubmit the transfer. This is a simple retry policy. A more sophisticated driver might count consecutive errors and give up after a threshold, or it might escalate by calling `usbd_transfer_unsetup` on itself. For many drivers, the default retry loop is sufficient.

26.6.4 The Write Callback

The write callback has the same shape but its `USB_ST_SETUP` branch is more interesting, because it has to decide whether there is any data to write:

```
static void
myfirst_usb_bulk_write_callback(struct usb_xfer *xfer, usb_error_t error)
{
    struct myfirst_usb_softc *sc = usbd_xfer_softc(xfer);
    struct usb_page_cache *pc;
    int actlen;
    unsigned int len;

    usbd_xfer_status(xfer, &actlen, NULL, NULL, NULL);

    switch (USB_GET_STATE(xfer)) {
    case USB_ST_TRANSFERRED:
        /* A previous write finished. Wake any blocked writer. */
        wakeup(&sc->sc_xfer[MYFIRST_USB_BULK_DT_WR]);
        /* FALLTHROUGH */
    case USB_ST_SETUP:
```

```

tr_setup:
    len = myfirst_usb_dequeue_write(sc);
    if (len == 0) {
        /* Nothing to send right now. Leave the channel idle. */
        break;
    }
    pc = usbd_xfer_get_frame(xfer, 0);
    myfirst_usb_copy_write_data(sc, pc, len);
    usbd_xfer_set_frame_len(xfer, 0, len);
    usbd_transfer_submit(xfer);
    break;

default: /* USB_ST_ERROR */
    if (error == USB_ERR_CANCELLED)
        break;
    if (error == USB_ERR_STALLED)
        usbd_xfer_set_stall(xfer);
    goto tr_setup;
}
}

```

The main change is the logic at `tr_setup`. For a read, the driver always wants another read armed, so the callback just sets the frame length and submits. For a write, the driver only submits if there is something to send. The helper `myfirst_usb_dequeue_write` returns the number of bytes pulled from an internal transmit queue; if zero, the callback breaks out of the switch without submitting anything, which leaves the channel idle. When userspace later writes more data into the device, the driver code that handles the `write()` system call queues the bytes and explicitly calls `usbd_transfer_start(sc->sc_xfer[MYFIRST_USB_BULK_DT_WR])`. That call fires an `USB_ST_SETUP` invocation of the callback, which now finds data in the queue and submits it.

This interaction between the userspace I/O path and the transfer state machine is the heart of an interactive USB driver. Reads are self-driving: once armed, they rearm themselves on every completion. Writes are demand-driven: they submit only when data is available and go idle otherwise. Both patterns run inside the same three-state machine; the difference is only in what happens at `USB_ST_SETUP`.

26.6.5 Control Transfers

Control transfers do not typically run on continuously-armed channels; they are usually issued one-shot, either synchronously from a system-call handler or as a one-shot callback triggered by some driver event. The `struct usb_config` for a control channel has `type = UE_CONTROL` and otherwise looks similar to the bulk and interrupt configurations. The buffer size must be at least eight bytes to hold the setup packet, and the callback deals with two frames: frame zero is the setup packet, and frame one is the optional data phase.

The typical one-shot use is to issue a vendor-specific request at driver-load time. The FTDI serial driver, for example, uses control transfers to set the baud rate and line parameters every time the user configures the serial

port. Because the control callback is scheduled by the framework just like any other transfer callback, the code pattern is identical. What differs is the construction of the setup packet in the USB_ST_SETUP branch.

For a control-read transfer, the code looks something like this:

```
case USB_ST_SETUP: {
    struct usb_device_request req;
    req.bmRequestType = UT_READ_VENDOR_DEVICE;
    req.bRequest      = MY_VENDOR_GET_STATUS;
    USETW(req.wValue,  0);
    USETW(req.wIndex,  0);
    USETW(req.wLength, sizeof(sc->sc_status));

    pc = usbd_xfer_get_frame(xfer, 0);
    usbd_copy_in(pc, 0, &req, sizeof(req));

    usbd_xfer_set_frame_len(xfer, 0, sizeof(req));
    usbd_xfer_set_frame_len(xfer, 1, sizeof(sc->sc_status));
    usbd_xfer_set_frames(xfer, 2);
    usbd_transfer_submit(xfer);
    break;
}
```

The USETW macro stores a sixteen-bit value in the request structure in the little-endian byte order USB requires. The usbd_copy_in helper copies from a kernel buffer into a USB frame. The usbd_xfer_set_frame_len and usbd_xfer_set_frames calls tell the framework how many frames the transfer spans and how long each is. For a control-read, frame zero is the setup packet (eight bytes) and frame one is the data phase; the framework transparently handles the status phase at the end.

In the USB_ST_TRANSFERRED branch, the driver reads the response out of frame one:

```
case USB_ST_TRANSFERRED:
    pc = usbd_xfer_get_frame(xfer, 1);
    usbd_copy_out(pc, 0, &sc->sc_status, sizeof(sc->sc_status));
    /* sc->sc_status now holds the device's response. */
    break;
```

Control transfers are the right tool for configuration operations where latency and bandwidth do not matter but correctness and sequencing do. They are the wrong tool for streaming data; use bulk or interrupt transfers for that.

26.6.6 Interrupt Transfers

Interrupt transfers are conceptually the simplest of the four types. An interrupt-IN channel runs a continuous state machine that polls a single endpoint at regular intervals. Each time a packet arrives from the device, the

callback wakes up with `USB_ST_TRANSFERRED`. The driver reads the packet, processes it (often by delivering it to userland), and falls through to rearm.

The callback for our interrupt channel is nearly identical to the bulk-read callback:

```
static void
myfirst_usb_intr_callback(struct usb_xfer *xfer, usb_error_t error)
{
    struct myfirst_usb_softc *sc = usbd_xfer_softc(xfer);
    struct usb_page_cache *pc;
    int actlen;

    usbd_xfer_status(xfer, &actlen, NULL, NULL, NULL);

    switch (USB_GET_STATE(xfer)) {
    case USB_ST_TRANSFERRED:
        pc = usbd_xfer_get_frame(xfer, 0);
        myfirst_usb_handle_interrupt(sc, pc, actlen);
        /* FALLTHROUGH */
    case USB_ST_SETUP:
    tr_setup:
        usbd_xfer_set_frame_len(xfer, 0, usbd_xfer_max_len(xfer));
        usbd_transfer_submit(xfer);
        break;

    default:
        if (error == USB_ERR_CANCELLED)
            break;
        if (error == USB_ERR_STALLED)
            usbd_xfer_set_stall(xfer);
        goto tr_setup;
    }
}
```

The only meaningful difference from the bulk-read callback is that the buffer is smaller (interrupt endpoint packets are typically eight to sixty-four bytes) and the semantics of the data are usually “status update” rather than “stream payload.” A USB HID device, for example, sends a sixty-four-byte report every few milliseconds describing key presses and mouse motions; an interrupt-IN channel polled continuously in this pattern is how the kernel receives those reports.

Interrupt-OUT channels work the same way but in reverse: the callback has to decide whether to send something at each `USB_ST_SETUP`, analogous to the bulk-write pattern.

26.6.7 Frame-Level Operations: What the Framework Gives You

USB transfers are composed of frames. A bulk transfer with a large buffer might be broken into multiple packets by the hardware; the framework hides that detail and presents the transfer as a single operation. A control transfer, on the other hand, has an explicit frame structure (setup, data, status). An isochronous transfer has one frame per scheduled packet. The framework exposes this structure through a small number of helper functions:

- `usbd_xfer_max_len(xfer)` returns the largest total length the channel can transfer in a single submit.
- `usbd_xfer_set_frame_len(xfer, frame, len)` sets the length of a specific frame.
- `usbd_xfer_set_frames(xfer, n)` sets the total number of frames in the transfer.
- `usbd_xfer_get_frame(xfer, frame)` returns a page-cache pointer for a specific frame, which is what you pass to `usbd_copy_in` and `usbd_copy_out`.
- `usbd_xfer_frame_len(xfer, frame)` returns how many bytes were actually transferred in a given frame (for completions).
- `usbd_xfer_max_framelen(xfer)` returns the maximum per-frame length for the channel.

For bulk and interrupt transfers, the vast majority of drivers only touch frame zero. For control transfers, they touch frames zero and one. For isochronous transfers (which we will not cover in this chapter), they loop over many frames. The point is that the framework gives you complete control over the per-frame data layout while hiding the hardware details that would otherwise make transfer scheduling a nightmare.

26.6.8 The `usbd_copy_in` and `usbd_copy_out` Helpers

USB buffers are not plain C buffers. They are allocated by the framework in a way that is addressable by the host controller hardware, which means they often live in DMA-accessible memory pages with platform-specific alignment requirements. The framework wraps these buffers in an opaque struct `usb_page_cache` object, and the driver accesses them through two helpers:

- `usbd_copy_in(pc, offset, src, len)` copies `len` bytes from the plain C buffer `src` into the framework-managed buffer at `offset`.
- `usbd_copy_out(pc, offset, dst, len)` copies `len` bytes out of the framework-managed buffer at `offset` into the plain C buffer `dst`.

You never dereference a struct `usb_page_cache *` directly. You never assume it points to a contiguous memory region. You always go through the helpers. This keeps the driver portable across platforms with different DMA constraints, and it is the standard convention throughout `/usr/src/sys/dev/usb/`.

If your driver needs to fill a USB buffer with data from a mbuf chain or from a userland pointer, there are dedicated helpers for that too: `usbd_copy_in_mbuf`, `usbd_copy_from_mbuf`, and the `uiomove` interaction is handled through `usbd_m_copy_in` and related routines. Search the USB framework source for the right helper; there is almost certainly one that matches your need.

26.6.9 Starting, Stopping, and Querying Transfers

Beyond the three callbacks, the driver interacts with transfer channels through a small number of control functions. The important ones are:

- `usbd_transfer_start(xfer)`: ask the framework to schedule a callback invocation in the `USB_ST_SETUP` state, even if the channel has been idle. Used when new data becomes available for a write channel.
- `usbd_transfer_stop(xfer)`: stop the channel. Any in-flight transfer is cancelled and the callback is invoked with `USB_ST_ERROR` (with `USB_ERR_CANCELLED`). No new callbacks happen until the driver calls `usbd_transfer_start` again.
- `usbd_transfer_pending(xfer)`: returns true if a transfer is currently outstanding. Useful for deciding whether to submit a new one or defer.
- `usbd_transfer_drain(xfer)`: blocks until any outstanding transfer completes and the channel is idle. Used in teardown paths that need to wait for in-flight I/O before continuing.

These functions are safe to call while holding the driver's mutex, and in fact most of them require it. The framework documentation and the existing driver code show the expected usage patterns; when in doubt, `grep` for the function name in `/usr/src/sys/dev/usb/` and read how existing drivers use it.

26.6.10 A Worked Example: Echo-Loop over USB

To make the transfer mechanics concrete, consider a small end-to-end scenario. The driver exposes a `/dev/my-first_usb0` entry that accepts writes and returns reads. A user process writes a string to the device; the driver sends those bytes to the USB device through the bulk-OUT channel. The device bounces the bytes back through its bulk-IN endpoint; the driver receives them and hands them to any process currently blocked in a `read()` on the `/dev` node. This is a useful exercise because it exercises both directions of the bulk pipeline and because it has a simple, observable success criterion: the string that goes in is the string that comes out.

The driver needs a small transmit queue and a small receive queue, both protected by the softc mutex. When userspace writes, the `d_write` handler acquires the mutex, copies the bytes into the transmit queue, and calls `usbd_transfer_start(sc->sc_xfer[MYFIRST_USB_BULK_DT_WR])`. When userspace reads, the `d_read` handler acquires the mutex, checks the receive queue; if empty, it sleeps on a channel related to the queue. The write callback, running under the mutex, dequeues bytes and submits the transfer. The read callback, also under the mutex, enqueues received bytes and wakes any blocked reader.

The complete flow from userspace `write("hi")` to userspace `read()` seeing "hi" involves three threads of execution interleaved through the state machines:

1. User thread runs `write()`. Driver enqueues "hi" on the TX queue. Driver calls `usbd_transfer_start`. User thread returns.
2. Framework schedules the TX callback with `USB_ST_SETUP`. Callback dequeues "hi", copies it into frame zero, sets frame length to 2, submits. Callback returns.
3. Hardware performs the bulk-OUT transaction. Device echoes "hi" on bulk-IN.
4. Framework schedules the RX callback with `USB_ST_TRANSFERRED` (because an earlier `USB_ST_SETUP` had armed a read). Callback reads "hi" from frame zero into the RX queue, wakes any blocked reader, falls through to re-arm the read, submits. Callback returns.
5. User thread, if it was blocked in `read()`, wakes up. The `d_read` handler copies "hi" out of the RX queue into userspace. User thread returns.

At each step, the mutex is held exactly where it needs to be, the state machine moves cleanly between `USB_ST_SETUP` and `USB_ST_TRANSFERRED`, and the driver does not have to think about packet boundaries, DMA mappings, or hardware scheduling. The framework handles all of that.

26.6.11 Putting the Whole Echo-Loop Driver Together

To make the echo-loop description concrete, let us walk through a complete skeleton for `myfirst_usb`. What follows is not a copy of the real driver files in `examples/`; it is a narrative presentation of how the pieces fit. The full code is in the `examples` directory.

The driver has one C source file, `myfirst_usb.c`, and a small header `myfirst_usb.h`. The header declares the `softc` structure, the constants for the transfer enumeration, and the prototypes for internal helper functions. The source file contains the match table, the transfer configuration array, the callback functions, the character-device methods, the Newbus probe/attach/detach, and the registration macros.

The `softc` is as we described earlier: a USB device pointer, a mutex, the transfer array, a character device pointer, an interface index, a flags byte, and two internal ring buffers for RX and TX queued data. Each ring buffer is a fixed-size array (say, 4096 bytes) plus head and tail indices, protected by the mutex.

The match table contains one entry:

```
static const STRUCT_USB_HOST_ID myfirst_usb_devs[] = {
    {USB_VPI(0x16c0, 0x05dc, 0)},
};
```

The 0x16c0/0x05dc VID/PID pair is the Van Oosting Technologies Incorporated / OBDEV generic test VID/PID, which is free to use for prototyping.

The transfer configuration array is the three-channel array from Section 3. The callbacks are the bulk-read, bulk-write, and interrupt-read patterns we walked through.

The bulk-read callback's `USB_ST_TRANSFERRED` branch calls a helper:

```
static void
myfirst_usb_rx_enqueue(struct myfirst_usb_softc *sc,
    struct usb_page_cache *pc, int len)
{
    int space;
    unsigned int tail;

    space = MYFIRST_USB_RX_BUFSIZE - sc->sc_rx_count;
    if (space < len)
        len = space; /* drop the excess; a real driver might flow-control */

    tail = (sc->sc_rx_head + sc->sc_rx_count) & (MYFIRST_USB_RX_BUFSIZE - 1);
    if (tail + len > MYFIRST_USB_RX_BUFSIZE) {
        /* wrap-around copy in two pieces */
        usbd_copy_out(pc, 0, &sc->sc_rx_buf[tail], MYFIRST_USB_RX_BUFSIZE - tail);
        usbd_copy_out(pc, MYFIRST_USB_RX_BUFSIZE - tail,
            &sc->sc_rx_buf[0], len - (MYFIRST_USB_RX_BUFSIZE - tail));
    } else {
        usbd_copy_out(pc, 0, &sc->sc_rx_buf[tail], len);
    }
}
```

```

    }
    sc->sc_rx_count += len;

    /* Wake any sleeper. */
    wakeup(&sc->sc_rx_count);
}

```

This is a ring-buffer enqueue with wrap-around handling. The `usb_copy_out` helper is used to move bytes from the USB frame into the ring buffer. If the ring buffer is full, bytes are dropped. A real driver would likely either apply USB-level flow control (stop arming new reads) or grow the buffer; for the lab, dropping is acceptable.

The bulk-write callback's helper to dequeue data is the mirror image:

```

static unsigned int
myfirst_usb_tx_dequeue(struct myfirst_usb_softc *sc,
    struct usb_page_cache *pc, unsigned int max_len)
{
    unsigned int len, head;

    len = sc->sc_tx_count;
    if (len > max_len)
        len = max_len;
    if (len == 0)
        return (0);

    head = sc->sc_tx_head;
    if (head + len > MYFIRST_USB_TX_BUFSIZE) {
        usb_copy_in(pc, 0, &sc->sc_tx_buf[head], MYFIRST_USB_TX_BUFSIZE - head);
        usb_copy_in(pc, MYFIRST_USB_TX_BUFSIZE - head,
            &sc->sc_tx_buf[0], len - (MYFIRST_USB_TX_BUFSIZE - head));
    } else {
        usb_copy_in(pc, 0, &sc->sc_tx_buf[head], len);
    }
    sc->sc_tx_head = (head + len) & (MYFIRST_USB_TX_BUFSIZE - 1);
    sc->sc_tx_count -= len;
    return (len);
}

```

The character-device methods are straightforward. Open checks that the device is not already open, sets the open flag, and arms the read channel:

```

static int
myfirst_usb_open(struct cdev *dev, int flags, int devtype, struct thread *td)

```



```

{
    struct myfirst_usb_softc *sc = dev->si_drv1;

    mtx_lock(&sc->sc_mtx);
    if (sc->sc_flags & MYFIRST_USB_FLAG_OPEN) {
        mtx_unlock(&sc->sc_mtx);
        return (EBUSY);
    }
    sc->sc_flags |= MYFIRST_USB_FLAG_OPEN;
    sc->sc_rx_head = sc->sc_rx_count = 0;
    sc->sc_tx_head = sc->sc_tx_count = 0;
    usbd_transfer_start(sc->sc_xfer[MYFIRST_USB_BULK_DT_RD]);
    usbd_transfer_start(sc->sc_xfer[MYFIRST_USB_INTR_DT_RD]);
    mtx_unlock(&sc->sc_mtx);

    return (0);
}

```

Close clears the open flag and stops the read channel:

```

static int
myfirst_usb_close(struct cdev *dev, int flags, int devtype, struct thread *td)
{
    struct myfirst_usb_softc *sc = dev->si_drv1;

    mtx_lock(&sc->sc_mtx);
    usbd_transfer_stop(sc->sc_xfer[MYFIRST_USB_BULK_DT_RD]);
    usbd_transfer_stop(sc->sc_xfer[MYFIRST_USB_INTR_DT_RD]);
    usbd_transfer_stop(sc->sc_xfer[MYFIRST_USB_BULK_DT_WR]);
    sc->sc_flags &= ~MYFIRST_USB_FLAG_OPEN;
    mtx_unlock(&sc->sc_mtx);

    return (0);
}

```

Read blocks until data is available, then copies bytes from the ring buffer to userspace:

```

static int
myfirst_usb_read(struct cdev *dev, struct uio *uio, int flags)
{
    struct myfirst_usb_softc *sc = dev->si_drv1;
    unsigned int len;
    char tmp[128];

```

```

int error = 0;

mtx_lock(&sc->sc_mtx);
while (sc->sc_rx_count == 0) {
    if (sc->sc_flags & MYFIRST_USB_FLAG_DETACHING) {
        mtx_unlock(&sc->sc_mtx);
        return (ENXIO);
    }
    if (flags & O_NONBLOCK) {
        mtx_unlock(&sc->sc_mtx);
        return (EAGAIN);
    }
    error = msleep(&sc->sc_rx_count, &sc->sc_mtx,
        PCATCH | PZERO, "myfirstusb", 0);
    if (error != 0) {
        mtx_unlock(&sc->sc_mtx);
        return (error);
    }
}

while (uio->uio_resid > 0 && sc->sc_rx_count > 0) {
    len = min(uio->uio_resid, sc->sc_rx_count);
    len = min(len, sizeof(tmp));
    /* Copy out of ring buffer into tmp (handles wrap-around) */
    myfirst_usb_rx_read_into(sc, tmp, len);
    mtx_unlock(&sc->sc_mtx);
    error = uiomove(tmp, len, uio);
    mtx_lock(&sc->sc_mtx);
    if (error != 0)
        break;
}
mtx_unlock(&sc->sc_mtx);
return (error);
}

```

Notice the pattern: the mutex is held while manipulating the ring buffer, but it is released around the `uiomove` call, because `uiomove` can sleep (to fault in user pages) and sleeping while holding a mutex is forbidden. The mutex is reacquired after `uiomove` returns.

Write is the mirror: copy bytes from user to TX buffer, then kick the write channel:

```

static int
myfirst_usb_write(struct cdev *dev, struct uio *uio, int flags)
{
    struct myfirst_usb_softc *sc = dev->si_drv1;

```

```

unsigned int len, space, tail;
char tmp[128];
int error = 0;

mtx_lock(&sc->sc_mtx);
while (uio->uio_resid > 0) {
    if (sc->sc_flags & MYFIRST_USB_FLAG_DETACHING) {
        error = ENXIO;
        break;
    }
    space = MYFIRST_USB_TX_BUFSIZE - sc->sc_tx_count;
    if (space == 0) {
        /* buffer is full; wait for the write callback to drain it */
        error = msleep(&sc->sc_tx_count, &sc->sc_mtx,
            PCATCH | PZERO, "myfirstusbw", 0);
        if (error != 0)
            break;
        continue;
    }
    len = min(uio->uio_resid, space);
    len = min(len, sizeof(tmp));

    mtx_unlock(&sc->sc_mtx);
    error = uiomove(tmp, len, uio);
    mtx_lock(&sc->sc_mtx);
    if (error != 0)
        break;

    /* Copy tmp into TX ring buffer (handles wrap-around) */
    tail = (sc->sc_tx_head + sc->sc_tx_count) & (MYFIRST_USB_TX_BUFSIZE - 1);
    myfirst_usb_tx_buf_append(sc, tail, tmp, len);
    sc->sc_tx_count += len;
    usbd_transfer_start(sc->sc_xfer[MYFIRST_USB_BULK_DT_WR]);
}
mtx_unlock(&sc->sc_mtx);
return (error);
}

```

Two things are worth noticing in write. First, when the TX buffer is full, the write handler sleeps on `sc_tx_count`; the write callback's `USB_ST_TRANSFERRED` branch calls `wakeup(&sc_tx_count)` after draining some bytes, which wakes the sleeping writer. Second, the write handler calls `usbd_transfer_start` on every chunk it enqueues. This is safe (starting an already-running channel is a no-op) and it ensures the write callback is nudged even if the channel had gone idle.

With these four cdev methods and the three transfer callbacks, you have a complete minimum-viable USB echo

driver. The full source is approximately three hundred lines: short enough to fit on a single screen, concrete enough to exercise the real API.

26.6.12 Choosing Between `usb_fifo` and a Custom `cdevsw`

When a USB driver needs to expose a `/dev` entry to userland, FreeBSD offers two approaches. The first is the `usb_fifo` framework, a generic byte-stream abstraction that gives you `/dev/ugenN.M.epM` style nodes with `read`, `write`, `poll`, and a small `ioctl` interface. You declare a `struct usb_fifo_methods` with `open`, `close`, `start-read`, `start-write`, `stop-read`, and `stop-write` callbacks, and the framework handles the `cdev` plumbing and the queueing. This is the path of least resistance; `uhid(4)` and `ucom(4)` both use it.

The second approach is a custom `cdevsw`, the same pattern you practised in Chapter 24. This gives you total control over the user interface at the cost of writing more code. It is appropriate when the driver needs a very specific `ioctl` surface, when the `read/write` semantics do not fit a byte stream (for example, a message-oriented protocol), or when the driver already fits poorly into the `usb_fifo` model.

For the running example we have built, a custom `cdevsw` is the right choice because we wrote the `attach` method that calls `make_dev` and the `detach` method that calls `destroy_dev`, which is exactly what a custom `cdevsw` requires. For a driver that exposes a byte stream (a serial adapter, say), `usb_fifo` is simpler. When you write your next USB driver, look at both options and pick the one whose interface matches your problem.

26.6.13 Error Handling and Retry Policy

The retry loop that our bulk-read callback uses, “on any error, rearm and try again,” is a reasonable default for robust drivers. But it is not the only policy, and sometimes it is the wrong one.

For a device that might genuinely go away mid-transfer (a USB adapter whose physical connection has been removed before the framework has had a chance to notice), rearming indefinitely is a waste; the transfers will keep failing until `detach` is called. Adding a small retry counter and giving up after, say, five consecutive errors, keeps the log from filling with noise.

For a device that implements a strict request-response protocol, an error might invalidate the entire session. In that case, the callback should not rearm; instead, it should mark the driver as “in error” and let the user close and reopen the device to reset.

For a device that supports stall-and-clear as a normal flow-control mechanism, the `usbdxfer_set_stall` path is in the happy path, not the error path. Some class protocols use stalls to signal “I am not ready right now”; the framework’s automatic clear-stall machinery handles this transparently.

Your choice of retry policy should match the real behaviour of the device you are writing for. When in doubt, start with the simple “rearm on error” default, observe what happens when you plug and unplug the device repeatedly, and refine from there.

26.6.14 Timeouts and Their Consequences

A timeout on a USB transfer is not just a safety net against hardware stalls; it is an explicit statement about how long the driver is willing to wait for an operation to complete before treating it as a failure. Choosing a timeout is

a design decision that interacts with many other parts of the driver, and getting it right requires thinking through several scenarios.

The configuration field `timeout` in `struct usb_config` is measured in milliseconds. A value of zero means “no timeout”; the transfer will wait indefinitely. A positive value means “if the transfer has not completed after this many milliseconds, cancel it and deliver a timeout error to the callback.”

For a read channel on a bulk endpoint, the usual choice is zero. Reads on bulk channels are waiting for the device to have something to say, and if the device is silent for minutes, that is not necessarily an error. A timeout would force the driver to rearm the read every few seconds, which wastes time and produces noise in the log.

For a write channel, the usual choice is a modest positive value like 5000 (five seconds). If the device fails to drain its FIFO in that time, something is wrong; rather than block an indefinite-length write, the driver returns an error to userland, which can retry if it wishes.

For an interrupt-IN channel polling for status updates, the usual choice is either zero (like a bulk read) or a timeout that matches the expected polling interval from the endpoint descriptor’s `bInterval` field. Matching `bInterval` gives the driver an explicit “I should have heard from the device by now” signal.

For a control transfer, timeouts matter most, because control transfers are how the driver configures the device, and a device that does not respond to configuration is wedged. A timeout of 500 to 2000 milliseconds is common. If the device does not respond to a configuration request in a few seconds, the driver should assume something is wrong.

What happens when a timeout fires? The framework calls the callback with `USB_ERR_TIMEOUT` as the error. The callback typically treats this as a transient failure and rearms (for repeating channels) or returns an error to the caller (for one-shot operations). A repeating read channel that keeps timing out is probably talking to a device that is not responding; after a few consecutive timeouts, it may be worth escalating by calling `usbd_transfer_unsetup` or by logging a more visible warning.

One subtle interaction is worth mentioning: if the transfer has a timeout and the driver also sets `pipe_bof` (pipe blocked on failure), a timeout will block the pipe until the driver explicitly clears the block. This is usually what you want, because the pipe may be in an inconsistent state, and clearing the block (by submitting a fresh setup, or by calling `usbd_transfer_start`) is a good point to log what happened and decide what to do next.

26.6.15 What Goes Wrong When Transfer Setup Fails

The `usbd_transfer_setup` call can fail for several reasons. Understanding each is useful both for debugging your own driver and for reading the FreeBSD source when you encounter failures.

Endpoint mismatch. If the configuration array asks for an endpoint with a specific type/direction pair that does not exist on the interface, the call fails with `USB_ERR_NO_PIPE`. This usually means the match table matched a device that has a different descriptor layout than the driver expected; it is a bug in the driver.

Unsupported transfer type. If the configuration specifies `UE_ISOCHRONOUS` on a host controller that does not support isochronous transfers, or if the bandwidth reservation cannot be satisfied, the call fails. Isochronous is the most complex transfer type and the most likely to have platform-specific limitations.

Out of memory. The framework allocates DMA-capable buffers for the channels. If memory is low, the allocation fails. This is rare on modern systems but can happen on embedded platforms with tight memory budgets.

Missing or invalid attributes. If the configuration has a buffer size of zero, or a negative frame count, or an invalid flag combination, the call fails. Check the configuration against the declarations in `/usr/src/sys/dev/usb/usbdi.h`.

Power management states. If the device has been suspended or is in a low-power state, some transfer setup requests will fail. This is mainly relevant for drivers that handle USB selective suspend.

When `usb_transfer_setup` fails, the error code is an `usb_error_t` value, not a standard `errno`. The definitions are in `/usr/src/sys/dev/usb/usbdi.h`. The function `usb_errstr` converts an error code to a printable string; use it in your `device_printf` to make diagnostic messages informative.

26.6.16 A Detail About `pipe_bof`

We mentioned `pipe_bof` (pipe blocked on failure) as a flag in the transfer configuration, but the motivation for it deserves a closer look. USB endpoints are conceptually single-threaded from the device's perspective. When the host submits a bulk-OUT packet, the device must process that packet before accepting another. If the packet fails, the device may be in an indeterminate state, and the next packet should not be sent until the driver has had a chance to resynchronise.

`pipe_bof` tells the framework to pause the pipe when a transfer fails. The next `usb_transfer_submit` will not actually start a hardware operation; instead, the framework waits until the driver explicitly calls `usb_transfer_start` on the channel, which acts as a “resume” signal. This lets the driver do a clear-stall or otherwise resynchronise before the next transfer begins.

Without `pipe_bof`, the framework would immediately submit the next transfer after a failure, which might run into the same failure before the driver has had a chance to react.

Setting `pipe_bof = 1` is the safe default for most drivers. Clearing it is appropriate for drivers that want to keep a pipeline full even through occasional errors (for example, audio drivers where a brief glitch is preferable to a synchronous resynchronisation).

26.6.17 `short_xfer_ok` and Data-Length Semantics

The `short_xfer_ok` flag is another configuration option whose meaning is worth spelling out. USB bulk transfers do not have an inherent end-of-message marker. If the host has a buffer of 512 bytes and the device only has 100 bytes to send, what should happen? There are two possible answers.

With `short_xfer_ok` clear (the default), a transfer that completes with less data than requested is treated as an error. The framework delivers `USB_ERR_SHORT_XFER` to the callback, and the driver must decide whether to retry, ignore, or escalate.

With `short_xfer_ok` set, a short transfer is treated as success. The callback gets `USB_ST_TRANSFERRED` with `act_len` set to the actual number of bytes received. This is almost always what you want for bulk-IN on message-oriented protocols, where the device decides how much data to send.

There is a corresponding flag for outgoing transfers: `force_short_xfer`. If set, a transfer whose data happens to be an exact multiple of the endpoint's maximum packet size will be padded with a zero-length packet at the end to signal “end of message.” USB treats a zero-length packet as a valid transaction, and many protocols use it as an explicit boundary marker. The FTDI driver sets this flag on its write channel, for example, because the FTDI protocol expects a trailing short packet.

Knowing which flag is appropriate requires knowing the protocol the device implements. When you write a driver for a device documented with a public protocol specification, check the specification for how it handles boundaries. When you write a driver for a poorly-documented device, set `short_xfer_ok` on reads (you can always count the bytes), and test both settings of `force_short_xfer` on writes to see which the device accepts.

26.6.18 Locking Rules Around Transfers

The USB framework imposes two locking rules that are essential to get right.

First, the mutex you pass to `usbd_transfer_setup` is held by the framework around every invocation of the callback. You do not need to acquire it inside the callback; it is already held. You also must not release it inside the callback; doing so breaks the framework's assumption and can cause random failures.

Second, every call from driver code (not from the callback) to one of `usbd_transfer_start`, `usbd_transfer_stop`, `usbd_transfer_submit`, `usbd_transfer_drain`, and `usbd_transfer_pending` must be made with the mutex held. This is because these functions read and write fields inside the transfer object that the callback also touches, and the mutex is what serialises access.

Practically, this means most driver code that interacts with transfers looks like:

```
mtx_lock(&sc->sc_mtx);
usbd_transfer_start(sc->sc_xfer[MYFIRST_USB_BULK_DT_WR]);
mtx_unlock(&sc->sc_mtx);
```

or in longer critical sections:

```
mtx_lock(&sc->sc_mtx);
/* enqueue data */
enqueue(sc, data, len);
/* nudge the channel if it is idle */
if (!usbd_transfer_pending(sc->sc_xfer[MYFIRST_USB_BULK_DT_WR]))
    usbd_transfer_start(sc->sc_xfer[MYFIRST_USB_BULK_DT_WR]);
mtx_unlock(&sc->sc_mtx);
```

Drivers that violate these rules occasionally appear to work but fail intermittently on load, under heavy I/O, or during detach. Getting the locking right from the start saves many hours of debugging later.

26.6.19 Wrapping Up Section 3

Section 3 has shown how data flows through a USB driver. A configuration array declares the channels, `usbd_transfer_setup` allocates them, the callbacks drive them through the three-state machine, and `usbd_transfer_unsetup` tears them down. The framework abstracts away the hardware details: DMA buffers, frame scheduling, endpoint arbitration, stall handling. The driver's job is to write callbacks that handle completion and to arrange the flow of data through the callbacks.

Three themes are worth carrying forward. First, the three-state state machine (`USB_ST_SETUP`, `USB_ST_TRANSFERRED`, `USB_ST_ERROR`) is the same in every channel, regardless of transfer type. Learning to read a USB callback means learning to parse this state machine; once you know it, every callback in every USB driver in the tree is legible. Second, the `usb_page_cache` abstraction is the only safe way to move data into and out of USB buffers. Never bypass `usbd_copy_in` and `usbd_copy_out`. Third, the locking discipline around `usbd_transfer_start`, `_stop`, and `_submit` is not optional; every call from driver code must be made under the mutex.

With Sections 1 through 3 in hand, you have a complete mental model of USB driver writing: the concepts, the skeleton, and the data pipeline. Section 4 now shifts to the serial side of Part 6. The UART subsystem is older, simpler in some ways, more constrained in others, and its idioms are different from USB's. But many of the same habits carry over: match against what you support, attach in phases that can be cleanly reversed, drive the hardware through a state machine, and respect the locking.

Take a breath. We have now worked through the USB half of the chapter: the host and device roles, the descriptor tree, the four transfer types, the probe/attach/detach skeleton, and the three-state `USB_ST_SETUP/USB_ST_TRANSFERRED/USB_ST_ERROR` callback machine that every USB driver runs. The rest of the chapter turns to the serial side: the `uart(4)` framework with its `ns8250` reference driver, integration with the TTY layer and `termios`, the `ucom(4)` bridge used by USB-to-serial adapters, and the tools and labs that let you test both kinds of drivers without real hardware. If you want to close the book and come back, this is a natural pause.

26.7 Section 4: Writing a Serial UART Driver

26.7.1 From USB to UART: A Shift of Landscape

Sections 2 and 3 gave you a complete USB driver. The framework there was modern in every sense: hot-plug, DMA-aware, message-oriented, richly abstracted. Section 4 now turns to `uart(4)`, FreeBSD's framework for driving Universal Asynchronous Receiver/Transmitters. The landscape is different. Many UART chips are older than USB itself, and the framework's design reflects that. There is no hot-plug (a serial port is usually soldered to the board). There is no DMA for most parts (the chip has a small FIFO you poll or an interrupt you handle). There is no descriptor hierarchy (the chip does not advertise its capabilities; you know what you built against). And there is no notion of transfer channels; there is just the port, into which bytes go and out of which bytes come.

What the framework does provide is a disciplined split of responsibilities between three layers. At the bottom sits your driver, which knows how the chip's registers work, how its interrupts fire, how its FIFOs behave, and what platform-specific resources (IRQ line, I/O port range, clock source) it needs. In the middle sits the `uart(4)` framework itself, which handles registration, baud-rate configuration calculations, buffering, TTY integration, and the scheduling of read and write work. At the top sits the TTY layer, which presents the port to userland as `/dev/ttyuN` and `/dev/cuauN` and handles terminal semantics: line editing, signal generation, control characters, and the vast vocabulary of `termios` knobs that `stty(1)` exposes.

You do not write the TTY layer. You do not write most of the `uart(4)` framework. Your job, when you write a UART driver, is to implement a small set of hardware-specific methods that the framework calls when it needs to do something at the register level. The framework then wires those methods into the rest of the kernel's serial machinery for free.

This section walks through that wiring. It covers the layout of the `uart(4)` framework, the structures and methods you have to fill in, the canonical `ns8250` driver as a concrete reference, and the integration with the TTY layer. It ends with the related `ucom(4)` framework, which is how USB-to-serial bridges expose themselves to userland using the same TTY interface as a real UART.

26.7.2 Where the `uart(4)` Framework Lives

The framework itself lives in `/usr/src/sys/dev/uart/`. If you list that directory, you see a handful of framework files and a family of hardware-specific drivers.

The framework files are:

- `/usr/src/sys/dev/uart/uart.h`: the top-level header that defines the framework's public API.
- `/usr/src/sys/dev/uart/uart_bus.h`: the structures for newbus integration and the per-port softc.
- `/usr/src/sys/dev/uart/uart_core.c`: the attachment logic, the interrupt dispatcher, the polling loop, the link between `uart(4)` and `tty(4)`.
- `/usr/src/sys/dev/uart/uart_tty.c`: the `ttydevsw` implementation that maps `uart(4)` operations onto `tty(4)` operations.
- `/usr/src/sys/dev/uart/uart_cpu.h`, `uart_dev_*.c`: platform glue and console registration.

The hardware-specific drivers are files of the form `uart_dev_NAME.c` and occasionally `uart_dev_NAME.h`. The most important of these is `uart_dev_ns8250.c`, which implements the `ns8250` family (including the 16450, 16550, 16550A, 16650, 16750, and many compatibles). Because the 16550A is effectively the standard UART for PC-style serial ports, this one driver handles the majority of actual serial hardware in the world. When you want to learn how a real FreeBSD UART driver looks, this is the file to open.

Other drivers in the directory handle chips that are not 16550-compatible: the Intel MID variant, the PL011 ARM UART used on Raspberry Pi and other ARM boards, the NXP i.MX UART, the Sun Microsystems Z8530, and so on. Each one follows the same pattern: fill in a `struct uart_class` and a `struct uart_ops`, register with the framework, and implement the hardware access methods.

26.7.3 The `uart_class` Structure

Every UART driver begins by declaring a `struct uart_class`, which is the hardware descriptor that the framework uses to identify the chip family. The definition lives in `/usr/src/sys/dev/uart/uart_bus.h`. The structure looks like this (paraphrased; the real declaration has a few more fields):

```
struct uart_class {
    KOBJ_CLASS_FIELDS;
    struct uart_ops *uc_ops;
    u_int          uc_range;
    u_int          uc_rclk;
    u_int          uc_rshift;
};
```

The `KOBJ_CLASS_FIELDS` macro pulls in the `kobj` machinery that Chapter 23 introduced (in the context of Newbus). A `uart_class` is, at the kernel's abstract-object level, a `kobj` class whose instances are `uart_softc`.

This is how the framework can call into driver-specific methods without needing an `if` ladder: the method dispatch is done by `kobj` lookup.

`uc_ops` is a pointer to the operations structure (coming next), which lists the chip-specific methods.

`uc_range` is how many bytes of register address space the chip uses. For an ns16550-compatible UART, this is 8.

`uc_rclk` is the reference clock frequency in hertz. The framework uses this to compute baud-rate divisors. For a PC-style UART, the reference clock is usually 1,843,200 hertz (a specific multiple of the standard baud rates).

`uc_rshift` is the register address shift. On some buses, UART registers are spaced at intervals other than one byte (for example, every four bytes on some memory-mapped designs). A shift of zero means tight packing; a shift of two means each logical register occupies four bytes of address space.

For our running example, the class declaration looks like this:

```
static struct uart_class myfirst_uart_class = {
    "myfirst_uart class",
    myfirst_uart_methods,
    sizeof(struct myfirst_uart_softc),
    .uc_ops    = &myfirst_uart_ops,
    .uc_range  = 8,
    .uc_rclk   = 1843200,
    .uc_rshift = 0,
};
```

The first three positional arguments are the `KOBJ_CLASS_FIELDS` entries: a name, a method table, and a per-instance size. The named fields are the UART-specific ones. For a driver targeting 16550-compatible chips, these values are the conventional defaults.

26.7.4 The `uart_ops` Structure

The `struct uart_ops` is where the real hardware-specific code lives. It is a table of function pointers that the framework calls at specific moments. The definition lives in `/usr/src/sys/dev/uart/uart_cpu.h`:

```
struct uart_ops {
    int  (*probe)(struct uart_bas *);
    void (*init)(struct uart_bas *, int, int, int, int);
    void (*term)(struct uart_bas *);
    void (*putc)(struct uart_bas *, int);
    int  (*rxready)(struct uart_bas *);
    int  (*getc)(struct uart_bas *, struct mtx *);
};
```

Each operation takes a `struct uart_bas *` as its first argument. The “bas” stands for “bus address space”; it is the framework’s abstraction for access to the chip’s registers. A driver does not know or care whether the chip

is in I/O space or in memory-mapped space; it just calls `uart_getreg(bas, offset)` and `uart_setreg(bas, offset, value)` (declared in `/usr/src/sys/dev/uart/uart.h`), and the framework routes the access correctly.

Let us go through the six operations in turn.

`probe` is called when the framework needs to know whether a chip of this class is present at a given address. The driver typically pokes a register, reads it back, and returns zero if the readback matches (suggesting the chip is really there) or a nonzero `errno` otherwise. For an `ns16550`, the probe writes a test pattern to the scratch register and reads it back.

`init` is called to initialise the chip to a known state. The arguments after the `bas` are `baudrate`, `databits`, `stopbits`, and `parity`. The driver computes the divisor, writes the divisor-latch-access bit, writes the divisor, clears the divisor-latch, sets the line control register for the requested data/stop/parity configuration, enables the FIFOs, and enables the chip's interrupts. The exact register sequence for a 16550 is several dozen lines of code and is documented in the chip's data sheet.

`term` is called to shut down the chip. It typically disables interrupts, flushes the FIFOs, and leaves the chip in a safe state.

`putc` sends a single character. This is used by the low-level console path and by polling-based diagnostic output. The driver busy-waits on the transmitter-holding-register-empty flag, then writes the byte to the transmit register.

`rxready` returns nonzero if at least one byte is available to read. The driver reads the line status register and checks the data-ready bit.

`getc` reads a single character. Used by the low-level console for input. The driver busy-waits on the data-ready flag (or the caller ensures `rxready` just returned true), then reads the receive register.

These six methods are the entire hardware-specific surface for a UART driver at the low level. Everything else (interrupt handling, buffering, TTY integration, hot-plug of PCIe UARTs, console selection) is provided by the framework. A new UART driver is, in effect, a six-function implementation plus a handful of declarations.

26.7.5 A Closer Look at ns8250

The `ns8250` driver at `/usr/src/sys/dev/uart/uart_dev_ns8250.c` is the best place to see these methods concretely. It is a mature, production-grade driver that handles every variant of the 8250/16450/16550/16550A family. The register definitions it uses (from `/usr/src/sys/dev/ic/ns16550.h`) are the same ones every UART-related header in the PC world uses. When you read this driver, you are reading, in effect, the reference implementation of a 16550 driver for FreeBSD.

The put-character implementation is instructive for its simplicity:

```
static void
ns8250_putc(struct uart_bas *bas, int c)
{
    int limit;

    limit = 250000;
```

```
while ((uart_getreg(bas, REG_LSR) & LSR_THRE) == 0 && --limit)
    DELAY(4);
uart_setreg(bas, REG_DATA, c);
uart_barrier(bas);
limit = 250000;
while ((uart_getreg(bas, REG_LSR) & LSR_TEMT) == 0 && --limit)
    DELAY(4);
}
```

The loop polls the line status register (LSR) for the transmitter-holding-register-empty (THRE) flag. When it is set, the transmit holding register is ready to accept a byte. The driver writes the byte to the data register (REG_DATA) and then polls again for the transmitter-empty (TEMT) flag to ensure the byte has been shifted out before returning.

The `uart_barrier` call is a memory barrier that ensures the write to the data register is visible to the hardware before subsequent reads. On platforms with weak memory ordering, missing this barrier would cause intermittent data loss.

The `DELAY(4)` yields four microseconds per iteration, and the `limit` counter is 250,000. Together, they give a one-second timeout before the loop gives up. For a real UART, 250,000 iterations is a cap that should never be reached in normal operation; it is a safety net for the pathological case where the chip is in an unexpected state.

The probe is equally direct:

```
static int
ns8250_probe(struct uart_bas *bas)
{
    u_char val;

    /* Check known 0 bits that don't depend on DLAB. */
    val = uart_getreg(bas, REG_IIR);
    if (val & 0x30)
        return (ENXIO);
    return (0);
}
```

Bits 4 and 5 of the Interrupt Identification Register (IIR) are defined as always-zero for every variant of the 16550 family. If those bits read as one, this is not a real 16550 register, and the probe rejects the address.

You could read the whole driver in an afternoon. What you would come away with is a clear mental model: the methods are narrow, the framework is large, and the real engineering is in handling the quirks of specific chip revisions (a FIFO bug in the 16550 predecessor, an erratum in some PC chipsets, a signal-detect issue on certain Oxford devices). A new UART driver for a well-behaved chip is genuinely a small file.

26.7.6 The `uart_softc` and How the Framework Uses It

Each instance of a UART driver has a struct `uart_softc`, defined in `/usr/src/sys/dev/uart/uart_bus.h`. The framework allocates one per attached port. Its important fields include a pointer to the `uart_bas` that describes the port's register layout, the I/O resources (the IRQ, the memory range or I/O port range), the TTY device attached to this port, the current line parameters, and two byte buffers (RX and TX) that the framework uses internally. The driver does not usually allocate its own `softc`; it uses the framework's `uart_softc`, with the hardware-specific extensions added through `kobj` class inheritance.

When the framework receives an interrupt from a UART, it calls a framework-internal function that reads the interrupt-identification register, decides what kind of work the chip has requested (transmit-ready, receive-data-available, line-status, modem-status), and dispatches to the appropriate handler. The handlers pull data out of the chip's RX FIFO into the framework's RX ring buffer, or push data from the framework's TX ring buffer into the chip's TX FIFO, or update state variables in response to modem-signal changes. The interrupt handler returns, and the TTY layer consumes the ring buffers at its own pace through the framework's `put-character` and `get-character` paths.

This is why the driver's `uart_ops` table is so small. The high-volume work (moving bytes between the chip and the ring buffers) is handled by shared framework code that reads the chip's registers through `uart_getreg` and `uart_setreg`. The driver only needs to expose the low-level primitives; the composition is done for it.

26.7.7 Integration with the TTY Layer

Above the `uart(4)` framework sits the TTY layer, defined in `/usr/src/sys/kern/tty.c` and friends. A UART port in FreeBSD appears to userland as two `/dev` nodes:

- `/dev/ttyuN`: the callin node. Opening it blocks until a carrier detect signal is asserted (which models an incoming call on a modem). It is used for devices that answer, not initiate, connections.
- `/dev/cuaun`: the callout node. Opening it does not wait for carrier detect. It is used for devices that initiate connections, or for developers who want to talk to a serial port without pretending it is a modem.

The distinction is historical, dating from the era when serial ports were genuinely connected to analog modems with separate “someone is calling” and “I am initiating a call” semantics. FreeBSD preserves the distinction because some embedded and industrial workflows still rely on it, and because the implementation cost is minimal once the TTY layer's “two sides of the same port” pattern is in place.

The TTY layer calls into the `uart(4)` framework through a `ttysw` structure whose methods map neatly onto UART operations. The important entries include:

- `tsw_open`: called when userland opens the port. The framework enables interrupts, powers on the chip, and applies the default `termios`.
- `tsw_close`: called when the last userland reference is released. The framework drains the TX buffer, disables interrupts (unless the port is also a console), and puts the chip in an idle state.
- `tsw_ioctl`: called for `ioctls` the TTY layer does not handle itself. Most UART-specific `ioctls` are handled by the framework.
- `tsw_param`: called when `termios` changes. The framework reprograms the chip's baud rate, data bits, stop bits, parity, and flow control.

- `tsw_outwakeup`: called when there is new data to transmit. The framework enables the transmit-ready interrupt if it was disabled; on the next IRQ, the framework pushes bytes from the ring buffer into the chip.

You do not usually have to write any of these. The framework in `uart_tty.c` implements them once for every UART driver. Your driver's only contribution is the six methods in `uart_ops`.

26.7.8 The termios Interface in Practice

When a user runs `stty 115200` on a serial port, the following chain of calls happens:

1. `stty(1)` opens the port and issues a `TIOCSETA` ioctl carrying the new `struct termios`.
2. The kernel TTY layer receives the ioctl and updates its internal copy of the port's `termios`.
3. The TTY layer calls `tsw_param` on the port's `ttydevsw`, passing the new `termios`.
4. The `uart(4)` framework's `uart_param` implementation looks at the `termios` fields (`c_ispeed`, `c_ospeed`, `c_cflag` with its `CSIZE`, `CSTOPB`, `PARENB`, `PARODD`, `CRTSCTS` sub-bits) and calls the driver's `init` method with the corresponding raw values.
5. The driver's `init` method computes the divisor, writes the line-control register, reconfigures the FIFO, and returns.

None of this requires the driver to know about `termios`. The translation from `termios` bits to raw integers is done by the framework. The driver sees only the raw values: baudrate in bits per second, databits (usually 5 through 8), stopbits (1 or 2), and a parity code.

This separation is what lets FreeBSD run the same `uart(4)` framework on top of radically different chips. A 16550 driver and a PL011 driver both implement the same six `uart_ops` methods. The `termios`-to-raw translation happens once, in framework code, for every chip family.

26.7.9 Flow Control at the Register Level

Hardware flow control is typically driven by two signals on the UART: CTS (clear to send) and RTS (request to send). When CTS is asserted by the remote side, it is telling the local transmitter "I am ready for more data." When the local side asserts RTS, it is telling the remote transmitter the same thing. When either signal is not asserted, the corresponding transmitter pauses.

In a 16550, RTS is driven by a bit in the modem control register (MCR), and CTS is read from a bit in the modem status register (MSR). The framework exposes flow control through `termios` (`CRTSCTS` flag), through ioctls (`TIOCMGET`, `TIOCMSET`, `TIOCMBS`, `TIOCMBSIC`), and through automatic responses to FIFO fill levels.

When the receive FIFO fills past a threshold, the driver deasserts RTS to ask the remote side to stop transmitting. When the FIFO drains below a different threshold, the driver reasserts RTS. When the modem-status-interrupt fires because CTS changed, the interrupt handler enables or disables the transmit path accordingly. All of this is framework logic; the driver only exposes the register-level primitives.

Software flow control (XON/XOFF) is handled entirely in the TTY layer, by inserting and interpreting the XON (0x11) and XOFF (0x13) bytes in the data stream. The UART driver has no role in it.

26.7.10 The Interrupt Handler Path in Detail

Beyond the six `uart_ops` methods, a real UART driver usually implements an interrupt handler. The framework provides a generic one in `uart_core.c` that works for the vast majority of chips, but the driver can supply its own for chips with unusual behaviour. To understand what the framework's generic handler does, and when you might want to override it, it helps to trace the handler's path.

When the hardware interrupt fires, the framework's ISR reads the interrupt identification register (IIR) through `uart_getreg`. The IIR encodes which of four conditions triggered the interrupt: line-status (a framing error or overrun occurred), received-data-available (at least one byte is in the receive FIFO), transmitter-holding-register-empty (the TX FIFO wants more data), or modem-status (a modem signal changed state).

For line-status interrupts, the framework logs a warning (or increments a counter) and continues.

For received-data-available, the framework reads bytes out of the chip's RX FIFO one at a time, pushing each into the driver's internal RX ring buffer. The loop continues until the receive-data-available flag clears. Once the ring buffer has bytes, the framework signals the TTY layer's input path, which will pull bytes out as the consumer is ready.

For transmitter-holding-register-empty, the framework pulls bytes out of its internal TX ring buffer and pushes them into the chip's TX FIFO one at a time. The loop continues until the TX FIFO is full or the ring buffer is empty. Once the ring buffer is empty, the framework disables the transmit interrupt so the chip does not keep firing; the next `tsw_outwakeup` call (from the TTY layer, when there is new data) will reenable it.

For modem-status changes, the framework updates its internal modem-signal state and signals the TTY layer if the change is significant (for example, CTS deassertion when hardware flow control is enabled).

This is all done in interrupt context with the driver's mutex held. The mutex is a spin mutex (`MTX_SPIN`) for UART drivers, because taking a sleepable mutex in an interrupt handler is forbidden. The framework's helpers know this and use appropriate primitives.

When might a driver want to override the generic handler? Three situations come to mind.

First, if the chip has unusual FIFO semantics. Some chips do not clear their interrupt identification registers in the obvious way; you have to drain the FIFO completely, or you have to read a specific register to acknowledge. If your chip's data sheet describes such a quirk, you override the handler with chip-specific logic.

Second, if the chip has DMA support you want to use. The framework's generic handler is PIO (programmed I/O): one byte per register access. A chip with a DMA engine could move many bytes per interrupt, significantly reducing CPU overhead at high baud rates. Implementing DMA requires chip-specific code.

Third, if the chip has hardware timestamping or other advanced features. Some embedded UARTs can timestamp individual received bytes with microsecond precision, which is invaluable for industrial protocols. The framework does not know about this, so the driver must implement it.

For typical hardware, the generic handler is correct and performant. Do not override it without a specific reason.

26.7.11 The TX and RX Ring Buffers

The `uart(4)` framework keeps two ring buffers inside each port's `softc`. These are separate from any buffering on the chip itself: even if the chip has a 64-byte FIFO, the framework has its own ring buffers of some

configurable size (typically 4 KB for each direction) that sit between the chip and the TTY layer.

The purpose of these ring buffers is to absorb bursts. Suppose the consumer of data is slow (a busy userland process), and the producer (the remote serial device) is pushing data at 115200 baud. Without a ring buffer, the chip's 64-byte FIFO would fill up in about 6 milliseconds, and bytes would be lost. With a 4 KB ring buffer, the buffer can absorb a 350-millisecond burst at 115200 baud, which is enough for userland to catch up in almost every realistic scenario.

The sizes of these ring buffers are not generally configurable per-driver; they are baked into the framework. The ring buffer implementation is in `uart_core.c` and uses the same kind of head/tail pointer arithmetic as the ring buffers in our USB echo driver.

When the TTY layer asks for bytes (through `ttydisc_rint`), the framework moves bytes out of the RX ring into the TTY layer's own input queue, which has its own buffering and line-discipline processing (canonical mode, echo, signal generation, and so on). When userland writes bytes, they arrive at the framework's `tsw_outwakeup` path and are moved into the TX ring; the framework's transmit-empty interrupt handler pushes them from the ring into the chip.

This arrangement has a nice property: the driver, the framework, and the TTY layer are all loosely coupled. The driver only knows about the chip. The framework only knows about registers and ring buffers. The TTY layer only knows about buffering and line discipline. Each layer can be tested and reasoned about independently.

26.7.12 Debugging Serial Drivers

When a serial driver does not work, the symptoms can be confusing. Bytes go in, bytes come out, but the two do not match. The clock ticks, but the characters look like gibberish. The port opens, but writes return zero bytes. This section lists the diagnostic techniques that help.

Log aggressively at attach. Use `device_printf(dev, "attached at %x, IRQ %d\n", ...)` to verify the address and IRQ your driver ended up with. If the address is wrong, no I/O will work; if the IRQ is wrong, no interrupts will fire. Attach messages are the first line of defence.

Use `sysctl dev.uart.0.*` to inspect port state. The `uart(4)` framework exports many per-port knobs and statistics through `sysctl`. Reading them shows the current baud rate, the number of bytes transmitted, the number of overruns, the modem signal state, and more. If `tx` is incrementing but `rx` is not, the transmitter works but the receiver does not; if both are zero, nothing is happening at all.

Probe the hardware with `kgdb`. If you have a kernel crash dump or the ability to attach a kernel debugger, you can inspect the `uart_softc` directly and read its register values. This is invaluable when the chip is in a confused state that the software abstraction hides.

Compare against a working driver. If your modification broke something, bisect the change against the upstream `ns8250.c`. The difference will be small, and once you identify it, the fix is usually clear.

Use `dd` for small, repeatable tests. Instead of `cu` for debugging, use `dd if=/dev/zero of=/dev/cuau0 bs=1 count=100` to write exactly 100 bytes. Then `dd if=/dev/cuau0 of=output.bin bs=1 count=100` to read exactly 100 bytes (with a suitable timeout or a second open). This isolates timing and character-encoding issues that interactive `cu` might mask.

Check the hardware flow control pins. Many flow-control bugs are hardware, not software. Use a break-out board, a multimeter, or an oscilloscope to verify that DTR, RTS, CTS, and DSR are at the voltages you expect.

If one is stuck floating, the chip's behaviour is undefined.

Compare behaviour under `nmdm(4)`. If your userland tool works with `nmdm(4)` but not with your driver, the bug is in the driver. If it fails with both, the bug is in the tool.

These techniques apply equally to `uart(4)` drivers and `ucom(4)` drivers. The difference is that `uart(4)` problems often come down to register manipulation (did you set the divisor correctly?), while `ucom(4)` problems often come down to USB transfers (did the control transfer to set the baud rate actually succeed?). The debugging tools (USB: `usbconfig`, transfer statistics; UART: `sysctl`, chip registers) are different, but the investigative mindset is the same.

26.7.13 Writing a UART Driver Yourself

Putting the pieces together, a minimal UART driver for an imaginary register-compatible chip would be organised like this:

1. Define register offsets and bit positions in a local header.
2. Implement the six `uart_ops` methods: `probe`, `init`, `term`, `putc`, `rxready`, `getc`.
3. Declare a `struct uart_ops` initialised with those six methods.
4. Declare a `struct uart_class` initialised with the ops and the hardware parameters (range, reference clock, register shift).
5. Implement the interrupt handler if the chip needs more than the framework's default dispatch.
6. Register the driver with Newbus using the standard macros.

Most new UART drivers in the tree are small. Oxford single-port PCIe UARTs, for example, are a few hundred lines because they are fundamentally 16550-compatible and only need a thin layer of PCI-specific attach code. Complex ones like the Z8530 are larger because the chip has a more complicated programming model; the driver size tracks the chip's complexity, not the framework's.

26.7.14 Looking at `myfirst_uart.c` in Skeleton Form

For our running example, the skeleton of a minimal UART driver looks like this:

```
#include <sys/param.h>
#include <sys/systm.h>
#include <sys/bus.h>
#include <sys/module.h>
#include <sys/kernel.h>

#include <dev/uart/uart.h>
#include <dev/uart/uart_cpu.h>
#include <dev/uart/uart_bus.h>

#include "uart_if.h"

static int    myfirst_uart_probe(struct uart_bas *);
```

```

static void myfirst_uart_init(struct uart_bas *, int, int, int, int);
static void myfirst_uart_term(struct uart_bas *);
static void myfirst_uart_putc(struct uart_bas *, int);
static int myfirst_uart_rxready(struct uart_bas *);
static int myfirst_uart_getc(struct uart_bas *, struct mtx *);

static struct uart_ops myfirst_uart_ops = {
    .probe    = myfirst_uart_probe,
    .init     = myfirst_uart_init,
    .term     = myfirst_uart_term,
    .putc     = myfirst_uart_putc,
    .rxready  = myfirst_uart_rxready,
    .getc     = myfirst_uart_getc,
};

struct myfirst_uart_softc {
    struct uart_softc base;
    /* any chip-specific state would go here */
};

static kobj_method_t myfirst_uart_methods[] = {
    /* Most methods inherit from the framework. */
    { 0, 0 }
};

struct uart_class myfirst_uart_class = {
    "myfirst_uart class",
    myfirst_uart_methods,
    sizeof(struct myfirst_uart_softc),
    .uc_ops    = &myfirst_uart_ops,
    .uc_range  = 8,
    .uc_rclk   = 1843200,
    .uc_rshift = 0,
};

```

The inclusion of `uart_if.h` is notable: that header is generated at build time by the `kobj` machinery from the interface definition in `/usr/src/sys/dev/uart/uart_if.m`. It declares the method prototypes that the framework expects drivers to implement. When you write a new driver, you depend on this header.

The six methods themselves are straightforward once you have the chip's programming manual open. `init` computes the divisor from `uc_rclk` and the baud rate, writes the line control register for the databits/stopbits/parity combination, enables FIFOs, and sets the interrupt enable register to the desired mask. `term` inverts `init`. `putc`, `getc`, and `rxready` each do a single-register access plus a spin on the status register.

A complete implementation of all six methods for a 16550-compatible chip is about three hundred lines. For a chip with quirks, it might grow to five hundred or more. The `ns8250` driver is longer than most because it

handles errata and variant detection for dozens of real chips, but the core logic of its six methods is still the standard pattern.

26.7.15 The ucom(4) Framework: USB-to-Serial Bridges

Not every serial port is a real UART on the system bus. Many are USB adapters: a PL2303, a CP2102, an FTDI FT232, a CH340G. These chips expose a serial port over USB, and FreeBSD's approach to supporting them is a small framework called ucom(4). It lives in `/usr/src/sys/dev/usb/serial/`, alongside the drivers for each chip family.

ucom(4) is distinct from uart(4). It does not use `uart_ops`, it does not use `uart_bas`, and it does not use the ring buffers inside `uart_core.c`. What it does is provide a TTY abstraction on top of USB transfers. A ucom(4) client declares itself through a struct `ucom_callback`:

```
struct ucom_callback {
    void (*ucom_cfg_get_status)(struct ucom_softc *, uint8_t *, uint8_t *);
    void (*ucom_cfg_set_dtr)(struct ucom_softc *, uint8_t);
    void (*ucom_cfg_set_rts)(struct ucom_softc *, uint8_t);
    void (*ucom_cfg_set_break)(struct ucom_softc *, uint8_t);
    void (*ucom_cfg_set_ring)(struct ucom_softc *, uint8_t);
    void (*ucom_cfg_param)(struct ucom_softc *, struct termios *);
    void (*ucom_cfg_open)(struct ucom_softc *);
    void (*ucom_cfg_close)(struct ucom_softc *);
    int (*ucom_pre_open)(struct ucom_softc *);
    int (*ucom_pre_param)(struct ucom_softc *, struct termios *);
    int (*ucom_ioctl)(struct ucom_softc *, uint32_t, caddr_t, int,
                     struct thread *);
    void (*ucom_start_read)(struct ucom_softc *);
    void (*ucom_stop_read)(struct ucom_softc *);
    void (*ucom_start_write)(struct ucom_softc *);
    void (*ucom_stop_write)(struct ucom_softc *);
    void (*ucom_tty_name)(struct ucom_softc *, char *pbuf, uint16_t buflen,
                          uint16_t unit, uint16_t subunit);
    void (*ucom_poll)(struct ucom_softc *);
    void (*ucom_free)(struct ucom_softc *);
};
```

The methods divide into three groups. Configuration methods (names prefixed with `ucom_cfg_`) are called to change the state of the underlying chip: set DTR, set RTS, change the baud rate, and so on. These methods run in the framework's configuration thread, which is designed for making synchronous USB control requests. Flow methods (`ucom_start_read`, `ucom_start_write`, `ucom_stop_read`, `ucom_stop_write`) are called to enable or disable the data path on the underlying USB channels. The pre-methods (`ucom_pre_open`, `ucom_pre_param`) run on the caller's context before the framework schedules a configuration task, which is where a driver validates userland-supplied arguments and returns an `errno` if they are unacceptable. The `ucom_ioctl` method translates chip-specific userland ioctls that the framework does not handle into USB requests.

A USB-to-serial driver's job is to implement these callbacks in terms of USB transfers. When `ucom_cfg_param` is called with a new baud rate, the driver issues a vendor-specific control transfer that programs the chip's baud-rate register. When `ucom_start_read` is called, the driver starts a bulk-IN channel that delivers incoming bytes. When `ucom_start_write` is called, the driver starts a bulk-OUT channel that flushes outgoing bytes.

The FTDI driver at `/usr/src/sys/dev/usb/serial/uftdi.c` is the concrete reference. Its `ucom_cfg_param` implementation translates the `termios` fields into the FTDI's proprietary baud-rate divisor format (which is weird, because FTDI chips use a sub-integer divisor scheme that is almost but not quite standard 16550) and issues a control transfer to `bRequest = FTDI_SIO_SET_BAUD_RATE`. Its `ucom_start_read` starts the bulk-IN channel that reads from the FTDI's RX FIFO. Its `ucom_start_write` starts the bulk-OUT channel that writes to the FTDI's TX FIFO.

From userland's perspective, a `ucom(4)` device looks identical to a `uart(4)` device. Both appear as `/dev/ttyuN` and `/dev/cuaN`. Both respond to `stty`, `cu`, `tip`, `minicom`, and every other serial tool. Both support the same `termios` flags. The distinction only matters to a driver writer.

26.7.16 Reading `uftdi.c` As a Complete Example

FTDI chips (FT232R, FT232H, FT2232H, and many others) are the most widely deployed USB-to-serial chips in the embedded world. If you ever work with microcontrollers, evaluation boards, 3D printers, or industrial sensors, you will encounter FTDI hardware. FreeBSD has supported FTDI since 4.x, and the current driver lives in `/usr/src/sys/dev/usb/serial/uftdi.c`. At roughly three thousand lines, it is not short, but most of that length is devoted to the large match table (FTDI products are legion) and to chip-variant quirks (every few years FTDI adds a new FIFO size, a new baud-rate divisor scheme, or a new register). The pedagogically interesting core is a few hundred lines, and reading it is a direct reward for the conceptual work of Section 4.

When you open the file, the first thing to notice is the enormous match table. FTDI assigns OEM-specific USB IDs to their customers, so the match table includes not just FTDI's own VID/PID pairs but also hundreds of VIDs and PIDs from companies that embed FTDI chips in their products. Sparkfun, Pololu, Olimex, Adafruit, various industrial vendors: every one has at least one entry in the `uftdi` match table. The `STRUCT_USB_HOST_ID` array is a few hundred entries long, grouped with comments indicating which product family each cluster belongs to.

The `softc` comes next. An FTDI `softc` includes the USB device pointer, a mutex, the transfer array for the bulk-IN and bulk-OUT channels (FTDI devices use bulk for data, not interrupt), a `struct ucom_super_softc` for the `ucom(4)` layer, a `struct ucom_softc` for the per-port state, and FTDI-specific fields: the current baud-rate divisor, the current line control register contents, the current modem control register contents, and a few flags for the variant family (FT232, FT2232, FT232H, and so on). Each variant requires slightly different code for some operations, so the driver keeps a variant identifier in the `softc` and branches on it in the operations that differ.

The transfer configuration array is where the FTDI driver's interaction with the USB framework is declared. It declares two channels: `UFTDI_BULK_DT_RD` for incoming data and `UFTDI_BULK_DT_WR` for outgoing. Each is a `UE_BULK` transfer with a moderate buffer size (the FTDI default is 64 bytes for low-speed and 512 bytes for full-speed, and the driver picks the right size at attach based on the chip variant). The callbacks are `uftdi_read_callback` and `uftdi_write_callback`, and they follow the three-state pattern exactly as described in Section 3.

The `ucom_callback` structure is the next important block. It wires the FTDI driver into the `ucom(4)` framework. The methods it provides include `uftdi_cfg_param` (called when the baud rate or byte format

changes), `uftdi_cfg_set_dtr` (called to assert or deassert DTR), `uftdi_cfg_set_rts` (same for RTS), `uftdi_cfg_open` and `uftdi_cfg_close` (called when a userland process opens or closes the device), and `uftdi_start_read`, `uftdi_start_write`, `uftdi_stop_read`, `uftdi_stop_write` (called to enable or disable the data channels). Each configuration method translates a high-level operation into a USB control transfer to the FTDI chip.

The baud-rate programming is one of the most instructive parts of the driver, because FTDI chips use a peculiar divisor scheme. Rather than the clean integer divisors a 16550 UART uses, FTDI supports a fractional divisor where the numerator is an integer and the denominator is computed from two bits that select one-eighth, one-quarter, three-eighths, one-half, or five-eighths. The function `uftdi_encode_baudrate` takes a requested baud rate and the chip's reference clock and computes the closest valid divisor. It handles the edge cases (very low baud rates, very high baud rates on newer chips, standard rates like 115200 which are exactly representable, nonstandard rates like 31250 used by MIDI). The resulting sixteen-bit value is passed to `uftdi_set_baudrate`, which issues a control transfer to the FTDI's baud-rate register.

The line control register (data bits, stop bits, parity) is programmed through a similar sequence: the `termios` structure arrives at `uftdi_cfg_param`, the driver extracts the relevant bits, encodes them into the FTDI's line-control format, and issues a control transfer.

The modem control signals (DTR, RTS) are programmed through `uftdi_cfg_set_dtr` and `uftdi_cfg_set_rts`. These are the simplest transfers: a control-out with no payload, which the chip interprets as “set DTR to X” or “set RTS to Y.”

The data path is in the two callbacks. `uftdi_read_callback` handles the bulk-IN channel. On `USB_ST_TRANSFERRED`, it extracts the received bytes from the USB frame (ignoring the first two bytes, which are FTDI status bytes) and feeds them into the `ucom(4)` layer for delivery to userland. On `USB_ST_SETUP`, it rearms the read for another buffer. `uftdi_write_callback` handles the bulk-OUT channel. On `USB_ST_SETUP`, it asks the `ucom(4)` layer for more data, copies it into a USB frame, and submits the transfer. On `USB_ST_TRANSFERRED`, it rearms to check for more data.

Reading through `uftdi.c` with Section 4 vocabulary in hand, you can see how the entire `ucom(4)` framework pattern is instantiated for a specific chip. The FTDI-specific logic (baud-rate encoding, line-control encoding, modem-control setting) is isolated into helper functions. The framework integration is handled by the `ucom_callback` structure. The data flow is handled by the two bulk transfers. If you were writing a driver for a different USB-to-serial chip, you would copy this structure and change the chip-specific parts.

The existence of this driver explains something that might otherwise be puzzling. Why did FreeBSD add `ucom(4)` as a separate framework rather than as part of `uart(4)`? Because the entire data-path machinery of a `uart(4)` driver (interrupt handlers, ring buffers, register accesses) has no analogue in a USB-to-serial world. The FTDI chip's “FIFO” is an on-chip buffer that the driver cannot directly access; it can only send bulk packets to the chip and receive them back. The `uart(4)` machinery would be unused overhead. By having `ucom(4)` as a separate framework with its own data-path abstractions, FreeBSD can make a USB-to-serial driver like `uftdi` weigh just a few hundred lines of core logic rather than wrap an unnecessary layer of 16550 emulation.

When you finish reading `uftdi.c`, open up `lcom.c` (the Prolific PL2303 driver) and `uslcom.c` (the Silicon Labs CP210x driver) in sequence. They follow the same structure with different chip-specific details. After reading all three, you will have a working understanding of how a USB-to-serial driver is organised in FreeBSD, and you will be ready to write one for any chip you encounter.

26.7.17 Choosing Between `uart(4)` and `ucom(4)`

The choice is mechanical. If the chip sits on the system bus (PCI, ISA, a platform I/O port, a memory-mapped SoC peripheral), you write a `uart(4)` driver. If the chip sits on USB and exposes a serial interface, you write a `ucom(4)` driver.

The two frameworks do not mix. You cannot take a `uart(4)` driver and plug it into USB, and you cannot take a `ucom(4)` driver and attach it to PCIe. They are independent implementations of the same user-visible abstraction (a TTY port), but with very different internals.

Beginners sometimes ask why the two frameworks exist at all, instead of a unified serial framework with a pluggable transport layer. The answer is historical: `uart(4)` was rewritten in its modern form in the early 2000s to replace the older `sio(4)` driver, and at that time USB serial support was a set of ad-hoc drivers. When USB serial support was unified, the natural approach was to add a thin TTY-integration layer (`ucom(4)`) rather than retrofit `uart(4)`. The two are now independent because decoupling them has been stable and useful. A unification effort would be a significant project with modest payoff.

For your purposes as a beginning driver writer, the rule is simple. If you are writing a driver for a chip that lives on your motherboard's serial ports or on a PCIe card, use `uart(4)`. If you are writing a driver for a USB dongle that pretends to be a serial port, use `ucom(4)`. The reference drivers for each case (`ns8250` for `uart(4)`, `uftdi` for `ucom(4)`) are the right places to learn the details.

26.7.18 Differences Between Chip Variants

Working with real UART hardware quickly teaches you that “16550-compatible” is a spectrum, not a fixed specification. Here are the variants you are most likely to encounter and the differences that matter.

8250. The original, from the late 1970s. Has no FIFO; every received byte must be collected by the CPU before the next arrives. Software for 16550A will usually work, with reduced performance.

16450. Like 8250 but with some register improvements and slightly more reliable behaviour. Still no FIFO.

16550. Introduced a 16-byte FIFO, but the original 16550 had buggy FIFO behaviour. Software should detect this and refuse to use the FIFO in the bad case.

16550A. Fixed the FIFO bugs. This is the canonical “16550” that every PC serial driver targets. Reliable, widely compatible.

16550AF. Further revisions for clocking and margin. For software purposes, identical to 16550A.

16650. Extended the FIFO to 32 bytes and added automatic hardware flow control. Mostly 16550A-compatible.

16750. Extended the FIFO to 64 bytes. Some chips with this label also have additional autobaud and high-speed modes. Software must decide whether to enable the extended FIFO.

16950 (Oxford Semiconductor). A 128-byte FIFO, additional flow-control features, and support for unusual baud rates through a modified divisor scheme. Often seen on high-performance PCIe serial cards.

UART-compatible SoC controllers. Many embedded processors have built-in UARTs that are register-compatible with 16550 but with quirks: some have different clock rates, some have different register offsets, some have DMA support, some have different interrupt semantics. The `ns8250` driver in FreeBSD probes for these variants during attach and adjusts its behaviour accordingly.

The `ns8250` driver's probe logic reads several registers to determine which variant is present. It checks the IIR bits we saw earlier, reads the FIFO control register to see what FIFO size is reported, checks for 16650/16750/16950 identification markers, and records the result in a variant field in the `softc`. The body of the driver then branches on this field at a few places where the variants differ.

When you write a driver for a new UART, decide upfront whether you want to target a single variant or a family. Targeting a single variant is simpler but limits the hardware you can support. Targeting a family requires variant detection logic like `ns8250`'s.

26.7.19 The Console Path

FreeBSD can use a serial port as the console. This is especially useful for embedded systems that do not have a display, for servers that do not have a keyboard and monitor, and for kernel debugging (so that `printf` output goes somewhere visible even when the display driver is broken).

The console path is tightly integrated with `uart(4)`. A UART that is designated as the console is probed early in boot, before most of the kernel is initialised. The console's `putc` and `getc` methods are used to emit boot messages and to read boot-time keyboard input. Only after the full kernel is up does the UART get attached to the TTY layer in the normal way.

Two mechanisms select which port is the console. The boot loader can set a variable (typically `console=comconsole`) in the environment, which the kernel reads at startup. Alternatively, the kernel can be configured at build time with a specific port as the console (via options `UART_EARLY_CONSOLE` in a kernel configuration file).

When a port is the console, it stays active across driver unload and detach. You cannot unload `uart` or disable the console port without losing console output. This constraint is enforced in the `uart(4)` framework and is usually invisible to driver writers (you do not need to special-case the console port), but it is worth knowing about in case you see console-related oddities during testing.

26.7.20 Comparing UART Drivers Across Architectures

One of FreeBSD's strengths is that the same `uart(4)` framework works across multiple architectures. An `x86_64` laptop with a 16550 on a PCIe card, an `aarch64` Raspberry Pi with a PL011 on-chip UART, and a `riscv64` development board with a SiFive-specific UART all expose the same TTY interface to userland. Only the driver differs.

Here is a quick survey of the UART drivers in FreeBSD 14.3:

- `uart_dev_ns8250.c`: the 16550 family for `x86` and many other platforms.
- `uart_dev_pl011.c`: the ARM PrimeCell PL011 UART, used on Raspberry Pi and many ARM SoCs.
- `uart_dev_imx.c`: the NXP i.MX UART, used on i.MX-based ARM boards.
- `uart_dev_z8530.c`: the Zilog Z8530, historically used on SPARC workstations.
- `uart_dev_ti8250.c`: a TI variant of the 16550 with additional features.
- `uart_dev_pl011.c` (sbsa variant): the SBSA-standardised ARM UART for server-class ARM hardware.
- `uart_dev_snps.c`: the Synopsys DesignWare UART, used on many RISC-V boards.

Open any two of these and compare their `uart_ops` implementations side by side. The structure is identical: six methods, each pointing at a function that reads or writes chip-specific registers. The chip-specific details differ, but the framework's API is the same.

This is the payoff of the layered design. A new UART driver is a contained project: a few hundred lines of code, reusing all the buffering and TTY integration from the framework. If FreeBSD had to reimplement buffering for every UART, the system would be much larger and much harder to verify.

26.7.21 What About the USB CDC ACM Standard?

USB has a standard class for serial devices, called CDC ACM (Communication Device Class, Abstract Control Model). Chips that implement CDC ACM advertise themselves with a specific class/subclass/protocol triple at the interface level, and they can be driven by a single generic driver rather than a vendor-specific one. FreeBSD's generic CDC ACM driver is `u3g.c` in `/usr/src/sys/dev/usb/serial/`, and it is also built on top of `ucom(4)`.

Many modern USB serial chips implement CDC ACM, so the generic driver just works for them without a vendor-specific file. Others (like FTDI) use proprietary protocols that require a vendor-specific driver. The class/subclass/protocol triple in the interface descriptor is what tells you which case you are in; `usbconfig -d ugenN.M dump_all_config_desc` will show it.

When you are shopping for a USB serial adapter for development work, prefer chips that implement CDC ACM. They are cheaper, more portable, and do not require proprietary drivers. FTDI chips are historically dominant in embedded development because of their reliability, and FreeBSD supports them well, but a modern CP2102 or CH340G running in CDC ACM mode is equally usable.

26.7.22 Wrapping Up Section 4

Section 4 has given you a complete picture of how serial drivers work in FreeBSD. You have seen the layering: `uart(4)` at the framework level, `ttydevsw` at the TTY integration level, `uart_ops` at the hardware level. You have seen the distinction between `uart(4)` for bus-attached UARTs and `ucom(4)` for USB-to-serial bridges, and the practical rule for deciding which to use. You have seen, at a high level, the six hardware methods a UART driver implements, the configuration callbacks a USB-to-serial driver implements, and how the TTY layer sits on top of both with one uniform interface to userland.

The level of depth in this section is necessarily lighter than the USB-side sections, because serial drivers in FreeBSD are more specialised than USB drivers and you are more likely to read an existing one (or modify one) than to write a brand-new one from scratch. If you do find yourself writing a new UART driver for a custom board, the path is clear: open `ns8250` in one window, open your chip's data sheet in another, and write the six methods one by one.

Two key takeaways frame Section 5. First, testing serial drivers does not require real hardware. FreeBSD ships a `nmdm(4)` null-modem driver that creates pairs of virtual TTYs you can wire together, letting you exercise `termios` changes, flow control, and data flow without plugging in anything. Second, testing USB drivers without hardware is harder but not impossible: you can use QEMU with USB redirection to test against real devices through a VM, or you can use FreeBSD's USB gadget mode to make one machine present itself as a USB device to another. Section 5 covers both. The goal is to enable a development loop that does not depend on cable-handling and on plugging things in and out.

26.8 Section 5: Testing USB and Serial Drivers Without Real Hardware

26.8.1 Why This Section Exists

A beginning driver writer often gets stuck at the same obstacle. They write a driver, compile it, want to try it, and discover they do not have the hardware, the hardware is behaving badly, the hardware is on the wrong machine, or the iteration loop of “change code, plug it in, see what happens, unplug it, change code again” is painfully slow and unreliable. Section 5 addresses this directly. FreeBSD provides several mechanisms that let you exercise driver code paths without physical hardware, and knowing these mechanisms will save you hours of frustration.

The goal is not to pretend hardware is present when it is not. The goal is to give you tools that cover the parts of driver development where hardware is incidental, so that when you do plug in real hardware, you already know your code path logic is correct and you are only validating the physical interaction. Debugging a register-level quirk is faster when you know that your locking, your state machines, and your user interface are already sound.

This section covers four such mechanisms: the `nmdm(4)` null-modem driver for serial testing, basic userland tools for exercising TTYs (`cu`, `tip`, `stty`, `comcontrol`), QEMU with USB redirection for USB driver testing, and FreeBSD’s USB gadget mode for presenting one machine as a USB device to another. It ends with a short discussion of techniques that do not require any special tooling: unit tests at the functional layer, logging discipline, and assertion-driven development.

26.8.2 The `nmdm(4)` Null-Modem Driver

`nmdm(4)` is a kernel module that creates pairs of linked virtual TTYs. When you write to one side, it comes out the other side, exactly as if you had connected two real serial ports with a null-modem cable. The driver is in `/usr/src/sys/dev/nmdm/nmdm.c`, and it is loaded with:

```
# kldload nmdm
```

Once loaded, you can instantiate pairs on demand simply by opening them. Run:

```
# cu -l /dev/nmdm0A
```

This opens the A side of pair 0. On another terminal, run:

```
# cu -l /dev/nmdm0B
```

Whatever you type into one `cu` session will appear in the other. You have now created a pair of virtual TTYs, with no hardware involved. You can change baud rates with `stty` and the change will be noticed on both sides. You can assert DTR and CTS through `ioctl`s and see the effect on the other side.

The utility of `nmdm(4)` for driver development is twofold. First, if you are writing a TTY-layer user (say, a driver that spawns a shell on a virtual TTY, or a userland program that implements a protocol over a TTY), you

can test it end-to-end against `nmdm(4)` without any hardware. Second, if you are writing a `ucom(4)` or `uart(4)` driver, you can compare its behaviour to `nmdm(4)`'s behaviour by running the same userland test against both. If your driver misbehaves where `nmdm(4)` does not, the bug is in your driver; if both misbehave, the bug is probably in your userland test.

A small caveat: `nmdm(4)` does not simulate baud rate delays. Whatever you write comes out the other side at memory speed. This is usually what you want (you do not want to wait through a real 9600-baud transmission for a hundred-kilobyte test payload), but it does mean that timing-sensitive protocols cannot be tested with `nmdm(4)` alone.

26.8.3 The `cu(1)`, `tip(1)`, and `stty(1)` Toolbox

Whether you are using `nmdm(4)`, a real UART, or a USB-to-serial dongle, the userland tools you use to interact with a TTY are the same. The most important three are `cu(1)`, `tip(1)`, and `stty(1)`.

`cu` is the classic “call up” program. It opens a TTY, puts the terminal into raw mode, and lets you type bytes to the port and see bytes coming back. To open a port at a specific baud rate:

```
# cu -l /dev/cuau0 -s 115200
```

The `-l` argument specifies the device, and `-s` specifies the baud rate. `cu` supports a handful of escape sequences (all starting with `~`) for exiting, sending files, and similar operations; `~.` is the standard “exit” escape and `~?` lists the others.

`tip` is a related tool with similar semantics but a different configuration mechanism. `tip` reads `/etc/remote` for named connection entries and can take a name argument rather than a device path. For most purposes, `cu` and `tip` are interchangeable; `cu` is more convenient for one-off use.

`stty` prints or changes the termios parameters of a TTY. Run `stty -a -f /dev/ttyu0` to see every termios flag on the port. Run `stty 115200 -f /dev/ttyu0` to set the baud rate. Run `stty cs8 -parenb -cstopb -f /dev/ttyu0` to set eight data bits, no parity, one stop bit (the most common configuration in modern embedded work). The manual page is extensive, and the flags map almost directly onto the bits of `c_cflag`, `c_iflag`, `c_lflag`, and `c_oflag` in the `termios` struct.

Using these three tools together gives you a flexible way to poke at your driver from userland. You can change settings with `stty`, open the port with `cu`, send and receive bytes, close the port, check the state with `stty` again, and repeat. If your driver's `tsw_param` implementation has a bug, `stty` will expose it: the settings you set will not read back correctly, or the port will behave differently than requested.

26.8.4 The `comcontrol(8)` Utility

`comcontrol` is a specialised utility for serial ports. It sets port-specific parameters that are not exposed through `termios`. The two most important are the `drainwait` and the `specific-RS-485` options. For beginner driver testing, the more common use is inspecting port state: `comcontrol /dev/ttyu0` shows the current modem signals (DTR, RTS, CTS, DSR, CD, RI) and the current `drainwait`. You can also set the signals:

```
# comcontrol /dev/ttyu0 dtr rts
```

sets DTR and RTS. This is useful for testing flow-control handling without writing a custom program.

26.8.5 The `usbconfig(8)` Utility

On the USB side, `usbconfig(8)` is the Swiss Army knife. You used it at the end of Section 1 to inspect a device's descriptors. Several other subcommands are useful during driver development:

- `usbconfig list`: list all attached USB devices.
- `usbconfig -d ugenN.M dump_all_config_desc`: print every descriptor for a device.
- `usbconfig -d ugenN.M dump_device_quirks`: print any quirks applied by the USB framework.
- `usbconfig -d ugenN.M dump_stats`: print per-transfer statistics.
- `usbconfig -d ugenN.M suspend`: put the device into the USB suspend state.
- `usbconfig -d ugenN.M resume`: wake it up.
- `usbconfig -d ugenN.M reset`: physically reset the device.

The `reset` command is particularly useful during development. A driver under test can easily leave a device in a confused state; `usbconfig reset` puts the device back to the just-plugged-in condition without requiring a physical unplug.

26.8.6 Testing USB Drivers with QEMU

QEMU, the generic CPU emulator, has strong USB support. You can run a FreeBSD guest inside QEMU and redirect real host USB devices into the guest. This is the single most useful technique for USB driver development, because it lets you test against real hardware while retaining all the iteration speed of working inside a VM.

On a FreeBSD host, install QEMU from ports:

```
# pkg install qemu
```

Install a FreeBSD guest image into a disk file (the mechanics are covered in Chapter 4 and Appendix A). When you boot the guest, add USB redirection options:

```
qemu-system-x86_64 \  
-drive file=freebsd.img,format=raw \  
-m 1024 \  
-device nec-usb-xhci,id=xhci \  
-device usb-host,bus=xhci.0,vendorid=0x0403,productid=0x6001
```

The `-device nec-usb-xhci` line adds a USB 3.0 controller to the guest. The `-device usb-host` line attaches a specific USB device from the host (identified by vendor and product) to that controller. When the guest boots, the device appears on the guest's USB bus and can be enumerated by the guest's kernel.

This setup gives you the full iteration loop inside the VM. You can load your driver, unload it, reload a rebuilt version, all without physically handling any cables. You can use serial console or networking to interact with the VM. You can snapshot the VM state before a risky test and revert if the test panics.

The main limitation is USB isochronous support, which is less stable across emulators. For bulk, interrupt, and control transfers (the three types most drivers use), QEMU USB redirection is reliable enough to be your primary development environment.

26.8.7 FreeBSD USB Gadget Mode

If QEMU is not available and you have two FreeBSD machines, there is another option: `usb_template(4)` and the dual-role USB support on some hardware let you make one machine present itself as a USB device to another. The host machine sees a normal USB peripheral; the gadget machine is actually running the device side of the USB protocol.

This is an advanced topic and the hardware support is variable. On x86 platforms with USB-on-the-Go-capable chipsets, on some ARM boards, and on specific embedded configurations, the setup works. On most desktop hardware, it does not. The gory details are in `/usr/src/sys/dev/usb/template/` and in the `usb_template(4)` manual page.

If you have the hardware to use this technique, it is the closest thing to a full end-to-end USB driver test without physical peripherals. If you do not, do not pursue it for a learning project; use QEMU instead.

26.8.8 Techniques That Do Not Require Special Tooling

Beyond the frameworks above, there are several techniques that rely only on good driver design.

First, design your driver so that the hardware-independent parts can be unit-tested in userland. If your driver has a protocol parser, a state machine, or a checksum calculator, factor those into functions that take plain C buffers and return plain C results. You can then compile those functions into a userland test program and run them against known inputs. This catches many bugs before they reach the kernel.

Second, log aggressively during development and quietly in production. The `DLOG_RL` macro from Chapter 25 is your friend: it lets you emit frequent diagnostic messages during development, with a `sysctl` to suppress them in production. Rate-limiting prevents log storms if something goes wrong.

Third, use assertions for invariants. `KASSERT(cond, ("message", args...))` will panic the kernel if `cond` is false, but only in `INVARIANTS` kernels. You can run your driver in an `INVARIANTS` kernel during development and in a production kernel later, without changing the code. The Chapter 20 discussion of `INVARIANTS` is the reference.

Fourth, be rigorous about concurrency testing. Use `INVARIANTS` plus `WITNESS` (which tracks lock ordering) during development. If your driver has a locking bug that almost always works but occasionally deadlocks, `WITNESS` will catch it on the first occurrence.

Fifth, write a simple userland client for your driver and use it as part of your development loop. Even a ten-line program that opens the device, writes a known string, reads a known response, and checks the result is enormously useful. You can run it in a loop during stress testing, you can run it with `ktrace -f cmd` to get a trace of system calls, and you can run it under a debugger if something surprises you.

26.8.9 A Walkthrough of QEMU USB Redirection

QEMU's USB support is the single most useful tool for USB driver development, so a more detailed walkthrough is in order. Suppose you want to develop a driver for a specific FT232 adapter. Your host is a FreeBSD 14.3 machine, and you want to run your driver on a guest FreeBSD 14.3 VM inside QEMU.

First, install QEMU and create a guest disk image:

```
# pkg install qemu
# truncate -s 16G guest.img
```

Install FreeBSD into the image. The exact procedure is covered in Appendix A, but the short version is: boot a FreeBSD installer ISO as the CD-ROM, install onto the disk image, reboot.

Once the guest is installed, locate the host USB device you want to redirect. Plug in the FT232 and note the vendor and product IDs from `usbconfig list`:

```
ugen0.3: <FTDI FT232R USB UART> at usb0
```

`usbconfig -d ugen0.3 dump_device_desc` will show `idVendor = 0x0403` and `idProduct = 0x6001`.

Now start QEMU with USB redirection:

```
qemu-system-x86_64 \
  -enable-kvm \
  -cpu host \
  -m 2048 \
  -drive file=guest.img,format=raw \
  -device nec-usb-xhci,id=xhci \
  -device usb-host,bus=xhci.0,vendorid=0x0403,productid=0x6001 \
  -net user -net nic
```

The `-device nec-usb-xhci` line adds a USB 3.0 controller to the VM. The `-device usb-host` line redirects the matching host device into the VM. When the VM boots, the FT232 will appear as if it were plugged directly into the VM's USB port.

Inside the VM, run `dmesg` and look for the USB attach:

```
uhub0: 4 ports with 4 removable, self powered
uftdi0 on usb0
uftdi0: <FTDI FT232R USB UART, class 255/0, rev 2.00/6.00, addr 2> on usb0
```

Your driver (whether `uftdi` or your own work-in-progress) will see a real FT232 with real descriptors, real transfer behaviour, and real quirks. You can unload and reload your driver inside the VM without disconnecting anything; you can run kernel with `INVARIANTS` and `WITNESS` without worrying about host-side impact; you can snapshot the VM and revert if a test goes badly.

A few subtleties to be aware of with USB redirection:

- Only one consumer can claim a USB device at a time. If you redirect a device into a VM, the host loses access to it until the VM releases it. This matters if you are redirecting something like a USB keyboard or mouse; choose a spare device for development.
- USB isochronous transfers have some quirks in QEMU. They work, but timing can be slightly off. For most driver development, you will be working with bulk, interrupt, and control transfers, so this is rarely a concern.
- Some host controllers (particularly xHCI) can reset under heavy I/O. If your driver behaves strangely under stress testing, try with a different `-device` type (`uhci`, `ehci`, `xhci`) to see whether the issue is in your driver or in the emulated controller.
- USB 3.0 SuperSpeed transfers are more reliable with `-device nec-usb-xhci`. Older `-usb` flag-based controllers are limited to USB 2.0.

When the VM is running, the iteration cycle becomes: edit code on the host, copy to the VM (or mount a shared directory), build inside the VM, load, test, reload, repeat. A Makefile with a `test`: target that does all of this can cut iteration time to tens of seconds.

26.8.10 Using devd(8) During Development

`devd(8)` is FreeBSD's device-event daemon. It reacts to kernel notifications about device attach and detach and can run configured commands in response. During driver development, `devd` is useful in two ways.

First, it can auto-load your module when a matching device is plugged in. If your module is in `/boot/modules/` and your `USB_PNP_HOST_INFO` is set, `devd` will run `kldload` automatically when it sees a device that would match.

Second, it can run diagnostic commands on attach. A `/etc/devd.conf` entry like:

```
attach 100 {
    device-name "myfirst_usb[0-9]+";
    action "logger -t myfirst-usb 'device attached: $device-name'";
};
```

will write a log line every time a `myfirst_usb` device attaches. For more elaborate diagnostics, you can invoke your own shell script that dumps state, starts userland consumers, or sends notifications.

During development, a useful pattern is to have `devd` open a `cu` session to a newly attached `ucom` device, so you can exercise the driver the moment it attaches:

```
attach 100 {
    device-name "cuaU[0-9]+";
    action "setsid screen -dmS usb-serial cu -l /dev/$device-name -s 9600";
};
```

This runs the test in a detached screen session, which you can later attach to with `screen -r usb-serial`.

26.8.11 Writing a Simple Userland Test Harness

Most driver bugs are exposed by actually running the driver against userland. Even a short test program catches more bugs than reading the driver's code carefully. For our echo driver, a minimal test program looks like:

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

int
main(int argc, char **argv)
{
    int fd;
    const char *msg = "hello";
    char buf[64];
    int n;

    fd = open("/dev/myfirst_usb0", O_RDWR);
    if (fd < 0) {
        perror("open");
        return (1);
    }

    if (write(fd, msg, strlen(msg)) != (ssize_t)strlen(msg)) {
        perror("write");
        close(fd);
        return (1);
    }

    n = read(fd, buf, sizeof(buf) - 1);
    if (n < 0) {
        perror("read");
        close(fd);
        return (1);
    }
    buf[n] = '\0';
    printf("got %d bytes: %s\n", n, buf);

    close(fd);
    return (0);
}
```

Compile with `cc -o lab03-test lab03-test.c`. Run with `./lab03-test`. The expected output is “got 5

bytes: hello”.

Extensions to this test harness that catch more bugs:

- Loop the open/write/read/close cycle a thousand times. Memory leaks and resource leaks show up after a few hundred iterations.
- Fork multiple processes and have them all read/write concurrently. Race conditions manifest as random data corruption or deadlocks.
- Intentionally kill the test process mid-transfer. Driver-side state machines sometimes get confused when a userland consumer disappears unexpectedly.
- Send random-length writes (1 byte, 10 bytes, 100 bytes, 1 KB, 10 KB). Edge cases around short and long transfers are where many subtle bugs live.

Build these extensions incrementally. Each one will probably reveal a bug the previous version did not; each bug you fix will make your driver more robust.

26.8.12 Logging Patterns for Development

During development, you want verbose logging. In production, you want silence. The pattern from Chapter 25 (DLOG_RL with a sysctl to control verbosity) carries over unchanged to USB and UART drivers. Define a rate-limited logging macro that compiles to a no-op in production builds, and sprinkle it through every branch that might be interesting during debugging:

```
#ifdef MYFIRST_USB_DEBUG
#define DLOG(sc, fmt, ...) \
    do { \
        if (myfirst_usb_debug) \
            device_printf((sc)->sc_dev, fmt "\n", ##__VA_ARGS__); \
    } while (0)
#else
#define DLOG(sc, fmt, ...) ((void)0)
#endif
```

Then in the callback:

```
case USB_ST_TRANSFERRED:
    DLOG(sc, "bulk read completed, actlen=%d", actlen);
    ...
```

Control myfirst_usb_debug through a sysctl:

```
static int myfirst_usb_debug = 0;
SYSCTL_INT(_hw_myfirst_usb, OID_AUTO, debug, CTLFLAG_RWTUN,
    &myfirst_usb_debug, 0, "Enable debug logging");
```


Now you can turn logging on and off at runtime with `sysctl hw.myfirst_usb.debug=1`. During development, turn it on. During stress tests, turn it off (logging rate-limiting helps, but zero logging is even cheaper). During post-mortem analysis of a bug, turn it on and reproduce.

26.8.13 A Test-Driven Workflow for Chapter 26

For the hands-on labs coming in the next section, a good workflow looks like this:

1. Write the driver code.
2. Compile it. Fix build errors.
3. Load it in a test VM. Observe `dmesg` for attach failures.
4. Run a small userland client that exercises the driver's I/O paths.
5. Unload. Make a change. Go back to step 2.
6. Once the driver behaves well in the VM, test it on real hardware as a sanity check.

Most of the time spent on this loop is in steps 1 through 4. Real hardware testing in step 6 is a validation step, not an iteration step. If you try to iterate on real hardware, you will waste time on plug-unplug cycles and on recovering from accidental misconfigurations; the VM saves you this.

A fresh install of FreeBSD in a small VM, configured to boot quickly and to have your driver's build directory mounted as a shared filesystem, is a highly productive development environment. Spending half a day to set one up pays back many times over in the days that follow.

26.8.14 Wrapping Up Section 5

Section 5 has given you the tools to develop USB and serial drivers without being tied to specific physical hardware. `nmdm(4)` covers the serial-port side for any test that does not need a real modem. QEMU USB redirection covers the USB side for nearly any driver you might write. The `cu`, `tip`, `stty`, `comcontrol`, and `usbconfig` utilities give you the userland tools to exercise driver code paths by hand. And the general techniques, from factoring hardware-independent code into userland-testable functions to using `INVARIANTS` and `WITNESS` for kernel-time correctness checking, work regardless of what transport you are writing for.

Having reached the end of Section 5, you have everything you need to start writing real USB and serial drivers for FreeBSD 14.3. The conceptual models, the code skeletons, the transfer mechanics, the TTY integration, and the testing environment are all in place. What remains is practice, which is the purpose of the next section.

26.9 Common Patterns Across Transport Drivers

Now that we have walked through USB and serial in detail, it is worth stepping back and noting the patterns that recur. These patterns appear in network drivers (Chapter 28), block drivers (Chapter 27), and most other transport-specific drivers in FreeBSD. Recognising them saves time when you read a new driver.

26.9.1 Pattern 1: Match Table, Probe, Attach, Detach

Every transport driver begins with a match table describing which devices it supports. Every transport driver has a probe method that tests a candidate against the match table and returns zero or `ENXIO`. Every transport

driver has an attach method that takes ownership of a matched device and allocates all per-device state. Every transport driver has a detach method that releases everything the attach method allocated, in reverse order.

The specifics vary. USB match tables use `STRUCT_USB_HOST_ID`. PCI match tables use `pcidev(9)` entries. ISA match tables use resource descriptions. The content differs, but the structure is identical.

When you read a new driver, the first thing to find is the match table. It tells you what hardware the driver supports. The second thing to find is the attach method. It tells you what resources the driver owns. The third thing to find is the detach method. It tells you the shape of the driver's resource hierarchy.

26.9.2 Pattern 2: Softc As The Single Source of Per-Device State

Every transport driver has a per-device softc. Every piece of mutable state lives in the softc. No global variables are used to hold per-device state (global configuration like module flags is fine). This pattern keeps multi-device drivers correct without surprise.

The softc's size is declared in the driver structure. The framework allocates and frees the softc automatically. The driver accesses it through `device_get_softc(dev)` inside Newbus methods and through whatever framework helper (like `usbd_xfer_softc`) is appropriate in callbacks.

Adding a new feature to a driver often means adding a new field to the softc, a new initialisation step in attach, a new cleanup step in detach, and whatever code uses the field in between. When you structure changes this way, you rarely forget to clean things up, because the shape of the change makes the cleanup step obvious.

26.9.3 Pattern 3: Labelled-Goto Cleanup Chain

When an attach method has to allocate several resources, each allocation has a failure path that unwinds all previous allocations. The labelled-goto chain from Chapter 25 implements this uniformly. Every resource has a label corresponding to "the state where this resource has been successfully allocated." A failure at any point jumps to the label for the state just before, which cleans up in reverse order.

This pattern is not aesthetically pleasing to some programmers (C's goto has a bad reputation), but it is pragmatically the cleanest way to handle an arbitrary number of cleanup steps in C. Alternatives like wrapping each resource in a separate function with its own cleanup are often more verbose. Alternatives like setting a flag per resource and testing it in a common cleanup routine add error-prone state management.

Whatever you think of goto, FreeBSD drivers use the labelled-goto pattern, and new drivers are expected to follow the convention.

26.9.4 Pattern 4: Frameworks Hide Transport Details

Each transport has a framework that hides transport-specific details behind a uniform API. The USB framework hides DMA buffer management behind `usb_page_cache` and `usbd_copy_in/out`. The UART framework hides interrupt dispatching behind `uart_ops`. The network framework (Chapter 28) will hide packet buffer management behind `mbufs` and `ifnet(9)`.

The value of these frameworks is that drivers become smaller and more portable. A 200-line UART driver that supports dozens of chip variants would be impossible without the framework. A 500-line USB driver that supports a complex protocol like USB audio would likewise be out of reach.

When you read a new driver, the parts you find most dense are usually the chip-specific logic. The parts that seem almost absent (the transfer scheduling, the buffer management, the TTY integration) are where the framework is doing its work.

26.9.5 Pattern 5: Callbacks with State Machines

The USB callback's three-state machine (`USB_ST_SETUP`, `USB_ST_TRANSFERRED`, `USB_ST_ERROR`) is the canonical example, but similar patterns appear in other transport drivers. A network driver's transmit completion callback has a similar structure. A block driver's request completion callback is similar. The framework calls the driver back at well-defined moments, and the driver uses a state machine to decide what to do.

Learning to read these state machines is learning a universal driver-reading skill. The specific states differ from framework to framework, but the pattern is recognisable.

26.9.6 Pattern 6: Mutexes and Wakeups

Every driver protects its softc with a mutex. Userland-facing code (read, write, ioctl) takes the mutex while manipulating softc fields. Callback code runs with the mutex held (the framework acquires it before calling). Userland code releases the mutex before sleeping and reacquires it after waking. Wakeup calls from callbacks release any sleeper waiting on the relevant channel.

The specifics vary by transport, but the pattern is universal. Modern FreeBSD drivers are uniformly multi-threaded and multi-CPU safe, which requires disciplined locking.

26.9.7 Pattern 7: Errno-Returning Helpers

Chapter 25 introduced the errno-returning helper function pattern: every internal function returns an integer errno (zero for success, nonzero for failure). Callers check the return value and propagate failure up through the stack. The attach method accumulates successful helpers in the labelled-goto chain; each helper's failure triggers the cleanup corresponding to its position.

This pattern requires discipline. Every helper must be consistent; no helper can return a "success value" that varies in meaning, and no helper can use global state to communicate failure. When followed rigorously, the pattern produces drivers where the control flow is legible and the error paths are easy to audit.

26.9.8 Pattern 8: Version Declarations and Module Dependencies

Every driver module declares its own version with `MODULE_VERSION`. Every driver module declares its dependencies with `MODULE_DEPEND`. Dependencies are versioned ranges (minimum, preferred, maximum), which allows parallel development of framework and driver to proceed without lockstep releases.

When a new major version of a framework is released with breaking API changes, the version range in `MODULE_DEPEND` is how a driver expresses "I work with framework v1 or v2, but not v3." The kernel module loader refuses to load a driver whose dependencies cannot be satisfied, which prevents many classes of silent breakage.

26.9.9 Pattern 9: Cross-Framework Layering

Some drivers sit on top of multiple frameworks. A USB-to-Ethernet driver sits on top of `usbdi(9)` (for USB transfers) and `ifnet(9)` (for network interface semantics). A USB-to-serial driver sits on top of `usbdi(9)` and `ucom(4)`. A USB mass-storage driver sits on top of `usbdi(9)` and CAM.

When you write a cross-framework driver, the structure is: you write callbacks for each framework, and you orchestrate the interaction between them in your driver's helper code. The framework on top of which you sit defines how userland sees your driver. The framework below handles the transport.

Reading `uftdi.c` showed you this pattern: the driver is a USB driver (it uses `usbdi(9)`) and a serial driver (it uses `ucom(4)`), and the orchestration between the two is the heart of the file.

26.9.10 Pattern 10: Early Attach Deferral

Some drivers cannot finish their attach work synchronously. For example, a driver might need to read a configuration EEPROM that takes a few hundred milliseconds, or it might need to wait for a PHY to autonegotiate a link. These drivers use a deferred-attach pattern: the Newbus attach method queues a taskqueue task that does the slow work, then returns quickly.

This pattern keeps the system boot fast (no single driver holds up boot by taking a long time in attach) and lets drivers do their work asynchronously. The caller must be aware that attach “finishing” does not mean the device is fully usable; a separate “ready” state has to be polled or signalled.

For USB and UART drivers, attach is usually fast enough that deferral is not needed. For more complex drivers (network cards in particular), deferral is common. Chapter 28 will show an example.

26.9.11 Pattern 11: Separate Data Path and Control Path

In every transport driver, two conceptual paths exist: the control path (configuration, state changes, error recovery) and the data path (the actual bytes moving through the device). Most drivers structure these as separate code paths, sometimes with separate locking.

The control path is low-bandwidth and infrequent. It can afford heavy locking and synchronous calls. The data path is high-bandwidth and continuous. It must be optimised for throughput: minimal locking, no synchronous calls, efficient buffer management.

The USB framework keeps them naturally separate: configuration through `usbd_transfer_setup` and control transfers; data through bulk and interrupt transfers. The UART framework likewise: configuration through `tsw_param`; data through the interrupt handler and ring buffers. Network drivers have the most pronounced separation: configuration through `ioctl`s; data through the TX and RX queues.

Reading a new driver, knowing this separation exists helps you parse what each code block is doing. A function with extensive locking and error handling is probably control path. A function with short, tight code and careful buffer management is probably data path.

26.9.12 Pattern 12: Reference Drivers

Every transport in FreeBSD has one or two “canonical” reference drivers that illustrate the patterns correctly and thoroughly. For USB, `uled.c` and `uftdi.c` are the references. For UART, `uart_dev_ns8250.c` is the reference. For networking, `em` (Intel Ethernet) and `rl` (Realtek) are the references. For block devices, `da` (direct-access storage) is the reference.

When you need to understand how to write a new driver in an existing transport, the reference driver is the right place to start. Do not try to understand the framework from its code alone; that is too abstract. Start from a working driver and let it ground your understanding.

26.10 Hands-On Labs

These labs give you a chance to turn the reading into muscle memory. Each lab is designed to fit in a single sitting, ideally under an hour. They assume a FreeBSD 14.3 lab environment (either on physical hardware or inside a virtual machine), root access, and a working build environment as described in Chapter 3. The companion files for every lab in this chapter are available under `examples/part-06/ch26-usb-serial/` in the book’s repository.

The labs build on each other but do not strictly depend on each other. You can skip a lab and come back to it later without losing continuity. The first three labs focus on USB; the last three focus on serial. Each lab has the same structure: a short summary, the steps, expected output, and a “what to watch for” note that highlights the learning goal.

26.10.1 Lab 1: Exploring a USB Device with `usbconfig`

This lab exercises the descriptor vocabulary from Section 1 by inspecting real USB devices on your machine. It does not involve writing any code.

Goal. Read the descriptors of three different USB devices and identify their interface class, the number of endpoints, and the endpoint types.

Requirements. A FreeBSD system with at least three USB devices plugged in. If you only have one machine and few USB ports, a USB hub with a few small peripherals (mouse, keyboard, flash drive) is ideal.

Steps.

1. Run `usbconfig list` as root. Record the `ugenN.M` identifiers of three devices.
2. For each device, run:

```
# usbconfig -d ugenN.M dump_all_config_desc
```

Read through the output. Identify the `bInterfaceClass`, `bInterfaceSubClass`, and `bInterfaceProtocol` for each interface. For each endpoint in each interface, record the `bEndpointAddress` (including direction bit), the `bmAttributes` (including transfer type), and the `wMaxPacketSize`.

3. Build a small table. For each device, write down: vendor ID, product ID, interface class (with name from the USB class list), number of endpoints, and the transfer type of each endpoint.

4. Match your table against `dmesg`. Confirm that the driver that claimed each device makes sense given the interface class you recorded.
5. Optional: repeat the exercise for a device you have not seen before (someone else's keyboard, a USB audio interface, a game controller). The more variety you see, the faster descriptor reading becomes.

Expected output. A filled-in table with at least three rows. The exercise is successful if you can answer, for any device in the table: "What class of driver would handle this?"

What to watch for. Pay attention to devices that expose multiple interfaces. A webcam, for example, often has an audio interface (for the microphone) in addition to its video interface. A multi-function printer might expose a printer interface, a scanner interface, and a mass-storage interface. Noticing these is what trains your eye for the multi-interface logic in the probe method.

26.10.2 Lab 2: Building and Loading the USB Driver Skeleton

This lab walks through building the skeleton driver from Section 2, loading it, and observing its behaviour when a matching device is plugged in.

Goal. Compile and load `myfirst_usb.ko`, and observe its attach and detach messages.

Requirements. The build environment from Chapter 3. The files under `examples/part-06/ch26-usb-serial/lab02-usb-skeleton/`. A USB device whose vendor/product you can match. For development, a VOTI/OBDEV test VID/PID (0x16c0/0x05dc) is free to use; otherwise, pick a cheap prototyping device (like an FT232 breakout board) and adjust the match table to match its IDs.

Steps.

1. Enter the lab directory:

```
# cd examples/part-06/ch26-usb-serial/lab02-usb-skeleton
```

2. Read `myfirst_usb.c` and `myfirst_usb.h`. Identify the match table, the probe method, the attach method, the `softc`, and the detach method. For each, trace how it relates to the Section 2 walkthrough.
3. Build the module:

```
# make
```

You should see `myfirst_usb.ko` created in the build directory.

4. Load the module:

```
# kldload ./myfirst_usb.ko
```

Run `kldstat | grep myfirst_usb` to confirm the module is loaded.

5. Plug in a matching device. Observe `dmesg`. You should see a line like:

```
myfirst_usb0: <Vendor Product> on uhub0
myfirst_usb0: attached
```

If the device does not match, nothing will happen. In that case, open `usbdevs` on the target machine, find the vendor/product of a device you do have, and edit the match table accordingly. Rebuild, reload, and try again.

6. Unplug the device. Observe `dmesg`. You should see the kernel remove the device. Your detach does not log anything explicitly in this minimal skeleton, but you can add a `device_printf(dev, "detached\n")` if you want confirmation.
7. Unload the module:

```
# kldunload myfirst_usb
```

Expected output. Attach messages in `dmesg` when the device is plugged in. Clean unload with no panics when the module is removed.

What to watch for. If `kldload` fails with an error about symbol lookups, you probably forgot a `MODULE_DEPEND` line or misspelled a symbol name. If `attach` is never called but the device is definitely present, the match table is wrong: check the vendor and product IDs in `usbconfig_list` and verify they match what you wrote in `myfirst_usb_devs`. If `attach` is called but fails, check `device_printf` output for the failure reason.

26.10.3 Lab 3: A Bulk Loopback Test

This lab adds the transfer mechanics from Section 3 to the skeleton from Lab 2 and sends a few bytes through a USB device that implements a loopback protocol. It is the first lab that actually moves data.

Goal. Add a bulk-OUT and bulk-IN channel to the driver, write a small userland client that sends a string and reads it back, and observe the roundtrip.

Requirements. A USB device that implements bulk loopback. The simplest such device for development is a USB gadget controller running a loopback program (possible on some ARM boards and on some development kits). If you do not have one, you can substitute a simpler exercise: attach the driver to a USB flash drive, open one of its `ugen` endpoints, and simply armed-submit-complete a single read transfer. The loop will fail (because flash drives do not echo data), but the mechanics of setup and submission will run correctly.

Steps.

1. Copy `lab02-usb-skeleton` to `lab03-bulk-loopback` as a working copy.
2. Add the bulk channels to the driver. Paste the config array from Section 3, the callback functions, and the userland interaction. Make sure the `/dev` entry your driver creates supports `read(2)` and `write(2)`, which are what the lab test program uses.
3. Rebuild and reload the module.
4. Run the userland client:

```
# ./lab03-test
```

which you will find alongside the driver in the lab directory. The program opens `/dev/myfirst_usb0`, writes “hello”, reads up to 16 bytes, and prints them. If loopback works, the output is “hello”.

5. Observe `dmesg` for any stall warnings or error messages.

Expected output. “hello” echoed back. If the remote device does not implement loopback, the read will return after the channel’s timeout with no data, which is also a valid test outcome for the purposes of exercising the state machine.

What to watch for. The most common mistake in this lab is mismatched endpoint directions. Remember: `UE_DIR_IN` means “the host reads from the device” and `UE_DIR_OUT` means “the host writes to the device”. If you swap them, the transfers will fail with stalls. Watch also for missing locking around the userland read/write handlers; if you manipulate the transmit queue without the softc mutex held, you can race with the write callback and see bytes disappear.

26.10.4 Lab 4: A Simulated Serial Driver with `nmdm(4)`

This lab is not about writing a driver; it is about learning the userland half of serial testing. The results will inform how you approach Lab 5 and how you debug any TTY-layer work in the future.

Goal. Create a pair of `nmdm(4)` virtual ports, observe how data flows, and exercise `stty` and `comcontrol` to see how termios and modem signals work.

Requirements. A FreeBSD system. No special hardware.

Steps.

1. Load the `nmdm` module:

```
# kldload nmdm
```

2. In terminal A, open the A side:

```
# cu -l /dev/nmdm0A -s 9600
```

3. In terminal B, open the B side:

```
# cu -l /dev/nmdm0B -s 9600
```

4. Type in terminal A. Observe that the characters appear in terminal B. Type in terminal B; they appear in terminal A.

5. Exit `cu` in both terminals (type `~.`). In a third terminal, run:

```
# stty -a -f /dev/nmdm0A
```

Read through the output. Notice `9600` for the baud rate, `cs8 -parenb -cstopb` for the byte format, and various flags for line discipline.

6. Change the baud rate on one side:

```
# stty 115200 -f /dev/nmdm0A
```

Then open the ports again with `cu -s 115200`. The baud rate change is visible, even though `nmdm(4)` does not actually wait for serialised bits.

7. Run:

```
# comcontrol /dev/ttyu0A
```

...or rather, the equivalent for the `nmdm` character devices. The `nmdm` pairs do not always have `comcontrol`-visible modem signals, depending on the FreeBSD version; if your version does not, skip this step.

Expected output. Text appears on the opposite side. `stty` shows termios flags. You now have a reproducible way to test TTY-layer behaviour on your machine.

What to watch for. The pair identifiers (0, 1, 2...) are implicit and allocated on first open. If you cannot open `/dev/nmdm5A` because nothing has opened `/dev/nmdm4A` yet, this is expected: pairs are created lazily in increasing order. Also note that `cu` uses a lock file in `/var/spool/lock/`; if you kill `cu` abruptly, the lock file may persist and prevent reopens. Delete it manually if you get a “port in use” error.

26.10.5 Lab 5: Talking to a Real USB-to-Serial Adapter

This lab brings real hardware into the loop. You will use a USB-to-serial adapter (an FT232, a CP2102, a CH340G, or anything else FreeBSD supports) and a terminal program to exercise the full path from `ucom(4)` through the TTY layer to userland.

Goal. Plug in a USB-to-serial adapter, verify it attaches, and use `cu` to send data to it (perhaps by looping the TX and RX pins together with a jumper).

Requirements. A USB-to-serial adapter. A jumper wire (if you want to do a hardware loopback) or a second serial device to talk to (a development board, an embedded computer, or an old serial modem).

Steps.

1. Plug in the adapter. Run `dmesg | tail` and confirm it attaches. You should see lines like:

```
uftdi0 on uhub0
uftdi0: <FT232R USB UART, class 0/0, ...> on usb0
```

and a `ucomN: <...>` line just after that.

2. Run `ls -l /dev/cuaU*`. The adapter’s port is usually `/dev/cuaU0` for the first adapter, `/dev/cuaU1` for the second, and so on. (Note the capital-U suffix, which distinguishes USB-provided ports from the real UART ports at `/dev/cua0`.)
3. Put a jumper wire between the TX and RX pins of the adapter. This creates a hardware loopback: whatever the adapter transmits comes back on its own RX line.
4. In one terminal, set the baud rate:

```
# stty 9600 -f /dev/cuaU0
```

5. Open the port with `cu`:

```
# cu -l /dev/cuaU0 -s 9600
```

Type characters. Every character you type should appear twice: once as local echo (if your terminal is echoing), and once as the character coming back through the loopback. Disable the local echo in `cu` if it is confusing; the `stty -echo` will help.

6. Remove the jumper. Type characters. Now they will not come back, because there is nothing connected to RX.
7. Exit `cu` with `~.`. Unplug the adapter. Run `dmesg | tail` and verify clean detach.

Expected output. Characters are echoed back when the jumper is in place and lost when it is not. The `dmesg` shows clean attach and detach.

What to watch for. If the adapter attaches but no `cuau` device appears, the underlying `ucom(4)` instance may have attached but failed to create its TTY. Check `dmesg` for errors. If characters come out garbled, the baud rate is probably wrong: make sure every stage of the path (your terminal, `cu`, `stty`, the adapter, and the far end) is set to the same rate. On older hardware, some USB-to-serial adapters do not reset their internal configuration when you open them; you may need to explicitly set the baud rate with `stty` before `cu` will work correctly.

26.10.6 Lab 6: Observing Hot-Plug Lifecycle

This lab does not require writing any new driver code. It exercises the hot-plug lifecycle we described conceptually in Section 1 and in code in Section 2, using the existing `uhid` or `ukbd` driver as the test subject.

Goal. Plug in and unplug a USB device repeatedly while monitoring kernel logs, observing the full attach/detach sequence.

Requirements. A USB device you can plug and unplug without disrupting your work session. A USB flash drive or a USB mouse are both safe; a USB keyboard is not (because detaching a keyboard in the middle of a session can strand your shell).

Steps.

1. Open a terminal window and run:

```
# tail -f /var/log/messages
```

or, if your system does not log kernel messages to that file:

```
# dmesg -w
```

(The `-w` flag is a FreeBSD 14 addition that streams new kernel messages as they arrive.)

2. Plug in your USB device. Observe the messages. You should see:

- A message from the USB controller about the new device appearing.
- A message from `uhub` about the port powering up.
- A message from the class driver that matched the device (e.g., `ums0` for a mouse, `umass0` for a flash drive).
- Possibly a message from the higher-level subsystem (e.g., `da0` for a mass-storage device).

3. Unplug the device. Observe the messages. You should see:

- A message from `uhub` about the port powering down.
- A detach message from the class driver.

4. Repeat several times. Watch that every attach is matched by a detach. Watch that no message is missed. Watch the timing; the attach sequence can take tens or hundreds of milliseconds because enumeration involves several control transfers.

5. Write a tiny shell loop that records the attach and detach times:

```
# dmesg -w | awk '/ums|umass/ { print systime(), $0 }'
```

(Adjust the regex for the device type you are using.) This gives you a machine-readable log of attach and detach timestamps.

Expected output. Clean attach and detach every time, with no dangling state.

What to watch for. Occasionally you will see a device attach and then immediately detach within a few hundred milliseconds. This usually indicates the device is failing enumeration: either a bad cable, insufficient power, or a buggy device firmware. If it happens consistently with one device, try a different USB port or a powered hub. Also watch for cases where the kernel reports a stall during enumeration; these are rarely harmful but indicate that the enumeration needed multiple tries.

26.10.7 Lab 7: Building a ucom(4) Skeleton from Scratch

This lab is an extended one that combines the USB and serial material from the chapter. You will build a minimal ucom(4) driver skeleton that presents itself as a serial port but is backed by a simple USB device.

Goal. Build a ucom(4) driver skeleton that attaches to a specific USB device, registers with the ucom(4) framework, and provides empty implementations of the key callbacks. The driver will not actually talk to the hardware, but it will exercise the full ucom(4) registration path.

Requirements. The materials from Lab 2 (the USB driver skeleton). A USB device you can match against (for testing, you can use the same VOTI/OBDEV VID/PID as in Lab 2, or any spare USB device whose IDs you can read).

Steps.

1. Start from Lab 2 as a template. Copy the directory to lab07-ucm-skeleton.
2. Modify the softc to include a struct ucom_super_softc and a struct ucom_softc:

```
struct lab07_softc {
    struct ucom_super_softc sc_super_ucm;
    struct ucom_softc      sc_ucm;
    struct usb_device      *sc_udev;
    struct mtx             sc_mtx;
    struct usb_xfer         *sc_xfer[LAB07_N_XFER];
    uint8_t                sc_iface_index;
    uint8_t                sc_flags;
};
```

3. Add a struct ucom_callback with stub implementations:

```
static void lab07_cfg_open(struct ucom_softc *);
static void lab07_cfg_close(struct ucom_softc *);
static int  lab07_pre_param(struct ucom_softc *, struct termios *);
static void lab07_cfg_param(struct ucom_softc *, struct termios *);
static void lab07_cfg_set_dtr(struct ucom_softc *, uint8_t);
static void lab07_cfg_set_rts(struct ucom_softc *, uint8_t);
static void lab07_cfg_set_break(struct ucom_softc *, uint8_t);
static void lab07_start_read(struct ucom_softc *);
```

```

static void lab07_stop_read(struct ucom_softc *);
static void lab07_start_write(struct ucom_softc *);
static void lab07_stop_write(struct ucom_softc *);
static void lab07_free(struct ucom_softc *);

static const struct ucom_callback lab07_callback = {
    .ucm_cfg_open      = &lab07_cfg_open,
    .ucm_cfg_close     = &lab07_cfg_close,
    .ucm_pre_param     = &lab07_pre_param,
    .ucm_cfg_param     = &lab07_cfg_param,
    .ucm_cfg_set_dtr   = &lab07_cfg_set_dtr,
    .ucm_cfg_set_rts   = &lab07_cfg_set_rts,
    .ucm_cfg_set_break = &lab07_cfg_set_break,
    .ucm_start_read    = &lab07_start_read,
    .ucm_stop_read     = &lab07_stop_read,
    .ucm_start_write   = &lab07_start_write,
    .ucm_stop_write    = &lab07_stop_write,
    .ucm_free          = &lab07_free,
};

```

`ucm_pre_param` runs on the caller's context before the configuration task is scheduled; use it to reject unsupported `termios` values by returning a nonzero `errno`. `ucm_cfg_param` runs in the framework's task context and is where you would issue the actual USB control transfer to reprogram the chip.

4. Implement each callback as a no-op for now. Add `device_printf(sc->sc_super_ucm, sc_dev, "%s\n", __func__)` to each so that you can see which callbacks are being invoked.
5. In the attach method, after `usbd_transfer_setup`, call:

```

error = ucom_attach(&sc->sc_super_ucm, &sc->sc_ucm, 1, sc,
    &lab07_callback, &sc->sc_mtx);
if (error != 0) {
    goto fail_xfer;
}

```

6. In the detach method, call `ucm_detach(&sc->sc_super_ucm, &sc->sc_ucm)` before `usbd_transfer_unsetup`.
7. Add `MODULE_DEPEND(lab07, ucom, 1, 1, 1)`; after the existing `MODULE_DEPEND`.
8. Build, load, plug in the device, and observe. In `dmesg`, you should see the driver attach, and you should see a `cuaU0` device appear in `/dev/`.
9. Run `cu -l /dev/cuaU0 -s 9600`. The `cu` command will open the device, which triggers several of the `ucm` callbacks. Watch `dmesg` to see which ones fire. Close `cu` with `~.` and observe more callbacks.
10. Run `stty -a -f /dev/cuaU0`. Observe that the port has default `termios` settings. Run `stty 115200 -f /dev/cuaU0` and observe that `lab07_cfg_param` is called.
11. Unplug the device. Observe clean detach.

Expected output. The driver attaches as a ucom device, creates `/dev/cuaU0`, and responds to configuration ioctls (even though the underlying USB device does not actually do anything). Every callback invocation is visible in `dmesg`.

What to watch for. If the driver attaches but `/dev/cuaU0` does not appear, check that `ucom_attach` succeeded. The return value is an `errno`; a nonzero value means failure. If it failed with `ENOMEM`, you are running out of memory for the TTY allocation. If it failed with `EINVAL`, one of the callback fields is probably null (look at `/usr/src/sys/dev/usb/serial/usb_serial.c` to see which fields are strictly required).

This lab is a building block. A real ucom driver (like `uftdi`) would fill in the callbacks with actual USB transfers to the chip. Starting from an empty skeleton and adding one callback at a time is a good way to build a new driver.

26.10.8 Lab 8: Troubleshooting a Hung TTY Session

This lab is a diagnostic exercise. Given a malfunctioning serial setup, you will use the tools from Section 5 to find the problem.

Goal. Find why a `cu` session does not echo characters back after connecting to an `nmdm(4)` pair that has an unconfigured baud rate on one side.

Steps.

1. Load `nmdm`:

```
# kldload nmdm
```

2. Set different baud rates on the two sides. This is contrived but mimics a real configuration bug:

```
# stty 9600 -f /dev/nmdm0A
# stty 115200 -f /dev/nmdm0B
```

3. Open both sides with `cu`, each with the mismatched rate:

```
(terminal 1) # cu -l /dev/nmdm0A -s 9600
(terminal 2) # cu -l /dev/nmdm0B -s 115200
```

4. Type in terminal 1. You will likely see characters appear in terminal 2, but possibly garbled. Or characters may not appear at all if the `nmdm(4)` driver enforces rate matching strictly.
5. Exit both `cu` sessions.
6. Run `stty -a -f /dev/nmdm0A` and `stty -a -f /dev/nmdm0B`. Find the discrepancy.
7. Fix: set both sides to the same rate. Reopen `cu` and verify that the issue is resolved.

What to watch for. This lab teaches the diagnostic habit of checking both ends of a link. A mismatch at any one end produces problems; finding it requires looking at both. The diagnostic tools (`stty`, `comcontrol`) work from the command line and produce human-readable output. Making use of them is a simple first check before diving into deeper debugging.

26.10.9 Lab 9: Monitoring USB Transfer Statistics

This lab explores the per-channel statistics the USB framework maintains, which can help identify performance issues or hidden errors.

Goal. Use `usbconfig dump_stats` to observe the transfer counts on a busy USB device and identify whether the device is performing as expected.

Steps.

1. Plug in a USB device that you can exercise meaningfully. A USB flash drive is a good choice because you can trigger bulk transfers by copying files.

2. Identify the device:

```
# usbconfig list
```

Note the `ugenN.M` identifier.

3. Dump the baseline statistics:

```
# usbconfig -d ugenN.M dump_stats
```

Record the output.

4. Perform significant I/O to the device. For a flash drive, copy a large file:

```
# cp /usr/src/sys/dev/usb/usb_transfer.c /mnt/usb_mount/
```

5. Dump the statistics again. Compare.

6. Note which counters changed. `xfer_completed` should have increased significantly. `xfer_err` should still be small.

7. Try to deliberately cause errors. Unplug the device mid-transfer. Then plug it back in. Dump the stats on the new `ugenN.M` (a new one is allocated on replug).

What to watch for. The statistics reveal invisible behaviours. A device that is mostly working but occasionally stalling will show `stall_count` nonzero. A device that is dropping transfers will show `xfer_err` climbing. In normal operation, a healthy device shows steady `xfer_completed` growth and zero errors.

If you are developing a driver and the statistics show unexpected errors, that is a clue that something is wrong. The statistics are maintained by the USB framework, not the driver, so they reflect reality regardless of whether the driver notices.

26.11 Challenge Exercises

Challenge exercises stretch your understanding. They are not strictly necessary for progressing to Chapter 27, but each one will deepen your grasp of USB and serial driver work. Take your time. Read relevant FreeBSD source. Write small programs. Expect some challenges to take several hours.

26.11.1 Challenge 1: Add a Third USB Endpoint Type

The skeleton in Section 2 supports bulk transfers. Extend it to also handle an interrupt endpoint. Add a new channel to the `struct usb_config` array with `.type = UE_INTERRUPT`, `.direction = UE_DIR_IN`, and a small buffer (say, sixteen bytes). Implement the callback as a continuous poll, reading a small status packet from the device on every interrupt-IN completion.

Test the change by comparing the behaviour of the three channels. Bulk channels should be quiet most of the time and only submit transfers when the driver has work to do. The interrupt channel should run continuously, quietly consuming interrupt-IN packets whenever the device sends them.

A stretch goal: make the interrupt callback deliver received bytes to the same `/dev` node as the bulk channel. When userspace reads the node, it gets a merged view of bulk-in and interrupt-in data. This is a useful pattern for devices that have both streaming data and asynchronous status events.

26.11.2 Challenge 2: Write a Minimal USB Gadget Driver

The running example is a host-side driver: the FreeBSD machine is the USB host, and the device is the peripheral. Turn the example around by writing a USB gadget driver that makes the FreeBSD machine present itself as a simple device to another host.

This requires USB-on-the-Go hardware, so the challenge is only feasible on specific boards (some ARM development boards support it). The relevant source is in `/usr/src/sys/dev/usb/template/`. Start from `usb_template_cdce.c`, which implements the CDC Ethernet class, and modify it to implement a simpler vendor-specific class with one bulk-OUT endpoint that just swallows whatever the host sends.

This challenge teaches you how the USB framework looks from the other side. Many of the concepts are mirror-imaged: what was a transfer from the host's perspective is a transfer from the device's perspective, but the direction of the bulk arrow is reversed.

26.11.3 Challenge 3: A Custom `termios` Flag Handler

The `termios` structure has many flags, and the `uart(4)` framework handles most of them automatically. Write a small modification to a UART driver (or to a copy of `uart_dev_ns8250.c` in a private build) that makes the driver log a `device_printf` message every time a specific `termios` flag changes value.

Pick, say, `CRTSCTS` (hardware flow control) as the flag to track. Add a log message in the driver's `param` path that prints “`CRTSCTS=on`” or “`CRTSCTS=off`” whenever the flag's new value differs from its old value.

Test the modification by running:

```
# stty crtscts -f /dev/cuau0
# stty -crtscts -f /dev/cuau0
```

Verify that the log messages appear in `dmesg` and that they correspond correctly to the `stty` changes.

This challenge is about understanding exactly where in the call chain the `termios` change arrives at the driver. The answer (in `param`) is documented in the source, but seeing it with your own eyes is different from reading about it.

26.11.4 Challenge 4: Parsing a Small USB Protocol

Pick a USB protocol you are curious about. HID is a good candidate because it is widely documented. CDC ACM is another good choice because it is simple. Pick one, read the specification on usb.org (the public parts), and write a small protocol parser in C that takes a buffer of bytes and prints what they mean.

For HID, the parser would consume reports: input reports, output reports, feature reports. It would look up the device's report descriptor to learn the layout. It would print, for each report, the usage (mouse motion, button press, keyboard scan code) and the value.

For CDC ACM, the parser would consume the AT command set: a small set of commands that terminal programs use to configure modems. It would recognise the commands and report which ones the driver would handle and which would be passed through to the device.

This is not a driver-writing challenge per se; it is a protocol-understanding challenge. Device drivers implement protocols, and being comfortable with protocol specifications is a core skill.

26.11.5 Challenge 5: Robustness Under Load

Take the echo-loop driver from Lab 3 (or a similar driver you have written) and stress-test it. Write a userland program that runs two threads: one constantly writes random bytes to the device, one constantly reads and verifies.

Run the program for an hour. Then run it for a day. Then unplug and replug the device during the run and see whether the program recovers cleanly.

You will probably find bugs. Common ones include: write callback locking issues under concurrent access, races between `close()` and in-flight transfers, memory leaks from buffers that are allocated but never freed on specific error paths, and state machine bugs when a stall arrives at an unexpected moment.

Each bug you find will teach you something about where the driver's contract with its callers is subtle. Fix the bugs. Log what you learned. This is exactly the kind of work that separates a good driver from a merely working one.

26.11.6 Challenge 6: Implement Suspend/Resume Properly

Most USB drivers do not implement suspend and resume handlers. The framework has defaults that work for the common case, but a driver that holds long-term state (a queue of pending commands, a streaming context, a negotiated session) may need to save and restore that state around suspend cycles.

Extend the echo-loop driver with `device_suspend` and `device_resume` methods. In suspend, flush any pending transfers and save a small amount of state. In resume, restore the state and resubmit any pending work.

Test by running the system through a suspend cycle (on a laptop that supports it) while the driver is running. Verify that after resume, the driver continues working correctly and no state was lost.

This challenge teaches the subtleties of suspend/resume, including that hardware may be in a different state after resume than it was before suspend, and that all in-flight state must be reconstructed or abandoned.

26.11.7 Challenge 7: Adding poll(2) Support

Most drivers shown in this chapter support `read(2)` and `write(2)` but not `poll(2)` or `select(2)`. These system calls let userland programs wait for I/O readiness on multiple descriptors at once, which is essential for servers and interactive programs.

Add a `d_poll` method to the echo driver's `cdevsw`. The method should return a bitmask indicating which I/O events are currently possible: `POLLIN` if there is data to read, `POLLOUT` if there is space to write.

The hardest part of adding poll support is the wakeup logic. When a transfer callback adds data to the RX queue, it must call `selwakeup` on the `selinfo` structure the poll mechanism uses. Similarly, when the write callback drains bytes from the TX queue and makes space, it must call `selwakeup` on the write `selinfo`.

This challenge will require reading `/usr/src/sys/kern/sys_generic.c` and `/usr/src/sys/sys/selinfo.h` to understand the `selinfo` mechanism.

26.11.8 Challenge 8: Writing a Character-Counter ioctl

Add an `ioctl` to the echo driver that returns the current TX and RX byte counters. The `ioctl` interface requires you to:

1. Define a magic number and struct for the `ioctl` in a header:

```
struct myfirst_usb_stats {
    uint64_t tx_bytes;
    uint64_t rx_bytes;
};
#define MYFIRST_USB_GET_STATS _IOR('U', 1, struct myfirst_usb_stats)
```

2. Implement a `d_ioctl` method that responds to `MYFIRST_USB_GET_STATS` by copying the counters out to userland.
3. Maintain the counters in the `softc`, incrementing them in the transfer callbacks.
4. Write a userland program that issues the `ioctl` and prints the results.

This challenge teaches the `ioctl` interface, which is the standard way drivers expose non-streaming operations to userland. It also introduces you to the `_IOR`, `_IOW`, and `_IOWR` macros from `<sys/ioccom.h>`.

26.12 Troubleshooting Guide

Despite your best efforts, problems will occur. This section documents the most common classes of problem you will hit while working on USB and serial drivers, with concrete steps to diagnose each.

26.12.1 The Module Will Not Load

Symptom: `kldload myfirst_usb.ko` returns an error, typically with a message about unresolved symbols.

Causes and fixes: - Missing `MODULE_DEPEND` entry. Add `MODULE_DEPEND(myfirst_usb, usb, 1, 1, 1);` to the driver. - Missing `MODULE_DEPEND` on a second module, such as `ucom`. If your driver uses `ucom_attach`, add a dependency on `ucom`. - Compiled against a kernel that does not match the running kernel. Rebuild the module against the currently-running sources. - Kernel symbol table out of date. After kernel upgrade, run `kldxref /boot/kernel` to refresh.

If the error message mentions a specific symbol you did not write (like `ttymcreate` or `cdevsw_open`), look up the missing symbol in the source tree to find out which subsystem it lives in, and add a `MODULE_DEPEND` on that module.

26.12.2 The Driver Loads but Never Attaches

Symptom: `kldstat` shows the driver loaded, but `dmesg` shows no attach message when the device is plugged in.

Causes and fixes: - Match table does not match the device. Compare the vendor and product IDs from `usbconfig list` against your `STRUCT_USB_HOST_ID` entries. - Interface number mismatch. If the device has multiple interfaces and your probe guards against `bInterfaceIndex != 0`, try a different interface. - Probe returns `ENXIO` for some other reason. Add `device_printf(dev, "probe with class=%x subclass=%x\n", uaa->info.bInterfaceClass, uaa->info.bInterfaceSubClass);` at the top of probe temporarily to see what the framework is offering. - Another driver is claiming the device first. Check `dmesg` for other driver attachments; you may need to explicitly unload the competing driver with `kldunload` before yours can bind. Alternatively, give your driver a higher priority through bus-probe return values (applies to PCI-like buses, not USB).

26.12.3 The Driver Attaches but /dev Node Does Not Appear

Symptom: attach message in `dmesg`, but `ls /dev/` shows no corresponding entry.

Causes and fixes: - `make_dev` call failed. Check the return value; if null, handle the error and log it. - Wrong `cdevsw`. Make sure `myfirst_usb_cdevsw` is declared correctly with `d_version = D_VERSION` and valid `d_name`, `d_open`, `d_close`, `d_read`, `d_write`, `d_ioctl` where relevant. - `si_drv1` not set. Although not strictly required for the node to appear, many bugs manifest as “the node appears but `ioctl`s see a NULL softc” because `si_drv1` was not initialised. - Permissions issue. The default 0644 permissions may restrict access; try 0666 temporarily during development.

26.12.4 The Driver Panics on Detach

Symptom: unplugging the device (or unloading the module) causes a kernel panic.

Causes and fixes: - Transfer callback running during detach. You must call `usbd_transfer_unsetup` before destroying the mutex. The framework’s cancellation and wait logic is what makes detach safe. - `/dev` node open when driver unloads. If userspace has the node open, the module cannot unload. Run `fstat | grep myfirst_usb` to see which process holds it, and kill the process or close the file. - Memory freed before all uses complete. If you use deferred work (taskqueue, callout), you must cancel and wait for it before freeing the softc. The `taskqueue_drain` and `callout_drain` functions exist for this. - Softc use-after-free. If you have

code outside the driver that holds a pointer to the `softc`, the `softc` can be freed while that pointer is still dangling. Redesign to avoid external `softc` pointers, or add reference counting.

26.12.5 Transfers Stall

Symptom: bulk transfers appear to succeed at the `submit` call but never complete, or they complete with `USB_ERR_STALLED`.

Causes and fixes: - Wrong endpoint direction. Verify the direction in your `struct usb_config` against the endpoint's `bEndpointAddress` high bit. - Wrong endpoint type. Verify that the `type` field matches the endpoint's `bmAttributes` low bits. - Transfer too large. If you set a frame length larger than the endpoint's `wMaxPacketSize`, the framework will usually slice it into packets, but some devices reject a transfer that exceeds an internal buffer. - Device firmware stall. The remote device is signalling “not ready.” The framework's automatic clear-stall should recover, but a persistent stall usually indicates a protocol error (wrong command, wrong sequence, missing authentication).

26.12.6 Serial Characters Garbled

Symptom: bytes appear on the wire but are wrong or contain extra characters.

Causes and fixes: - Baud rate mismatch. Every stage must agree. Use `stty` to check all stages. - Byte format mismatch. Set databits, parity, and stopbits to match. `stty cs8 -parenb -cstopb` is the most common configuration. - Incorrect `termios` flag handling in the driver. If you modify `uart_dev_ns8250.c` and `breakparam`, the chip will be programmed wrong. Compare against the upstream file. - Flow-control mismatch. If one side has `CRTSCTS` enabled and the other does not, bytes will be lost under load. Set both sides consistently. - Cable issue. A bad cable or a cable with unusual pinouts (some RJ45-to-DB9 cables have nonstandard pinouts) can introduce bit errors. Swap cables to rule this out.

26.12.7 A Process is Stuck in `read(2)` and Will Not Exit

Symptom: a program blocked on the driver's `read()` path will not respond to `Ctrl+C` or `kill`.

Causes and fixes: - Driver `d_read` sleeps without checking for signals. Use `msleep(..., PCATCH, ...)` (with the `PCATCH` flag) so the sleep returns `EINTR` when a signal arrives, and propagate the `errno` back to userspace. - Driver `d_read` holds a non-interruptible lock. Verify that the sleep is on an interruptible condition variable and that the mutex is dropped during the sleep. - Transfer callback is never arming the channel. If your `d_read` waits on a flag that only the read callback sets, and the read callback is never fired, the wait will never complete. Make sure the channel is started on `d_open` or at attach time.

26.12.8 High CPU Usage When Idle

Symptom: the driver consumes significant CPU even when no data is flowing.

Causes and fixes: - Polling-based implementation. If your driver polls a flag in a busy loop, rewrite it to sleep on an event. - Callback firing excessively. The framework should not fire a callback without a state change, but some misconfigured channels can enter a “retry on error” loop that fires the callback as fast as the hardware can

respond. Add a retry counter or a rate-limiter. - Read callback with no work but always rearming. If the device sends zero-byte transfers to signal “I have nothing to say,” make sure your callback handles these gracefully without treating them as normal data.

26.12.9 `usbconfig` Shows the Device but `dmesg` Is Silent

Symptom: `usbconfig list` shows the device, but no driver attach message appears.

Causes and fixes: - Device attached to `ugen` (the generic driver) because no specific driver matched. This is the normal behaviour when there is no matching driver. Check the match tables of the available drivers. `pciconf -lv` will not help here because this is USB, not PCI; the USB equivalent is `usbconfig -d ugenN.M`. `dump_device_desc`. - `devd` is disabled and auto-load is not happening. Enable `devd` by running `service devd onestart`, then plug the device in again. - Module file is not in a loadable path. `kldload` can take a full path (`kldload /path/to/module.ko`), but for automatic loading by `devd`, the module has to be in a directory `devd` is configured to search. `/boot/modules/` is the conventional location for out-of-tree modules on a production system.

26.12.10 Debugging a Deadlock with `WITNESS`

Symptom: the kernel hangs with the CPU stuck in a specific function, and `WITNESS` is enabled.

Causes and fixes: - Lock order violation. `WITNESS` will log the violation on the serial console. Read the log: it will tell you which locks were taken in which order, and where the reverse order was observed. Fix by establishing a consistent lock acquisition order throughout your driver. - Lock held across a sleep. If you hold a mutex and then call a function that sleeps, you can deadlock with any other thread that wants the mutex. Identify the sleeping function (often hidden in an allocation or in a USB transfer wait), and restructure to release the mutex before the sleep. - Lock taken in interrupt context that was first taken outside interrupt context without `MTX_SPIN`. FreeBSD mutexes have two forms: default (`MTX_DEF`) can sleep, spin (`MTX_SPIN`) cannot. Taking a sleep mutex from an interrupt handler is a bug.

Enabling `WITNESS` during development (by building the kernel with options `WITNESS` or by using `GENERIC-NODEBUG`'s `INVARIANTS`-enabled counterpart) catches many of these problems before they appear on a user's machine.

26.12.11 A Driver That Appears Twice for the Same Device

Symptom: `dmesg` shows your driver attaching twice for a single device, creating `myfirst_usb0` and `myfirst_usb1` with the same USB IDs.

Causes and fixes: - The device has two interfaces and the driver is matching both. Check `bIfaceIndex` in the probe method and match only the interface you actually support. - The device has multiple configurations and both are active. This is rare; if so, select the correct configuration explicitly in the attach method. - Another driver is attached to one of the interfaces. This is not a bug; it just means the device is multi-interface and different drivers claim different interfaces. If you see `myfirst_usb0` and `ukbd0` for the same device, the device has both a vendor-specific interface and a HID interface, and the two drivers attach independently.

26.12.12 USB Serial Baud Rate Does Not Take Effect

Symptom: You `stty 115200 -f /dev/cuaU0`, but data exchange happens at a different rate.

Causes and fixes: - The control transfer to program the baud rate failed. Check `dmesg` for error messages from `ucom_cfg_param`. Instrument the driver to log the result of the control transfer. - The chip's divisor encoding is wrong. Different FTDI variants use slightly different divisor formulas; check the variant detection in the driver. - The peer is running at a different rate. As noted earlier in the chapter, both ends must agree. - The cable or adapter is introducing its own rate limitation. Some USB-to-serial adapters silently renegotiate; this is rare but can happen with poor-quality cables.

26.12.13 A Kernel Panic with “Spin lock held too long”

Symptom: The kernel panics with this message, usually during high I/O on the driver.

Causes and fixes: - A UART driver's `uart_ops` method is sleeping or blocking. The six methods in `uart_ops` run with spin locks held (on some paths) and must not sleep, call non-spin-safe functions, or do long loops. Review the offending method for any expensive calls. - The interrupt handler is not draining the interrupt source fast enough. If the handler takes longer than the interrupt rate, interrupts accumulate. Speed up the handler. - Lock contention is causing priority inversion. Reduce the scope of the critical section, or break it up.

26.12.14 A Device Never Completes Enumeration

Symptom: Plugging in a device produces a `dmesg` line or two about enumeration starting, but never a completion message.

Causes and fixes: - The device is violating the USB specification. Some cheap or counterfeit devices have buggy firmware. If possible, try a different device. - Insufficient power. Devices that claim more power than the port can supply will fail to enumerate. Try a powered hub. - Electromagnetic interference. A bad cable or a bad port can cause bit errors during enumeration. Try different cables or ports. - The USB host controller is in a confused state. Try unloading and reloading the host controller driver, or (as a last resort) rebooting.

26.12.15 Diagnostic Checklist When You Are Stuck

When a driver under development is not behaving correctly and you do not know why, walk through this checklist in order. Each step eliminates a large class of possible problems.

1. Compile cleanly with `-Wall -Werror`. Many subtle bugs produce warnings.
2. Load in a kernel built with `INVARIANTS` and `WITNESS`. Any locking or invariant violations will be caught immediately.
3. Enable your driver's debug logging. Run a minimal reproduction scenario and capture the logs.
4. Compare the driver's behaviour against a known-working driver for similar hardware. Diffing behaviour reveals bugs that staring at your own code does not.
5. Simplify the scenario. Write a minimal userland test program. Use a minimal USB device (or an `nmdm` pair for serial). Remove every variable you can.

6. Use `dtrace` on the USB framework functions. `usbd_transfer_submit:entry` and `usbd_transfer_submit:return` probes let you trace exactly which transfers were submitted and what happened to them.
7. Run the driver with `WITNESS_CHECKORDER` enabled. Each time a mutex is taken, the order is verified against the accumulated history.
8. If the issue is intermittent, run under a stress-test harness that generates load for hours. Intermittent bugs become reproducible under sustained load.

This checklist is not exhaustive, but it covers the techniques that find the majority of driver bugs.

26.13 Reading the FreeBSD USB Source Tree: A Guided Tour

The `myfirst_usb` skeleton and the FTDI walkthrough have given you the shape of a USB driver. But the real learning happens when you read existing drivers in the tree. Each one is a small lesson in how to apply the framework to a specific class of device. This section gives you a guided tour of five drivers, ordered from simplest to most representative, and points out what each one teaches.

The pattern we recommend is this. Open each driver's source file next to this section. Read the opening comment block and the structure definitions first; those tell you what the driver is for and what state it maintains. Then trace the lifecycle: match table, probe, attach, detach, registration. Only after the lifecycle is clear should you move on to the data path. This ordering mirrors how the framework itself treats the driver: first as a match candidate, then as an attached driver, and only then as something that moves data.

26.13.1 Tour 1: `uled.c`, the Simplest USB Driver

File: `/usr/src/sys/dev/usb/misc/uled.c`.

Start here. `uled.c` is the Dream Cheeky USB LED driver. It is under 400 lines. It implements a single output (setting the LED colour) through a single control transfer. There is no input, no bulk transfer, no interrupt transfer, no concurrent I/O. Everything about it is minimal, and for that reason everything about it is easy to read.

Key things to study in `uled.c`:

The match table has a single entry: `{USB_VPI(USB_VENDOR_DREAM_CHEEKY, USB_PRODUCT_DREAM_CHEEKY_WEBMAIL_NOTIFICATION), 0}`. This is the minimal match-by-VID/PID idiom. No subclass or protocol filtering; just vendor and product.

The `softc` is tiny. It contains a mutex, the `usb_device` pointer, the `usb_xfer` array, and the LED state. This is the minimum every USB driver needs.

The probe method is two lines: check that the device is in host mode and return the result of `usbd_lookup_id_by_uaa` against the match table. No interface-index check, no complex matching. For a simple device with a single function, this is enough.

The attach method allocates the transfer channel, creates a device-file entry with `make_dev`, and stores the pointers. No complex negotiation; the device is ready after attach returns.

The I/O path is a single control transfer with a fixed setup. The driver sets the frame length, fills in the color bytes with `usbd_copy_in`, and calls `usbd_transfer_submit`. That is it.

Read `uled.c` first. When you have read it once, the rest of the USB subsystem opens up. Every more complex driver is a variation on this pattern.

26.13.2 Tour 2: `ugold.c`, Adding Interrupt Transfers

File: `/usr/src/sys/dev/usb/misc/ugold.c`.

`ugold.c` drives a USB thermometer. It is still very short, under 500 lines, but it introduces interrupt transfers, which are the staple of HID-class devices.

Key things to learn from `ugold.c`:

The device publishes temperature readings periodically via an interrupt endpoint. The driver's job is to listen on that endpoint and deliver the readings to userland via `sysctl`.

The `usb_config` array now has an entry for `UE_INTERRUPT`, with `UE_DIR_IN`. This tells the framework to set up a channel that polls the interrupt endpoint.

The interrupt callback shows the canonical pattern: on `USB_ST_TRANSFERRED`, extract the received bytes with `usbd_copy_out`, parse them, update the `softc`. On `USB_ST_SETUP` (including the initial callback after start), set the frame length and submit. On `USB_ST_ERROR`, decide whether to recover or give up.

The driver exposes readings through `sysctl` nodes created in `attach` and torn down in `detach`. This is a common pattern for devices that produce occasional readings: the interrupt callback writes to `softc` state, and userland reads from `sysctl` when it wants a value.

Compare `ugold.c` to `uled.c` after reading both. The control-transfer-only driver and the interrupt-transfer driver represent the two most common skeleton patterns. Most other USB drivers are composed of variations of these two.

26.13.3 Tour 3: `udbp.c`, Bidirectional Bulk Transfers

File: `/usr/src/sys/dev/usb/misc/udbp.c`.

`udbp.c` is the USB Double Bulk Pipe driver. It exists to test bidirectional bulk data flow between two computers connected by a special USB-to-USB cable. It is about 700 lines and gives you a complete working example of bulk read and bulk write.

Key things to learn from `udbp.c`:

The `usb_config` has two entries: one for `UE_BULK UE_DIR_OUT` (host-to-device) and one for `UE_BULK UE_DIR_IN` (device-to-host). This is the standard bulk-duplex pattern.

Each callback does the same three-state dance. On `USB_ST_SETUP`, set the frame length (or if it is a read, just submit). On `USB_ST_TRANSFERRED`, consume the completed data and re-arm. On `USB_ST_ERROR`, decide the recovery policy.

The driver uses the netgraph framework to integrate with higher layers. This is a choice specific to the Double Bulk Pipe device. For a simple application, you would expose the bulk channels through a character device, as `myfirst_usb` does.

Trace how the softc maintains the state of each direction independently. The receive callback rearms only when a buffer is available. The transmit callback rearms only when there is something to send. The two callbacks coordinate only through shared softc fields (counter of pending operations, queue pointers).

26.13.4 Tour 4: `uplcom.c`, a USB-to-Serial Bridge

File: `/usr/src/sys/dev/usb/serial/uplcom.c`.

`uplcom.c` drives the Prolific PL2303, one of the most common USB-to-serial chips. At around 1400 lines, it is more substantial than the previous three, but every part of it maps directly onto the serial-driver pattern from Section 4 of this chapter.

Key things to learn from `uplcom.c`:

The `ucom_callback` structure fills in every configuration method you would expect a real driver to implement: `ucom_cfg_open`, `ucom_cfg_param`, `ucom_cfg_set_dtr`, `ucom_cfg_set_rts`, `ucom_cfg_set_break`, `ucom_cfg_get_status`, `ucom_cfg_close`. Each of these calls the framework-provided `ucom` primitives after issuing the chip-specific USB control transfer.

Look at `uplcom_cfg_param`. It takes a `termios` structure, extracts the baud rate and framing, and constructs a vendor-specific control transfer to program the chip. This is how a user's `stty 9600` call propagates through the layers: `stty` updates `termios`, the TTY layer calls `ucom_param`, the framework schedules the control transfer, and `uplcom_cfg_param` programs the chip.

Compare `uplcom_cfg_param` with the corresponding function in `uftdi.c`. Both translate a `termios` to a vendor-specific control sequence, but the vendor protocols are entirely different. This illustrates why the framework insists on per-vendor drivers: each chip has its own command set, and the framework's job is only to give each driver a uniform way to be called.

Note how the driver handles reset, modem signals, and break. Each modem-line operation is a separate USB control transfer. The cost of changing, say, DTR is one round-trip to the device, which on a 12 Mbps bus takes about 1 ms. This tells you why line signals change more slowly over USB-to-serial than over a native UART, and why protocols that toggle DTR frequently can behave differently through a USB-to-serial adapter.

26.13.5 Tour 5: `uhid.c`, the Human Interface Device Driver

File: `/usr/src/sys/dev/usb/input/uhid.c`.

`uhid.c` is the generic HID driver. HID stands for Human Interface Device; it covers keyboards, mice, gamepads, touchscreens, and countless vendor-specific devices that conform to the HID class standard. `uhid.c` is roughly 1000 lines.

Key things to learn from `uhid.c`:

The match table uses class-based matching. Instead of listing every VID/PID, the driver matches any device that advertises the HID interface class. `UIFACE_CLASS(UICLASS_HID)` tells the framework to match any HID interface, no matter which vendor made the device.

The driver exposes the device through a character device, not through `ucom` or a networking framework. The character device pattern lets userland programs open `/dev/uhidN` and issue `ioctl` calls to read HID descriptors, read reports, and set feature reports.

The interrupt endpoint delivers HID reports, and the driver hands them up to userland through a ring buffer and read. This is the USB equivalent of a character-device interrupt-driven read loop.

Study how `uhid.c` uses the HID report descriptor to understand what the device is. The descriptor is parsed at attach time, and the driver populates its internal tables from the parse. Every HID device describes itself this way; the driver does not hard-code device semantics.

26.13.6 How to Study a Driver You Have Never Seen

Beyond the tour, you will encounter drivers in the tree that you have never seen. A general-purpose reading strategy helps:

Open the source file and scroll to the bottom. The registration macros are there. They tell you what the driver attaches to (`uhub`, `usb`) and its name (`udbp`, `uhid`). Already you know where the driver fits in the tree.

Scroll back up to the `usb_config` array (or the transfer declarations for non-USB drivers). Each entry is one channel. Count them; look at their types and directions. You now know the shape of the data path.

Look at the probe method. If it matches by VID/PID, the device is vendor-specific. If it matches by class, the driver supports a family of devices. This tells you the scope of the driver.

Look at the attach method. Follow its labelled-goto chain. The labels give you the order of resource allocation: mutex, channels, character device, sysctls, and so on.

Finally, look at the data-path callbacks. Each one is a three-state state machine. Read `USB_ST_TRANSFERRED` first; that is where the actual work happens. Then read `USB_ST_SETUP`; that is the kickoff. Then read `USB_ST_ERROR`; that is the recovery policy.

With this reading order, you can make sense of any USB driver in the tree in about 15 minutes. With practice, you will start to recognise patterns across drivers and know which ones are idiomatic (the ones to copy) and which ones are historical oddities (the ones to understand but not copy).

26.13.7 Where to Go Beyond the Tour

The `/usr/src/sys/dev/usb/` tree has four subdirectories that are worth exploring:

`/usr/src/sys/dev/usb/misc/` contains simple, single-purpose drivers: `uled`, `ugold`, `udbp`. If you are writing a new device-specific driver that does not fit an existing class, read the drivers here to see how small drivers are structured.

`/usr/src/sys/dev/usb/serial/` contains the USB-to-serial bridge drivers: `uftdi`, `uplcom`, `uslcom`, `u3g` (3G modems, which present as serial to userland), `uark`, `uipaq`, `uchcom`. If you are writing a new USB-to-serial driver, start here.

`/usr/src/sys/dev/usb/input/` contains keyboard, mouse, and HID drivers. `ukbd`, `ums`, `uhid`. If you are writing a new input driver, these are the patterns to follow.

`/usr/src/sys/dev/usb/net/` contains USB network drivers: `axge`, `axe`, `cdce`, `ure`, `smc`. These are the drivers that bridge Chapter 26 to Chapter 27, because they combine the USB framework of this chapter with the `ifnet(9)` framework of the next. Reading one of them after finishing Chapter 27 is a productive exercise.

The `/usr/src/sys/dev/uart/` tree has fewer files but each is worth reading:

`/usr/src/sys/dev/uart/uart_core.c` is the framework core. Read this to understand what happens above your driver: how bytes flow in and out, how the TTY layer connects, how interrupts are dispatched.

`/usr/src/sys/dev/uart/uart_dev_ns8250.c` is the canonical reference driver. Read this after the framework core so you can see how a driver plugs in.

`/usr/src/sys/dev/uart/uart_bus_pci.c` shows the PCI bus-attach glue for UARTs. If you ever need to write a UART driver that attaches to PCI, this is your starting point.

Each of these files is small enough to read in one sitting. Reading the source is not homework; it is how you learn a subsystem. Chapter 26 has given you the vocabulary and the mental model; the source is where you apply them.

26.14 Performance Considerations for Transport Drivers

Most of Chapter 26 has focused on correctness: getting a driver to attach, do its work, and detach cleanly. Correctness always comes first. But once your driver works, you will often want to know how fast it is, and whether its performance matches what the transport can sustain. This section gives you a practical frame for thinking about USB and UART performance without turning the chapter into a benchmarking manual.

26.14.1 The USB Bus as a Shared Resource

Every device on a USB bus shares the bus with every other device. The bandwidth is not divided fairly; it is allocated according to USB's scheduling rules. Control and interrupt endpoints get guaranteed periodic service. Bulk endpoints get what is left over, in a fair-share sense. Isochronous endpoints reserve bandwidth up front; if there is not enough, the allocation fails.

For a bulk-transferring driver, the practical upshot is this. Your effective bandwidth is the theoretical link speed (12, 480, 5000 Mbps) minus the overhead of other devices' periodic traffic, minus the USB protocol overhead (roughly 10% on full-speed, less on higher speeds), minus the overhead of short transfers.

The last item is the one you can influence. A transfer of 16 KB is not 16 times more expensive than a transfer of 1 KB; the overhead of initiating and completing a transfer is fixed, and the data-transfer portion is close to linear in size. For high-throughput bulk transfers, use large buffers. The hardware is designed for this; the framework is designed for this; your driver should be designed for this.

For an interrupt-transferring driver, the constraint is different. The interrupt endpoint polls at a fixed interval (configured by the device). The framework delivers a callback whenever the polled transfer completes. The maximum report rate is the endpoint's polling rate. If the device has a 1-ms interval, you get at most 1000 reports per second. Planning for interrupt-driven performance means planning around the polling rate.

26.14.2 Latency: What Costs Microseconds, What Costs Milliseconds

USB is not a low-latency bus. A single control transfer on full-speed USB takes roughly 1 ms round-trip. A single bulk transfer takes roughly 1 ms of framing overhead plus the time to move the data. Interrupt transfers are scheduled at the polling interval, so the minimum latency is the interval itself.

Compare this to native UART, where a character transmission takes roughly 1 ms at 9600 baud, 100 us at 115200 baud, and 10 us at 1 Mbps. A native UART driver can push out a byte in hundreds of microseconds if it is well designed; a USB-to-serial bridge cannot match that, because each byte has to traverse USB first.

For your driver, this means: think about where latency matters for your use case. If you are building a monitoring driver that reports once a second, USB is fine. If you are building an interactive controller where the user can feel each character round-trip, native UART is much better. If you are building a real-time control loop where characters must traverse in tens of microseconds, neither USB nor general-purpose UART is appropriate; you need a dedicated bus with known timing.

26.14.3 When to Rearm: The Classic USB Tradeoff

A key decision in any streaming USB driver is where in the callback to re-arm the transfer. There are two viable patterns:

Rearm after work. In `USB_ST_TRANSFERRED`, do the work (parse the data, hand it up, update state), then rearm. Simple to implement. Has a latency cost: the time between the previous completion and the next submission is the time it took to do the work.

Rearm before work, using multiple buffers. In `USB_ST_TRANSFERRED`, immediately rearm with a fresh buffer, then do the work on the just-completed buffer. This requires multiple frames in the `usb_config` (so the frame-work rotates through a pool of buffers) or two parallel transfer channels. Has near-zero latency between transfers because the hardware always has a buffer ready.

Most drivers in the tree use the first pattern because it is simpler. The second pattern is used in high-throughput drivers where hiding the work latency matters. `ugold.c` is the first pattern; some of the USB Ethernet drivers in `/usr/src/sys/dev/usb/net/` are the second.

26.14.4 Buffer Sizing

For bulk transfers, the buffer size is a knob. Larger buffers amortise the per-transfer overhead, but they also delay the delivery of partial data and increase memory usage. The typical values in the tree are between 1 KB and 64 KB.

For interrupt transfers, the buffer size is usually small (8 to 64 bytes) because the endpoint itself limits the report size. Do not make this larger than the endpoint's `wMaxPacketSize`; the extra buffer is wasted.

For control transfers, the buffer size is determined by the protocol of the specific operation. The `usb_device_request` header is always 8 bytes; the data portion depends on the request.

26.14.5 UART Performance

For a UART driver, performance is usually a question of interrupt efficiency. A 16550A with a FIFO depth of 16 bytes at 115200 baud needs to be serviced roughly every 1.4 ms in the worst case. If your interrupt handler takes longer than that, the FIFO overflows and data is lost. Modern UARTs (16750, 16950, ns16550 variants on embedded SoCs) often have deeper FIFOs (64, 128, or 256 bytes) specifically to relax this constraint.

The `uart(4)` framework handles the FIFO management for you through `uart_ops->rxready` and the ring buffer. What you control as the driver author is: how fast your implementation of `getc` is, how fast `putc` is, and whether your interrupt handler is sharing the CPU with other work.

For higher baud rates (921600, 1.5M, 3M), a raw 16550A is not enough. These rates require either a chip with a larger FIFO or a driver that uses DMA to move characters directly to memory. The `uart(4)` framework supports DMA-backed drivers, but the vast majority of drivers (including `ns8250`) do not use it. DMA support is usually reserved for embedded platforms that specifically provide it.

26.14.6 Concurrency and Lock Hold Times

A USB callback runs with the driver's mutex held. If the callback takes a long time (copying a large buffer, doing complex processing), no other callback can run, and no detach can complete. Keep callback work short.

The idiomatic pattern for non-trivial work is: in the callback, copy the data out of the framework buffer into a private buffer in `softc`, then mark the data as ready and wake a consumer. The consumer (userland via `read`, or a worker taskqueue) does the heavy processing without the driver mutex.

For a UART driver, the same principle applies. The `rxready` and `getc` methods must be fast because they run in interrupt context. Heavy processing is done later, outside the interrupt, by the TTY layer and user processes.

26.14.7 Measuring, Not Guessing

The best way to answer a performance question is to measure. The `dtrace` hooks on `usb_transfer_submit` and related functions let you time transfers to microsecond precision. `sysctl -a | grep usb` exposes per-device statistics. For UARTs, `sysctl -a dev.uart` and the TTY statistics in `vmstat` tell you where time is going.

Do not optimise a driver blindly. Run the workload, measure, find the bottleneck, and fix what actually matters. For most drivers, the bottleneck is not the transfer itself but something surrounding it: memory allocation, locking, or a poorly sized buffer.

26.15 Common Mistakes When Writing Your First Transport Driver

The patterns in this chapter are the right way to write a transport driver. But patterns are easier to describe than to apply. Most first-time drivers are written with every pattern followed correctly in principle but misapplied in practice. This section lists the specific mistakes that appear most often when someone sits down to write a USB or UART driver for the first time. Each mistake is paired with the correction and a short explanation of why the correction is necessary.

Read this section once before you write your first driver, and again when you are debugging one. The mistakes are surprisingly universal; almost every experienced FreeBSD driver author has made several of them at some point.

26.15.1 Mistake 1: Taking the Framework Mutex Explicitly in a Callback

The mistake looks like this:

```
static void
my_bulk_read_callback(struct usb_xfer *xfer, usb_error_t error)
{
    struct my_softc *sc = usbd_xfer_softc(xfer);

    mtx_lock(&sc->sc_mtx);    /* <-- wrong */
    /* ... do work ... */
    mtx_unlock(&sc->sc_mtx);
}
```

The framework has already acquired the mutex before calling the callback. Taking it a second time is a self-deadlock on most mutex implementations and an extra uncontested acquisition on others. On some kernel configurations, it will panic immediately with a “recursive lock” assertion from WITNESS.

The correction is to simply not lock. The framework guarantees that callbacks are invoked with the softc mutex held. Your callback just does its work and returns; the framework releases the mutex on return.

26.15.2 Mistake 2: Calling Framework Primitives Without the Mutex Held

The opposite mistake is also common:

```
static int
my_userland_write(struct cdev *dev, struct uio *uio, int ioflag)
{
    struct my_softc *sc = dev->si_drv1;

    /* no lock taken */
    usbd_transfer_start(sc->sc_xfer[MY_BULK_TX]);    /* <-- wrong */
    return (0);
}
```

Most framework primitives (`usbd_transfer_start`, `usbd_transfer_stop`, `usbd_transfer_submit`) expect the caller to hold the associated mutex. Calling them without the mutex is a race: the framework’s own state can be modified by a concurrent callback while you are issuing the primitive.

The correction is to take the mutex around the call:

```
mtx_lock(&sc->sc_mtx);
usbd_transfer_start(sc->sc_xfer[MY_BULK_TX]);
mtx_unlock(&sc->sc_mtx);
```

This is the idiomatic pattern. The framework provides the locking; the driver provides the mutex.

26.15.3 Mistake 3: Forgetting USB_ERR_CANCELLED Handling

The framework uses USB_ERR_CANCELLED to tell a callback that its transfer is being torn down (typically during detach). If your callback handles this error the same way it handles other errors (for example by rearming the transfer), detach will hang forever because the transfer never actually stops.

The correct pattern is:

```
case USB_ST_ERROR:
    if (error == USB_ERR_CANCELLED) {
        return; /* do not rearm; the framework is tearing us down */
    }
    /* handle other errors, possibly rearm */
    break;
```

Omitting the cancellation check is one of the most common reasons a driver detaches cleanly in development (because the ref count happens to be zero) but hangs in production (because a read was in-flight when detach ran).

26.15.4 Mistake 4: Submitting to a Channel That Has Not Been Started

A transfer channel is inert until usbd_transfer_start has been called on it. Calling usbd_transfer_submit on an inactive channel is a no-op in some framework versions and a panic in others.

The correct pattern is to call usbd_transfer_start from userland-initiated work (in response to an open, for instance) and leave the channel active until detach. Do not call usbd_transfer_submit directly; let usbd_transfer_start schedule the first callback, and rearm from USB_ST_SETUP or USB_ST_TRANSFERRED.

26.15.5 Mistake 5: Assuming USB_GET_STATE Returns the Real Hardware State

USB_GET_STATE(xfer) returns the state the framework wants the callback to handle at this moment. It does not report the underlying hardware state. The three states USB_ST_SETUP, USB_ST_TRANSFERRED, and USB_ST_ERROR are framework concepts, not hardware concepts.

In particular, USB_ST_TRANSFERRED means “the framework thinks this transfer completed.” If the hardware is misbehaving (spurious transfer complete interrupts, split completions), the callback may be called with USB_ST_TRANSFERRED even when the actual transfer has not fully drained. This is rare, but when debugging, do not assume the framework state is ground truth about the hardware.

26.15.6 Mistake 6: Using M_WAITOK in a Callback

A USB callback runs in an environment where sleeping is not allowed. Memory allocations in a callback must use M_NOWAIT. Using M_WAITOK will assert or panic.

A more subtle version of this mistake is calling a helper that internally uses M_WAITOK. For example, some framework helpers sleep; calling them from a callback is forbidden. If you need to do work that would require

sleeping (DNS lookup, disk I/O, USB control transfers from a USB callback), queue it to a taskqueue and let the taskqueue worker do the work outside the callback.

26.15.7 Mistake 7: Forgetting `MODULE_DEPEND` on `usb`

A USB driver module that does not declare `MODULE_DEPEND(my, usb, 1, 1, 1)` will fail to load with a cryptic unresolved-symbol error:

```
link_elf_obj: symbol usbd_transfer_setup undefined
```

The symbol is undefined because the `usb` module has not been loaded, and the linker cannot resolve the driver's dependency on it. Adding the correct `MODULE_DEPEND` directive causes the kernel module loader to automatically load `usb` before your driver, which resolves the symbol and lets your driver attach.

Every USB driver must have `MODULE_DEPEND(drivename, usb, 1, 1, 1)`. Every UART-framework driver must have `MODULE_DEPEND(drivename, uart, 1, 1, 1)`. Every `ucom(4)` driver must depend on both `usb` and `ucom`.

26.15.8 Mistake 8: Mutable State in a Read-Only Path

Imagine a driver that exposes a status field through a `sysctl`. The `sysctl` handler reads the field from the `softc` without taking the mutex:

```
static int
my_sysctl_status(SYSCTL_HANDLER_ARGS)
{
    struct my_softc *sc = arg1;
    int val = sc->sc_status;    /* <-- unlocked read */
    return (SYSCTL_OUT(req, &val, sizeof(val)));
}
```

If the field can be updated by a callback (which runs under the mutex), and read by the `sysctl` handler (which does not take the mutex), you have a data race. On modern platforms, word-sized reads are usually atomic, so the race is often invisible. But on platforms where they are not, or when the field is wider than a word, you can get torn reads.

The correction is to take the mutex for the read:

```
mtx_lock(&sc->sc_mtx);
val = sc->sc_status;
mtx_unlock(&sc->sc_mtx);
```

Even if the race is invisible on x86, taking the lock documents your intent and protects against future changes (like widening the field to 64 bits).

26.15.9 Mistake 9: Stale Pointers After `usbd_transfer_unsetup`

`usbd_transfer_unsetup` frees the transfer channels. The pointer in `sc->sc_xfer[i]` is no longer valid after the call returns. If any other code in your driver uses that pointer after `unsetup`, the behaviour is undefined.

The correction is to zero the array after `unsetup`:

```
usbd_transfer_unsetup(sc->sc_xfer, MY_N_TRANSFERS);  
memset(sc->sc_xfer, 0, sizeof(sc->sc_xfer)); /* optional but defensive */
```

More importantly, structure your detach so that nothing in the driver can observe the stale pointers. This usually means setting a “detaching” flag in the softc before calling `unsetup`, and having every other code path check the flag before using the pointers.

26.15.10 Mistake 10: Not Zeroing the Softc’s detaching Flag at Attach Time

If your softc uses a detaching flag to coordinate detach, the flag must start at zero when `attach` is called. This is normally automatic (the framework zero-fills the softc), but if you have any field that needs a non-zero initial value, be careful not to accidentally initialise `detaching` to a nonzero value.

A driver that starts with `detaching = 1` will appear to “detach before it ever attached,” which shows up as a driver that attaches normally but refuses to respond to any I/O.

26.15.11 Mistake 11: Forgetting to Destroy the Device Node on Detach

If your driver creates a character device with `make_dev` in `attach`, you must destroy it with `destroy_dev` in `detach`. Forgetting this leaves a stale `/dev` entry that points to freed memory. Userland programs that open the stale node will panic the kernel.

The correction is to call `destroy_dev(sc->sc_cdev)` in `detach`, and always before the softc fields it references are freed.

A stronger pattern is to order the `destroy_dev` call first in `detach` (before any other cleanup). This blocks new opens and waits for existing opens to close, so that by the time the rest of `detach` runs, no userland code can reach the driver.

26.15.12 Mistake 12: Racing on the Character Device Open

Even with `destroy_dev` in the right place, there is a window between `attach` succeeding and the first `open()` succeeding where the driver’s state is being initialised. If your open handler assumes certain softc fields are valid, and `attach` has not finished initialising them when the first open arrives, the open will see garbage.

The correction is to call `make_dev` last in `attach`, only after everything else is fully initialised. This way, the `/dev` entry does not appear until the driver is ready to service opens. Correspondingly, call `destroy_dev` first in `detach`, before tearing anything down.

26.15.13 Mistake 13: Overlooking the TTY Layer's Own Locking

UART drivers integrate with the TTY layer, which has its own locking rules. In particular, the TTY layer holds `tty_lock` when it calls into the driver's `tsw_param`, `tsw_open`, and `tsw_close` methods. If the driver then takes another lock inside these methods, the lock order is `tty_lock -> driver_mutex`. If any other code path takes the driver mutex and then the tty lock, you have a lock order inversion, and WITNESS will catch it.

The correction is to respect the lock order that the framework establishes. For UART drivers, the order is documented in `/usr/src/sys/dev/uart/uart_core.c`. When in doubt, run under WITNESS with `WITNESS_CHECKORDER` enabled; it will detect any violation immediately.

26.15.14 Mistake 14: Not Handling Zero-Length Data in Read or Write

A userland read or write with a zero-length buffer is legal. Your driver must handle it, either by immediately returning zero or by propagating the zero-length request through the framework. Forgetting this case often produces a driver that “mostly works” but fails weird test scenarios.

The simplest correction is:

```
if (uio->uio_resid == 0)
    return (0);
```

at the top of your read and write functions.

26.15.15 Mistake 15: Copying Data Before Checking the Transfer Status

In a read path, a common mistake is to unconditionally copy data out of the USB buffer:

```
case USB_ST_TRANSFERRED:
    usbd_copy_out(pc, 0, sc->sc_rx_buf, actlen);
    /* hand data up to userland */
    break;
```

If the transfer was a short read (`actlen < wMaxPacketSize`), the copy is correct for exactly `actlen` bytes but the driver code may assume more. If the transfer was empty (`actlen == 0`), the copy does nothing and any subsequent code that operates on “just-received data” works on stale data from the previous transfer.

The correction is to always check `actlen` before acting on the data:

```
case USB_ST_TRANSFERRED:
    if (actlen == 0)
        goto rearm; /* nothing received */
    usbd_copy_out(pc, 0, sc->sc_rx_buf, actlen);
    /* work with exactly actlen bytes */
    rearm:
```

```
/* re-submit */  
break;
```

26.15.16 Mistake 16: Assuming termios Values Are in Standard Units

The `termios` structure's `c_ispeed` and `c_ospeed` fields contain baud rate values, but the encoding has historical oddities. On FreeBSD, speeds are integer values (9600, 38400, 115200). On some other systems, they are indices into a table. Porting code that assumed index-based speeds to FreeBSD without checking is a common source of “the driver thinks the baud rate is 13 instead of 115200” bugs.

The correction is to look at the actual FreeBSD implementation: `/usr/src/sys/sys/termios.h` and `/usr/src/sys/kern/tty.c`. The baud rate in FreeBSD `termios` is an integer bit rate. When your driver receives a `termios` in `param`, read `c_ispeed` and `c_ospeed` as integers.

26.15.17 Mistake 17: Missing `device_set_desc` or `device_set_desc_copy`

The `device_set_desc` family of calls sets the human-readable description that `dmesg` shows when the device attaches. Without it, `dmesg` shows a generic label (like “`my_drv0:` ”), which is confusing for users and for your own debugging.

The correction is to call `device_set_desc` in `probe` (not `attach`), before returning `BUS_PROBE_GENERIC` or similar:

```
static int  
my_probe(device_t dev)  
{  
    /* ... match check ... */  
    device_set_desc(dev, "My Device");  
    return (BUS_PROBE_DEFAULT);  
}
```

Use `device_set_desc_copy` when the string is dynamic (constructed from device data); the framework will free the copy when the device is detached.

26.15.18 Mistake 18: `device_printf` in the Data Path Without Rate Limiting

The `device_printf` call is fine for occasional messages. In a data-path callback, it is not, because every single transfer prints a line to `dmesg` and to the console. A 1 Mbps stream of characters becomes a flood of log messages.

The correction is the `DLOG_RL` pattern from Chapter 25: rate-limit data-path log messages to one per second, or one per thousand events, whichever is appropriate. Keep full logging in the configuration and error paths; rate-limit in the data path.

26.15.19 Mistake 19: Not Waking Readers on Device Removal

If a userland program is blocked in `read()` waiting for data, and the device is unplugged, the driver must wake the reader and return an error (typically `ENXIO` or `ENODEV`). Forgetting to do this leaves the read blocked forever, which is a resource leak and a hang.

The correction is to wake all sleepers in detach before returning:

```
mtx_lock(&sc->sc_mtx);
sc->sc_detaching = 1;
wakeup(&sc->sc_rx_queue);
wakeup(&sc->sc_tx_queue);
mtx_unlock(&sc->sc_mtx);
```

And in the read path, check the flag after waking:

```
while (sc->sc_rx_head == sc->sc_rx_tail && !sc->sc_detaching) {
    error = msleep(&sc->sc_rx_queue, &sc->sc_mtx, PZERO | PCATCH, "myrd", 0);
    if (error != 0)
        break;
}
if (sc->sc_detaching)
    return (ENXIO);
```

This is the idiomatic pattern and avoids the classic “userland process hangs after you unplug the device” bug.

26.15.20 Mistake 20: Thinking “It Works on My Machine” Is Enough

Driver bugs can be hardware-dependent. A driver that works on one machine may fail on another because of timing differences, interrupt delivery differences, or hardware quirks in the USB controller. A driver that works with one model of a device may fail with another model of the same family because of firmware differences.

The correction is to test on multiple machines, multiple USB hosts (xHCI, EHCI, OHCI), and multiple devices if possible. When something works on one and fails on another, the difference is information. Trace both, compare, and the bug usually becomes clear.

26.15.21 What To Do After You Make One of These Mistakes

You will make several of these mistakes. This is normal. The way to learn is: debug the failure, identify which mistake it was, understand why it caused the specific symptom, and add the correction to your mental toolkit. Keep a note of which mistakes you have made in practice. When you see a new driver failure, check your note; the answer is usually a mistake you have already solved once.

The specific mistakes above are collected from the author’s own experience writing and debugging USB and UART drivers on FreeBSD. They are not exhaustive, but they are representative of the kinds of issues that come up. Reading drivers in the tree, attending FreeBSD developer forums, and submitting your work for code review are all ways to accelerate this kind of learning.

26.16 Wrapping Up

Chapter 26 has taken you on a long tour. It began with the idea that a transport-specific driver is a Newbus driver plus a set of rules about how the transport works. It then built out the two transport-specific layers we are focusing on in Part 6: USB and serial.

On the USB side, you learned the host-and-device model, the descriptor hierarchy, the four transfer types, and the hot-plug lifecycle. You walked through a complete driver skeleton: the match table, the probe method, the attach method, the `softc`, the detach method, and the registration macros. You saw how `struct usb_config` declares transfer channels and how `usbd_transfer_setup` brings them to life. You followed the three-state callback state machine through bulk, interrupt, and control transfers, and you saw how `usbd_copy_in` and `usbd_copy_out` move data between the driver and the framework's buffers. You learned the locking rules around transfer operations and the retry policies drivers should choose. By the end of Section 3, you had a mental model that would let you write a bulk-loopback driver from scratch.

On the serial side, you learned that the TTY layer sits on top of two distinct frameworks: `uart(4)` for bus-attached UARTs and `ucom(4)` for USB-to-serial bridges. You saw the six-method structure of a `uart(4)` driver, the role of `uart_ops` and `uart_class`, and how the `ns8250` canonical driver implements each method. You learned how `termios` settings flow from `stty` through the TTY layer into the driver's `param` path, and how hardware flow control is implemented at the register level. For USB-to-serial devices, you saw the distinct `ucom_callback` structure and how configuration methods translate `termios` changes into vendor-specific USB control transfers.

For testing, you learned about `nmdm(4)` for pure-TTY testing, QEMU USB redirection for USB development, and a handful of userland tools (`cu`, `tip`, `stty`, `comcontrol`, `usbconfig`) that make driver development manageable even without constant hardware access. You saw that much of driver work is not register-level wrestling but careful arrangement of data flow through well-defined abstractions.

The hands-on labs and challenge exercises gave you concrete problems to work on. Each lab is short enough to finish in a sitting, and each challenge extends one of the core ideas from the main text.

Three habits from earlier chapters extended naturally into Chapter 26. The labelled-goto cleanup chain from Chapter 25 is the same pattern used in USB and UART attach routines. The `softc`-as-single-source-of-truth discipline from Chapter 25 is applied identically to USB and UART driver state. The `errno`-returning helper function pattern is unchanged. What Chapter 26 added was transport-specific vocabulary and transport-specific abstractions built on top of those habits.

There is also a habit that Chapter 26 has introduced which will stay with you: the three-state callback state machine (`USB_ST_SETUP`, `USB_ST_TRANSFERRED`, `USB_ST_ERROR`). Every USB driver uses it. Learning to read this state machine is learning to read every USB callback in the tree. When you open `uftdi.c`, `ucycom.c`, `uchcom.c`, or any other USB driver, you will see the same pattern. Recognising it is recognising the USB framework's core abstraction.

Transport-specific drivers are where the book's abstract framework concepts become concrete. From here on, every chapter in Part 6 will deepen your practical skill with one more transport or one more kind of kernel service. The Newbus foundation from Part 3, the character-device basics from Part 4, and the discipline themes from Part 5 are all in play simultaneously. You are no longer learning concepts in isolation; you are using them together.

26.17 Bridge to Chapter 27

Chapter 27 turns to network drivers. Much of the structure will feel familiar: there is a Newbus attachment, there is per-device state (called `if_softc` in network drivers), there is a match table, there is a probe-and-attach sequence, there are hot-plug considerations, and there is an integration with a higher framework. But the higher framework here is `ifnet(9)`, the interface-framework abstraction for network devices, and its idioms are different from those of USB and serial.

A network driver does not expose a character device. It exposes an interface, which is visible to userland through `ifconfig(8)`, through `netstat -i`, and through the socket layer. Instead of `read(2)` and `write(2)`, network drivers handle packet input and packet output through the network stack's pipeline. Instead of `termios` for configuration, they handle `SIOCSIFFLAGS`, `SIOCADDMULTI`, `SIOCSIFMEDIA`, and a host of other network-specific `ioctl`s.

Many network cards also happen to use USB or PCIe as their underlying transport. A USB Ethernet adapter, for example, sits on USB (via `if_cdce` or a vendor-specific driver) and exposes an `ifnet(9)` interface. A PCIe Ethernet card sits on PCIe and also exposes an `ifnet(9)` interface. Chapter 27 will show how the same `ifnet(9)` framework sits on top of these very different transports, and how the separation lets you write a driver that focuses on the packet-level protocol without worrying about the details of its transport.

One specific thing to look forward to is the contrast between how USB delivers packets (as transfer completions, one buffer at a time, with explicit flow control at the transfer level) and how PCIe-based network cards deliver packets (as DMA-from-hardware events with descriptor rings). The packet pipeline in the network stack is designed to hide this difference from the upper layers, but a driver author has to understand both models because they determine the driver's internal structure.

Chapter 27 will then turn to block device drivers (storage). That chapter will cover the GEOM framework, which is FreeBSD's layered block-device infrastructure. Block drivers have their own idioms: a different way of matching devices, a different way of exposing state (through GEOM providers and consumers), and a fundamentally different data flow model (read and write operations on sectors, with a strong consistency model).

Parts 7, 8, and 9 then cover the more specialised topics: kernel services and advanced kernel idioms, debugging and testing in depth, and distribution and packaging. By the end of the book, you will have written and maintained drivers across several transport layers and several kernel subsystems. The foundation you have built in Chapters 21 through 26 will be the common ground across all of that work.

For now, keep your `myfirst_usb` driver. You will not extend it in later chapters, but the patterns it demonstrates will appear again in network, storage, and kernel-service contexts. Having your own working example on hand, something you wrote and understand completely, is a resource that pays back many times over as the book progresses.

26.18 Quick Reference

This reference collects the most important APIs, constants, and file locations from Chapter 26 into one place. Keep it open while writing or reading a driver; it is faster than rediscovering each name from the source tree.

26.18.1 USB Driver APIs

Function	Purpose
<code>usbd_lookup_id_by_uaa(table, size, uaa)</code>	Match attach arg against match table
<code>usbd_transfer_setup(udev, &ifidx, xfer, config, n, priv, mtx)</code>	Allocate transfer channels
<code>usbd_transfer_unsetup(xfer, n)</code>	Free transfer channels
<code>usbd_transfer_submit(xfer)</code>	Queue a transfer for execution
<code>usbd_transfer_start(xfer)</code>	Activate a channel
<code>usbd_transfer_stop(xfer)</code>	Deactivate a channel
<code>usbd_transfer_pending(xfer)</code>	Query whether a transfer is outstanding
<code>usbd_transfer_drain(xfer)</code>	Wait for any pending transfer to complete
<code>usbd_xfer_softc(xfer)</code>	Retrieve the_softc from a transfer
<code>usbd_xfer_status(xfer, &actlen, &sumlen, &aframes, &nframes)</code>	Query transfer results
<code>usbd_xfer_get_frame(xfer, i)</code>	Get page-cache pointer for frame i
<code>usbd_xfer_set_frame_len(xfer, i, len)</code>	Set length of frame i
<code>usbd_xfer_set_frames(xfer, n)</code>	Set total frame count
<code>usbd_xfer_max_len(xfer)</code>	Query max transfer length
<code>usbd_xfer_set_stall(xfer)</code>	Schedule clear-stall on this pipe
<code>usbd_copy_in(pc, offset, src, len)</code>	Copy into framework buffer
<code>usbd_copy_out(pc, offset, dst, len)</code>	Copy out of framework buffer
<code>usbd_errstr(err)</code>	Error code to string
<code>USB_GET_STATE(xfer)</code>	Current callback state
<code>USB_VPI(vendor, product, info)</code>	Compact match table entry

26.18.2 USB Transfer Types (`usb.h`)

- `UE_CONTROL`: control transfer (request-response)
- `UE_ISOCHRONOUS`: isochronous (periodic, no retry)
- `UE_BULK`: bulk (reliable, no timing guarantee)
- `UE_INTERRUPT`: interrupt (periodic, reliable)

26.18.3 USB Transfer Direction

- `UE_DIR_IN`: device to host
- `UE_DIR_OUT`: host to device
- `UE_ADDR_ANY`: framework picks any matching endpoint

26.18.4 USB Callback States (`usbdi.h`)

- `USB_ST_SETUP`: ready to submit a new transfer
- `USB_ST_TRANSFERRED`: previous transfer succeeded
- `USB_ST_ERROR`: previous transfer failed

26.18.5 USB Error Codes (`usbdi.h`)

- `USB_ERR_NORMAL_COMPLETION`: success
- `USB_ERR_PENDING_REQUESTS`: outstanding work
- `USB_ERR_NOT_STARTED`: transfer not started
- `USB_ERR_CANCELLED`: transfer cancelled (e.g., detach)
- `USB_ERR_STALLED`: endpoint stalled
- `USB_ERR_TIMEOUT`: timeout expired
- `USB_ERR_SHORT_XFER`: received less data than requested
- `USB_ERR_NOMEM`: out of memory
- `USB_ERR_NO_PIPE`: no matching endpoint

26.18.6 Registration Macros

- `DRIVER_MODULE(name, parent, driver, evh, arg)`: register driver with kernel
- `MODULE_DEPEND(name, dep, min, pref, max)`: declare module dependency
- `MODULE_VERSION(name, version)`: declare module version
- `USB_PNP_HOST_INFO(table)`: export match table to devd
- `DEVMETHOD(name, func)`: declare method in method table
- `DEVMETHOD_END`: terminate method table

26.18.7 UART Framework APIs

Function	Header	Purpose
<code>uart_getreg(bas, offset)</code>	<code>uart.h</code>	Read a UART register
<code>uart_setreg(bas, offset, value)</code>	<code>uart.h</code>	Write a UART register
<code>uart_barrier(bas)</code>	<code>uart.h</code>	Memory barrier for register access
<code>uart_bus_probe(dev, regshft, regiowidth, rclk, rid, chan, quirks)</code>	<code>uart_bus.h</code>	Framework probe helper
<code>uart_bus_attach(dev)</code>	<code>uart_bus.h</code>	Framework attach helper
<code>uart_bus_detach(dev)</code>	<code>uart_bus.h</code>	Framework detach helper

26.18.8 uart_ops Methods

- `probe(bas)`: chip present?
- `init(bas, baud, databits, stopbits, parity)`: initialise chip
- `term(bas)`: shut down chip
- `putc(bas, c)`: send one character (polling)
- `rxready(bas)`: is data available?
- `getc(bas, mtx)`: read one character (polling)

26.18.9 ucom_callback Methods

- `ucom_cfg_open, ucom_cfg_close`: open/close hooks
- `ucom_cfg_param`: termios changed
- `ucom_cfg_set_dtr, ucom_cfg_set_rts, ucom_cfg_set_break, ucom_cfg_set_ring`: signal control
- `ucom_cfg_get_status`: read line and modem status bytes
- `ucom_pre_open, ucom_pre_param`: validation hooks (return `errno`)
- `ucom_ioctl`: chip-specific ioctl handler
- `ucom_start_read, ucom_stop_read`: enable/disable read
- `ucom_start_write, ucom_stop_write`: enable/disable write
- `ucom_tty_name`: customise the TTY device-node name
- `ucom_poll`: poll for events
- `ucom_free`: final cleanup

26.18.10 Key Source Files

- `/usr/src/sys/dev/usb/usb.h`: USB protocol definitions
- `/usr/src/sys/dev/usb/usbdi.h`: USB driver interface, `USB_ERR_*` codes
- `/usr/src/sys/dev/usb/usbdi_util.h`: convenience helpers
- `/usr/src/sys/dev/usb/usbdevs.h`: Vendor/product constants (build-generated by the FreeBSD build system from `/usr/src/sys/dev/usb/usbdevs`; not present in a clean source tree until the kernel or driver is built)
- `/usr/src/sys/dev/usb/controller/`: Host controller drivers
- `/usr/src/sys/dev/usb/misc/uled.c`: Simple LED driver (reference)
- `/usr/src/sys/dev/usb/serial/uftdi.c`: FTDI driver (reference)
- `/usr/src/sys/dev/usb/serial/usb_serial.h`: `ucom_callback` definition
- `/usr/src/sys/dev/usb/serial/usb_serial.c`: `ucom` framework
- `/usr/src/sys/dev/uart/uart.h`: `uart_getreg, uart_setreg, uart_barrier`
- `/usr/src/sys/dev/uart/uart_bus.h`: `uart_class, uart_softc`, bus helpers
- `/usr/src/sys/dev/uart/uart_cpu.h`: `uart_ops`, CPU-side glue
- `/usr/src/sys/dev/uart/uart_core.c`: UART framework body
- `/usr/src/sys/dev/uart/uart_tty.c`: UART-TTY integration
- `/usr/src/sys/dev/uart/uart_dev_ns8250.c`: ns8250 reference driver
- `/usr/src/sys/dev/ic/ns16550.h`: 16550 register definitions
- `/usr/src/sys/dev/nmdm/nmdm.c`: null-modem driver

26.18.11 Userland Diagnostic Commands

Command	Purpose
<code>usbconfig list</code>	List USB devices
<code>usbconfig -d ugenN.M dump_all_config_desc</code>	Dump descriptors
<code>usbconfig -d ugenN.M dump_stats</code>	Transfer statistics
<code>usbconfig -d ugenN.M reset</code>	Reset device
<code>stty -a -f /dev/device</code>	Show termios settings
<code>stty 115200 -f /dev/device</code>	Set baud rate
<code>comcontrol /dev/device</code>	Show modem signals
<code>cu -l /dev/device -s speed</code>	Interactive session
<code>tip name</code>	Named connection (via <code>/etc/remote</code>)
<code>kldload mod.ko</code>	Load kernel module
<code>kldunload mod</code>	Unload kernel module
<code>kldstat</code>	List loaded modules
<code>dmesg -w</code>	Stream kernel messages
<code>sysctl hw.usb.*</code>	Query USB framework
<code>sysctl dev.uart.*</code>	Query UART instances

26.18.12 Standard Development Flags

Debug-mode kernel options to enable during development: - options `INVARIANTS`: assertion checking - options `INVARIANT_SUPPORT`: required alongside `INVARIANTS` - options `WITNESS`: lock order checking - options `WITNESS_SKIPSPIN`: skip spin locks in `WITNESS` (perf) - options `WITNESS_CHECKORDER`: verify every lock acquisition - options `DDB`: kernel debugger - options `KDB`: kernel debugger support - options `USB_DEBUG`: extensive USB logging

These options should be enabled on development machines, not production.

26.19 Glossary

The following terms appeared in this chapter. Some are new; others were introduced earlier and are repeated here for convenience. Definitions are brief and intended as a quick reminder, not as a replacement for the main-text explanations.

Address (USB). A number from 1 to 127 that the host assigns to a device during enumeration. Each physical device on a bus has a unique address.

Attach. The framework-called method where a driver takes ownership of a newly discovered device, allocates resources, initialises state, and begins operation. Paired with `detach`.

Bulk transfer. A USB transfer type designed for reliable, high-throughput, non-time-critical data. Used for mass storage, printers, network adapters.

Callout. A FreeBSD mechanism for scheduling a function to run after a specific delay. Used by drivers for timeouts and periodic tasks.

Callin node. A TTY device node (usually `/dev/ttyuN`) where opening blocks until carrier detect is asserted. Historically used for answering incoming modem calls.

Callout node. A TTY device node (usually `/dev/cuaN`) where opening does not wait for carrier detect. Used for initiating connections or for non-modem devices.

CDC ACM. Communication Device Class, Abstract Control Model. The USB standard for virtual serial ports. Handled in FreeBSD by the `u3g` driver.

Character device. A UNIX device abstraction for byte-oriented devices. Exposed to userland through `/dev` entries. Introduced in Chapter 24.

Class driver. A USB driver that handles an entire class of devices (all HID devices, all mass-storage devices) rather than a single vendor's product. Matches on interface class/subclass/protocol.

Clear-stall. A USB operation that clears a stall condition on an endpoint. Handled by the FreeBSD USB framework when `usbd_xfer_set_stall` is called.

Configuration (USB). A named set of interfaces and endpoints a USB device can expose. A device usually has one configuration but may have several.

Control transfer. A USB transfer type designed for small, infrequent, request-response exchanges. Used for configuration and status.

cuau. Naming prefix for the callout-side TTY device of a bus-attached UART. Example: `/dev/cuau0`.

cuaU. Naming prefix for the callout-side TTY device of a USB-provided serial port. Example: `/dev/cuaU0`.

Descriptor (USB). A small data structure a USB device provides, describing itself or one of its components. Types include device, configuration, interface, endpoint, and string descriptors.

Detach. The framework-called method where a driver releases all resources and prepares for the device to vanish. Paired with `attach`.

devd. The FreeBSD device-event daemon that reacts to kernel notifications about device attach and detach. Responsible for auto-loading modules for newly-discovered devices.

Device (USB). A single physical USB peripheral connected to a port. Contains one or more configurations.

DMA. Direct Memory Access. A mechanism where hardware can read or write memory without CPU involvement. Used by high-performance USB host controllers and PCIe network cards.

Echo loopback. A test configuration in which a device echoes whatever it receives, used to validate bidirectional data flow.

Endpoint. A USB communication channel within an interface. Each endpoint has a direction (IN or OUT) and a transfer type. Matches one hardware FIFO on the device.

Enumeration. The USB process by which a newly attached device is discovered, assigned an address, and has its descriptors read by the host.

FIFO (hardware). A small buffer on a UART or USB chip that holds bytes during transfer. Typical 16550 FIFO is 16 bytes; many modern UARTs have 64 or 128.

FTDI. A company that makes popular USB-to-serial adapter chips. Drivers for FTDI chips are in `/usr/src/sys/dev/usb/serial/uftdi.c`.

ifnet(9). The FreeBSD framework for network device drivers. Covered in Chapter 27.

Interface (USB). A logical grouping of endpoints within a USB device. A multi-function device can expose multiple interfaces.

Interrupt handler. A function the kernel runs in response to a hardware interrupt. In the UART context, the framework provides a default interrupt handler.

Interrupt transfer. A USB transfer type designed for low-bandwidth, periodic, latency-critical data. Used for keyboards, mice, HIDs.

Isochronous transfer. A USB transfer type designed for real-time streams with guaranteed bandwidth but no delivery guarantee. Used for audio and video.

kldload, kldunload. FreeBSD commands for loading and unloading kernel modules.

kobj. FreeBSD's object-oriented kernel framework. Used for method dispatch in Newbus and other subsystems.

Match table. An array of `STRUCT_USB_HOST_ID` (for USB) or equivalent entries that a driver uses to declare which devices it supports.

Modem control register (MCR). A 16550 register that controls modem output signals (DTR, RTS).

Modem status register (MSR). A 16550 register that reports modem input signals (CTS, DSR, CD, RI).

nmdm(4). FreeBSD's null-modem driver. Creates pairs of linked virtual TTYs for testing. Loaded with `kldload nmdm`.

ns8250. A canonical 16550-compatible UART driver for FreeBSD. At `/usr/src/sys/dev/uart/uart_dev_ns8250.c`.

Pipe. A term for a bidirectional USB transfer channel from the host's perspective. A host has one pipe per endpoint.

Port (USB). A downstream attachment point on a hub. Each port can have one device (which may itself be a hub).

Probe. The framework-called method where a driver examines a candidate device and decides whether to attach. Returns zero for a match, nonzero `errno` for a reject.

Probe-and-attach. The two-phase handshake by which Newbus binds drivers to devices. Probe tests the match; attach does the work.

Retry policy. A driver's rule for what to do when a transfer fails. Common policies: rearm on every error, rearm up to N times then give up, rearm only for specific errors.

Ring buffer. A fixed-size circular buffer used by the UART framework to buffer data between the chip and the TTY layer.

RTS/CTS. Request To Send / Clear To Send. Hardware flow-control signals on a serial port.

Softc. The per-device state a driver maintains. Named after “software context” by analogy with hardware register state.

Stall (USB). A signal from a USB endpoint that it is not ready to accept more data until explicitly cleared by the host.

stty(1). Userland utility for inspecting and changing TTY settings. Maps directly onto `termios` fields.

Taskqueue. A FreeBSD mechanism for deferring work to a worker thread. Used by drivers that need to do something that cannot run in an interrupt context.

termios. A POSIX structure that describes a TTY’s configuration: baud rate, parity, flow control, line discipline flags, and many others. Set and queried by `tcsetattr(3)` and `tcgetattr(3)` from userland, or by `stty(1)`.

Transfer (USB). A single logical operation on a USB channel. Can be a single packet or many.

TTY. Teletype. The UNIX abstraction for a serial device. Character-at-a-time I/O, line discipline, signal generation, terminal control.

ttydevsw. The structure a TTY driver uses to register its operations with the TTY layer. Analogous to `cdevsw` for character devices.

ttyu. Naming prefix for the callin-side TTY device of a bus-attached UART. Example: `/dev/ttyu0`.

uart(4). FreeBSD’s framework for UART drivers. Handles registration, buffering, TTY integration. Drivers implement `uart_ops` hardware methods.

uart_bas. “UART Bus Access Structure.” The framework’s abstraction for register access to a UART, hiding whether the registers are in I/O space or memory-mapped.

uart_class. The framework descriptor that identifies a UART chip family. Pairs with `uart_ops` to give the framework everything it needs.

uart_ops. The table of six hardware-specific methods (`probe`, `init`, `term`, `putc`, `rxready`, `getc`) that a UART driver implements.

ucom(4). FreeBSD’s framework for USB-to-serial device drivers. Sits on top of USB transfers, provides TTY integration.

ucom_callback. The structure a `ucom(4)` client uses to register its callbacks with the framework.

ugen(4). FreeBSD’s generic USB driver. Exposes raw USB access through `/dev/ugenN.M` for userland programs. Used when no specific driver matches.

uhub. The FreeBSD driver for USB hubs (including the root hub). A class driver attaches to `uhub`, not to the USB bus directly.

usbconfig(8). Userland utility for inspecting and controlling USB devices. Can dump descriptors, reset devices, enumerate state.

usb_config. A C structure a USB driver uses to declare each of its transfer channels: type, endpoint, direction, buffer size, flags, callback.

usb_fifo. A USB framework abstraction for byte-stream `/dev` nodes. Generic alternative to writing a custom `cdevsw`.

usb_template(4). FreeBSD's USB device-side (gadget) framework. Used on hardware that can act as both USB host and USB device.

usb_xfer. An opaque structure representing a single USB transfer channel. Allocated by `usbd_transfer_setup`, freed by `usbd_transfer_unsetup`.

usbd_copy_in, usbd_copy_out. Helpers for copying data between plain C buffers and USB framework buffers. Must be used instead of direct pointer access.

usbd_lookup_id_by_uaa. Framework helper that compares a USB attach argument against a match table and returns zero on match.

usbd_transfer_setup, _unsetup. The calls that allocate and free transfer channels. Called from `attach` and `detach` respectively.

usbd_transfer_submit. The call that hands a transfer to the framework for execution on the hardware.

usbd_transfer_start, _stop. The calls that activate or deactivate a channel. Activate triggers a callback in `USB_ST_SETUP`; deactivate cancels in-flight transfers.

USB_ST_SETUP, _TRANSFERRED, _ERROR. The three states of a USB transfer callback, as returned by `USB_GET_STATE(xfer)`.

USB_ERR_CANCELLED. The error code the framework passes to a callback when a transfer is being torn down (typically during detach).

USB_ERR_STALLED. The error code when a USB endpoint returns a STALL handshake. Usually handled by calling `usbd_xfer_set_stall`.

VID/PID. Vendor ID / Product ID. A pair of 16-bit numbers that uniquely identifies a USB device model.

WITNESS. A FreeBSD kernel debugging option that tracks lock acquisition order and warns about violations.

Callin device. A TTY device (named `/dev/ttyuN` or `/dev/ttyUN`) that blocks on open until the modem's carrier detect (CD) signal is asserted. Used by programs that accept incoming calls.

Callout device. A TTY device (named `/dev/cuaun` or `/dev/cuaUN`) that opens immediately without waiting for carrier detect. Used by programs that initiate connections.

comcontrol(8). Userland utility for controlling TTY options (drain behaviour, DTR, flow control) that are not exposed through `stty`.

Descriptor (USB). A data structure that a USB device returns when the host asks for its identity, configuration, interfaces, or endpoints. Hierarchical: device descriptor contains configuration descriptors; configurations contain interfaces; interfaces contain endpoints.

Endpoint (USB). A named, typed communication channel inside a USB device. Has an address (1 through 15), a direction (IN or OUT), a type (control, bulk, interrupt, isochronous), and a maximum packet size.

Line discipline. The TTY layer's pluggable layer between the driver and userland. Standard disciplines include `termios` (canonical and raw modes). Line disciplines translate between raw bytes and the behaviour a user program expects.

msleep(9). The kernel sleep primitive used to block a thread on a channel with a mutex held. Paired with `wakeup(9)`, it implements producer-consumer patterns inside drivers.

mtx_sleep. A synonym for `msleep` used in some parts of the tree. Functionally identical.

Open/close pair. The character device methods `d_open` and `d_close`. Every driver that exposes a `/dev` node must handle these. Opens are usually where channels are started; closes are usually where channels are stopped.

Short transfer. A USB transfer that completes with fewer bytes than requested. Normal for bulk IN (where the device sends a short packet to signal “end of message”) and for interrupt IN (where the device sends a short packet when it has less data than the maximum). Always check `actlen`.

USETW. A FreeBSD macro for setting a little-endian 16-bit field inside a USB descriptor buffer. The USB wire format is always little-endian, so `USETW` hides the byte-swap.

This glossary is not exhaustive; it covers the terms this chapter actually used. For a broader FreeBSD USB reference, the `usbdi(9)` manual page is the definitive source. For the UART framework, the source in `/usr/src/sys/dev/uart/` is the reference. When you encounter an unfamiliar term in either place, check here first; if not defined, go to the source.

26.19.1 A Closing Note on Terminology Precision

One last piece of advice on vocabulary. The USB, TTY, and FreeBSD communities each have their own careful distinctions between terms that sound like synonyms. Confusing these in conversation with more experienced developers is a quick way to sound unsure; using them precisely is a quick way to sound at home.

“Device” in the USB context means the whole USB peripheral (the keyboard, the mouse, the serial adapter). “Interface” means a logical grouping of endpoints inside the device. An interface implements one function; a device can have multiple interfaces. When you say “the USB device is a composite device,” you are saying it has multiple interfaces.

“Endpoint” and “pipe” are related but distinct. An endpoint is on the device; a pipe is the host’s view of a connection to that endpoint. In FreeBSD driver code, the term “transfer channel” is often used in place of “pipe,” because “pipe” overloads a more common meaning in UNIX.

“Transfer” and “transaction” are also distinct. A transfer is a logical operation (a read request for *N* bytes); a transaction is the USB-level packet exchange that realises it. A bulk transfer of 64 bytes to an endpoint with a maximum packet size of 64 is one transfer and one transaction. A bulk transfer of 512 bytes to the same endpoint is one transfer and eight transactions.

“UART” and “serial port” are closely related but not identical. A UART is the chip (or the chip’s logic block); a serial port is the physical connector and its wiring. One UART can back multiple serial ports in some configurations; one serial port is always backed by exactly one UART.

“TTY” and “terminal” are related. A TTY is the kernel abstraction for character-at-a-time I/O; a terminal is the userland view. A TTY has a controlling terminal property; a terminal has a TTY that it uses. In driver code, TTY is almost always the more precise term.

Getting these right in writing and in code comments signals that you understand the design. And when you read someone else’s code or documentation, noticing which term they chose tells you which layer of abstraction they are thinking about.

27 Working with Storage Devices and the VFS Layer

27.1 Introduction

In the previous chapter, we walked carefully through the life of a USB serial driver. We followed the device from the moment the kernel noticed it on the bus, through probe and attach, into its active life as a character device, and finally out through detach when the hardware was unplugged. That walkthrough taught us how transport-specific drivers live inside FreeBSD. They participate in a bus, they expose a user-facing abstraction, and they accept that they may vanish at any moment because the underlying hardware is removable.

Storage drivers live in a very different country. The hardware is still real, and many storage devices can still be removed unexpectedly, but the role of the driver shifts in an important way. A USB serial adapter offers a stream of bytes to one process at a time. A storage device offers a block-addressable, long-lived, structured surface on which filesystems are built. When a user plugs in a USB serial adapter, they might immediately open `/dev/cuaU0` and begin a session. When a user plugs in a disk, they almost never read it as a raw stream. They mount it, and from that moment on the disk disappears behind a filesystem, behind a cache, behind the Virtual File System layer, and behind the many processes that share files on it.

This chapter teaches you what happens on the driver side of that arrangement. You will learn what the VFS layer is, how it differs from `devfs`, and how storage drivers plug into the GEOM framework rather than speaking to the VFS layer directly. You will write a small pseudo block device from scratch, expose it as a GEOM provider, give it a working backing store, watch `newfs_ufs` format it, mount the result, create files on it, unmount it cleanly, and detach it without leaving footprints in the kernel. By the end of the chapter you will have a working mental model of the storage stack and a concrete example driver that exercises every layer we discuss.

The chapter is long because the topic is layered. Unlike a character driver, where the main unit of interaction is a single `read` or `write` call from a process, a storage driver lives inside a chain of frameworks. Requests travel from a process through VFS, through the buffer cache, through the filesystem, through GEOM, and only then reach the driver. Replies travel back the other way. Understanding that chain is essential before writing any real storage code, and it is essential again when diagnosing the kind of subtle failures that appear only under load or during unmount. We will move slowly through the foundations, then gradually bring more layers into view.

As with Chapter 26, the goal here is not to ship a production block driver. The goal is to give you a sturdy, correct, readable first block driver that you understand completely. Real production storage drivers, for SATA disks, NVMe drives, SCSI controllers, SD cards, or virtual block devices, build on the same patterns. Once the foundations are clear, the step from pseudo to real is mostly a matter of replacing the backing store with code that talks to hardware registers and DMA engines, and of handling the much richer error and recovery surface that real disks expose.

You will also see how storage drivers interact with tools the reader already knows from the user side of FreeBSD. `mdconfig(8)` will appear as a close cousin of our driver, since the kernel's `md(4)` RAM disk is exactly the kind of thing we are building. `newfs_ufs(8)`, `mount(8)`, `umount(8)`, `diskinfo(8)`, `gstat(8)`, and `geom(8)` will

become tools of verification, not just tools that other people use. The chapter is structured so that by the time you finish, you can look at the output of `gstat -I 1` while running `dd` against your device and read it with understanding.

Finally, a note on what we will not cover here. We will not write a real bus driver that talks to a physical storage controller. We will not discuss the internals of UFS, ZFS, FUSE, or other specific filesystems beyond what is needed to understand how they meet a block device at the boundary. We will not cover DMA, PCIe, NVMe queues, or SCSI command sets. All of those topics deserve their own treatment and, where relevant, will appear in later chapters that cover specific buses and specific subsystems. What we will do here is give you a complete, self-contained block-layer experience that is representative of how all storage drivers in FreeBSD integrate with the kernel.

Take your time with this chapter. Read slowly, type the code, boot the module, format it, mount it, break it on purpose, watch what happens. The storage stack rewards patience and punishes shortcuts. You are not in a race.

27.2 How Part 6 Differs from Parts 1 Through 5

A short framing note before the chapter begins. Chapter 27 sits inside a Part that asks you to change one specific habit, and that shift is easier to make when it is named up front.

Parts 1 through 5 built a single running driver, `myfirst`, through twenty consecutive chapters, each adding one discipline to the same source tree. Chapter 26 extended that family with `myfirst_usb` as a transport sibling so the step into real hardware would not also be a step into unfamiliar source. **From Chapter 27 onward, the running `myfirst` driver pauses as the backbone of the book.** Part 6 shifts to new, self-contained demos that fit each subsystem it teaches: a pseudo block device for storage here in Chapter 27, and a pseudo network interface for networking in Chapter 28. These demos are parallel to `myfirst` in spirit but distinct in code, because the patterns that define a storage driver or a network driver do not fit the character-device mould that `myfirst` grew from.

The **discipline and didactic shape continue unchanged**. Each chapter still walks you through probe, attach, the primary data path, the cleanup path, labs, challenge exercises, troubleshooting, and a bridge to the next chapter. Each chapter still grounds its examples in real FreeBSD source under `/usr/src`. The habits you built in Chapters 25 and earlier, the labelled-goto cleanup chain, rate-limited logging, `INVARIANTS` and `WITNESS`, the production-readiness checklist, carry over without modification. What changes is the code artefact in front of you: a small, focused driver whose shape matches the subsystem under study, rather than one more stage in the `myfirst` timeline.

This is a deliberate didactic choice, not an accident of scope. A storage driver and a network driver each have their own lifecycle, their own data flow, their own preferred idioms, and their own frameworks to plug into. Teaching them as fresh drivers, rather than as further mutations of `myfirst`, keeps the focus on what makes each subsystem distinctive. A reader who tries to stretch `myfirst` into a block device or a network interface quickly ends up with code that teaches nothing about storage or networking. Fresh demos are the cleaner path, and they are the path this Part takes.

Part 7 returns to cumulative learning, but rather than resuming a single running driver, it revisits the drivers you have already written (`myfirst`, `myfirst_usb`, and the Part 6 demos) and teaches the production-minded topics

that matter once a first version of a driver exists: portability across architectures, advanced debugging, performance tuning, security review, and contribution to the upstream project. The habit of building cumulatively stays with you; only the specific artefact in front of you changes.

Keep this framing in mind as Chapter 27 unfolds. If the switch from `myfirst` to a new pseudo block device feels jarring after twenty chapters of the same source tree, that reaction is expected and passes quickly, usually by the end of Section 3.

27.3 Reader Guidance: How to Use This Chapter

This chapter is designed as a guided course through the storage side of the FreeBSD kernel. It is one of the longer chapters in the book because the subject matter is layered and every layer has its own vocabulary, its own concerns, and its own failure modes. You do not need to rush through it.

If you choose the **reading-only path**, expect to spend around two to three hours going through the chapter carefully. You will come away with a clear picture of how VFS, the buffer cache, filesystems, GEOM, and the block device boundary fit together, and you will have a concrete driver in front of you as an anchor for your mental model. This is a legitimate way to use the chapter, especially on a first pass.

If you choose the **reading-plus-labs path**, plan for four to six hours spread across one or two evenings, depending on your comfort with kernel modules from Chapter 26. You will build the driver, format it, mount it, watch it under load, and take it apart safely. Expect the mechanics of `kldload`, `kldunload`, `newfs_ufs`, and `mount` to become second nature by the end.

If you choose the **reading-plus-labs-plus-challenges path**, plan for a weekend or two evenings spread over a week. The challenges extend the driver in small directions that matter in practice: adding optional flush semantics, responding to `BIO_DELETE` with zeroing, supporting multiple units, exporting extra attributes through `disk_getattr`, and enforcing read-only mode cleanly. Each challenge is self-contained and uses only what the chapter has already covered.

Whichever path you choose, do not skip the troubleshooting section. Storage bugs tend to look alike from the outside, and the ability to recognise them by symptom is far more useful in practice than memorising the names of every function in GEOM. The troubleshooting material is placed near the end for readability, but you may find yourself turning back to it while working through the labs.

A word on prerequisites. This chapter builds directly on Chapter 26, so at minimum you should be comfortable writing a small kernel module, declaring a softc, allocating and freeing resources, and walking through the load and unload path. You should also be comfortable enough with the shell to run `kldload`, `kldstat`, `dmesg`, `mount`, and `umount` without stopping to look up flags. If any of that feels unfamiliar, it is worth revisiting Chapters 5, 14, and 26 before continuing.

You should work on a throwaway FreeBSD 14.3 system, a virtual machine, or a branch where you do not mind the occasional kernel panic. A panic is unlikely if you follow the text carefully, but the cost of a mistake on your development laptop is much higher than the cost of a mistake on a VM snapshot you can roll back. We have said this before and we will keep saying it: kernel work is safe when you work in a safe place.

27.3.1 Work Section by Section

The chapter is organised as a progression. Section 1 introduces VFS. Section 2 contrasts devfs with VFS and positions our driver in that contrast. Section 3 registers a minimal pseudo block device. Section 4 exposes it as a GEOM provider. Section 5 implements real read and write paths. Section 6 mounts a filesystem on top. Section 7 gives the device persistence. Section 8 teaches safe unmount and cleanup. Section 9 talks about refactoring, versioning, and what to do as the driver grows.

You are meant to read them in order. Each section assumes the previous sections are fresh in your mind, and the labs build on each other. If you jump in the middle, pieces will look strange.

27.3.2 Type the Code

Typing the code by hand remains the most effective way to internalise kernel idioms. The companion files under `examples/part-06/ch27-storage-vfs/` exist so that you can check your work, not so that you can skip the typing. Reading code is not the same as writing it.

27.3.3 Open the FreeBSD Source Tree

You will be asked several times to open real FreeBSD source files, not only the companion examples. The files of interest include `/usr/src/sys/geom/geom.h`, `/usr/src/sys/sys/bio.h`, `/usr/src/sys/geom/geom_disk.h`, `/usr/src/sys/dev/md/md.c`, and `/usr/src/sys/geom/zero/g_zero.c`. Each of these is a primary reference, and the prose in this chapter will often refer back to them. If you have not already cloned or installed the 14.3 source tree, now is a good moment to do so.

27.3.4 Use Your Lab Logbook

Keep your lab logbook from Chapter 26 open while you work. You will want to record the output of `gstat -I 1`, the messages emitted by `dmesg` when you load and unload the module, the time it takes to format the device, and any warnings or panics you see. Kernel work is much easier when you keep notes, because many symptoms look similar at first glance and the logbook lets you compare across sessions.

27.3.5 Pace Yourself

If you feel your understanding blurring in a particular section, stop. Read it again. Try a small experiment on the running module. Do not push through a section that has not settled. Storage drivers punish confusion more severely than character drivers, because confusion at the block layer often becomes filesystem corruption at the higher layer, and filesystem corruption takes time and care to repair even in a throwaway VM.

27.4 How to Get the Most Out of This Chapter

The chapter is structured so that every section adds exactly one new concept on top of what came before. To make the most of that structure, treat the chapter as a workshop rather than as a reference. You are not here to find a quick answer. You are here to build a correct mental model.

27.4.1 Work in Sections

Do not read the whole chapter end to end without stopping. Read one section, then pause. Try the experiment or lab that goes with it. Look at the related FreeBSD source. Write a few lines in your logbook. Only then move on. Storage programming in the kernel is strongly cumulative, and skipping ahead usually means that you will be confused about the next thing for a reason that was explained two sections ago.

27.4.2 Keep the Driver Running

Once you have loaded the driver in Section 3, keep it loaded as much as possible while you read. Modify it, reload, poke it with `gstat`, run `dd` against it, call `diskinfo` on it. Having a live, observable example is far more valuable than any amount of reading. You will notice things that no chapter could ever tell you about, because no chapter can show you real timing, real jitter, or real corner cases in your particular setup.

27.4.3 Consult Manual Pages

FreeBSD's manual pages are part of the teaching material, not a separate formality. Section 9 of the manual is where the kernel interfaces live. We will refer several times to pages such as `g_bio(9)`, `geom(4)`, `DEVICE_IDENTIFY(9)`, `disk(9)`, `bus_dma(9)`, and `devstat(9)`. Read them alongside this chapter. They are shorter than they look, and they are written by the same community that wrote the kernel you are working inside.

27.4.4 Type the Code, Then Mutate It

When you build the driver from the companion examples, type it first. Once it works, start changing things. Rename a method and watch the build fail. Remove an `if` branch and watch what happens when you load the module. Hardcode a smaller media size and watch `newfs_ufs` react. Kernel code becomes understandable through deliberate mutation far more than through pure reading.

27.4.5 Trust the Tooling

FreeBSD gives you a wealth of tools for inspecting the storage stack: `geom`, `gstat`, `diskinfo`, `dd`, `mdconfig`, `dmesg`, `kldstat`, `sysctl`. Use them. When something goes wrong, the first move is almost never to read more source. It is to ask the system what state it is in. `geom disk list` and `geom part show` are often more informative than five minutes of `grep`.

27.4.6 Take Breaks

Kernel work is cognitively dense. Two or three focused hours are usually more productive than a seven-hour sprint. If you catch yourself making the same typo three times, or copy-pasting without reading, that is your cue to stand up for ten minutes.

With those habits established, let us begin.

27.5 Section 1: What Is the Virtual File System Layer?

When a process opens a file on FreeBSD, it calls `open(2)` with a path. That path might resolve to a file on UFS, a file on ZFS, a file on a remotely mounted NFS share, a pseudo-file in `devfs`, a file under `procfs`, or even a file inside a FUSE-mounted userland filesystem. The process cannot tell. The process receives a file descriptor and then reads and writes as if there were only one kind of file in the world. That uniformity is not an accident. It is the work of the Virtual File System layer.

27.5.1 The Problem VFS Solves

Before VFS, UNIX kernels generally knew how to talk to one filesystem only. If you wanted a new filesystem, you modified the code paths for `open`, `read`, `write`, `stat`, `unlink`, `rename`, and every other system call that touched files. That approach worked for a while, but it did not scale. New filesystems arrived: NFS, for remote access. MFS, for in-memory scratch space. `Procfs`, for exposing process state. ISO 9660, for CD-ROM media. FAT, for interoperability. Every addition meant new forks in every file-related system call.

Sun Microsystems introduced the Virtual File System architecture in the mid-1980s as a way out of this mess. The idea is simple. The kernel talks to a single abstract interface, defined in terms of generic operations on generic file objects. Each concrete filesystem registers implementations of those operations, and the kernel calls them through function pointers. When the kernel needs to read a file, it does not know or care whether the file lives on UFS or NFS or ZFS. It knows there is a node with a `VOP_READ` method, and it calls that method.

FreeBSD adopted this architecture and has extended it significantly over the decades. The result is that adding a filesystem to FreeBSD no longer requires modifying the core system calls. A filesystem is a separate kernel module that registers a set of operations with VFS, and from that moment VFS routes the right requests to it.

27.5.2 The VFS Object Model

VFS defines three main kinds of objects.

The first is the **mount point**, represented in the kernel by `struct mount`. Every mounted filesystem has one, and it records where in the namespace the filesystem is attached, what flags it has, and which filesystem code is responsible for it.

The second is the **vnode**, represented by `struct vnode`. A vnode is the kernel's handle on a single file or directory within a mounted filesystem. It is not the file itself. It is the kernel's runtime representation of that file for as long as something in the kernel cares about it. Every file that a process has open has a vnode. Every directory the kernel is walking through has a vnode. When nothing holds a reference to a vnode, it can be reclaimed, and the kernel keeps a pool of them to avoid pressure on small-inode cases.

The third is the **vnode operations vector**, represented by `struct vop_vector`, which lists the operations each filesystem must implement on vnodes. The operations have names like `VOP_LOOKUP`, `VOP_READ`, `VOP_WRITE`, `VOP_CREATE`, `VOP_REMOVE`, `VOP_GETATTR`, and `VOP_SETATTR`. Each filesystem provides a pointer to its own vector, and the kernel invokes operations through these vectors whenever it needs to do anything to a file.

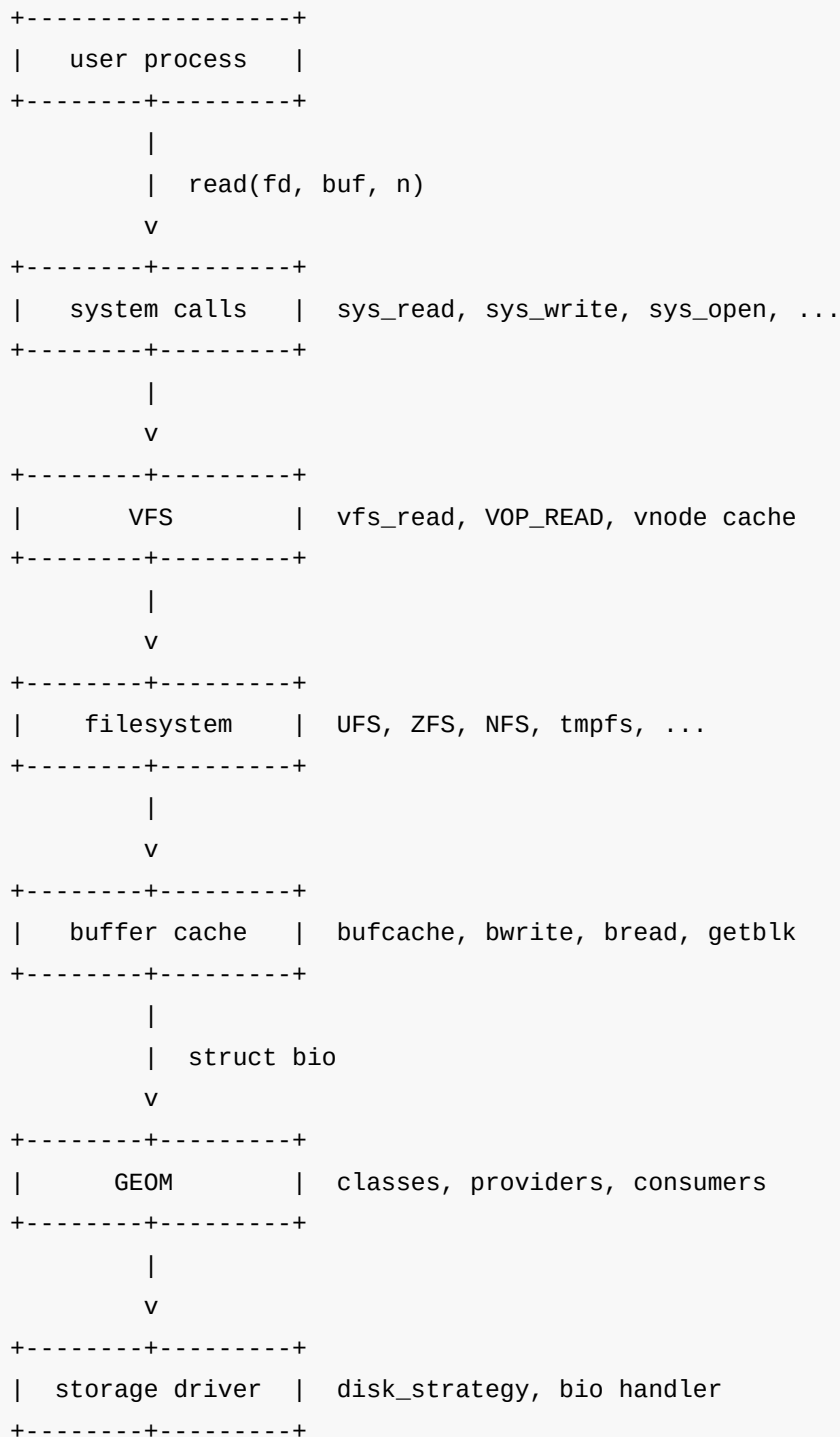
The elegant thing about this design is that from the system call side of the kernel, only the abstract interface matters. The system call layer calls `VOP_READ(vp, uio, ioflag, cred)` and does not care whether `vp` belongs to UFS, ZFS, NFS, or `tmpfs`. From the filesystem side, only the abstract interface matters too. UFS implements the vnode operations and never sees the system call code.

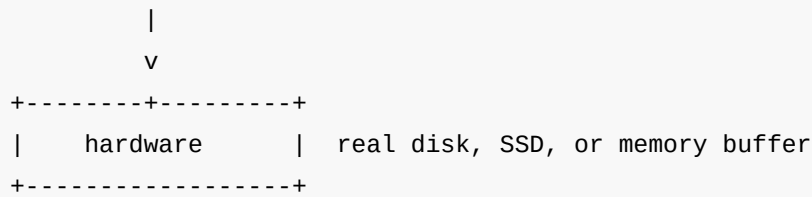
27.5.3 Where Storage Drivers Fit

Here is the question that matters for this chapter. If VFS is where filesystems live, where do storage drivers live?

The answer is: not directly inside VFS. A storage driver does not implement `VOP_READ`. It implements a much lower-level abstraction that looks like a disk. Filesystems then sit on top, consuming the disk-like abstraction, translating file-level operations into block-level operations, and calling down.

The chain of layers between a process and a block device in FreeBSD typically looks like this.





Each layer in this stack has a job. VFS hides filesystem differences from system calls. The filesystem translates files into blocks. The buffer cache holds recently used blocks in RAM. GEOM routes block requests through transforms, partitions, and mirrors. The storage driver converts block requests into real I/O. The hardware does the work.

For this chapter, almost everything we do happens at the bottom two layers: GEOM and the storage driver. We will touch the filesystem layer briefly when we mount UFS on our device, and we will touch VFS only in the sense that `mount(8)` calls it. The layers above GEOM are not our code.

27.5.4 VFS in the Kernel Source

If you want to look at VFS directly, the entry points are under `/usr/src/sys/kern/vfs_*.c`. The vnode layer lives in `vfs_vnops.c` and `vfs_subr.c`. The mount side lives in `vfs_mount.c`. The vnode operations vector is defined and handled in `vfs_default.c`. UFS, our primary filesystem in this chapter, lives under `/usr/src/sys/ufs/ufs/` and `/usr/src/sys/ufs/ffs/`. You do not need to read any of those to follow this chapter. You should know where they are so that you understand what sits above the code you are about to write.

27.5.5 What This Means for Our Driver

Because VFS is not our direct caller, we do not need to implement `VOP_` methods. We need to implement the block-layer interface that the filesystem ultimately calls into. That interface is defined by GEOM and, for disk-like devices in particular, by the `g_disk` subsystem. Our driver will expose a GEOM provider. A filesystem will consume it. The flow of I/O will go through `struct bio` rather than through `struct uio`, and the unit of work will be a block rather than a byte range.

This is also why storage drivers rarely interact with `cdevsw` or `make_dev` directly the way character drivers do. The `/dev` node for a disk is created by GEOM, not by the driver. The driver describes itself to GEOM, and GEOM publishes a provider, which then appears in `/dev` with an automatically generated name.

27.5.6 The VFS Call Chain in Practice

Let us trace what happens when a user runs `cat /mnt/myfs/hello.txt`, assuming `/mnt/myfs` is mounted on our future block device.

First, the process calls `open("/mnt/myfs/hello.txt", O_RDONLY)`. That goes to `sys_openat` in the system call layer, which asks VFS to resolve the path. VFS walks the path one component at a time, calling `VOP_LOOKUP` on each directory vnode. When it reaches `myfs`, it notices that the vnode is a mount point and crosses into the mounted filesystem. It eventually arrives at the vnode for `hello.txt` and returns a file descriptor.

Second, the process calls `read(fd, buf, 64)`. That goes to `sys_read`, which calls `vn_read`, which calls `VOP_READ` on the `vnode`. The UFS implementation of `VOP_READ` consults its `inode`, figures out which disk blocks hold the requested bytes, and asks the buffer cache for those blocks. If the blocks are not cached, the buffer cache calls `bread`, which ultimately builds a `struct bio` and hands it to GEOM.

Third, GEOM looks at the provider the filesystem is consuming. Through a chain of providers and consumers, the `bio` ends up at the bottom provider, which is our driver's provider. Our strategy function receives the `bio`, reads the requested bytes from our backing store, and calls `biodone` or `g_io_deliver` to complete the request.

Fourth, the reply travels back the other way. The buffer cache gets its data, the filesystem returns to `vn_read`, `vn_read` copies the data into the user buffer, and `sys_read` returns.

None of that code is ours except the last hop. But understanding the whole chain is what lets you make sensible design choices when you write the last hop.

27.5.7 Wrapping Up Section 1

VFS is the layer that unifies filesystems in FreeBSD. It sits between the system call interface and the various concrete filesystems, and it provides the abstraction that makes files look identical regardless of where they live. Storage drivers do not live inside VFS. They live at the bottom of the stack, far below VFS, behind GEOM and the buffer cache. Our job in this chapter is to write a driver that participates correctly in that lower layer, and to understand enough about the upper layers to avoid confusion when diagnosing problems.

In the next section, we will sharpen the distinction between `devfs` and VFS, because that distinction determines which mental model applies when you think about a given device node.

27.6 Section 2: `devfs` vs VFS

Beginners often assume that `devfs` and the Virtual File System layer are two names for the same thing. They are not. They are related, but they play very different roles. Getting this distinction right early saves a great deal of confusion later, especially when thinking about storage drivers, because storage drivers straddle both of them.

27.6.1 What `devfs` Is

`devfs` is a filesystem. That sounds circular, but it is true. `devfs` is implemented as a filesystem module, registered with VFS, and mounted at `/dev` on every FreeBSD system. When you read a file under `/dev`, you are reading through VFS, which hands the request to `devfs`, which recognises that the “file” you are reading is really a kernel device node and routes the call to the appropriate driver.

`devfs` has several special properties that distinguish it from an ordinary filesystem like UFS.

First, its contents are not stored on disk. The “files” in `devfs` are synthesised by the kernel based on which drivers are currently loaded and which devices are currently present. When a driver calls `make_dev(9)` to create `/dev/mybox`, `devfs` adds the corresponding node to its view. When the driver destroys that device with `destroy_dev(9)`, `devfs` removes the node. The user sees `/dev/mybox` appear and disappear in real time.

Second, the read and write paths for `devfs` nodes are not file data paths. When you write to `/dev/myserial0`, you are not appending bytes to a stored file. You are invoking the driver's `d_write` function through `cdevsw`, and that function decides what those bytes mean. In the case of a USB serial driver, they mean bytes to transmit on the wire. In the case of a pseudo device like `/dev/null`, they mean bytes to discard.

Third, the metadata of `devfs` nodes, such as permissions and ownership, is managed by a policy layer in the kernel rather than by the filesystem itself. `devfs_ruleset(8)` and the `devd` framework configure that policy.

Fourth, `devfs` supports cloning, which character drivers like `pty`, `tun`, and `bpf` use to create a new minor device whenever a process opens the node. This is how `/dev/ptyp0`, `/dev/ptyp1`, and their successors come into existence on demand.

27.6.2 What VFS Is

VFS, as we saw in Section 1, is the abstract filesystem layer. Every filesystem on a FreeBSD system, including `devfs`, is registered with VFS and invoked through VFS. VFS is not a filesystem. It is the framework that filesystems plug into.

When you open a file on UFS, the chain is: system call -> VFS -> UFS -> buffer cache -> GEOM -> driver. When you open a node in `devfs`, the chain is: system call -> VFS -> `devfs` -> driver. Both go through VFS. Only the UFS chain involves GEOM.

27.6.3 Why Storage Drivers Live on Both Sides

This is where storage drivers become interesting.

A storage driver exposes a block device, and that block device eventually appears as a node under `/dev`. For example, if we register our driver and tell GEOM about it, a node called `/dev/myblk0` may appear in `devfs`. When a user writes `dd if=image.iso of=/dev/myblk0`, they are writing through `devfs` to a special character interface that GEOM provides on top of our disk. The requests flow as BIO through GEOM and into our strategy function.

But when a user runs `newfs_ufs /dev/myblk0` and then `mount /dev/myblk0 /mnt`, the usage pattern changes. The kernel now mounts UFS on top of the device. When a process later reads a file under `/mnt`, the path is: system call -> VFS -> UFS -> buffer cache -> GEOM -> driver. The `/dev/myblk0` node in `devfs` is not even involved in the hot path. UFS and the buffer cache talk directly to the GEOM provider. The `devfs` node is essentially a handle that tools use to refer to the device, not the pipe that file data flows through during normal operation.

27.6.4 A Closer Look at the Buffer Cache

Between the filesystem and GEOM in the storage path sits the buffer cache. We have mentioned it several times without pausing to describe it. Let us pause now, because it explains several of the behaviours you will observe when testing your driver.

The buffer cache is a pool of fixed-size buffers in kernel memory, each of which holds one filesystem block. When a filesystem reads a block, the buffer cache gets involved: the filesystem asks the cache for the block, and the cache either returns a hit (the block is already in memory) or issues a miss (the cache allocates a buffer,

calls down through GEOM to fetch the data, and returns the buffer once the read completes). When a filesystem writes a block, the same cache path applies in reverse: the write fills a buffer, the buffer is marked dirty, and the cache schedules a write-back at some later point.

The buffer cache is why consecutive reads of the same file data do not always hit the driver. The first read misses, causing a BIO to travel to the driver. The second read hits the cache and returns immediately. This is a great feature for performance. It can be mildly confusing when you are first debugging a driver, because your `printf` in the strategy function does not fire on every user-space read.

The buffer cache is also why writes can appear to happen faster than the underlying driver. A `dd if=/dev/zero of=/mnt/myblk/big bs=1m count=16` may appear to complete in a fraction of a second because the writes land in the cache and the cache defers the actual BIOs for a while. The filesystem issues the real writes to GEOM over the next second or two. If the system crashes before that happens, the file on disk is incomplete. `sync(2)` forces the cache to flush to the underlying device. `fsync(2)` flushes only the buffers associated with a single file descriptor.

The buffer cache is distinct from the page cache. FreeBSD has both, and they cooperate. The page cache holds memory pages that back memory-mapped files and anonymous memory. The buffer cache holds buffers that back filesystem-block operations. Modern FreeBSD has largely unified them for many data paths, but the distinction still shows up in the source tree, particularly around `bread`, `bwrite`, `getblk`, and `brelease`, which are the buffer-cache side of the interface.

The buffer cache has a single most important implication for our driver: we will almost never see completely synchronous BIO traffic. When a filesystem wants to read a block, a BIO arrives in our strategy function; when a filesystem wants to write a block, another BIO arrives, but usually some time later than the write system call that prompted it. BIOs also arrive in bursts when the cache flushes. This is normal, and your driver must not make assumptions about timing or ordering across BIOs other than what is strictly documented. Each BIO is an independent request.

27.6.5 The Read and Write Paths

Let us trace a concrete example through the whole chain.

When a user runs `cat /mnt/myblk/hello.txt`, the shell runs `cat`, which calls `open("/mnt/myblk/hello.txt", O_RDONLY)`. The `open` goes to `sys_openat`, which hands off to VFS. VFS calls `namei` to walk the path. For each path component, VFS calls `VOP_LOOKUP` on the current directory's vnode. When VFS reaches the `myblk` mount, it crosses into UFS, which walks the UFS directory structure to find `hello.txt`. UFS returns the vnode for that file, and VFS returns a file descriptor.

The user then calls `read(fd, buf, 64)`. `sys_read` calls `vn_read`, which calls `VOP_READ` on the vnode. UFS's `VOP_READ` consults the inode to find the block address of the requested bytes, then calls `bread` on the buffer cache to fetch the block. The buffer cache either returns a hit or issues a BIO.

If it is a cache miss, the buffer cache allocates a fresh buffer, builds a BIO that asks for the relevant block from the underlying GEOM provider, and hands it off. The BIO travels down through GEOM, through our strategy function, and back. When the BIO completes, the buffer cache unblocks the waiting `bread` call. UFS then copies the requested bytes from the buffer into the user's `buf`. `read` returns.

For writes, the chain is symmetric but the timing is different. UFS's `VOP_WRITE` calls `bread` or `getblk` to obtain the target buffer, copies the user's data into the buffer, marks the buffer dirty, and calls `bwrite` or `bawrite` to

schedule the write-back. The user's `write` call returns long before the BIO is issued to the driver. Later, the buffer cache's syncer thread picks up dirty buffers and issues `BIO_WRITE` requests to the driver.

The net effect is that our driver's strategy function sees a stream of BIOs that is related to, but not identical to, the stream of user-space reads and writes. The buffer cache mediates the two.

In other words, the same storage driver can be reached two different ways.

1. **Raw access through `/dev`:** a user-space program opens `/dev/myblk0` and issues `read(2)` or `write(2)` calls. Those calls go through `devfs` and the GEOM character interface, ending up in our strategy function.
2. **Filesystem access through `mount`:** the kernel mounts a filesystem on the device. File I/O flows through VFS, the filesystem, the buffer cache, and GEOM. `devfs` is not part of the hot path for those requests.

Both paths converge at the GEOM provider, which is why GEOM is the correct abstraction for storage drivers even though character drivers typically deal with `devfs` more directly.

27.6.6 Why This Distinction Matters

This matters for two reasons.

First, it clarifies why we will not use `make_dev` for our block driver. `make_dev` is the right call for character drivers that want to publish a `cdevsw` under `/dev`. It is the wrong call for a block device, because GEOM creates the `/dev` node for us as soon as we publish a provider. If you call `make_dev` in a storage driver, you typically end up with two `/dev` nodes competing for the same device, one of which is not connected to the GEOM topology, which leads to confusing behaviour.

Second, the distinction explains why the kernel has two sets of tools for inspecting device state. `devfs_ruleset(8)`, `devfs.rules`, and per-node permissions belong to `devfs`. `geom(8)`, `gstat(8)`, `diskinfo(8)`, and the GEOM class tree belong to GEOM. When you are diagnosing a permissions problem, you look at `devfs`. When you are diagnosing an I/O problem, you look at GEOM.

27.6.7 A Concrete Example: `/dev/null` and `/dev/ada0`

Compare two examples you already know.

`/dev/null` is a classic character device. It lives under `/dev` because `devfs` creates it. The driver is `null(4)`, and its source is in `/usr/src/sys/dev/null/null.c`. When you write to `/dev/null`, `devfs` routes the request through `cdevsw` to the `null` driver's write function, which simply discards the bytes. There is no GEOM, no buffer cache, no filesystem. It is a raw `devfs` character node.

`/dev/ada0` is a block device. It also lives under `/dev`. But the node is created by GEOM, not by a direct `make_dev` call in the `ada` driver. When you read raw bytes from `/dev/ada0`, those bytes flow through GEOM's character interface layer and arrive at the `ada` driver's strategy function. When you mount UFS on `/dev/ada0` and then read a file, the file data flows through VFS, UFS, the buffer cache, and GEOM, and ends up in the same strategy function, without passing through `devfs` for each request.

The node in `devfs` is the same. The usage pattern is different. The driver must handle both.

27.6.8 How We Will Proceed

We will not write a character driver in this chapter. We already wrote one in Chapter 26. Instead, we will write a driver that registers with GEOM as a disk, and we will let GEOM create the `/dev` node for us. The devfs integration will be automatic.

This is the dominant pattern for block drivers in FreeBSD 14.3. You can see it in `md(4)`, in `ata(4)`, in `nvme(4)`, and in almost every other storage driver. Each of them registers with GEOM, each of them receives bio requests, and each of them lets GEOM handle the `/dev` node.

27.6.9 Wrapping Up Section 2

devfs and VFS are distinct layers. devfs is a filesystem mounted at `/dev`, and VFS is the abstract framework that all filesystems plug into, including devfs. Storage drivers interact with both, but through GEOM, which takes care of creating the `/dev` node and of routing requests from both raw-access and filesystem-access paths. For this chapter, we will use GEOM as our entry point and let it manage devfs on our behalf.

In the next section, we will begin building the driver. We will start with the minimum needed to register a pseudo block device with GEOM, without yet implementing real I/O. Once that is in place, we will add the backing store, the bio handler, and everything else in later sections.

27.7 Section 3: Registering a Pseudo Block Device

In this section we will create a skeleton driver that registers a pseudo block device with the kernel. We will not yet implement read or write. We will not yet wire it up to a backing store. Our goal is more modest and more important: we want to understand exactly what it takes to make the kernel recognise our code as a storage driver, publish a `/dev` node for it, and let tools like `geom(8)` see it.

Once that is working, everything we add later will be purely incremental. The registration itself is the step that feels most mysterious, and it is the one the rest of the driver builds on.

27.7.1 The `g_disk` API

FreeBSD gives storage drivers a high-level registration API called `g_disk`. It lives in `/usr/src/sys/geom/-geom_disk.c` and `/usr/src/sys/geom/geom_disk.h`. The API wraps the lower-level GEOM class machinery and exposes a simpler interface that matches what disk drivers usually need.

Using `g_disk` saves us from implementing a full `g_class` by hand. With `g_disk`, we allocate a `struct disk`, fill in a handful of fields and callback pointers, and call `disk_create`. The API takes care of building the GEOM class, creating the `geom`, publishing the provider, wiring up the character interface, starting devstat accounting, and making our device visible to userland through `/dev`.

Not every storage driver uses `g_disk`. GEOM classes that do transformations on other providers, like `g_nop`, `g_mirror`, `g_stripe`, or `g_el`, are built directly on the lower-level `g_class` machinery because they are not disk-shaped. But for anything that looks like a disk, and certainly for a pseudo-disk like ours, `g_disk` is the right starting point.

You can see the public structure in `/usr/src/sys/geom/geom_disk.h`. The shape is roughly the following, abbreviated for clarity.

```
struct disk {
    struct g_geom    *d_geom;
    struct devstat   *d_devstat;

    const char      *d_name;
    u_int           d_unit;

    disk_open_t      *d_open;
    disk_close_t     *d_close;
    disk_strategy_t   *d_strategy;
    disk_ioctl_t      *d_ioctl;
    disk_getattr_t    *d_getattr;
    disk_gone_t       *d_gone;

    u_int            d_sectorsize;
    off_t            d_mediasize;
    u_int            d_fwsectors;
    u_int            d_fwheads;
    u_int            d_maxsize;
    u_int            d_flags;

    void             *d_drv1;

    /* other fields elided */
};
```

The fields break down into three groups.

Identification: `d_name` is a short string like "myblk" that names the disk class, and `d_unit` is a small integer that distinguishes multiple instances. Together they form the `/dev` node name. A driver with `d_name = "myblk"` and `d_unit = 0` publishes `/dev/myblk0`.

Callbacks: the `d_open`, `d_close`, `d_strategy`, `d_ioctl`, `d_getattr`, and `d_gone` pointers are the functions the kernel will call into our driver. Of these, only `d_strategy` is strictly required, because that is the function that handles actual I/O. The others are optional and we will discuss them as they become relevant.

Geometry: `d_sectorsize`, `d_mediasize`, `d_fwsectors`, `d_fwheads`, and `d_maxsize` describe the disk's physical and logical shape. `d_sectorsize` is the size of a sector in bytes, typically 512 or 4096. `d_mediasize` is the total size of the device in bytes. `d_fwsectors` and `d_fwheads` are advisory hints used by partitioning tools. `d_maxsize` is the largest single I/O the driver can accept, which GEOM will use to split large requests.

Driver state: `d_drv1` is a generic pointer for the driver to stash its own context. It is the closest equivalent to `device_get_softc(dev)` in the Newbus world.

27.7.2 A Minimum Skeleton

Let us now sketch a minimum skeleton. We will place this in `examples/part-06/ch27-storage-vfs/myfirst_blk.c`. This initial version does almost nothing useful. It registers a disk, returns success on every operation, and unregisters cleanly on unload. But it is enough to appear in `/dev`, to be visible in `geom disk list`, and to be probed by `newfs_ufs` or `fdisk` without the kernel crashing.

```
/*
 * myfirst_blk.c - a minimal pseudo block device driver.
 *
 * This driver registers a single pseudo disk called myblk0 with
 * the g_disk subsystem. It is intentionally not yet capable of
 * performing real I/O. Sections 4 and 5 of Chapter 27 will add
 * the BIO handler and the backing store.
 */

#include <sys/param.h>
#include <sys/system.h>
#include <sys/kernel.h>
#include <sys/module.h>
#include <sys/malloc.h>
#include <sys/lock.h>
#include <sys/mutex.h>
#include <sys/bio.h>

#include <geom/geom.h>
#include <geom/geom_disk.h>

#define MYBLK_NAME      "myblk"
#define MYBLK_SECTOR    512
#define MYBLK_MEDIASIZE (1024 * 1024) /* 1 MiB to start */

struct myblk_softc {
    struct disk    *disk;
    struct mtx     lock;
    u_int          unit;
};

static MALLOC_DEFINE(M_MYBLK, "myblk", "myfirst_blk driver state");

static struct myblk_softc *myblk_unit0;

static void
myblk_strategy(struct bio *bp)
{

```

```
/*
 * No real I/O yet. Mark every request as successful
 * but unimplemented so the caller does not hang.
 */
bp->bio_error = ENXIO;
bp->bio_flags |= BIO_ERROR;
bp->bio_resid = bp->bio_bcount;
biodone(bp);
}

static int
myblk_attach_unit(struct myblk_softc *sc)
{
    sc->disk = disk_alloc();
    sc->disk->d_name      = MYBLK_NAME;
    sc->disk->d_unit      = sc->unit;
    sc->disk->d_strategy   = myblk_strategy;
    sc->disk->d_sectorsize = MYBLK_SECTOR;
    sc->disk->d_mediasize  = MYBLK_MEDIASIZE;
    sc->disk->d_maxsize    = MAXPHYS;
    sc->disk->d_drv1       = sc;

    disk_create(sc->disk, DISK_VERSION);
    return (0);
}

static void
myblk_detach_unit(struct myblk_softc *sc)
{
    if (sc->disk != NULL) {
        disk_destroy(sc->disk);
        sc->disk = NULL;
    }
}

static int
myblk_loader(struct module *m, int what, void *arg)
{
    struct myblk_softc *sc;
    int error;
```

```

switch (what) {
case MOD_LOAD:
    sc = malloc(sizeof(*sc), M_MYBLK, M_WAITOK | M_ZERO);
    mtx_init(&sc->lock, "myblk lock", NULL, MTX_DEF);
    sc->unit = 0;
    error = myblk_attach_unit(sc);
    if (error != 0) {
        mtx_destroy(&sc->lock);
        free(sc, M_MYBLK);
        return (error);
    }
    myblk_unit0 = sc;
    printf("myblk: loaded, /dev/%s%u size=%jd bytes\n",
        MYBLK_NAME, sc->unit,
        (intmax_t)sc->disk->d mediasize);
    return (0);

case MOD_UNLOAD:
    sc = myblk_unit0;
    if (sc == NULL)
        return (0);
    myblk_detach_unit(sc);
    mtx_destroy(&sc->lock);
    free(sc, M_MYBLK);
    myblk_unit0 = NULL;
    printf("myblk: unloaded\n");
    return (0);

default:
    return (EOPNOTSUPP);
}

static moduledata_t myblk_mod = {
    "myblk",
    myblk_loader,
    NULL
};

DECLARE_MODULE(myblk, myblk_mod, SI_SUB_DRIVERS, SI_ORDER_MIDDLE);
MODULE_VERSION(myblk, 1);

```

Take a moment to read this through. Only a handful of moving pieces are visible, but each of them is doing real work.

The `myblk_softc` structure is the driver-local context. It holds a pointer to our `struct disk`, a mutex for future use, and the unit number. We allocate it on module load and free it on unload.

The `myblk_strategy` function is the callback that GEOM will invoke whenever a bio is directed at our device. In this first version, we simply fail every request with `ENXIO`. That is not polite, but it is correct as a placeholder: the kernel will not block waiting for us, and we will not pretend that I/O succeeded when it did not. In Section 5 we will replace this with a working handler.

The `myblk_attach_unit` function allocates a `struct disk`, fills in the identification, callbacks, and geometry fields, and publishes it with `disk_create`. The call to `disk_create` is what actually produces the `/dev` node and registers the disk in the GEOM topology.

The `myblk_detach_unit` function reverses that. `disk_destroy` asks GEOM to wither the provider, cancel any pending I/O, and remove the `/dev` node. We set `sc->disk` to `NULL` so that subsequent unload attempts do not try to free an already-freed structure, though in the load/unload path we follow that cannot happen.

The module loader is a standard `moduldata_t` boilerplate that you saw in Chapter 26. On `MOD_LOAD` it allocates the `_softc` and calls `myblk_attach_unit`. On `MOD_UNLOAD` it calls `myblk_detach_unit`, frees the `_softc`, and returns.

One line deserves special attention.

The call `disk_create(sc->disk, DISK_VERSION)` passes the current ABI version of the disk structure. `DISK_VERSION` is defined in `/usr/src/sys/geom/geom_disk.h` and increments every time the `g_disk` ABI changes incompatibly. If you compile a driver against the wrong tree, the kernel will refuse to register the disk and will print a diagnostic. This versioning is what allows the kernel to evolve without silently breaking out-of-tree drivers.

You may wonder why we do not use `MODULE_DEPEND` to declare a dependency on `g_disk`. The reason is that `g_disk` is not a loadable kernel module in the usual sense. It is a GEOM class declared in the kernel via `DECLARE_GEOM_CLASS(g_disk_class, g_disk)` in `/usr/src/sys/geom/geom_disk.c`, and it is always present whenever GEOM itself is compiled into the kernel. There is no separate `g_disk.ko` file you can unload or reload independently, and `MODULE_DEPEND(myblk, g_disk, ...)` would not resolve to a real module. The symbols we call (`disk_alloc`, `disk_create`, `disk_destroy`) come from the kernel itself.

27.7.3 The Makefile

The Makefile for this module is almost identical to the one from Chapter 26.

```
# Makefile for myfirst_blk.
#
# Companion file for Chapter 27 of
# "FreeBSD Device Drivers: From First Steps to Kernel Mastery".

KMOD      = myblk
SRCS      = myfirst_blk.c

# Where the kernel build machinery lives.
.include <bsd.kmod.mk>
```


Place this in the same directory as `myfirst_blk.c`. Running `make` will build `myblk.ko`. Running `make load` will load it if you have the kernel sources installed in the usual place. Running `make unload` will unload it.

27.7.4 Loading and Inspecting the Skeleton

Once the module is loaded, the kernel will have created a pseudo disk and a `/dev` node for it. Let us walk through what you should see.

```
# kldload ./myblk.ko
# dmesg | tail -n 1
myblk: loaded, /dev/myblk0 size=1048576 bytes
# ls -l /dev/myblk0
crw-r-----  1 root  operator  0x8b Apr 19 18:04 /dev/myblk0
# diskinfo -v /dev/myblk0
/dev/myblk0
      512          # sectorsize
    1048576        # mediasize in bytes (1.0M)
      2048         # mediasize in sectors
         0         # stripesize
         0         # stripeoffset
    myblk0        # Disk ident.
```

The `c` at the beginning of the permissions string tells us that GEOM has created a character device node, which is how FreeBSD exposes block-oriented devices under `/dev` in the modern kernel. The device major number, here `0x8b`, is assigned dynamically.

Now let us look at the GEOM topology.

```
# geom disk list myblk0
Geom name: myblk0
Providers:
1. Name: myblk0
   Mediasize: 1048576 (1.0M)
   Sectorsize: 512
   Mode: r0w0e0
   descr: (null)
   ident: (null)
   rotationrate: unknown
   fwsectors: 0
   fwheads: 0
```

Mode: `r0w0e0` means zero readers, zero writers, zero exclusive holders. Nobody is using the disk.

Now try something harmless.

```
# dd if=/dev/myblk0 of=/dev/null bs=512 count=1
dd: /dev/myblk0: Device not configured
0+0 records in
0+0 records out
0 bytes transferred in 0.000123 secs (0 bytes/sec)
```

The `Device not configured` error is the `ENXIO` we deliberately returned. Our strategy function ran, marked the BIO as failed, and `dd` faithfully reported the failure. This is the first real evidence that our driver is being reached by the kernel's block-layer code.

Try a read that expects success to fail loudly.

```
# newfs_ufs /dev/myblk0
newfs: /dev/myblk0: read-only
# newfs_ufs -N /dev/myblk0
/dev/myblk0: 1.0MB (2048 sectors) block size 32768, fragment size 4096
        using 4 cylinder groups of 0.31MB, 10 blks, 40 inodes.
super-block backups (for fsck_ffs -b #) at:
192, 832, 1472, 2112
```

The `-N` flag tells `newfs` to plan the filesystem layout without writing anything. We can see that it thinks of our device as a small disk with 2048 sectors of 512 bytes each. That matches the geometry we declared. It is not yet actually writing anything because our strategy function would still fail, but the planning works.

Finally, let us unload the module cleanly.

```
# kldunload myblk
# dmesg | tail -n 1
myblk: unloaded
# ls /dev/myblk0
ls: /dev/myblk0: No such file or directory
```

That is the complete life cycle of the skeleton.

27.7.5 Why the Failures Are Expected

At this stage, any user-space tool that actually tries to read or write data will fail. That is correct. Our strategy function does not yet know how to do anything, and we must not fake success. Faking success would lead to corruption the moment a filesystem tried to read back what it thought it had written.

The fact that the kernel and the tools gracefully handle our failure is evidence that the block layer is doing its job. A bio came down, the driver rejected it, the error propagated back up to user space, and no one crashed. That is the kind of behaviour we want.

27.7.6 How Things Fit Together

Before moving on, let us name the pieces so we can refer to them later without ambiguity.

Our **driver module** is `myblk.ko`. It is what the user loads with `kldload`.

Our **softc** is `struct myblk_softc`. It holds driver-local state. There is exactly one instance in this first version.

Our **disk** is a `struct disk` allocated by `disk_alloc` and registered with `disk_create`. The kernel owns its memory. We do not free it directly. We ask the kernel to free it by calling `disk_destroy`.

Our **geom** is the GEOM object the `g_disk` subsystem creates on our behalf. We do not see it directly in our code. It exists in the GEOM topology as the parent of our provider.

Our **provider** is the producer-facing face of our device. It is what other GEOM classes consume when they connect to us. GEOM automatically creates a character device node for our provider under `/dev`.

Our **consumer** is still empty. We have no one connected to us yet. Consumers are how GEOM classes that sit above us, like a partitioning layer or a filesystem's GEOM consumer, attach.

Our **/dev node** is `/dev/myblk0`. It is a live handle that user-space tools can use to issue raw I/O. When a filesystem is later mounted on the device, it will also refer to the device by this name, even though the hot I/O path will not pass through `devfs` for each request.

27.7.7 Wrapping Up Section 3

We built the smallest possible driver that participates in the FreeBSD storage stack. It registers a pseudo disk with the `g_disk` subsystem, publishes a `/dev` node through GEOM, accepts BIO requests, and declines them politely. It loads, it appears in `geom disk list`, and it unloads without leaks.

In the next section, we will look at GEOM more directly. We will understand what a provider really is, what a consumer really is, and how the class-based design lets transformations like partitioning, mirroring, encryption, and compression compose with our driver for free. That understanding will set us up for Section 5, where we replace the placeholder strategy function with one that actually serves reads and writes from a backing store.

27.8 Section 4: Exposing a GEOM-Backed Provider

The previous section let us register a disk with `g_disk` and take the word of the framework for what happens under the hood. That is a reasonable first step, and for many drivers it is all the involvement with GEOM that they ever need. But storage work rewards understanding the layer you are sitting on. When a filesystem mount fails, when `gstat` shows requests piling up, or when a `kldunload` blocks for longer than you expect, you will want to know the vocabulary of GEOM well enough to ask the right questions.

This section is a tour of GEOM from the storage-driver perspective. It is not an exhaustive reference. There are entire chapters in the FreeBSD Developer's Handbook devoted to GEOM, and we will not duplicate them. What we will do is describe the concepts and objects that matter to a driver author, and show how `g_disk` fits into that picture.

27.8.1 GEOM in One Page

GEOM is a storage framework. It sits between filesystems and the block drivers that talk to real hardware, and it composes by design. That composition is the whole point.

The idea is that a storage stack is built out of small transformations. One transformation presents a raw disk. Another transformation splits it into partitions. Another transformation mirrors two disks into one. Another transformation encrypts a partition. Another transformation compresses a filesystem. Each transformation is a small piece of code that takes in I/O requests from above, does something to them, and either returns a result directly or passes them along to the next layer below.

In GEOM's vocabulary, each transformation is a **class**. Each instance of a class is a **geom**. Each geom has some number of **providers**, which are its outputs, and some number of **consumers**, which are its inputs. Providers face upward toward the next layer. Consumers face downward toward the previous layer. A geom with no consumer is at the bottom of the stack: it must produce I/O on its own. A geom with no provider is at the top of the stack: it must terminate I/O and deliver it somewhere outside GEOM, typically into a filesystem or into a `devfs` character device.

Requests flow from providers to consumers through the stack. Replies flow back. The unit of I/O is a `struct bio`, which we will study in detail in Section 5.

27.8.2 A Concrete Example of Composition

Imagine you have a 1 TB SATA SSD. The kernel's `ada(4)` driver runs on the SATA controller and publishes a disk provider called `ada0`. That is a geom with no consumer at the bottom and one provider at the top.

You slice the SSD with `gpart`. The `PART` class creates a geom whose single consumer is attached to `ada0`, and which publishes multiple providers, one per partition: `ada0p1`, `ada0p2`, `ada0p3`, and so on.

You encrypt `ada0p2` with `geli`. The `ELI` class creates a geom whose single consumer is attached to `ada0p2`, and which publishes a single provider called `ada0p2.eli`.

You mount UFS on `ada0p2.eli`. UFS opens that provider, reads its superblock, and begins serving files.

When a process reads a file, the request travels from UFS, to `ada0p2.eli`, through the `geli` geom which decrypts the relevant blocks, to `ada0p2`, through the `PART` geom which offsets the block addresses, to `ada0`, where the `ada` driver talks to the SATA controller.

At no point does UFS know that its underlying storage is encrypted, partitioned, or even a physical disk. It just sees a provider. The layers below it can be as simple or as elaborate as the administrator chooses.

That composition is the reason GEOM exists. A single storage driver only needs to know how to be a reliable bottom-of-stack producer of I/O. Everything above it is reusable.

27.8.3 Providers and Consumers in Code

In the kernel, a provider is a `struct g_provider` and a consumer is a `struct g_consumer`. Both are defined in `/usr/src/sys/geom/geom.h`. As a disk driver author, you almost never allocate either directly. `g_disk` allocates a provider on your behalf when you call `disk_create`, and you never need a consumer, because a disk driver does not attach to anything underneath.

What you do need is a mental model of what they mean.

A provider is a named, seekable, block-addressable surface that something can read and write. It has a size, a sector size, a name, and some access counters. GEOM publishes providers in `/dev` via its character-device integration, so the administrator can refer to them by name.

A consumer is a channel from one geom into another geom's provider. The consumer is where the upper geom issues I/O requests, and it is where the upper geom registers access rights. When you mount UFS on `ada0p2.eli`, the mount operation causes a consumer to be attached inside UFS's GEOM hook, and that consumer acquires access rights on the `ada0p2.eli` provider.

27.8.4 Access Rights

Providers have three access counters: read (r), write (w), and exclusive (e). They are visible in `gstat` and `geom disk list` as `r0w0e0` or similar. Each number is incremented when a consumer asks for that kind of access and decremented when the consumer releases it.

An exclusive access is what `mount`, `newfs`, and similar administrative tools acquire when they need to be sure no other process is writing the device. An exclusive count of zero means no exclusive access is held. An exclusive count greater than zero means the provider is busy.

The access counts are not trivia. They are a real synchronisation tool. When you call `disk_destroy` to remove a disk, the kernel will refuse to destroy the provider if it still has open users, because destroying it under the feet of a mounted filesystem would be catastrophic. This is the same mechanism that makes `kldunload block` if the module is in use, but it operates at the GEOM layer, one level higher than the module subsystem.

You can watch the access counters change in real time.

```
# geom disk list myblk0 | grep Mode
Mode: r0w0e0
# dd if=/dev/myblk0 of=/dev/null bs=512 count=1 &
# geom disk list myblk0 | grep Mode
Mode: r1w0e0
```

When `dd` finishes, the mode returns to `r0w0e0`.

27.8.5 The BIO Object and Its Life Cycle

The unit of work in GEOM is the BIO, defined as `struct bio` in `/usr/src/sys/sys/bio.h`. A BIO represents one I/O request. It has a command (`bio_cmd`), an offset (`bio_offset`), a length (`bio_length`), a data pointer (`bio_data`), a byte count (`bio_bcount`), a residual (`bio_resid`), an error (`bio_error`), flags (`bio_flags`), and a number of other fields that we will meet as we need them.

The `bio_cmd` values tell the driver what kind of I/O is being requested. The most common values are `BIO_READ`, `BIO_WRITE`, `BIO_DELETE`, `BIO_GETATTR`, and `BIO_FLUSH`. `BIO_READ` and `BIO_WRITE` are what you expect. `BIO_DELETE` asks the driver to release the blocks in the range, the way `TRIM` does on SSDs or `mdconfig -d` does on a memory disk. `BIO_GETATTR` queries an attribute by name and is how GEOM layers discover partition

types, media labels, and other metadata. `BIO_FLUSH` asks the driver to commit outstanding writes to stable storage.

A BIO travels downward from one geom to the next via `g_io_request`. When it reaches the bottom of the stack, the driver's strategy function is called. When the driver is done, it completes the BIO by calling `biodone` or, at the GEOM class level, `g_io_deliver`. The completion call releases the BIO back up the stack.

`g_disk` drivers get a slightly simpler view because the `g_disk` infrastructure translates GEOM-level BIO handling into `biodone`-style completion. When you implement `d_strategy`, you receive a `struct bio` and you must eventually call `biodone(bp)` to complete it. You do not call `g_io_deliver` directly. The framework does.

27.8.6 The GEOM Topology Lock

GEOM has a global lock called the topology lock. It protects modifications to the tree of geoms, providers, and consumers. When a provider is created or destroyed, when a consumer is attached or detached, when access counts change, or when GEOM walks the tree to route a request, the topology lock is taken.

The topology lock is held across operations that can take time, which is unusual for kernel locks, so GEOM operates much of its real work asynchronously through a dedicated thread called the event queue. When you look at `g_class` definitions in the source tree, the `init`, `fini`, `access`, and similar methods are invoked in the context of the GEOM event thread, not in the context of the user process that triggered the operation.

For a driver using `g_disk`, this matters in one specific way. You should not hold your own driver lock across a call into GEOM-level functions, because GEOM may acquire the topology lock inside those functions, and nested locking in the wrong order leads to deadlock. `g_disk` is written carefully enough that you do not usually have to think about this as long as you follow the patterns we show. But the fact is worth knowing.

27.8.7 The GEOM Event Queue

GEOM processes many events on a single dedicated kernel thread called `g_event`. If you have the kernel running with debugging enabled, you can see it in `procstat -kk`. This thread picks up events placed on its queue and processes them one at a time. Typical events include creating a geom, destroying a geom, attaching a consumer, detaching a consumer, and retasting a provider.

A practical consequence is that some actions you take from your driver, such as `disk_destroy`, do not happen synchronously in the context of the calling thread. They get queued for the event thread, and the actual destruction happens a moment later. `disk_destroy` handles the waiting correctly so that by the time it returns, the disk is gone. But if you are chasing a subtle ordering bug, remembering that GEOM has its own thread can help.

27.8.8 How `g_disk` Wraps All of This

With that vocabulary in hand, we can now describe what `g_disk` does for us more precisely.

When we call `disk_alloc`, we receive a `struct disk` that is pre-initialised enough to be filled in. We set the name, unit, callbacks, and geometry, then call `disk_create`.

`disk_create` does the following for us, through the event queue:

1. creates a GEOM class if one does not already exist for this disk name,
2. creates a geom under that class,
3. creates a provider associated with the geom,
4. sets up devstat accounting so that `iostat` and `gstat` have data,
5. wires up GEOM's character-device interface so that `/dev/<name><unit>` appears,
6. arranges for BIO requests to flow into our `d_strategy` callback.

It also sets up a few optional behaviours. If we provide a `d_ioctl`, the kernel routes user-space `ioctl` calls on the `/dev` node through to our function. If we provide a `d_getattr`, GEOM routes `BIO_GETATTR` requests through to it. If we provide a `d_gone`, the kernel calls it if something outside our driver decides the disk is gone, such as a hotplug removal event.

On the teardown side, `disk_destroy` queues the removal, waits for all pending I/O to drain, releases the provider, destroys the geom, and frees the `struct disk`. We do not call `free` on the disk ourselves. The framework does that.

27.8.9 Where to Read Source

You now have enough vocabulary to benefit from reading the `g_disk` source directly. Open `/usr/src/sys/geom/geom_disk.c` and look for the following.

The function `disk_alloc` is early in the file. It is a simple allocator that returns a zeroed `struct disk`. Nothing dramatic.

The function `disk_create` is longer. Skim it and notice the event-based approach: most of the real work is queued rather than performed inline. Also notice the sanity checks on the disk's fields, which catch drivers that forget to set the sector size, the media size, or the strategy function.

The function `disk_destroy` is similarly event-queued. It guards the teardown with an access-count check, because destroying a disk that is still open would be a bug.

The function `g_disk_start` is the inner strategy function. It validates a BIO, updates devstat, and calls the driver's `d_strategy`.

Take a moment to look at the code. You do not need to understand every branch. You do need to recognise the overall shape: events for structural changes, inline work for I/O. That is the shape of most GEOM-based code.

27.8.10 Comparing `md(4)` and `g_zero`

Two real drivers make good reading as counterpoints to `g_disk`. The first is the `md(4)` driver, in `/usr/src/sys/dev/md/md.c`. This is a memory-disk driver that uses both `g_disk` and directly managed GEOM structures. It is the most thorough example of a storage driver in the tree, supporting multiple backing-store types, resizing, dumping, and many other features. It is a large file, but it is the closest relative of what we are building.

The second is `g_zero`, in `/usr/src/sys/geom/zero/g_zero.c`. This is a minimal GEOM class that reads always return zeroed memory and writes are discarded. It is roughly 145 lines and uses the lower-level `DECLARE_GEOM_CLASS` API directly rather than `g_disk`. It is a great counterpoint because it shows the GEOM

class mechanics without any of the disk-specific adornment. When you want to understand what `g_disk` hides, read `g_zero`.

27.8.11 Why Our Driver Uses `g_disk`

You might ask whether we should build our driver directly on the lower-level `g_class` API, the way `g_zero` does, to expose more of the machinery. We will not, for three reasons.

First, `g_disk` is the idiomatic choice for anything that looks like a disk, which our pseudo block device does. Reviewers of real FreeBSD driver patches would push back on a driver that used `g_class` directly when `g_disk` would do.

Second, `g_disk` gives us `devstat` integration, standard `ioctl`s, and `/dev` node management for free. Re-implementing those by hand would be a substantial distraction from the teaching goal of this chapter.

Third, the simpler the first working driver, the easier it is to reason about. We have plenty of code to write in the next few sections. We do not need to spend pages on class-level GEOM plumbing that `g_disk` already gets right.

That said, if you are curious, you should absolutely read `g_zero.c`. It is a small file and it reveals the mechanics that `g_disk` abstracts. The Wrapping Up for this section will point you to it one last time.

27.8.12 A Walk Through `g_class`

For readers who want a little more of the underlying machinery, let us walk through what a `g_class` structure looks like in code, without yet building one of our own.

The following is reproduced (slightly simplified) from `/usr/src/sys/geom/zero/g_zero.c`.

```
static struct g_class g_zero_class = {
    .name = G_ZERO_CLASS_NAME,
    .version = G_VERSION,
    .start = g_zero_start,
    .init = g_zero_init,
    .fini = g_zero_fini,
    .destroy_geom = g_zero_destroy_geom
};

DECLARE_GEOM_CLASS(g_zero_class, g_zero);
```

`.name` is the class name, used in `geom -t` output. `.version` must match `G_VERSION` for the running kernel; mismatched versions are rejected at load time. `.start` is the function called when a BIO arrives at a provider of this class. `.init` is called when the class is first instantiated, typically to create the initial geom and its provider. `.fini` is the teardown counterpart to `.init`. `.destroy_geom` is called when a specific geom under this class is being removed.

`DECLARE_GEOM_CLASS` is a macro that expands to a module declaration that loads this class into the kernel when the module is loaded. It hides the `moduledata_t`, the `SYSINIT`, and the `g_modevent` wiring behind a single line.

Our driver does not use `g_class` directly. `g_disk` does it for us, and the class it declares under the hood is the universal `DISK` class that all disk-shaped drivers share. But understanding the structure is useful because, if you ever write a transformation class (a GEOM-level encrypt, compress, or partition layer), you will define your own `g_class`.

27.8.13 The Life of a BIO, In Detail

We covered the BIO life cycle briefly earlier. Here it is in more detail, because every storage-driver bug touches this life cycle at some point.

A BIO originates somewhere above the driver. For our driver, the most common origins are:

1. **A filesystem's buffer-cache write-back.** UFS calls `bwrite` or `bawrite` on a buffer, which builds a BIO and hands it to GEOM through `g_io_request`.
2. **A filesystem's buffer-cache read.** UFS calls `bread`, which checks the cache and, on a miss, issues a BIO.
3. **A raw access through `/dev/myb1k0`.** A program calls `read(2)` or `write(2)` on the node. `devfs` and GEOM's character-device integration build a BIO and issue it.
4. **A tool-issued operation.** `newfs_ufs`, `diskinfo`, `dd`, and similar tools issue BIOs the same way as a raw access.

Once built, the BIO is routed through GEOM's topology. Each consumer -> provider hop along the way may transform or validate the BIO. For a simple stack (our driver with no intermediate geoms), there are no intermediate hops; the BIO arrives at our provider and is dispatched to our strategy function.

Inside `g_disk`, the strategy function is preceded by three small pieces of bookkeeping:

1. Some sanity checks (for instance, verifying that the BIO's offset and length are within the media).
2. A call to `devstat_start_transaction_bio` to start timing the request.
3. A call to the driver's `d_strategy`.

On completion, `g_disk` intercepts the `biodone` call, records the end time with `devstat_end_transaction_bio`, and forwards the completion up the stack.

From the driver's point of view, the only thing that matters is that `d_strategy` gets called, and that `biodone` is called once per BIO. Everything else is plumbing.

27.8.14 Error Propagation

When a BIO fails, the driver sets `bio_error` to an `errno` value and sets the `BIO_ERROR` flag in `bio_flags`. `biodone` is then called as normal.

Above the driver, GEOM's completion code checks for the error. If set, the error is propagated up the stack. The filesystem sees the error and decides what to do; typically, a read error on metadata is fatal and the filesystem reports EIO to user space. A write error is often delayed; the filesystem may retry, or may mark the associated buffer as needing attention on the next sync.

Common `errno` values in the BIO path:

- `EIO`: a generic I/O error. The kernel assumes the device is having trouble.

- `ENXIO`: the device is not configured or has gone away.
- `EOPNOTSUPP`: the driver does not support this operation.
- `EROFS`: the medium is read-only.
- `ENOSPC`: no space available.
- `EFAULT`: an address in the request is invalid. Very rare in the BIO path.

For our in-memory driver, the only errors that should ever appear are the bounds-check error (`EIO`) and the unknown-command error (`EOPNOTSUPP`).

27.8.15 What `g_disk` Does That You Do Not See

We have mentioned that `g_disk` takes care of several things on our behalf. Here is a fuller list.

- It creates the GEOM class for the `DISK` type if it does not already exist, and it shares this class among all disk drivers.
- It creates a geom under that class when we call `disk_create`.
- It creates a provider on the geom and publishes it in `/dev`.
- It wires up `devstat` accounting automatically.
- It handles the GEOM access protocol, converting user-space `open` and `close` calls on `/dev/myblk0` into provider access-count changes.
- It handles the GEOM character-device interface, converting read and write on `/dev/myblk0` into BIOs to our strategy function.
- It handles the `BIO_GETATTR` default cases (most attributes have sensible defaults).
- It handles withering on `disk_destroy`, waiting for in-flight BIOs.
- It forwards `d_ioctl` calls for ioctls it does not handle itself.

Each of these is a piece of code you would have to write if you built directly on `g_class`. Reading `/usr/src/sys/geom/geom_disk.c` is a good way to appreciate just how much `g_disk` does for us.

27.8.16 Inspecting Our Provider

Let us take our skeleton driver from Section 3, load it, and inspect it through GEOM's eyes.

```
# kldload ./myblk.ko
# geom disk list myblk0
Geom name: myblk0
Providers:
1. Name: myblk0
   Mediasize: 1048576 (1.0M)
   Sectorsize: 512
   Mode: r0w0e0
   descr: (null)
   ident: (null)
   rotationrate: unknown
   fwsectors: 0
   fwheads: 0
```

`geom disk list` shows us only the DISK class's geoms. Each of those geoms has one provider. We can also see the full class tree.

```
# geom -t | head -n 40
Geom      Class      Provider
ada0       DISK        ada0
  ada0p1    PART        ada0p1
  ada0p2    PART        ada0p2
  ada0p3    PART        ada0p3
myblk0     DISK        myblk0
```

Our geom is a sibling of the real disks, without any upper-layer class attached to it yet. In later sections we will see what happens when a filesystem attaches.

```
# geom stats myblk0
```

`geom stats` returns detailed performance counters. On an idle, unused device like ours, all the counters are zero.

```
# gstat -I 1
dT: 1.002s  w: 1.000s
 L(q)  ops/s    r/s    kBps    ms/r    w/s    kBps    ms/w    %busy Name
    0      0      0      0      0.0    0      0      0.0    0.0| ada0
    0      0      0      0      0.0    0      0      0.0    0.0| myblk0
```

`gstat` is a more compact view that updates live. We will use this heavily in later sections.

27.8.17 Wrapping Up Section 4

GEOM is a composable block-layer framework made of classes, geoms, providers, and consumers. Requests flow through it as `struct bio` objects, with `BIO_READ`, `BIO_WRITE`, and a handful of other commands. Access rights, topology locking, and event-driven structure management are the mechanisms that keep the framework safe to evolve under load. `g_disk` wraps all of this for disk-shaped drivers and gives them a friendlier interface with little loss of expressiveness.

Our skeleton driver is now a first-class GEOM participant, even though it cannot yet do any real I/O. In the next section, we will give it that missing piece. We will allocate a backing buffer, implement a strategy function that actually reads and writes, and watch the kernel's storage stack exercise our code from both raw-access and filesystem-access directions.

27.9 Section 5: Implementing Basic Read and Write

In Section 3 we returned `ENXIO` for every BIO. In Section 4 we learned enough about GEOM to know exactly what kind of request our strategy function receives and what its obligations are. In this section we will replace

that placeholder with a working handler that reads and writes real bytes against an in-memory backing store. By the end, our driver will serve traffic through `dd`, will return sane data, and will survive being formatted by `newfs_ufs`.

27.9.1 The Backing Store

Our backing store for now is simply an array of bytes in kernel memory, sized to match `d mediasize`. It is the simplest possible representation of a disk: a flat buffer. Real storage drivers replace this with hardware DMA, with a vnode-backed file, or with a swap-backed VM object, but a flat buffer is enough to teach every other concept in this chapter without distraction.

For 1 MiB we can simply `malloc` the buffer. For larger sizes we would need a different allocator, because the kernel heap does not scale gracefully to contiguous allocations of tens or hundreds of megabytes. `md(4)` avoids the issue for large memory disks by using page-at-a-time allocation and a custom indirection structure. We do not need that level of sophistication yet, but we will note the limitation in the code.

Let us update `myblk_softc` to include the backing store.

```
struct myblk_softc {
    struct disk      *disk;
    struct mtx       lock;
    u_int            unit;
    uint8_t          *backing;
    size_t           backing_size;
};
```

Two new fields: `backing` is the pointer to the kernel memory we allocated, and `backing_size` is the number of bytes we allocated. These should always be equal to `d mediasize`, but storing the size explicitly is cleaner than relying on indirection through `disk->d mediasize`.

Now, in `myblk_attach_unit`, allocate the backing buffer.

```
static int
myblk_attach_unit(struct myblk_softc *sc)
{
    sc->backing_size = MYBLK_MEDIASIZE;
    sc->backing = malloc(sc->backing_size, M_MYBLK, M_WAITOK | M_ZERO);

    sc->disk = disk_alloc();
    sc->disk->d_name      = MYBLK_NAME;
    sc->disk->d_unit      = sc->unit;
    sc->disk->d_strategy   = myblk_strategy;
    sc->disk->d_sectorsize = MYBLK_SECTOR;
    sc->disk->d_mediasize  = MYBLK_MEDIASIZE;
    sc->disk->d_maxsize    = MAXPHYS;
}
```

```

    sc->disk->d_drv1      = sc;

    disk_create(sc->disk, DISK_VERSION);
    return (0);
}

```

malloc with `M_WAITOK | M_ZERO` returns a zeroed buffer or sleeps until one is available. It cannot fail for small allocations on a healthy system, which is why we do not check the return value here. If we were allocating a very large buffer we might want `M_NOWAIT` and explicit error handling, but for 1 MiB `M_WAITOK` is the idiomatic choice.

`myblk_detach_unit` must free the backing store after destroying the disk.

```

static void
myblk_detach_unit(struct myblk_softc *sc)
{
    if (sc->disk != NULL) {
        disk_destroy(sc->disk);
        sc->disk = NULL;
    }
    if (sc->backing != NULL) {
        free(sc->backing, M_MYBLK);
        sc->backing = NULL;
        sc->backing_size = 0;
    }
}

```

Order matters here. We destroy the disk first, which ensures there are no more BIOs in flight. Only then do we free the backing buffer. If we freed the buffer first, an in-flight BIO might try to memcpy into or out of a pointer that no longer refers to our memory, and the kernel would crash on the next I/O.

27.9.2 The Strategy Function

Now for the heart of the change. Replace the placeholder `myblk_strategy` with a function that actually services BIOs.

```

static void
myblk_strategy(struct bio *bp)
{
    struct myblk_softc *sc;
    off_t offset;
    size_t len;
}

```

```
sc = bp->bio_disk->d_drv1;
offset = bp->bio_offset;
len = bp->bio_bcount;

if (offset < 0 ||
    offset > sc->backing_size ||
    len > sc->backing_size - offset) {
    bp->bio_error = EIO;
    bp->bio_flags |= BIO_ERROR;
    bp->bio_resid = len;
    biodone(bp);
    return;
}

switch (bp->bio_cmd) {
case BIO_READ:
    mtx_lock(&sc->lock);
    memcpy(bp->bio_data, sc->backing + offset, len);
    mtx_unlock(&sc->lock);
    bp->bio_resid = 0;
    break;

case BIO_WRITE:
    mtx_lock(&sc->lock);
    memcpy(sc->backing + offset, bp->bio_data, len);
    mtx_unlock(&sc->lock);
    bp->bio_resid = 0;
    break;

case BIO_DELETE:
    mtx_lock(&sc->lock);
    memset(sc->backing + offset, 0, len);
    mtx_unlock(&sc->lock);
    bp->bio_resid = 0;
    break;

case BIO_FLUSH:
    /*
     * In-memory backing store is always "flushed".
     * Nothing to do.
     */
    bp->bio_resid = 0;
    break;
```

```
default:
    bp->bio_error = EOPNOTSUPP;
    bp->bio_flags |= BIO_ERROR;
    bp->bio_resid = len;
    break;
}

biodone(bp);
}
```

Let us read this carefully. It is not a long function, but every line is doing something that matters.

The first line finds our softc. GEOM gives us the BIO with a pointer to the disk in `bp->bio_disk`. We stashed our softc in `d_drv1` during `disk_create`, so we retrieve it from there. This is the block-driver equivalent of `device_get_softc(dev)` in the Newbus world.

The second pair of lines extract the offset and length of the request. `bio_offset` is a byte offset into the media. `bio_bcount` is the number of bytes to transfer. GEOM has already translated file-level operations through whatever layers sit above us into a linear byte range.

The bounds check that follows is defensive programming. GEOM will not normally send us a request that exceeds the media size, because it splits and validates BIOs on our behalf. But defensive drivers check anyway, because a silently accepted out-of-bounds write can smash kernel memory, and because the cost of the check is a few instructions per request. We also guard against arithmetic overflow by rewriting the obvious `offset + len > backing_size` check as `len > backing_size - offset`, which cannot overflow because `offset <= backing_size` at this point.

The switch is where the real work happens. Each BIO command gets its own case.

`BIO_READ` copies `len` bytes from our backing store at `offset` into `bp->bio_data`. GEOM has allocated `bp->bio_data` for us, and it will be released when the BIO completes. Our job is just to fill it.

`BIO_WRITE` copies `len` bytes from `bp->bio_data` into our backing store at `offset`. Symmetrical to the read case.

`BIO_DELETE` zeroes the range. For a real disk, `BIO_DELETE` is how filesystems signal that a range of blocks is no longer in use, and the disk is free to reclaim it. SSDs use it to drive TRIM. For our in-memory driver, there is nothing to reclaim, but zeroing the range is a reasonable response because it reflects the “data is gone” semantics.

`BIO_FLUSH` is a request to commit outstanding writes to stable storage. Our storage is never volatile in the sense that a FLUSH would help: every memcopy is already visible to the next memcopy in the same order it was issued. We return success with nothing to do.

Any other command we do not recognise gets `EOPNOTSUPP`. GEOM layers above us will see this and react accordingly.

At the end, `biodone(bp)` completes the BIO. This is not optional. Every BIO that enters the strategy function must leave through `biodone` exactly once, or the BIO will be leaked, the caller will block forever, and you will have a difficult time diagnosing the issue.

27.9.3 The Role of `bio_resid`

Notice the handling of `bp->bio_resid`. This field represents the number of bytes remaining to transfer after the driver is done. When the full transfer succeeds, `bio_resid` is zero. When the transfer fails completely, `bio_resid` equals `bio_bcount`. When the transfer partially succeeds, `bio_resid` is the number of bytes that did not make it.

Our driver either transfers everything or nothing, so we set `bio_resid` to either 0 (success) or `len` (error). A real hardware driver might set it to an intermediate value if a transfer stopped partway through. Filesystems and user-space tools use `bio_resid` to figure out how much data actually moved.

27.9.4 The Lock

We take `sc->lock` around the `memcpy`. For an in-memory driver that services one request at a time, the lock is not doing much visible work: the kernel's BIO scheduling makes truly concurrent requests unlikely on our toy device. But the lock is good hygiene. GEOM does not promise that your strategy function will be invoked serially, and even if it did, a future change to the driver to add an asynchronous worker thread would require the lock anyway. Adding it now is cheaper than adding it later.

A more sophisticated driver might use a fine-grained lock, or might use an MPSAFE approach that relies on atomic operations. For now, a coarse mutex around the `memcpy` is fine. It is correct, it is easy to reason about, and it does not hurt performance on a pseudo device.

27.9.5 Rebuilding and Reloading

After updating the source and `kldunload`-ing the old version, rebuild and reload.

```
# make
cc -O2 -pipe -fno-strict-aliasing ...
# kldunload myblk
# kldload ./myblk.ko
# dmesg | tail -n 1
myblk: loaded, /dev/myblk0 size=1048576 bytes
```

Now let us try some real I/O.

```
# dd if=/dev/zero of=/dev/myblk0 bs=4096 count=16
16+0 records in
16+0 records out
65536 bytes transferred in 0.001104 secs (59 MB/sec)
# dd if=/dev/myblk0 of=/dev/null bs=4096 count=16
16+0 records in
16+0 records out
65536 bytes transferred in 0.000512 secs (128 MB/sec)
```


We wrote 64 KiB of zeros and read them back. The speeds you see will depend on your hardware and on how much the buffer cache helps, but any speed above a few MB/sec is fine for a first run.

```
# dd if=/dev/random of=/dev/myblk0 bs=4096 count=16
16+0 records in
16+0 records out
65536 bytes transferred in 0.001233 secs (53 MB/sec)
# dd if=/dev/myblk0 of=pattern.bin bs=4096 count=16
16+0 records in
16+0 records out
# dd if=/dev/myblk0 of=pattern2.bin bs=4096 count=16
16+0 records in
16+0 records out
# cmp pattern.bin pattern2.bin
#
```

We wrote random data, read it back twice, and confirmed that both reads return the same content. Our driver is now a coherent store.

27.9.6 A Quick Look Under Load

Let us run a short stress test and watch `gstat`.

In one terminal:

```
# while true; do dd if=/dev/urandom of=/dev/myblk0 bs=4096 \
    count=256 2>/dev/null; done
```

In another terminal:

```
# gstat -I 1 -f myblk0
dT: 1.002s w: 1.000s
L(q) ops/s r/s kBps ms/r w/s kBps ms/w %busy Name
    0   251    0    0  0.0  251  1004  0.0  2.0| myblk0
```

About 250 write operations per second at 4 KiB each, approximately 1 MB/sec. The latency is very low because the backing store is RAM. For a real disk the numbers would be very different, but the structure of what you are watching is the same.

Stop the stress test with `Ctrl-C` on the first terminal.

27.9.7 Refining the Driver with `ioctl` Support

Many storage tools send an `ioctl` to the device to query geometry or to issue commands. GEOM handles the common ones for us, but if we provide a `d_ioctl` callback, the kernel will route unknown `ioctls` through to our function. For now we do not implement any custom `ioctl`. We only note that the hook exists.

```
static int
myblk_ioctl(struct disk *d, u_long cmd, void *data, int flag,
            struct thread *td)
{
    (void)d; (void)data; (void)flag; (void)td;

    switch (cmd) {
        /* No custom ioctls yet. */
        default:
            return (ENOIOCTL);
    }
}
```

We register the callback by assigning `sc->disk->d_ioctl = myblk_ioctl`; before calling `disk_create`. Returning `ENOIOCTL` from the default case tells GEOM that we do not handle the command and gives it the chance to pass the request to its own default handler.

27.9.8 Refining the Driver with getattr Support

GEOM uses `BIO_GETATTR` to ask storage devices for named attributes. A filesystem might ask for `GEOM::rotation_rate` to know whether it is on spinning media. The partitioning layer might ask for `GEOM::ident` to get a stable identifier. A `d_getattr` callback is the hook that lets us respond.

```
static int
myblk_getattr(struct bio *bp)
{
    struct myblk_softc *sc;

    sc = bp->bio_disk->d_drv1;

    if (strcmp(bp->bio_attribute, "GEOM::ident") == 0) {
        if (bp->bio_length < sizeof("MYBLK0"))
            return (EFAULT);
        strncpy(bp->bio_data, "MYBLK0", bp->bio_length);
        bp->bio_completed = strlen("MYBLK0") + 1;
        return (0);
    }

    /* Let g_disk fall back to default behaviour. */
    (void)sc;
    return (-1);
}
```

The return-value convention for `d_getattr` is worth pausing on, because it trips up many first-time readers. Returning 0 with `bio_completed` set tells `g_disk` that we handled the attribute successfully. Returning a positive `errno` value (such as `EFAULT` for a too-small buffer) tells `g_disk` that we handled the attribute but the operation failed. Returning -1 tells `g_disk` that we did not recognise the attribute and it should try its built-in default handler. That is why we return -1 at the bottom: we want `g_disk` to answer standard attributes such as `GEOM::fwsectors` on our behalf. For our driver, responding to `GEOM::ident` with a short string is enough to show up in `diskinfo -v`. Register this with `sc->disk->d_getattr = myblk_getattr`; before `disk_create`.

27.9.9 Partial Writes and Short Reads

Our driver does not actually produce partial writes or short reads, because the backing store is in RAM and every transfer either fully succeeds or fully fails. But for a real hardware driver, partial transfers are normal: a disk may return a few sectors successfully and then fail on a bad sector. The BIO framework supports this through `bio_resid`, and a driver should set `bio_resid` to the number of bytes that did not complete.

The practical guidance is to always set `bio_resid` explicitly before calling `biodone`. If the transfer fully succeeded, set it to zero. If it partially succeeded, set it to the residual. If it fully failed, set it to `bio_bcount`. Forgetting to set `bio_resid` leaves whatever garbage was in the field when the BIO was allocated, which can confuse callers.

27.9.10 Common Mistakes in Strategy Functions

Before we continue, let us name three common mistakes that appear in first-time strategy functions.

Forgetting `biodone`. Every path out of the strategy function must call `biodone(bp)` on the BIO. If you forget, the BIO is leaked and the caller hangs. This is the single most common source of “my mount hangs” problems.

Holding a lock across `biodone`. `biodone` may call upward into GEOM or into a filesystem’s completion handler. Those handlers may take other locks, or may need to acquire locks you already hold, leading to lock-order reversal and potential deadlock. The safest pattern is to drop your lock before calling `biodone`. Our simple version does this implicitly: the `mtx_unlock` is always inside the switch, and `biodone` runs after the switch.

Returning from the strategy function with an error code. `d_strategy` is a void function. Errors are reported by setting `bio_error` and the `BIO_ERROR` flag on the BIO, not by returning. Compilers catch this if you declare the function correctly, but beginners sometimes write it as returning `int`, which causes compiler warnings that should not be ignored.

27.9.11 Chained BIOs and BIO Hierarchies

A BIO can have a child. GEOM uses this when a transformation class needs to split, combine, or transform a request into one or more downstream requests. For example, a mirror class might take a `BIO_WRITE` and issue two child BIOs, one to each mirror member. A partition class might take a `BIO_READ` and issue a single child BIO with the offset shifted into the underlying provider’s address space.

The parent-child relationship is recorded in `bio_parent`. When a child completes, its error is propagated to the parent by `biodone`, which accumulates errors and delivers the parent when all children have completed.

Our driver does not produce child BIOs. It receives them as leaves of the chain. From the driver's perspective, every BIO is self-contained: it has an offset, a length, and a data buffer, and our job is to service it.

But if you ever find yourself needing to split a BIO inside your driver (for example, if a request spans a boundary that your backing store handles in separate chunks), you can use `g_clone_bio` to create a child BIO, `g_io_request` to dispatch it, and `g_std_done` or a custom completion handler to reassemble the parent. The pattern is visible in several places in the kernel, including `g_mirror` and `g_raid`.

27.9.12 The Thread Context of the Strategy Function

The strategy function runs in whatever thread submitted the BIO. For filesystem-originated BIOs, that is typically the filesystem's syncer thread or a buffer-cache worker. For direct user-space access, it is the user thread that called `read` or `write` on `/dev/myblk0`. For GEOM transformations, it might be the GEOM event thread or a class-specific worker thread.

What this means for your driver is that `d_strategy` can run in many different thread contexts. You cannot assume that `curthread` belongs to any particular process, and you cannot block for a long time or the calling filesystem (or user program) will stall.

If your strategy function needs to do something slow (I/O against a vnode, waiting for hardware, or complex locking), the right pattern is to enqueue the BIO on an internal queue and have a dedicated worker thread process it. This is what `md(4)` does for all backing types, because vnode I/O (for instance) can block arbitrarily long.

Our driver is entirely in-memory and does `memcpy` only, so we do not need a worker thread. But understanding the pattern is important for the future.

27.9.13 A Worked Example: Reading Across a Boundary

Suppose a filesystem issues a `BIO_READ` with offset 100000 and length 8192. That spans bytes 100000 through 108191. Let us trace how our strategy function handles it.

1. `bp->bio_cmd` is `BIO_READ`.
2. `bp->bio_offset` is 100000.
3. `bp->bio_bcount` is 8192.
4. `bp->bio_data` points to a kernel buffer (or a user buffer mapped into the kernel) where the 8192 bytes should go.

Our code computes `offset = 100000` and `len = 8192`. The bounds check passes: $100000 + 8192 = 108192$, which is less than our `backing_size` of 32 MiB (33554432).

The switch enters the `BIO_READ` case. We acquire the lock, `memcpy` 8192 bytes from `sc->backing + 100000` into `bp->bio_data`, and release the lock. We set `bp->bio_resid = 0` to indicate a complete transfer. We fall through to `biodone(bp)`, which completes the BIO.

The filesystem receives the completion, notices the error is zero, and uses the 8192 bytes. The read is complete.

Now suppose, instead, the offset was 33554431 and the length was 2 bytes. That is one byte inside the backing store and one byte past the end.

1. `offset = 33554431.`
2. `len = 2.`

The bounds check: `offset > sc->backing_size` evaluates `33554431 > 33554432`, which is false. `len > sc->backing_size - offset` evaluates `2 > 33554432 - 33554431`, which evaluates to `2 > 1`, which is true. The check fails, and we fall into the error path: set `bio_error = EIO`, set the `BIO_ERROR` flag, set `bio_resid = 2`, and call `biodone`. The filesystem sees the error and handles it.

Notice how we used subtraction to avoid the overflow risk. Had we written `offset + len > sc->backing_size`, and had `offset` and `len` both been close to the maximum of `off_t`, the addition could wrap around to a small number and the check would silently pass for a malformed request. Defensive bounds checks always rearrange arithmetic to avoid overflow.

27.9.14 The Devstat Side Effect

One pleasant feature of using `g_disk` is that devstat accounting is automatic. Every BIO we service is counted by `iostat` and `gstat`. No extra code is needed.

You can verify this with `iostat -x 1` in another terminal while running the stress loop.

extended device statistics										
device	r/s	w/s	kr/s	kw/s	ms/r	ms/w	ms/o	ms/t	qlen	%b
ada0	0	2	0	48	0.0	0.1	0.0	0.1	0	0
myblk0	0	251	0	1004	0.0	0.0	0.0	0.0	0	2

If our driver were built on the raw `g_class` API rather than `g_disk`, we would have to wire up devstat ourselves. This is one of the small quality-of-life features that `g_disk` gives us for free.

27.9.15 Wrapping Up Section 5

We replaced the placeholder strategy function with a working handler. Our driver now services `BIO_READ`, `BIO_WRITE`, `BIO_DELETE`, and `BIO_FLUSH` correctly against an in-memory backing store. It participates in devstat, it cooperates with `gstat`, and it accepts real traffic from `dd`.

In the next section, we will cross the boundary from raw block access to filesystem access. We will format the device with `newfs_ufs`, mount it, create files on it, and observe how the request path changes when a real filesystem sits above the provider.

27.10 Section 6: Mounting a Filesystem on the Device

Up to this point, our driver has been exercised through raw access: `dd`, `diskinfo`, and similar tools reading and writing the entire surface as a flat byte range. That is a valuable mode, but it is not the mode most storage devices live in. Storage devices in real life serve filesystems. This section takes our driver the last mile: we will format it, mount a real filesystem on it, create files, and observe how the kernel's block-layer plumbing routes requests when a filesystem is in the picture.

This is also the first section where the theoretical distinction between raw access and filesystem access becomes concrete. Understanding the difference, and being able to see it in action, is one of the most useful insights a storage-driver author can acquire.

27.10.1 The Plan

We will do the following in this section, in order.

1. Increase the media size of our driver from 1 MiB to something large enough to hold a usable UFS filesystem.
2. Build and load the updated driver.
3. Run `newfs_ufs` against the device to create a filesystem.
4. Mount the filesystem on a scratch directory.
5. Create some files and verify that the data is read back correctly.
6. Unmount the filesystem.
7. Reload the module and watch what happens.

By the end, you will have seen a complete filesystem on top of your own block driver.

27.10.2 Increasing the Media Size

UFS has a minimum practical size. You can create tiny UFS filesystems, but the overhead of the superblock, cylinder groups, and inode tables takes a noticeable fraction of the space on anything smaller than a few megabytes. For our purposes, 32 MiB is a comfortable size: it is small enough that the backing store still fits in a plain `malloc`, and large enough that UFS has room to breathe.

Update the size definitions at the top of `myfirst_blk.c`.

```
#define MYBLK_SECTOR      512
#define MYBLK_MEDIASIZE  (32 * 1024 * 1024) /* 32 MiB */
```

Rebuild.

```
# make clean
# make
# kldunload myblk
# kldload ./myblk.ko
# diskinfo -v /dev/myblk0
/dev/myblk0
    512          # sectorsize
 33554432       # mediasize in bytes (32M)
   65536        # mediasize in sectors
    0           # stripesize
    0           # stripeoffset
```

32 MiB is enough.

27.10.3 Formatting with newfs_ufs

`newfs_ufs` is the standard UFS formatter on FreeBSD. It lays down the superblock, the cylinder groups, the root inode, and all the other structures a UFS filesystem requires. Let us run it on our device.

```
# newfs_ufs /dev/myblk0
/dev/myblk0: 32.0MB (65536 sectors) block size 32768, fragment size 4096
    using 4 cylinder groups of 8.00MB, 256 blks, 1280 inodes.
super-block backups (for fsck_ffs -b #) at:
192, 16576, 32960, 49344
```

A few things happened under the hood.

`newfs_ufs` opened `/dev/myblk0` for writing, which caused the GEOM access count to tick up. Our strategy function then received a stream of writes: the superblock first, then the cylinder groups, then the empty root directory, then the several backup superblocks. Each of those writes is a BIO, and each BIO was handled by our driver.

You can verify that `newfs_ufs` really wrote to the device by reading a few bytes back.

```
# dd if=/dev/myblk0 bs=1 count=16 2>/dev/null | hexdump -C
00000000  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
```

The first few bytes of a UFS partition are deliberately zero because the superblock does not sit at offset zero: it is at offset 65536 (block 128) to leave room for boot blocks and other preambles. Let us peek there.

```
# dd if=/dev/myblk0 bs=512 count=2 skip=128 2>/dev/null | hexdump -C | head
00010000  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00010010  80 00 00 00 80 00 00 00  a0 00 00 00 00 00 00 00
...
```

You should see non-zero bytes now. That is the superblock that `newfs_ufs` laid down on our backing store.

27.10.4 Mounting the Filesystem

Create a mount point and mount the filesystem.

```
# mkdir -p /mnt/myblk
# mount /dev/myblk0 /mnt/myblk
# mount | grep myblk
/dev/myblk0 on /mnt/myblk (ufs, local)
# df -h /mnt/myblk
```

Filesystem	Size	Used	Avail	Capacity	Mounted on
/dev/myblk0	31M	8.0K	28M	0%	/mnt/myblk

Our pseudo device is now a real filesystem. Watch the GEOM access counts.

```
# geom disk list myblk0 | grep Mode
Mode: r1w1e1
```

r1w1e1 means one reader, one writer, one exclusive holder. The exclusive hold is UFS: it has told GEOM that it is the sole authority over writes to the device until it is unmounted.

27.10.5 Creating and Reading Files

Let us actually use the filesystem.

```
# echo "hello from myblk" > /mnt/myblk/hello.txt
# ls -l /mnt/myblk
total 4
-rw-r--r--  1 root  wheel  17 Apr 19 18:17 hello.txt
# cat /mnt/myblk/hello.txt
hello from myblk
```

Note what just happened. The call `echo "hello from myblk" > /mnt/myblk/hello.txt` traveled through the system call layer to `sys_openat`, then to VFS, then to UFS, which opened the root directory's inode, created a new inode for `hello.txt`, allocated a data block, copied the 17 bytes into the buffer cache, and scheduled a write-back. The buffer cache eventually called down to GEOM, which called down to our strategy function, which copied those bytes into our backing store.

When you then ran `cat`, the request traveled down the same stack. Except, because the data was still in the buffer cache from the recent write, UFS did not actually need to read from our device. The buffer cache served the read from RAM. If you unmount and remount, you will see an actual read.

```
# umount /mnt/myblk
# mount /dev/myblk0 /mnt/myblk
# cat /mnt/myblk/hello.txt
hello from myblk
```

That second `cat` probably did cause `BIO_READ` requests to reach our driver, because the unmount-and-remount cycle invalidated the buffer cache for that filesystem.

27.10.6 Watching the Traffic

`gstat` shows us the BIO traffic in real time. Open another terminal and run `gstat -I 1 -f myblk0`. Then in the first terminal, create a big file.


```
# dd if=/dev/zero of=/mnt/myblk/big bs=1m count=16
16+0 records in
16+0 records out
16777216 bytes transferred in 0.150 secs (112 MB/sec)
```

In the `gstat` terminal, you should see a burst of writes, perhaps spread across a second or two depending on how quickly the buffer cache flushes.

L(q)	ops/s	r/s	kBps	ms/r	w/s	kBps	ms/w	%busy	Name
0	128	0	0	0.0	128	16384	0.0	12.0	myblk0

These are the 4 KiB or 32 KiB (depending on UFS's block size) writes that UFS is issuing to fill the file. We can verify the file's presence.

```
# ls -lh /mnt/myblk
total 16460
-rw-r--r--  1 root  wheel   16M Apr 19 18:19 big
-rw-r--r--  1 root  wheel   17B Apr 19 18:17 hello.txt
# du -ah /mnt/myblk
16M      /mnt/myblk/big
4.5K     /mnt/myblk/hello.txt
16M      /mnt/myblk
```

And we can delete it again to watch the `BIO_DELETE` traffic.

```
# rm /mnt/myblk/big
```

UFS by default does not issue `BIO_DELETE` unless the filesystem was mounted with the `trim` option, so on a plain mount you will see almost no BIO traffic on delete: UFS just marks the blocks as free in its own metadata. To see `BIO_DELETE`, we would need to mount with `-o trim`, which we will cover briefly in the labs.

27.10.7 Unmounting

Unmount the filesystem before unloading the module.

```
# umount /mnt/myblk
# geom disk list myblk0 | grep Mode
Mode: r0w0e0
```

The access count dropped back to zero as soon as UFS released its exclusive hold. Our driver is now free to be unloaded or further mucked with.

27.10.8 Attempting to Unload While Mounted

What happens if you forget the `umount` and try to unload the module?

```
# mount /dev/myblk0 /mnt/myblk
# kldunload myblk
kldunload: can't unload file: Device busy
```

The kernel refuses. The `g_disk` subsystem knows that our provider still has an active exclusive holder, and it will not let `disk_destroy` proceed until the hold is released. This is the same mechanism we saw in Chapter 26 protecting the USB serial device during an active session, lifted to the GEOM layer.

This is a safety feature. Unloading the module while a filesystem is mounted on the backing device would cause the kernel to panic on the next BIO: the strategy function would no longer exist, but UFS would still try to call into it.

Unmount first, then unload.

```
# umount /mnt/myblk
# kldunload myblk
# kldstat | grep myblk
#
```

Clean.

27.10.9 A Brief Anatomy of UFS on Top of Our Driver

Now that we have UFS mounted on our device, it is worth pausing to notice what is actually on the backing store. UFS is a well-documented filesystem, and seeing its structures in place on a device we control is illuminating.

The first 65535 bytes of a UFS filesystem are reserved for the boot area. On our device, these bytes are all zero because `newfs_ufs` does not write a boot sector by default.

At offset 65536 lives the superblock. The superblock is a fixed-size structure that describes the geometry of the filesystem: the block size, the fragment size, the number of cylinder groups, the location of the root inode, and many other invariants. `newfs_ufs` writes the superblock first, and it also writes backup copies at predictable offsets in case the primary is corrupted.

Following the superblock come the cylinder groups. Each cylinder group holds inodes, data blocks, and meta-data for a chunk of the filesystem's address space. The number and size of cylinder groups depends on the filesystem size. Our 32 MiB filesystem has four cylinder groups of 8 MiB each.

Within each cylinder group sit inode blocks. Each inode is a small structure (256 bytes on FreeBSD UFS2) that describes a single file or directory: its type, owner, permissions, timestamps, size, and the block addresses of its data.

Finally, the data blocks themselves hold file contents. These are allocated from the free-block map in the cylinder group.

When we wrote "hello from myblk" into `/mnt/myblk/hello.txt`, the kernel did roughly the following:

1. VFS asked UFS to create a new file `hello.txt` in the root directory.
2. UFS allocated an inode from the root cylinder group's inode table.
3. UFS updated the root directory's inode to include an entry for `hello.txt`.
4. UFS allocated a data block for the file.
5. UFS wrote the 17 bytes of content into that data block.
6. UFS wrote the updated inode back.
7. UFS wrote the updated directory entry back.
8. UFS updated its internal bookkeeping.

Each of those steps turned into one or more BIOs to our driver. Most were small writes on metadata blocks. The file content itself was one BIO. UFS's Soft Updates feature orders the writes to ensure crash consistency.

If you want to see these BIOs in action, run your DTrace one-liner from Lab 7 while creating a file. You will see a small burst of writes around the time of the echo.

27.10.10 How Mount Actually Works

The `mount(8)` command is a wrapper around the `mount(2)` system call. That system call takes a filesystem type, a source device, and a target mount point, and it asks the kernel to perform the mount.

The kernel's response is to find the appropriate filesystem code by type (UFS, ZFS, tmpfs, etc.) and to call its mount handler, which in UFS's case is `ufs_mount` in `/usr/src/sys/ufs/ffs/ffs_vfsops.c`. The mount handler validates the source, opens it as a GEOM consumer, reads the superblock, verifies that it is well-formed, allocates an in-memory mount structure, and installs it in the namespace.

From our driver's point of view, none of this is visible. We see a series of BIOs: first a few reads for the superblock, then whatever UFS needs to bootstrap its in-memory state. Once mount has succeeded, UFS issues BIOs on its own schedule as the filesystem is used.

If mount fails, UFS reports an error and the kernel's mount code cleans up. The GEOM consumer is detached, the access count drops, and the namespace is left alone. Our driver does not need to do anything special on mount failure.

27.10.11 The GEOM Character Interface

Earlier in the chapter we said that raw access through `/dev/myblk0` goes through "GEOM's character interface". Here is what that means in more detail.

GEOM publishes a character device for every provider. This is not the same as a `cdev` created with `make_dev`; it is a specialised path within GEOM that presents a provider as a character device to `devfs`. The code for this lives in `/usr/src/sys/geom/geom_dev.c`.

When a user program opens `/dev/myblk0`, `devfs` routes the open to GEOM's character-interface code, which attaches a consumer to our provider with the requested access mode. When the program writes, GEOM's character-interface code builds a BIO and issues it to our provider, which routes it to our strategy function. When the program closes the file descriptor, GEOM detaches the consumer, releasing the access.

The character-interface layer translates between `struct uio` (the user-space I/O descriptor) and `struct bio` (the block-layer I/O descriptor). It splits large user I/O into multiple BIOs when necessary, respecting the `d_maxsize` we specified.

All of this is invisible to our driver. We just receive BIOS. But knowing that the character interface exists helps you understand why certain user-space operations map to certain BIO patterns, and why `d_maxsize` matters.

27.10.12 What Filesystems Need from a Block Driver

Now that we have actually mounted a filesystem on our driver, we can describe more precisely what a filesystem requires from a block driver underneath.

A filesystem needs **correct reads and writes**. If a write at offset `X` is followed by a read at offset `X`, the read must return what the write put there, up to the granularity of the sector size. We guaranteed this with `memcpy` into and out of our backing store.

A filesystem needs **correct bounds**. The block driver must not accept reads or writes that extend beyond the media size. We check this explicitly in the strategy function.

A filesystem needs **stable media size**. The size of the device must not change under the filesystem's feet once it is mounted, because filesystem metadata encodes offsets and counts that assume a fixed size. Our driver holds the media size constant.

A filesystem needs **crash safety**, to the extent the underlying storage provides it. UFS can recover from an unclean shutdown if the backing store does not lose previously committed writes. Our RAM-backed driver loses everything on reboot, but it is at least self-consistent while running. In Section 7, we will introduce options for persistence.

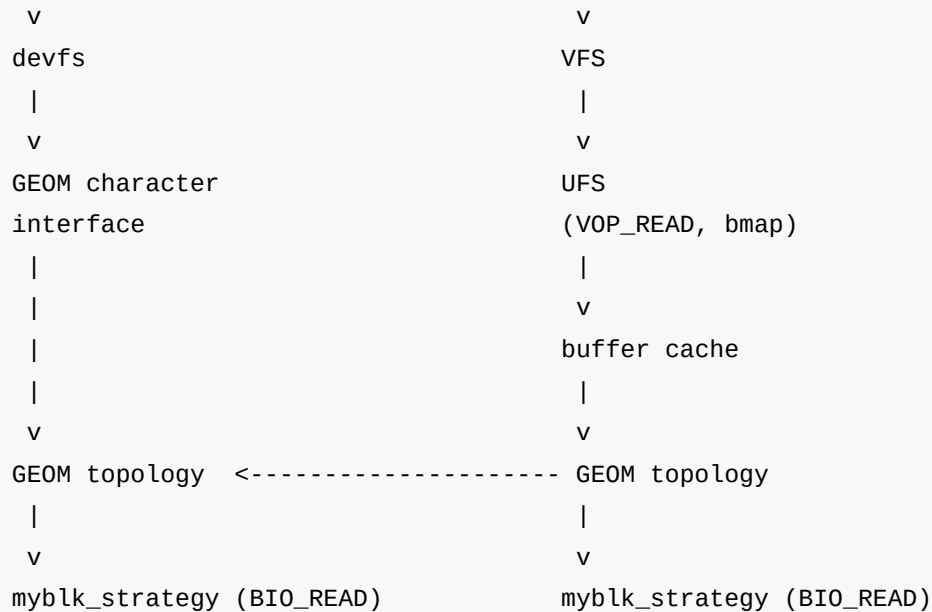
A filesystem sometimes needs **flush semantics**. A call to `BIO_FLUSH` should ensure that all previously issued writes are durable before returning. Our RAM-backed driver trivially satisfies this, because there is no deferred writeback in its path.

Finally, a filesystem benefits from **fast sequential access**. This is a quality-of-service matter rather than a correctness matter, but our driver is fine in this regard because `memcpy` is fast.

27.10.13 Raw Access Versus Filesystem Access, Visualised

Let us draw the two access paths side by side, using our actual driver as the anchor.

Raw access:	Filesystem access:
<code>dd(1)</code>	<code>cat(1)</code>
<code>v</code>	<code>v</code>
<code>open("/dev/myblk0")</code>	<code>open("/mnt/myblk/hello.txt")</code>
<code>v</code>	<code>v</code>
<code>read(fd, ...)</code>	<code>read(fd, ...)</code>
<code>v</code>	<code>v</code>
<code>sys_read</code>	<code>sys_read</code>



The last two hops are identical. Our strategy function is called exactly the same way regardless of whether the request came from `dd` or from `cat` on a mounted file. This is the great advantage of living at the block layer: we do not need to distinguish between the two paths. The upper layers sort out how to translate file-level operations into block-level operations, and we deal in blocks.

27.10.14 Watching the Request Path with DTrace

If you want to see the request path explicitly, DTrace can help.

```
# dtrace -n 'fbt::myblk_strategy:entry { printf("cmd=%d off=%lld len=%u", \
    args[0]->bio_cmd, args[0]->bio_offset, args[0]->bio_bcount); }'
```

With the probe running, do something on the mounted filesystem in another terminal and watch the BIOs arrive. You will see reads come through in 512-byte to 32 KiB chunks, depending on UFS's block size and what operation you performed. Running `dd if=/dev/zero of=/mnt/myblk/test bs=1m count=1` produces a burst of 32 KiB writes.

DTrace is one of the most capable observability tools FreeBSD provides, and it comes alive with storage work because the BIO path is so instrumented. We will use it more in later chapters, but even a one-liner like the above is enough to make the abstract path concrete.

27.10.15 Wrapping Up Section 6

Our pseudo block device now plays the full role of a storage device: raw access through `dd`, filesystem access through UFS, and safe coexistence with kernel unmount protections. The strategy function we wrote in Section 5 did not need to change at all for UFS to work, because UFS and `dd` share the same block-layer protocol below them.

We have also seen the end-to-end flow: VFS at the top, UFS just below, the buffer cache between, GEOM below that, and our driver at the very bottom. That flow is the same for every storage driver in FreeBSD. You now know how to occupy the bottom of it.

In the next section, we will turn our attention to persistence. A RAM-backed device is convenient for testing but loses its contents on every reload. We will discuss options for making the backing store persistent, what trade-offs each option brings, and how to add one of them to our driver.

27.11 Section 7: Persistence and In-Memory Backing Stores

Our driver is self-consistent while running. If you write a byte at offset X, you can read it back at offset X moments later. If you create a file on the mounted filesystem, you can read it again until you unmount or unload. This is already useful for testing and for short-lived workloads.

It is not, however, durable. Unload the module and the backing buffer is freed. Reboot the machine and every byte vanishes. For a teaching driver that is arguably a feature: it reboots clean, it does not accumulate state across runs, and it cannot silently corrupt a previous session. But understanding the options for making storage persistent is essential for real driver work, so this section walks through the major choices and then shows how to add the simplest kind of persistence to our driver.

27.11.1 Why Persistence Is Hard

Storage persistence is not just about where the bytes live. It is about three intertwined properties.

Durability means that once a write returns, the data is safe against a crash. On a hardware disk, durability is typically coupled with the disk's own cache policy: the write hits the drive's internal buffer, then the platter, then the drive reports completion. `BIO_FLUSH` is the hook that gives filesystems a way to demand flush-to-platter semantics.

Consistency means that a read at offset X returns the most recent write at offset X, not some earlier or partial version. Consistency is usually provided by the hardware or by careful locking in the driver.

Crash safety means that after an unclean shutdown, the state of the storage is usable. Either it reflects all committed writes, or it reflects a well-defined prefix of them. UFS has SU+J (Soft Updates with Journaling) to help recover from a crash; ZFS uses copy-on-write and atomic transactions. All of that relies on a block layer that behaves predictably.

For a teaching driver, we do not need to address all three with full rigour. We need to understand what the choices are and to pick one that fits our goals.

27.11.2 The Options

There are four common ways to back a pseudo block device.

In-memory backing (our current choice). Fast, simple, lost on reload. Implemented as a `malloc`'d buffer. Scales poorly past a few MiB because it demands contiguous kernel memory.

Page-at-a-time in-memory backing. `md(4)` uses this internally for large memory disks. Instead of one big buffer, the driver keeps an indirection table of page-sized allocations and fills them on demand. This scales to very large sizes and avoids wasting memory on sparse regions, but it is more complex.

Vnode backing. The driver opens a file in the host filesystem and uses it as the backing store. `mdconfig -t vnode` is the classic example. Reads and writes go through the host's filesystem, which gives persistence at the cost of speed and of a dependency on the host filesystem's correctness. This is how FreeBSD often boots from a memory-disk image embedded in the kernel: the kernel loads the image, presents it as `/dev/md0`, and the root filesystem runs on it.

Swap backing. The driver uses a swap-backed VM object as the backing store. `mdconfig -t swap` uses this. It provides persistence across reboots only to the extent that swap is persistent, which on most systems it is not. But it provides a very large sparse address space without consuming physical memory until touched, which is useful for scratch storage.

For this chapter, we will stick with the in-memory option. It is the simplest, it is enough for the labs, and it demonstrates every other concept cleanly. We will discuss how to switch to vnode-backed storage as an exercise, and we will point to `md(4)` for those who want to see a full-featured implementation.

27.11.3 Saving and Restoring the Buffer

If we want our device to remember its contents across reloads, without changing the backing approach, we can save the buffer to a file on unload and restore it on load. This is not elegant, but it is direct, and it illustrates the contract clearly: the driver is responsible for getting the backing bytes into memory before the first BIO arrives and for flushing them to safety before the last BIO leaves.

In our case, the mechanics would look like this.

On module load, after allocating the backing buffer but before calling `disk_create`, optionally read a file on the host filesystem into the buffer. On module unload, after `disk_destroy` has completed, optionally write the buffer back to that file.

Doing this cleanly from inside the kernel requires the vnode API. The kernel provides `vn_open`, `vn_rdw`, and `vn_close`, which together let a module read or write a path in the host filesystem. These are not APIs we want to use casually, because they are not designed for high-throughput I/O from inside a driver, and because they run on whatever filesystem happens to be mounted at that path, which is not always safe. But for a one-shot save and restore at load and unload time, they are acceptable.

For teaching purposes we will not implement this. The correct way to persist a block device's contents is to use a real backing store, not to snapshot a RAM buffer. But understanding the technique helps clarify the contract.

27.11.4 The Contract With Upper Layers

Whatever your backing store, the contract with upper layers is precise.

A BIO_WRITE that completes successfully must be visible to all subsequent BIO_READ requests, regardless of buffering layers. Our in-memory driver satisfies this because `memcpy` is the visible effect.

A BIO_FLUSH that completes successfully must have made all previously successful BIO_WRITE requests durable. Our in-memory driver satisfies this trivially because there is no lower layer between our `memcpy`

and the backing memory; all writes are “durable” in the sense we can offer. A real disk driver typically issues a cache-flush command to the hardware in response to `BIO_FLUSH`.

A `BIO_DELETE` may discard data but must not corrupt neighbouring blocks. Our in-memory driver satisfies this by zeroing only the requested range. A real SSD driver might issue TRIM for the range; a real HDD driver typically has no hardware support for DELETE and can safely ignore it.

A `BIO_READ` must return the media contents or an error; it must not return uninitialised memory, stale cached data from a different transaction, or random bytes. Our in-memory driver satisfies this by zeroing the backing on allocation and writing only through the strategy function.

If you keep these four rules in mind as you design a new driver, you will avoid nearly every correctness bug that plagues new storage drivers.

27.11.5 What `md(4)` Does Differently

The kernel’s `md(4)` driver is a mature, multi-type memory-disk driver. It supports five backing types: `malloc`, `preload`, `swap`, `vnode`, and `null`. Each type has its own strategy function that knows how to serve requests for that backing kind. Reading `/usr/src/sys/dev/md/md.c` is a valuable follow-up to this chapter because it shows how a real driver handles all of the cases we are glossing over.

A few specific things `md(4)` does that we do not.

`md(4)` uses a dedicated worker thread per unit. Incoming BIOs are queued on the `softc`, and the worker thread dequeues them one by one and dispatches them. This lets the strategy function be very simple: just enqueue and signal. It also isolates blocking work in the worker, which matters for the `vnode` backing type because `vn_rdwr` can block.

`md(4)` uses `DEV_BSHIFT` (which is 9, meaning 512-byte sectors) consistently and uses integer arithmetic rather than floating point to handle offsets. This is standard practice in the block layer.

`md(4)` has a full `ioctl` surface for configuration. The `mdconfig` tool talks to the kernel through `ioctls` on `/dev/mdctl`, and the driver supports `MDIOCATTACH`, `MDIOCDETACH`, `MDIOCQUERY`, and `MDIOCRESIZE`. We have not implemented anything comparable, because for our pseudo device the configuration is baked in at compile time.

`md(4)` uses `DISK_VERSION_06`, which is the current version of the `g_disk` ABI. Our driver does the same, through the `DISK_VERSION` macro.

If you want to see a production-quality pseudo block device, `md(4)` is the canonical reference. Almost everything we are building would, in a real driver, grow to resemble the shape of `md(4)` over time.

27.11.6 A Note on Swap-Backed Memory

One technique worth naming, even though we will not use it here, is swap-backed memory. Instead of a `malloc`’d buffer, a driver can allocate a VM object of type `OBJT_SWAP` and map pages from it on demand. The pages are backed by swap space, which means they can be paged out when the system is under memory pressure and paged back in when touched. This gives you a very large, sparse, on-demand backing store that behaves like RAM when hot and like disk when cold.

md(4) uses exactly this approach for its swap-backed memory disks. The swap VM object acts as a backing store that the kernel's VM subsystem manages for us, without the driver needing to allocate contiguous physical memory up front. The OBJT_SWAP object can hold terabytes of addressable space on a system with only gigabytes of RAM, because most of that space is never touched.

If you ever need to prototype a block device larger than a few hundred MiB, swap-backed memory is likely the right tool. The VM API for it lives in `/usr/src/sys/vm/swap_pager.c`. Reading it is not light work, but it is educational.

27.11.7 A Note on Preloaded Images

FreeBSD has a mechanism called **preloaded modules**. During boot, the loader can bring in not just kernel modules but also arbitrary data blobs, which are made available to the kernel through `preload_fetch_addr` and `preload_fetch_size`. md(4) uses this to expose preloaded filesystem images as `/dev/md*` devices, which is one of the ways FreeBSD can boot entirely from a memory-disk root.

Preloaded images are not a persistence mechanism per se. They are a way to ship data with a kernel module. But they are often used in embedded systems, where the root filesystem is too precious to live on writable storage.

27.11.8 A Small Extension: Persisting Across Module Reloads Only

We are not going to add real persistence to our driver, but this is a good moment to talk about what it would actually take to make a backing store survive a module unload and reload within the same kernel boot. The naive first idea, and one that beginners reach for quickly, is to put the backing pointer in a file-scope static variable and simply not free it in the unload handler. Let us look at why that does not work and what does.

Consider this sketch:

```
static uint8_t *myblk_persistent_backing; /* wishful thinking */
static size_t  myblk_persistent_size;
```

The intuition is that if we allocate `myblk_persistent_backing` on first attach and refuse to free it on detach, a subsequent `kldload` will see the pointer still set and reuse the buffer. The problem is that this picture ignores how a KLD is actually loaded and unloaded. When `kldunload` removes our module, the kernel reclaims the module's text, data, and `.bss` segments along with the rest of its image. Our static pointer does not persist in some stable location; it vanishes together with the module. When `kldload` then brings the module back, the kernel allocates a fresh `.bss`, zeroes it, and our pointer starts life as `NULL` again. The `malloc'd` buffer we allocated on the previous attach is still sitting in kernel heap somewhere, but we have lost every handle to it. We have leaked it.

`SYSUNINIT` does not help either, because in a KLD context it fires on `kldunload`, not on some later “final tear-down” event. Registering a `SYSUNINIT` to free the buffer would free it on every unload, which is exactly what we did not want. There is no KLD-level hook that means “the module file is really, truly being removed from memory for good” distinct from plain `kldunload`.

Two techniques actually achieve cross-unload persistence, and both are used by md(4) in production. The first is a **file-backed store**. Instead of allocating a kernel heap buffer, the driver opens a file on an existing filesystem

using the vnode I/O API (`VOP_READ`, `VOP_WRITE`, and the vnode reference taken via `vn_open`) and services BIOs by reading from and writing to that file. On unload, the driver closes the file; on the next load, it re-opens it. The persistence is real because it lives in a filesystem whose state is independent of our module. This is exactly what `md -t vnode -f /path/to/image.img` does, and you can study it in `/usr/src/sys/dev/md/md.c`.

The second technique is a **swap-backed store**. The driver allocates a VM object of type `OBJT_SWAP`, as we mentioned earlier, and maps pages from it on demand. The pager lives at a higher level of the kernel than our module, so the object can outlive any particular `kldunload` as long as something else holds a reference to it. In practice, `md(4)` uses this for swap-backed memory disks, and it ties the object's lifetime to a kernel-wide list rather than to a module instance.

For our teaching driver, we will not implement either technique. The point of showing this discussion is to make sure you understand why the apparent shortcut does not work, so that you do not spend an afternoon debugging a buffer that keeps disappearing after `kldunload`. If you want to experiment with real cross-unload persistence, read `md.c` carefully, particularly the `MD_VNODE` and `MD_SWAP` branches in `mdstart_vnode` and `mdstart_swap`, and note how the backing objects are attached to the per-unit `struct md_s` rather than to module-scope globals. That structural choice is what makes those backends work across module lifecycles.

27.11.9 Sketching a Vnode-Backed Strategy Function

To make the earlier discussion concrete, let us sketch what a vnode-backed strategy function looks like at the code level. We are not going to drop this into our teaching driver. We are showing it so that you can see what the “real” solution involves and can recognise the same shape in `md.c` when you read it.

The idea is that the per-unit `softc` holds a reference to a vnode, acquired at attach time from a path provided by the administrator. The strategy function translates each BIO into a `vn_rdwr` call at the right offset and completes the BIO based on the result.

Attach acquires the vnode:

```
static int
myblk_vnode_attach(struct myblk_softc *sc, const char *path)
{
    struct nameidata nd;
    int flags, error;

    flags = FREAD | FWRITE;
    NDINIT(&nd, LOOKUP, FOLLOW, UIO_SYSSPACE, path);
    error = vn_open(&nd, &flags, 0, NULL);
    if (error != 0)
        return (error);
    NDFREE_PNBUF(&nd);
    VOP_UNLOCK(nd.ni_vp);
    sc->vp = nd.ni_vp;
    sc->vp_cred = curthread->td_ucred;
    crhold(sc->vp_cred);
}
```

```

    return (0);
}

```

`vn_open` looks up the path and returns a locked, referenced `vnode`. We then drop the lock, because we want to hold a reference without blocking other operations, and we hang the `vnode` pointer on our `softc`. We also keep a reference to the credentials we will use for subsequent I/O.

The strategy function services BIOs against the `vnode`:

```

static void
myblk_vnode_strategy(struct bio *bp)
{
    struct myblk_softc *sc = bp->bio_disk->d_drv1;
    int error;

    switch (bp->bio_cmd) {
    case BIO_READ:
        error = vn_rdwr(UIO_READ, sc->vp, bp->bio_data,
            bp->bio_length, bp->bio_offset, UIO_SYSSPACE,
            IO_DIRECT, sc->vp_cred, NOCRED, NULL, curthread);
        break;
    case BIO_WRITE:
        error = vn_rdwr(UIO_WRITE, sc->vp, bp->bio_data,
            bp->bio_length, bp->bio_offset, UIO_SYSSPACE,
            IO_DIRECT | IO_SYNC, sc->vp_cred, NOCRED, NULL,
            curthread);
        break;
    case BIO_FLUSH:
        error = VOP_FSYNC(sc->vp, MNT_WAIT, curthread);
        break;
    case BIO_DELETE:
        /* Vnode-backed devices usually do not support punching
         * holes through BIO_DELETE without additional plumbing.
         */
        error = EOPNOTSUPP;
        break;
    default:
        error = EOPNOTSUPP;
        break;
    }

    if (error != 0) {
        bp->bio_error = error;
        bp->bio_flags |= BIO_ERROR;
        bp->bio_resid = bp->bio_bcount;
    }
}

```

```

    } else {
        bp->bio_resid = 0;
    }
    biodone(bp);
}

```

Notice how the shape of the switch is identical to our RAM-backed strategy function. The only difference is what the case arms do: instead of memcpy into a buffer, we call `vn_rdwr` against a vnode. The framework above us, GEOM and the buffer cache, does not know or care which backend we chose.

Detach releases the vnode:

```

static void
myblk_vnode_detach(struct myblk_softc *sc)
{
    if (sc->vp != NULL) {
        (void)vn_close(sc->vp, FREAD | FWRITE, sc->vp_cred,
            curthread);
        sc->vp = NULL;
    }
    if (sc->vp_cred != NULL) {
        crfree(sc->vp_cred);
        sc->vp_cred = NULL;
    }
}

```

`vn_close` releases the vnode reference and, if this was the last reference, allows the vnode to be recycled. The credentials are reference-counted the same way.

Why does this give us cross-unload persistence? Because the state we care about, namely the contents of the backing store, lives in a file on a real filesystem whose lifetime is completely independent of our module. When we call `kldunload`, the vnode reference is released and the file closes; its contents on disk are preserved by the filesystem. When we call `kldload` again and attach, we open the file again and pick up where we left off.

The remaining subtleties are substantial. Error paths need to release the vnode if `vn_open` succeeded but subsequent registration steps failed. Calls to `vn_rdwr` can sleep, which means the strategy function must not be called from a context that disallows sleeping; in practice, that is why `md(4)` uses a dedicated worker thread for vnode-backed units. Reading a file can race with the administrator modifying it, so production drivers usually take measures to detect concurrent external changes. `VOP_FSYNC` is not free, so a fast path that batches writes before flushing is typical. And the vnode lifetime itself is bound by VFS's own reference counting, which interacts with unmount of the containing filesystem.

We will not add this to our teaching driver, but when you read `mdstart_vnode` in `/usr/src/sys/dev/md/md.c`, you will recognise every one of these issues handled carefully and explicitly.

27.11.10 Wrapping Up Section 7

Persistence is a layered concept. Durability, consistency, and crash safety are all part of what a real storage device must provide, and different backing stores give different subsets of those guarantees. For a teaching driver, a `malloc`'d in-memory buffer is a reasonable choice, and we can add “survives module reload” semantics without much code by detaching the buffer from the per-instance `softc`.

For production, the techniques grow more elaborate: page-at-a-time allocation, swap-backed VM objects, vnode-backed files, dedicated worker threads, `BIO_FLUSH` coordination, and careful handling of every error path. `md(4)` is the canonical example in the FreeBSD tree, and reading it is strongly recommended.

In the next section, we will focus on the teardown path in detail. We will look at how GEOM coordinates unmount, detach, and cleanup; how access counts gate the module unload path; and how our driver should behave when something goes wrong mid-teardown. Storage unmount bugs are some of the nastier kinds of kernel bug, and careful attention here pays off for the rest of your driver-writing career.

27.12 Section 8: Safe Unmount and Cleanup

Storage drivers handle the end of their lives with more care than character drivers because the stakes are higher. When a character driver unloads cleanly, the worst thing that can happen is that an open session is torn down, possibly with some bytes in flight being lost. When a storage driver unloads while a filesystem is mounted on it, the worst thing that can happen is that the kernel panics on the next BIO, and the user is left with a filesystem image that may or may not have been in a consistent state when the driver disappeared.

The good news is that the kernel's defences make the catastrophic case almost impossible if you use `g_disk` correctly. The refusal of `kldunload` to proceed when the GEOM access count is non-zero, which we saw in Section 6, is the primary safety net. But it is not the only concern. This section walks through the teardown path in detail so that you know what to expect, what to implement, and what to test.

27.12.1 The Expected Teardown Sequence

The nominal sequence of events when a user wants to remove a storage driver is as follows.

1. The user unmounts every filesystem that is mounted on the device.
2. The user closes any program that has `/dev/myblk0` open for raw access.
3. The user calls `kldunload`.
4. The module unload function calls `disk_destroy`.
5. `disk_destroy` queues the provider for withering, which runs on the GEOM event thread.
6. The withering process waits for any in-flight BIO to complete.
7. The provider is removed from the GEOM topology and the `/dev` node is destroyed.
8. `disk_destroy` returns control to our unload function.
9. Our unload function frees the `softc` and the backing store.
10. The kernel unloads the module.

Each step has its own failure modes. Let us walk through them.

27.12.2 Step 1: Unmount

The user runs `umount /mnt/myblk`. VFS asks UFS to flush the filesystem, which causes the buffer cache to issue any pending writes to GEOM, which routes them to our driver. Our strategy function services the writes and calls `biodone`. The buffer cache reports success; UFS disposes of its in-memory state; VFS releases the mount point. The consumer that UFS had attached to our provider is detached. The access count drops.

Our driver does not do anything special during this phase. We keep handling BIOs as they arrive until UFS stops issuing them.

27.12.3 Step 2: Close Raw Access

The user ensures that no program holds `/dev/myblk0` open. If a `dd` is running, kill it. If a shell has the device open via `exec`, close that. Until every open handle is released, the access count will remain non-zero on at least one of the `r`, `w`, or `e` counters.

Again, our driver does nothing special. The `close(2)` calls on `/dev/myblk0` propagate through `devfs`, through GEOM's character-device integration, and release their access. No BIOs are issued for close.

27.12.4 Step 3: kldunload

The user runs `kldunload myblk`. The kernel's module subsystem calls our unload function with `MOD_UNLOAD`. Our unload function calls `myblk_detach_unit`, which calls `disk_destroy`.

At this point, our driver is about to stop existing. We must not be holding any lock that could block, we must not be blocking on our own worker threads (we do not have any in this design), and we must not be issuing new BIOs. Nothing we do now should cause new work for the kernel.

27.12.5 Step 4: disk_destroy

`disk_destroy` is the point of no return. Reading the source in `/usr/src/sys/geom/geom_disk.c` reveals that it does three things:

1. It sets a flag on the disk to indicate that destruction is in progress.
2. It queues a GEOM event that will actually dismantle the provider.
3. It waits for the event to complete.

While we are waiting, the GEOM event thread picks up the event and walks our geom. If the access counts are zero, the event proceeds. If they are not zero, the event panics with a message about trying to destroy a disk that still has users.

This is where the importance of Step 1 and Step 2 shows up. If you skip them and try to unload while the filesystem is mounted, the panic happens here. Fortunately, `g_disk` refuses to reach the panic because the module subsystem has already refused the unload earlier, but if you were to bypass the module subsystem and call `disk_destroy` directly from some other context, this is the check that protects the kernel.

27.12.6 Step 5 to 7: Withering

The GEOM withering process is how providers are removed from the topology. It works by marking the provider as withered, cancelling any BIOs that were queued but not yet delivered, waiting for any in-flight BIOs to complete, removing the provider from the geom's provider list, and then removing the geom from the class. The `/dev` node is removed as part of this.

During withering, the strategy function may still be called for BIOs that were in flight before the withering started. Our strategy function will handle them normally, because our driver does not know or care that withering is in progress. The framework is responsible for ensuring no new BIOs are issued after the point of no return.

If our driver had worker threads, a queue, or other internal state, we would need to coordinate with withering carefully. `md(4)` is a good example of a driver that does this: its worker thread watches for a shutdown flag and drains its queue before exiting. Since our driver is entirely synchronous and single-threaded, we do not have this complication.

27.12.7 Step 8 to 9: Free Resources

Once `disk_destroy` returns, the disk is gone, the provider is gone, and no more BIOs will arrive. It is safe to free the backing store and destroy the mutex.

```
static void
myblk_detach_unit(struct myblk_softc *sc)
{
    if (sc->disk != NULL) {
        disk_destroy(sc->disk);
        sc->disk = NULL;
    }
    if (sc->backing != NULL) {
        free(sc->backing, M_MYBLK);
        sc->backing = NULL;
        sc->backing_size = 0;
    }
}
```

Our unload function then destroys the mutex and frees the softc.

```
case MOD_UNLOAD:
    sc = myblk_unit0;
    if (sc == NULL)
        return (0);
    myblk_detach_unit(sc);
    mtx_destroy(&sc->lock);
    free(sc, M_MYBLK);
```

```
myblk_unit0 = NULL;
printf("myblk: unloaded\n");
return (0);
```

27.12.8 Step 10: Module Unload

The module subsystem unloads the .ko file. At this point, the driver is gone. Any attempt to reference the module by name will fail until the user loads it again.

27.12.9 What Can Go Wrong

The happy path is smooth. Let us enumerate the unhappy paths and how to recognise them.

kldunload returns Device busy. The filesystem is still mounted, or a program still has the raw device open. Unmount and close, then retry. This is the most common failure, and it is benign.

disk_destroy never returns. Something is holding a BIO that will never complete, and the withering process is waiting for it. In practice, this happens if your strategy function fails to call biodone on some path. Look at `procstat -kk` of the `g_event` thread; if it is stuck in `g_waitfor_event`, you have a leaked BIO. The fix is in your strategy function: ensure that every path calls `biodone` exactly once.

The kernel panics with “g_disk: destroy with open count”. Your driver called `disk_destroy` while the provider still had users. This should not happen if you only call `disk_destroy` from the module unload path, because the module subsystem refuses to unload busy modules. But if you call `disk_destroy` in response to some other event, you must check the access count yourself or tolerate the panic.

The kernel panics with “Freeing free memory”. Your driver tried to free the softc or the backing store twice. Check your detach path for race conditions or for early exits that free and then fall through to free again.

The kernel panics with “Page fault in kernel mode”. Something is dereferencing a freed pointer, most often the backing store after it has been freed while a BIO is still in flight. The fix is to ensure `disk_destroy` completes before freeing anything the strategy function touches.

27.12.10 The d_gone Callback

There is one more piece of the teardown story worth discussing. The `d_gone` callback is invoked when something other than our driver decides the disk should go away. The canonical example is hotplug removal: a user yanks a USB drive, the USB stack tells the storage driver the device is gone, and the storage driver wants to tell GEOM to tear down the disk as gracefully as possible even though I/O will start failing.

Our driver is a pseudo device; it does not have a physical disappearance event. But registering a `d_gone` callback costs nothing and makes the driver slightly more robust against future extensions.

```
static void
myblk_disk_gone(struct disk *dp)
{
```



```
    printf("myblk: disk_gone(%s%u)\n", dp->d_name, dp->d_unit);  
}
```

Register it with `sc->disk->d_gone = myblk_disk_gone;` before `disk_create`. The function is called by `g_disk` when `disk_gone` is invoked. You can trigger it manually during development by calling `disk_gone(sc->disk)` from a test path; you will not usually call it yourself in a pseudo driver.

Note the difference between `disk_gone` and `disk_destroy`. `disk_gone` says “this disk has physically vanished; stop accepting I/O and mark the provider as error-returning”. `disk_destroy` says “remove this disk from the topology and free its resources”. In a hot-unplug path, `disk_gone` is usually called first (by the bus driver, when it notices the device is gone), and `disk_destroy` is called later (by the module unload, or by the bus driver’s detach function). Between the two calls, the disk still exists in the topology but all I/O fails. Our driver does not implement this dual-phase teardown; a USB mass storage driver, for instance, must.

27.12.11 Testing the Teardown

Teardown bugs are often discovered not by careful testing but by accident, months later, when some user finds an unusual sequence that triggers them. It is much cheaper to test teardown deliberately.

Here are the tests I recommend running on any new storage driver.

Basic unload. Load, format, mount, unmount, unload. Verify `dmesg` shows our load and unload messages and nothing else. Repeat ten times to catch slow leaks.

Unload without unmount. Load, format, mount. Attempt to unload. Verify the unload is refused. Unmount, then unload. Verify no lingering state.

Unload under load. Load, format, mount, start a `dd if=/dev/urandom of=/mnt/myblk/stress bs=1m count=64`. While the `dd` runs, attempt to unload. Verify the unload is refused. Wait for `dd` to finish. Unmount. Unload. Verify clean.

Unload with raw open. Load. In another terminal, run `cat > /dev/myblk0` to hold the device open. Attempt to unload. Verify the unload is refused. Kill the `cat`. Unload. Verify clean.

Reload stress. Load, unload, load, unload in a tight loop for a minute. If `vmstat -m` or `zpool list` starts showing leaks, investigate.

Panic on corruption. This one is harder: deliberately corrupt the module state via a kernel debugger hook and verify that the driver does not silently return bad data. In practice, few beginners do this, and it is not required for a teaching driver.

If all of these pass, you have a reasonably robust teardown. Continue testing every change that touches the unload path.

27.12.12 The Idempotency Principle

A good teardown path is idempotent: calling it twice is no worse than calling it once. This matters because error paths during attach may call the teardown before everything has been set up.

Write your teardown to check whether each resource was actually allocated before trying to free it.

```
static void
myblk_detach_unit(struct myblk_softc *sc)
{
    if (sc == NULL)
        return;

    if (sc->disk != NULL) {
        disk_destroy(sc->disk);
        sc->disk = NULL;
    }
    if (sc->backing != NULL) {
        free(sc->backing, M_MYBLK);
        sc->backing = NULL;
        sc->backing_size = 0;
    }
}
```

Setting pointers to NULL after freeing them is a small discipline that pays off. It makes double-free mistakes obvious at runtime (they become no-ops rather than corruptions), and it makes the teardown function idempotent.

27.12.13 Ordering and Reverse Order

A general teardown guideline: free resources in the reverse order of allocation. If attach goes A -> B -> C, detach should go C -> B -> A.

In our driver, attach goes malloc backing -> disk_alloc -> disk_create. So detach goes disk_destroy -> free backing. We skip freeing the disk because disk_destroy frees it for us.

This pattern is universal. Every well-written teardown function reverses the allocation order. When you see a detach that runs in the same order as the attach, suspect a bug.

27.12.14 The MOD_QUIESCE Event

There is a third module event we have not mentioned: MOD_QUIESCE. It is delivered before MOD_UNLOAD and gives the module a chance to refuse unload if the driver is in a state where unloading is unsafe.

For most drivers, the GEOM access-count check is sufficient, and implementing MOD_QUIESCE is not needed. But if your driver has internal state that makes unload unsafe independent of GEOM (for example, a cache that must be flushed), MOD_QUIESCE is where you decline the unload by returning an error.

Our driver does not implement MOD_QUIESCE. The default behaviour is to accept it silently, which is the right thing for us.

27.12.15 Coordinating With Future Worker Threads

If you ever add a worker thread to the driver, the teardown contract changes. You must:

1. Signal the worker to stop, typically by setting a flag on the softc.
2. Wake the worker if it is sleeping, typically with `wakeup` or `cv_signal`.
3. Wait for the worker to exit, typically with a `kthread_exit`-visible termination flag.
4. Only then call `disk_destroy`.
5. Free the softc and backing store.

Skipping any of these steps is a recipe for a panic. The usual failure mode is that the worker thread is sleeping inside a function that touches softc state after the softc has been freed. `md(4)` handles this carefully, and it is worth reading its worker shutdown code if you plan to add a worker to your own driver.

27.12.16 Cleanup in the Face of Errors

One last concern: what happens if `attach` fails partway through? Suppose `disk_alloc` succeeds, but `disk_create` fails. Or suppose we add code that validates the sector size and rejects invalid configurations before calling `disk_create`.

The pattern for handling this is “single cleanup path”. Write the `attach` function so that any failure jumps to a cleanup label that unwinds everything allocated so far, in reverse order.

```
static int
myblk_attach_unit(struct myblk_softc *sc)
{
    int error;

    sc->backing_size = MYBLK_MEDIASIZE;
    sc->backing = malloc(sc->backing_size, M_MYBLK, M_WAITOK | M_ZERO);

    sc->disk = disk_alloc();
    sc->disk->d_name      = MYBLK_NAME;
    sc->disk->d_unit      = sc->unit;
    sc->disk->d_strategy   = myblk_strategy;
    sc->disk->d_ioctl      = myblk_ioctl;
    sc->disk->d_getattr    = myblk_getattr;
    sc->disk->d_gone       = myblk_disk_gone;
    sc->disk->d_sectorsize = MYBLK_SECTOR;
    sc->disk->d_mediasize  = MYBLK_MEDIASIZE;
    sc->disk->d_maxsize    = MAXPHYS;
    sc->disk->d_drv1       = sc;

    error = 0; /* disk_create is void; no error path from it */
    disk_create(sc->disk, DISK_VERSION);
    return (error);
}
```

```
/*
 * Future expansion: if we add a step that can fail between
 * disk_alloc and disk_create, use a cleanup label here.
 */
}
```

For our driver, `disk_alloc` does not fail in practice (it uses `M_WAITOK`), and `disk_create` is a void function that queues the real work asynchronously. So the attach path cannot really fail. But the pattern of preparing a single cleanup label is worth keeping in mind for drivers that grow more complex.

27.12.17 Wrapping Up Section 8

Safe unmount and cleanup for a storage driver comes down to a small set of disciplines: handle every BIO through to `biodone`, never hold locks during completion callbacks, only call `disk_destroy` when the provider has no users, free resources in the reverse order of allocation, and test the teardown under load. The `g_disk` framework handles most of the hard parts; your job is to avoid breaking its invariants.

In the next section, we will step back from the teardown specifics and talk about how to let a storage driver grow. We will discuss refactoring, versioning, how to support multiple units cleanly, and what to do when the driver becomes more than a single source file. These are the habits that turn a teaching driver into something you can keep evolving for a long time.

27.13 Section 9: Refactoring and Versioning

Our driver fits in a single file and solves one problem: it exposes a single pseudo disk of a fixed size, backed by RAM. That is a useful teaching starting point, but it is not where most real drivers live. A real storage driver evolves. It grows `ioctl` support. It grows multi-unit support. It grows tunable parameters. It splits into multiple source files. Its on-disk representation, if any, goes through format changes. It accumulates a history of compatibility choices.

This section is about the habits that let a driver grow gracefully. We will not add massive new features here; the companion labs and challenges will do that. What we will do is survey the refactoring and versioning questions that arise as any storage driver matures, and we will point to the FreeBSD-idiomatic answers for each.

27.13.1 Multi-Unit Support

Right now our driver supports exactly one instance, hardcoded as `myblk0`. If you wanted two or three pseudo disks, the current code would need duplicate `softcs` and duplicate disk registrations. Real drivers solve this with a data structure that can hold any number of units.

The idiomatic FreeBSD pattern is a global list protected by a lock. The `softc` is allocated per unit and linked into the list. A loader-time tunable or an `ioctl`-driven call decides when to create a new unit. The unit number is allocated from a `unrhdr` (unique number range) allocator.

A sketch:

```

static struct mtx      myblk_list_lock;
static LIST_HEAD(, myblk_softc) myblk_list =
    LIST_HEAD_INITIALIZER(myblk_list);
static struct unrhdr    *myblk_unit_pool;

static int
myblk_create_unit(size_t mediasize, struct myblk_softc **scp)
{
    struct myblk_softc *sc;
    int unit;

    unit = alloc_unr(myblk_unit_pool);
    if (unit < 0)
        return (ENOMEM);

    sc = malloc(sizeof(*sc), M_MYBLK, M_WAITOK | M_ZERO);
    mtx_init(&sc->lock, "myblk unit", NULL, MTX_DEF);
    sc->unit = unit;
    sc->backing_size = mediasize;
    sc->backing = malloc(mediasize, M_MYBLK, M_WAITOK | M_ZERO);

    sc->disk = disk_alloc();
    sc->disk->d_name      = MYBLK_NAME;
    sc->disk->d_unit      = sc->unit;
    sc->disk->d_strategy  = myblk_strategy;
    sc->disk->d_sectorsize = MYBLK_SECTOR;
    sc->disk->d_mediasize  = mediasize;
    sc->disk->d_maxsize    = MAXPHYS;
    sc->disk->d_drv1       = sc;
    disk_create(sc->disk, DISK_VERSION);

    mtx_lock(&myblk_list_lock);
    LIST_INSERT_HEAD(&myblk_list, sc, link);
    mtx_unlock(&myblk_list_lock);

    *scp = sc;
    return (0);
}

static void
myblk_destroy_unit(struct myblk_softc *sc)
{
    mtx_lock(&myblk_list_lock);

```

```
LIST_REMOVE(sc, link);
mtx_unlock(&myblk_list_lock);

disk_destroy(sc->disk);
free(sc->backing, M_MYBLK);
mtx_destroy(&sc->lock);
free_unr(myblk_unit_pool, sc->unit);
free(sc, M_MYBLK);
}
```

The loader initialises the unit pool once, and then individual units can be created and destroyed independently. This is very close to the pattern `md(4)` uses.

We will not refactor our chapter driver to multi-unit yet, because the added code distracts from the other teaching goals. But you should know that this is where the driver would go. Supporting multiple units is one of the first extensions real drivers need.

27.13.2 Ioctl Surface for Runtime Configuration

With multiple units comes the need to configure them at runtime. You do not want to compile a new module every time you want a second unit or a different size. The answer is an ioctl on a control device.

`md(4)` follows this pattern. There is a single `/dev/mdctl` device, and `mdconfig(8)` talks to it with ioctls. `MDIO-ATTACH` creates a new unit with a specified size and backing type. `MDIOCDETACH` destroys a unit. `MDIOQUERY` reads the state of a unit. `MDIOCRESIZE` changes the size.

For a driver of any sophistication, this is the right place to invest. Compile-time configuration via macros is fine for a toy. Runtime configuration via ioctls is what real administrators want.

If you were to add this to our driver, you would:

1. Create a `cdev` for the control device using `make_dev`.
2. Implement `d_ioctl` on the `cdev`, switching on a small set of ioctl numbers you define.
3. Write a user-space tool that issues the ioctls.

This is a substantial addition, which is why we mention it here without implementing it. Chapter 28 and later chapters will revisit this pattern.

27.13.3 Splitting the Source File

At some point, a driver outgrows a single file. The usual decomposition for a FreeBSD storage driver is roughly:

- `driver_name.c`: the public module entry, ioctl dispatch, and attach/detach wiring.
- `driver_name_bio.c`: the strategy function and BIO path.
- `driver_name_backing.c`: the backing-store implementation.
- `driver_name_util.c`: small helpers, validation, and debug printing.

- `driver_name.h`: the shared header that declares the softc, enums, and function prototypes.

The Makefile is updated to list all of them in `SRCS`, and the build system handles the rest. This is the shape of `md(4)`, of `ata(4)`, and of most substantial drivers in the tree.

We will keep our driver in one file for the chapter. But when the challenges or your own extensions push it past, say, 500 lines, a decomposition like the above is the right move. Readers who want a concrete example should look at `/usr/src/sys/dev/ata/`, which splits a complex driver across many files along clean lines.

27.13.4 Versioning

A storage driver has several kinds of versioning to care about.

Module version, declared with `MODULE_VERSION(myblk, 1)`. This is a monotonically increasing integer that other modules or userland tools can check. Bump it whenever you change the module's external behaviour in a way that cannot be detected from the code.

Disk ABI version, encoded in `DISK_VERSION`. This is the version of the `g_disk` interface that your driver was compiled against. If the kernel's `g_disk` changes incompatibly, it increments the version, and a driver compiled against the old version will fail to register. You do not set this directly; you pass the `DISK_VERSION` macro through `disk_create`, and it picks up whatever version the compile found in `geom_disk.h`. You should recompile drivers against the kernel you are targeting.

On-disk format version, for drivers that have any on-disk metadata. If your driver stamps a magic number and a version into a reserved sector, you must handle upgrades. Our driver has no on-disk format, so this does not apply yet, but it would if we added a proper backing-store header.

Ioctl number version. Once you define ioctls, their numbers are part of the userland ABI. Changing them breaks older userland tools. Use `_IO`, `_IOR`, `_IOW`, `_IOWR` with stable magic letters, and do not repurpose numbers.

For our chapter driver, the only version we care about right now is the module version. But keeping these four kinds of versioning in mind saves pain later.

27.13.5 Debugging and Observability Helpers

As the driver grows, you will want to observe its state more richly than `dmesg` alone allows. Three tools are worth introducing now.

sysctl nodes. FreeBSD's `sysctl(3)` framework lets a module publish read-only or read-write variables that user-space tools can query. You create a tree under a chosen name and attach values to it. The pattern is standard; in roughly ten lines of code you can expose the number of BIOs serviced, the number of bytes read and written, and the current media size.

```
SYSCTL_NODE(_dev, OID_AUTO, myblk, CTLFLAG_RD, 0,  
    "myblk driver parameters");  
static u_long myblk_reads = 0;  
SYSCTL_ULONG(_dev_myblk, OID_AUTO, reads, CTLFLAG_RD, &myblk_reads,  
    0, "Number of BIO_READ requests serviced");
```

Devstat. We are already using this through `g_disk`. It gives `iostat` and `gstat` their data. No further work needed.

DTrace probes. The SDT framework lets a module define static DTrace probes that incur zero cost when the probe is not being watched. These are especially useful in the BIO path because they let you see live request flow without recompiling.

```
#include <sys/sdt.h>
SDT_PROVIDER_DECLARE(myblk);
SDT_PROBE_DEFINE3(myblk, , strategy, request,
    "int" /* cmd */, "off_t" /* offset */, "size_t" /* length */);

/* inside myblk_strategy: */
SDT_PROBE3(myblk, , strategy, request,
    bp->bio_cmd, bp->bio_offset, bp->bio_bcount);
```

You can then watch with `dtrace -n 'myblk::strategy:request {...}'`.

For the chapter driver we will not add all of this, but these are the patterns you should reach for as the driver grows.

27.13.6 Naming Stability

One habit that is easy to overlook: do not rename things casually. The name `myblk` is in the device node, in the module version record, in the `devstat` name, possibly in `sysctl` nodes, in DTrace probes, and in documentation. Renaming it cascades through all of those. For a project driver, pick a name you can live with forever. `md`, `ada`, `nvd`, `zvol`, and other storage drivers have kept their names for years because renaming is an ABI-affecting change for user-space tooling.

27.13.7 Keeping the Teaching Driver Simple

Everything in this section is a direction your driver might grow in. None of it is required for the teaching driver in this chapter. We are pointing at the directions so that you can recognise them when you see them in real driver source, and so that when you extend your own driver you do not have to invent these patterns from scratch.

The companion `myfirst_blk.c` remains a single file at the end of this chapter. Its README documents the extension points, and the challenge exercises add some of them. Beyond that, you are free to keep extending it, and every extension you make will use these patterns in some form.

27.13.8 A Short Design Patterns Recap

At this point we have accumulated enough patterns that listing them helps. When you start your next storage driver, these are the patterns to reach for.

The `softc` pattern. One per-instance struct to hold everything the driver needs. Pointed at by `d_drv1`. Retrieved inside callbacks via `bp->bio_disk->d_drv1`.

The attach/detach pair. Attach allocates, initialises, and registers. Detach reverses the sequence. Both must be idempotent.

The switch-and-biodone pattern. Every strategy function switches on `bio_cmd`, services each command, sets `bio_resid`, and calls `biodone` exactly once.

The defensive bounds check. Validate offset and length against media size, using subtraction to avoid overflow.

The coarse lock pattern. A single mutex around the hot path is often enough for a teaching driver. Split it only when performance demands.

The reverse-order teardown. Free resources in the opposite order of allocation.

The null-after-free pattern. After freeing a pointer, set it to `NULL`. Catches double-frees.

The single cleanup label. In attach functions that can fail, all failures jump to a single cleanup label that unwinds state so far.

The versioned ABI. Pass `DISK_VERSION` to `disk_create`. Declare `MODULE_VERSION`. Use `MODULE_DEPEND` on every kernel module you rely on.

The deferred-work pattern. Work that must block (like `vnode` I/O) belongs in a worker thread, not in `d_strategy`.

The observability-first habit. Add `printf`, `sysctl`, or `DTrace` probes as you build. Observability retrofitted late is harder than observability designed in.

These are not exhaustive, but they are the patterns you will use most often. Each of them appears somewhere in our driver, and each of them appears throughout the real FreeBSD storage code.

27.13.9 Wrapping Up Section 9

A maturing storage driver grows in predictable directions: multi-unit support, runtime configuration through `ioctl`s, multiple source files, and stable versioning of every interface it exposes. None of this has to appear in the first version. Knowing where the growth will happen lets you make early choices that do not need to be undone later.

We have now covered every concept the chapter set out to teach. Before the hands-on labs, one more topic deserves a dedicated section, because it will repay you many times over as a driver author: observing a running storage driver. In the next section, we will look at the tools FreeBSD gives you for watching your driver in real time and for measuring its behaviour in a disciplined way.

27.14 Section 10: Observability and Measuring Your Driver

Writing a storage driver is mostly a matter of getting the structure right. Once the structure is right, the driver just runs. But for the structure to stay right, you must be able to observe what is happening while the driver runs. You will want to know how many BIOs per second are hitting the strategy function, how long each one takes, how the latency distribution looks, how much memory the backing store consumes, whether any BIOs are getting retried, and whether any path is leaking completion.

FreeBSD gives you a remarkable set of tools for this, many of which we have already used casually. In this section we will walk through the most important ones in turn, with the goal of making you comfortable enough to reach for the right tool when the next strange symptom appears.

27.14.1 gstat

`gstat` is the first tool to reach for. It updates a per-provider view of I/O activity in real time, and it shows you exactly what is happening at the GEOM layer.

```
# gstat -I 1
dT: 1.002s  w: 1.000s
L(q)  ops/s    r/s    kBps    ms/r    w/s    kBps    ms/w    %busy Name
  0    117      0      0      0.0    117    468     0.1    1.1| ada0
  0      0      0      0      0.0      0      0     0.0    0.0| myblk0
```

The columns, from left to right, are:

- `L(q)`: queue length. The number of BIOs currently outstanding on this provider.
- `ops/s`: total operations per second, regardless of direction.
- `r/s`: reads per second.
- `kBps` (for reads): read throughput in kilobytes per second.
- `ms/r`: average read latency, in milliseconds.
- `w/s`: writes per second.
- `kBps` (for writes): write throughput in kilobytes per second.
- `ms/w`: average write latency, in milliseconds.
- `%busy`: the percentage of time the provider was not idle.
- `Name`: the provider name.

For a driver you have just built, `gstat` tells you at a glance whether the kernel is sending traffic to your device and how your driver is performing relative to real disks. If the numbers look wildly different from what you expect, you have a starting point for investigation.

`gstat -p` shows only providers (the default). `gstat -c` shows only consumers, which is less useful for driver debugging. `gstat -f <regex>` filters by name. `gstat -b` batches the output one screen at a time instead of refreshing in place.

27.14.2 iostat

`iostat` has a more traditional style but provides the same underlying data. It is useful when you want a text log rather than an interactive display.

```
# iostat -x myblk0 1

                extended device statistics
device      r/s      w/s      kr/s      kw/s  ms/r  ms/w  ms/o  ms/t qlen  %b
myblk0         0     128         0     512   0.0   0.1   0.0   0.1    0    2
myblk0         0     128         0     512   0.0   0.1   0.0   0.1    0    2
```

`iostat` can watch multiple devices at once and can be redirected to a log file for later analysis. For quick live views, `gstat` is usually better.

27.14.3 diskinfo

`diskinfo` is less about live traffic and more about static properties. We have already used it to confirm our media size.

```
# diskinfo -v /dev/myblk0
/dev/myblk0
    512          # sectorsize
 33554432      # mediasize in bytes (32M)
   65536      # mediasize in sectors
     0         # stripesize
     0         # stripeoffset
  myblk0      # Disk ident.
```

`diskinfo -c` runs a timing test, reading a few hundred megabytes and reporting the sustained rate. This is useful for a first-order performance comparison.

```
# diskinfo -c /dev/myblk0
/dev/myblk0
    512          # sectorsize
 33554432      # mediasize in bytes (32M)
   65536      # mediasize in sectors
     0         # stripesize
     0         # stripeoffset
  myblk0      # Disk ident.
```

I/O command overhead:

time to read 10MB block	0.000234 sec	=	0.000 msec/sector
time to read 20480 sectors	0.001189 sec	=	0.000 msec/sector
calculated command overhead		=	0.000 msec/sector

Seek times:

Full stroke:	250 iter in	0.000080 sec =	0.000 msec
Half stroke:	250 iter in	0.000085 sec =	0.000 msec
Quarter stroke:	500 iter in	0.000172 sec =	0.000 msec
Short forward:	400 iter in	0.000136 sec =	0.000 msec
Short backward:	400 iter in	0.000137 sec =	0.000 msec
Seq outer:	2048 iter in	0.000706 sec =	0.000 msec
Seq inner:	2048 iter in	0.000701 sec =	0.000 msec

Transfer rates:

```

outside:      102400 kbytes in  0.017823 sec =  5746 MB/sec
inside:       102400 kbytes in  0.017684 sec =  5791 MB/sec

```

These numbers are unusually fast because the backing store is RAM. On a real disk they would look very different, and comparing the numbers across devices is often the first diagnosis step for performance problems.

27.14.4 sysctl

`sysctl` is how the kernel exposes its internal variables to user space. Many subsystems publish data through `sysctl`. You can browse the storage-related `sysctls` with:

```

# sysctl -a | grep -i kern.geom
# sysctl -a | grep -i vfs

```

Adding your own `sysctl` tree to your driver, as we discussed in Section 9, lets you expose whatever metrics your driver needs to track, without the ceremony of defining a new tool.

27.14.5 vmstat

`vmstat -m` shows memory allocation by `MALLOC_DEFINE` tag. Our driver uses `M_MYBLK`, so we can see how much memory our driver has allocated.

```

# vmstat -m | grep myblk
      myblk      1  32768K      -      12  32K, 32M

```

The columns are type, number of allocations, current size, protection requests, total requests, and possible sizes. For a driver that holds a 32 MiB backing store, the current size of 32 MiB is exactly what we expect. If it grew over time without an equivalent decrease on unload, we would have a leak.

`vmstat -z` shows zone allocator statistics. Much storage-related state lives in zones (GEOM providers, BIOS, disk structures), and `vmstat -z` is where to look if you suspect GEOM-level leaks.

27.14.6 procstat

`procstat` shows per-thread kernel stacks. It is indispensable when something is stuck.

```

# procstat -kk -t $(pgrep -x g_event)
  PID   TID COMM                TDNAME                KSTACK
  4 100038 geom                -                mi_switch sleepq_switch ...

```

If the `g_event` thread is sleeping, the GEOM layer is idle. If it is stuck in a function with your driver's name on its stack, you have a BIO that is not completing.

```
# procstat -kk $(pgrep -x kldload)
```

If `kldload` or `kldunload` is stuck, this shows you exactly where. Most often the culprit is a `disk_destroy` waiting for BIOs to drain.

27.14.7 DTrace for the Block Layer

We introduced DTrace briefly in Section 6 and in Lab 7. Here let us go a little deeper, because DTrace is the single most effective tool for understanding live storage behaviour.

The Function Boundary Tracing (FBT) provider lets you place probes on the entry and return of nearly any kernel function. For our driver's strategy function, the probe name is `fbt::myblk_strategy:entry` for the entry and `fbt::myblk_strategy:return` for the return.

A simple one-liner that counts BIOs by command:

```
# dtrace -n 'fbt::myblk_strategy:entry \
    { @c[args[0]->bio_cmd] = count(); }'
```

When you interrupt the script (with `Ctrl-C`), it prints a count per command value. `BIO_READ` is 1, `BIO_WRITE` is 2, `BIO_DELETE` is 3, `BIO_GETATTR` is 4, and `BIO_FLUSH` is 5. (The exact numbers are in `/usr/src/c/sys/sys/bio.h`.)

A latency histogram:

```
# dtrace -n '
fbt::myblk_strategy:entry { self->t = timestamp; }
fbt::myblk_strategy:return /self->t/ {
    @lat = quantize(timestamp - self->t);
    self->t = 0;
}'
```

This gives you a log-scale histogram of how long each strategy-function execution took. For our in-memory driver, most buckets should be in the hundreds-of-nanoseconds range; anything in the millisecond range for an in-memory driver is suspicious.

A breakdown of I/O size:

```
# dtrace -n 'fbt::myblk_strategy:entry \
    { @sz = quantize(args[0]->bio_bcount); }'
```

This shows you the distribution of BIO sizes. For a UFS-backed filesystem, you should see peaks at 4 KiB, 8 KiB, 16 KiB, and 32 KiB. For a raw `dd` with `bs=1m`, you should see a peak at 1 MiB (or the `MAXPHYS` cap, whichever is smaller).

DTrace is extraordinarily capable. The one-liners above barely scratch the surface. Two books to pick up, if you want to go deeper, are the original Sun “DTrace Guide” and Brendan Gregg’s “DTrace Book”. Both are older than FreeBSD 14.3 but the fundamentals still apply.

27.14.8 kgdb and Crash Dumps

When your driver panics, FreeBSD can capture a crash dump. Configure the dump device in `/etc/rc.conf` (typically `dumpdev="AUTO"`) and verify with `dumpon`.

After a panic, reboot. `/var/crash/vmcore.last` (a symlink) points to the most recent dump. `kgdb /boot/kernel/kernel /var/crash/vmcore.last` opens the dump for inspection. Useful commands inside `kgdb`:

- `bt`: backtrace of the thread that panicked.
- `info threads`: list all threads in the crashed system.
- `thread N` then `bt`: backtrace of thread `N`.
- `print *var`: inspect a variable.
- `list function`: show source around a function.

If you have compiled your module with debugging symbols (the default for most kernel configurations), `kgdb` can show you source-level variables in your own code. This is a transformative capability once you get used to it.

27.14.9 ktrace

`ktrace` is a user-space-oriented tool, but it can be useful for storage debugging when you want to see exactly what system calls a user program is making. If `newfs_ufs` is behaving oddly, you can trace it:

```
# ktrace -f /tmp/newfs.ktr newfs_ufs /dev/myblk0
# kdump /tmp/newfs.ktr | head -n 50
```

The resulting trace shows the sequence of system calls, their arguments, and their results. For storage tools, this reveals exactly which `ioctl`s are being issued and which file descriptors are being opened.

27.14.10 dmesg and the Kernel Log

The humble `dmesg` is often the fastest way to diagnose a problem. Our driver prints to it on load and unload. The kernel prints to it on many other events, including GEOM class creation, access-count violations, and panics that the system recovers from.

Pro tip: redirect `dmesg -a` to a file at the start of each lab session. If something goes wrong you will have a complete log.

```
# dmesg -a > /tmp/session.log
# # ... work ...
# dmesg -a > /tmp/session-final.log
# diff /tmp/session.log /tmp/session-final.log
```

This gives you a precise log of what the kernel reported during your session.

27.14.11 A Simple Measurement Recipe

Here is a recipe you can use to produce a one-page performance profile of your driver.

1. Load the driver.
2. Run `diskinfo -c /dev/myblk0` and record the three transfer-rate numbers.
3. Format the device and mount it.
4. In one terminal, start `gstat -I 1 -f myblk0 -b` redirected to a file.
5. In another terminal, run `dd if=/dev/zero of=/mnt/myblk/stress bs=1m count=128`.
6. Stop `gstat` after `dd` completes and save the log.
7. Parse the log with `awk` to extract the peak ops/s, the peak throughput, and the average latency.
8. Unmount and unload.

This recipe scales. For a real driver you would automate it, run it on a matrix of block sizes, and plot the results. For a teaching driver, running it once or twice gives you a feeling for the numbers and a baseline to compare against after future changes.

27.14.12 Comparing Against md(4)

One of the most useful exercises is to load `md(4)` in the same configuration as your driver and compare.

```
# mdconfig -a -t malloc -s 32m
md0
# diskinfo -c /dev/md0
```

The numbers will likely be within a small factor of your driver's. If they are very different, something interesting is going on. The usual differences are:

- `md(4)` uses a worker thread that receives BIOs from the strategy function and processes them in a separate context. This adds a small amount of latency per BIO but allows higher concurrency.
- `md(4)` uses page-at-a-time backing, which is slightly slower per byte for sequential I/O but scales to much larger sizes.
- `md(4)` supports more BIO commands and attributes than our driver.

Comparing against `md(4)` is a form of debugging: if your driver is much slower or much faster than `md(4)` on the same workload, either you have done something unusual, or you have uncovered a difference worth understanding.

27.14.13 Wrapping Up Section 10

Observability is not an afterthought. For a storage driver, it is how you keep your bearings. `gstat`, `iostat`, `diskinfo`, `sysctl`, `vmstat`, `procstat`, and `DTrace` are the tools you will reach for most often. `kgdb` and crash dumps are your backstop when things go catastrophically wrong.

Learn these tools now, while the driver is simple, because they will be the same tools you use when the driver is complex. A developer who can observe a running driver is much more effective than one who can only read source.

We have now covered every concept the chapter set out to teach, plus observability and measurement. Before we move on to the hands-on labs, let us spend some time reading real FreeBSD source. The case studies that follow anchor everything we have learned in code from the tree.

27.15 Case Studies in Real FreeBSD Storage Code

Reading production driver source is the fastest way to internalise patterns. In this section we will walk through excerpts from three real drivers in `/usr/src/sys/`, with commentary that points out what each excerpt is doing and why. The excerpts are short on purpose; we will not read every line of each driver. We will pick the lines that matter.

Open the files alongside the text and follow along. The point is for you to see the same patterns in our driver reappearing in real drivers, under different names and with different constraints.

27.15.1 Case Study 1: `g_zero.c`

`g_zero.c` is the simplest GEOM class in the tree. It is a read-always-zero, write-discard provider, with no real backing store and no real work to do. Its purpose is to give you a standard “null disk” you can test against. It is also an excellent teaching reference because it exercises the full `g_class` API in fewer than 150 lines.

Let us look at its strategy function, called `g_zero_start`.

```
static void
g_zero_start(struct bio *bp)
{
    switch (bp->bio_cmd) {
        case BIO_READ:
            bzero(bp->bio_data, bp->bio_length);
            g_io_deliver(bp, 0);
            break;
        case BIO_WRITE:
            g_io_deliver(bp, 0);
            break;
        case BIO_GETATTR:
        default:
            g_io_deliver(bp, EOPNOTSUPP);
            break;
    }
}
```

Three behaviours, with `BIO_GETATTR` intentionally folded into the default case. Reads get zeroed. Writes are silently accepted. Anything else, including attribute queries, gets `EOPNOTSUPP`. The real `/usr/src/sys/geom/zero/g_zero.c` also handles `BIO_DELETE` in the successful-write path; our simplified excerpt above drops that case so you can see the shape clearly. Notice the call to `g_io_deliver` rather than `biodone`. That is because

`g_zero` is a class-level GEOM module, not a `g_disk` module. `g_io_deliver` is the class-level completion call; `biodone` is the `g_disk` wrapper.

If you re-read our driver's strategy function side by side with this, you will see the same structure: a switch on `bio_cmd`, a case for each supported operation, a default error path. Our driver has more cases and it has a real backing store, but the shape is identical.

The `init` function that `g_zero` registers with the class is also small:

```
static void
g_zero_init(struct g_class *mp)
{
    struct g_geom *gp;

    gp = g_new_geomf(mp, "gzero");
    gp->start = g_zero_start;
    gp->access = g_std_access;
    g_new_providerf(gp, "%s", gp->name);
    g_error_provider(g_provider_by_name(gp->name), 0);
}
```

When the `g_zero` module is loaded, this runs. It creates a new geom under the class, points the start method at the strategy function, uses the standard access handler, and creates a provider. That is everything it takes to expose `/dev/gzero`.

In our driver, `g_disk` does the equivalent of all this when `disk_create` is called. You can see here, once more, what `g_disk` is abstracting away. For most disk drivers that is a good trade; for `g_zero`, which does not want `g_disk`'s disk-specific features, using the class API directly is the better fit.

27.15.2 Case Study 2: `md.c`, the Malloc Strategy Function

`md(4)` is a memory-disk driver with several backing types. The malloc backing type is the closest to our driver, and its strategy function is worth reading in detail.

Here is a simplified version of what happens when `md(4)`'s worker thread picks up a BIO for a `MD_MALLOC`-type disk. (In real `md(4)`, this is the function `mdstart_malloc`.)

```
static int
mdstart_malloc(struct md_s *sc, struct bio *bp)
{
    u_char *dst, *src;
    off_t offset;
    size_t resid, len;
    int error;

    error = 0;
```

```

    resid = bp->bio_length;
    offset = bp->bio_offset;

    switch (bp->bio_cmd) {
    case BIO_READ:
        /* find the page that contains offset */
        /* copy len bytes out of it */
        /* advance, repeat until resid == 0 */
        break;
    case BIO_WRITE:
        /* find the page that contains offset */
        /* allocate it if not allocated yet */
        /* copy len bytes into it */
        /* advance, repeat until resid == 0 */
        break;
    case BIO_DELETE:
        /* free pages in the range */
        break;
    }

    bp->bio_resid = 0;
    return (error);
}

```

The key difference from our driver is the page-at-a-time backing. `md(4)` does not allocate one big buffer. It allocates 4 KiB pages on demand and indexes them through a data structure inside the `softc`. The benefit is that memory disks can be much larger than a single contiguous `malloc` would allow, and sparse regions (never written) consume no memory.

The cost is that every BIO may span multiple pages, so the strategy function has to loop. Each iteration copies `len` bytes into the current page, decrements `resid`, advances `offset`, and either exits when `resid` hits zero or moves on to the next page.

Our driver avoids this complexity at the cost of supporting only contiguous backing, which is fine up to a few tens of megabytes but not beyond.

If you wanted to extend our driver to match `md(4)`'s scale, the page-at-a-time pattern is the direction you would go. It is straightforward once you have `md(4)` in front of you as a reference.

27.15.3 Case Study 3: `md.c`, the Module Load Path

Another piece of `md(4)` worth studying is how it bootstraps its class and sets up the control device.

```

static void
g_md_init(struct g_class *mp __unused)

```

```

{
    /*
     * Populate sc_list with pre-loaded memory disks
     * (preloaded kernel images, ramdisks from boot, etc.)
     */
    /* ... */

    /*
     * Create the control device /dev/mdctl.
     */
    status = make_dev_p(MAKEDEV_CHECKNAME | MAKEDEV_WAITOK,
        &status_dev, &mdctl_cdevsw, 0, UID_ROOT, GID_WHEEL,
        0600, MDCTL_NAME);
    /* ... */
}

```

The `g_md_init` function runs once per kernel boot, when the `md(4)` class is first instantiated. It handles any memory disks that the loader preloaded into memory (so that the kernel can boot from a memory-disk root) and it creates the control device `/dev/mdctl` through which `mdconfig` will later talk to the driver.

Compare this to our loader, which is a simple `moduledata_t` that calls `disk_create` directly. `md(4)` does not create any memory disks by default. It only creates them in response to preload events or in response to `MDIOCATTACH` ioctls on the control device.

The pattern here is generalisable. If you want a storage driver that creates units on demand rather than at load time, you:

1. Register the class (or, for `g_disk`-based drivers, set up the infrastructure).
2. Create a control device with a `cdevsw` that supports ioctls.
3. Implement create, destroy, and query ioctls.
4. Write a user-space tool that talks to the control device.

`md(4)` is the canonical example. Other drivers, like `geli(4)` and `gmirror(4)`, use a slightly different pattern because they are GEOM transformation classes rather than disk drivers, but the overall shape is similar.

27.15.4 Case Study 4: The newbus Side of a Real Storage Driver

For contrast, let us look briefly at how a real hardware-backed storage driver attaches. The `ada(4)` driver, for example, is a CAM-based ATA driver. Its attach path is not directly visible as a single function, because CAM mediates between the driver and the hardware, but the end of the chain looks like this (abbreviated from `/usr/src/sys/cam/ata/ata_da.c`):

```

static void
adaregister(struct cam_periph *periph, void *arg)
{
    struct ada_softc *softc;

```

```

/* ... */

softc->disk = disk_alloc();
softc->disk->d_open = adaopen;
softc->disk->d_close = adaclose;
softc->disk->d_strategy = adastrategy;
softc->disk->d_getattr = adagetattr;
softc->disk->d_dump = adadump;
softc->disk->d_gone = adadiskgonecb;
softc->disk->d_name = "ada";
softc->disk->d_drv1 = periph;
/* ... */
softc->disk->d_unit = periph->unit_number;
/* ... */
softc->disk->d_sectorsize = softc->params.secsz;
softc->disk->d_mediasize = ...;
/* ... */

disk_create(softc->disk, DISK_VERSION);
/* ... */
}

```

The structure is identical to ours: fill in a struct `disk` and call `disk_create`. The differences are:

- `d_strategy` is `adastrategy`, which translates BIOs into ATA commands and issues them to the controller via CAM.
- `d_dump` is implemented, because `ada(4)` supports kernel crash dumps. Our driver does not implement this.
- The fields like `d_sectorsize` and `d_mediasize` come from hardware probing, not from macros.

From `g_disk`'s perspective, however, `ada0` and our `myblk0` are the same kind of thing. Both are disks. Both receive BIOs. Both are completed with `biodone`. The difference is in where the bytes actually go.

This is the uniformity that `g_disk` provides. Your driver can choose any backing technology, and as long as it fills in struct `disk` correctly, it looks like any other disk to the rest of the kernel.

27.15.5 Takeaways from the Case Studies

Three patterns become clearer after reading these excerpts.

First, the strategy function is always a switch on `bio_cmd`. The cases vary, but the switch is always there. Memorise this pattern: incoming BIO -> switch -> case per command -> completion. It is the heart of every storage driver.

Second, `g_disk` drivers are structurally identical at the registration level. Whether the driver is a RAM disk or a real SATA drive, the registration code looks the same. The differences are in what happens when the BIO arrives.

Third, more sophisticated drivers enqueue work to a dedicated thread. Our driver does not, because it can do its work synchronously in any thread. Drivers that do slow or blocking work must enqueue, because strategy functions run in the caller's thread context.

With these patterns in mind, you can now read almost any storage driver in the FreeBSD tree and follow its overall structure, even if specific details about hardware or sub-frameworks require further study.

We have now covered every concept the chapter set out to teach, plus observability, measurement, and a few real case studies. In the next part of the chapter, we will put this knowledge to work through hands-on labs. The labs build on the driver you have been writing and on the skills you have been using, and they take you from the minimum working driver through persistence, mount, and cleanup scenarios. Let us begin.

27.16 Hands-On Labs

Each lab is a self-contained checkpoint. They are designed to be done in order, but you can revisit any lab later if you want to practise a specific skill. Every lab has a companion folder under `examples/part-06/ch27-storage-vfs/`, which contains the reference implementation and the artifacts you would produce if you typed the code by hand.

Before you start, make sure you have the chapter's driver building cleanly against your local kernel. From a fresh checkout of the examples tree:

```
# cd examples/part-06/ch27-storage-vfs
# make
# ls myblk.ko
myblk.ko
```

If that works, you are ready. If not, revisit the Makefile and the advice in Chapter 26, section “Your Build Environment”.

27.16.1 Lab 1: Explore GEOM on a Running System

Goal. Build comfort with the GEOM inspection tools before you touch any code.

What you do.

On your FreeBSD 14.3 system, run the following commands and take notes in your lab logbook.

```
# geom disk list
# geom part show
# geom -t | head -n 40
# gstat -I 1
# diskinfo -v /dev/ada0    # or whatever your primary disk is called
```

What you look for.

Identify every DISK class geom. For each, note its provider name, its media size, its sector size, and its current mode. Notice which geoms have partitioning layers on top and which do not. If your system has geli or zfs, notice the chain of classes.

Stretch question. Which of your geoms have non-zero access counts right now? Which ones are free? What would happen if you tried to run newfs_ufs on each?

Reference implementation. examples/part-06/ch27-storage-vfs/lab01-explore-geom/README.md contains a suggested walkthrough and a sample output transcript from a typical system.

27.16.2 Lab 2: Build the Skeleton Driver

Goal. Get the Section 3 skeleton driver compiling and loading on your system.

What you do.

Copy examples/part-06/ch27-storage-vfs/lab02-skeleton/myfirst_blk.c and its Makefile into a working directory. Build it.

```
# cp -r examples/part-06/ch27-storage-vfs/lab02-skeleton /tmp/myblk
# cd /tmp/myblk
# make
```

Load the module.

```
# kldload ./myblk.ko
# dmesg | tail -n 2
# ls /dev/myblk0
# geom disk list myblk0
```

Unload it.

```
# kldunload myblk
# ls /dev/myblk0
```

What you look for.

Confirm that the kernel printed your myblk: loaded message. Confirm that /dev/myblk0 appeared. Confirm that geom disk list reported the expected media size. Confirm that the node disappeared after unload.

Stretch question. What happens if you try newfs_ufs -N /dev/myblk0 with the skeleton driver? Can you read the output? Why does the dry run succeed even though real writes would fail?

27.16.3 Lab 3: Implement the BIO Handler

Goal. Add the working strategy function from Section 5 to the skeleton driver.

What you do.

Starting from the skeleton, implement `myblk_strategy` with support for `BIO_READ`, `BIO_WRITE`, `BIO_DELETE`, and `BIO_FLUSH`. Allocate the backing buffer in `myblk_attach_unit` and free it in `myblk_detach_unit`.

Build, load, and test.

```
# dd if=/dev/zero of=/dev/myblk0 bs=4096 count=16
# dd if=/dev/myblk0 of=/dev/null bs=4096 count=16
# dd if=/dev/random of=/dev/myblk0 bs=4096 count=16
# dd if=/dev/myblk0 of=/tmp/a bs=4096 count=16
# dd if=/dev/myblk0 of=/tmp/b bs=4096 count=16
# cmp /tmp/a /tmp/b
```

What you look for.

The last `cmp` must succeed with no output. If it prints `differ: byte N`, your strategy function is racing or returning stale data.

Stretch question. Put a `printf` in the strategy function that reports `bio_cmd`, `bio_offset`, and `bio_bcount`. Run `dd if=/dev/myblk0 of=/dev/null bs=1m count=1` and look at `dmesg`. What size did `dd` actually issue? Do you see fragmentation?

Reference implementation. `examples/part-06/ch27-storage-vfs/lab03-bio-handler/myfirst_blk.c`.

27.16.4 Lab 4: Increase Size and Mount UFS

Goal. Increase the backing store to 32 MiB and mount UFS on the device.

What you do.

Change `MYBLK_MEDIASIZE` to `(32 * 1024 * 1024)` and rebuild. Load the module. Format and mount.

```
# newfs_ufs /dev/myblk0
# mkdir -p /mnt/myblk
# mount /dev/myblk0 /mnt/myblk
# echo "hello" > /mnt/myblk/greeting.txt
# cat /mnt/myblk/greeting.txt
# umount /mnt/myblk
# mount /dev/myblk0 /mnt/myblk
# cat /mnt/myblk/greeting.txt
# umount /mnt/myblk
# kldunload myblk
```

What you look for.

Verify that the file survives an unmount-and-remount. Verify that the access counts in `geom disk list` are zero after unmount. Verify that `kldunload` succeeds cleanly.

Stretch question. Watch `gstat -I 1` while running `dd if=/dev/zero of=/mnt/myblk/big bs=1m count=16`. Can you see the writes arrive in bursts? What size are the individual BIOs? Hint: UFS's default block size is typically 32 KiB on a filesystem this small.

Reference implementation. `examples/part-06/ch27-storage-vfs/lab04-mount-ufs/myfirst_blk.c`.

27.16.5 Lab 5: Observing Real Cross-Reload Persistence with md(4)

Goal. Confirm experimentally that cross-reload persistence requires external backing, as Section 7 argued, by using `md(4)`'s vnode mode as a control.

What you do.

First, demonstrate that our RAM-backed `myblk` loses its filesystem on reload. Load, format, mount, write, unmount, unload, reload, mount again, and observe the empty filesystem.

```
# kldload ./myblk.ko
# newfs_ufs /dev/myblk0
# mount /dev/myblk0 /mnt/myblk
# echo "not persistent" > /mnt/myblk/token.txt
# umount /mnt/myblk
# kldunload myblk
# kldload ./myblk.ko
# mount /dev/myblk0 /mnt/myblk
# ls /mnt/myblk
```

The `ls` should show an empty or fresh UFS directory; the `token.txt` is gone because the backing buffer was reclaimed by the kernel when the module unloaded.

Now do the same sequence with `md(4)`'s vnode backend, which uses a real file on disk:

```
# truncate -s 64m /var/tmp/mdimage.img
# mdconfig -a -t vnode -f /var/tmp/mdimage.img -u 9
# newfs_ufs /dev/md9
# mount /dev/md9 /mnt/md
# echo "persistent" > /mnt/md/token.txt
# umount /mnt/md
# mdconfig -d -u 9
# mdconfig -a -t vnode -f /var/tmp/mdimage.img -u 9
# mount /dev/md9 /mnt/md
# cat /mnt/md/token.txt
persistent
```

What you look for.

The first sequence loses the file; the second preserves it. The difference is that `md9` is backed by a real file on disk, whose state survives regardless of what happens inside the kernel. Contrast this with `myblk0`, which is backed by kernel heap that disappears on `kldunload`.

Stretch question. Read the `MD_VNODE` branch of `mdstart_vnode` in `/usr/src/sys/dev/md/md.c`. Identify where the vnode reference is stored (hint: it lives on the per-unit struct `md_s`, not a module-scope global). Explain in your own words why that design is what lets the backing survive module lifecycles.

Reference implementation. `examples/part-06/ch27-storage-vfs/lab05-persistence/README.md` walks through both sequences and their diagnostic output.

27.16.6 Lab 6: Safe Unmount Under Load

Goal. Verify that the teardown path handles an active filesystem correctly.

What you do.

Load the module, format it, mount it. In one terminal, start a stress loop.

```
# while true; do dd if=/dev/urandom of=/mnt/myblk/stress bs=4k \
    count=512 2>/dev/null; sync; done
```

In another terminal, attempt to unload.

```
# kldunload myblk
kldunload: can't unload file: Device busy
```

Stop the stress loop. Unmount. Unload.

What you look for.

The initial unload must fail gracefully. After unmount, the final unload must succeed. `dmesg` must not show any kernel warnings.

Stretch question. Instead of killing the stress loop, try `umount /mnt/myblk` directly. Does UFS let you unmount while writes are in flight? What is the error, and what does it mean?

Reference implementation. `examples/part-06/ch27-storage-vfs/lab06-safe-unmount/` includes a test script that performs the sequence above and reports failures.

27.16.7 Lab 7: Observing BIO Traffic with DTrace

Goal. Use DTrace to see the BIO path as it happens.

What you do.

With the driver loaded and a filesystem mounted, run the following DTrace one-liner in one terminal:

```
# dtrace -n 'fbt::myblk_strategy:entry { \
    printf("cmd=%d off=%lld len=%u", \
        args[0]->bio_cmd, args[0]->bio_offset, \
        args[0]->bio_bcount); \
    @count[args[0]->bio_cmd] = count(); \
}'
```

In another terminal, create and read files on the mounted filesystem.

What you look for.

Note which BIO commands you see and in what quantities. Note the typical offsets and lengths. Compare the patterns from `dd` traffic versus `cp` traffic versus `tar` traffic. Note how `cp` or `mv` might produce very different BIO patterns depending on what the buffer cache decides to flush.

Stretch question. Issue `sync` while the DTrace is running. What BIO commands does `sync` cause? What about `newfs_ufs`?

Reference implementation. `examples/part-06/ch27-storage-vfs/lab07-dtrace/README.md` with sample DTrace output and notes.

27.16.8 Lab 8: Adding a `getattr` Attribute

Goal. Implement a `d_getattr` callback that responds to `GEOM::ident`.

What you do.

Add the `myblk_getattr` function from Section 5 to the driver, and register it on the disk before `disk_create`. Rebuild, reload, and check `diskinfo -v /dev/myblk0`.

What you look for.

The `ident` field should now show `MYBLK0` instead of `(null)`.

Stretch question. What other attributes might a filesystem query? Look at `/usr/src/sys/geom/geom.h` for named attributes like `GEOM::rotation_rate`. Try implementing that too.

Reference implementation. `examples/part-06/ch27-storage-vfs/lab08-getattr/myfirst_blk.c`.

27.16.9 Lab 9: Exploring `md(4)` for Comparison

Goal. Read a real FreeBSD storage driver and identify the patterns we have used.

What you do.

Open `/usr/src/sys/dev/md/md.c`. It is a long file. Do not try to read every line. Instead, find and understand these specific things:

1. The `g_md_class` structure at the top of the file.
2. The `struct md_softc`.
3. The `mdstart_malloc` function that handles `BIO_READ` and `BIO_WRITE` for `MD_MALLOC` memory disks.
4. The worker-thread pattern in `md_kthread` (or its equivalent in your version).
5. The `MDIOCATTACH` ioctl handler that creates new units on demand.

Compare each of these to the corresponding code in our driver.

What you look for.

Spot the differences. Where does `md(4)` have features we do not? Where does our driver have the same mechanism in simpler form? Where would you need to extend our driver to add one of `md(4)`'s features?

Reference notes. `examples/part-06/ch27-storage-vfs/lab09-md-comparison/NOTES.md` contains a mapped walkthrough of the relevant sections of `md.c` for FreeBSD 14.3.

27.16.10 Lab 10: Break It On Purpose

Goal. Induce known failure modes so that you can recognise them quickly in real work.

What you do.

Take a clean copy of the completed driver from Lab 8. In separate copies (do not mix the breakages together), introduce the following bugs one at a time, rebuild, load, and observe.

Breakage 1: Forget biodone. Comment out the `biodone(bp)` call in the `BIO_READ` case. Load, mount, and run `cat` on a file. The cat will hang forever. Attempt to kill it with `Ctrl-C`; it may not respond. Use `procstat -kk` on the stuck PID to see where the process is waiting. This is the classic leaked-BIO symptom.

Breakage 2: Free backing before disk_destroy. In `myblk_detach_unit`, swap the order so that `free(sc->backing, ...)` comes before `disk_destroy(sc->disk)`. Load, format, mount, unmount, and try to unload. If no BIO is in flight during the unload window, you will escape unharmed. If any BIO is in flight (use a running `dd` to ensure this), you will panic with a page fault.

Breakage 3: Skip bio_resid. Remove the `bp->bio_resid = 0` line from the `BIO_READ` case. Load, format, mount, and create a file. Read it back. Depending on what garbage was in `bio_resid` at allocation time, the filesystem may report incorrect read sizes and may log errors. Sometimes it works; sometimes it does not. This is the characteristic intermittent failure of a forgotten `bio_resid`.

Breakage 4: Off-by-one bounds. Change the bounds check from `offset > sc->backing_size` to `offset >= sc->backing_size`. This rejects valid reads at the last offset. Load, format, mount. Try to write a file that extends to the very last block. Observe whether UFS notices; whether `dd` notices; what error is reported.

What you look for.

In each case, describe in your logbook what you observed, which tool revealed the problem (`dmesg`, `procstat`, `gstat`, panic trace), and what the fix would be. Then apply the fix and confirm normal operation.

Stretch question. What sequence of commands reliably reproduces each failure? Can you write a shell script that deterministically triggers Breakage 1 or Breakage 2?

Reference notes. `examples/part-06/ch27-storage-vfs/lab10-break-on-purpose/BREAKAGES.md` contains short descriptions and a test script for each failure mode.

27.16.11 Lab 11: Measure Under Varying Block Sizes

Goal. Understand how BIO size affects throughput.

What you do.

With the driver loaded and a filesystem mounted, run `dd` with progressively larger block sizes and time each run.

```
# for bs in 512 4096 32768 131072 524288 1048576; do
    rm -f /mnt/myblk/bench
    time dd if=/dev/zero of=/mnt/myblk/bench bs=$bs count=$((16*1024*1024/bs))
done
```

Record the throughput in each case.

What you look for.

Throughput should increase as block size increases, then plateau at or near `d_maxsize` (typically 128 KiB). Very small block sizes will be dominated by per-BIO overhead.

Stretch question. At what block size does the curve visibly plateau? Why?

27.16.12 Lab 12: Race Test With Two Processes

Goal. Observe how the driver handles simultaneous access from multiple processes.

What you do.

With the driver loaded and a filesystem mounted, run two `dd` processes in parallel writing to different files.

```
# dd if=/dev/urandom of=/mnt/myblk/a bs=4k count=1024 &
# dd if=/dev/urandom of=/mnt/myblk/b bs=4k count=1024 &
# wait
```

Record the combined throughput.

What you look for.

Both writes should complete without corruption. Verify with `md5` or `sha256` on each file. The combined throughput may be slightly less than twice the single-process throughput because of lock contention in our coarse mutex.

Stretch question. Does removing the mutex affect throughput? Does it cause corruption? Why or why not?

27.16.13 A Word on Lab Discipline

Every lab is small, and none of them is an exam. If you get stuck, the reference implementation is there for you to compare against. Do not copy-paste it as your first attempt, though. The copying is not the skill. The skill is in typing, reading, diagnosing, and verifying.

Keep your logbook open. Record what you ran, what you saw, and what surprised you. Storage bugs often repeat across projects, and your future self will thank your current self for the notes.

27.17 Challenge Exercises

The challenge exercises stretch the driver a little further. Each one is scoped to something a beginner can accomplish with the material already covered in the chapter, combined with a thoughtful reading of FreeBSD source. They are not timed. Take your time. Open the source tree. Consult manual pages. Compare your solution to `md(4)` when in doubt.

Every challenge below has a stub folder under `examples/part-06/ch27-storage-vfs/`, but no reference solution is provided. The point is to work through them yourself. Solutions are left as follow-ups that you can compare with peers or post in your study notes.

27.17.1 Challenge 1: Expose a Read-Only Mode

Add a module-load tunable that lets the driver come up in read-only mode. In read-only mode, `BIO_WRITE` and `BIO_DELETE` should fail with `EROFS`. `newfs_ufs` should refuse to format the device, and `mount` without `-r` should refuse to mount it.

Hint. The tunable can be a `sysctl_int` bound to a static variable. `TUNABLE_INT` is another way, used at load time only. Your strategy function can check the variable before dispatching writes. Remember that changing the mode at runtime while a filesystem is mounted is a recipe for corruption; you can either disallow the change or document that the tunable only takes effect at module load.

27.17.2 Challenge 2: Implement a Second Unit

Add support for exactly two units: `myblk0` and `myblk1`. Each should have its own backing store of its own size. Do not try to implement fully dynamic unit allocation; just hardcode two `softcs` and two `attach` calls in the module loader.

Hint. Move the backing allocation, disk allocation, and disk creation into `myblk_attach_unit` parameterised by unit number and size, and call it twice from the loader. Make sure the detach path walks both units.

27.17.3 Challenge 3: Honour `BIO_DELETE` with a `Sysctl` Counter

Extend `BIO_DELETE` handling to also bump a `sysctl` counter that reports total bytes deleted. Verify with `sysctl dev.myblk` while running `fstrim /mnt/myblk` or while `dd` writes and overwrites files.

Hint. UFS by default does not issue `BIO_DELETE`. To see delete traffic, mount with `-o trim`. You can verify the trim flow with your DTrace one-liner from Lab 7.

27.17.4 Challenge 4: Respond to `BIO_GETATTR` for `rotation_rate`

Extend `myblk_getattr` to answer `GEOM::rotation_rate` with `DISK_RR_NON_ROTATING` (defined in `/usr/src/sys/geom/geom_disk.h`). Verify with `gpart show` and `diskinfo -v` that the device reports as non-rotating.

Hint. The attribute is returned as a plain `u_int`. Look at how `md(4)` handles `BIO_GETATTR` for comparable attributes.

27.17.5 Challenge 5: Resizing the Device

Add an `ioctl` that lets user space resize the backing store while nothing is mounted. If a filesystem is mounted, the `ioctl` must fail with `EBUSY`. If the resize succeeds, update `d mediasize` and notify `GEOM` so that `diskinfo` reports the new size.

Hint. Look at `md(4)`'s `MDIOCRESIZE` handling for the pattern. This is a non-trivial challenge; take your time and test with throwaway filesystems. Do not attempt this on any backing that you would be sad to lose.

27.17.6 Challenge 6: A Write Counter and Rate Display

Add per-second write-byte counters, exposed via `sysctl`, and a small user-space shell script that reads the `sysctl` every second and prints a human-readable rate. This is useful for testing and it gives you experience wiring metrics through the kernel's observability machinery.

Hint. Use `atomic_add_long` on the counters. The shell script is a one-liner in `while true` loops.

27.17.7 Challenge 7: A Fixed Pattern Backing Store

Implement a backing-store mode where reads always return a fixed byte pattern and writes are silently discarded. This is similar to `g_zero` but with a configurable pattern byte. It is useful for stress-testing the layers above when you do not care about data content.

Hint. Branch on a mode variable inside the strategy function. Keep the in-memory backing for the normal mode and skip the `memcpy` in pattern mode.

27.17.8 Challenge 8: Write a `mdconfig`-Like Control Utility

Write a small user-space program that talks to a control device on your driver (you will need to add one) and can create, destroy, and query units at runtime. The program should accept command-line flags similar to `mdconfig`.

Hint. This is a substantial challenge. Start with a single `ioctl` that prints "hello" and build up from there. `make_dev` on a `cdev` for your control device, then implement `d_ioctl` on that `cdev`.

27.17.9 Challenge 9: Survive a Simulated Crash

Add a mode where the driver drops every Nth write silently (pretending the write succeeded but actually doing nothing). Use this to test UFS's resilience to lost writes.

Hint. This is a dangerous mode. Only run it on throwaway filesystems. You should be able to reproduce interesting `fsck_ffs` repair scenarios with it. Be ready to explain to yourself why this mode is only safe on pseudo devices you can regenerate from scratch.

27.17.10 Challenge 10: Understand `md(4)` Well Enough to Teach It

Write a one-page explanation of how `md(4)` creates a new unit in response to `MDIOCATTACH`. Cover the `ioctl` path, the softc allocation, the backing-type-specific initialisation, the `g_disk` wiring, and the worker thread creation. This is a reading challenge rather than a coding challenge, but it is one of the most useful exercises you can do to deepen your grasp of the storage stack.

Hint. `/usr/src/sys/dev/md/md.c` and `/usr/src/sbin/mdconfig/mdconfig.c` are the two files to read. Pay attention to the `struct md_ioctl` structure in `/usr/src/sys/sys/mdioctl.h`, because that is the ABI between user space and the kernel.

27.17.11 When to Attempt Challenges

You do not need to do all of them. Pick one or two that speak to something you are curious about or something you can imagine using later. A single carefully-done challenge is worth more than five half-done ones. The reference `md(4)` implementation will be there whenever you want to compare your approach to a production driver.

27.18 Troubleshooting

Storage drivers have a particular family of failure modes. Some of them are obvious as they happen. Others are silent at first and become obvious only after reboot, sometimes with data corruption in between. This section lists the symptoms you are most likely to see while working through the chapter and the labs, along with the usual causes and fixes. Use it as a reference when things go wrong, and read it through at least once before you start, because it is much easier to recognise a failure mode the second time.

27.18.1 `kldload` Succeeds but No `/dev` Node Appears

Symptom. `kldload` returns zero. `kldstat` shows the module loaded. But `/dev/myblk0` does not exist.

Likely causes.

- You forgot to call `disk_create`. The softc is allocated, the disk is allocated, but the disk is not registered with GEOM.
- You called `disk_create` with `d_name` set to a null pointer or an empty string.
- You called `disk_create` with `d_mediasize` set to zero. `g_disk` silently refuses to create a provider with zero size.
- You called `disk_create` before filling in the fields. The framework captures the field values at registration time and does not re-read them later.

Fix. Check the kernel message buffer with `dmesg`. `g_disk` prints a diagnostic when it rejects a registration. Fix the field value and rebuild.

27.18.2 `kldload` Fails with “module version mismatch”

Symptom. Loading the module reports `kldload: can't load ./myblk.ko: No such file or directory` or a more explicit error about version mismatch.

Likely causes.

- You compiled against a different kernel than the one currently running.
- You changed `DISK_VERSION` on your own, which you should never do.
- You forgot `MODULE_VERSION(myblk, 1)`.

Fix. Check `uname -a` and the kernel version your build chose. Recompile against the running kernel.

27.18.3 `diskinfo` Prints the Wrong Size

Symptom. `diskinfo -v /dev/myblk0` prints a size that does not match `MYBLK_MEDIASIZE`.

Likely causes.

- You set `d_mediasize` to the wrong expression. A common off-by-one is setting it to the sector count rather than the byte count.
- You have `MYBLK_MEDIASIZE` defined as something other than `(size * 1024 * 1024)` and the macro is being interpreted differently than you intended. Parenthesise aggressively.

Fix. Print the size in your load message and sanity-check against `diskinfo -v`.

27.18.4 `newfs_ufs` Fails with “Device not configured”

Symptom. `newfs_ufs /dev/myblk0` prints `newfs: /dev/myblk0: Device not configured`.

Likely causes.

- Your strategy function is still the placeholder that returns `ENXIO` for everything. `ENXIO` is mapped to the `Device not configured` message by `errno`.

Fix. Implement the strategy function from Section 5.

27.18.5 `newfs_ufs` Hangs

Symptom. `newfs_ufs /dev/myblk0` starts up but never completes.

Likely causes.

- Your strategy function does not call `biodone` on some path. `newfs_ufs` issues a BIO, waits for its completion, and will wait forever if completion never comes.
- Your strategy function calls `biodone` twice on some path. The first call returns success; the second call usually panics, but in some cases the BIO state is corrupted enough to hang.

Fix. Audit your strategy function. Every control-flow path must end with exactly one call to `biodone(bp)`. A useful pattern is to use a single exit point at the end of the function.

27.18.6 `mount` Fails with “bad superblock”

Symptom. `mount /dev/myblk0 /mnt/myblk` reports `mount: /dev/myblk0: bad magic`.

Likely causes.

- Your strategy function is returning wrong data for some offsets. The superblock is at offset 65536, and UFS validates it carefully.
- Your bounds check is rejecting a legitimate read.
- Your `memcpy` is copying from the wrong address (usually an off-by-one in the offset arithmetic).

Fix. Write a known pattern to the device with `dd`, then read it back with `dd` at various offsets and compare with `cmp`. If the pattern round-trips, the basic I/O is correct. If not, find the first offset where it diverges and inspect the code at the corresponding bounds check or address arithmetic.

27.18.7 `kldunload` Hangs

Symptom. `kldunload myblk` does not return.

Likely causes.

- A BIO is in flight and your strategy function never calls `biodone`. `disk_destroy` is waiting for the BIO to complete.
- You added a worker thread and it is sleeping inside a function that will never be awakened.

Fix. Run `procstat -kk` in another terminal. Look at the `g_event` thread's stack and at any of your driver's threads. If they are stuck in a `sleep` or `waitfor` state, you have a leaked BIO or a misbehaving worker.

27.18.8 `kldunload` Returns “Device busy”

Symptom. `kldunload myblk` reports `Device busy` and exits.

Likely causes.

- A filesystem is still mounted on `/dev/myblk0`.
- A program still has `/dev/myblk0` open for raw access.
- A `dd` from a previous terminal session is still running in the background.

Fix. Run `mount | grep myblk` to check for active mounts. Run `fuser /dev/myblk0` to find open handles. Unmount, close, then unload.

27.18.9 Kernel Panic with “freeing free memory”

Symptom. The kernel panics with a message about freeing already-freed memory, showing a stack trace through your driver.

Likely causes.

- The detach path is freeing the `softc` or the backing twice.
- A worker thread survived `disk_destroy` and tried to access freed state.

Fix. Review the detach order. Destroy the disk first (which waits for in-flight BIOs), then free the backing, then destroy the mutex, then free the `softc`. If you added a worker thread, make sure it has exited before any `free` is called.

27.18.10 Kernel Panic with “`vm_fault: kernel mode`”

Symptom. The kernel panics with a page fault inside your driver, typically in the strategy function or in the detach path.

Likely causes.

- You dereferenced a null or freed pointer. The most common case is using `sc->backing` after it has been freed.
- You confused `bp->bio_data` with `bp->bio_disk` and read from the wrong pointer.

Fix. Audit pointer lifetimes. If the backing store is freed during detach, ensure no BIOs can still reach the strategy function after that point. The order `disk_destroy -> free(backing)` is the correct order.

27.18.11 gstat Shows No Activity

Symptom. You are running `dd` or `newfs_ufs` against the device, but `gstat -f myblk0` shows zero ops/s.

Likely causes.

- You are watching the wrong device. `gstat -f myblk0` uses a regex; make sure your device name matches.
- Your driver uses a custom GEOM class name that `gstat` is filtering out.

Fix. Run `gstat` without the filter and look for your device. Check the name field carefully.

27.18.12 “Operation not supported” for DELETE

Symptom. `Mount-with-trim` fails or `fstrim` prints “Operation not supported”.

Likely causes.

- Your strategy function does not handle `BIO_DELETE` and returns `EOPNOTSUPP`.
- The filesystem probed `BIO_DELETE` support during mount and cached the negative result.

Fix. Implement `BIO_DELETE` in the strategy function, then unmount and remount. Most filesystems only probe at mount time.

27.18.13 /dev/myblk0 Does Not Appear Until Several Seconds After kldload

Symptom. Immediately after `kldload`, `ls /dev/myblk0` fails. A few moments later, it succeeds.

Likely causes.

- GEOM processes events asynchronously. `disk_create` queues an event, and the provider is not published until the event thread picks it up.
- On a system under load, the event queue may be slow.

Fix. This is normal behaviour. If your scripts depend on the node existing immediately after `kldload`, add a small sleep or a polling loop.

27.18.14 Data Written Is Readable but Garbled

Symptom. A read after a write returns the right number of bytes but with different content.

Likely causes.

- Off-by-one in the backing-store offset arithmetic.
- A concurrent BIO is overlapping with the one you expect, and your lock is not held for long enough.
- The strategy function is reading from `bp->bio_data` before the kernel has finished setting it up (this is extremely unlikely for normal BIOs, but can happen with bugs in how you parse attributes).

Fix. Add a `printf` to the strategy function that logs the first few bytes before and after the `memcpy`. Repeat the test with a known pattern and look for the mismatch.

27.18.15 Backing Store Not Freed, Memory Grows With Every Reload

Symptom. `vmstat -m | grep myblk` shows the allocation bytes growing with each load/unload cycle.

Likely causes.

- The `MOD_UNLOAD` handler returned without calling `myblk_detach_unit`, so the `free(sc->backing, M_MYBLK)` was skipped.
- An error path in `MOD_UNLOAD` returned early before reaching the `free`. Every error path needs to free or the allocation leaks.
- A worker thread is holding a reference to the `softc` and the handler refuses to free while that reference exists.

Fix. Audit the `MOD_UNLOAD` path. `vmstat -m` is a blunt but effective tool. Add a `printf` in the free path to confirm it is being reached.

27.18.16 gstat Shows Very High Queue Length

Symptom. `gstat -I 1` shows `L(q)` rising into the tens or hundreds and never returning to zero.

Likely causes.

- Your strategy function is slow or blocking, causing BIOs to queue faster than they are serviced.
- You added a worker thread but it is scheduled less often than it should be.
- A synchronisation bottleneck (a heavily contested mutex) is serialising the work.

Fix. Profile with DTrace to find what the strategy function is doing. If the latency per BIO has grown, investigate why. For an in-memory driver, this should almost never happen; if it does, you have likely introduced a `vn_rdw` or other blocking call into the hot path.

27.18.17 Strategy Function Called With NULL `bio_disk`

Symptom. Kernel panic in the strategy function when dereferencing `bp->bio_disk->d_drv1`.

Likely causes.

- The BIO was synthesised incorrectly by code outside your driver.
- You are accessing `bp->bio_disk` from the wrong context. In some GEOM paths, `bp->bio_disk` is only valid inside the strategy function of a `g_disk` driver.

Fix. If you need to access the `softc`, do it at the start of the strategy function. Cache the pointer in a local variable. Do not access `bp->bio_disk` from a different thread or from a deferred callback.

27.18.18 Mysterious I/O Errors After Reload

Symptom. After `kldunload` and `kldload`, reads return EIO on offsets that worked before the unload.

Likely causes.

- You are working with a file-backed or vnode-backed experiment (from Lab 5 or from your own modifications) and the file's size or contents have been changed between loads.
- A type mismatch between the saved and new offsets (for instance, changing `d_sectorsize` between loads).
- `d_mediasize` has changed but the underlying file still reflects the old layout.

Fix. Ensure the backing file and the driver's geometry agree on both size and sector layout. If you change `d_mediasize` or `d_sectorsize`, regenerate the backing file to match. For a straightforward reload with no changes, the buffer on a RAM-backed driver is always fresh, so mysterious post-reload EIOs usually point to a geometry mismatch rather than data loss.

27.18.19 Access Count Stuck at Non-Zero After Unmount

Symptom. After `umount`, `geom disk list` still shows non-zero access counts.

Likely causes.

- A program still has the raw device open. `fuser /dev/myblk0` will reveal it.
- The filesystem did not unmount cleanly. Check `mount | grep myblk` to see if it is still mounted.
- A lingering NFS client or similar is holding the filesystem open. Unlikely for a local memory disk, but possible on shared systems.

Fix. Find and close the open handle. If `umount` reports success but the access count remains, rebooting is the safest recovery.

27.18.20 Driver Loaded But Not Visible in `geom -t`

Symptom. `kldstat` shows the module loaded, but `geom -t` does not show any geom of our name.

Likely causes.

- The loader ran but never called `disk_create`.
- `disk_create` was called but the event thread has not run yet.

Fix. Add a `printf` to confirm `disk_create` ran. Wait one or two seconds after `kldload` before checking, to give the event thread a chance.

27.18.21 Panic on Second Load

Symptom. Loading the module once works. Unloading works. Loading a second time panics.

Likely causes.

- A `MOD_UNLOAD` handler did not reset all the state that `MOD_LOAD` assumes is fresh.

- A static pointer holds a reference to a freed structure across the unload boundary; the next load sees a dangling pointer.
- A GEOM class that was registered on first load did not unregister.

Fix. Audit your load and unload paths as a matched pair. Every allocation on load needs a corresponding free on unload, and every pointer written on load needs to be cleared on unload. For GEOM classes, `DECLARE_GEOM_CLASS` handles the unregistration for you, but if you bypass it you must do the work.

27.18.22 `newfs_ufs` Aborts with “File system too small”

Symptom. `newfs_ufs /dev/myblk0` aborts with `newfs: /dev/myblk0: partition smaller than minimum UFS size`.

Likely causes.

- `MYBLK_MEDIASIZE` is too small for UFS’s minimum practical size.
- You forgot to rebuild the module after changing the size.

Fix. Ensure the media size is at least a few megabytes. UFS’s absolute minimum is around 1 MiB, but practical minimums are 4-8 MiB and comfortable minimums are 32 MiB or more.

27.18.23 `mount -o trim` Does Not Trigger `BIO_DELETE`

Symptom. Mounting with `-o trim` succeeds, but `gstat` shows no delete operations even during heavy deletion.

Likely causes.

- UFS issues `BIO_DELETE` only on certain patterns; it does not unconditionally trim every freed block.
- Your driver does not advertise `BIO_DELETE` support in its `d_flags`.

Fix. Set `sc->disk->d_flags |= DISKFLAG_CANDELETE`; before `disk_create`. This tells GEOM and filesystems that your driver supports `BIO_DELETE` and is willing to handle them.

27.18.24 UFS Complains About “Fragment out of bounds”

Symptom. After a mount, UFS logs an error about a fragment being out of bounds, and file operations start returning EIO.

Likely causes.

- Your driver is returning wrong data on some offset, and UFS has read a corrupted metadata block.
- The backing store was partially overwritten during some other test.
- Bounds-check arithmetic is returning incorrect ranges.

Fix. Unmount, run `fsck_ffs -y /dev/myblk0` to repair, then re-test. If the error recurs with fresh filesystems, look for offset-computation bugs in the strategy function.

27.18.25 Kernel Printing “interrupt storm” Messages

Symptom. dmesg shows messages about interrupt storms, and system responsiveness degrades.

Likely causes.

- A real hardware driver (not yours) is misbehaving.
- Your driver is fine; this is a different subsystem’s problem.

Fix. Verify that the storm is not related to your module. If it is, the issue is almost certainly in an interrupt handler, which our pseudo driver does not have.

27.18.26 Reboot Hangs on Unmount During Shutdown

Symptom. On shutdown, the system hangs while unmounting, with a message like “Syncing disks, vnodes remaining...”.

Likely causes.

- A filesystem is still mounted on your device, and your driver is holding a BIO.
- A syncer thread is stuck waiting for completion.

Fix. Ensure your driver unmounts cleanly before system shutdown. A robust way is to add a shutdown_post_sync event handler that unmounts the filesystem and unloads the module. For development, unmount and unload manually before issuing shutdown `-r now`.

27.18.27 General Advice

Whenever something goes wrong, the first step is to read dmesg and look for messages from your own printf’s and from kernel subsystems. The second step is to run `procstat -kk` and look at what threads are doing. The third step is to consult `gstat`, `geom disk list`, and `geom -t` for the storage topology. These three tools will tell you most of what you need in nearly every case.

If a panic happens, FreeBSD drops you into the debugger. Capture a backtrace with `bt` and a register dump with `show registers`, then reboot with `reboot`. If a crash dump was taken, `kgdb` on `/var/crash/vmcore.last` will let you inspect the state offline. Keeping crash dumps around, at least in a development environment, pays off immediately when you are chasing intermittent bugs.

And above all, when something fails, try to reproduce it. Intermittent bugs in storage drivers are almost always caused by timing differences in how many BIOs are in flight, how long they take, and when the scheduler decides to run your thread. If you can find a reliable reproduction, you are most of the way to a fix.

27.19 Wrapping Up

This has been a long chapter. Let us take a moment to step back and see what we have covered.

We started by situating storage drivers in FreeBSD’s layered architecture. The Virtual File System layer sits between system calls and filesystems, giving every filesystem a common shape. `devfs` is itself a filesystem, providing the `/dev` directory that user-space tools and administrators use to refer to devices. Storage drivers do

not live inside VFS. They live at the bottom of the stack, below GEOM and below the buffer cache, and they communicate with the rest of the kernel through `struct bio`.

We built a working pseudo block device driver from scratch. In Section 3 we wrote the skeleton that registered a disk with `g_disk` and published a `/dev` node. In Section 4 we explored GEOM's concepts of classes, geoms, providers, and consumers, and we understood how the topology composes and how access counts keep the system safe during teardown. In Section 5 we implemented the strategy function that actually services `BIO_READ`, `BIO_WRITE`, `BIO_DELETE`, and `BIO_FLUSH` against an in-memory backing store. In Section 6 we formatted the device with `newfs_ufs`, mounted a real filesystem on it, and saw the two access paths (raw and filesystem) converge in our strategy function. In Section 7 we surveyed persistence options and added a simple technique for surviving module reloads. In Section 8 we walked through the teardown path in detail and learned how to test it. In Section 9 we looked at the directions a growing driver tends to go: multi-unit support, ioctl surfaces, source-file splits, and stable versioning.

We exercised the driver through labs and stretched it with challenges. We collected the common failure modes in a troubleshooting section. And throughout, we kept our eyes on the real FreeBSD source tree, because the goal of this book is not to teach toy kernel code but to teach the real thing.

You should now be able to read `md(4)` with real comprehension rather than just staring at it. You should be able to read `g_zero.c` and recognise every function it calls. You should be able to diagnose the common classes of storage-driver bug by symptom. And you should have a working, if simple, pseudo block device that you wrote yourself.

That is a substantial amount of ground covered. Take a moment to notice how far you have come. In Chapter 26 you knew how to write a character driver. Now you can also write a block driver. The two chapters together give you the foundation for nearly every other kind of driver in FreeBSD, because most drivers are either character-oriented or block-oriented at the boundary where they meet the rest of the kernel.

27.19.1 A Summary of the Key Moves

For quick recall, here are the moves that define a minimal storage driver.

1. Include the right headers: `sys/bio.h`, `geom/geom.h`, `geom/geom_disk.h`.
2. Allocate a `struct disk` with `disk_alloc`.
3. Fill in `d_name`, `d_unit`, `d_strategy`, `d_sectorsize`, `d_mediasize`, `d_maxsize`, and `d_drv1`.
4. Call `disk_create(sc->disk, DISK_VERSION)`.
5. In `d_strategy`, switch on `bio_cmd` and service the request. Always call `biodone` exactly once.
6. In the unload path, call `disk_destroy` before freeing anything the strategy function touches.
7. Declare `MODULE_DEPEND` on `g_disk`.
8. Use `MAXPHYS` for `d_maxsize` unless you have a specific reason to be smaller.
9. Test the unload path under load. Test it with a mounted filesystem. Test it with a raw cat holding the device open.
10. Read `dmesg`, `gstat`, `geom disk list`, and `procstat -kk` when something goes wrong.

These ten moves are the skeleton of every FreeBSD storage driver you will ever write. They appear in different clothing in `ada(4)`, in `nvme(4)`, in `mmcsd(4)`, in `zvol(4)`, and in every other driver in the tree. Once you see the pattern, the variety across real drivers becomes much less mysterious.

27.19.2 A Reminder About Raw Access

Even with a filesystem mounted, your driver is still reachable as a raw block device. `/dev/myblk0` remains a valid handle that tools like `dd`, `diskinfo`, `gstat`, and `dtrace` can use. The two access paths coexist through GEOM's discipline: both paths issue BIOSs, both paths respect the access counts, and your strategy function services both without distinguishing between them. That uniformity is the great gift of GEOM to storage-driver authors.

27.19.3 A Reminder About Safety

Working on a shared system while developing a storage driver is an invitation for pain. Use a virtual machine, or at the very least a system you can reinstall. Keep a rescue image handy. Keep backups of anything you cannot afford to lose, including code you are in the middle of writing. The chapter's driver is well-behaved and should not damage anything, but drivers you write in the future may not be, and the cost of being prepared is very small compared to the cost of being unprepared.

27.19.4 Where to Look Next in the FreeBSD Tree

If you want to keep exploring storage before the next chapter, three areas of the tree repay careful reading.

- `/usr/src/sys/geom/` has the GEOM framework itself, including `g_class`, `g_disk`, and many transformation classes like `g_mirror`, `g_stripe`, and `g_eli`.
- `/usr/src/sys/dev/md/md.c` is the full-featured memory-disk driver, already mentioned many times in this chapter.
- `/usr/src/sys/ufs/` is the UFS filesystem. Not required reading for driver work, but it helps to see the layer immediately above yours.

Reading these is not a prerequisite for the next chapter. It is a recommendation for your own growth.

27.20 Bridge to the Next Chapter

In this chapter we built a storage driver from scratch. The data flowing through it was internal to the system: bytes written to a file, bytes read from a file, superblocks and cylinder groups and inodes shuffling around in the buffer cache. No byte ever left the machine. The driver's entire world was the kernel's own memory and the processes that consume it.

Chapter 28 takes us into a different world. We will write a network interface driver. Network drivers are transport drivers like the USB serial driver of Chapter 26 and the storage driver of this chapter, but their conversation partner is not a process and not a filesystem. It is a network stack, and the unit of work is not a byte range and not a block but a packet. The packet is a structured object with headers and payload, and the driver participates in a stack that includes IP, ARP, ICMP, TCP, UDP, and many other protocols.

The patterns you have internalised in this chapter will reappear, with different names. Instead of `struct bio`, you will see `struct mbuf`. Instead of `g_disk`, you will see the `ifnet` interface. Instead of `disk_strategy`, you will see the `if_transmit` and `if_input` hooks. Instead of GEOM providers and consumers, you will see

network interface objects linked into the kernel's network stack. The role is the same: a transport driver takes requests from above, delivers them below, accepts responses below, and delivers them above.

Many of the concerns will also be the same. Locking. Hot unplug. Resource cleanup on detach. Observability through kernel tools. Safety in the face of errors. The fundamentals carry over. What changes is the vocabulary, the structure of the unit of work, and some of the specific tools.

Before you move on, take a short break. Unload your storage driver. Run `kldstat` and confirm that nothing from this chapter is still loaded. Close your lab logbook. Stand up. Refill your coffee. The next chapter is going to be just as substantive as this one, and you will want a clear head.

When you come back, Chapter 28 will start the same way this one did: with a gentle introduction and a clear picture of where we are going. See you there.

27.21 Quick Reference

The tables below are intended as a quick lookup when you are writing or debugging a storage driver and need to remember a name, a command, or a path. They are not a substitute for the full explanations earlier in the chapter.

27.21.1 Key Headers

Header	Defines
<code>sys/bio.h</code>	<code>struct bio</code> , <code>BIO_READ</code> , <code>BIO_WRITE</code> , <code>BIO_DELETE</code> , <code>BIO_FLUSH</code> , <code>BIO_GETATTR</code>
<code>geom/geom.h</code>	<code>struct g_class</code> , <code>struct g_geom</code> , <code>struct g_provider</code> , <code>struct g_consumer</code> , topology primitives
<code>geom/geom_disk.h</code>	<code>struct disk</code> , <code>DISK_VERSION</code> , <code>disk_alloc</code> , <code>disk_create</code> , <code>disk_destroy</code> , <code>disk_gone</code>
<code>sys/module.h</code>	<code>DECLARE_MODULE</code> , <code>MODULE_VERSION</code> , <code>MODULE_DEPEND</code>
<code>sys/malloc.h</code>	<code>MALLOC_DEFINE</code> , <code>malloc</code> , <code>free</code> , <code>M_WAITOK</code> , <code>M_NOWAIT</code> , <code>M_ZERO</code>
<code>sys/lock.h</code> , <code>sys/mutex.h</code>	<code>struct mtx</code> , <code>mtx_init</code> , <code>mtx_lock</code> , <code>mtx_unlock</code> , <code>mtx_destroy</code>

27.21.2 Key Structures

Structure	Role
<code>struct disk</code>	The <code>g_disk</code> representation of a disk. Filled in by the driver, owned by the framework.
<code>struct bio</code>	One I/O request, passed between GEOM layers and into the driver's strategy function.
<code>struct g_provider</code>	The producer-facing interface of a geom. Filesystems and other geoms consume from providers.
<code>struct g_consumer</code>	The connection from one geom into another geom's provider.
<code>struct g_geom</code>	An instance of a <code>g_class</code> .
<code>struct g_class</code>	The template from which geoms are created. Defines methods like <code>init</code> , <code>fini</code> , <code>start</code> , <code>access</code> .

27.21.3 Common BIO Commands

Command	Meaning
<code>BIO_READ</code>	Read bytes from the device into a buffer.
<code>BIO_WRITE</code>	Write bytes from a buffer into the device.
<code>BIO_DELETE</code>	Discard a range of blocks. Used for TRIM.
<code>BIO_FLUSH</code>	Commit outstanding writes to durable storage.
<code>BIO_GETATTR</code>	Query a named attribute from the provider.
<code>BIO_ZONE</code>	Zoned-block-device operations. Not commonly used.

27.21.4 Common GEOM Tools

Tool	Purpose
<code>geom disk list</code>	List registered disks and their providers.
<code>geom -t</code>	Show the entire GEOM topology as a tree.
<code>geom part show</code>	Show partition geoms and their providers.
<code>gstat</code>	Live per-provider I/O statistics.
<code>diskinfo -v /dev/xxx</code>	Show disk geometry and attributes.

Tool	Purpose
<code>iostat -x 1</code>	Live per-device throughput and latency.
<code>dd if=... of=...</code>	Raw block I/O for testing.
<code>newfs_ufs /dev/xxx</code>	Create a UFS filesystem on a device.
<code>mount /dev/xxx /mnt</code>	Mount a filesystem.
<code>umount /mnt</code>	Unmount a filesystem.
<code>mdconfig</code>	Create or destroy memory disks.
<code>fuser</code>	Find processes holding a file open.
<code>procstat -kk</code>	Show kernel stack traces for all threads.

27.21.5 Key Callback Typedefs

Typedef	Purpose
<code>disk_strategy_t</code>	Handles BIOs. The core I/O function. Required.
<code>disk_open_t</code>	Called when a new access is being granted. Optional.
<code>disk_close_t</code>	Called when an access is being released. Optional.
<code>disk_ioctl_t</code>	Handles ioctls on the <code>/dev</code> node. Optional.
<code>disk_getattr_t</code>	Answers <code>BIO_GETATTR</code> queries. Optional.
<code>disk_gone_t</code>	Notifies the driver when the disk is being forced away. Optional.

27.21.6 File and Path Reference

Path	What lives there
<code>/usr/src/sys/geom/geom_disk.c</code>	The kernel disk implementation.
<code>/usr/src/sys/geom/geom_public.h</code>	The public <code>g_disk</code> interface.
<code>/usr/src/sys/geom/geom.c</code>	Core GEOM structures and functions.
<code>/usr/src/sys/sys/bio.h</code>	The <code>struct bio</code> definition.
<code>/usr/src/sys/dev/md/md.c</code>	The reference memory-disk driver.
<code>/usr/src/sys/geom/zero/g_zero.c</code>	A minimal GEOM class, useful as a reading reference.
<code>/usr/src/sys/ufs/ffs/ffs_mount.c</code>	UFS's mount path. Read if you want to see what mount does at the filesystem side.
<code>/usr/src/share/man/man9/geom.9</code>	This disk(9) manual page.

Path	What lives there
/usr/src/share/man/man9/man9job.9	The job(9) manual page.

27.21.7 Common Disk Flags

Flag	Meaning
DISKFLAG_CANDELETE	The driver handles BIO_DELETE.
DISKFLAG_CANFLUSHCACHE	The driver handles BIO_FLUSH.
DISKFLAG_UNMAPPED_BIO	The driver accepts unmapped BIOs (advanced).
DISKFLAG_WRITE_PROTECT	The device is read-only.
DISKFLAG_DIRECT_COMPLETION	Completion is safe from any context (advanced).

These flags are set on `sc->disk->d_flags` before `disk_create`. They let the kernel make smarter choices about how to issue BIOs to your driver.

27.21.8 Patterns for `d_strategy`

Here are the three most common shapes of a strategy function.

Pattern 1: Synchronous, in-memory. Our driver uses this. The function serves the BIO inline and returns after calling `biodone`.

```
void strategy(struct bio *bp) {
    /* validate */
    switch (bp->bio_cmd) {
        case BIO_READ: memcpy(bp->bio_data, ...); break;
        case BIO_WRITE: memcpy(..., bp->bio_data); break;
    }
    bp->bio_resid = 0;
    biodone(bp);
}
```

Pattern 2: Enqueue to a worker thread. `md(4)` uses this. The function attaches the BIO to a queue and signals a worker.

```
void strategy(struct bio *bp) {
    mtx_lock(&sc->lock);
    TAILQ_INSERT_TAIL(&sc->queue, bp, bio_queue);
    wakeup(&sc->queue);
}
```

```
mtx_unlock(&sc->lock);  
}
```

The worker dequeues BIOs, services them one at a time (perhaps calling `vn_rdwr` or issuing hardware commands), and completes each with `biodone`.

Pattern 3: Hardware DMA with interrupt completion. Real hardware drivers use this. The function programs the hardware, sets up DMA, and returns. A later interrupt handler completes the BIO.

```
void strategy(struct bio *bp) {  
    /* validate */  
    program_hardware(bp);  
    /* strategy returns, interrupt will call biodone eventually */  
}
```

Each pattern has trade-offs. Pattern 1 is simplest but cannot block. Pattern 2 handles blocking work but adds latency. Pattern 3 is needed for real hardware but requires interrupt handling, which adds a whole other layer of complexity.

Our chapter driver uses Pattern 1. `md(4)` uses Pattern 2. `ada(4)`, `nvme(4)`, and friends use Pattern 3.

27.21.9 Minimum Registration Sequence

For the driver writer in a hurry, the minimum sequence to register a block device is:

```
sc->disk = disk_alloc();  
sc->disk->d_name      = "myblk";  
sc->disk->d_unit      = sc->unit;  
sc->disk->d_strategy   = myblk_strategy;  
sc->disk->d_sectorsize = 512;  
sc->disk->d mediasize  = size_in_bytes;  
sc->disk->d_maxsize    = MAXPHYS;  
sc->disk->d_drv1       = sc;  
disk_create(sc->disk, DISK_VERSION);
```

And the minimum teardown sequence is:

```
disk_destroy(sc->disk);  
free(sc->backing, M_MYBLK);  
mtx_destroy(&sc->lock);  
free(sc, M_MYBLK);
```

27.22 Glossary

Access count. A tuple of three counters on a GEOM provider that tracks how many readers, writers, and exclusive holders currently have access to it. Displayed as `rNwNeN` in `geom disk list`.

Attach. In the Newbus sense, the step where a driver takes responsibility for a device. In the storage sense, the step where the driver calls `disk_create` to register with `g_disk`. The word overloads; use context.

Backing store. The place where the bytes of a storage device actually live. For our driver, the backing store is a `malloc`'d buffer in kernel memory. For real disks, it is the platter or the flash. For `md(4)` in `vnode` mode, it is a file in the host filesystem.

BIO. A `struct bio`. The unit of I/O that flows through GEOM.

BIO_DELETE. A BIO command that asks the driver to discard a range of blocks. Used for TRIM on SSDs.

BIO_FLUSH. A BIO command that asks the driver to make all previous writes durable before returning.

BIO_GETATTR. A BIO command that asks the driver to return the value of a named attribute.

BIO_READ. A BIO command that asks the driver to read a range of bytes.

BIO_WRITE. A BIO command that asks the driver to write a range of bytes.

Block device. A device that is addressed in fixed-size blocks, with seekable random access. Historically distinct from character devices in BSD; in modern FreeBSD, block and character access converge through GEOM but the mental distinction still matters.

Buffer cache. The kernel subsystem that holds recently used filesystem blocks in RAM. Sits between filesystems and GEOM. Not to be confused with the page cache; they are related but distinct in FreeBSD.

Cache coherency. The property that reads and writes see each other in a consistent order. The strategy function must not return data that is stale relative to recent writes on the same offset.

Cdev. A character device node as represented by `struct cdev`. Character drivers create them with `make_dev`. Block drivers usually do not.

Consumer. The input-facing side of a geom. A consumer attaches to a provider and issues BIOs into it.

d_drv1. A generic pointer in `struct disk` where the driver stores its private context, typically the `softc`.

d_mediasize. The total size of the device in bytes.

d_maxsize. The largest single BIO the driver can accept. Usually `MAXPHYS` for pseudo devices.

d_sectorsize. The size of a sector in bytes. Typically 512 or 4096.

d_strategy. The driver's BIO-handling function.

Devfs. A pseudo-filesystem mounted at `/dev` that synthesises file nodes for kernel devices.

Devstat. The kernel's device statistics subsystem, used by `iostat`, `gstat`, and others. Storage drivers using `g_disk` get `devstat` integration automatically.

Disk_alloc. Allocates a `struct disk`. Never fails; uses `M_WAITOK` internally.

Disk_create. Registers a filled-in `struct disk` with `g_disk`. The real work is done asynchronously.

Disk_destroy. Unregisters and destroys a `struct disk`. Waits for in-flight BIOs to complete. Panics if the provider still has users.

Disk_gone. Notifies `g_disk` that the underlying media is gone. Used in hot-unplug scenarios. Distinct from `disk_destroy`.

DISK_VERSION. The ABI version of the `struct disk` interface. Defined in `geom_disk.h` and passed to `disk_create`.

DTrace. FreeBSD's dynamic tracing facility. Especially useful for observing BIO traffic.

Event thread. The single kernel thread that GEOM uses to process topology events such as creating and destroying geoms. Usually called `g_event` in `procstat` output.

Exclusive access. A kind of access on a provider that forbids other writers. Filesystems acquire exclusive access on the devices they mount.

Filesystem. A concrete implementation of file storage semantics, such as UFS, ZFS, tmpfs, or NFS. Plugs into VFS.

GEOM. The FreeBSD framework for composable block-layer transformations. Classes, geoms, providers, and consumers are its main objects.

g_disk. The GEOM subsystem that wraps disk-shaped drivers with a simpler API. Our driver uses it.

g_event. The GEOM event thread that processes topology changes.

g_io_deliver. The function used at the class level to complete a BIO. `g_disk` calls it for us; our driver calls `biodone`.

g_io_request. The function used at the class level to issue a BIO downward. Only used in drivers that implement their own GEOM class.

Hotplug. A device that can appear or disappear without reboot.

ioctl. A control operation on a device, distinct from read or write. In the storage path, `ioctls` on `/dev/diskN` go through GEOM and may be handled by `g_disk` or by the driver's `d_ioctl`.

md(4). The FreeBSD memory-disk driver. The canonical pseudo block device in the tree, and a recommended reading reference.

Mount. The act of attaching a filesystem to a point in the namespace. Calls VFS, which calls the filesystem's own mount routine, which typically opens a GEOM provider.

Newbus. FreeBSD's bus framework. Used for character and hardware drivers. Our storage driver does not directly use Newbus because it is a pseudo device; real storage drivers almost always do.

Provider. The output-facing side of a geom. Other geoms or `/dev` nodes consume providers.

Softc. The per-instance state structure of a driver.

Strategy function. The driver's BIO handler. Called `d_strategy` in the `struct disk` API.

Superblock. A small on-disk structure that describes the layout of a filesystem. UFS's is at offset 65536.

Topology. The tree of GEOM classes, geoms, providers, and consumers. Protected by the topology lock.

Topology lock. The global lock protecting the GEOM topology from concurrent modification.

UFS. The Unix File System, FreeBSD's default filesystem. Lives under `/usr/src/sys/ufs/`.

Unit. A numbered instance of a driver. `myblk0` is unit 0 of the `myblk` driver.

VFS. The Virtual File System layer. Sits between system calls and concrete filesystems.

Vnode. The kernel's runtime handle on an open file or directory. Lives inside VFS.

Withering. The GEOM process of removing a provider from the topology. Queued on the event thread, waits for in-flight BIOs, and finally destroys the provider.

Zone. In VM-subsystem vocabulary, a pool of fixed-size objects allocated through the UMA (Universal Memory Allocator). Many kernel structures, including BIOs and GEOM providers, are allocated from zones.

BIO_ORDERED. A BIO flag that asks the driver to execute this BIO only after all previously issued BIOs have completed. Used for write barriers.

BIO_UNMAPPED. A BIO flag that indicates `bio_data` is not a mapped kernel virtual address but rather a list of unmapped pages. Drivers that can handle unmapped data should set `DISKFLAG_UNMAPPED_BIO`.

Direct completion. Completing a BIO in the same thread that submitted it, without going through a deferred callback. Usually faster but not always safe.

Drivers in Tree. Drivers that live inside the FreeBSD source tree and are built as part of the standard kernel build. Contrast with out-of-tree drivers, which are maintained separately.

Out-of-tree driver. A driver that is not part of the FreeBSD source tree. These need to be compiled against a matching kernel and may need updates when the kernel's ABI changes.

ABI. Application Binary Interface. The set of conventions for function calling, structure layout, and type sizes that allow two pieces of compiled code to interoperate. `DISK_VERSION` is one kind of ABI marker.

API. Application Programming Interface. The set of function signatures and types that code uses at the source level. Distinct from ABI: two kernels with the same API might have different ABIs if they were compiled differently.

KPI. Kernel Programming Interface. FreeBSD's preferred term for the kernel's API. Guarantees about KPI stability are limited; always recompile against the kernel you are running.

KLD. Kernel loadable module. The `.ko` file we produce. The "KLD" stands for Kernel Loadable Driver, though modules are not necessarily drivers.

Module. See KLD.

Taste. In GEOM vocabulary, the process of offering a provider to all classes so that each class can decide whether to attach to it. Tasting happens automatically when new providers appear.

Retaste. Forcing GEOM to taste a provider again, usually after its contents have changed. `geom provider retaste` triggers this for one provider; `geom retaste` triggers it globally.

Orphan. In GEOM vocabulary, a provider whose underlying storage has gone away. Orphans are cleaned up by the event thread.

Spoil. A GEOM concept related to cache invalidation. If a provider's contents change in a way that could invalidate caches, it is said to have been spoiled.

Bufobj. A kernel object that associates a vnode (or a GEOM consumer) with a buffer cache. Each block device and each file has one.

bdev_strategy. A legacy synonym for `d_strategy`. Modern code uses `d_strategy` directly.

Schedule. The act of placing a BIO onto an internal queue for later execution. Distinct from "executing".

Plug/unplug. In some kernels, the plug is a batching mechanism for BIO submission. FreeBSD does not have plug/unplug; it delivers BIOs immediately.

Elevator. A BIO scheduler that sorts BIOs by offset to reduce disk seek time. FreeBSD's GEOM does not implement an elevator at the GEOM layer; it is the block device's responsibility, if relevant.

Superblock. The first metadata block of a filesystem. Describes geometry. At offset 65536 for UFS.

Cylinder group. A UFS concept. The filesystem is divided into regions, each with its own inode table and block allocation bitmap. Keeps related data physically close on a spinning disk and limits the damage a single bad region can cause.

Inode. A UFS (and POSIX) structure describing one file: its mode, owner, size, timestamps, and pointers to data blocks. Filenames live in directory entries, not in inodes.

Vop_vector. The dispatch table that a filesystem provides to VFS, listing all the operations VFS knows how to ask about (open, close, read, write, lookup, rename, and so on). VFS calls these as indirect function pointers.

Devstat. A kernel structure attached to devices that records aggregate I/O statistics. `iostat(8)` reads devstat data; `g_disk` allocates and feeds a devstat structure for every disk it creates.

bp. Shorthand in kernel source for a BIO pointer. Used almost universally in strategy functions and completion callbacks. When you see `struct bio *bp`, read it as “the current request”.

Bread. Buffer-cache function that reads a block, consulting the cache first and issuing I/O only on a miss. Used by filesystems, not by drivers.

Bwrite. Buffer-cache function that writes a block synchronously. The filesystem uses it; your strategy function eventually sees the resulting BIO.

Getblk. Buffer-cache function that returns a buffer for a given block, allocating it if necessary. Used by filesystems as the entry point to both reading and writing.

Bdwrite. Delayed buffer-cache write. Marks the buffer dirty but does not issue I/O immediately. Will be written later by the syncer or by buffer cache pressure.

Bawrite. Asynchronous buffer-cache write. Like `bwrite` but does not wait for completion.

Syncer. A kernel thread that periodically flushes dirty buffers to their backing devices. Closing a filesystem cleanly requires the syncer to finish.

Taskqueue. A kernel facility for running callbacks in a separate thread. Useful when your strategy function wants to defer work. Covered in more depth when we discuss interrupt handlers in later chapters.

Callout. A kernel facility for scheduling a one-shot or periodic callback at a given time. Not commonly used in simple storage drivers but very common in hardware drivers that implement timeouts.

Witness. A kernel subsystem that detects lock-order violations and prints warnings. Always enabled in debug kernels; saves hours of debugging.

INVARIANTS. A kernel compile option that adds runtime assertions. Always enabled in debug kernels; catches many storage bugs before they become silent corruption.

Debug kernel. A kernel built with `INVARIANTS`, `WITNESS`, and related options. Slower but much safer for driver development. Use one during lab work.

27.23 Frequently Asked Questions

27.23.1 Do I need to support BIO_ORDERED?

For a pseudo device that services BIOs synchronously, no. Each BIO completes before the next one is processed, which trivially preserves ordering. For an asynchronous driver, you must respect BIO_ORDERED by deferring subsequent BIOs until the ordered one completes.

27.23.2 What is the relationship between d_maxsize and MAXPHYS?

d_maxsize is the maximum BIO size your driver can accept. MAXPHYS is the compile-time upper bound on BIO size, defined in /usr/src/sys/sys/param.h. On 64-bit systems such as amd64 and arm64, MAXPHYS is 1 MiB; on 32-bit systems it is 128 KiB. FreeBSD 14.3 also exposes a runtime tunable, maxphys, which some subsystems consult through the MAXPHYS macro or the maxphys variable. Setting d_maxsize = MAXPHYS accepts whatever the kernel is willing to issue. For most pseudo drivers this is fine.

27.23.3 Can my driver issue BIOs to itself?

Technically yes, but it rarely makes sense. The pattern is used by GEOM transformation classes (they take in BIOs from above and issue new BIOs downward). A g_disk driver is at the bottom of the stack and has no downward; if you need to split work across multiple backing units, you probably want worker threads rather than nested BIOs.

27.23.4 Why do some fields in struct disk use u_int and others off_t?

u_int is used for unsigned integer sizes that fit in 32 bits (sector size, number of heads, etc.). off_t is a signed 64-bit type used for byte offsets and sizes that can exceed 32 bits (media size, request offsets). The distinction matters for large disks; a media size of 10 TB requires more than 32 bits.

27.23.5 Is disk_alloc safe to call at any time?

disk_alloc uses M_WAITOK and will sleep if memory is tight. Do not call it while holding a spin lock or a mutex you cannot release. Call it at attach time, outside any lock.

27.23.6 What happens if I call disk_create twice with the same name?

disk_create will happily create multiple disks with the same name if the unit numbers differ. If both the name and unit number match, GEOM will reject the second registration and the resulting behaviour is implementation-defined. Avoid this case.

27.23.7 Can the strategy function sleep?

Technically yes, but it should not. The strategy function runs in the caller's thread context, and sleeping there stalls the caller. For work that must block, use a worker thread.

27.23.8 How do I know when all BIOs have finished for a given filesystem?

You usually do not need to. `umount(2)` does the work: it flushes dirty buffers, drains in-flight BIOs, and returns only after the filesystem is fully quiesced. After `umount` returns, no BIOs will arrive for that mount point unless something else opens the device.

27.23.9 Can I pass pointers between threads through `bio_caller1` or similar fields?

Yes. `bio_caller1` and `bio_caller2` are opaque fields meant for the issuer of the BIO to stash context that the completion handler can use. As long as you own the BIO (which you do, because you issued it), the fields are yours. `g_disk` drivers do not usually need them because the BIO arrives from above and is completed by calling `biodone`, with `g_disk` handling the callback routing.

27.23.10 My driver works on my laptop but not on the server. Why?

Possibilities: different kernel ABI (recompile against the server's kernel), different `MAXPHYS` (should be identical on 14.3 systems but check), different GEOM classes loaded (unlikely but possible), different memory size (your allocation might fail on a smaller system), different clock speed (affecting timing). Compare `uname -a` and `sysctl -a | grep kern.maxphys` to start.

27.23.11 Where does the `/dev` node's name actually come from?

From `d_name` and `d_unit` in the struct `disk` you pass to `disk_create`. GEOM concatenates them without a separator: `d_name = "myblk", d_unit = 0` produces `/dev/myblk0`. If you want a different convention, set `d_name` accordingly. There is no separator character between the name and the unit.

27.23.12 What is the maximum number of units I can create?

Limited by `d_unit`, which is `u_int`, so $2^{32} - 1$ in theory. In practice, per-unit memory consumption and the practical limits of `/dev` name space will stop you long before that.

27.23.13 Can I change `d_mediasize` after `disk_create`?

Yes, but carefully. Filesystems mounted on the disk will not pick up the change automatically; most will require unmount and remount. `md(4)` supports `MDIOCRESIZE` and there is infrastructure for signalling the change to GEOM, but the pattern is non-trivial.

27.23.14 What happens if I forget `MODULE_DEPEND`?

The kernel may fail to load your module if `g_disk` is not already loaded, or may load it successfully if `g_disk` happens to be built into the kernel. Always declare `MODULE_DEPEND` explicitly to avoid surprise.

27.23.15 Should I use `biodone` or `g_io_deliver` in my driver?

Use `biodone`. The `g_disk` wrapper provides a `d_strategy` style interface where the correct completion call is `biodone`. If you write your own `g_class`, you will call `g_io_deliver` instead, but that is a different path and a different chapter's worth of complexity.

27.23.16 How does `BIO_DELETE` relate to TRIM and UNMAP?

`BIO_DELETE` is the in-kernel abstraction. For SATA SSDs it maps to the ATA TRIM command, for SCSI/SAS to UNMAP, and for NVMe to Dataset Management with the deallocate bit. Userland triggers it through `fstrim(8)` or the `-o trim` mount option on UFS. Our driver is free to treat it as a hint or to honour it by zeroing memory, since backing is in RAM.

27.23.17 Why does my strategy function sometimes receive a BIO with `bio_length` of zero?

In normal operation you should never see this. If it happens, treat it as a defensive case: call `biodone(bp)` with no error and return. A length-zero BIO is not illegal, but it indicates something odd upstream. Filing a PR against the issuing code is reasonable.

27.23.18 What is the difference between `d_flags` and `bio_flags`?

`d_flags` is static configuration for the whole disk, set once at registration and describing what the driver can do (handles DELETE, can FLUSH, accepts unmapped BIOs, and so on). `bio_flags` is dynamic metadata on a single BIO, changing per request (ordered, unmapped, direct completion). Do not confuse them.

27.23.19 Can my driver present itself as removable media?

Yes, set `DISKFLAG_CANDELETE` and consider honouring `disk_gone` to simulate ejection. Tools like `camcontrol` and filesystem handlers generally treat any GEOM provider uniformly, so “removable” in the user-visible sense is less distinct than in other operating systems.

27.23.20 What thread actually calls my strategy function?

It depends. For synchronous submission from the buffer cache, it is the thread that called `bwrite` or `bread`. For asynchronous completion paths it is often a GEOM worker or the buffer cache's flusher thread. Your strategy must be written to tolerate any caller. Do not assume a specific thread identity or a specific priority.

27.23.21 How do I know which process caused a given BIO?

You usually cannot, because BIOs can be reordered, coalesced, merged, and issued from background threads that are not the original requester. `dtrace` with the `io:::start` probe plus stack captures can get you close, but it is investigative work, not a routine driver responsibility.

27.23.22 Can two different filesystems be mounted simultaneously on two unit numbers of my driver?

Yes, if you implemented multi-unit support. Each unit presents its own GEOM provider. Their backing stores are independent. The only shared state is your module's global variables and the kernel itself, so the two mounts do not interact unless you make them.

27.23.23 Should my driver handle power management events?

For a pseudo device, no. For a real hardware driver, yes: suspend and resume events flow through Newbus as method calls, and the driver must quiesce I/O on suspend and revalidate device state on resume. Storage drivers on laptops are a common source of suspend-related bugs, so real drivers take this seriously.

27.23.24 What is the practical impact of choosing 512 versus 4096 as `d_sectorsize`?

On modern filesystems, very little: UFS, ZFS, and most other FreeBSD filesystems work happily with either. On the driver side, a larger sector size reduces the number of BIOs for large transfers. On the workload side, applications doing `O_DIRECT` or aligned I/O may care. When in doubt, pick 4096 for new drivers; it matches modern flash and avoids alignment penalties.

27.23.25 If I reload my driver many times, does memory leak?

Only if you have a bug. In our design, `MOD_UNLOAD` calls `myblk_detach_unit`, which frees the backing store and the softc. The persistence variant deliberately retains the backing store across reloads but uses a single global pointer, so there is no leak; the same memory is reused. If `vmstat -m | grep myblk` climbs across reloads, investigate.

27.23.26 Why does `mount` sometimes succeed on my raw device but `newfs_ufs` fail?

`newfs_ufs` writes structured metadata (superblock, cylinder groups, inode tables) and then reads some of it back to verify. If the device is too small, corrupts writes silently, or returns errors only under certain conditions, `newfs_ufs` catches it first. `mount` is much less strict at the write path; it can read in a broken superblock and produce odd errors later. A successful `newfs` is a stronger correctness signal than a successful `mount`.

27.23.27 How do I verify that my `BIO_FLUSH` implementation actually makes data durable?

For our in-memory driver, durability is bounded by the host's power: flushing does nothing useful because a power cut takes everything with it. For a real driver backed by persistent storage, issuing a flush command to the underlying media and confirming completion before calling `biodone` is the contract. Testing requires a power-cycle harness or a simulator; there is no shortcut.

27.23.28 What are the correct locking rules inside `d_strategy`?

Hold the driver's lock long enough to protect the backing store against concurrent access, and release it before calling `biodone`. Never hold a lock across a call into another subsystem. Never call `malloc(M_WAITOK)` with a lock held. Never sleep. If you need to sleep, schedule the work on a taskqueue and call `biodone` from the worker.

27.23.29 Why is `BIO_FLUSH` not a percent-of-capacity barrier like write barriers on Linux?

FreeBSD's `BIO_FLUSH` is a point-in-time barrier: when it completes, all previously issued writes are durable. It is not associated with a particular range or percentage of the device. Drivers can implement it as a strict barrier or as an opportunistic flush, but the minimum contract is the point-in-time guarantee.

27.23.30 Are there any tools that generate BIO traffic to help me test?

Yes. `dd(1)` with various `bs=` values, `fio(1)` from ports, `ioping(8)` from ports, plus the usual suspects: `newfs`, `tar`, `rsync`, `cp`. `diskinfo -t` runs a suite of benchmark reads and is useful for rough throughput numbers. The test harnesses under `/usr/src/tools/regression/` can also be adapted.

27.24 What This Chapter Did Not Cover

This chapter is long, but there are several related topics we deliberately left for later. Naming them here helps you plan future study and prevents the false impression that storage drivers end at the BIO handler.

Real hardware storage drivers such as those for SATA, SAS, and NVMe controllers live under CAM and require significant additional machinery: command-block allocation, tagged queueing, hot-plug event handling, firmware upload, SMART data, and error-recovery protocols. We introduced the CAM world briefly through the `ada_da.c` excerpt but did not explore it in depth. Chapters 33 through 36 will tackle these interfaces, and the `md(4)` driver you read in this chapter is a deliberately small staircase by comparison.

ZFS integration is its own world. ZFS consumes GEOM providers through its vdev layer but adds copy-on-write semantics, end-to-end checksums, pooled storage, and snapshots that no simple block driver would ever need to know about. If your driver works under UFS it almost certainly works under ZFS, but the reverse is not guaranteed: ZFS exercises BIO paths, especially flushing and write ordering, that less demanding filesystems skip.

GEOM class authoring is a larger topic than `g_disk` wrapping. A full class implements `taste`, `start`, `access`, `attach`, `detach`, `dumpconf`, `destroy_geom`, and `orphan` methods. It can also create and destroy consumers, build multi-level topologies, and respond to configuration via `gctl`. The mirror, stripe, and crypt classes are good starting points once you decide to dig in.

Quotas, ACLs, and extended attributes are filesystem features that live above the GEOM layer entirely. They matter for userland but do not touch the storage driver. This is a useful piece of clarity: the driver's job ends at the BIO boundary.

Tracing and debugging kernel crashes deserves its own chapter. Kernel core dumps land on a dump device configured via `dumpon(8)`, analysed with `kgdb(1)` or `crashinfo(8)`. If your driver panics the system, being able to load the core file and inspect backtraces is a professional-level skill that this chapter only gestured at.

High-performance storage paths use features such as unmapped I/O, direct-dispatch completion, CPU pinning, NUMA-aware allocation, and dedicated queues. These optimisations matter for gigabytes-per-second workloads but are irrelevant to a teaching driver. When you start chasing microseconds, come back to `/usr/src/sys/dev/nvme/` and study how the real professionals do it.

Filesystem-specific behaviour varies widely. UFS asks for one set of BIOS; ZFS asks for a different set; `msdosfs` and `ext2fs` ask for something different again. A good storage driver is filesystem-agnostic, but observing different filesystems on your driver is a fantastic way to build intuition. Try `msdosfs`, `ext2fs`, and `tmpfs` for contrast after you are comfortable with UFS.

iSCSI and network block devices present themselves as GEOM providers too, but they are created by userland control daemons and talk to the network stack. Chapter 28 begins the networking work that makes those providers possible.

Our treatment of the storage path was deliberately focused. We wrote a driver that filesystems accept as real, we understood why and how it is seen that way, and we traced the data path from `write(2)` to RAM. That foundation is enough to make the unexplored topics above readable rather than bewildering.

27.25 Final Reflection

Storage drivers have a reputation as forbidding territory. This chapter should have replaced some of that reputation with familiarity: the BIO is just a structure, the strategy function is just a dispatcher, GEOM is just a graph, and `disk_create` is just a registration call. What elevates storage work from routine is not the underlying APIs, which are compact, but the operational demands that accumulate around them: performance, durability, error recovery, and correctness under contention.

Those demands do not go away when you leave pseudo devices for real hardware. They multiply. But you already have the vocabulary to understand them. You know what a BIO is and where it comes from. You know which thread calls your code and what it expects. You know how to register with GEOM, how to unregister cleanly, and how to recognise an in-flight request from its shadow in `gstat`. When you sit in front of a real SATA controller driver and start reading, you will recognise the shape of the code even though the specifics differ.

The craft of storage-driver writing is, ultimately, patient. You learn by writing small drivers, reading the source tree, reproducing simple experiments, and building instincts about when something that looks right is actually right. The chapter you just finished is a single long step along that journey. The next chapters will step again, each time in different directions.

27.26 Further Reading

If this chapter has whetted your appetite for storage internals, here are some places to go next.

Manual pages. `disk(9)`, `g_bio(9)`, `geom(4)`, `devfs(5)`, `ufs(5)`, `newfs(8)`, `mdconfig(8)`, `gstat(8)`, `diskinfo(8)`, `mount(2)`, `mount(8)`. Read them in that order.

The FreeBSD Architecture Handbook. The storage chapter is a good complement to this one.

Kirk McKusick et al., “The Design and Implementation of the FreeBSD Operating System”. The book’s chapters on the filesystem are especially relevant.

DTrace books. Brendan Gregg’s “DTrace Book” is a practical reference; Sun’s “Dynamic Tracing Guide” is the original tutorial.

The FreeBSD source tree. `/usr/src/sys/geom/`, `/usr/src/sys/dev/md/`, `/usr/src/sys/ufs/`, and `/usr/src/sys/cam/ata/` (where `ata_da.c` implements the ada disk driver). Every pattern discussed in this chapter is grounded in that code.

The mailing list archives. `freebsd-geom@` and `freebsd-fs@` are the two most relevant lists. Reading historic threads is one of the best ways to pick up the institutional knowledge that books do not capture.

Commit history on GitHub mirrors. The FreeBSD source tree has a long, well-annotated commit history. For any file you open, running `git log --follow` against its mirror will often reveal the rationale behind design choices, the bugs that shaped the current code, and the people who maintain it. Historical context makes the present code much easier to read.

The Transactions of the FreeBSD Developer Summit. Several summits have included storage-focused sessions. Recordings and slides, when available, are excellent for picking up the state of the art and the open design debates.

Reading other operating systems’ storage stacks. Once you know FreeBSD’s storage path, Linux’s block layer, Illumos’s SD framework, and macOS’s IOKit storage classes all become comprehensible in a way they probably were not before. The specific APIs differ, but the fundamental shapes, BIOS or their equivalents, filesystems above, hardware below, are universal.

Testing frameworks for kernel code. The `kyua(1)` harness runs regression tests against real kernels. The `/usr/tests/sys/geom/` tree has examples of what well-written tests for storage code look like; reading them builds both testing instincts and confidence that your code is right.

FreeBSD Foundation blog posts. The Foundation funds several storage-related projects and publishes readable summaries that complement the source tree.

End of Chapter 27. Close your lab logbook, make sure your driver is unloaded and your mount points are released, and take a break before Chapter 28.

You have just written a storage driver, mounted a filesystem on it, traced data through the buffer cache, into GEOM, through your strategy function, and back again. That is a real accomplishment. Rest on it for a moment before you turn the page.

28 Writing a Network Driver

28.1 Introduction

In the previous chapter you built a storage driver. A filesystem sat on top of it, a buffer cache fed it BIO requests, and your code delivered blocks of data to a piece of RAM and back. That was already a step away from the character-device world of earlier chapters, because a storage driver is not polled by a single process holding a file descriptor. It is driven by many layers above it, all cooperating to turn `write(2)` calls into durable blocks, and your driver had to sit quietly at the bottom of that chain and honour each request in turn.

A network driver is a third kind of animal. It is not a stream of bytes to one process, like a character device. It is not a block-addressable surface for a filesystem to mount, like a storage device. It is an **interface**. It sits between the machine's network stack on one side and a medium, real or simulated, on the other side. Packets arrive on that medium and the driver turns them into mbufs and hands them up to the stack. Packets leave the stack in the form of mbufs and the driver turns them into bits on the wire, or into whatever stand-in for a wire you have chosen to use. Link state changes, and the driver reports that. Media speed changes, and the driver reports that too. The user types `ifconfig mynet0 up`, and the kernel routes that request through `if_ioctl` into your code. The kernel is expecting a particular shape of cooperation, not a particular sequence of reads and writes.

This chapter teaches you that shape. You will learn what FreeBSD expects a network driver to be. You will learn the core object that represents an interface in the kernel, the struct called `ifnet`, along with the modern `if_t` opaque handle that wraps it. You will learn how to allocate an `ifnet`, how to register it with the stack, how to expose it as a named interface that `ifconfig` can see. You will learn how packets enter your driver through the transmit callback and how you push packets the other way into the stack through `if_input`. You will learn how mbufs carry those packets, how link state and media state are reported, how flags like `IFF_UP` and `IFF_DRV_RUNNING` are used, and how a driver detaches cleanly when it is unloaded. You will finish the chapter with a working pseudo-Ethernet driver called `mynet` that you can load, configure, exercise with `ping`, `tcpdump`, and `netstat`, and then unload without leaving anything behind.

The driver you will build is small on purpose. Real Ethernet drivers in modern FreeBSD are usually written against `iflib(9)`, the shared framework that takes care of ring buffers, interrupt moderation, and packet-steering for most production NICs. That machinery is wonderful when you are shipping a driver for a 100-gigabit card, and we will return to it in later chapters. But it is too much scaffolding to hide the core ideas behind. To teach you what a network driver really is, we will write the classical, pre-`iflib` form: a plain `ifnet` driver with its own transmit function and its own receive path. Once you understand that clearly, `iflib` will feel like a layer of convenience on top of something you already know.

Like Chapter 27, this chapter is long because the topic is layered. Unlike `/dev` drivers, network drivers are wrapped in a vocabulary of their own: Ethernet frames, interface cloners, link state, media descriptors, `if_transmit`, `if_input`, `bpfattach`, `ether_ifattach`. We will introduce that vocabulary carefully, one concept at a time, and we will ground every concept in code from the real FreeBSD tree. You will see where

`epair(4)`, `disc(4)`, and the UFS stack loan patterns that we can adapt for our own driver. By the end you will recognise the shape of a network driver in any FreeBSD source file you open.

The goal is not a production NIC driver. The goal is to give you a complete, honest, correct understanding of the layer between a piece of hardware and the FreeBSD network stack, built up through prose, code, and practice. Once that mental model is solid, reading `if_em.c`, `if_bge.c`, or `if_ixl.c` becomes a matter of recognising patterns and looking up the unfamiliar parts. Without that mental model, they look like a storm of macros and bit operations. With it, they look like one more driver that does the same things your `mynet` driver does, only with hardware underneath.

Take your time. Open a FreeBSD shell while you read. Keep a lab logbook. Think of the network stack not as a black box above your code, but as a peer that expects a clear, contractual handshake with the driver. Your job is to fulfil that contract cleanly.

28.2 Reader Guidance: How to Use This Chapter

This chapter continues the pattern established in Chapter 27: long, cumulative, deliberately paced. The topic is new and the vocabulary is new, so we will move a little more carefully than usual through the opening sections before we let you type code.

If you choose the **reading-only path**, plan for around two to three focused hours. You will come away with a clear mental model of what a network driver is, how it fits into FreeBSD's network stack, and what the code in real drivers is doing. This is a legitimate way to use the chapter on a first pass, and it is often the right choice on a day when you do not have time to rebuild a kernel module.

If you choose the **reading-plus-labs path**, plan for about five to eight hours spread across one or two evenings. You will write, build, and load a working pseudo-Ethernet driver, bring it up with `ifconfig`, watch its counters move, feed packets into it from `ping`, look at them with `tcpdump`, and then shut everything down and unload the module cleanly. The labs are designed to be safe on any recent FreeBSD 14.3 system, including a virtual machine.

If you choose the **reading-plus-labs-plus-challenges path**, plan for a weekend or a handful of evenings. The challenges extend the driver in directions that matter in practice: adding a real simulated link partner with a shared queue between two interfaces, supporting different link states, exposing a `sysctl` to inject errors, and measuring the behaviour under `iperf3`. Each challenge is self-contained and uses only what the chapter has already covered.

Regardless of the path you choose, do not skip the troubleshooting section near the end. Network drivers fail in a handful of characteristic ways, and learning to recognise those patterns is more valuable in the long run than memorising the names of every function in `ifnet`. The troubleshooting material is placed late for readability, but you may find yourself returning to it while running the labs.

A word on prerequisites. You should be comfortable with everything from Chapter 26 and Chapter 27: writing a kernel module, allocating and freeing a softc, reasoning about the load and unload path, and testing your work under `kldload` and `kldunload`. You should also be comfortable enough with FreeBSD's userland to run `ifconfig`, `netstat -in`, `tcpdump`, and `ping` without stopping to check the flags. If any of that feels uncertain, a quick skim of the corresponding earlier chapters will save you time later.

You should work on a throwaway FreeBSD 14.3 machine. A dedicated virtual machine is the best option, because network drivers, by their nature, can interact with the host system's routing tables and interface list. A small lab VM lets you experiment without worrying that you are going to confuse your main system. A snapshot before you begin is cheap insurance.

28.2.1 Work Section by Section

The chapter is organised as a progression. Section 1 explains what a network driver does and how it differs from the character and storage drivers you have already written. Section 2 introduces the `ifnet` object, the central data structure of the entire networking subsystem. Section 3 walks through allocation, naming, and registration of an interface, including interface cloners. Section 4 handles the transmit path, from `if_transmit` through mbuf processing. Section 5 handles the receive path, including `if_input` and simulated packet generation. Section 6 covers media descriptors, interface flags, and link-state notifications. Section 7 shows you how to test the driver with FreeBSD's standard networking tools. Section 8 closes with clean detach, module unload, and refactoring advice.

You are meant to read these sections in order. Each one assumes the previous ones are fresh in your mind, and the labs build on each other. If you jump in the middle, pieces will look strange.

28.2.2 Type the Code

Typing remains the most effective way to internalise kernel idioms. The companion files under `examples/part-06/ch28-network-driver/` exist so that you can check your work, not so that you can skip the typing. Reading code is not the same as writing it, and reading a network driver is particularly easy to do passively because the code often looks like a long switch statement. Writing it forces you to think about each branch.

28.2.3 Open the FreeBSD Source Tree

You will be asked several times to open real FreeBSD source files, not only the companion examples. The files of interest for this chapter include `/usr/src/sys/net/if.h`, `/usr/src/sys/net/if_var.h`, `/usr/src/sys/net/if_disc.c`, `/usr/src/sys/net/if_epair.c`, `/usr/src/sys/net/if_ethersubr.c`, `/usr/src/sys/net/if_clone.c`, `/usr/src/sys/net/if_media.h`, and `/usr/src/sys/sys/mbuf.h`. Each of these is a primary reference, and the prose in this chapter refers back to them repeatedly. If you have not already cloned or installed the 14.3 source tree, now is a good moment to do so.

28.2.4 Use Your Lab Logbook

Keep the logbook you started in Chapter 26 open while you work. You will want to record the `ifconfig` output before and after you load the module, the exact commands you use to send traffic, the counters reported by `netstat -in`, the output of `tcpdump -i mynet0`, and any warnings or panics. Network work is particularly friendly to logbooks because the same command, `ifconfig mynet0`, produces different output at different points in the load-configure-use-unload cycle, and seeing those differences in your own notes makes the concepts stick.

28.2.5 Pace Yourself

If your understanding blurs during a particular section, stop. Re-read the previous subsection. Try a small experiment, for example `ifconfig lo0` or `netstat -in` to see a real interface, and think about how it corresponds to what the chapter is teaching. Network programming in the kernel rewards slow, deliberate exposure. Skimming the chapter for terms to recognise later is much less useful than reading one section well, doing one lab, and moving on.

28.3 How to Get the Most Out of This Chapter

The chapter is structured so that every section adds exactly one new concept on top of what came before. To make the most of that structure, treat the chapter as a workshop rather than as a reference. You are not here to find a quick answer. You are here to build a correct mental model of what an interface is, how a driver talks to the kernel, and how the network stack talks back.

28.3.1 Work in Sections

Do not read the whole chapter end to end without stopping. Read one section, then pause. Try the experiment or lab that goes with it. Look at the related FreeBSD source. Write a few lines in your logbook. Only then move on. Networking in the kernel is strongly cumulative, and skipping ahead usually means that you will be confused about the next thing for a reason that was explained two sections ago.

28.3.2 Keep the Driver Running

Once you have loaded the driver in Section 3, keep it loaded as much as possible while you read. Modify it, reload, poke it with `ifconfig`, feed it packets with `ping`, watch them with `tcpdump`. Having a live, observable example is far more valuable than any amount of reading, especially for network code, because the feedback loop is fast: the kernel either accepts your configuration or refuses it, and the counters either move or they do not.

28.3.3 Consult Manual Pages

FreeBSD's manual pages are part of the teaching material, not a separate formality. Section 9 of the manual is where the kernel interfaces live. We will refer in this chapter to pages such as `ifnet(9)`, `mbuf(9)`, `ifmedia(9)`, `ether(9)`, and `ng_ether(4)`, and to userland pages such as `ifconfig(8)`, `netstat(1)`, `tcpdump(1)`, `ping(8)`, and `ngctl(8)`. Read them alongside this chapter. They are shorter than they look, and they are written by the same community that wrote the kernel you are learning about.

28.3.4 Type the Code, Then Mutate It

When you build the driver from the companion examples, type it first. Once it works, start changing things. Rename a method and watch the build fail. Remove an `if` branch in the transmit function and watch what happens under `ping`. Hardcode a smaller MTU and watch `ifconfig` react. Kernel code becomes understandable

through deliberate mutation far more than through pure reading, and network code is particularly well-suited to mutation because every change produces an immediately visible effect in `ifconfig` or `netstat`.

28.3.5 Trust the Tooling

FreeBSD gives you a wealth of tools for inspecting the network stack: `ifconfig`, `netstat`, `tcpdump`, `ngctl`, `sysctl net.`, `arp`, `ndp`. Use them. When something goes wrong, the first move is almost never to read more source. It is to ask the system what state it is in. A minute with `ifconfig mynet0` and `netstat -in` is often more informative than five minutes of `grep`.

28.3.6 Take Breaks

Network code is full of small, precise steps. A missed flag or an unset callback will produce behaviour that looks mysterious until you stop, breathe, and trace the data flow again. Two or three focused hours are usually more productive than a seven-hour sprint. If you catch yourself making the same typo three times, or copy-pasting without reading, that is your cue to stand up for ten minutes.

With those habits in place, let us begin.

28.4 Section 1: What a Network Driver Does

A network driver has one job that sounds simple and turns out to be layered: it moves packets between a transport and the FreeBSD network stack. Everything else follows from that. To understand what that sentence really means, we need to slow down and examine each of its pieces. What is a packet? What is a transport? What exactly is “the stack”? And how does a driver sit between them without becoming a bottleneck or a source of subtle bugs?

28.4.1 A Packet, in the Kernel

In userland you rarely deal with raw packets. You open a socket, call `send` or `recv`, and the kernel takes care of encapsulating your payload in TCP, wrapping that in IP, adding an Ethernet header, and finally handing the whole construct to a driver. In the kernel, the same packet is represented by a linked list of structures called **mbufs**. An mbuf is a small memory cell, typically 256 bytes, that holds packet data and a small header. If the packet is larger than a single mbuf can hold, the kernel chains multiple mbufs together through a `m_next` pointer, and the total length of the payload is recorded in `m->m_pkthdr.len`. If the packet does not fit in a single mbuf cluster, the kernel uses external buffers referenced by the mbuf, through a mechanism we will revisit in later chapters.

From the driver’s perspective, a packet is almost always presented as an mbuf chain, and the first mbuf carries the packet header. That first mbuf has `M_PKTHDR` set in its flags, which tells you that `m->m_pkthdr` contains valid fields such as the total packet length, the VLAN tag, checksum flags, and the receive interface. Every driver that handles transmitted packets begins by inspecting the mbuf it has been handed, and every driver that delivers received packets begins by building a correctly shaped mbuf.

We will cover mbuf construction and teardown in more detail in Sections 4 and 5. For now, the vocabulary is what matters. A mbuf is a packet. A mbuf chain is a packet whose payload spans several mbufs. The first mbuf in a chain carries the packet header. The rest of the chain continues the payload, and each mbuf points to the next through `m_next`.

28.4.2 A Transport

The transport is whatever the driver talks to on the hardware side. For a physical Ethernet NIC it is the actual wire, reached through a combination of DMA buffers, hardware rings, and interrupts from the chip. For a USB Ethernet adapter it is the USB endpoint pipeline we introduced in Chapter 26. For a wireless card it is the radio. For a pseudo-device, and that is what we will build in this chapter, the transport is simulated: we will pretend that a packet we transmit shows up on some other virtual wire, and we will pretend that incoming packets arrive from it at regular intervals driven by a timer.

The beauty of the `ifnet` abstraction is that the network stack does not care which of these transports you have. The stack sees an interface. It gives the interface mbufs to transmit. It expects the interface to hand it mbufs that were received. Whether the packets actually travel across Category 6 cable, radio waves, a USB bus, or a piece of memory we control, the surface is the same. That uniformity is what lets FreeBSD support dozens of network devices without rewriting its network code for each one.

28.4.3 The Network Stack

“The stack” is a shorthand for the collection of code that sits above the driver and implements the protocols. Layer by layer, from the lowest to the highest: Ethernet framing, ARP and neighbour discovery, IPv4 and IPv6, TCP and UDP, socket buffers, and the system call layer that translates `send` and `recv` into stack operations. In FreeBSD, the code lives in `/usr/src/sys/net/`, `/usr/src/sys/netinet/`, `/usr/src/sys/netinet6/`, and related directories, and it communicates with drivers through a small, well-defined set of function pointers carried on every `ifnet`.

For this chapter, you do not need to know the inside of the stack. You need to know its outer interface as seen by a driver. That interface is:

- The stack calls your transmit function, `if_transmit`, and hands you a mbuf. Your job is to turn that mbuf into something the transport will accept.
- The stack calls your ioctl handler, `if_ioctl`, in response to commands from userland such as `ifconfig mynet0 up` or `ifconfig mynet0 mtu 1400`. Your job is to honour the request or return a reasonable error.
- The stack calls your init function, `if_init`, when the interface transitions to an up state. Your job is to prepare the transport for use.
- You call `ifp->if_input(ifp, m)` or, in the modern idiom, `if_input(ifp, m)`, to hand a received packet to the stack. Your job is to ensure the mbuf is well-formed and the packet is complete.

That is the contract. The rest is detail.

28.4.4 How a Network Driver Differs From a Character Driver

You already built character drivers in Chapters 14 and 18. A character driver sits inside `/dev/`, it is opened by userland through `open(2)`, and it exchanges bytes with one or more processes through `read(2)` and `write(2)`. It has a `cdevsw` table. It is polled and pushed by whoever opens it.

A network driver is none of those things. It does not live in `/dev/`. It is not `open(2)`ed by a process. There is no `cdevsw`. The closest thing to a user-visible file handle for a network interface is the socket that is bound to it, and even that is mediated by the stack, not by the driver.

Instead of a `cdevsw`, a network driver has a `struct ifnet`. Instead of `d_read`, it has `if_input`, but on the other end: the driver calls it, rather than having it called by userland. Instead of `d_write`, it has `if_transmit`, called by the stack. Instead of `d_ioctl`, it has `if_ioctl`, called by the stack in response to `ifconfig` and related tools. The top-level structure looks similar, but the relationships between actors are different. In a character driver you wait for reads and writes from userland. In a network driver you are embedded in a pipeline where the stack is your main collaborator, and userland is a spectator rather than a direct peer.

This shift in perspective is worth internalising before you write any code. When something goes wrong in a character driver, the question is usually “what did userland do?” When something goes wrong in a network driver, the question is usually “what did the stack expect my driver to do, and how did I fail to do it?”

28.4.5 How a Network Driver Differs From a Storage Driver

A storage driver, as you saw in Chapter 27, is also not a `/dev/` endpoint in the usual sense. It does expose a block device node, but access to it is almost always mediated by a filesystem sitting on top. Requests come down as BIOs, the driver handles them, and completion is signalled by `biodone(bp)`.

A network driver shares the “I sit below a subsystem, not next to userland” shape of a storage driver, but the subsystem above it is very different. The storage subsystem is deeply synchronous at the BIO level, in the sense that every request has a well-defined completion event. Network traffic is not like that. A driver transmits a packet, but there is no per-packet completion callback bubbling up from the driver to any specific requester. The stack trusts the driver to succeed or fail cleanly, increments counters, and moves on. Likewise, received packets are not replies to specific earlier transmissions: they just arrive, and the driver must funnel them into `if_input` whenever they show up.

Another difference is concurrency. A storage driver usually has a single BIO path and handles each BIO in turn. A network driver is frequently called from multiple CPU contexts at once, because the stack serves many sockets in parallel, and modern hardware delivers receive events on multiple queues. We will not cover that complexity in this chapter, but you should already be aware that locking conventions for network drivers are strict. The `mynet` driver we will build is small enough that a single mutex suffices, but even then the discipline around when to take it, and when to drop it before calling up, matters.

28.4.6 The Role of `ifconfig`, `netstat`, and `tcpdump`

Every FreeBSD user knows `ifconfig`. From the network driver author’s perspective, `ifconfig` is the primary way the kernel expects user commands to reach your driver. When the user runs `ifconfig mynet0 up`, the kernel translates that into a `SIOCSIFFLAGS` ioctl on the interface whose name is `mynet0`. The call arrives in

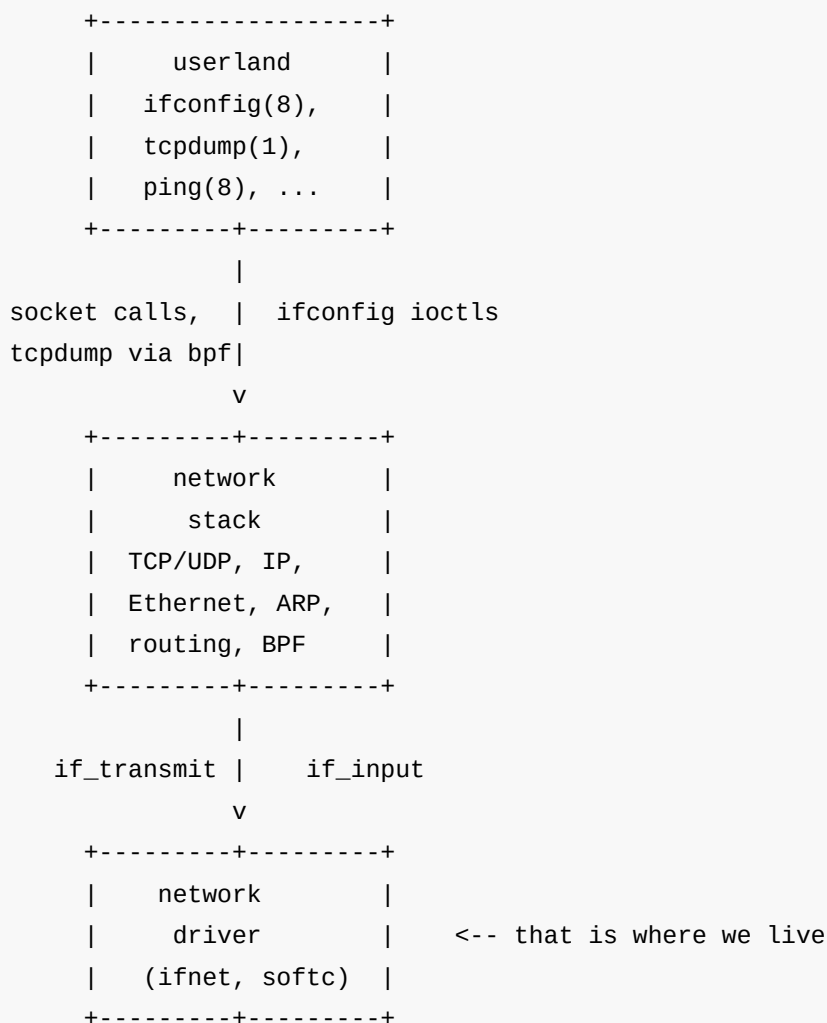
your `if_ioctl` callback, and you decide what to do with it. The symmetry between the userland command and the kernel-side callback is almost one-to-one.

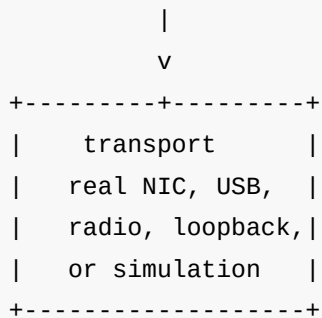
`netstat -in` asks the kernel for the interface statistics carried on every `ifnet`. Your driver updates those counters by calling `if_inc_counter(ifp, IFCOUNTER_*, n)` at the appropriate moments in the transmit and receive paths. The set of counters is defined in `/usr/src/sys/net/if.h` and includes `IFCOUNTER_IPACKETS`, `IFCOUNTER_OPACKETS`, `IFCOUNTER_IBYTES`, `IFCOUNTER_OBYTES`, `IFCOUNTER_IERRORS`, `IFCOUNTER_OERRORS`, `IFCOUNTER_IMCASTS`, `IFCOUNTER_OMCASTS`, and `IFCOUNTER_OQDROPS`, among others. These counters are what users see in `netstat` and `systat`.

`tcpdump` relies on a separate subsystem called the Berkeley Packet Filter, or BPF. Every interface that wants to be visible to `tcpdump` must register with BPF through `bpfattach()`, and every packet that the driver transmits or receives must be presented to BPF through `BPF_MTAP()` or `bpf_mtap2()` before being either sent out or handed up. We will do this in our driver. It is one of the small courtesies you pay to the rest of the system so that the tools work.

28.4.7 A Useful Picture

It is worth closing the section with a diagram. The picture below shows how the pieces we have described fit together. Do not memorise it yet. Just get used to the shape. We will come back to every box in later sections.





The boxes above the driver are the stack and userland. The box below is the transport. Your driver, on that middle line, is the only place in the system where a `struct ifnet` meets a `struct mbuf` meets a wire. That is your territory.

28.4.8 Tracing a Packet Through the Stack

It is useful to follow one specific packet from birth to death, because that fixes the relationships in the diagram above to real code. Let us trace an outbound ICMP echo request generated by `ping 192.0.2.99` on an interface named `mynet0` that has been assigned the address `192.0.2.1/24`.

The `ping(8)` program opens a raw ICMP socket and writes an echo request payload through `sendto(2)`. Inside the kernel, the socket layer in `/usr/src/sys/kern/uipc_socket.c` copies the payload into a fresh `mbuf` chain. The socket is unconnected, so each write carries a destination address that the socket layer forwards to the protocol layer. The protocol layer, in `/usr/src/sys/netinet/raw_ip.c`, attaches an IP header and calls `ip_output` in `/usr/src/sys/netinet/ip_output.c`. `ip_output` performs the route lookup and finds a routing entry that points at `mynet0`. It also notices that the destination is not the broadcast address and not an on-link neighbour whose MAC it already knows, so it must trigger ARP.

At this point the IP layer calls `ether_output`, defined in `/usr/src/sys/net/if_etherubr.c`. `ether_output` notices that the next-hop address is unresolved and issues an ARP request first. The ARP machinery, in `/usr/src/sys/netinet/if_ether.c`, constructs a broadcast ARP frame, wraps it in a new `mbuf`, and calls `ether_output_frame`, which in turn calls `ifp->if_transmit`. That is our `mynet_transmit` function. The `mbuf` we receive in the transmit callback already contains a complete Ethernet frame: destination MAC `ff:ff:ff:ff:ff:ff`, source MAC our fabricated address, EtherType `0x0806` (ARP), and the ARP payload.

We do what every driver does at that point: validate, count, tap BPF, and release. Because we are a pseudo-driver, we free the frame rather than handing it to hardware. In a real NIC driver we would hand the `mbuf` to DMA and free it later when the completion interrupt fires. Either way, the `mbuf` has reached the end of its life from the driver's perspective.

While the ARP request hangs unanswered, the stack queues the original ICMP payload in the ARP pending queue. When the ARP reply does not arrive within a configurable timeout, the stack gives up on that packet and increments `IFCOUNTER_OQDROPS`. In our pseudo-driver, of course, no reply will ever come because there is nothing on the other end of the simulated wire. That is why `ping` eventually prints “100.0% packet loss” and exits without success. The absence of a reply is not a bug in our driver; it is a property of the transport we have chosen to simulate.

Now trace the reverse path. The synthetic ARP request we generate every second in `mynet_rx_timer` starts life as memory we allocate with `MGETHDR` inside our driver. We fill in the Ethernet header, the ARP header, and the ARP payload. We tap BPF. We call `if_input`, which dereferences `ifp->if_input` and lands in `ether_input`. `ether_input` looks at the `EtherType` and dispatches the payload to `arpintr` (or its modern equivalent, a direct call from within `ether_demux`). The ARP code inspects the sender and target IPs, notices that the target is not us, and silently drops the frame. Done.

In both directions the driver is a brief pass-through: a mbuf arrives, a mbuf departs, counters move, and BPF sees everything in between. That simplicity is deceptive, because every step has a contract that must not be broken, but the pattern is genuinely this short.

28.4.9 The Queue Disciplines Above You

You do not see them from the driver, but the stack has queue disciplines that govern how packets are delivered to `if_transmit`. Historically, drivers had an `if_start` callback and the stack would place packets on an internal queue (`if_snd`) for later dispatch. Modern drivers use `if_transmit` and receive the mbuf directly, letting the driver or the `drbr(9)` helper library manage any per-CPU queues internally.

In practice, almost all modern drivers use `if_transmit` and let the stack hand them packets one at a time. Because `if_transmit` is called on the thread that produced the packet (typically a TCP retransmit timer or the thread that wrote to the socket), the transmit path is usually on a regular kernel thread with preemption enabled. This matters because it means you generally cannot assume that `transmit` runs at elevated priority, and you must not hold a mutex across a long operation.

A small number of drivers still use the classic `if_start` model, where the stack fills a queue and calls `if_start` to drain it. That model is simpler for drivers with simple hardware queueing, but less flexible under load. `epair(4)` uses `if_transmit` directly. `disc(4)` implements its own tiny discoutput that is called from `ether_output`'s pre-transmit path. Most real NIC drivers use `if_transmit` with internal per-CPU queues powered by `drbr`.

For `mynet`, we use `if_transmit` and no internal queue. This is the simplest possible design and it matches what a minimal real driver would do for low-bandwidth links.

28.4.10 A Note on Packet-Tap Visibility

Packet taps, discussed in the next few sections, are one of the key reasons a network driver feels different from a character driver. A character driver's traffic is invisible to external observers, because there is no analogue of `tcpdump` for arbitrary `/dev/` traffic. A network driver's traffic, by contrast, is observable at multiple levels simultaneously: BPF captures at the driver level, `pflog` at the packet-filter level, interface counters at the kernel level, and socket buffers at the userland level. All of that observability is free for the driver author, provided the driver taps BPF and updates counters at the correct points.

This unusual level of external visibility is a blessing for debugging. When you cannot tell why a packet did or did not flow, you can almost always answer the question with a combination of `tcpdump`, `netstat`, `arp`, and `route monitor`. That is a capable toolset, and we will put it to work throughout the labs.

28.4.11 Wrapping Up Section 1

We have set the scene. A network driver moves mbufs between the stack and a transport. It presents a standardised interface called `ifnet`. It is driven by calls from the stack into fixed callbacks. It pushes received traffic up through `if_input`. It is visible to `ifconfig`, to `netstat`, and to `tcpdump` through a handful of kernel conventions.

With that rough shape in mind, we can look at the `ifnet` object itself. That is the subject of Section 2.

28.5 Section 2: Introducing `ifnet`

Every network interface on a running FreeBSD system is represented in the kernel by a `struct ifnet`. That structure is the central object of the networking subsystem. When `ifconfig` lists interfaces, it is essentially iterating over a list of `ifnet` objects. When the stack picks a route, it eventually lands on an `ifnet` and calls its transmit function. When a driver reports link state, it updates fields inside an `ifnet`. Learning `ifnet` is not optional. Everything else in this chapter is built on it.

28.5.1 Where `ifnet` Lives

The declaration of `struct ifnet` is in `/usr/src/sys/net/if_var.h`. Over the years FreeBSD has moved toward treating it as opaque, and the recommended way to refer to it in new driver code is through the typedef `if_t`, which is a pointer to the underlying structure:

```
typedef struct ifnet *if_t;
```

Old driver code reaches directly into `ifp->if_softc`, `ifp->if_flags`, `ifp->if_mtu`, and similar fields. New driver code prefers accessor functions such as `if_setsoftc(ifp, sc)`, `if_getflags(ifp)`, `if_setflags(ifp, flags)`, and `if_setmtu(ifp, mtu)`. Both styles still exist in the tree, and existing drivers such as `/usr/src/sys/net/if_disc.c` still use direct field access. The opaque style is the direction the kernel is moving, but you will see both for years to come.

Throughout this chapter we will use whatever is clearest in the given context. When the direct-field style makes the code smaller and easier to read, we will use it. When an accessor makes the intent clearer, we will use that. You should be able to read either form.

28.5.2 The Minimum Fields You Care About

A `struct ifnet` has dozens of fields. The good news is that a driver only touches a handful of them directly. The fields you will set or inspect in the driver we build are, broadly:

- **Identity.** `if_softc` points back at your driver's private structure, `if_xname` is the interface name (for example `mynet0`), `if_dname` is the family name ("`mynet`"), and `if_dunit` is the unit number.
- **Capabilities and counts.** `if_mtu` is the maximum transmission unit, `if_baudrate` is a reported line rate in bits per second, `if_capabilities` and `if_capenable` describe offload capabilities like VLAN tagging and checksum offload.

- **Flags.** `if_flags` holds the interface-level flags set by userland: `IFF_UP`, `IFF_BROADCAST`, `IFF_SIMPLEX`, `IFF_MULTICAST`, `IFF_POINTOPOINT`, `IFF_LOOPBACK`. `if_drv_flags` holds driver-private flags; the most important is `IFF_DRV_RUNNING`, which means the driver has allocated its per-interface resources and is ready to move traffic.
- **Callbacks.** `if_init`, `if_ioctl`, `if_transmit`, `if_qflush`, and `if_input` are the function pointers the stack invokes. Some of these have long-standing direct fields; the accessor equivalents are `if_setinitfn`, `if_setioctlnfn`, `if_settransmitfn`, `if_setqflushfn`, and `if_setinputfn`.
- **Statistics.** The per-counter accessors `if_inc_counter(ifp, IFCOUNTER_*, n)` increment the counters that `netstat -in` displays.
- **BPF hook.** `if_bpf` is an opaque pointer used by BPF. Your driver does not normally read it directly, but when you call `bpfattach(ifp, ...)` and `BPF_MTAP(ifp, m)`, the system will manage it.
- **Media and link state.** `ifmedia` lives in your `softc`, not in the `ifnet`, but the interface reports link state by calling `if_link_state_change(ifp, LINK_STATE_*)`.

If the list looks long, remember that most drivers set each field once and then leave it alone. The work of a driver is in the callbacks, not in the `ifnet` fields themselves.

28.5.3 The `ifnet` Life Cycle

A struct `ifnet` goes through the same high-level stages as a `device_t` or a `softc`: allocation, configuration, registration, active life, and teardown. The call graph is:

```

if_alloc(type)          -> returns a fresh ifnet, not yet attached
|
| configure fields
|  if_initname()         set the name
|  if_setsoftc()         point at your softc
|  if_setinitfn()        set if_init callback
|  if_setioctlnfn()      set if_ioctl
|  if_settransmitfn()    set if_transmit
|  if_setqflushfn()      set if_qflush
|  if_setflagbits()      set IFF_BROADCAST, etc.
|  if_setmtu()           set MTU
v
if_attach(ifp)           OR ether_ifattach(ifp, mac)
|
| live interface
|  if_transmit called by stack
|  if_ioctl called by stack
|  driver calls if_input to deliver received packets
|  driver calls if_link_state_change on link events
v
ether_ifdetach(ifp)      OR if_detach(ifp)
|
| finish teardown

```

```
v
if_free(ifp)
```

There are two common variants of the attach and detach calls. A plain pseudo-interface that does not need Ethernet wiring uses `if_attach` and `if_detach`. A pseudo or real Ethernet interface uses `ether_ifattach` and `ether_ifdetach` instead. The Ethernet variants wrap the plain ones and add the extra setup needed for a layer-2 Ethernet interface, including `bpfattach`, address registration, and wiring up `ifp->if_input` and `ifp->if_output` to `ether_input` and `ether_output`. We will use the Ethernet variant in our driver because it gives us a familiar MAC-addressed interface that `ifconfig`, `ping`, and `tcpdump` all understand without special treatment.

If you open `/usr/src/sys/net/if_etherubr.c` and look at `ether_ifattach`, you will see precisely this logic: set `if_addr_len` to `ETHER_ADDR_LEN`, set `if_hdr_len` to `ETHER_HDR_LEN`, set `if_mtu` to `ETHERMTU`, call `if_attach`, then install the common Ethernet input and output routines and finally call `bpfattach`. It is worth reading that function in full. It is short and it shows you exactly what a driver gets for free by using `ether_ifattach` instead of the bare `if_attach`.

28.5.4 Why ifnet Is Not a cdevsw

It is tempting to see `ifnet` as just “a `cdevsw` for networking”. It is not. A `cdevsw` is an entry table used by `devfs` to dispatch `read`, `write`, `ioctl`, `open`, and `close` from userland to a driver. An `ifnet` is the first-class object the network stack itself maintains for every interface. Even if no userland process has ever touched the interface, the stack still cares about its `ifnet`, because routing tables, ARP, and packet forwarding all depend on it.

You can see this if you think about how `ifconfig` talks to the kernel. It does not open `/dev/mynet0`. It opens a socket and issues `ioctls` on that socket, passing the interface name as an argument. The kernel then looks up the `ifnet` by name and invokes `if_ioctl` on it. There is no file descriptor pointing at your interface on the userland side. The interface is a stack-level entity, not a `/dev/` entity.

That is why we need a whole new object: because networking requires a persistent, kernel-internal handle that exists regardless of which process is doing what. `ifnet` is that handle.

28.5.5 Pseudo-Interfaces vs Real NIC Interfaces

Every interface in the kernel, pseudo or real, has an `ifnet`. The loopback interface `lo0` has one. The `disc` interface we will study has one. Every `emX` Ethernet adapter has one. Every `wlanX` wireless interface has one. The `ifnet` is the universal currency.

Pseudo-interfaces differ from real NICs in how they are instantiated. A real NIC interface is created by the driver’s attach method during bus probe, the same way the USB and PCI drivers in Chapter 26 attach their devices. A pseudo-interface is created at module load time, or on demand through `ifconfig mynet0 create`, via a mechanism called the **interface cloner**. We will use an interface cloner for `mynet`, which means users will be able to create interfaces dynamically, just as they can create `epair` interfaces today:

```
# ifconfig mynet create
mynet0
# ifconfig mynet0 up
# ifconfig mynet0
mynet0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> metric 0 mtu 1500
```

We will describe cloners in Section 3. For now, it is enough to know that cloning is how a module contributes one or more `ifnet` objects to the running system at the user's request.

28.5.6 A Closer Look at Key `ifnet` Fields

Because the `ifnet` is the structure your driver writes to most often, it helps to survey a few of its fields in slightly more depth before we open the code. You do not need to memorise the complete declaration. What you need is enough familiarity with the layout to read driver code without constantly flipping back to `if_var.h`.

`if_xname` is a character array holding the interface's user-visible name, such as `mynet0`. It is set by `if_initname` and from that moment is treated by the stack as read-only. When you read `ifconfig -a` output, every line that begins with an interface name is printing a copy of `if_xname`.

`if_dname` and `if_dunit` record the driver family name and the unit number separately. `if_dname` is "mynet" for every instance of our driver, and `if_dunit` is 0 for `mynet0`, 1 for `mynet1`, and so on. The network stack uses these fields to index the interface into various hashes, and `ifconfig` uses them when matching an interface name to a driver family.

`if_softc` is the back-pointer to your driver's private per-interface structure. Every callback the stack invokes will pass an `ifp` argument, and the first thing most callbacks do is pull the `softc` out of `ifp->if_softc` (or `if_get_softc(ifp)`). If you forget to set `if_softc` during creation, your callbacks will dereference a NULL pointer and the kernel will panic.

`if_type` is the type constant from `/usr/src/sys/net/if_types.h`. `IFT_ETHER` for an Ethernet-like interface, `IFT_LOOP` for loopback, `IFT_IEEE80211` for wireless, `IFT_TUNNEL` for a generic tunnel, and dozens of others. The stack occasionally specialises behaviour based on `if_type`, for example in deciding how to format a link-layer address for display.

`if_addr_len` and `if_hdr_len` describe the link-layer address length (six bytes for Ethernet, eight bytes for InfiniBand, zero for a pure L3 tunnel) and the link-layer header length (14 bytes for plain Ethernet, 22 for tagged Ethernet). `ether_ifattach` sets both of these for you with the Ethernet defaults. Other link-layer helpers set them with their own values.

`if_flags` is a bitmask of user-visible flags like `IFF_UP` and `IFF_BROADCAST`. `if_drv_flags` is a bitmask of driver-private flags like `IFF_DRV_RUNNING`. They are separate because they have different access rules. The user can write `if_flags`; only the driver writes `if_drv_flags`. Mixing them is a classic bug.

`if_capabilities` and `if_capenable` describe offload features. `if_capabilities` is what the hardware claims it can do. `if_capenable` is what is currently turned on. The split allows userland to toggle offloads at runtime through `ifconfig mynet0 -rxcsom` or `ifconfig mynet0 +tso`, and the driver to honour that choice. We will see this interact with `SIOCSIFCAP` in Section 6.

`if_mtu` is the maximum transmission unit in bytes. It is the largest L3 payload the interface can carry, not counting the link-layer header. Ethernet defaults to 1500. Jumbo-frame Ethernet typically supports 9000 or 9216. `if_baudrate` is an informational line-rate field in bits per second; it is advisory only.

`if_init` is a function pointer invoked when the interface transitions to the up state. Its signature is `void (*)(void *softc)`. `if_ioctl` is invoked for socket ioctls destined at this interface; signature `int (*)(struct ifnet *, u_long, caddr_t)`. `if_transmit` is invoked to send a packet; signature `int (*)(struct ifnet *, struct mbuf *)`. `if_qflush` is invoked to flush driver-private queues; signature `void (*)(struct ifnet *)`. `if_input` is a function pointer in the other direction: the driver calls it (usually through the `if_input(ifp, m)` helper) to hand a received mbuf to the stack.

`if_snd` is the legacy send queue, used by drivers that still have an `if_start` callback rather than an `if_transmit`. For modern drivers with `if_transmit`, `if_snd` is unused. Most textbook examples you will read in the tree (including our `if_disc.c` reference) do not touch `if_snd` any more.

`if_bpf` is the BPF attachment pointer. BPF itself manages the value; drivers treat it as opaque. `BPF_MTAP` and related macros use it internally.

`if_data` is a large structure carrying per-interface statistics, media descriptors, and miscellaneous fields. Modern drivers avoid touching `if_data` directly and instead go through `if_inc_counter` and friends. The `if_data` structure is still there for backward compatibility and for userland-visible stats.

This is far from an exhaustive list; `struct ifnet` has more than fifty fields in total. But the ones above are the ones your driver is most likely to touch, and being comfortable naming them will make every later code listing easier to read.

28.5.7 The Accessor API in More Detail

The `if_t` opaque handle has been growing a family of accessor functions since FreeBSD 12. The pattern is consistent: where you would have written `ifp->if_flags |= IFF_UP`, you now write `if_setflagbits(ifp, IFF_UP, 0)`. Where you would have written `ifp->if_softc = sc`, you now write `if_setsoftc(ifp, sc)`. The motivation is to let the kernel evolve the internal layout of `struct ifnet` without breaking drivers.

The accessor functions include:

- `if_setsoftc(ifp, sc)` and `if_getsoftc(ifp)` for the `softc` pointer.
- `if_setflagbits(ifp, set, clear)` and `if_getflags(ifp)` for `if_flags`.
- `if_setdrvflagbits(ifp, set, clear)` and `if_getdrvflags(ifp)` for `if_drv_flags`.
- `if_setmtu(ifp, mtu)` and `if_getmtu(ifp)` for the MTU.
- `if_setbaudrate(ifp, rate)` and `if_getbaudrate(ifp)` for the advertised line rate.
- `if_sethwassist(ifp, assist)` and `if_gethwassist(ifp)` for checksum-offload hints.
- `if_settransmitfn(ifp, fn)` for `if_transmit`.
- `if_setioctlnfn(ifp, fn)` for `if_ioctl`.
- `if_setinitfn(ifp, fn)` for `if_init`.
- `if_setqflushfn(ifp, fn)` for `if_qflush`.
- `if_setinputfn(ifp, fn)` for `if_input`.
- `if_inc_counter(ifp, ctr, n)` for the statistics counters.

Some of these are inlines that still reach through to a direct field access behind the scenes; others are wrappers

that may in the future refer to a subtly different field layout. Using the accessors now costs nothing and protects your driver against future churn.

For mynet we mostly use the direct-field style, because that is what the existing reference drivers like `if_disc.c` and `if_epair.c` still use, and consistency with the rest of the tree is valuable for readers. When you graduate to writing your own new driver, feel free to prefer the accessors. Both styles are correct.

28.5.8 A First Glimpse of the Code

Before we move on, let us look at a tiny code fragment that summarises the shape of a driver's relationship with `ifnet`. This is the pattern you will type more fully in Section 3, but it is already useful to see the skeleton:

```
struct mynet_softc {
    struct ifnet    *ifp;
    struct mtx      mtx;
    uint8_t         hwaddr[ETHER_ADDR_LEN];
    /* ... fields for simulation state ... */
};

static int
mynet_transmit(struct ifnet *ifp, struct mbuf *m)
{
    /* pass packet to the transport, or drop it */
}

static int
mynet_ioctl(struct ifnet *ifp, u_long cmd, caddr_t data)
{
    /* handle SIOCSIFFLAGS, SIOCSIFMTU, ... */
}

static void
mynet_init(void *arg)
{
    /* make the interface ready to move traffic */
}

static void
mynet_create(void)
{
    struct mynet_softc *sc = malloc(sizeof(*sc), M_MYNET, M_WAITOK | M_ZERO);
    struct ifnet *ifp = if_alloc(IFT_ETHER);

    sc->ifp = ifp;
    mtx_init(&sc->mtx, "mynet", NULL, MTX_DEF);
}
```



```
ifp->if_softc = sc;
if_initname(ifp, "mynet", 0);
ifp->if_flags = IFF_BROADCAST | IFF_SIMPLEX | IFF_MULTICAST;
ifp->if_init = mynet_init;
ifp->if_ioctl = mynet_ioctl;
ifp->if_transmit = mynet_transmit;
ifp->if_qflush = mynet_qflush;

/* fabricate a MAC address ... */
ether_ifattach(ifp, sc->hwaddr);
}
```

Do not type this yet. It is only a sketch, and several pieces are missing. We will fill them in during Section 3. What matters now is the shape: allocate, configure, attach. Every driver in the tree does this, with variations for the bus it lives on and the transport it talks to.

28.5.9 Wrapping Up Section 2

The `ifnet` object is the kernel's representation of a network interface. It has identity fields, capability fields, flags, callbacks, counters, and media state. It is created with `if_alloc`, configured by the driver, and installed in the system with `if_attach` or `ether_ifattach`. A pseudo-interface driver creates `ifnets` on demand through an interface cloner. A real NIC driver creates its `ifnet` during probe and attach.

You now have the vocabulary. In Section 3 we will put it to use by creating and registering a real working network interface. Before we do, though, it is worth spending a little time reading a real driver that uses the same patterns we are about to write. The next subsection guides you through `if_disc.c`, the canonical “simplest pseudo-Ethernet” driver in the FreeBSD source tree.

28.5.10 A Guided Tour of `if_disc.c`

Open `/usr/src/sys/net/if_disc.c` in your editor. It is about two hundred lines of code, and every line is instructive. The `disc(4)` driver creates an interface whose only job is to silently drop every packet it receives for transmission. It is the moral equivalent of `/dev/null` for packets. Because it is so small, it shows the shape of a pseudo-driver without any distraction.

The file begins with the standard license header, then a cluster of `#include` directives that should now look familiar. `net/if.h` and `net/if_var.h` for the interface structure, `net/ethernet.h` for Ethernet-specific helpers, `net/if_clone.h` for the cloner API, `net/bpf.h` for packet taps, and `net/vnet.h` for VNET awareness. That is almost exactly the include set we will use in `mynet.c`.

Next come a handful of module-level declarations. The string `discname = "disc"` is the family name the cloner will expose. `M_DISC` is the memory-type tag for `vmstat -m` accounting. `VNET_DEFINE_STATIC(struct if_clone *, disc_cloner)` declares a per-VNET cloner variable, and the `V_disc_cloner` macro provides the access shim. These are all pieces you will recognise when we write the same three lines in our own driver a few pages from now.

The `softc` declaration is particularly short. `struct disc_softc` holds only an `ifnet` pointer. That is all the state a discard driver needs: one interface per `softc`, no counters, no queues, no timers. Our `mynet` `softc` will be longer because we have a simulated receive path, a media descriptor, and a mutex, but the pattern of “one `softc` per interface” is the same.

Move down the file to `disc_clone_create`. It begins by allocating the `softc` with `M_WAITOK | M_ZERO`, because the cloner is called from user context and can afford to sleep. Then it allocates the `ifnet` with `if_alloc(IFT_LOOP)`. Note that `disc` uses `IFT_LOOP` rather than `IFT_ETHER`, because its link-layer semantics are more loopback-like than Ethernet-like. The choice of `IFT_*` constant matters because the stack queries `if_type` to decide which link-layer helper to invoke. Our driver will use `IFT_ETHER` because we want to use `ether_ifattach`.

Then `disc_clone_create` calls `if_initname(ifp, discname, unit)`, sets the `softc` pointer, sets the `if_mtu` to `DSMTU` (a locally-defined value), and sets `if_flags` to `IFF_LOOPBACK | IFF_MULTICAST`. The callbacks `if_ioctl`, `if_output`, and `if_init` are set. Notice that `disc` sets `if_output` rather than `if_transmit`, because loopback-style drivers are still wired to the classical output path. Our Ethernet driver will use `if_transmit` through `ether_ifattach`.

Then comes `if_attach(ifp)`, which registers the interface with the stack without Ethernet-specific setup. `bpfattach(ifp, DLT_NULL, sizeof(u_int32_t))` follows, registering with BPF using the null link-type (which tells `tcpdump` to expect a four-byte header carrying the address family of the payload). Our driver will use `DLT_EN10MB`, automatically, via `ether_ifattach`.

The destroy path, `disc_clone_destroy`, is symmetric: it calls `bpfdetach`, `if_detach`, `if_free`, and finally `free(sc, M_DISC)`. Our driver will be slightly more involved because we have callouts and a media descriptor to tear down, but the skeleton is identical.

The transmit path, `discoutput`, is three lines of code. It inspects the packet family, fills in the four-byte BPF header, taps BPF, updates counters, and frees the mbuf. That is all a “discard everything” driver needs to do. Our `mynet_transmit` will be longer, but structurally it does exactly the same things with slightly more discipline: validate, tap, count, release.

The `ioctl` handler, `discioctl`, handles `SIOCSIFADDR`, `SIOCSIFFLAGS`, and `SIOCSIFMTU`, and returns `EINVAL` for everything else. For a minimal pseudo-driver this is plenty. Our driver will be more elaborate because we add media descriptors and delegate unknown `ioctls` to `ether_ioctl`, but the switch-statement shape is the same.

Finally, the cloner registration is done in `vnet_disc_init` through `if_clone_simple(discname, disc_clone_create, disc_clone_destroy, 0)`, wrapped in `VNET_SYSINIT` and matched by a `VNET_SYSUNINIT` that calls `if_clone_detach`. Again, this is exactly the pattern we will use.

The take-away from reading `disc` is that a working pseudo-driver in the FreeBSD tree is about two hundred lines of code. Most of those lines are boilerplate that you set once and forget. The interesting parts are the `softc`, the cloner, and the handful of callbacks. Everything else is rhythm.

Do not feel obligated to memorise `disc`. Just read it once, slowly, now. When we start writing `mynet`, return to this section and you will see that most of what we type is the same pattern with a few additions for Ethernet-like behaviour, packet reception, and media descriptors. The pattern is worth seeing once in its purest form before we elaborate it.

28.6 Section 3: Creating and Registering a Network Interface

Time to write code. In this section we will build the skeleton of `mynet`, a pseudo-Ethernet driver. It will appear as a normal Ethernet interface to the rest of the system. Userland will be able to create an instance with `ifconfig mynet create`, assign an IPv4 address, bring it up, take it down, and destroy it, just like `epair` and `disc`. We will not yet handle real packet movement. Section 4 and Section 5 will handle the transmit and receive paths. Here we focus on the creation, registration, and basic metadata.

28.6.1 Project Layout

All companion files for this chapter live under `examples/part-06/ch28-network-driver/`. The skeleton in this section is in `examples/part-06/ch28-network-driver/lab01-skeleton/`. Create the directory if you are typing along manually, or look at the files if you prefer to read first and experiment later. The top-level layout we will use for the chapter is:

```
examples/part-06/ch28-network-driver/  
  Makefile  
  mynet.c  
  README.md  
  shared/  
  lab01-skeleton/  
  lab02-transmit/  
  lab03-receive/  
  lab04-media/  
  lab05-bpf/  
  lab06-detach/  
  lab07-reading-tree/  
  challenge01-shared-queue/  
  challenge02-link-flap/  
  challenge03-error-injection/  
  challenge04-iperf3/  
  challenge05-sysctl/  
  challenge06-netgraph/
```

The top-level `mynet.c` is the reference driver for the whole chapter and evolves from the skeleton in Section 3 through the final cleanup code in Section 8. The `lab0x` directories contain README files that walk you through the corresponding lab step. The challenges each add a small feature on top of the finished driver, and `shared/` holds helper scripts and notes referenced by more than one lab.

28.6.2 The Makefile

Let us start with the build file. A kernel module for a pseudo-Ethernet driver is one of the simplest Makefiles in the entire tree. Ours will look like this:

```
# Makefile for mynet - Chapter 28 (Writing a Network Driver).
#
# Builds the chapter's reference pseudo-Ethernet driver,
# mynet.ko, which demonstrates ifnet registration through an
# interface cloner, minimal transmit and receive paths, and
# safe load and unload lifecycle.

KMOD=    mynet
SRCS=    mynet.c opt_inet.h opt_inet6.h

SYSDIR?=    /usr/src/sys

.include <bsd.kmod.mk>
```

This is very close to the Makefile used by `/usr/src/sys/modules/if_disc/Makefile`, which is exactly what you want for a clone-based pseudo-interface driver. Two small differences: we do not set `.PATH`, because our source file is in the current directory rather than in `/usr/src/sys/net/`, and we set `SYSDIR` explicitly so that the build works on machines that may not ship a system configuration for it. Otherwise, it is the standard `bsd.kmod.mk` pattern you have seen since Chapter 10.

28.6.3 Preliminary Includes and Module Glue

Open your editor and start `mynet.c` with the following preamble. Each include has a specific role, so we will annotate them as we go:

```
#include <sys/param.h>
#include <sys/system.h>
#include <sys/kernel.h>
#include <sys/module.h>
#include <sys/malloc.h>
#include <sys/lock.h>
#include <sys/mutex.h>
#include <sys/mbuf.h>
#include <sys/socket.h>
#include <sys/sockio.h>
#include <sys/callout.h>

#include <net/if.h>
#include <net/if_var.h>
#include <net/if_arp.h>
#include <net/ethernet.h>
#include <net/if_types.h>
#include <net/if_clone.h>
#include <net/if_media.h>
```

```
#include <net/bpf.h>
#include <net/vnet.h>
```

The first block pulls in the core kernel headers you already know from earlier chapters: parameters, system calls, module machinery, memory allocator, locking, mbufs, socket IO constants, and the callout subsystem. The second block pulls in the networking-specific headers: `if.h` for the `ifnet` structure and flags, `if_var.h` for the inline helpers, `if_arp.h` for address resolution constants, `ethernet.h` for Ethernet framing, `if_types.h` for interface-type constants like `IFT_ETHER`, `if_clone.h` for the cloner API, `if_media.h` for media descriptors, `bpf.h` for tcpdump support, and `vnet.h` for VNET-awareness, which we use in the same way that `/usr/src/sys/net/if_disc.c` uses it.

Next, a module-wide memory type and the interface family name:

```
static const char mynet_name[] = "mynet";
static MALLOC_DEFINE(M_MYNET, "mynet", "mynet pseudo Ethernet driver");

VNET_DEFINE_STATIC(struct if_clone *, mynet_cloner);
#define V_mynet_cloner VNET(mynet_cloner)
```

`mynet_name` is the string we will pass to `if_initname` so that interfaces are named `mynet0`, `mynet1`, and so on. `M_MYNET` is the memory type tag so that `vmstat -m` shows you how much memory the driver is using. `VNET_DEFINE_STATIC` is VNET-aware: it gives each virtual network stack its own cloner variable. This mirrors the `VNET_DEFINE_STATIC(disc_cloner)` declaration in `/usr/src/sys/net/if_disc.c`. We will return to VNET briefly in Section 8.

Function, macro, and structure names are the durable reference into the FreeBSD tree. Line numbers drift release to release. For FreeBSD 14.3 orientation only: in `/usr/src/sys/net/if_disc.c`, the `VNET_DEFINE_STATIC(disc_cloner)` declaration sits near line 79 and the `if_clone_simple` call inside `vnet_disc_init` near line 134; in `/usr/src/sys/net/if_epair.c`, `epair_transmit` begins near line 324 and `epair_ioctl` near line 429; in `/usr/src/sys/sys/mbuf.h`, the `MGETHDR` compatibility macro sits near line 1125. Open the file and jump to the symbol.

28.6.4 The Softc

A `softc`, as you know from earlier chapters, is the private per-instance structure your driver allocates to track the state of one device. For a network driver, the `softc` is per-interface. Here is what ours looks like at this stage:

```
struct mynet_softc {
    struct ifnet    *ifp;
    struct mtx      mtx;
    uint8_t         hwaddr[ETHER_ADDR_LEN];
    struct ifmedia   media;
    struct callout  rx_callout;
    int             rx_interval_hz;
```

```

    bool                running;
};

#define MYNET_LOCK(sc)      mtx_lock(&(sc)->mtx)
#define MYNET_UNLOCK(sc)   mtx_unlock(&(sc)->mtx)
#define MYNET_ASSERT(sc)   mtx_assert(&(sc)->mtx, MA_OWNED)

```

The fields are straightforward. `ifp` is the interface object we create. `mtx` is a mutex to protect the softc during concurrent transmit, ioctl, and teardown. `hwaddr` is the six-byte Ethernet address we fabricate. `media` is the media descriptor we expose through `SIOCGIFMEDIA`. `rx_callout` and `rx_interval_hz` are used by the simulated receive path we build in Section 5. `running` reflects the driver's sense of whether the interface is currently active.

The macros at the bottom give us short, readable locking primitives. They are a stylistic convention used in many FreeBSD drivers, including `/usr/src/sys/dev/e1000/if_em.c` and `/usr/src/sys/net/if_epair.c`.

28.6.5 The Skeleton of `mynet_create`

Now the main action of this section. We will write a function that is called from the cloner to create and register a new interface. This function is the heart of the initialisation code. Let us build it step by step, then assemble the pieces.

First, allocate the softc and the ifnet:

```

struct mynet_softc *sc;
struct ifnet *ifp;

sc = malloc(sizeof(*sc), M_MYNET, M_WAITOK | M_ZERO);
ifp = if_alloc(IFT_ETHER);
if (ifp == NULL) {
    free(sc, M_MYNET);
    return (ENOSPC);
}
sc->ifp = ifp;
mtx_init(&sc->mtx, "mynet", NULL, MTX_DEF);

```

We use `M_WAITOK | M_ZERO` because this is called from a user-context path (the cloner) and we want zero-initialised memory. `IFT_ETHER` is from `/usr/src/sys/net/if_types.h`: it declares our interface as an Ethernet interface for the kernel's bookkeeping purposes, which is important because the stack uses `if_type` to decide what link-layer semantics to apply.

Next, fabricate a MAC address. In real NIC drivers, the hardware has an EEPROM with a unique factory-assigned MAC. We do not have that luxury, so we invent one. A locally-administered unicast address starts with a byte whose second-lowest bit is set and whose lowest bit is clear. The classic way is `02:xx:xx:xx:xx:xx`. We will do something similar to what `epair(4)` does in its `epair_generate_mac` function:

```
arc4rand(sc->hwaddr, ETHER_ADDR_LEN, 0);
sc->hwaddr[0] = 0x02; /* locally administered, unicast */
```

`arc4rand` is a kernel-internal entropy-backed random function, defined in `/usr/src/sys/libkern/arc4random.c`. It is fine for MAC-address fabrication. We then force the first byte to `0x02` so that the address is both locally-administered and unicast, which is what the IEEE reserves for addresses that are not factory-assigned.

Next, configure the interface fields:

```
if_initname(ifp, mynet_name, unit);
ifp->if_softc = sc;
ifp->if_flags = IFF_BROADCAST | IFF_SIMPLEX | IFF_MULTICAST;
ifp->if_capabilities = IFCAP_VLAN_MTU;
ifp->if_capenable = IFCAP_VLAN_MTU;
ifp->if_transmit = mynet_transmit;
ifp->if_qflush = mynet_qflush;
ifp->if_ioctl = mynet_ioctl;
ifp->if_init = mynet_init;
ifp->if_baudrate = IF_Gbps(1);
```

`if_initname` sets both `if_xname`, the unique name of the interface, and the driver's family name and unit number. `if_softc` ties the interface back to our private structure so that callbacks can find it. The flags mark the interface as broadcast-capable, simplex (meaning it cannot hear its own transmissions, which is true of an Ethernet NIC), and multicast-capable. `IFCAP_VLAN_MTU` says we can forward VLAN-tagged frames whose total payload exceeds the baseline Ethernet MTU by four bytes. The callbacks are the functions we will implement shortly. `if_baudrate` is informational; `IF_Gbps(1)` reports one gigabit per second, roughly matching what an average simulated link might claim.

Next, set up the media descriptor. This is what `SIOCGIFMEDIA` will return, and it is what `ifconfig mynet0` uses to print the media line:

```
ifmedia_init(&sc->media, 0, mynet_media_change, mynet_media_status);
ifmedia_add(&sc->media, IFM_ETHER | IFM_1000_T | IFM_FDX, 0, NULL);
ifmedia_add(&sc->media, IFM_ETHER | IFM_AUTO, 0, NULL);
ifmedia_set(&sc->media, IFM_ETHER | IFM_AUTO);
```

`ifmedia_init` registers two callbacks: one the stack calls when the user changes the media, and one it calls to learn the current media state. `ifmedia_add` declares a specific media type the interface supports. `IFM_ETHER | IFM_1000_T | IFM_FDX` means “Ethernet, 1000BaseT, full duplex”; `IFM_ETHER | IFM_AUTO` means “Ethernet, auto-negotiate”. `ifmedia_set` chooses the default. `ifconfig mynet0` will reflect this choice.

Next, initialise the simulated receive callout. We will implement it in Section 5, but we prepare the field now so that `mynet_create` leaves the `softc` fully usable:

```
callout_init_mtx(&sc->rx_callout, &sc->mtx, 0);
sc->rx_interval_hz = hz; /* one simulated packet per second */
```

`callout_init_mtx` registers our callout with the `softc`'s mutex so that the callout system acquires and releases the lock for us when it invokes the handler. This is a widely-used pattern in the kernel and it avoids a whole class of lock-ordering bugs.

Finally, attach the interface to the Ethernet layer:

```
ether_ifattach(ifp, sc->hwaddr);
```

This single call does a lot of work. It sets `if_addr_len` and `if_hdr_len` and `if_mtu` to Ethernet defaults, it calls `if_attach` to register the interface, it installs `ether_input` and `ether_output` as the link-layer input and output handlers, and it calls `bpfattach(ifp, DLT_EN10MB, ETHER_HDR_LEN)` so that `tcpdump -i mynet0` works immediately. After this call, the interface is live: userland can see it, assign addresses to it, and begin issuing `ioctl`s on it.

28.6.6 The Skeleton of `mynet_destroy`

Destruction mirrors creation but in reverse order. Here is the skeleton:

```
static void
mynet_destroy(struct mynet_softc *sc)
{
    struct ifnet *ifp = sc->ifp;

    MYNET_LOCK(sc);
    sc->running = false;
    MYNET_UNLOCK(sc);

    callout_drain(&sc->rx_callout);

    ether_ifdetach(ifp);
    if_free(ifp);

    ifmedia_removeall(&sc->media);
    mtx_destroy(&sc->mtx);
    free(sc, M_MYNET);
}
```

We mark the `softc` as no longer running, drain the callout so no scheduled receive event can fire, call `ether_ifdetach` to unregister the interface, free the `ifnet`, remove any allocated media entries, destroy the mutex, and free the `softc`. The order matters: you must not free the `ifnet` while the callout might still run against it, and you must not destroy the mutex while the callout might still acquire it. `callout_drain` is what gives us the synchronous guarantee that no more callbacks will fire after it returns.

28.6.7 Registering the Cloner

Two pieces tie `mynet_create` and `mynet_destroy` to the kernel: the cloner registration and the module handler. Here is the cloner code:

```
static int
mynet_clone_create(struct if_clone *ifc, int unit, caddr_t params)
{
    return (mynet_create_unit(unit));
}

static void
mynet_clone_destroy(struct ifnet *ifp)
{
    mynet_destroy((struct mynet_softc *)ifp->if_softc);
}

static void
vnet_mynet_init(const void *unused __unused)
{
    V_mynet_cloner = if_clone_simple(mynet_name, mynet_clone_create,
                                     mynet_clone_destroy, 0);
}
VNET_SYSINIT(vnet_mynet_init, SI_SUB_PSEUDO, SI_ORDER_ANY,
             vnet_mynet_init, NULL);

static void
vnet_mynet_uninit(const void *unused __unused)
{
    if_clone_detach(V_mynet_cloner);
}
VNET_SYSUNINIT(vnet_mynet_uninit, SI_SUB_INIT_IF, SI_ORDER_ANY,
              vnet_mynet_uninit, NULL);
```

`if_clone_simple` registers a simple cloner, meaning a cloner whose name matching is by exact prefix (`mynet` followed by an optional unit number). `/usr/src/sys/net/if_disc.c` uses this same call inside `vnet_disc_init`, the VNET init routine for the disc driver. The create function receives a unit number and is responsible for producing a new interface. The destroy function receives an `ifnet` and is responsible for taking it away. The `SYSINIT` and `SYSUNINIT` macros ensure that the cloner is registered when the module loads and unregistered when it unloads.

The `mynet_create_unit` helper glues the two halves together. It takes a unit number, does the allocation we described above, calls `ether_ifattach`, and returns zero on success or an error on failure. The full listing is in the companion file under `lab01-skeleton/`.

28.6.8 The Module Handler

Finally, the standard module boilerplate:

```
static int
mynet_modevent(module_t mod, int type, void *data __unused)
{
    switch (type) {
        case MOD_LOAD:
        case MOD_UNLOAD:
            return (0);
        default:
            return (EOPNOTSUPP);
    }
}

static moduledata_t mynet_mod = {
    "mynet",
    mynet_modevent,
    NULL
};

DECLARE_MODULE(mynet, mynet_mod, SI_SUB_PSEUDO, SI_ORDER_ANY);
MODULE_DEPEND(mynet, ether, 1, 1, 1);
MODULE_VERSION(mynet, 1);
```

The module handler itself does nothing interesting. The real initialisation happens in `vnet_mynet_init`, which `VNET_SYSINIT` arranges to be called at `SI_SUB_PSEUDO`. This split is not strictly necessary for a non-VNET driver, but following the pattern from `disc(4)` and `epair(4)` keeps our driver ready for VNET use and matches the convention used by the rest of the tree.

`MODULE_DEPEND(mynet, ether, 1, 1, 1)` declares a dependency on the `ether` module so that Ethernet support is loaded before we try to use `ether_ifattach`. `MODULE_VERSION(mynet, 1)` declares our own version number so that other modules can depend on us if they ever want to.

28.6.9 A Closer Look at Interface Cloners

Interface cloners are worth a short detour because they drive much of the lifecycle of a pseudo-driver, and because the API is slightly richer than the `if_clone_simple` call we have used so far.

A cloner is a named factory registered with the network stack. It carries a name prefix, a create callback, a destroy callback, and optionally a match callback. When userland runs `ifconfig mynet create`, the stack walks its list of cloners looking for one whose prefix matches the string `mynet`. If it finds one, it picks a unit number, calls the create callback, and returns the resulting interface's name.

The API has two flavours. `if_clone_simple` registers a cloner with the default matching rule: the name must begin with the cloner's prefix and may be followed by a unit number. `if_clone_advanced`

registers a cloner with a caller-provided match function, which allows more flexible naming. `epair(4)` uses `if_clone_advanced` because its interfaces come in pairs named `epairXa` and `epairXb`. We use `if_clone_simple` because `mynet0`, `mynet1`, and so on are good enough.

Within the create callback you have two pieces of information: the cloner itself (through which you can look up sibling interfaces) and the requested unit number (which may be `IF_MAXUNIT` if the user did not specify one, in which case you pick a free unit). In our driver we accept whatever unit the cloner tells us and pass it straight to `if_initname`.

The destroy callback is simpler: it receives the `ifnet` pointer of the interface to destroy and must tear everything down. The cloner framework handles the list of interfaces for us; we do not need to maintain one ourselves.

When the module unloads, `if_clone_detach` walks the list of interfaces the cloner created and calls the destroy callback for each. After that, the cloner itself is unregistered. This two-step teardown is what makes `kldunload` clean: even if the user forgot to `ifconfig mynet0 destroy` before unloading, the cloner takes care of it.

If your driver ever needs to expose additional arguments to the create path (for example, the partner interface name in an `epair`-style driver), the cloner framework supports a `caddr_t params` argument to the create callback, which carries bytes the user supplied through `ifconfig mynet create foo bar`. We do not use that mechanism here, but it is present and worth knowing about.

28.6.10 What Happens Inside `ether_ifattach`

We called `ether_ifattach(ifp, sc->hwaddr)` at the end of `mynet_create_unit` and said only that it “does a lot of work”. Let us open `/usr/src/sys/net/if_ETHERSUBR.c` and look at what that work really is, because understanding it makes the rest of our driver’s behaviour predictable rather than mysterious.

`ether_ifattach` begins by setting `ifp->if_addrlen = ETHER_ADDR_LEN` and `ifp->if_hdrlen = ETHER_HDR_LEN`. These fields tell the stack how many bytes of link-layer addressing and header prepend a frame. For Ethernet both values are constants: six bytes of MAC and fourteen bytes of header.

Next it sets `ifp->if_mtu = ETHERMTU` (1500 bytes, the IEEE Ethernet default) if the driver has not already set a larger value. Our driver left `if_mtu` at zero after `if_alloc`, so `ether_ifattach` gives us the default. We could override it afterwards; a jumbo-capable driver might set `if_mtu` to 9000 before `ether_ifattach`.

Then it sets the link-layer output function, `if_output`, to `ether_output`. `ether_output` is the generic L3-to-L2 handler: it receives a packet with an IP header and a destination address, resolves ARP or neighbour discovery if needed, constructs the Ethernet header, and calls `if_transmit`. This chain of indirection is what allows an IP packet from a socket to travel transparently through the stack and reach our driver.

It sets `if_input` to `ether_input`. `ether_input` is the inverse: it receives a complete Ethernet frame, strips the Ethernet header, dispatches on the `EtherType`, and hands the payload up to the appropriate protocol (IPv4, IPv6, ARP, LLC, and so on). When our driver calls `if_input(ifp, m)`, it is effectively calling `ether_input(ifp, m)`.

Then it stores the MAC address in the interface’s address list, making it visible to userland through `getifaddrs(3)` and through `ifconfig`. This is how `ifconfig mynet0` prints an ether line.

Then it calls `if_attach(ifp)`, which registers the interface with the global list, allocates any stack-side state needed, and makes the interface visible to userland.

Finally it calls `bpfattach(ifp, DLT_EN10MB, ETHER_HDR_LEN)`, which registers the interface with BPF using the Ethernet link type. From that moment `tcpdump -i mynet0` will find the interface and will expect frames with 14-byte Ethernet headers.

It is a lot of work for one function call. Doing it all by hand is legal (and many older drivers do) but error-prone. `ether_ifattach` is one of those helpers whose existence makes writing a driver genuinely easier, and reading its body is rewarding because it demystifies what happens between “I allocated an ifnet” and “the stack is fully aware of my interface”.

The complementary function `ether_ifdetach` performs the inverse operations in the correct reverse order. It is the right function to call during teardown, and it is what we call in `mynet_destroy`.

28.6.11 Build, Load, and Verify

At this point, even without transmit and receive logic, the skeleton should build and load. Here is what the verification flow looks like:

```
# cd examples/part-06/ch28-network-driver
# make
# kldload ./mynet.ko

# ifconfig mynet create
mynet0
# ifconfig mynet0
mynet0: flags=8802<BROADCAST,SIMPLEX,MULTICAST> metric 0 mtu 1500
    ether 02:a3:f1:22:bc:0d
    media: Ethernet autoselect
    status: no carrier
    groups: mynet
```

The exact MAC address will differ because `arc4rand` gives you a different random address each time. The rest of the output should closely match. If it does, you have succeeded: you have a live, registered, named, MAC-addressed network interface, visible to all the standard tools, without yet handling any real packets. That is already a significant accomplishment.

Destroy the interface and unload the module to close the lifecycle:

```
# ifconfig mynet0 destroy
# kldunload mynet
```

`kldstat` should show that the module is gone. `ifconfig -a` should no longer list `mynet0`. If anything is left behind, we will cover how to diagnose it in Section 8.

28.6.12 What the Stack Now Knows About Us

After `ether_ifattach` returns, the stack knows several important facts about our interface:

- It is of type `IFT_ETHER`.
- It supports broadcast, simplex, and multicast.
- It has a specific MAC address.
- It has a default MTU of 1500 bytes.
- It has a transmit callback, an `ioctl` callback, an `init` callback, and a media handler.
- It is attached to BPF with `DLT_EN10MB` encapsulation.
- Its state-of-the-link is currently undefined (we have not called `if_link_state_change` yet).

Everything else, packet movement, counter updates, link state, will come alive in the following sections. The skeleton is intentionally small. It is the first time you can point at something on your system and say, honestly, “that is my network interface.” Pause on that sentence. It marks a real milestone in the book.

28.6.13 Common Mistakes

Two mistakes are easy to make in this section, and both of them produce confusing symptoms.

The first is forgetting to call `ether_ifattach` and calling `if_attach` directly. That is perfectly legal, and it results in a non-Ethernet pseudo-interface, but your driver then has to install its own `if_input` and `if_output` handlers, and `tcpdump` does not work until you `bpfattach` yourself. If you see an interface that looks like it should work but `tcpdump -i mynet0` complains about the link type, check whether you used `ether_ifattach`.

The second mistake is to allocate the softc with `M_NOWAIT` instead of `M_WAITOK`. `M_NOWAIT` is correct in interrupt context, but `mynet_clone_create` runs in a regular user context through the `ifconfig create` path, and `M_WAITOK` is the right choice. Using `M_NOWAIT` here introduces a rare allocation-failure path for no benefit.

28.6.14 Wrapping Up Section 3

You now have a working skeleton. The interface exists, is registered, has an Ethernet address, and can be created and destroyed on demand. The stack is ready to call into our driver through `if_transmit`, `if_ioctl`, and `if_init`, but we have not yet implemented the bodies of those callbacks. Section 4 tackles the transmit path. That is the one you will feel most viscerally, because once it works, `ping` starts to push real bytes through your code.

28.7 Section 4: Handling Packet Transmission

Transmission is the outbound half of packet flow. When the kernel’s network stack decides that a packet needs to leave through `mynet0`, it packages the packet in an mbuf chain and invokes our `if_transmit` callback. Our job is to accept the mbuf, do whatever is appropriate with it, and free it. In this section we will build a complete transmit path that validates the mbuf, updates counters, taps BPF so that `tcpdump` sees the packet, and disposes of the frame. Because `mynet` is a pseudo-device without a real wire, we will initially drop the packet after counting it. That is similar to what `disc(4)` does in `/usr/src/sys/net/if_disc.c`, and it is enough to demonstrate the full transmit flow end-to-end.

28.7.1 How the Stack Reaches Us

Before we open our editor, let us trace how a packet gets from a process to our driver. When a process calls `send(2)` on a TCP socket that is bound to an IP address assigned to `mynet0`, the following sequence happens, in broad strokes. Do not worry about memorising every step; the point is to see where our code sits in the bigger picture.

1. The socket layer copies the user payload into mbufs and passes it to TCP.
2. TCP segments the payload, adds TCP headers, and passes the segments to IP.
3. IP adds IP headers, looks up the route, and passes the result to the Ethernet layer through `ether_output`.
4. `ether_output` resolves the next-hop MAC address (through ARP if required), prepends an Ethernet header, and calls `if_transmit` on the output interface.
5. Our `if_transmit` function is invoked with `ifp` pointing at `mynet0` and the mbuf pointing at the complete Ethernet frame ready to be transmitted.

From that moment the frame is ours. We must either send it out, drop it cleanly, or queue it for later delivery. Whichever we choose, we must free the mbuf exactly once. Double-free leads to kernel corruption, use-after-free leads to mystery panics, and forgetting to free leaks mbufs until the machine runs out.

28.7.2 The Transmit Callback Signature

The prototype of an `if_transmit` callback is:

```
int mynet_transmit(struct ifnet *ifp, struct mbuf *m);
```

It is declared in `/usr/src/sys/net/if_var.h` as the typedef `if_transmit_fn_t`. The return value is an `errno`: zero on success, or an error such as `ENOBUFS` if the packet could not be queued. Real NIC drivers rarely return non-zero, because they prefer to silently drop and increment `IFCOUNTER_OERRORS`. Pseudo-drivers that mimic real behaviour usually do the same.

Here is the full callback we will implement:

```
static int
mynet_transmit(struct ifnet *ifp, struct mbuf *m)
{
    struct mynet_softc *sc = ifp->if_softc;
    int len;

    if (m == NULL)
        return (0);
    M_ASSERTPKTHDR(m);

    /* Reject oversize frames. Leave a little slack for VLAN. */
    if (m->m_pkthdr.len > (ifp->if_mtu + sizeof(struct ether_vlan_header))) {
        m_freem(m);
        if_inc_counter(ifp, IFCOUNTER_OERRORS, 1);
    }
}
```

```

        return (E2BIG);
    }

    /* If the interface is administratively down, drop. */
    if ((ifp->if_flags & IFF_UP) == 0 ||
        (ifp->if_drv_flags & IFF_DRV_RUNNING) == 0) {
        m_freem(m);
        if_inc_counter(ifp, IFCOUNTER_OERRORS, 1);
        return (ENETDOWN);
    }

    /* Let tcpdump see the outgoing packet. */
    BPF_MTAP(ifp, m);

    /* Count it. */
    len = m->m_pkthdr.len;
    if_inc_counter(ifp, IFCOUNTER_OPACKETS, 1);
    if_inc_counter(ifp, IFCOUNTER_OBYTES, len);
    if (m->m_flags & (M_BCAST | M_MCAST))
        if_inc_counter(ifp, IFCOUNTER_OMCASTS, 1);

    /* In a real NIC we would DMA this to hardware. Here we just drop. */
    m_freem(m);
    return (0);
}

```

Let us walk through it. This is where the shape of a transmit routine becomes clear, so it is worth reading the code slowly.

28.7.3 The NULL Check

The first two lines handle the defensive case where the stack calls us with a NULL pointer. This is not supposed to happen in normal operation, but the kernel is a place where defensive programming earns its keep. Returning 0 on a NULL input is the standard idiom; `if_epair.c` does the same at the top of `epair_transmit`.

28.7.4 M_ASSERTPKTHDR

The next line is a macro from `/usr/src/sys/sys/mbuf.h` that asserts the mbuf has `M_PKTHDR` set. Every mbuf that reaches a driver's transmit callback must be the head of a packet, and must therefore carry a valid packet header. Asserting this catches bugs caused by mbuf surgery elsewhere in the system. In production kernels the assertion is compiled out, but having it in the source tree documents the contract, and in `INVARIANTS` kernels it catches bad usage during development.

28.7.5 MTU Validation

The block under the comment `/* Reject oversize frames. */` rejects packets that are larger than the interface MTU plus a small slack for a VLAN header. `epair_transmit` in `/usr/src/sys/net/if_epair.c` does exactly the same check; look for the `if (m->m_pkthdr.len > (ifp->if_mtu + sizeof(struct ether_vlan_header)))` guard that `m_freem`s the frame and bumps `IFCOUNTER_OERRORS`. We leave slack for `ether_vlan_header` because VLAN-tagged frames carry four extra bytes beyond the base Ethernet header, and we advertised `IFCAP_VLAN_MTU` in Section 3, so we should honour that capability.

On rejection, we free the mbuf with `m_freem(m)` and increment `IFCOUNTER_OERRORS`. We also return `E2BIG` as a hint to the caller, though in practice the stack rarely looks at the return value other than to decide whether to drop locally.

28.7.6 State Validation

The `if` block under the comment `/* If the interface is administratively down, drop. */` checks two conditions. `IFF_UP` is set by `ifconfig mynet0 up` and cleared by `ifconfig mynet0 down`, and it is the userland's way of saying that the interface should or should not be carrying traffic. `IFF_DRV_RUNNING` is the driver's internal "I have allocated my resources and I am ready to move traffic" flag. If either is clear, we have no business sending the packet, so we drop it and increment the error counter.

This check is not strictly necessary for correctness in all cases, because the stack usually avoids routing traffic through a down interface. But defensive drivers check anyway, because races between the stack's view of state and the driver's view of state do happen, especially during interface teardown.

28.7.7 BPF Tap

`BPF_MTAP(ifp, m)` is a macro that conditionally calls into BPF if any packet capture session is active on the interface. It expands to `bpf_mtap_if((_ifp), (_m))` in the current tree. The macro is defined in `/usr/src/sys/net/bpf.h`. When `tcpdump -i mynet0` is running, BPF has attached to the interface's `if_bpf` pointer, and the macro hands it a copy of the outgoing packet. When no one is listening, the macro quickly returns and has negligible cost.

Placement matters. We tap before we drop, because we want `tcpdump` to see the packet even if we are simulating a down interface. Real NIC drivers tap slightly earlier, right before they hand the frame to hardware DMA, but the idea is the same.

28.7.8 Counter Updates

Four counters are relevant on every transmit:

- `IFCOUNTER_OPACKETS`: the number of packets transmitted.
- `IFCOUNTER_OBYTES`: the total bytes transmitted.
- `IFCOUNTER_OMCASTS`: the number of multicast or broadcast frames transmitted.
- `IFCOUNTER_OERRORS`: the number of errors observed during transmit.

`if_inc_counter(ifp, IFCOUNTER_*, n)` is the correct way to update these. It is defined in `/usr/src/sys/net/if.c` and uses per-CPU counters internally so that concurrent calls from multiple CPUs do not contend. Do not reach into the `if_data` fields directly: the internals have changed over the years, and the accessor is the stable interface.

Because the stack has already computed packet length and populated `m->m_pkthdr.len`, we cache that into a local `len` variable before we free the mbuf. Reading `m->m_pkthdr.len` after `m_freem(m)` would be use-after-free, so the local variable is not a stylistic choice. It is a correctness choice.

28.7.9 The Final Drop

`m_freem(m)` frees an entire mbuf chain. It walks the chain through the `m_next` pointers and frees every mbuf in it. You do not need to free each one by hand. If you had only `m_free(m)` you would free the first mbuf and leak the rest. Confusing `m_freem` and `m_free` is one of the most common beginner mistakes. The conventional names are:

- `m_free(m)`: free a single mbuf. Rarely called in drivers.
- `m_freem(m)`: free a whole chain. This is what you almost always want.

In a real NIC driver, instead of `m_freem(m)`, we would hand the frame to hardware DMA and free the mbuf later, in the transmit-completion interrupt. For our pseudo-driver, we drop it. This is the behaviour of `if_disc.c` in the tree: simulate the transmit, free the mbuf, and return.

28.7.10 The Queue-Flush Callback

Alongside `if_transmit`, the stack expects a trivial callback called `if_qflush`. It is invoked when the stack wants to flush any packets the driver has queued internally. Because our driver does not queue, the callback has no work to do:

```
static void
mynet_qflush(struct ifnet *ifp __unused)
{
}
```

That is identical to `epair_qflush` in `/usr/src/sys/net/if_epair.c`. Drivers that maintain their own packet queues, which is less common now than it was, have more work to do here. We do not.

28.7.11 The `mynet_init` Callback

The third callback assigned in Section 3 was `mynet_init`, the function the stack calls when the interface transitions to the up state. It is a simple one for us:

```
static void
mynet_init(void *arg)
{
}
```

```
struct mynet_softc *sc = arg;

Mynet_LOCK(sc);
sc->running = true;
sc->ifp->if_drv_flags |= IFF_DRV_RUNNING;
sc->ifp->if_drv_flags &= ~IFF_DRV_OACTIVE;
callout_reset(&sc->rx_callout, sc->rx_interval_hz,
              mynet_rx_timer, sc);
Mynet_UNLOCK(sc);

if_link_state_change(sc->ifp, LINK_STATE_UP);
}
```

On init we mark ourselves running, clear `IFF_DRV_OACTIVE` (a flag meaning “transmit queue is full, do not call me again until I clear it”), start the receive-simulation callout we will describe in Section 5, and announce that the link is up. The `if_link_state_change` call at the end causes `ifconfig` to report status: `active` on this interface. Keep the placement in mind: we set `IFF_DRV_RUNNING` first, then announce the link, in that order. Reversing the order would tell the stack that the link is up on an interface whose driver is still initialising, and the stack might start pushing traffic at us before we are ready.

28.7.12 A Close Look at Ordering and Locking

The code above is simple enough that locking feels like overkill. Why do we need a mutex at all? There are two reasons.

The first reason is that `if_transmit` and `if_ioctl` run concurrently. The stack can call `if_transmit` on one CPU while userland issues `ifconfig mynet0 down` on another, which translates into `if_ioctl(SIOCSIFFLAGS)` running on that other CPU. Without a mutex, these two callbacks can both read and write softc state simultaneously. The mutex is what lets us reason about state transitions.

The second reason is that the callout-based receive simulation in Section 5 touches the softc when it fires. Without a mutex, the callout and `if_ioctl` can collide, and you get the classic “the list I was walking just changed out from under me” style of bug. Again, a single per-softc mutex is enough to make these interactions safe.

We have chosen a simple locking rule: the softc mutex is the big lock. Every softc access outside the transmit fast path takes it. The transmit fast path in `mynet_transmit` does not take the mutex, because `if_transmit` is designed for concurrent callers and we only touch ifnet counters and BPF, both of which are thread-safe on their own. If we were to add driver-specific shared state that transmit updates, we would add a finer-grained lock for that state.

This is a simplification. Real high-performance NIC drivers use far more complex locking, often with per-queue locks, per-CPU state, and per-packet sanity checks. The single-mutex design is absolutely fine for a pseudo-driver and for any low-rate interface; for a production 100-gigabit driver it would become a bottleneck, which is one reason why the modern `iflib` framework exists. We will touch on `iflib` in later chapters.

28.7.13 Packet Surgery with `m_pullup`

Real network drivers frequently need to read fields from deep inside a packet before deciding what to do with it. A VLAN driver needs to read the 802.1Q tag. A bridging driver needs to read the source MAC to update a forwarding table. A hardware-offloading driver needs to read the IP and TCP headers to decide whether a checksum can be computed in hardware.

The trouble is that a received mbuf chain does not guarantee that any particular byte lives in any particular mbuf. The first mbuf might hold only the first fourteen bytes (the Ethernet header) while the next mbuf holds the rest. A driver that casts `mtod(m, struct ip *)` and reaches past the Ethernet header will read nonsense unless it first ensures that the bytes it needs are contiguous.

The kernel provides `m_pullup(m, len)` for exactly this purpose. `m_pullup` guarantees that the first `len` bytes of the mbuf chain live in the first mbuf. If they already do, it is a no-op. If they do not, it reshapes the chain by moving bytes into the first mbuf, possibly allocating a new mbuf if the first one is too small. It returns a (possibly different) mbuf pointer, or NULL on allocation failure, in which case the mbuf chain has been freed for you.

The idiom for a driver that needs to inspect headers is:

```
m = m_pullup(m, sizeof(struct ether_header) + sizeof(struct ip));
if (m == NULL) {
    if_inc_counter(ifp, IFCOUNTER_IQDROPS, 1);
    return;
}
eh = mtod(m, struct ether_header *);
ip = (struct ip *)(eh + 1);
```

`mynet` does not need to do this, because we do not inspect packet contents in the transmit path. But you will see `m_pullup` sprinkled throughout real drivers, especially on the receive side and in L2 helpers.

A related function, `m_copydata(m, offset, len, buf)`, copies bytes out of a mbuf chain into a caller-provided buffer. It is the right tool when you want to read some bytes without modifying the chain. `m_copyback` goes the other way: write bytes into a chain at a given offset, extending the chain if needed.

Another frequently-used helper is `m_defrag(m, how)`, which flattens a chain into a single (large) mbuf. This is used by drivers whose hardware has a maximum scatter-gather count. If a transmit frame spans more mbufs than the hardware can handle, the driver falls back to `m_defrag`, which copies the payload into a contiguous single cluster.

You will meet all of these functions in the course of reading real drivers. For now, knowing that they exist, and that mbuf layout is something a real driver must take seriously, is enough.

28.7.14 A Deeper Look at mbuf Structure

Because mbufs are the currency of the network stack, spending a few more pages on their structure is time well spent. The decisions a driver makes about mbufs are the decisions that determine whether the driver is fast, correct, and maintainable.

The mbuf structure itself lives in `/usr/src/sys/sys/mbuf.h`. The on-disk layout, as of FreeBSD 14.3, is something like this (simplified for teaching):

```
struct mbuf {
    struct m_hdr    m_hdr;      /* fields common to every mbuf */
    union {
        struct {
            struct pkthdr m_pkthdr; /* packet header, if M_PKTHDR */
            union {
                struct m_ext m_ext; /* external storage, if M_EXT */
                char          m_pktdat[MLEN - sizeof(struct pkthdr)];
            } MH_dat;
        } MH;
        char    M_databuf[MLEN]; /* when no packet header */
    } M_dat;
};
```

Two union variants inside two unions. The layout captures the fact that an mbuf can be in one of several modes:

- A plain mbuf with its data stored inline (about 200 bytes available).
- A packet-header mbuf with its data stored inline (slightly less available because of the header).
- A packet-header mbuf with its data stored in an external cluster (`m_ext`).
- A non-header mbuf with its data stored in an external cluster.

The `m_flags` field indicates which variant is in effect through the bits `M_PKTHDR` and `M_EXT`.

A cluster is a larger pre-allocated buffer, typically 2048 bytes on modern FreeBSD. The mbuf holds a pointer to the cluster in `m_ext.ext_buf`, and the cluster is reference-counted through `m_ext.ext_count`. Clusters exist because many packets are larger than a plain mbuf can hold, and allocating a fresh buffer for every large packet would be costly.

When you call `MGETHDR(m, M_NOWAIT, MT_DATA)`, you get a packet-header mbuf with inline data. When you call `m_getcl(M_NOWAIT, MT_DATA, M_PKTHDR)`, you get a packet-header mbuf with an external cluster attached. The second form can hold about 2000 bytes without chaining, which is convenient for Ethernet-sized packets.

28.7.15 mbuf Chains and Scatter-Gather

Because a single mbuf can hold only limited bytes, many packets span several mbufs chained through `m_next`. The `m_pkthdr.len` field on the head mbuf holds the total packet length; the `m_len` on each mbuf in the chain holds that mbuf's contribution. Their relationship is `m_pkthdr.len == sum(m_len across chain)`, and any mismatch is a bug.

This chaining has several advantages. It lets the stack prepend headers cheaply: to add an Ethernet header, the stack can allocate a new mbuf, fill in the header, and link it as the new head. It lets the stack split packets

cheaply: TCP can segment a large payload by walking a chain rather than copying data. It lets the hardware use scatter-gather DMA: a NIC can transmit a chain by issuing multiple DMA descriptors, one per mbuf.

The cost is that drivers must walk chains carefully. If you cast `mtod(m, struct ip *)` and the IP header is split across the first and second mbufs, you read garbage. `m_pullup` is the defence against that mistake, and every serious driver uses it when it needs to inspect headers.

28.7.16 mbuf Types and What They Mean

The `m_type` field on every mbuf classifies its purpose:

- `MT_DATA`: ordinary packet data. This is what you use for network packets.
- `MT_HEADER`: a mbuf dedicated to holding protocol headers.
- `MT_SONAME`: a socket address structure. Used by socket-layer code.
- `MT_CONTROL`: ancillary socket control data.
- `MT_NOINIT`: an uninitialised mbuf. Never seen by drivers.

For driver code, `MT_DATA` is almost always correct. The stack handles the others internally.

28.7.17 Packet Header Fields

The `m_pkthdr` structure on a header mbuf carries fields that travel with the packet through the stack. Some of the most relevant for driver authors:

- `len`: total length of the mbuf chain.
- `rcvif`: the interface on which the packet was received. Drivers set this when constructing a received mbuf.
- `flowid` and `rsstype`: hash of the packet's flow, used for multi-queue dispatch.
- `csum_flags` and `csum_data`: hardware checksum state. Drivers with TX checksum offload read these; drivers with RX checksum offload write them.
- `ether_vtag` and `M_VLANTAG` flag in `m_flags`: hardware-extracted VLAN tag, if VLAN hardware tagging is in use.
- `vt_nrecs` and other VLAN fields: for more elaborate VLAN configurations.
- `tso_segsz`: segment size for TSO frames.

Most of these fields are set by higher layers before the packet reaches the driver. For our purposes, setting `rcvif` during receive and reading `len` during transmit is enough. The other fields are hooks that `iflib` and its predecessors use for offload coordination; a pseudo-driver can safely ignore them.

28.7.18 Reference-Counted External Buffers

When a cluster is attached to a mbuf, the cluster is reference-counted. This allows packet duplication (via `m_copypacket`) without copying the payload: two mbufs can share the same cluster, and the cluster is freed only when both mbufs release their reference. BPF uses this mechanism to tap a packet without forcing a copy.

For driver code this is mostly transparent. You call `m_freem` on your mbuf, and if the mbuf has an external cluster, the cluster's reference count is decremented; if it reaches zero, the cluster is freed. You do not have

to think about the reference counts explicitly. But you should know that they exist, because they explain why BPF_MTAP can be cheap: it does not copy the packet, it merely grabs an additional reference.

28.7.19 The Receive Allocation Pattern

A real NIC driver usually allocates mbufs and attaches clusters to them at initialisation time, fills the receive ring with those mbufs, and lets the hardware DMA into them. The pattern is:

```
for (i = 0; i < RX_RING_SIZE; i++) {
    struct mbuf *m = m_getcl(M_WAITOK, MT_DATA, M_PKTHDR);
    rx_ring[i].mbuf = m;
    rx_ring[i].dma_addr = pmap_kextract((vm_offset_t)mtod(m, char *));
    rx_ring[i].desc->addr = rx_ring[i].dma_addr;
    rx_ring[i].desc->status = 0;
}
```

When the hardware receives a packet, it writes the packet data into the cluster pointed at by one of the descriptors, sets the status to indicate completion, and raises an interrupt. The driver's receive routine looks at the status, grabs the mbuf, sets `m->m_pkthdr.len` and `m->m_len` from the descriptor's length field, taps BPF, calls `if_input`, and then allocates a replacement mbuf for the descriptor.

Our pseudo-driver uses a much simpler pattern: allocate a fresh mbuf every time the receive timer fires. This is perfectly fine for a teaching driver because the allocation rate is low. At higher rates you would want the pre-allocation pattern, because allocating mbufs in bulk at init time and recycling them is much cheaper than allocating one per packet.

28.7.20 Common mbuf-Related Mistakes

Even with knowledge of the above, a few mistakes keep occurring in driver code:

- Using `m_free` instead of `m_freem` on a chain head. You free the first mbuf and leak the rest.
- Forgetting to set `m_pkthdr.len` correctly when building a packet. The stack reads `m_pkthdr.len` rather than walking the chain, so if the two disagree, decoding silently fails.
- Reading `m_pkthdr.len` after `m_freem`. Always cache the length into a local before freeing.
- Confusing `m->m_len` (length of this mbuf) with `m->m_pkthdr.len` (total length of the chain). For a single-mbuf packet they are equal; for chains they differ.
- Reading past `m_len` without walking the chain. If you need bytes beyond the first mbuf, use `m_pullup` or `m_copydata`.
- Modifying a mbuf that you do not own. Once you have handed a mbuf to `if_input`, it is not yours any more.
- Allocating without checking for NULL. `m_gethdr(M_NOWAIT, ...)` can return NULL under memory pressure, and the driver must handle that gracefully.

These mistakes are easy to avoid if you know the rules, and reading other drivers is the best way to internalise them.

28.7.21 Multi-Queue Transmit in Real Drivers

Modern hardware NICs can transmit on many queues in parallel. A 10-gigabit NIC commonly has eight or sixteen transmit queues, each with its own hardware ring buffer, its own DMA descriptors, and its own completion interrupt. The driver distributes outgoing packets among these queues based on a hash of the packet's source and destination addresses, so that traffic from different flows goes to different queues and can be processed concurrently on different CPU cores.

This is far beyond what our pseudo-driver needs. But the pattern is worth recognising, because it shows up prominently in production drivers. The key pieces are:

- A queue-selection function that takes a mbuf and returns an index into the driver's queue array. `mynet` has only one queue (or zero, depending on how you count), so this step is trivial. Real drivers often use `m->m_pkthdr.flowid` as a precomputed hash.
- A per-queue lock and a per-queue software queue (typically managed by `drbr(9)`) that allows concurrent producers to enqueue packets without contention.
- A transmit kick that drains the software queue into hardware when a producer has enqueued and the hardware is idle.
- A completion callback, usually from a hardware interrupt, that frees the mbufs whose transmission has completed.

The `if_transmit` prototype is designed to fit this pattern naturally. The caller produces a mbuf and hands it to `if_transmit`. The driver either queues it immediately (in a simple case like ours) or dispatches it to the appropriate hardware queue (in a multi-queue case). Either way, the caller sees a single function call and does not need to know how many queues live underneath.

We will return to multi-queue design when we discuss `iflib` in a later chapter. For now, it is enough to know that the single-queue model we are building here is a simplification that real drivers elaborate.

28.7.22 A Digression on the `drbr(9)` Helper

`drbr` stands for “driver ring buffer” and it is a helper library for drivers that want to maintain their own per-queue software queue. The API is defined and implemented as static `__inline` functions in `/usr/src/sys/net/ifq.h`; there is no separate `drbr.c` or `drbr.h` file. The helpers wrap the underlying `buf_ring(9)` ring buffers with explicit enqueue and dequeue operations, plus helpers for tapping BPF, counting packets, and synchronising with the transmit thread. The shape `drbr` is built for is multi-producer, single-consumer, which is the typical shape of a transmit queue where many threads enqueue but a single dequeue thread drains the ring into hardware.

A driver that uses `drbr` typically has a transmit function that looks like this sketch:

```
int
my_transmit(struct ifnet *ifp, struct mbuf *m)
{
    struct mydrv_softc *sc = ifp->if_softc;
    struct mydrv_txqueue *txq = select_queue(sc, m);
    int error;
```

```

    error = drbr_enqueue(ifp, txq->br, m);
    if (error)
        return (error);
    taskqueue_enqueue(txq->tq, &txq->tx_task);
    return (0);
}

```

The producer enqueues into a ring buffer and kicks a taskqueue. The taskqueue consumer then dequeues from the ring buffer and hands frames to hardware. This decouples the producer (which can be any CPU) from the consumer (which runs on a dedicated worker thread per queue), which is exactly the structure that works well on multi-core systems.

mynet does not use `drbr`, because we have neither multiple queues nor hardware to kick. But the pattern is worth seeing once, because it shows up in every performance-conscious driver in the tree.

28.7.23 Testing the Transmit Path

Build, load, and create the interface as in Section 3, then send traffic at it:

```

# kldload ./mynet.ko
# ifconfig mynet create
mynet0
# ifconfig mynet0 inet 192.0.2.1/24 up
# ping -c 1 192.0.2.99
PING 192.0.2.99 (192.0.2.99): 56 data bytes
--- 192.0.2.99 ping statistics ---
1 packets transmitted, 0 packets received, 100.0% packet loss
# netstat -in -I mynet0

```

Name	Mtu	Network	Address	Ipkts	Ierrs	...	Opkts	Oerrs
mynet0	1500	<Link#12>	02:a3:f1:22:bc:0d	0	0		1	0
mynet0	-	192.0.2.0/24	192.0.2.1	0	-		0	-

The key line is `Opkts 1`. Even though the ping received no reply, we can see that one packet was transmitted through our driver. The reason there was no reply is that `mynet0` is a pseudo-interface with nothing on the other side. We will give it a simulated arrival path in Section 5.

Leave `tcpdump -i mynet0 -n` running in another terminal, repeat the ping, and you will see the outgoing ARP request and IPv4 packet being captured. That confirms `BPF_MTAP` is wired up correctly.

28.7.24 Pitfalls

A handful of mistakes appear repeatedly in student code and even in experienced drivers. Let us walk through them so you learn to recognise them.

Freeing the mbuf twice. If your transmit function has multiple exit paths and one of them forgets to skip `m_freem`, the same mbuf ends up freed twice. The kernel usually panics with a message about a corrupted free list. The fix is to structure the function with a single exit that owns the free, or to null out `m` after you free it and check before freeing again.

Not freeing the mbuf at all. The other side of the same mistake. If you return from `if_transmit` without freeing or queuing the mbuf, you leak it. In a low-rate driver this might take hours to notice; in a high-rate driver the machine runs out of mbuf memory quickly. `vmstat -z | grep mbuf` is your best friend for spotting this.

Assuming the mbuf fits in a single memory block. Even a simple Ethernet frame can be spread across multiple mbufs in a chain, especially after IP fragmentation or TCP segmentation. If you need to examine the headers, either use `m_pullup` to pull the headers into the first mbuf, or walk the chain carefully.

Forgetting to tap BPF. `tcpdump -i mynet0` will still work for received packets but will miss transmitted ones, and your debugging will be harder because the two halves of the conversation will appear asymmetric.

Updating counters after `m_freem`. We already mentioned this. Always read `m->m_pkthdr.len` into a local before freeing, or do all your counter updates before freeing.

Calling `if_link_state_change` with the wrong argument. `LINK_STATE_UP`, `LINK_STATE_DOWN`, and `LINK_STATE_UNKNOWN` are the three values defined in `/usr/src/sys/net/if.h`. Passing a random integer like 1 might coincidentally match `LINK_STATE_DOWN` but make the code unreadable and fragile.

28.7.25 Wrapping Up Section 4

The transmit path is the clearest demonstration of how the network stack and the driver cooperate. We accept a mbuf, validate it, count it, let BPF see it, and free it. Real hardware drivers add DMA and hardware rings at the bottom; the skeleton remains the same.

We have one large piece missing: the receive path. Without it, our interface talks but never listens. Section 5 builds that half.

28.8 Section 5: Handling Packet Reception

Reception is the inbound half of packet flow. Packets arrive from the transport, and the driver is responsible for turning them into mbufs, giving them to BPF, and handing them up to the stack through `if_input`. In a real NIC driver the arrival is an interrupt or a descriptor-ring completion. In our pseudo-driver, we will simulate arrival with a callout that fires every second and constructs a synthetic packet. The mechanism is artificial, but the code path is identical to what real drivers do after the initial descriptor-ring dequeue.

28.8.1 The Callback Direction

Transmit flows down: stack calls driver. Reception flows up: driver calls stack. You do not register a receive callback for the stack to invoke. Instead, whenever a packet arrives, you call `if_input(ifp, m)` (or equivalently `(*ifp->if_input)(ifp, m)`) and the stack takes over. `ether_ifattach` arranged for `ifp->if_input`

to point at `ether_input`, so when we call `if_input` the Ethernet layer receives the frame, strips the Ethernet header, dispatches on the `EtherType`, and hands the payload up to IPv4, IPv6, ARP, or wherever it belongs.

This is an important mental shift from `if_transmit`. The stack does not poll your driver. It waits to be called. Your driver is the active party for reception. Whenever you have a frame ready, you make the call. The stack does the rest.

28.8.2 The Simulated Arrival

Let us build a simulated arrival path. The idea: once a second, wake up, build a small mbuf containing a valid Ethernet frame, and feed it to the stack. The frame will be a broadcast ARP request targeting a nonexistent IP address. That is easy to construct, useful for testing because `tcpdump` will clearly display it, and harmless to the rest of the system.

First, the callout handler:

```
static void
mynet_rx_timer(void *arg)
{
    struct mynet_softc *sc = arg;
    struct ifnet *ifp = sc->ifp;

    MYNET_ASSERT(sc);
    if (!sc->running) {
        return;
    }
    callout_reset(&sc->rx_callout, sc->rx_interval_hz,
        mynet_rx_timer, sc);
    MYNET_UNLOCK(sc);

    mynet_rx_fake_arp(sc);

    MYNET_LOCK(sc);
}
```

The callout is initialised with `callout_init_mtx` and the softc mutex, so the system acquires our mutex before calling us. That gives us `MYNET_ASSERT` for free: the lock is already held. We check if we are still running, reschedule the timer for the next tick, drop the lock, do the actual work, and reacquire the lock on the way back. Dropping the lock is important, because `if_input` can take its time and may acquire other locks. Calling up into the stack while holding a driver mutex is a recipe for lock-order reversals.

Next, the packet construction itself:

```
static void
mynet_rx_fake_arp(struct mynet_softc *sc)
{
```

```
struct ifnet *ifp = sc->ifp;
struct mbuf *m;
struct ether_header *eh;
struct arphdr *ah;
uint8_t *payload;
size_t frame_len;

frame_len = sizeof(*eh) + sizeof(*ah) + 2 * (ETHER_ADDR_LEN + 4);
MGETHDR(m, M_NOWAIT, MT_DATA);
if (m == NULL) {
    if_inc_counter(ifp, IFCOUNTER_IQDROPS, 1);
    return;
}

m->m_pkthdr.len = m->m_len = frame_len;
m->m_pkthdr.rcvif = ifp;

eh = mtod(m, struct ether_header *);
memset(eh->ether_dhost, 0xff, ETHER_ADDR_LEN); /* broadcast */
memcpy(eh->ether_shost, sc->hwaddr, ETHER_ADDR_LEN);
eh->ether_type = htons(ETHERTYPE_ARP);

ah = (struct arphdr *)(eh + 1);
ah->ar_hrd = htons(ARPHRD_ETHER);
ah->ar_pro = htons(ETHERTYPE_IP);
ah->ar_hln = ETHER_ADDR_LEN;
ah->ar_pln = 4;
ah->ar_op = htons(ARPOP_REQUEST);

payload = (uint8_t *)(ah + 1);
memcpy(payload, sc->hwaddr, ETHER_ADDR_LEN); /* sender MAC */
payload += ETHER_ADDR_LEN;
memset(payload, 0, 4); /* sender IP 0.0.0.0 */
payload += 4;
memset(payload, 0, ETHER_ADDR_LEN); /* target MAC */
payload += ETHER_ADDR_LEN;
memcpy(payload, "\xc0\x00\x02\x63", 4); /* target IP 192.0.2.99 */

BPF_MTAP(ifp, m);

if_inc_counter(ifp, IFCOUNTER_IPACKETS, 1);
if_inc_counter(ifp, IFCOUNTER_IBYTES, frame_len);

if_input(ifp, m);
```

```
}
```

There is a lot to unpack, but most of it is simple. Let us walk through.

28.8.3 MGETHDR: Allocating the Head of the Chain

`MGETHDR(m, M_NOWAIT, MT_DATA)` allocates a new mbuf and prepares it as the head of a packet chain. It expands to `m_gethdr(M_NOWAIT, MT_DATA)` through the compatibility macro block in `/usr/src/sys/sys/mbuf.h` (the `#define MGETHDR(m, how, type) ((m) = m_gethdr((how), (type)))` entry, right next to `MGET` and `MCLGET`). `M_NOWAIT` tells the allocator to fail rather than sleep, which is appropriate because we may run in contexts where sleeping is forbidden (this particular callback is a callout, which cannot sleep). `MT_DATA` is the mbuf type for generic data.

On allocation failure we increment `IFCOUNTER_IQDROPS` (input queue drops) and return. Drops caused by mbuf starvation are counted this way in most drivers.

28.8.4 Setting Packet Header Fields

Once we have the mbuf, we set three fields in the packet header:

- `m->m_pkthdr.len`: the total length of the packet. This is the sum of `m_len` across the chain. For a single-mbuf packet like ours, `m_pkthdr.len` equals `m_len`.
- `m->m_len`: the length of data in this mbuf. We are storing the whole frame in the first (and only) mbuf.
- `m->m_pkthdr.rcvif`: the interface on which the packet arrived. The stack uses this for routing decisions and for reporting.

A small mbuf (about 256 bytes) holds our 42-byte Ethernet ARP frame comfortably. If we were building a larger frame, we would use `MGET` and external buffers, or `m_getcl` for a cluster-backed mbuf, or chain several mbufs together. We will revisit those patterns in later chapters.

28.8.5 Writing the Ethernet Header

`mtod(m, struct ether_header *)` is a macro from `/usr/src/sys/sys/mbuf.h` that casts `m_data` to a pointer to the requested type. It stands for “mbuf to data”. We use it to get a writeable `struct ether_header` pointer at the start of the packet and fill in the destination MAC (broadcast `ff:ff:ff:ff:ff:ff`), the source MAC (our interface’s MAC), and the EtherType (`ETHERTYPE_ARP`, in network byte order).

The Ethernet header is the minimum layer-2 encapsulation the stack expects on our interface, because we attached with `ether_ifattach`. `ether_input` will strip this header and dispatch on the EtherType.

28.8.6 Writing the ARP Body

Past the Ethernet header comes the ARP header proper, then the ARP payload (sender MAC, sender IP, target MAC, target IP). The field names and constants come from `/usr/src/sys/net/if_arp.h`. We put a real sender MAC (ours), a sender IP of `0.0.0.0`, a zero target MAC, and a target IP of `192.0.2.99`. That last

address is in the TEST-NET-1 range reserved by RFC 5737 for documentation and examples, which is a polite choice for a synthetic packet that will never leave our system.

None of this is production-grade ARP code. We are not trying to resolve anything. We are generating a well-formed frame that the Ethernet input layer will recognise, parse, log in counters, and drop (because the target IP is not ours). It is exactly the right level of realism for a teaching driver.

28.8.7 Handing It to BPF

`BPF_MTAP(ifp, m)` gives `tcpdump` a chance to see the incoming frame. We tap before we call `if_input`, because `if_input` may mutate the mbuf in ways that would make the tap show confusing data. Real drivers always tap before consuming.

28.8.8 Incrementing Input Counters

`IFCOUNTER_IPACKETS` and `IFCOUNTER_IBYTES` count received packets and bytes respectively. If the frame is a broadcast or multicast, we would also increment `IFCOUNTER_IMCASTS`. We omit that here for brevity but the full companion file includes it.

28.8.9 Calling `if_input`

`if_input(ifp, m)` is the final step. It is an inline helper in `/usr/src/sys/net/if_var.h` that dereferences `ifp->if_input` (which `ether_ifattach` set to `ether_input`) and invokes it. From that moment, the mbuf is the stack's responsibility. If the stack accepts the packet, it uses it and eventually frees it. If the stack rejects the packet, it frees it and increments `IFCOUNTER_IERRORS`. Either way, we must not touch `m` again.

This is the complementary rule to transmit: in transmit, the driver owns the mbuf until it is freed or handed to hardware; in receive, the stack takes ownership the moment you call `if_input`. Getting these ownership rules right is the single most important discipline in writing network drivers.

28.8.10 Verifying the Receive Path

Build and load the updated driver, bring up the interface, and watch `tcpdump`:

```
# kldload ./mynet.ko
# ifconfig mynet create
mynet0
# ifconfig mynet0 inet 192.0.2.1/24 up
# tcpdump -i mynet0 -n
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on mynet0, link-type EN10MB (Ethernet), capture size 262144 bytes
14:22:01.000 02:a3:f1:22:bc:0d > ff:ff:ff:ff:ff:ff, ethertype ARP, Request who-has
↪ 192.0.2.99 tell 0.0.0.0, length 28
14:22:02.000 02:a3:f1:22:bc:0d > ff:ff:ff:ff:ff:ff, ethertype ARP, Request who-has
↪ 192.0.2.99 tell 0.0.0.0, length 28
...
```

Every second, you should see one synthesised ARP request fly by. If you then check `netstat -in -I mynet0`, the `IpKts` counter should be climbing. The stack accepts the packet, inspects the ARP, decides it is not a question addressed to it (because `192.0.2.99` is not assigned to the interface), and silently drops it. That is exactly what we want, and it demonstrates that the full receive path works.

28.8.11 Ownership: A Diagram

Because the ownership rules are so important, it helps to draw them. The following diagram summarises who owns the mbuf at each stage:

Transmit:

```
stack allocates mbuf
stack calls if_transmit(ifp, m)    <-- ownership handed to driver
driver inspects, counts, taps, drops or sends
driver must m_freem(m) exactly once
return 0 to stack
```

Receive:

```
driver allocates mbuf (MGETHDR/MGET)
driver fills in data
driver taps BPF
driver calls if_input(ifp, m)      <-- ownership handed to stack
driver MUST NOT touch m again
```

If you keep these two diagrams in your head, you will not get mbuf ownership wrong in your own drivers.

28.8.12 Keeping Receive Safe Under Contention

A production driver's receive path is usually called from an interrupt handler or a hardware queue completion routine, running on one CPU while another CPU may be transmitting or handling `ioctl`s. The pattern we have shown here is safe because:

- We hold our mutex around the “am I running?” check.
- We drop the mutex before the heavy work of mbuf allocation and packet construction.
- We drop the mutex before calling `if_input`, which may in turn call into the stack and acquire other locks.
- We reacquire our mutex after `if_input` returns, so that the callout framework can see a consistent state.

Real drivers often add per-CPU receive queues, deferred processing via taskqueues, and lock-free counters. All of that is a refinement of the same pattern. The core invariants remain the same: do not call up with a driver lock held, and do not touch an mbuf after it has been handed up.

28.8.13 Alternative: Using `if_epoch`

FreeBSD 12 introduced a network epoch mechanism, `net_epoch`, for accessing certain data structures without long-lived locks. Modern drivers often enter the net epoch around receive code to make their access to the routing table, ARP tables, and some parts of the `ifnet` list safe and fast. You will see `NET_EPOCH_ENTER(et)` and `NET_EPOCH_EXIT(et)` in many drivers. For our simple pseudo-driver, entering `net_epoch` would add complexity we do not need. We mention it here so that you recognise it when you read `if_em.c` or `if_bge.c`, and we will return to it in later chapters.

28.8.14 Receive Paths in Real NIC Drivers

Our simulated receive path is artificial, but the surrounding structure is exactly what real drivers use. The differences are in where the mbuf comes from and who calls the receive routine, not in what the receive routine does afterwards. This subsection walks through the typical real-driver receive path so that you recognise it the next time you open an Ethernet driver in the tree.

On a real NIC, packets arrive as DMA writes to receive descriptors in ring buffers. The hardware populates each descriptor with a pointer to a pre-allocated mbuf (provided by the driver during initialisation), a length, and a status field indicating whether the descriptor is ready for the driver to process. When a descriptor is ready, the hardware either raises an interrupt or sets a bit that the driver will notice through polling, or both.

The driver's receive routine walks the ring starting at the last-processed index. For each ready descriptor, it reads the length and status, fixes up the corresponding mbuf to have the correct `m_len` and `m_pkthdr.len`, sets `m->m_pkthdr.rcvif = ifp`, taps BPF, updates counters, and calls `if_input`. It then allocates a replacement mbuf to put back into the descriptor, so that future packets have somewhere to land, and advances the head pointer.

This loop continues until either the ring is empty or the driver has processed its per-invocation budget. Processing too many packets in one interrupt starves other interrupts and hurts latency for other devices; processing too few wastes context switches. A budget of 32 or 64 packets is typical.

After the receive loop, the driver updates the hardware's tail pointer to reflect the newly-replenished descriptors. If any descriptors are still ready, the driver either re-arms the interrupt or schedules itself to run again through a taskqueue.

The completion routine for transmit is the mirror image: it walks the transmit ring looking for descriptors whose status indicates the hardware has finished with them, frees the corresponding mbufs, and updates the driver's sense of available transmit slots.

You will see all of this in `/usr/src/sys/dev/e1000/em_txrx.c` and its equivalents for other Ethernet hardware. The ring-buffer machinery looks intimidating at first, but its purpose is always the same: produce mbufs from hardware DMA and hand them up through `if_input`. Our pseudo-driver produces mbufs from `malloc` and hands them up through `if_input`. The hand-up is identical; only the source of mbufs differs.

28.8.15 Deferred Receive Processing with Taskqueues

A common refinement in high-rate drivers is to defer the actual receive processing out of the interrupt context and into a taskqueue. The interrupt handler does the minimum amount of work (typically acknowledging the

interrupt to hardware and scheduling the task), and the taskqueue worker thread does the ring walk and the `if_input` calls.

Why defer? Because `if_input` can do significant work inside the stack, including TCP processing, socket-buffer deposition, and sleep operations. Holding a CPU in an interrupt handler for that long is bad for interrupt latency on other devices. Moving receive processing to a taskqueue lets the scheduler interleave it with other work.

FreeBSD's taskqueue subsystem, `/usr/src/sys/kern/subr_taskqueue.c`, provides per-CPU worker threads that can be targeted by drivers. A receive interrupt handler looks like:

```
static void
my_rx_intr(void *arg)
{
    struct mydrv_softc *sc = arg;

    /* Acknowledge the interrupt. */
    write_register(sc, RX_INT_STATUS, RX_READY);

    /* Defer the actual work. */
    taskqueue_enqueue(sc->rx_tq, &sc->rx_task);
}

static void
my_rx_task(void *arg, int pending __unused)
{
    struct mydrv_softc *sc = arg;

    mydrv_rx_drain(sc);      /* walk the ring and if_input each packet */
}
```

Again, `mynet` is a pseudo-driver and does not need this complexity. But seeing the pattern means that when you read `if_em.c` or `if_ixl.c` and see `taskqueue_enqueue`, you know what is being deferred and why.

28.8.16 Understanding `net_epoch`

The `net_epoch` framework in FreeBSD is an implementation of epoch-based reclamation adapted for the networking subsystem. Its purpose is to let readers of networking data structures (routing tables, ARP tables, interface lists, and so on) read those structures without acquiring locks, while ensuring that writers do not free a structure while a reader might still be looking at it.

The API is simple. A reader enters the epoch with `NET_EPOCH_ENTER(et)` and exits with `NET_EPOCH_EXIT(et)`, where `et` is a per-call tracker variable. Between enter and exit, the reader can safely dereference pointers into the protected data structures. Writers that want to free a protected object call `epoch_call` to defer the free until all current readers have exited.

For driver code, the relevance is this: the stack routines you call from your receive path, including `ether_input` and its downstream callers, expect to be invoked while the caller is inside the net epoch. Some drivers therefore wrap their `if_input` calls in `NET_EPOCH_ENTER/NET_EPOCH_EXIT`. Others (and this includes most callout-based pseudo-drivers) rely on the fact that `if_input` itself enters the epoch on entry if not already inside it.

For `mynet`, we do not enter the epoch explicitly. `if_input` handles that for us. If you want to be extra careful or are operating in a context where the epoch is known not to be entered, you can wrap your call like this:

```
struct epoch_tracker et;

NET_EPOCH_ENTER(et);
if_input(ifp, m);
NET_EPOCH_EXIT(et);
```

This is the idiom you will see in more recent drivers. We have omitted it in the main chapter text because it adds noise without changing behaviour for our pseudo-driver. In a driver that may fire `if_input` from unusual contexts (for example, a workqueue or a timer tick scheduled on a non-network CPU), you would want to wrap explicitly.

28.8.17 Receive Backpressure

A driver that receives packets faster than the stack can process them will eventually overrun its ring buffer. Real drivers handle this in one of two ways: they drop the oldest pending packets and update `IFCOUNTER_IQDROPS`, or they stop taking new descriptors and let the hardware itself drop.

In software pseudo-drivers there is no hardware to run out of descriptors, but you should still think about backpressure. If your simulated receive path is generating packets faster than the stack can consume them, you will eventually see mbuf allocation failures, or the system will start queueing packets in socket buffers without ever emptying them. The practical defence is to rate-limit yourself through the callout interval and to watch `vmstat -z | grep mbuf` during long-running tests.

For `mynet`, we generate one synthetic ARP per second. That is several orders of magnitude below any reasonable backpressure threshold. But if you increase `sc->rx_interval_hz` to something aggressive like `hz / 1000` (one packet per millisecond), you are asking the kernel to absorb a thousand ARPs per second from a single driver, and you will see the costs.

28.8.18 Common Mistakes

The most common receive-path mistakes are the following.

Forgetting `M_PKTHDR` discipline. If you construct the mbuf without `MGETHDR`, you do not get a packet header, and the stack will assert or misbehave. Always use `MGETHDR` (or `m_gethdr`) for the head mbuf, and `MGET` (or `m_get`) for subsequent ones.

Forgetting to set `m_len` and `m_pkthdr.len`. The stack uses `m_pkthdr.len` to decide how big the packet is, and it uses `m_len` to walk the chain. If these are wrong, decoding fails silently.

Holding the driver mutex across `if_input`. The stack can take a long time inside `if_input`, and it may attempt to acquire other locks. Releasing the driver lock before calling up is a discipline that avoids deadlocks.

Touching `m` after `if_input`. The stack may have already freed or re-queued the mbuf. Treat `if_input` as a one-way door.

Feeding raw data without a link-layer header. Because we used `ether_ifattach`, `ether_input` expects a full Ethernet frame. If you feed it a bare IPv4 packet, it will reject the frame and increment `IFCOUNTER_IERRORS`.

28.8.19 Wrapping Up Section 5

We now have two-way traffic through our driver. Transmit consumes mbufs from the stack; receive produces mbufs for the stack. In between we have BPF hooks, counter updates, and mutex discipline. What we do not yet have is a careful story for link state, media descriptors, and interface flags. That is Section 6.

28.9 Section 6: Media State, Flags, and Link Events

Up to this point we have focused on packets. But a network interface is more than a packet mover. It is a stateful participant in the network stack. It goes up and it goes down. It has a media type, and that media can change. Its link can come and go. The stack cares about all of those transitions, and userland tools present them to the administrator. In this section we add the state-management layer to mynet.

28.9.1 Interface Flags: `IFF_` and `IFF_DRV_`

You have already met `IFF_UP` and `IFF_DRV_RUNNING`. There are many more, and they divide into two families that work in distinct ways.

The `IFF_` flags, defined in `/usr/src/sys/net/if.h`, are the user-visible flags. They are what `ifconfig` reads and writes. Common ones include:

- `IFF_UP` (0x1): the interface is administratively up.
- `IFF_BROADCAST` (0x2): the interface supports broadcast.
- `IFF_POINTOPOINT` (0x10): the interface is point-to-point.
- `IFF_LOOPBACK` (0x8): the interface is a loopback.
- `IFF_SIMPLEX` (0x800): the interface cannot hear its own transmissions.
- `IFF_MULTICAST` (0x8000): the interface supports multicast.
- `IFF_PROMISC` (0x100): the interface is in promiscuous mode.
- `IFF_ALLMULTI` (0x200): the interface is receiving all multicast.
- `IFF_DEBUG` (0x4): debug tracing is requested by the user.

These flags are set and cleared primarily by userland through `SIOCSIFFLAGS`. Your driver should react to changes in them: when `IFF_UP` goes from clear to set, initialise; when it goes from set to clear, quiesce.

The `IFF_DRV_` flags, also in `if.h`, are driver-private. They live in `ifp->if_drv_flags` (not `if_flags`). Userland cannot see or modify them. The two most important are:

- `IFF_DRV_RUNNING (0x40)`: the driver has allocated its per-interface resources and can move traffic. Identical to the older `IFF_RUNNING` alias.
- `IFF_DRV_OACTIVE (0x400)`: the driver's output queue is full. The stack should not call `if_start` or `if_transmit` again until this flag clears.

Think of `IFF_UP` as the user's intent and `IFF_DRV_RUNNING` as the driver's readiness. Both need to be true for traffic to flow.

28.9.2 The `SIOCSIFFLAGS` `ioctl`

When userland runs `ifconfig mynet0 up`, it sets `IFF_UP` in the interface's flag field and issues `SIOCSIFFLAGS`. The stack dispatches this `ioctl` through our `if_ioctl` callback. Our job is to notice the flag change and react.

Here is the canonical pattern for handling `SIOCSIFFLAGS` in a network driver:

```
case SIOCSIFFLAGS:
    MYNET_LOCK(sc);
    if (ifp->if_flags & IFF_UP) {
        if ((ifp->if_drv_flags & IFF_DRV_RUNNING) == 0) {
            MYNET_UNLOCK(sc);
            mynet_init(sc);
            MYNET_LOCK(sc);
        }
    } else {
        if (ifp->if_drv_flags & IFF_DRV_RUNNING) {
            MYNET_UNLOCK(sc);
            mynet_stop(sc);
            MYNET_LOCK(sc);
        }
    }
    MYNET_UNLOCK(sc);
    break;
```

Let us parse this.

If `IFF_UP` is set, we check whether the driver is already running. If not, we invoke `mynet_init` to initialise. If the driver is already running, we do nothing: the user setting the flag again is a no-op.

If `IFF_UP` is not set, we check whether we were running. If so, we call `mynet_stop` to quiesce. If not, again a no-op.

We drop the lock before calling `mynet_init` or `mynet_stop`, because those functions may take time and may internally reacquire the lock. The pattern of “unlock, call, relock” is a standard idiom for `ioctl` handlers.

28.9.3 Writing `mynet_stop`

`mynet_init` we wrote in Section 4. Its counterpart `mynet_stop` is similar but in reverse:

```
static void
mynet_stop(struct mynet_softc *sc)
{
    struct ifnet *ifp = sc->ifp;

    MYNET_LOCK(sc);
    sc->running = false;
    ifp->if_drv_flags &= ~IFF_DRV_RUNNING;
    callout_stop(&sc->rx_callout);
    MYNET_UNLOCK(sc);

    if_link_state_change(ifp, LINK_STATE_DOWN);
}
```

We clear our running flag, drop the IFF_DRV_RUNNING bit so the stack knows we are not carrying traffic, stop the receive callout, and announce link-down to the stack. This is the symmetric partner to the init function.

28.9.4 Link State: `if_link_state_change`

`if_link_state_change(ifp, state)` is the canonical way for a driver to report link transitions. The values come from `/usr/src/sys/net/if.h`:

- `LINK_STATE_UNKNOWN (0)`: the driver does not know the link state. This is the initial value.
- `LINK_STATE_DOWN (1)`: no carrier, no link partner reachable.
- `LINK_STATE_UP (2)`: link is up, link partner is reachable, carrier is present.

The stack records the new state, sends a routing socket notification, wakes up any sleepers on the interface state, and lets userland know through `ifconfig's status: line`. Real NIC drivers call `if_link_state_change` from the link-state-change interrupt handler, typically in response to PHY autonegotiation completion or loss. For pseudo-drivers, we choose when to call it based on the driver's own logic.

It is worth being deliberate about when you call this function. In `mynet_init` we call it with `LINK_STATE_UP` after we have set `IFF_DRV_RUNNING`. In `mynet_stop` we call it with `LINK_STATE_DOWN` after we have cleared `IFF_DRV_RUNNING`. If you reverse the order, you will briefly be reporting a link up on an interface that is not running, or a link down on an interface that still claims to be running. The stack can cope, but the symptoms of the reversal are confusing.

28.9.5 Media Descriptors

Above link state sits media. Media is the description of what kind of connection is in use: 10BaseT, 100BaseT, 1000BaseT, 10GBaseSR, and so on. It is not the same as link state: a connection can have a known media type even when the link is down.

FreeBSD's media subsystem lives in `/usr/src/sys/net/if_media.c` and its header `/usr/src/sys/net/if_media.h`. Drivers use it through a small API:

- `ifmedia_init(ifm, dontcare_mask, change_fn, status_fn)`: initialise the descriptor.
- `ifmedia_add(ifm, word, data, aux)`: add a media entry.
- `ifmedia_set(ifm, word)`: choose the default entry.
- `ifmedia_ioctl(ifp, ifr, ifm, cmd)`: handle `SIOCGIFMEDIA` and `SIOCSIFMEDIA`.

The “word” is a bitfield combining the media subtype and flags. For Ethernet drivers you combine `IFM_ETHER` with a subtype like `IFM_1000_T` (1000BaseT), `IFM_10G_T` (10GBaseT), or `IFM_AUTO` (autonegotiate). The full set of subtypes is enumerated in `if_media.h`.

We set up the descriptor in Section 3:

```
ifmedia_init(&sc->media, 0, mynet_media_change, mynet_media_status);
ifmedia_add(&sc->media, IFM_ETHER | IFM_1000_T | IFM_FDX, 0, NULL);
ifmedia_add(&sc->media, IFM_ETHER | IFM_AUTO, 0, NULL);
ifmedia_set(&sc->media, IFM_ETHER | IFM_AUTO);
```

The callbacks are what the stack invokes when userland queries or sets the media:

```
static int
mynet_media_change(struct ifnet *ifp __unused)
{
    /* In a real driver, program the PHY here. */
    return (0);
}

static void
mynet_media_status(struct ifnet *ifp, struct ifmediareq *imr)
{
    struct mynet_softc *sc = ifp->if_softc;

    imr->ifm_status = IFM_AVALID;
    if (sc->running)
        imr->ifm_status |= IFM_ACTIVE;
    imr->ifm_active = IFM_ETHER | IFM_1000_T | IFM_FDX;
}
```

`mynet_media_change` is the stub: there is no PHY to reprogram in a pseudo-driver. `mynet_media_status` is what `ifconfig` reports through `SIOCGIFMEDIA`: `ifm_status` gets `IFM_AVALID` (the status fields are valid) and `IFM_ACTIVE` (the link is currently active) when we are running, and `ifm_active` tells the caller which media we are actually using.

The `ioctl` handler routes media requests to `ifmedia_ioctl`:

```
case SIOCGIFMEDIA:
case SIOCSIFMEDIA:
```

```
error = ifmedia_ioctl(ifp, ifr, &sc->media, cmd);  
break;
```

This is exactly the pattern used by the SIOCSIFMEDIA / SIOCGIFMEDIA case inside `epair_ioctl` in `/usr/src/sys/net/if_epair.c`.

With this in place, `ifconfig mynet0` will report something like:

```
mynet0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> metric 0 mtu 1500  
ether 02:a3:f1:22:bc:0d  
inet 192.0.2.1 netmask 0xffffffff broadcast 192.0.2.255  
media: Ethernet autoselect (1000baseT <full-duplex>)  
status: active
```

28.9.6 Handling MTU Changes

SIOCSIFMTU is the `ioctl` the user issues when running `ifconfig mynet0 mtu 1400`. A well-behaved driver checks that the requested value is within its supported range and then updates `if_mtu`. Our code:

```
case SIOCSIFMTU:  
    if (ifr->ifr_mtu < 68 || ifr->ifr_mtu > 9216) {  
        error = EINVAL;  
        break;  
    }  
    ifp->if_mtu = ifr->ifr_mtu;  
    break;
```

The lower limit of 68 bytes matches the smallest IPv4 payload plus headers. The upper limit of 9216 is a generous jumbo-frame bound. Real drivers have narrower ranges that match what their hardware can fragment. We keep the range permissive because this is a pseudo-driver.

28.9.7 Handling Multicast Group Changes

SIOCADDMULTI and SIOCDELMULTI signal that the user has added or removed a multicast group on the interface. For a real NIC that implements hardware multicast filtering, the driver would reprogram the filter each time. Our pseudo-driver has no filter, so we simply acknowledge the request:

```
case SIOCADDMULTI:  
case SIOCDELMULTI:  
    /* Nothing to program. */  
    break;
```

This is good enough for correct operation. The stack will deliver multicast traffic to the interface based on its internal group list, and we do not need to do anything special.

28.9.8 Putting the ioctl Handler Together

With all of the above, the full `mynet_ioctl` looks like this:

```
static int
mynet_ioctl(struct ifnet *ifp, u_long cmd, caddr_t data)
{
    struct mynet_softc *sc = ifp->if_softc;
    struct ifreq *ifr = (struct ifreq *)data;
    int error = 0;

    switch (cmd) {
    case SIOCSIFFLAGS:
        MYNET_LOCK(sc);
        if (ifp->if_flags & IFF_UP) {
            if ((ifp->if_drv_flags & IFF_DRV_RUNNING) == 0) {
                MYNET_UNLOCK(sc);
                mynet_init(sc);
                MYNET_LOCK(sc);
            }
        } else {
            if (ifp->if_drv_flags & IFF_DRV_RUNNING) {
                MYNET_UNLOCK(sc);
                mynet_stop(sc);
                MYNET_LOCK(sc);
            }
        }
        MYNET_UNLOCK(sc);
        break;

    case SIOCSIFMTU:
        if (ifr->ifr_mtu < 68 || ifr->ifr_mtu > 9216) {
            error = EINVAL;
            break;
        }
        ifp->if_mtu = ifr->ifr_mtu;
        break;

    case SIOCADDMULTI:
    case SIOCDELMULTI:
        break;

    case SIOCGIFMEDIA:
    case SIOCSIFMEDIA:
        error = ifmedia_ioctl(ifp, ifr, &sc->media, cmd);
    }
```

```
        break;

    default:
        /* Let the common ethernet handler process this. */
        error = ether_ioctl(ifp, cmd, data);
        break;
    }

    return (error);
}
```

The default case delegates to `ether_ioctl`, which handles the `ioctl`s that every Ethernet driver handles the same way (for example `SIOCSIFADDR`, `SIOCSIFCAP` in the common cases). That saves us writing fifteen lines of boilerplate. `/usr/src/sys/net/if_epair.c` does the same in the default arm of the switch in `epair_ioctl`.

28.9.9 Flag Coherence Rules

There are a few coherence rules you should keep in mind as you write driver state transitions:

1. `IFF_DRV_RUNNING` follows `IFF_UP`, not the other way around. The user sets `IFF_UP`, and the driver sets or clears `IFF_DRV_RUNNING` in response.
2. Link state changes should happen after `IFF_DRV_RUNNING` transitions, not before.
3. Callouts and taskqueues that were started when you set `IFF_DRV_RUNNING` should be stopped or drained when you clear it.
4. `if_input` calls should only happen when `IFF_DRV_RUNNING` is set. Otherwise, you are delivering packets on an interface that the stack has not finished bringing up.
5. `if_transmit` may be called even when `IFF_UP` is clear, because of a race between userland and the stack. Your transmit path should check the flags and drop gracefully if either is clear.

These rules are implicit in the code of every well-written driver. Making them explicit is useful when you are first learning.

28.9.10 Interface Capabilities in Depth

We touched on capabilities in Section 3 when we set `IFCAP_VLAN_MTU`. Capabilities deserve a fuller treatment here, because they are how a driver tells the stack which offloads it can perform, and they are increasingly central to how fast drivers stay fast.

The `if_capabilities` field, defined in `/usr/src/sys/net/if.h`, is a bitmask of capabilities the hardware can perform. The `if_capenable` field is a bitmask of capabilities currently enabled. They are separated because userland can toggle individual offloads at runtime through `ifconfig mynet0 -rxcsu` or `ifconfig mynet0 +tso`, and the driver must honour that choice.

The common capabilities are:

- `IFCAP_RXCSUM` and `IFCAP_RXCSUM_IPV6`: the driver will verify IPv4 and IPv6 checksums in hardware and mark correctly-summed packets with `CSUM_DATA_VALID` in the mbuf's `m_pkthdr.csum_flags`.
- `IFCAP_TXCSUM` and `IFCAP_TXCSUM_IPV6`: the driver will compute TCP, UDP, and IP checksums in hardware for outbound packets whose `m_pkthdr.csum_flags` requests it.
- `IFCAP_TS04` and `IFCAP_TS06`: the driver accepts large TCP segments and the hardware splits them into MTU-sized frames on the wire. This dramatically reduces CPU load for TCP-heavy workloads.
- `IFCAP_LRO`: the driver aggregates multiple received TCP segments into a single large mbuf before handing up. Symmetric to TSO on the receive side.
- `IFCAP_VLAN_HWTAGGING`: the driver will add and strip 802.1Q VLAN tags in hardware rather than in software. This saves a mbuf copy per VLAN frame.
- `IFCAP_VLAN_MTU`: the driver can carry VLAN-tagged frames whose total length slightly exceeds the standard Ethernet MTU because of the extra 4-byte tag.
- `IFCAP_JUMBO_MTU`: the driver supports frames larger than 1500 bytes of payload.
- `IFCAP_WOL_MAGIC`: wake-on-LAN using the magic packet.
- `IFCAP_POLLING`: classic device polling, now rarely used.
- `IFCAP_NETMAP`: the driver supports `netmap(4)` kernel-bypass packet I/O.
- `IFCAP_TOE`: TCP offload engine. Rare, but exists on some high-end NICs.

Advertising a capability makes a promise to the stack that you will honour it. If you claim `IFCAP_TXCSUM` but do not actually compute the TCP checksum for outbound frames, the kernel will happily hand you packets with an uncomputed checksum and expect you to finish the job. The receiver will get corrupt frames and discard them. The symptom is silent data loss, which is painful to debug.

For `mynet`, we honestly advertise only what we can deliver. `IFCAP_VLAN_MTU` is the only capability we claim, and we honour it by accepting frames up to `ifp->if_mtu + sizeof(struct ether_vlan_header)` in our transmit path.

A well-behaved driver also handles `SIOCSIFCAP` in its `ioctl` handler so that the user can toggle specific offloads:

```
case SIOCSIFCAP:
    mask = ifr->ifr_reqcap ^ ifp->if_capenable;
    if (mask & IFCAP_VLAN_MTU)
        ifp->if_capenable ^= IFCAP_VLAN_MTU;
    /* Reprogram hardware if needed. */
    break;
```

For a pseudo-driver there is no hardware to reprogram, but the user-visible toggle still works because the `ioctl` updates `if_capenable` and every subsequent transmit decision reads that field.

28.9.11 The `ether_ioctl` Common Handler

We saw earlier that `mynet_ioctl` delegates unknown `ioctls` to `ether_ioctl`. It is worth peeking at what that function does, because it explains why most drivers can get away with handling only a handful of `ioctls` explicitly.

`ether_ioctl`, defined in `/usr/src/sys/net/if_etherubr.c`, is a generic handler for the `ioctls` that every Ethernet interface treats the same way. Its responsibilities include:

- `SIOCSIFADDR`: the user is assigning an IP address to the interface. `ether_ioctl` handles the ARP probe and the address registration. It invokes the driver's `if_init` callback if the interface is down and should be brought up.
- `SIOCGIFADDR`: return the link-layer address of the interface.
- `SIOCSIFMTU`: if the driver does not provide its own handler, `ether_ioctl` performs the generic MTU change by updating `if_mtu`.
- `SIOCADDMULTI` and `SIOCDELMULTI`: update the driver's multicast filter, if one exists.
- Various capability-related `ioctls`.

Because the default handler handles so much, drivers typically only need to handle `ioctls` that require driver-specific logic: `SIOCSIFFLAGS` for the up/down transition, `SIOCSIFMEDIA` to reprogram media, and `SIOCSIFCAP` to toggle capabilities. Everything else falls through to `ether_ioctl`.

This delegation model is one of the things that makes writing a small Ethernet driver pleasant: you write the code that is specific to your driver, and the common code takes care of the rest.

28.9.12 Hardware Multicast Filtering

For a real NIC, multicast filtering is often done in hardware. The driver programs a set of MAC addresses into a hardware filter table, and the NIC only delivers frames whose destination matches an address in the table. When the user runs `ifconfig mynet0 addm 01:00:5e:00:00:01` to join a multicast group, the stack issues `SIOCADDMULTI`, and the driver must update the filter table.

The typical pattern in a real driver is:

```
case SIOCADDMULTI:
case SIOCDELMULTI:
    if (ifp->if_drv_flags & IFF_DRV_RUNNING) {
        MYDRV_LOCK(sc);
        mydrv_setup_multicast(sc);
        MYDRV_UNLOCK(sc);
    }
    break;
```

`mydrv_setup_multicast` walks the interface's multicast list (accessed through `if_maddr_r` lock and friends) and programs each address into the hardware filter. The code is boring but important; getting it wrong means multicast applications like mDNS (Bonjour, Avahi), IGMP-based routing, and IPv6 neighbour discovery quietly misbehave.

For `mynet` we have no hardware filter, so we simply accept `SIOCADDMULTI` and `SIOCDELMULTI` without doing anything. The stack still tracks the list of multicast groups for us, and our receive path does not filter, so everything works.

If you ever write a driver with hardware multicast filtering, read `/usr/src/sys/dev/e1000/if_em.c`'s `em_multi_set` function for a clear example of the pattern.

28.9.13 Wrapping Up Section 6

We have covered the state half of a network driver. Flags, link state, media descriptors, and the ioctls that tie them all together. Combined with the transmit and receive paths from Sections 4 and 5, we now have a driver that is indistinguishable from a simple real Ethernet driver at the `ifnet` boundary.

Before we can call the driver finished, we need to make sure we can test it thoroughly with the tools the FreeBSD ecosystem provides. That is Section 7.

28.10 Section 7: Testing the Driver With Standard Networking Tools

A driver is only as good as your confidence that it works. Confidence does not come from staring at code. It comes from running the driver, interacting with it from the outside, and observing the results. This section walks through the standard FreeBSD networking tools and shows how to use each one to exercise a specific aspect of `mynet`.

28.10.1 Load, Create, Configure

Start with a clean slate. If the module is loaded, unload it, then load the fresh build and create the first interface:

```
# kldstat | grep mynet
# kldload ./mynet.ko
# ifconfig mynet create
mynet0
```

`ifconfig mynet0` should show the interface with a MAC address, no IP, no flags beyond the default set, and a media descriptor saying “autoselect”. Assign an address and bring it up:

```
# ifconfig mynet0 inet 192.0.2.1/24 up
# ifconfig mynet0
mynet0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> metric 0 mtu 1500
    ether 02:a3:f1:22:bc:0d
    inet 192.0.2.1 netmask 0xffffffff broadcast 192.0.2.255
    media: Ethernet autoselect (1000baseT <full-duplex>)
    status: active
    groups: mynet
```

The `UP` and `RUNNING` flags confirm that the user’s intent and the driver’s readiness are both present. The `status: active` line comes from our media callback. The media description includes `1000baseT` because that is what `mynet_media_status` returned.

28.10.2 Inspecting With netstat

`netstat -in -I mynet0` shows per-interface counters. Initially, everything is zero; wait a few seconds for the receive simulation to kick in and the counter should climb:

```
# netstat -in -I mynet0
```

Name	Mtu	Network	Address	Ipkts	Ierrs	...	Opkts	Oerrs
mynet0	1500	<Link#12>	02:a3:f1:22:bc:0d	3	0		0	0
mynet0	-	192.0.2.0/24	192.0.2.1	0	-		0	-

The first line's `Ipkts` counts the synthetic ARP requests our receive timer produces. It should rise by about one every second. If it does not, the `rx_interval_hz` setting is wrong, or the callout is not being started in `mynet_init`, or `running` is false.

28.10.3 Capturing With tcpdump

`tcpdump -i mynet0 -n` captures all traffic on our interface. You should see the synthetic ARP requests being generated every second, along with any traffic caused by your own ping attempts:

```
# tcpdump -i mynet0 -n
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on mynet0, link-type EN10MB (Ethernet), capture size 262144 bytes
14:30:12.000 02:a3:f1:22:bc:0d > ff:ff:ff:ff:ff:ff, ethertype ARP, Request who-has
  ↪ 192.0.2.99 tell 0.0.0.0, length 28
14:30:13.000 02:a3:f1:22:bc:0d > ff:ff:ff:ff:ff:ff, ethertype ARP, Request who-has
  ↪ 192.0.2.99 tell 0.0.0.0, length 28
...
```

The “link-type EN10MB (Ethernet)” confirms that BPF saw us as an Ethernet interface, which is the consequence of `ether_ifattach` having called `bpfattach(ifp, DLT_EN10MB, ETHER_HDR_LEN)` for us. Switch to `-v` or `-vv` to see fuller protocol decoding.

28.10.4 Generating Traffic With ping

Trigger outbound traffic by pinging an IP in the subnet we assigned:

```
# ping -c 3 192.0.2.99
PING 192.0.2.99 (192.0.2.99): 56 data bytes
--- 192.0.2.99 ping statistics ---
3 packets transmitted, 0 packets received, 100.0% packet loss
```

All three pings are lost, because our pseudo-driver simulates a wire with nothing at the other end. But the transmit counter moves:

```
# netstat -in -I mynet0
Name      Mtu Network      Address                Ipkts Ierrs ... Opkts Oerrs
mynet0 1500 <Link#12>    02:a3:f1:22:bc:0d      30    0      6    0
```

The 6 transmitted packets are three pings plus three ARP broadcast requests the stack issued trying to resolve 192.0.2.99. You can verify this with `tcpdump`.

28.10.5 `arp -an`

`arp -an` shows the system's ARP cache. Entries for 192.0.2.99 should appear as incomplete while the stack waits for an ARP reply that will never come. After a minute or so they expire.

28.10.6 `sysctl net.link` and `sysctl net.inet`

The network subsystems expose a wealth of per-interface and per-protocol sysctls. `sysctl net.link.ether` controls Ethernet-layer behaviour. `sysctl net.inet.ip` controls IP-layer behaviour. While none of these are specific to `mynet`, they are good to know. A common one when diagnosing pseudo-driver behaviour is `sysctl net.link.ether.inet.log_arp_wrong_iface=0`, which silences log messages about ARP traffic appearing on unexpected interfaces.

28.10.7 Monitoring Link Events With `ifstated` or `devd`

FreeBSD propagates link-state changes through the routing socket. You can observe this live with `route monitor`:

```
# route monitor
```

When you run `ifconfig mynet0 down` followed by `ifconfig mynet0 up`, `route monitor` prints `RTM_IFINFO` messages corresponding to the link-state changes we are announcing through `if_link_state_change`. That is the same mechanism `devd` uses for its notify events, and it is how scripts can react to link flaps.

28.10.8 Testing MTU Changes

```
# ifconfig mynet0 mtu 9000
# ifconfig mynet0
mynet0: ... mtu 9000
```

Change the MTU to something reasonable and watch `ifconfig` reflect the change. Try an out-of-range value and verify that the kernel rejects it:

```
# ifconfig mynet0 mtu 10
ifconfig: ioctl SIOCSIFMTU (set mtu): Invalid argument
```

That error comes from our SIOCSIFMTU handler returning EINVAL.

28.10.9 Testing Media Commands

```
# ifconfig mynet0 media 10baseT/UTP
ifconfig: requested media type not found
```

This fails because we did not register IFM_ETHER | IFM_10_T as an acceptable media type. Register it in mynet_create_unit and rebuild to see the command succeed.

```
# ifconfig mynet0 media 1000baseT
# ifconfig mynet0 | grep media
      media: Ethernet 1000baseT <full-duplex>
```

28.10.10 Comparing Against if_disc

Load if_disc alongside and compare:

```
# kldload if_disc
# ifconfig disc create
disc0
# ifconfig disc0 inet 192.0.2.50/24 up
```

disc0 is a simpler pseudo-driver. It ignores every outbound packet by dropping it in its discoutput function (not discoutput we wrote, but the one in if_disc.c). It has no receive path. Running tcpdump -i disc0 while pinging 192.0.2.50 shows outbound ICMP frames but no inbound ARP activity. Compare that with our mynet0, which still shows its synthetic ARP frames arriving once per second.

The contrast is useful because it shows how small the step is from “drop everything” to “simulate a full Ethernet interface”. We added a MAC address, a media descriptor, a callout, and a packet builder. Everything else, including the interface registration, the BPF hook, the flags, was already in the pattern.

28.10.11 Stress Test With iperf3

iperf3 can saturate a real Ethernet link. On our pseudo-driver it will not produce meaningful throughput numbers (the packets go nowhere), but it exercises if_transmit very hard:

```
# iperf3 -c 192.0.2.99 -t 10
Connecting to host 192.0.2.99, port 5201
iperf3: error - unable to connect to server: Connection refused
```

The connection fails because there is no server, but `netstat -in -I mynet0` will show `Opkts` climbing rapidly with the TCP retransmissions and ARP requests `iperf3` caused. Watch `vmstat 1` in another terminal and make sure the system load remains reasonable. If you see a lot of time spent in the driver, you may have a locking hot spot that is worth investigating.

28.10.12 Scripted Test Runs

You can wrap the above commands into a small shell script that exercises the driver in a known sequence. Here is a minimal example:

```
#!/bin/sh

set -e

echo "== load =="
kldload ./mynet.ko

echo "== create =="
ifconfig mynet create

echo "== configure =="
ifconfig mynet0 inet 192.0.2.1/24 up

echo "== traffic =="
(tcpdump -i mynet0 -nn -c 5 > /tmp/mynet-tcpdump.txt 2>&1) &
sleep 3
ping -c 2 192.0.2.99 || true
wait
cat /tmp/mynet-tcpdump.txt

echo "== counters =="
netstat -in -I mynet0

echo "== teardown =="
ifconfig mynet0 destroy
kldunload mynet
```

Save it under `examples/part-06/ch28-network-driver/lab05-bpf/run.sh`, mark it executable, and run it as root. It walks the driver through its entire lifecycle in under ten seconds. When something breaks later, a scripted baseline like this is priceless for spotting regressions.

28.10.13 What to Watch For

While testing, keep an eye on:

- `dmesg` output during load and unload, for unexpected warnings.
- `netstat -in -I mynet0` before and after operations, to confirm counters move in the expected direction.
- `kldstat` after unload, to confirm the module is gone.
- `ifconfig -a` after destroy, to confirm no orphan interface is left.
- `vmstat -m | grep mynet` to confirm memory is returned on unload.
- `vmstat -z | grep mbuf` across load test runs, to confirm mbuf counts stabilise.

A driver that is correct on a cold load can still leak on unload, or leak under load, or panic the kernel in a rare race. The tools listed above are the first line of defence against all those classes of bugs.

28.10.14 Deeper Observability with DTrace

FreeBSD's DTrace implementation is a formidable tool for driver observability, and once you know a few patterns you will reach for it often. The basic idea is that every function entry and exit in the kernel is a probe point, and every probe point can be instrumented from userland without modifying the code.

To count how often our transmit function is called:

```
# dtrace -n 'fbt::mynet_transmit:entry { @c = count(); }'
```

Run that in one terminal, generate traffic in another, and you will see the count climb. To observe each call with the packet length:

```
# dtrace -n 'fbt::mynet_transmit:entry { printf("len=%d", args[1]->m_pkthdr.len); }'
```

DTrace scripts can be much more elaborate. Here is one that counts transmitted packets grouped by source IP, if the interface is carrying IPv4 traffic:

```
# dtrace -n 'fbt::mynet_transmit:entry /args[1]->m_pkthdr.len > 34/ {  
    this->ip = (struct ip *) (mtod(args[1], struct ether_header *) + 1);  
    @src[this->ip->ip_src.s_addr] = count();  
}'
```

This kind of observability is difficult to add to a driver by hand, but DTrace gives it to you for free. Use it. When you cannot tell why a packet did or did not flow, DTrace probes on your own functions will almost always reveal the answer.

Some additional useful one-liners for network-driver work:

```
# dtrace -n 'fbt::if_input:entry { @ifs[stringof(args[0]->if_xname)] = count(); }'
```


This counts every call to `if_input` across the whole system, grouped by interface name. It is a quick way to verify that your receive path is reaching the stack.

```
# dtrace -n 'fbt::if_inc_counter:entry /args[1] == 1/ {
    @[stringof(args[0]->if_xname)] = count();
}'
```

This counts calls to `if_inc_counter` for `IFCOUNTER_IPACKETS` (which is value 1 in the enum) grouped by interface name. Compared with `netstat -in`, it lets you see the increments in real time.

Do not be afraid of DTrace. It looks intimidating at first because of the script-like syntax, but a driver debugging session with DTrace often takes minutes where the equivalent `printf`-debugging takes hours. Every minute you invest in learning DTrace idioms repays itself many times over.

28.10.15 Kernel Debugger Tips for Drivers

When a network driver panics or hangs, the kernel debugger (`ddb` or `kgdb`) is the tool of last resort. A few tips specific to driver work:

- After a panic, show `mbuf` (or show `pcpu`, show `alltrace`, show `lockchain` depending on what you are investigating) walks the mbuf allocations or the per-CPU data or the blocked-thread chains. Knowing which of these to invoke is a matter of practice.
- `show ifnet <pointer>` prints the contents of an `ifnet` structure given its address. Useful when a panic message says “`ifp = 0xffff...`”. The equivalent for a softc depends on the driver.
- `bt` prints a stack trace. Mostly you want `bt <tid>` where `<tid>` is the thread ID of interest.
- `continue` resumes execution, but after a real panic it is usually not safe. Collect information and then reboot.

For non-panic debugging, `kgdb /boot/kernel/kernel /var/crash/vmcore.0` lets you post-mortem a crash dump. Driver development on a lab VM with a crash dump partition is a comfortable workflow: panic, reboot, look at the dump at leisure.

28.10.16 `systat -if` for Live Counter Views

`systat -if 1` opens an ncurses view that refreshes every second and shows per-interface counter rates. It is a useful complement to `netstat -in` because you can watch traffic rise and fall in real time without reading the terminal log.

	/0	/1	/2	/3	/4	/5	/6	/7	/8	/9	/10
Load Average											
Interface	Traffic					Peak		Total			
mynet0	in	0.000 KB/s				0.041 KB/s		0.123 KB			
	out	0.000 KB/s				0.047 KB/s		0.167 KB			

The rates in this view are computed by `systat` from the counters we increment in `if_transmit` and in our receive path. If the rates do not match what you expect, the first suspicion should be that a counter is being

updated twice, or that it is being updated after `m_freem`, or that it is using `IFCOUNTER_OPACKETS` where it should use `IFCOUNTER_IPACKETS`. `systat -if` makes those mistakes very visible.

28.10.17 Wrapping Up Section 7

You now have a tested driver. It loads, configures, carries traffic both directions, reports its state to userland, cooperates with BPF, and reacts to link events. What remains is the final phase of the lifecycle: clean detachment, module unload, and some refactoring advice. That is Section 8.

28.11 Section 8: Cleanup, Detach, and Refactoring of the Network Driver

Every driver has a beginning and an end. The beginning is the pattern we have built up over the chapter: allocate, configure, register, run. The end is the symmetric teardown: quiesce, deregister, free. A driver that leaks a single byte on unload is not a correct driver, no matter how good it is during its active life. In this section we finalise the cleanup path, review the unload discipline, and offer refactoring advice so that the code stays maintainable as it grows.

28.11.1 The Full Teardown Sequence

Putting everything we have said together, the full teardown of a `mynet` interface looks like this:

```
static void
mynet_destroy(struct mynet_softc *sc)
{
    struct ifnet *ifp = sc->ifp;

    MYNET_LOCK(sc);
    sc->running = false;
    ifp->if_drv_flags &= ~IFF_DRV_RUNNING;
    MYNET_UNLOCK(sc);

    callout_drain(&sc->rx_callout);

    ether_ifdetach(ifp);
    if_free(ifp);

    ifmedia_removeall(&sc->media);
    mtx_destroy(&sc->mtx);
    free(sc, M_MYNET);
}
```

The order matters. Let us walk through it.

Step 1: mark not running. Setting `sc->running = false` and clearing `IFF_DRV_RUNNING` under the mutex means that any concurrent callout invocation sees the update and exits cleanly. This alone is not enough to stop running callouts, but it does stop new work from being scheduled.

Step 2: drain the callout. `callout_drain(&sc->rx_callout)` blocks the calling thread until any in-progress callout invocation has finished and no further invocation will occur. After `callout_drain` returns, it is safe to access the `softc` without worrying that the callout will fire again. This is the cleanest way to synchronise with a callout and is the pattern we recommend in every driver that uses them.

Step 3: detach the interface. `ether_ifdetach(ifp)` undoes what `ether_ifattach` did. It calls `if_detach`, which removes the interface from the global list, revokes its addresses, and invalidates any cached pointers. It also calls `bpfdetach` so that BPF releases its handle. After this call, the interface is no longer visible to userland or to the stack.

Step 4: free the ifnet. `if_free(ifp)` releases the memory. After this call, the `ifp` pointer is invalid and must not be used.

Step 5: clean up driver-private state. `ifmedia_removeall` frees the media entries we added. `mtx_destroy` tears down the mutex. `free` releases the `softc`.

Getting this sequence wrong in any way leads to subtle bugs. Freeing the `softc` before draining the callout produces use-after-free when the callout fires. Freeing the `ifnet` before detaching it produces cascading failures all over the stack. Destroying the mutex before draining the callout (which reacquires the mutex on entry) produces a classic “destroying locked mutex” panic. The discipline of “quiesce, detach, free” is what keeps the teardown clean.

28.11.2 The Cloner Destroy Path

Recall that we registered our cloner with `if_clone_simple`, passing `mynet_clone_create` and `mynet_clone_destroy`. The destroy function is called by the cloner framework when userland runs `ifconfig mynet0 destroy` or when the module is unloaded and the cloner is detached. Our implementation is a trivial wrapper:

```
static void
mynet_clone_destroy(struct ifnet *ifp)
{
    mynet_destroy((struct mynet_softc *)ifp->if_softc);
}
```

The cloner framework walks the list of interfaces it has created and calls the destroy function for each one. It does not do the draining or unlocking itself. That is the driver’s responsibility, and `mynet_destroy` does it correctly.

28.11.3 Module Unload

When `kldunload mynet` is invoked, the kernel calls the module event handler with `MOD_UNLOAD`. Our module handler does nothing interesting; the heavy lifting is done by the VNET sysunit we registered:

```
static void
vnet_mynet_uninit(const void *unused __unused)
{
    if_clone_detach(V_mynet_cloner);
}
```

`if_clone_detach` does two things. First, it destroys every interface that was created through the cloner by calling our `mynet_clone_destroy` for each one. Second, it unregisters the cloner itself so that no new interfaces can be created. After this call, every trace of our driver is gone from the kernel's state.

Try it:

```
# ifconfig mynet create
mynet0
# ifconfig mynet create
mynet1
# kldunload mynet
# ifconfig -a
```

`mynet0` and `mynet1` should be gone. No messages on the console, no stray counters, no leftover cloners. That is a successful unload.

28.11.4 Memory Accounting

`vmstat -m | grep mynet` shows the current allocation of our `M_MYNET` tag:

```
# vmstat -m | grep mynet
      Type InUse MemUse Requests  Size(s)
      mynet    0    0K         7  2048
```

`InUse 0` and `MemUse 0K` after unload confirm that we are not leaking. `Requests` counts the lifetime allocations. If you unload and reload several times, `Requests` climbs but `InUse` returns to zero each time. If `InUse` ever stays above zero after unload, you have a leak.

28.11.5 Dealing With Stuck Callouts

Occasionally during development you will tweak the driver and end up with a callout that does not drain cleanly. The symptom is that `kldunload` hangs, or the system panics with a message about a locked mutex. The root cause is almost always one of:

- The callout handler reacquires the mutex but does not reschedule itself, and `callout_drain` is called before the last scheduled firing completes.
- The callout handler is stuck waiting on a lock that another thread holds.
- The callout itself was never properly stopped before drain.

The first line of defence is `callout_init_mtx` with the softc mutex: this sets up an automatic acquisition pattern that makes drain correct by construction. The second line is to use `callout_stop` or `callout_drain` consistently and to avoid mixing the two on the same callout.

If the unload hangs, use `ps -auxw` to find the offending thread, and `kgdb` on a running kernel (through `/dev/mem` and `bin/kgdb /boot/kernel/kernel`) to see what it is stuck on. The stuck frame is almost always in the callout code, and the fix is almost always to drain before destroying the mutex.

28.11.6 VNET Considerations

FreeBSD's network stack supports VNETs, virtual network stacks associated with a jail or a VNET instance. A driver can be VNET-aware if it wants to allow per-VNET creation of interfaces, or it can be non-VNET-aware if one set of interfaces per system is enough.

We used `VNET_DEFINE_STATIC` and `VNET_SYSINIT/VNET_SYSUNINIT` in our cloner registration. That choice makes our driver implicitly VNET-aware: each VNET gets its own cloner, and `mynet` interfaces can be created in any VNET. For a small pseudo-driver this costs us nothing and gains us flexibility.

The deeper aspects of VNET, including moving an interface between VNETs with `if_vmove` and handling VNET teardown, are beyond the scope of this chapter and will be covered later in the book, in Chapter 30. For now it is enough to know that our driver follows the conventions that make it VNET-compatible.

28.11.7 Refactoring Advice

The driver we have built is a single C file with about 500 lines of code. That is comfortable for a teaching example. In a production driver with more features, the file would grow, and you would want to split it up. Here are the splits that almost every driver eventually makes.

Separate the ifnet glue from the data path. The `ifnet` registration, cloner logic, and `ioctl` handling tend to be stable over time. The data path, transmit and receive, evolves as hardware features change. Splitting them into `mynet_if.c` and `mynet_data.c` keeps most files small and focused.

Isolate the backend. In a real NIC driver, the backend is hardware-specific code: register access, DMA, MSI-X, ring buffers. In a pseudo-driver, the backend is the simulation. Either way, putting the backend in `mynet_backend.c` with a clean interface makes it possible to replace the backend without touching the `ifnet` code.

Separate sysctl and debugging. As your driver grows, you will add `sysctls` for diagnostic controls, counters for debugging, and maybe DTrace SDT probes. These tend to accumulate in messy ways. Keeping them in `mynet_sysctl.c` keeps the main files readable.

Keep the header public. A `mynet_var.h` or `mynet.h` header that declares the softc and the cross-file prototypes is the glue that keeps the split compiling. Treat that header as a mini public API.

Version the driver. `MODULE_VERSION(mynet, 1)` is the bare minimum. When you add a significant feature, increment the version. Downstream consumers who depend on your module can then require a minimum version, and kernel users can tell which version of the driver they have loaded via `kldstat -v`.

28.11.8 Feature Flags and Capabilities

Ethernet drivers advertise capabilities through `if_capabilities` and `if_capenable`. We set `IFCAP_VLAN_MTU`. Other capabilities a real driver might advertise include:

- `IFCAP_HWCSUM`: hardware checksum offload.
- `IFCAP_TS04`, `IFCAP_TS06`: TCP segmentation offload for IPv4 and IPv6.
- `IFCAP_LRO`: large receive offload.
- `IFCAP_VLAN_HWTAGGING`: hardware VLAN tagging.
- `IFCAP_RXCSUM`, `IFCAP_TXCSUM`: receive and transmit checksum offload.
- `IFCAP_JUMBO_MTU`: jumbo frames support.
- `IFCAP_LINKSTATE`: hardware link-state events.
- `IFCAP_NETMAP`: netmap(4) support for high-speed packet I/O.

For a pseudo-driver most of these are not relevant. Advertising them falsely causes problems because the stack will then attempt to use them and expect them to work. Keep the capability set honest: advertise only what your driver actually supports.

28.11.9 Writing a Run Script

One of the most useful artefacts to produce alongside a driver is a small shell script that exercises its entire lifecycle. The skeleton we showed in Section 7 is already 80% of that script. Extend it with:

- Consistency checks after each operation (`ifconfig -a | grep mynet0` or `netstat -in -I mynet0 | ...`).
- Optional logging of each step to a file for after-the-fact inspection.
- A cleanup block at the end that ensures the system is left in a known state even if an earlier step fails.

A good run script is the single most valuable tool for regression-free development. We encourage you to maintain one as you extend the driver in the challenges.

28.11.10 Tidying Up the File

Finally, a word on code style. Real FreeBSD drivers follow KNF (Kernel Normal Form), the coding style documented in `style(9)`. Summarised: tabs for indentation, braces on the same line as function definitions but on the next line for structures and enums, 80-column lines where possible, no spaces before the opening parenthesis of a function call, and so on. Your driver will be easier to merge upstream (and easier to read a year from now) if you follow KNF consistently.

28.11.11 Handling Partial Initialisation Failure

We have focused on the happy path. What happens if `mynet_create_unit` fails partway through? Say `if_alloc` succeeds, `mtx_init` runs, `ifmedia_init` sets up the media, and then `malloc` of some auxiliary buffer returns `NULL`. We need to roll back cleanly, because the user just got `ifconfig mynet create` to fail, and we must leave no trace.

The idiom for rollback is a block of labels near the end of the function, each label undoing one step of the initialisation:

```
static int
mynet_create_unit(int unit)
{
    struct mynet_softc *sc;
    struct ifnet *ifp;
    int error = 0;

    sc = malloc(sizeof(*sc), M_MYNET, M_WAITOK | M_ZERO);
    ifp = if_alloc(IFT_ETHER);
    if (ifp == NULL) {
        error = ENOSPC;
        goto fail_alloc;
    }

    sc->ifp = ifp;
    mtx_init(&sc->mtx, "mynet", NULL, MTX_DEF);
    /* ... other setup ... */

    ether_ifattach(ifp, sc->hwaddr);
    return (0);

fail_alloc:
    free(sc, M_MYNET);
    return (error);
}
```

This pattern, common in kernel code, makes rollback boring. Each label takes responsibility for the step immediately above it. The overall shape is “if something in step N fails, jump to label N-1 and unwind from there”.

For our driver the only realistic failure point early in create is `if_alloc`. If that succeeds, the rest of the setup (mutex init, media init, `ether_ifattach`) is either infallible or sufficiently idempotent that no rollback is needed. But the shape of the rollback matters, because a more complex driver will have more failure points, and the same pattern scales cleanly.

28.11.12 Synchronising With In-Flight Callbacks

Beyond callouts, other asynchronous code might be in flight when we tear down an interface. Taskqueue tasks, interrupt handlers, and timer-based rearm functions all need to be stopped before memory is freed.

The kernel provides `taskqueue_drain(tq, task)` for taskqueue tasks, analogous to `callout_drain` for callouts. For interrupts, `bus_teardown_intr` and `bus_release_resource` ensure the interrupt handler will

not be invoked again. For rearmable callouts where the handler reschedules itself, `callout_drain` still does the right thing: it waits for the current invocation to finish and prevents further rearms.

A general rule for a teardown path:

1. Clear any “running” or “armed” flags that the asynchronous code checks.
2. Drain each asynchronous source in turn (taskqueue, callout, interrupt).
3. Detach from upper layers (`ether_ifdetach`).
4. Free the memory.

Skipping step 1 is usually the cause of “destroying locked mutex” panics, because the asynchronous code is still running when the mutex gets destroyed. Skipping step 2 is the cause of use-after-free. Step 3 and step 4 must happen in that order or the stack may try to call into our callbacks after they have been freed.

28.11.13 A Worked Error Scenario

To make the above concrete, imagine a subtle bug. Suppose that during development we call `mtx_destroy` before `callout_drain`. The callout is scheduled, the user runs `ifconfig mynet0 destroy`, our destroy function destroys the mutex, and then the scheduled callout fires. The callout tries to acquire the mutex (because we registered it with `callout_init_mtx`), sees a destroyed mutex, and triggers an assertion: “acquiring a destroyed mutex”. The system panics with a stack trace pointing into the callout code.

The fix is to reverse the order: `callout_drain` first, `mtx_destroy` later. The general principle is that synchronisation primitives are destroyed last, after all consumers are known to have stopped.

This kind of bug is easy to introduce and hard to diagnose if you have not seen it before. Having an explicit mental model of “quiesce, detach, free” prevents it.

28.11.14 Wrapping Up Section 8

The full lifecycle is now in your hands. Load, cloner registration, per-interface creation, active life with transmit, receive, ioctl, and link events, per-interface destruction, cloner detach, module unload. You can build, test, tear down, and rebuild with confidence that the kernel is returned to a clean state.

The sections that come next are the hands-on part of the chapter: labs that walk you through the milestones we have described, challenges that extend the driver, troubleshooting pointers, and a wrap-up.

28.12 Hands-On Labs

The labs below are ordered to mirror the chapter’s flow. Each one builds on the previous, so do them in sequence. The companion files live under `examples/part-06/ch28-network-driver/`, and each lab has its own README with the specific commands.

Before starting, make sure you are on a FreeBSD 14.3 lab VM with root access, a clean workspace directory where you can build a kernel module, and a freshly-snapshot state you can return to if something goes wrong. A snapshot before beginning the labs is a small investment that pays for itself the first time you need it.

Each lab ends with a brief “checkpoint” block listing the specific observations you should record in your logbook. If your logbook already has those observations, you can move on. If it does not, return to the previous step

and redo it. The cumulative structure of the labs means that a missed observation in Lab 2 will make Lab 4 confusing.

28.12.1 Lab 1: Build and Load the Skeleton

Goal. Build the skeleton driver from Section 3, load it, create an instance, and observe the default state.

Steps.

1. `cd examples/part-06/ch28-network-driver/`
2. `make` and watch for warnings. The build should produce `mynet.ko` without warnings.
3. `kldload ./mynet.ko`. No messages should appear on the console; `kldstat` should list `mynet` as present.
4. `ifconfig mynet create` should print `mynet0`.
5. `ifconfig mynet0` and record the output in your logbook. Note especially the flags, the MAC address, the media line, and the status.
6. `kldstat -v | grep mynet` and verify the module is present and loaded at the expected address.
7. `sysctl net.generic.ifclone` and confirm that `mynet` appears in the list of cloners.
8. `ifconfig mynet0 destroy`. The interface should disappear.
9. `kldunload mynet`. The module should unload cleanly.
10. `kldstat` and `ifconfig -a` to confirm nothing is left.

What to watch for. The `ifconfig mynet0` output should show flags `BROADCAST`, `SIMPLEX`, `MULTICAST`, a MAC address, a media line of “Ethernet autoselect”, and a status of “no carrier”. If any of these is missing, recheck the `mynet_create_unit` function and the `ifmedia_init` call.

Logbook checkpoint.

- Record the exact MAC address assigned to `mynet0`.
- Record the initial value of `if_mtu`.
- Note the flags reported before and after `ifconfig mynet0 up`.
- Note whether status: changes between “no carrier” and “active”.

If things go wrong. The most common Lab 1 failure is a build error caused by a missing header. Make sure your kernel source tree under `/usr/src/sys/` matches the running kernel version. If `kldload` fails with “module already present”, unload any prior instance with `kldunload mynet` and try again. If `ifconfig mynet create` returns “Operation not supported”, the cloner did not register, and you need to re-check the `VNET_SYSINIT` call.

28.12.2 Lab 2: Exercise the Transmit Path

Goal. Verify that `if_transmit` is called when traffic leaves the interface.

Steps.

1. Create the interface and bring it up as in Lab 1.
2. `ifconfig mynet0 inet 192.0.2.1/24 up`. Both `UP` and `RUNNING` flags should now appear.
3. In one terminal, run `tcpdump -i mynet0 -nn`.
4. In another, run `ping -c 3 192.0.2.99`.

5. Observe the ARP and ICMP traffic printed by `tcpdump`.
6. `netstat -in -I mynet0` and record the counters. The `Opkts` column should show at least four (three ICMP requests plus the ARP broadcast attempts).
7. Modify the transmit function to return `ENOBUFS` for every call and rebuild.
8. Unload and reload, repeat the ping, observe that `Opkts` stops climbing and that `Oerrors` increases instead.
9. Revert the modification and rebuild.
10. Optional: run the DTrace one-liner `dtrace -n 'fbt::mynet_transmit:entry { @c = count(); }'` while generating traffic to confirm each call reaches your transmit function.

What to watch for. In step 5, each ping produces one ARP broadcast (because the stack does not know the MAC for 192.0.2.99) and one ICMP echo request per ping attempt, but the ARP reply never comes, so subsequent pings only add ICMP requests. Understanding why that is, and what it looks like in `tcpdump`, is an important part of this lab.

Logbook checkpoint.

- Record the exact `Opkts` count after three pings.
- Record the `Obytes` count and verify that it matches the expected sum of ARP frame (42 bytes) plus three ICMP frames.
- Note what changes in `Oerrors` when you deliberately return `ENOBUFS`.

If things go wrong. If `Opkts` is zero after the pings, your `if_transmit` callback is not being called. Check that `ifp->if_transmit = mynet_transmit` is set during create. If `Obytes` is growing but `Opkts` is not, one of the counter calls is missing or reaching the wrong counter. If `tcpdump` shows no outbound traffic at all, the BPF tap in `transmit` is missing; add `BPF_MTAP(ifp, m)` before the free.

28.12.3 Lab 3: Exercise the Receive Path

Goal. Verify that `if_input` delivers packets into the stack.

Steps.

1. Create the interface and bring it up.
2. `tcpdump -i mynet0 -nn`.
3. Wait five seconds and confirm that one synthesised ARP request per second appears.
4. `netstat -in -I mynet0` and confirm `Ipkts` matches the packet count.
5. Change `sc->rx_interval_hz = hz / 10;` and rebuild.
6. Unload, reload, re-create. Observe that the rate becomes ten packets per second.
7. Revert to one packet per second.
8. Optional: comment out the `BPF_MTAP` call in the receive path, rebuild, and observe that `tcpdump` no longer shows the synthesised ARP but `Ipkts` still increments. This confirms that BPF visibility and counter updates are independent.
9. Optional: comment out the `if_input` call (leave `BPF_MTAP` in place), rebuild, and observe the opposite behaviour: `tcpdump` sees the frame, but `Ipkts` does not move because the stack never actually got the frame.

What to watch for. The `Ipkts` counter should increment exactly once per synthesised frame. If it does not, the BPF tap may be seeing the frame but `if_input` is not being called, or the calls are racing with teardown.

Logbook checkpoint.

- Record the interval between consecutive synthesised ARPs as shown by tcpdump timestamps.
- Record the MAC addresses in the ARP frame and confirm that the source MAC matches the interface's address.
- Observe what `arp -an` shows before and after; entries for `192.0.2.99` should remain incomplete.

If things go wrong. If no synthesised ARP appears in tcpdump, the callout is not firing. Check that `callout_reset` is called in `mynet_init` and that `sc->running` is true at the time. If tcpdump shows the ARP but `Ipkts` is zero, the counter is not being updated (or is being updated after `if_input` which has already freed the mbuf).

28.12.4 Lab 4: Media and Link State

Goal. Observe the difference between link state, media, and interface flags.

Steps.

1. Create and configure the interface.
2. `ifconfig mynet0` and note the status and media lines.
3. `ifconfig mynet0 down`.
4. `ifconfig mynet0` and note that status changes.
5. `ifconfig mynet0 up`.
6. In another terminal, `route monitor` and repeat steps 3 and 5 while watching the output.
7. `ifconfig mynet0 media 1000baseT mediaopt full-duplex` and confirm `ifconfig mynet0` reflects the change.
8. Add a third media entry `IFM_ETHER | IFM_100_TX | IFM_FDX` to `mynet_create_unit`, rebuild, and verify that `ifconfig mynet0 media 100baseTX mediaopt full-duplex` now succeeds.
9. Remove the entry and rebuild. Verify that the same command now fails with “requested media type not found”.

What to watch for. `route monitor` prints `RTM_IFINFO` messages on every link-state transition. The `ifconfig mynet0 status:` line shows `active` when the driver is running and the link is up, and `no carrier` when the driver has called `LINK_STATE_DOWN`.

Logbook checkpoint.

- Record the exact `RTM_IFINFO` message text from `route monitor`.
- Note the difference between `IFF_UP` and `LINK_STATE_UP` by capturing the output of `ifconfig mynet0` in each of the four possible combinations (up or down cross with link up or down).
- Observe whether `status:` and the interface flags stay consistent across all four states.

If things go wrong. If `status:` stays at “no carrier” even after the interface is up, you are not calling `if_link_state_change(ifp, LINK_STATE_UP)` from `mynet_init`. If `ifconfig mynet0 media 1000baseT` fails with “requested media type not found”, you did not register `IFM_ETHER | IFM_1000_T` via `ifmedia_add`, or you registered it with the wrong flags.

28.12.5 Lab 5: tcpdump and BPF

Goal. Confirm that BPF sees both outbound and inbound packets.

Steps.

1. Create and configure the interface with IP 192.0.2.1/24.
2. `tcpdump -i mynet0 -nn > /tmp/dump.txt &`
3. Wait ten seconds.
4. `ping -c 3 192.0.2.99.`
5. Wait another ten seconds.
6. `kill %1.`
7. `cat /tmp/dump.txt` and identify the synthesised ARP requests, the ARP broadcasts generated by your ping, and the ICMP echo requests.
8. Remove the BPF_MTAP call from `mynet_transmit` and rebuild. Repeat. Note that outbound ICMP no longer appears in the `tcpdump` output.
9. Restore the BPF_MTAP call.
10. Experiment with filters: `tcpdump -i mynet0 -nn 'arp'` should show only the synthesised ARPs and ARP from your pings, while `tcpdump -i mynet0 -nn 'icmp'` should show only the ICMP echo requests.
11. Observe the link-type line in `tcpdump`'s startup output. It should say EN10MB (Ethernet), because `ether_ifattach` set that up for us. If it says NULL, the interface was attached without Ethernet semantics.

What to watch for. The exercise demonstrates that BPF visibility is not automatic for every packet. It is the driver's job to tap on both the transmit and receive paths.

Logbook checkpoint.

- Record one complete line of `tcpdump` output for each type of frame you observed: synthesised ARP, outbound ARP, outbound ICMP echo request.
- Record the link-type line as printed by `tcpdump`.
- Note what happens to the output when you remove BPF_MTAP from `transmit`.

If things go wrong. If `tcpdump` never shows any packets at all, `bpfattach` was not called (usually because you forgot `ether_ifattach`). If it shows received packets but not transmitted ones, your transmit tap is missing. If it shows transmitted packets but not received ones, your receive tap is missing. If the link-type is wrong, the interface type or the `bpfattach` call is wrong.

28.12.6 Lab 6: Clean Detach

Goal. Verify that `unload` returns the system to a clean state.

Steps.

1. Create three interfaces: `mynet` create three times.
2. Configure each with a different IP in 192.0.2.0/24 (for example, 192.0.2.1/24, 192.0.2.2/24, 192.0.2.3/24).
3. `vmstat -m | grep mynet` and record the allocation count.
4. `kldunload mynet` (do not destroy first).

5. `ifconfig -a` and confirm none of `mynet0`, `mynet1`, `mynet2` remains.
6. `vmstat -m | grep mynet` and confirm `InUse` returns to zero.
7. Repeat steps 1 through 6 five times in sequence. Each round should leave `InUse` at zero and should not leave any orphaned state.
8. Optional: introduce an artificial bug by removing the `callout_drain` call from `mynet_destroy`. Re-build, load, create an interface, and unload. Observe what happens (it is usually a panic, and it is a dramatic way to learn why `callout_drain` exists).
9. Restore the `callout_drain` call.

What to watch for. The cloner detach path should iterate over all three interfaces, call `mynet_clone_destroy` on each, and free all memory. If any interface remains, or if `InUse` is non-zero, something in the teardown is broken.

Logbook checkpoint.

- Record the `InUse` values before and after each round of load-create-unload.
- Note the `Requests` column in `vmstat -m | grep mynet`; it should climb monotonically because it records lifetime allocations.
- Record any unexpected messages in `dmesg`.

If things go wrong. If `kldunload` hangs, a callout or a taskqueue task is still running. Use `ps -auxw` to find the kernel thread and `procstat -k <pid>` to see its backtrace. If `InUse` stays above zero after unload, you have a memory leak; the usual suspect is that `mynet_destroy` is not being called on one of the interfaces, which means `if_clone_detach` did not find it.

28.12.7 Lab 7: Reading the Real Tree

Goal. Connect what you have built to what lives in `/usr/src/sys/net/`.

Steps.

1. Open `/usr/src/sys/net/if_disc.c` side by side with your `mynet.c`. For each of the following, locate the corresponding code in both files:
 - Cloner registration.
 - Softc allocation.
 - Interface type (`IFT_LOOP` vs `IFT_ETHER`).
 - BPF attach.
 - Transmit path.
 - Ioctl handling.
 - Cloner destroy.
2. Open `/usr/src/sys/net/if_epair.c` and do the same exercise. Note the use of `if_clone_advanced`, the pairing logic, and the use of `ifmedia_init`.
3. Open `/usr/src/sys/net/if_ethersubr.c` and locate `ether_ifattach`. Trace through it line by line, and cross-reference each line with what we said it does in Section 3.
4. Open `/usr/src/sys/net/bpf.c` and locate `bpf_mtap_if`, which is the function `BPF_MTAP` expands to. Note the fast-path check for active peers.

What to watch for. The goal of this lab is recognition, not comprehension. You do not need to understand every line of `epair(4)` or `ether_ifattach`. You just need to see that the same patterns we used in our driver appear in the real tree, and that the new code you might encounter elsewhere is a variation on themes you already know.

Logbook checkpoint.

- Record one function name from each of `if_disc.c`, `if_epair.c`, and `if_ethersubr.c` that you now understand well enough to explain aloud.
- Note any pattern in these files that surprised you or that contradicted an assumption you had built from our chapter.

28.13 Challenge Exercises

The challenges below extend the driver in small, self-contained directions. Each one is meant to be doable in one or two focused sessions and relies only on what the chapter has taught.

28.13.1 Challenge 1: Shared Queue Between Paired Interfaces

Brief. Modify `mynet` so that creating two paired interfaces (`mynet0a` and `mynet0b`) behaves like `epair(4)`: transmitting on one interface causes the frame to appear on the other.

Hints. Use `if_clone_advanced` with a match function, as `epair.c` does. Share a queue between the two `softc` structures. Use a callout or a taskqueue to dequeue on the other side and call `if_input`.

Expected result. When you ping an IP assigned to `mynet0a` from an IP assigned to `mynet0b`, the replies should actually come back. You have built a software simulation of two cables plugged into each other.

Key design questions. Where do you store the shared queue? How do you ensure that a packet sent on one side cannot be seen by the original sender (the `IFF_SIMPLEX` contract)? How do you handle the case where only one side of the pair is up?

Suggested structure. Add a struct `mynet_pair` that owns two `softcs`, and have each `softc` carry a pointer to the pair. The transmit function on side A enqueues the mbuf on side B's input queue and schedules a taskqueue. The taskqueue dequeues and calls `if_input` on side B. Use a mutex in the pair structure to protect the queue.

28.13.2 Challenge 2: Link Flap Simulation

Brief. Add a sysctl `net.mynet.flap_interval` that, when non-zero, causes the driver to flap the link up and down every `flap_interval` seconds.

Hints. Use a callout that calls `if_link_state_change` alternately with `LINK_STATE_UP` and `LINK_STATE_DOWN`. Observe the effect on `route monitor`.

Expected result. While flapping is enabled, `ifconfig mynet0` should alternate between `status: active` and `status: no carrier` at the chosen interval. `route monitor` should print `RTM_IFINFO` messages at each transition.

Extension. Make the flap interval per-interface rather than global. You can do this by creating a sysctl node per interface under `net.mynet.<ifname>`, which requires using `sysctl_add_oid` and similar dynamic-sysctl APIs.

28.13.3 Challenge 3: Error Injection

Brief. Add a sysctl `net.mynet.drop_rate` that sets a percentage of outbound frames to be dropped with an error.

Hints. In `mynet_transmit`, roll a random number via `arc4random`. If it falls below the configured percentage, increment `IFCOUNTER_OERRORS`, free the mbuf, and return. Otherwise continue as before.

Expected result. With `drop_rate` set to 50, ping should show roughly 50% packet loss instead of 100%. (Remember, the “100% loss” without `drop_rate` came from no reply ever returning, not from a transmit drop. So with `drop_rate=50` you still get 100% ping loss; but if you combine this challenge with Challenge 1’s pair, the combined behaviour should be 50% ping loss.)

Extension. Add a separate `rx_drop_rate` that drops synthesised receive frames. Observe how the receive-counter output differs between transmit and receive drops.

28.13.4 Challenge 4: iperf3 Stress

Brief. Use `iperf3` to stress the transmit path and measure how quickly the driver can process frames.

Hints. Run `iperf3 -c 192.0.2.99 -t 10 -u -b 1G` to generate a UDP flood. Observe `netstat -in -I mynet0` before and after. Observe `vmstat 1` for system load. Consider what you would need to change in the driver to support higher rates: per-CPU counters, lock-free transmit paths, taskqueue-based deferred processing.

Expected result. The `iperf3` run will not produce meaningful bandwidth numbers (because there is no server to acknowledge anything), but it will push `0pkts` up rapidly. Watch for any CPU hotspot on the transmit path. If you have combined this with Challenge 1, the paired interface setup should show packets crossing the simulated link.

Measurement tips. Use `pmcstat` or `dtrace` to profile where time is spent. The transmit path is a reasonable place to look for lock contention. If you see a high rate of `mtx_lock` on the `softc` mutex in `mynet_transmit`, that is a sign that you are contending on a lock that real drivers would split per-queue.

28.13.5 Challenge 5: Per-Interface sysctl Tree

Brief. Expose per-interface runtime controls and statistics under `net.mynet.mynet0.*`.

Hints. Use `sysctl_add_oid` to dynamically add per-interface sysctls when the interface is created, and remove them when the interface is destroyed. A common pattern is to create a per-instance context under a static root node, and to attach child leaves for the specific controls and stats.

Expected result. `sysctl net.mynet.mynet0.rx_interval_hz` should read and write the receive interval, overriding the compile-time default. `sysctl net.mynet.mynet0.rx_packets_generated` should read a counter that increments every time the synthetic receive timer fires.

Extension. Add an `rx_enabled` sysctl that pauses and resumes the synthetic receive timer. Verify the behaviour by watching `tcpdump` while toggling the sysctl.

28.13.6 Challenge 6: Netgraph Node

Brief. Expose `mynet` as a netgraph node so that it can be wired into the netgraph framework.

Hints. This is a longer challenge because it requires familiarity with `netgraph(4)`. Read `/usr/src/sys/net-graph/ng_ether.c` for a reference example of an interface exposed as a netgraph node. Add a single hook that provides packet interception before or after our `if_transmit` and `if_input`.

Expected result. After the netgraph node is present, you should be able to use `ngctl` to attach a filter or a redirection node and watch packets flow through the netgraph chain.

This challenge is the most open-ended of the set. If you reach a working skeleton, you have essentially completed the path from “hello world” driver to a driver that participates fully in FreeBSD’s advanced networking infrastructure.

28.14 Troubleshooting and Common Mistakes

Network drivers fail in a handful of characteristic ways. Learn to spot them and you save hours of debugging.

28.14.1 Symptom: `ifconfig mynet create` Returns “Operation not supported”

Likely cause. The cloner is not registered, or the cloner name does not match. Check that `v_mynet_cloner` is initialised in `vnet_mynet_init`, and that the `mynet_name` string is the one the user is typing.

Diagnostic. `sysctl net.generic.ifclone` lists all registered cloners. If `mynet` is absent, the registration did not happen.

28.14.2 Symptom: `ifconfig mynet0 up` Hangs or Panics

Likely cause. The `mynet_init` function is doing something that sleeps while holding the `softc` mutex, or is calling up into the stack with the mutex held.

Diagnostic. If the system hangs, enter the debugger (`Ctrl-Alt-Esc` in a console) and type `ps` to see which thread is stuck, then `trace TID` to get a backtrace. Look for the offending lock acquisition.

28.14.3 Symptom: `tcpdump -i mynet0` Sees No Packets

Likely cause. `BPF_MTAP` is not being called, or `bpfattach` was not called during interface setup.

Diagnostic. `bpf_peers_present(ifp->if_bpf)` should return true when `tcpdump` is running. If it does not, check that `ether_ifattach` was called. If `ether_ifattach` was called but `BPF_MTAP` is not in the data path, add the call in both `transmit` and `receive`.

28.14.4 Symptom: ping Shows 100% Loss (Expected) but Opkts Stays Zero

Likely cause. `if_transmit` is not being called, or it is returning early without incrementing counters.

Diagnostic. `dtrace -n 'fbt::mynet_transmit:entry { @[probefunc] = count(); }'` counts how often the function is called. If it is zero, the stack is not dispatching to us, and the assignment to `ifp->if_transmit` (or, if you switched to the helper, the `if_settransmitfn` call) during setup is suspect.

28.14.5 Symptom: kldunload Panics with “destroying locked mutex”

Likely cause. The mutex is being destroyed while another thread (typically a callout) still holds it.

Diagnostic. Audit the teardown order. `callout_drain` must be called before `mtx_destroy`. `ether_ifdetach` must be called before `if_free`. If the callout locks the softc mutex, `callout_drain` must happen before that mutex goes away.

28.14.6 Symptom: netstat -in -I mynet0 Shows Opkts Higher Than Opkts in systat -if

Likely cause. One of the counters is being incremented twice in the transmit path.

Diagnostic. Inspect the code paths. A common mistake is to increment `IFCOUNTER_OPACKETS` both in the driver and in a helper function.

28.14.7 Symptom: Module Loads But ifconfig mynet create Produces a Kernel Warning

Likely cause. A field of the `ifnet` is not properly initialised, or `ether_ifattach` is called without a valid MAC address.

Diagnostic. Run `dmesg` after the warning. The kernel usually prints enough context to identify the offending field.

28.14.8 Symptom: kldunload Returns But ifconfig -a Still Shows mynet0

Likely cause. The cloner detach did not iterate all interfaces. This is usually a sign that the interface was created outside the cloner path, or that the `if_clone` data structures are out of sync.

Diagnostic. `sysctl net.generic.ifclone` after the unload should not list `mynet`. If it does, `if_clone_detach` did not complete.

28.14.9 Symptom: Intermittent Panics Under iperf3 Load

Likely cause. A race between the transmit path and the `ioctl` path, typically one not locking when the other does.

Diagnostic. Run the kernel with `INVARIANTS` and `WITNESS` enabled. These options add lock-order and assertion checks that catch most races immediately. They are the single best development tool for network drivers.

28.14.10 Symptom: `ifconfig mynet0 mtu 9000` Succeeds But Jumbo Frames Fail

Likely cause. The driver advertises an MTU range it cannot actually transport. Our reference driver uses a wide range for simplicity, but a real driver has a hard upper limit dictated by hardware.

Diagnostic. Send a frame larger than the configured MTU and observe that `IFCOUNTER_OERRORS` increments. Align the advertised upper bound with the actual capability.

28.14.11 Symptom: `dmesg` Shows “acquiring a destroyed mutex”

Likely cause. A callout, taskqueue task, or interrupt handler is acquiring a mutex after `mtx_destroy` has been called. Almost always caused by incorrect teardown ordering.

Diagnostic. Trace through your `mynet_destroy`. `callout_drain` and equivalent drain operations must happen before `mtx_destroy`. The correct order is “quiesce, detach, destroy”, not “destroy, quiesce”.

28.14.12 Symptom: `WITNESS` Reports a Lock-Order Reversal

Likely cause. Two threads acquire the same pair of locks in opposite orders. In a network driver this most commonly happens between the softc mutex and a stack-internal lock like the ARP table lock or the routing table lock.

Diagnostic. Read the `WITNESS` output carefully; it shows both backtraces. The fix is usually to release the driver mutex before calling into the stack (for example, before `if_input` or `if_link_state_change`), which we recommend throughout this chapter.

28.14.13 Symptom: Packet Loss Under Moderate Load

Likely cause. Either mbuf exhaustion (check `vmstat -z | grep mbuf`) or a transmit queue that has no backpressure and silently drops.

Diagnostic. `vmstat -z | grep mbuf` before and after the load. If mbuf or `mbuf_cluster` allocations are climbing but not being returned, you have a mbuf leak. If they are being returned but your driver’s internal queue is dropping, you need to either enlarge the queue or implement backpressure.

28.14.14 Symptom: `ifconfig mynet0 inet6 2001:db8::1/64` Has No Effect

Likely cause. IPv6 is not compiled into your kernel, or the interface does not advertise `IFF_MULTICAST` (which IPv6 requires).

Diagnostic. `sysctl net.inet6.ip6.v6only` and similar tell you whether IPv6 is present. `ifconfig mynet0` shows the flags; ensure `MULTICAST` is one of them.

28.14.15 Symptom: Module Loads But `ifconfig mynet create` Produces No Interface and No Error

Likely cause. The cloner’s create function is returning success but never actually allocating an interface. Easy to cause by returning 0 before calling `if_alloc`.

Diagnostic. Add a `printf("mynet_clone_create called\n")` at the start of your create callback. If the message appears but no interface is created, the bug is between the `printf` and the `if_attach` call.

28.14.16 Symptom: `sysctl net.link.generic` Returns Unexpected Results

Likely cause. The driver has corrupted a field of `ifnet` that the generic `sysctl` handler reads. This is rare but indicative of deeper bugs.

Diagnostic. Run the kernel with `INVARIENTS` and look for assertion failures. The offending write is usually near where `ifnet` fields are being initialised.

28.15 Quick Reference Tables

The tables below summarise the most frequently-used APIs and constants introduced in this chapter. Keep the page open as you work through the labs.

28.15.1 Lifecycle Functions

Function	Purpose
<code>if_alloc(type)</code>	Allocate a new <code>ifnet</code> of the given <code>IFT_</code> type.
<code>if_free(ifp)</code>	Free an <code>ifnet</code> after detach.
<code>if_attach(ifp)</code>	Register the interface with the stack.
<code>if_detach(ifp)</code>	Unregister the interface.
<code>ether_ifattach(ifp, mac)</code>	Register an Ethernet-like interface. Wraps <code>if_attach</code> plus <code>bpfattach</code> and sets Ethernet defaults.
<code>ether_ifdetach(ifp)</code>	Undo <code>ether_ifattach</code> .
<code>if_initname(ifp, family, unit)</code>	Set the interface name.
<code>bpfattach(ifp, dlt, hdrlen)</code>	Register with BPF manually. Done automatically by <code>ether_ifattach</code> .
<code>bpfdetach(ifp)</code>	Unregister from BPF. Done automatically by <code>ether_ifdetach</code> .
<code>if_clone_simple(name, create, destroy, minifs)</code>	Register a simple cloner.
<code>if_clone_advanced(name, minifs, match, create, destroy)</code>	Register a cloner with a custom match function.
<code>if_clone_detach(cloner)</code>	Tear down a cloner and all its interfaces.
<code>callout_init_mtx(co, mtx, flags)</code>	Initialise a callout associated with a mutex.
<code>callout_reset(co, ticks, fn, arg)</code>	Schedule or rearm a callout.

Function	Purpose
<code>callout_stop(co)</code>	Cancel a callout.
<code>callout_drain(co)</code>	Synchronously wait for a callout to finish.
<code>ifmedia_init(ifm, mask, change, status)</code>	Initialise a media descriptor.
<code>ifmedia_add(ifm, word, data, aux)</code>	Add a supported media entry.
<code>ifmedia_set(ifm, word)</code>	Choose the default media.
<code>ifmedia_ioctl(ifp, ifr, ifm, cmd)</code>	Handle SIOCGIFMEDIA and SIOCSIFMEDIA.
<code>ifmedia_removeall(ifm)</code>	Free all media entries on teardown.

28.15.2 Data-Path Functions

Function	Purpose
<code>if_transmit(ifp, m)</code>	The driver's outbound callback.
<code>if_input(ifp, m)</code>	Deliver a mbuf to the stack.
<code>if_qflush(ifp)</code>	Flush any driver-internal queues.
<code>BPF_MTAP(ifp, m)</code>	Tap a frame to BPF if any observers.
<code>bpf_mtap2(bpf, data, dlen, m)</code>	Tap with a prepended header.
<code>m_freem(m)</code>	Free an entire mbuf chain.
<code>m_free(m)</code>	Free a single mbuf.
<code>MGETHDR(m, how, type)</code>	Allocate a mbuf as the head of a packet.
<code>MGET(m, how, type)</code>	Allocate a mbuf as a chain continuation.
<code>m_gethdr(how, type)</code>	Alternative form of MGETHDR.
<code>m_pullup(m, len)</code>	Ensure the first len bytes are contiguous.
<code>m_copydata(m, off, len, buf)</code>	Read bytes from a chain without consuming it.
<code>m_defrag(m, how)</code>	Flatten a chain into a single mbuf.
<code>mtod(m, type)</code>	Cast <code>m_data</code> to the requested type.
<code>if_inc_counter(ifp, ctr, n)</code>	Increment a per-interface counter.
<code>if_link_state_change(ifp, state)</code>	Report a link transition.

28.15.3 Common IFF_ Flags

Flag	Meaning
IFF_UP	Administratively up. User-controlled.
IFF_BROADCAST	Supports broadcast.
IFF_DEBUG	Debug tracing requested.
IFF_LOOPBACK	Loopback interface.
IFF_POINTOPOINT	Point-to-point link.
IFF_RUNNING	Alias for IFF_DRV_RUNNING.
IFF_NOARP	ARP disabled.
IFF_PROMISC	Promiscuous mode.
IFF_ALLMULTI	Receive all multicast.
IFF_SIMPLEX	Cannot hear own transmissions.
IFF_MULTICAST	Supports multicast.
IFF_DRV_RUNNING	Driver-private: resources allocated.
IFF_DRV_OACTIVE	Driver-private: transmit queue full.

28.15.4 Common IFCAP_ Capabilities

Capability	Meaning
IFCAP_RXCSUM	IPv4 receive checksum offload.
IFCAP_TXCSUM	IPv4 transmit checksum offload.
IFCAP_RXCSUM_IPV6	IPv6 receive checksum offload.
IFCAP_TXCSUM_IPV6	IPv6 transmit checksum offload.
IFCAP_TS04	IPv4 TCP segmentation offload.
IFCAP_TS06	IPv6 TCP segmentation offload.
IFCAP_LR0	Large receive offload.
IFCAP_VLAN_HWTAGGING	Hardware VLAN tagging.
IFCAP_VLAN_MTU	VLAN over standard MTU.
IFCAP_JUMBO_MTU	Jumbo frames supported.
IFCAP_POLLING	Polled rather than interrupt-driven.
IFCAP_WOL_MAGIC	Wake-on-LAN magic packet.
IFCAP_NETMAP	netmap(4) support.
IFCAP_TOE	TCP offload engine.
IFCAP_LINKSTATE	Hardware link-state events.

28.15.5 Common IFCOUNTER_ Counters

Counter	Meaning
IFCOUNTER_IPACKETS	Packets received.
IFCOUNTER_IERRORS	Receive errors.
IFCOUNTER_OPACKETS	Packets transmitted.
IFCOUNTER_OERRORS	Transmit errors.
IFCOUNTER_COLLISIONS	Collisions (Ethernet).
IFCOUNTER_IBYTES	Bytes received.
IFCOUNTER_OBYTES	Bytes transmitted.
IFCOUNTER_IMCASTS	Multicast packets received.
IFCOUNTER_OMCASTS	Multicast packets transmitted.
IFCOUNTER_IQDROPS	Receive queue drops.
IFCOUNTER_OQDROPS	Transmit queue drops.
IFCOUNTER_NOPROTO	Packets for unknown protocol.

28.15.6 Common Interface Ioctls

Ioctl	When issued	Driver responsibility
SIOCSIFFLAGS	ifconfig up / down	Bring the driver up or down.
SIOCSIFADDR	ifconfig inet 1.2.3.4	Address assignment. Usually handled by ether_ioctl.
SIOCSIFMTU	ifconfig mtu N	Validate and update if_mtu.
SIOCADDMULTI	Multicast group joined	Reprogram hardware filter.
SIOCDELMULTI	Multicast group left	Reprogram hardware filter.
SIOCGIFMEDIA	ifconfig display	Return current media.
SIOCSIFMEDIA	ifconfig media X	Reprogram PHY or equivalent.
SIOCSIFCAP	ifconfig ±offloads	Toggle offloads.
SIOCSIFNAME	ifconfig name X	Rename the interface.

28.16 Reading Real Network Drivers

One of the best ways to solidify your understanding is to read real drivers in the FreeBSD tree. This section walks you through a handful of drivers that illustrate important patterns, and it suggests a reading order that builds on what this chapter has taught you. You do not need to understand every line in these files. The goal is

recognition: seeing the familiar bones of `ether_ifattach`, `if_transmit`, `if_input`, `ifmedia_init`, and so on inside drivers of very different sizes and purposes.

28.16.1 Reading `/usr/src/sys/net/if_tuntap.c`

The `tun(4)` and `tap(4)` drivers are implemented together in this file. They give userland a file descriptor through which packets can flow in and out of the kernel. Reading `if_tuntap.c` shows you how a driver can bridge the userland-character-device world of Chapter 14 with the network-stack world of this chapter.

Open the file and look for the following landmarks:

- The `cdevsw` declaration at the top, which is how userland opens `/dev/tun0` or `/dev/tap0`.
- The `tunstart` function, which moves packets from the kernel interface queue to userland reads.
- The `tunwrite` function, which moves packets from userland writes into the kernel via `if_input`.
- The `tuncreate` function, which allocates an `ifnet` and registers it.

You will see `ether_ifattach` for `tap` and plain `if_attach` for `tun`, because the two flavours differ in link-layer semantics: `tap` is an Ethernet-looking tunnel, while `tun` is a pure IP tunnel without a link layer. This file is an excellent case study in how the choice of `ether_ifattach` vs `if_attach` ripples through the rest of the driver.

Notice that `tuntap` does not use an interface cloner in the same way `disc` does. It creates interfaces on-demand when userland opens `/dev/tapN`, which shows yet another way interfaces can be brought into being. This is a variant of the cloner pattern, rather than a departure from it.

28.16.2 Reading `/usr/src/sys/net/if_bridge.c`

The bridge driver implements software Ethernet bridging between multiple interfaces. It is a larger file (more than three thousand lines) but its core is the same: it creates an `ifnet` per bridge, receives frames from member interfaces via `if_input` hooks, looks up destinations in a MAC-address-to-port table, and forwards frames through `if_transmit` on the outbound port.

What makes `if_bridge.c` particularly instructive is that it is itself both a client and a provider of the `ifnet` interface. It is a client because it transmits frames to member interfaces. It is a provider because it exposes a bridge interface that other code can use. Reading it shows you how to write a driver that is transparently layered on top of other drivers.

28.16.3 Reading `/usr/src/sys/dev/e1000/if_em.c`

The `em(4)` driver is the canonical example of a PCI Ethernet driver for Intel's e1000-class hardware. It is significantly larger than our pseudo-driver because it does everything real hardware requires: PCI attach, register programming, EEPROM reading, MSI-X allocation, ring-buffer management, DMA, interrupt handling, and so on.

However, if you squint past the hardware-specific parts, you will see our familiar patterns everywhere:

- `em_if_attach_pre` allocates a softc.
- `em_if_attach_post` populates the `ifnet`.

- `em_if_init` is the `if_init` callback.
- `em_if_ioctl` is the `if_ioctl` callback.
- `em_if_tx` is the transmit callback (wrapped through `iflib`).
- `em_if_rx` is the receive callback (wrapped through `iflib`).
- `em_if_detach` is the detach function.

The driver uses `iflib(9)` rather than raw `ifnet` calls, but `iflib` is itself a thin layer over the same APIs we have been using. Reading `em` is a good way to see how a real driver scales up from our small teaching example.

Focus on the transmit function first. You will see descriptor-ring management, DMA mapping, TSO handling, and checksum offload decisions. The amount of state is larger, but each decision has a clear purpose that maps to one of the concepts we discussed.

28.16.4 Reading `/usr/src/sys/dev/virtio/network/if_vtnet.c`

The `vtnet(4)` driver is for VirtIO network adapters used by virtual machines. It is smaller than `em` but still larger than our pseudo-driver. It uses `virtio(9)` as its transport rather than `bus_space(9)` plus DMA rings, which makes the code easier to follow if you are less familiar with PCI hardware.

`vtnet` is a particularly good second real driver to read after `mynet` because:

- It is used in almost every FreeBSD VM.
- Its source is clean and well-commented.
- It demonstrates multi-queue transmit and receive.
- It shows how offloads interact with the transmit path.

Spend an evening reading the transmit path and the receive path. You will probably find yourself recognising 70 to 80 percent of the patterns immediately, and the unfamiliar 20 percent will be things like VirtIO queue management that belong to the transport rather than to the network driver contract.

28.16.5 Reading `/usr/src/sys/net/if_lagg.c`

The link aggregation driver implements 802.3ad LACP, round-robin, failover, and other bonding protocols. It is an `ifnet` itself and aggregates over member `ifnets`. Reading it is an exercise in seeing how aggregate drivers can be layered on top of leaf drivers, and it shows you the full power of the `ifnet` abstraction: a bond interface looks the same to the stack as a single NIC.

28.16.6 Suggested Reading Order

If you have time for a deeper study, read in this order:

1. `if_disc.c`: the smallest pseudo-driver. You will recognise everything.
2. `if_tuntap.c`: pseudo-driver plus userland character-device interface.
3. `if_epair.c`: paired pseudo-drivers with simulated wire.
4. `if_bridge.c`: `ifnet`-layered driver.
5. `if_vtnet.c`: small real driver for VirtIO.
6. `if_em.c`: full-featured real driver using `iflib`.

7. `if_lagg.c`: aggregate driver.
8. `if_wg.c`: WireGuard tunnel driver. Modern, cryptographic, interesting.

After this sequence you will have seen enough drivers that almost any driver in the tree becomes readable. The unfamiliar parts will fall into “this is hardware-specific” or “this is a subsystem I have not studied yet”, both of which are finite and conquerable.

28.16.7 Reading as a Habit

Cultivate the habit of reading one driver per month. Pick one at random, read the attach function, and skim the transmit and receive paths. You will be surprised how quickly your vocabulary and your reading speed grow. By the end of a year, you will recognise patterns in drivers you have never seen before, and the instinct for “where should I look for this feature” becomes sharper.

Reading is also the best preparation for writing. The moment you need to add a new feature to a driver you have never touched, the experience of having read thirty drivers means you know roughly where to look and what to imitate.

28.17 Production Considerations

Most of this chapter has been about understanding. Before we close, a short section on what changes when you move from a teaching driver to a driver that will live in a production environment.

28.17.1 Performance

A production driver is usually measured in packets per second, bytes per second, or latency in microseconds. The pseudo-driver we built in this chapter is not stressed in any of those dimensions. If you try to take `mynet` to a real workload, you will quickly hit the limits of a single-mutex design, synchronous `m_freem`, and single-queue dispatch.

The refinements typically include:

- Per-queue locks rather than a softc-level lock.
- `drbr(9)` for per-CPU transmit rings.
- Taskqueue-based deferred receive processing.
- Pre-allocated mbuf pools with `m_getcl`.
- Bypass of `if_input` via direct dispatch helpers in some paths.
- Flow hashing to pin sockets to specific CPUs.
- Netmap support for kernel-bypass workloads.

Each of these optimisations adds code. A production-quality driver for a 10 Gbps NIC might be 3000 to 10000 lines of C, compared with our 500-line teaching driver.

28.17.2 Reliability

A production driver is expected to survive months of continuous operation without leaking memory, crashing the kernel, or drifting in its counter values. The practices that make this possible include:

- Running the kernel with INVARIANTS and WITNESS in QA, so that assertions catch bugs early.
- Writing regression tests that exercise every lifecycle path.
- Running the driver through stress tests (like `iperf3`, `pktgen`, or `netmap pkt-gen`) for extended periods.
- Instrumenting the driver with counters for every error path, so that operators can diagnose problems in the field.
- Providing clear diagnostics through `dmesg`, `sysctl`, and SDT probes.

These practices are not optional for a driver that will be deployed at scale. They are the cost of entry.

28.17.3 Observability

A well-written production driver exposes enough state through `sysctl`, counters, and DTrace probes that an operator can diagnose most problems without adding `printfs` or rebuilding the kernel. The rule of thumb is that every significant code path should have a counter or a probe point, and every decision that depends on runtime state should be queryable without a kernel rebuild.

For `mynet` we have only the built-in `ifnet` counters. A production version would add per-driver counters for things like transmit-path entries, receive-path drops, and interrupt-handler invocations. These counters are cheap to increment and priceless when a problem arrives.

28.17.4 Backward Compatibility

A driver that ships in a release must work on future releases too, ideally without modification. The FreeBSD kernel evolves its internal APIs over time, and drivers that reach too deep into structures may break when those structures change.

The accessor API we introduced in Section 2 is one of the defences. Using `if_setflagbits` rather than `ifp->if_flags |= flag` insulates you from layout changes. Similarly, `if_inc_counter` rather than direct counter updates insulates you from counter-representation changes.

For production drivers, prefer the accessor style whenever it is readily available.

28.17.5 Licensing and Upstreaming

A driver you intend to merge upstream must be licensed compatibly with the FreeBSD tree, which is typically a two-clause BSD license. It should also follow KNF (`style(9)`), include manual pages under `share/man/man4`, include a module Makefile under `sys/modules`, and be submitted through the FreeBSD contribution process (Phabricator reviews as of this writing).

Teaching drivers like `mynet` do not need to worry about upstreaming, but if you are writing a driver with the intent to ship it to others, these are the additional considerations that turn your C code into a community artefact.

28.18 Wrapping Up

Take a moment to appreciate what you have just done. You have:

- Built your first network driver, from scratch.

- Registered it with the network stack through `ifnet` and `ether_ifattach`.
- Implemented a transmit path that accepts mbufs, taps BPF, updates counters, and cleans up.
- Implemented a receive path that constructs mbufs, hands them to BPF, and delivers them to the stack.
- Handled interface flags, link-state transitions, and media descriptors.
- Tested the driver with `ifconfig`, `netstat`, `tcpdump`, `ping`, and `route monitor`.
- Cleaned up on interface destroy and on module unload, without leaks.

More important than any of these individual accomplishments, you have internalised a mental model. A network driver is a participant in a contract with the kernel's network stack. The contract has a fixed shape: a few callbacks going down, one call going up, a handful of flags, a few counters, a media descriptor, a link state. Once you can see that shape clearly, every network driver in the FreeBSD tree becomes comprehensible. The production drivers are bigger, but they are not fundamentally different.

28.18.1 What This Chapter Did Not Cover

A few topics are within reach but were deliberately deferred to keep this chapter tractable.

iflib(9). The modern NIC driver framework that most production drivers use on FreeBSD 14. `iflib` shares transmit and receive ring buffers across many drivers and provides a simpler callback-oriented model for hardware NICs. The patterns we have written by hand in this chapter are exactly what `iflib` automates, so everything you learned here is still valid. We will look at `iflib` in later chapters when we study specific hardware drivers.

DMA for receive and transmit. Real NICs move packet data through DMA-mapped ring buffers. The `bus_dma(9)` API we introduced in earlier chapters is how that is done. Adding DMA to a driver turns the mbuf construction story into “map the mbuf, hand the mapped address to hardware, wait for the completion interrupt, unmap”. That is a significant amount of additional code, and it deserves its own treatment in a later chapter.

MSI-X and interrupt moderation. Modern NICs have multiple interrupt vectors and support interrupt coalescing. We used a callout because we are a pseudo-driver. Real drivers use interrupt handlers. Interrupt moderation (letting the hardware aggregate several completion events into a single interrupt) is critical for performance.

netmap(4). The kernel-bypass fast path used by some high-performance applications. Drivers opt in by calling `netmap_attach()` and exposing per-queue ring buffers. It is a specialisation for throughput-sensitive use cases.

polling(4). An older technique where the driver is polled for packets by a kernel thread rather than driven by interrupts. Still available but less commonly used than it once was.

VNET in detail. We set up the driver to be VNET-compatible, but we did not explore what it means to move interfaces between VNETs with `if_vmove`, or what a VNET teardown looks like from the driver's perspective. Chapter 30 will visit that territory.

Hardware offloads. Checksum offload, TSO, LRO, VLAN tagging, encryption offload. All of these are capabilities a real NIC might expose. A driver that advertises them has to honour them, and that leads into a rich design space we have not touched.

Wireless. `wlan(4)` drivers are radically different from Ethernet drivers because they deal with 802.11 frame formats, scanning, authentication, and management frames. The `ifnet` is still present, but it sits on top of a very different link layer. We will visit wireless drivers in a later chapter.

Network graph (netgraph(4)). FreeBSD’s packet-filtering and classification framework. It is largely orthogonal to driver writing but worth knowing about for advanced network architectures.

Bridging and VLAN interfaces. Virtual interfaces that aggregate or modify traffic. They are built on top of `ifnet`, exactly like our driver, but their role is quite different.

Each of these topics deserves its own chapter. What you have built here is the stable base camp from which those expeditions set out.

28.18.2 Final Reflection

Network drivers have a reputation as a demanding subfield of kernel engineering. They deserve it: the constraints are tight, the interactions with the stack are many, the performance expectations are high, and the user-facing commands are numerous. But the structure of a network driver is clean once you can see it. That is what this chapter has given you: the ability to see the structure.

Read `if_em.c`, `if_bge.c`, or `if_tuntap.c` now. You will recognise the skeleton. The `softc`. The `ether_ifattach` call. The `if_transmit`. The `if_ioctl` switch. The `if_input` in the receive handler. The `bpfattach` and `BPF_MTAP`. Wherever the code adds complexity, it is adding complexity to a skeleton you have already built in miniature.

Like Chapter 27, this chapter is long because the topic is layered. We have tried to make every layer land softly before the next arrives. If a particular section did not settle, go back and do the corresponding lab again. Kernel learning is strongly cumulative. A second pass through one section often does more than a first pass through the next one.

28.18.3 Further Reading

Manual pages. `ifnet(9)`, `ifmedia(9)`, `mbuf(9)`, `ether(9)`, `bpf(9)`, `polling(9)`, `ifconfig(8)`, `netstat(1)`, `tcpdump(1)`, `route(8)`, `ngctl(8)`. Read them in that order.

The FreeBSD Architecture Handbook. The networking chapters are a good complement.

Kirk McKusick et al., “The Design and Implementation of the FreeBSD Operating System”. The network-stack chapters are especially relevant.

“TCP/IP Illustrated, Volume 2” by Wright and Stevens. A classic walk-through of a BSD-derived network stack. Dated but still unique in its depth.

The FreeBSD source tree. `/usr/src/sys/net/`, `/usr/src/sys/netinet/`, `/usr/src/sys/dev/e1000/`, `/usr/src/sys/dev/bge/`, `/usr/src/sys/dev/mlx5/`. Every pattern discussed in this chapter is grounded in that code.

The mailing list archives. `freebsd-net@` is the most relevant list. Reading historic threads is a great way to pick up idioms that never made it into formal documentation.

Commit history on GitHub mirrors. The FreeBSD repository has an excellent history. `git log --follow sys/net/if_var.h` is a good starting point for seeing how the `ifnet` abstraction evolved.

The FreeBSD Developer Summit slides. When available, these often include networking-focused sessions.

Other BSDs. NetBSD and OpenBSD have slightly different network-driver frameworks, but the core ideas are identical. Reading a driver in another BSD after reading its FreeBSD equivalent is a good way to understand what is universal and what is FreeBSD-specific.

28.19 Field Guide to Related ifnet Subsystems

You have built a driver. You have read a handful of real drivers. Before we close the chapter, let us survey the surrounding subsystems so that you know where to look when you need them.

28.19.1 arp(4) and Neighbour Discovery

ARP for IPv4 lives in `/usr/src/sys/netinet/if_ether.c`. It is the subsystem that maps IP addresses to MAC addresses. Drivers do not usually interact with ARP directly; they carry packets (including ARP requests and replies) through their transmit and receive paths, and the ARP code inside `ether_input` and `arpresolve` does the rest.

The IPv6 equivalent is neighbour discovery, in `/usr/src/sys/netinet6/nd6.c`. It uses ICMPv6 rather than a separate protocol, but the role is the same: map IPv6 addresses to MAC addresses for on-link delivery.

28.19.2 bpf(4)

The Berkeley Packet Filter subsystem lives in `/usr/src/sys/net/bpf.c`. BPF is the userland-visible mechanism for packet capture. `tcpdump(1)`, `libpcap(3)`, and many other tools use BPF. Drivers register with BPF through `bpfattach` (done automatically by `ether_ifattach`) and tap packets to BPF through `BPF_MTAP` (which you do manually).

BPF filters are programs written in the BPF pseudo-machine language, compiled to bytecode in userland and executed in the kernel. They are what allows `tcpdump -i 'port 80'` to work efficiently: the filter runs before the packet is copied to userland, so only matching packets are transferred.

28.19.3 route(4)

The routing subsystem lives in `/usr/src/sys/net/route.c` and is growing over time (the recent `nhop(9)` next-hop abstraction is a notable change). Drivers interact with routing indirectly: when they report link state changes, the routing subsystem updates metrics; when they transmit, the stack has already done the route lookup. `route monitor`, which we used in a lab, subscribes to routing events and displays them.

28.19.4 if_clone(4)

The cloner subsystem in `/usr/src/sys/net/if_clone.c` is what we have been using throughout this chapter. It manages the list of per-driver cloners and dispatches `ifconfig create` and `ifconfig destroy` requests to the correct driver.

28.19.5 pf(4)

The packet filter lives in `/usr/src/sys/net/pfil/pf/`. It is independent of any specific driver and runs as a hook on packet paths through `pfil(9)`. Drivers do not usually interact with `pf` directly; traffic passing through the stack is filtered transparently.

28.19.6 netmap(4)

`netmap(4)` is a kernel-bypass packet-I/O framework in `/usr/src/sys/dev/netmap/`. Drivers that support `netmap` expose their ring buffers directly to userland, bypassing the normal `if_input` and `if_transmit` paths. This allows applications to receive and transmit packets at line rate without kernel involvement. Only a handful of drivers support `netmap` natively; the rest use a shim that emulates `netmap` semantics at the cost of some performance.

28.19.7 netgraph(4)

`netgraph(4)` is FreeBSD's modular packet-processing framework, in `/usr/src/sys/netgraph/`. It lets you build arbitrary graphs of packet-processing nodes in the kernel, configured from userland via `ngctl`. Drivers can expose themselves as `netgraph` nodes (see `ng_ether.c`), and `netgraph` can be used to implement tunnels, PPP over Ethernet, encrypted links, and many other features without modifying the stack itself.

28.19.8 iflib(9)

`iflib(9)` is the modern framework for high-performance Ethernet drivers in `/usr/src/sys/net/iflib.c`. It takes over the rote parts of a NIC driver (ring-buffer management, interrupt handling, TSO fragmentation, LRO aggregation) and leaves the driver writer to provide the hardware-specific callbacks. On the drivers that have been converted to `iflib` to date, driver code typically shrinks by 30 to 50 percent compared with the equivalent plain-`ifnet` implementation. See Appendix F for a reproducible line-count comparison across the `iflib` and non-`iflib` driver corpora. Appendix F's `iflib` section also pins a specific per-driver measurement on the `ixgbe` conversion commit at roughly the lower edge of this range.

For now, `iflib` is beyond the scope of this chapter. Appendix F's `iflib` section offers a reproducible line-count comparison that shows how much code the framework saves on drivers that have been converted to it.

28.19.9 Summary of the Landscape

A network driver lives in a rich environment. Above it are ARP, IP, TCP, UDP, and the socket layer. Alongside it are BPF, `pf`, `netmap`, and `netgraph`. Below it is hardware, or a transport simulation, or a pipe to userland. Each of these components has its own conventions, and learning any of them in depth is a meaningful investment. What this chapter has given you is enough familiarity with the central object, the `ifnet`, that you can approach any of these subsystems without being intimidated.

28.20 Debugging Scenarios: A Worked Example

One of the best ways to close a chapter about driver-writing is to walk through a specific debugging session. The scenario below is composite: it combines symptoms and fixes from several different driver bugs into one narrative, so that the full arc of “something is wrong, let us find it” is visible.

28.20.1 The Problem

You load `mynet`, create an interface, assign an IP, and run `ping`. The ping reports 100% loss, as expected (our pseudo-driver has nothing on the other end). But `netstat -in -I mynet0` shows `Opkts 0` even after several pings. Something in the transmit path is broken.

28.20.2 First Hypothesis: the transmit function is not being called.

You run `dtrace -n 'fbt::mynet_transmit:entry { printf("called"); }'`. No output, even during the ping. That confirms that `if_transmit` is not being invoked.

28.20.3 Investigating Why

You open the source and find that `ifp->if_transmit = mynet_transmit;` is present. You check `ifp->if_transmit` at runtime by grabbing the `ifnet` pointer via `ifconfig`'s reporting (there is no direct way to read function pointers from userland, but a DTrace probe can do it):

```
# dtrace -n 'fbt::ether_output_frame:entry {  
    printf("if_transmit = %p", args[0]->if_transmit);  
}'
```

The output shows a different address than you expected. Closer inspection reveals that `ether_ifattach` overwrote `if_transmit` with its own wrapper. You `grep` for `if_transmit` in `if_ethersubr.c` and confirm that `ether_ifattach` sets `ifp->if_output = ether_output` but does not touch `if_transmit`. So `if_transmit` should still be your function.

You go back to your source and notice that you set `ifp->if_transmit = mynet_transmit;` before `ether_ifattach`, but you inadvertently set it through the legacy `if_start` field in a second assignment that you forgot to remove. The legacy `if_start` mechanism takes precedence under some conditions, and the kernel ends up calling `if_start` instead of `if_transmit`.

You remove the stray `if_start` assignment and rebuild. The transmit function is now called.

28.20.4 Second Problem: counter disagreement

Transmit is now called, and `Opkts` climbs. But `Obytes` is suspiciously low: it is incremented by one per ping, not by the ping's byte length. You re-inspect the counter update code:

```
if_inc_counter(ifp, IFCOUNTER_OBYTES, 1);
```

The constant 1 should be `len`. You typed the wrong argument. You change it to `if_inc_counter(ifp, IFCOUNTER_OBYTES, len)` and rebuild. Obytes now climbs by the expected amount.

28.20.5 Third Problem: the receive path seems intermittent

Synthesised ARPs appear most of the time but occasionally stop for several seconds. You add a DTrace probe to `mynet_rx_timer` and see that the function is being called at regular intervals, but that some calls early-return without generating a frame.

You inspect `mynet_rx_fake_arp` and find that it uses `M_NOWAIT` for its mbuf allocation. Under memory pressure, `M_NOWAIT` returns `NULL`, and the receive path silently drops. You instrument the allocation failure path:

```
if (m == NULL) {
    if_inc_counter(ifp, IFCOUNTER_IQDROPS, 1);
    return;
}
```

And you check the counter: it matches the missing frames. You have found the cause: transient mbuf pressure on your test VM. The fix is to either accept occasional drops (they are legitimate and correctly counted) or to switch to `M_WAITOK` if the callout can tolerate sleeping (it cannot, because callouts run in a non-sleepable context).

In this case, accepting the drops is correct. The fix is therefore to make the behaviour visible in a dashboard: you add a sysctl that exposes `IFCOUNTER_IQDROPS` on this specific interface, and note it in the driver documentation.

28.20.6 What This Scenario Teaches

Three separate bugs. None of them was catastrophic. Each one required a different combination of tools to diagnose: DTrace for function tracing, code reading for understanding the API, and counters for observing the runtime effect.

The lesson is that driver bugs tend to hide in plain sight. The first rule of driver debugging is “do not trust, verify”. The second rule is “the counters and the tools will tell you”. The third rule is “if the counters do not tell you what you need, add more counters or more probes”.

With practice, debugging sessions like this become faster. You develop an instinct for which tool to reach for first, and the difference between a driver that works on the first load and a driver that took six iterations becomes a shorter debugging cycle.

28.21 A Note on Testing Discipline

Before we truly close the chapter, a few paragraphs on testing discipline. A teaching driver can be tested casually. A driver you intend to maintain over time deserves a more rigorous approach.

28.21.1 Unit-Level Thinking

Every callback in your driver has a small, well-defined contract. `mynet_transmit` takes an `ifnet` and a `mbuf`, validates, counts, taps, and frees. `mynet_ioctl` takes an `ifnet` and an `ioctl` code, dispatches, and returns an `errno`. Each of these can be exercised independently.

In practice, unit testing kernel code is hard because the kernel is not easy to embed in a userland test harness. But you can approximate the discipline by designing the code so that most of each callback is pure: given inputs, produce outputs, without touching global state. The validation block in `mynet_transmit` is a good example: it does not touch anything except `ifp->if_mtu` and local variables.

A mental model of “this callback has a contract; here are the cases that exercise the contract; here are the expected behaviours for each case” is the foundation of good testing.

28.21.2 Lifecycle Tests

Every driver should be tested across its full lifecycle: load, create, configure, carry traffic, stop, destroy, unload. The Lab 6 script is a minimum version of such a test. A more rigorous version would include:

- Multiple interfaces created concurrently.
- Unload while traffic is flowing (at low rate, to be safe).
- Repeated load/unload cycles to catch leaks.
- Tests with `INVARIANTS` and `WITNESS` enabled.

28.21.3 Error-Path Tests

Every error-path in the driver needs to be exercisable. If `if_alloc` fails, does the create function roll back cleanly? If an `ioctl` returns an error, does the caller cope? If the callout fails to allocate a `mbuf`, does the driver stay consistent?

One useful technique is fault injection: add a `sysctl` that probabilistically fails specific operations (`if_alloc`, `m_gethdr`, and so on), and run your lifecycle tests with the fault injection enabled. This exposes error paths that almost never fire in production but can still happen under load.

28.21.4 Regression Tests

Whenever you fix a bug, add a test that would have caught it. Even a simple shell script that loads the driver, exercises a specific feature, and checks a counter is a regression test.

Over time, a regression-test suite becomes a guardrail against reintroducing bugs. It also documents the behaviour you guarantee. A new contributor reading the test suite gains a clearer picture of what the driver promises than any amount of code reading could provide.

28.21.5 Watching for Latent Issues

Some issues only manifest after hours or days of operation: slow memory leaks, counter drift, rare race conditions. Long-running tests are the only way to find these. A driver deployed to production without at least a 24-hour soak under representative load is not ready.

For `mynet` the soak might be as simple as “leave the driver loaded for a day and check `vmstat -m` and `vmstat -z` at the end”. For a real driver, the soak might involve terabyte-hours of traffic under a real workload. The scale differs; the principle is the same.

28.22 A Complete Walkthrough of `mynet.c`

Before we close the chapter, it is worth showing a concise end-to-end walkthrough of the reference driver. The goal is to see the entire driver once in one place, with short annotations at each step, so that you can visualise the full shape without having to jump between Sections 3 through 6.

28.22.1 File-Level Preamble

The driver opens with a license header, the copyright notice, and the include block we described in Section 3. After the includes, the file declares the memory type, the cloner variable, and the `softc` structure:

```
static const char mynet_name[] = "mynet";
static MALLOC_DEFINE(M_MYNET, "mynet", "mynet pseudo Ethernet driver");

VNET_DEFINE_STATIC(struct if_clone *, mynet_cloner);
#define V_mynet_cloner VNET(mynet_cloner)

struct mynet_softc {
    struct ifnet      *ifp;
    struct mtx        mtx;
    uint8_t           hwaddr[ETHER_ADDR_LEN];
    struct ifmedia     media;
    struct callout     rx_callout;
    int               rx_interval_hz;
    bool              running;
};

#define MYNET_LOCK(sc)      mtx_lock(&(sc)->mtx)
#define MYNET_UNLOCK(sc)   mtx_unlock(&(sc)->mtx)
#define MYNET_ASSERT(sc)   mtx_assert(&(sc)->mtx, MA_OWNED)
```

Every field and macro here has a purpose we discussed earlier. The `softc` carries the per-instance state; the locking macros document when the mutex should be held; the VNET-aware cloner is the mechanism by which `ifconfig mynet create` produces new interfaces.

28.22.2 Forward Declarations

A small block of forward declarations for the static functions the driver exposes as callbacks:

```

static int      mynet_clone_create(struct if_clone *, int, caddr_t);
static void     mynet_clone_destroy(struct ifnet *);
static int      mynet_create_unit(int unit);
static void     mynet_destroy(struct mynet_softc *);
static void     mynet_init(void *);
static void     mynet_stop(struct mynet_softc *);
static int      mynet_transmit(struct ifnet *, struct mbuf *);
static void     mynet_qflush(struct ifnet *);
static int      mynet_ioctl(struct ifnet *, u_long, caddr_t);
static int      mynet_media_change(struct ifnet *);
static void     mynet_media_status(struct ifnet *, struct ifmediareq *);
static void     mynet_rx_timer(void *);
static void     mynet_rx_fake_arp(struct mynet_softc *);
static int      mynet_modevent(module_t, int, void *);
static void     vnet_mynet_init(const void *);
static void     vnet_mynet_uninit(const void *);

```

Forward declarations are a courtesy to the reader. They let you scan the top of the file and see every named function the driver exports, without searching for definitions.

28.22.3 Cloner Dispatch

The cloner create and destroy functions are thin wrappers that delegate the real work to the per-unit helpers:

```

static int
mynet_clone_create(struct if_clone *ifc __unused, int unit, caddr_t params __unused)
{
    return (mynet_create_unit(unit));
}

static void
mynet_clone_destroy(struct ifnet *ifp)
{
    mynet_destroy((struct mynet_softc *)ifp->if_softc);
}

```

Keeping the cloner callbacks small is a convention worth following. It makes it easy to test the real work functions (`mynet_create_unit`, `mynet_destroy`) in isolation and it makes the cloner glue boring.

28.22.4 Per-Unit Creation

The per-unit creation function is where the real setup happens:

```

static int
mynet_create_unit(int unit)
{
    struct mynet_softc *sc;
    struct ifnet *ifp;

    sc = malloc(sizeof(*sc), M_MYNET, M_WAITOK | M_ZERO);
    ifp = if_alloc(IFT_ETHER);
    if (ifp == NULL) {
        free(sc, M_MYNET);
        return (ENOSPC);
    }
    sc->ifp = ifp;
    mtx_init(&sc->mtx, "mynet", NULL, MTX_DEF);

    arc4rand(sc->hwaddr, ETHER_ADDR_LEN, 0);
    sc->hwaddr[0] = 0x02; /* locally administered, unicast */

    if_initname(ifp, mynet_name, unit);
    ifp->if_softc = sc;
    ifp->if_flags = IFF_BROADCAST | IFF_SIMPLEX | IFF_MULTICAST;
    ifp->if_capabilities = IFCAP_VLAN_MTU;
    ifp->if_capenable = IFCAP_VLAN_MTU;
    ifp->if_transmit = mynet_transmit;
    ifp->if_qflush = mynet_qflush;
    ifp->if_ioctl = mynet_ioctl;
    ifp->if_init = mynet_init;
    ifp->if_baudrate = IF_Gbps(1);

    ifmedia_init(&sc->media, 0, mynet_media_change, mynet_media_status);
    ifmedia_add(&sc->media, IFM_ETHER | IFM_1000_T | IFM_FDX, 0, NULL);
    ifmedia_add(&sc->media, IFM_ETHER | IFM_AUTO, 0, NULL);
    ifmedia_set(&sc->media, IFM_ETHER | IFM_AUTO);

    callout_init_mtx(&sc->rx_callout, &sc->mtx, 0);
    sc->rx_interval_hz = hz;

    ether_ifattach(ifp, sc->hwaddr);
    return (0);
}

```

You can see every concept from Section 3 in one place: softc and ifnet allocation, MAC fabrication, field configuration, media setup, callout initialisation, and the final `ether_ifattach` that registers the interface with the stack.

28.22.5 Destruction

Destruction mirrors creation in reverse:

```
static void
mynet_destroy(struct mynet_softc *sc)
{
    struct ifnet *ifp = sc->ifp;

    MYNET_LOCK(sc);
    sc->running = false;
    ifp->if_drv_flags &= ~IFF_DRV_RUNNING;
    MYNET_UNLOCK(sc);

    callout_drain(&sc->rx_callout);

    ether_ifdetach(ifp);
    if_free(ifp);

    ifmedia_removeall(&sc->media);
    mtx_destroy(&sc->mtx);
    free(sc, M_MYNET);
}
```

Again, every step is one we discussed. The order is quiesce, detach, free.

28.22.6 Init and Stop

The transitions between “not running” and “running” are handled by two small functions:

```
static void
mynet_init(void *arg)
{
    struct mynet_softc *sc = arg;

    MYNET_LOCK(sc);
    sc->running = true;
    sc->ifp->if_drv_flags |= IFF_DRV_RUNNING;
    sc->ifp->if_drv_flags &= ~IFF_DRV_OACTIVE;
    callout_reset(&sc->rx_callout, sc->rx_interval_hz,
        mynet_rx_timer, sc);
    MYNET_UNLOCK(sc);

    if_link_state_change(sc->ifp, LINK_STATE_UP);
}
```

```
}

static void
mynet_stop(struct mynet_softc *sc)
{
    MYNET_LOCK(sc);
    sc->running = false;
    sc->ifp->if_drv_flags &= ~IFF_DRV_RUNNING;
    callout_stop(&sc->rx_callout);
    MYNET_UNLOCK(sc);

    if_link_state_change(sc->ifp, LINK_STATE_DOWN);
}
```

Both are symmetric, both honour the rule about dropping the lock before `if_link_state_change`, and both maintain the coherence rules we described in Section 6.

28.22.7 The Data Path

Transmit and the simulated receive are the heart of the driver:

```
static int
mynet_transmit(struct ifnet *ifp, struct mbuf *m)
{
    struct mynet_softc *sc = ifp->if_softc;
    int len;

    if (m == NULL)
        return (0);
    M_ASSERTPKTHDR(m);

    if (m->m_pkthdr.len > (ifp->if_mtu + sizeof(struct ether_vlan_header))) {
        m_freem(m);
        if_inc_counter(ifp, IFCOUNTER_OERRORS, 1);
        return (E2BIG);
    }

    if ((ifp->if_flags & IFF_UP) == 0 ||
        (ifp->if_drv_flags & IFF_DRV_RUNNING) == 0) {
        m_freem(m);
        if_inc_counter(ifp, IFCOUNTER_OERRORS, 1);
        return (ENETDOWN);
    }
}
```

```

    BPF_MTAP(ifp, m);

    len = m->m_pkthdr.len;
    if_inc_counter(ifp, IFCOUNTER_OPACKETS, 1);
    if_inc_counter(ifp, IFCOUNTER_OBYTES, len);
    if (m->m_flags & (M_BCAST | M_MCAST))
        if_inc_counter(ifp, IFCOUNTER_OMCASTS, 1);

    m_freem(m);
    return (0);
}

static void
mynet_qflush(struct ifnet *ifp __unused)
{
}

static void
mynet_rx_timer(void *arg)
{
    struct mynet_softc *sc = arg;

    MYNET_ASSERT(sc);
    if (!sc->running)
        return;
    callout_reset(&sc->rx_callout, sc->rx_interval_hz,
        mynet_rx_timer, sc);
    MYNET_UNLOCK(sc);

    mynet_rx_fake_arp(sc);

    MYNET_LOCK(sc);
}

```

The fake-ARP helper builds the synthetic frame and hands it to the stack:

```

static void
mynet_rx_fake_arp(struct mynet_softc *sc)
{
    struct ifnet *ifp = sc->ifp;
    struct mbuf *m;
    struct ether_header *eh;
    struct arphdr *ah;
    uint8_t *payload;

```

```

size_t frame_len;

frame_len = sizeof(*eh) + sizeof(*ah) + 2 * (ETHER_ADDR_LEN + 4);
MGETHDR(m, M_NOWAIT, MT_DATA);
if (m == NULL) {
    if_inc_counter(ifp, IFCOUNTER_IQDROPS, 1);
    return;
}
m->m_pkthdr.len = m->m_len = frame_len;
m->m_pkthdr.rcvif = ifp;

eh = mtod(m, struct ether_header *);
memset(eh->ether_dhost, 0xff, ETHER_ADDR_LEN);
memcpy(eh->ether_shost, sc->hwaddr, ETHER_ADDR_LEN);
eh->ether_type = htons(ETHERTYPE_ARP);

ah = (struct arphdr *)(eh + 1);
ah->ar_hrd = htons(ARPHRD_ETHER);
ah->ar_pro = htons(ETHERTYPE_IP);
ah->ar_hln = ETHER_ADDR_LEN;
ah->ar_pln = 4;
ah->ar_op = htons(ARPOP_REQUEST);

payload = (uint8_t *)(ah + 1);
memcpy(payload, sc->hwaddr, ETHER_ADDR_LEN);
payload += ETHER_ADDR_LEN;
memset(payload, 0, 4);
payload += 4;
memset(payload, 0, ETHER_ADDR_LEN);
payload += ETHER_ADDR_LEN;
memcpy(payload, "\xc0\x00\x02\x63", 4);

BPF_MTAP(ifp, m);
if_inc_counter(ifp, IFCOUNTER_IPACKETS, 1);
if_inc_counter(ifp, IFCOUNTER_IBYTES, frame_len);
if_inc_counter(ifp, IFCOUNTER_IMCASTS, 1); /* broadcast counts as multicast */

if_input(ifp, m);
}

```

28.22.8 Ioctl and Media Callbacks

The ioctl handler and the two media callbacks:


```
static int
mynet_ioctl(struct ifnet *ifp, u_long cmd, caddr_t data)
{
    struct mynet_softc *sc = ifp->if_softc;
    struct ifreq *ifr = (struct ifreq *)data;
    int error = 0;

    switch (cmd) {
    case SIOCSIFFLAGS:
        MYNET_LOCK(sc);
        if (ifp->if_flags & IFF_UP) {
            if ((ifp->if_drv_flags & IFF_DRV_RUNNING) == 0) {
                MYNET_UNLOCK(sc);
                mynet_init(sc);
                MYNET_LOCK(sc);
            }
        } else {
            if (ifp->if_drv_flags & IFF_DRV_RUNNING) {
                MYNET_UNLOCK(sc);
                mynet_stop(sc);
                MYNET_LOCK(sc);
            }
        }
        MYNET_UNLOCK(sc);
        break;

    case SIOCSIFMTU:
        if (ifr->ifr_mtu < 68 || ifr->ifr_mtu > 9216) {
            error = EINVAL;
            break;
        }
        ifp->if_mtu = ifr->ifr_mtu;
        break;

    case SIOCADDMULTI:
    case SIOCDELMULTI:
        break;

    case SIOCGIFMEDIA:
    case SIOCSIFMEDIA:
        error = ifmedia_ioctl(ifp, ifr, &sc->media, cmd);
        break;

    default:
```

```

        error = ether_ioctl(ifp, cmd, data);
        break;
    }

    return (error);
}

static int
mynet_media_change(struct ifnet *ifp __unused)
{
    return (0);
}

static void
mynet_media_status(struct ifnet *ifp, struct ifmediareq *imr)
{
    struct mynet_softc *sc = ifp->if_softc;

    imr->ifm_status = IFM_AVALID;
    if (sc->running)
        imr->ifm_status |= IFM_ACTIVE;
    imr->ifm_active = IFM_ETHER | IFM_1000_T | IFM_FDX;
}

```

28.22.9 Module Glue and Cloner Registration

The bottom of the file holds the module handler, the VNET sysinit/sysuninit functions, and the module declarations:

```

static void
vnet_mynet_init(const void *unused __unused)
{
    V_mynet_cloner = if_clone_simple(mynet_name, mynet_clone_create,
        mynet_clone_destroy, 0);
}

VNET_SYSINIT(vnet_mynet_init, SI_SUB_PSEUDO, SI_ORDER_ANY,
    vnet_mynet_init, NULL);

static void
vnet_mynet_uninit(const void *unused __unused)
{
    if_clone_detach(V_mynet_cloner);
}

VNET_SYSUNINIT(vnet_mynet_uninit, SI_SUB_INIT_IF, SI_ORDER_ANY,

```

```
vnet_mynet_uninit, NULL);

static int
mynet_modevent(module_t mod __unused, int type, void *data __unused)
{
    switch (type) {
        case MOD_LOAD:
        case MOD_UNLOAD:
            return (0);
        default:
            return (EOPNOTSUPP);
    }
}

static moduledata_t mynet_mod = {
    "mynet",
    mynet_modevent,
    NULL
};

DECLARE_MODULE(mynet, mynet_mod, SI_SUB_PSEUDO, SI_ORDER_ANY);
MODULE_DEPEND(mynet, ether, 1, 1, 1);
MODULE_VERSION(mynet, 1);
```

28.22.10 Line Count and Density

The complete `mynet.c` is about 500 lines of C code. The entire teaching driver, from the license header at the top to the `MODULE_VERSION` at the bottom, is shorter than many single functions in production drivers. That compactness is not a coincidence: pseudo-drivers have no hardware to talk to, so they can focus on the `ifnet` contract and nothing else.

Read through the complete file in the companion materials. Type your own copy if you have not already done so. Build it. Load it. Mutate it. Until you have internalised the shape, do not move on to the next chapter.

28.23 A Complete Lifecycle Trace

It helps to see, end to end, the sequence of events that happens when the reader runs the ordinary commands from a shell. The trace below follows the mental model we have built up, but stitches it together into one continuous story. Read it as an animated flipbook rather than as a reference table.

28.23.1 Trace 1: From `kldload` to `ifconfig up`

Imagine you are sitting at the keyboard with a fresh FreeBSD 14.3 machine. You have never loaded `mynet` before. You type the first command:

```
# kldload ./mynet.ko
```

What happens next? The loader reads the ELF header of `mynet.ko`, relocates the module into kernel memory, and walks the module's `modmetadata_set` linker set. It finds the `DECLARE_MODULE` record for `mynet` and calls `mynet_modevent(mod, MOD_LOAD, data)`. Our handler returns zero without doing any work. The loader also processes the `MODULE_DEPEND` records, and because `ether` is already part of the base kernel, the dependency is satisfied immediately.

Then the linker set for `VNET_SYSINIT` is walked. Our `vnet_mynet_init()` fires. It calls `if_clone_simple()` with the name `mynet` and the two callbacks `mynet_clone_create` and `mynet_clone_destroy`. The kernel registers a new cloner in the `VNET` cloner list. At this point, no interface exists yet: the cloner is only a factory.

The shell prompt returns. You type:

```
# ifconfig mynet create
```

`ifconfig(8)` opens a datagram socket and issues the `SIOCIFCREATE2` ioctl on it, passing the name `mynet`. The kernel's clone dispatcher finds the `mynet` cloner and calls `mynet_clone_create(cloner, unit, params, params_len)` with the first available unit number, which is zero. Our callback allocates a `mynet_softc`, locks its mutex, calls `if_alloc(IFT_ETHER)`, fills in the callbacks, initialises the media table, generates a MAC address, calls `ether_ifattach()`, and returns zero. Inside `ether_ifattach()`, the kernel calls `if_attach()` which links the interface into the global interface list, calls `bpfattach()` so `tcpdump(8)` can observe it, publishes the device to userland through `devd(8)`, and runs any registered `ifnet_arrival_event` handlers.

The shell prompt returns again. You type:

```
# ifconfig mynet0 up
```

Same socket, same kind of ioctl, different command: `SIOCSIFFLAGS`. The kernel looks up the interface by name, finds `mynet0`, and calls `mynet_ioctl(ifp, SIOCSIFFLAGS, data)`. Our handler observes that `IFF_UP` is set but `IFF_DRV_RUNNING` is not, so it calls `mynet_init()`. That function flips `running` to true, sets `IFF_DRV_RUNNING` on the interface, schedules the first callout tick, and returns. The ioctl returns zero. The shell prompt returns.

You type:

```
# ping -c 1 -t 1 192.0.2.99
```

At this point the network stack attempts ARP resolution. It builds an ARP request packet, formatted as Ethernet + ARP, and calls `ether_output()` for the interface. `ether_output()` prepends the Ethernet header, calls `if_transmit()`, which is a macro that calls our `mynet_transmit()` function. Our transmit function increments counters, taps BPF, frees the mbuf, and returns zero. `tcpdump -i mynet0` would have seen the ARP request in flight.

Meanwhile, because our driver is also generating fake inbound ARP responses on its callout timer, the next callout tick synthesises an ARP reply, calls `if_input()`, and the stack believes it has heard from `192.0.2.99`.

ping sends the ICMP echo request, our driver taps it, frees the mbuf, and records success. ping never gets a reply, because our driver only fakes ARP; but the lifecycle worked exactly as expected, and nothing crashed.

This sequence, trivial as it looks, exercises almost every code path in your driver. Internalise it.

28.23.2 Trace 2: From ifconfig down to kldunload

Now you are cleaning up. You type:

```
# ifconfig mynet0 down
```

SIOCSIFFLAGS again, this time with IFF_UP cleared. Our ioctl handler sees IFF_DRV_RUNNING set but IFF_UP unset, so it calls mynet_stop(). That function flips running to false, clears IFF_DRV_RUNNING, drains the callout, and returns. Subsequent transmit attempts will be refused by mynet_transmit() because of the running check.

```
# ifconfig mynet0 destroy
```

SIOCIFDESTROY ioctl. The kernel finds the cloner that owns this interface, and calls mynet_clone_destroy(cloner, ifp). Our callback calls mynet_stop() (belt and braces: the interface was already down), then ether_ifdetach(), which calls if_detach() internally. if_detach() unlinks the interface from the global list, drains any references, calls bpfdetach(), notifies devd(8), and runs the ifnet_departure_event handlers. Our callback then calls ifmedia_removeall() to free the media list, destroys the mutex, frees the ifnet with if_free(), and frees the softc with free().

```
# kldunload mynet
```

The loader walks VNET_SYSUNINIT and calls vnet_mynet_uninit(), which detaches the cloner with if_clone_detach(). Then mynet_modevent(mod, MOD_UNLOAD, data) runs and returns zero. The loader unmaps the module from kernel memory. The system is clean.

Each command in the sequence corresponds to a specific callback in your driver. If a command hangs, the broken callback is usually obvious. If a command crashes, the stack trace points directly at it. Practise this trace until it feels mechanical; you will spend the rest of your career as a driver author walking variants of it.

28.24 Common Misconceptions About Network Drivers

Beginners arrive at this chapter with a handful of recurring misconceptions. Naming them explicitly helps you avoid subtle bugs later.

“The driver parses Ethernet headers.” Not quite. For reception, the driver does not parse the Ethernet header at all: it hands the raw frame to ether_input() (called from if_input() under the Ethernet framework), and ether_input() does the parsing. For transmission, ether_output() in the generic layer prepends the Ethernet header; your transmit callback usually sees the complete frame and simply moves its bytes out to the wire. The driver’s job is to move frames, not to understand protocols.

“The driver must know about IP addresses.” No. An Ethernet driver operates below IP entirely. It handles MAC addresses, frame sizes, multicast filters, and link state, but it never looks at IP headers. When you attach a network driver to an address using `ifconfig mynet0 192.0.2.1/24`, the assignment is stored in a protocol-family-specific structure (a `struct in_ifaddr`) which the driver never touches. The driver only sees outbound frames and only produces inbound frames: whether those frames carry IPv4, IPv6, ARP, or something exotic is above its pay grade.

“IFF_UP means the interface can send packets.” Partially true. `IFF_UP` means the administrator has said, “I want this interface to be active.” The driver responds by initialising hardware (or, in our case, flipping `running` to `true`) and setting `IFF_DRV_RUNNING`. The distinction matters. `IFF_UP` is user intent; `IFF_DRV_RUNNING` is driver state. Only the latter reliably indicates that the driver is prepared to send frames. If you check `IFF_UP` alone before transmitting, you will occasionally send frames into a half-initialised hardware state and watch the machine panic.

“BPF is something you enable for debugging.” BPF is always on for every network driver. The `bpfattach()` call inside `ether_ifattach()` registers the interface with the Berkeley Packet Filter framework unconditionally. When no BPF listeners exist, `BPF_MTAP()` is cheap; it checks an atomic counter and returns. When listeners exist, the mbuf is cloned and delivered to each one. You do not need to do anything special to make `tcpdump` work on your driver; you only need to call `BPF_MTAP()` on both paths. Forgetting that single call is the most common reason a new driver shows packets in counters but nothing in `tcpdump`.

“The kernel will clean up if my driver crashes.” False. A crash inside a driver is a crash inside the kernel. There is no process boundary to contain the damage. If your transmit function dereferences a null pointer, the machine panics. If your callback leaks a mutex, every subsequent call that touches the interface will hang. Write defensively. Test under load. Use `INVARIANTS` and `WITNESS` builds.

“Network drivers are slower than storage drivers.” Not inherently. Modern NICs process tens of millions of packets per second, and a well-written driver using `iflib(9)` can keep pace. The confusion comes from the fact that each individual packet is tiny compared to a storage request, so the per-packet overhead of clumsy designs becomes visible immediately. A sloppy storage driver might still hit 80% of line rate because a single I/O moves 64 KiB; a sloppy network driver will fall apart at 10% of line rate because each frame is 1.5 KiB and the per-frame overhead dominates.

“Once my driver passes `ifconfig`, I’m done.” Not even close. A driver that passes `ifconfig` up but fails under `jail`, `vnet`, or module unload can still break production systems. The rigorous test matrix you built in the testing section is the real bar. Many production bugs are discovered only at the intersection of features: VLAN plus TSO, jumbo frames plus checksum offload, promiscuous mode plus multicast filtering, rapid up/down cycles plus BPF listeners.

Each misconception can trace back to a piece of earlier reading that was technically accurate but incomplete. Now that you have written a driver, these edges come into focus.

28.25 How Packets Actually Reach and Leave Your Driver

It is worth slowing down and tracing the exact path a packet takes. The geography of the FreeBSD network stack is older than you might think, and much of it is invisible from the driver’s perspective. Knowing the geography makes the bugs you encounter easier to diagnose.

28.25.1 The outbound path

When a userland process calls `send()` on a UDP socket, the path looks like this:

1. The system call enters the kernel and reaches the socket layer. The data is copied from userland into kernel mbufs using `sosend()`.
2. The socket layer hands the mbufs to the protocol layer, which in this case is UDP. UDP prepends a UDP header and hands the packet to IP.
3. IP prepends the IP header, selects an output route using the routing table, and hands the packet to the interface-specific output function via the route's `rt_ifp` pointer. For an Ethernet interface, that function is `ether_output()`.
4. `ether_output()` calls `arpresolve()` to find the destination MAC address. If the ARP cache has an entry, execution continues. If not, the packet is queued inside ARP and an ARP request is transmitted; the queued packet will be released later when the reply arrives.
5. `ether_output()` prepends the Ethernet header and calls `if_transmit(ifp, m)`, which is a thin macro over the driver's `if_transmit` callback.
6. Your `mynet_transmit()` runs. It may queue the mbuf onto hardware, tap BPF, update counters, and free or retain the mbuf depending on whether it owns it.

Six layers, and only one of them is your driver. The rest is scenery you never have to touch. But when a bug occurs, understanding which layer might be responsible is the difference between a two-hour fix and a two-day fix.

28.25.2 The inbound path

For the receive side, the path runs in the opposite direction:

1. A frame arrives on the wire (or, in our case, is synthesised by the driver).
2. The driver builds an mbuf with `m_gethdr()`, populates `m_pkthdr.rcvif`, taps BPF with `BPF_MTAP()`, and calls `if_input(ifp, m)`.
3. `if_input()` is a thin wrapper that calls the interface's `if_input` callback. For Ethernet interfaces, `ether_ifattach()` sets this callback to `ether_input()`.
4. `ether_input()` examines the Ethernet header, looks up the Ethernet type (IPv4, IPv6, ARP, etc.), and calls the appropriate demux routine: `netisr_dispatch(NETISR_IP, m)` for IPv4, `netisr_dispatch(NETISR_ARP, m)` for ARP, and so on.
5. The netisr framework optionally defers the packet to a worker thread, then delivers it to the protocol-specific input routine. For IPv4, this is `ip_input()`.
6. IP parses the header, performs source/destination checks, consults the routing table to decide whether the packet is local or transit, and either delivers it up to the transport layer or sends it back down for forwarding.
7. If the packet is for the local host and the protocol is UDP, `udp_input()` validates the UDP checksum and delivers the payload to the matching socket's receive buffer.
8. The userland process that called `recv()` wakes up and reads the data.

Eight layers on reception, and again, only one of them is your driver. But look at how many places `m_pullup()` might be called to make a header contiguous in memory, how many places the mbuf might be freed, how many places a counter might be bumped. If you see `ifconfig mynet0` reporting packets received but `tcpdump -i`

mynet0 showing nothing, the gap is most likely between step 2 and step 3 (your `BPF_MTAP()` is missing or wrong). If `tcpdump` shows the packets but `netstat -s` shows them being dropped, the gap is most likely between step 6 and step 7 (the routing table doesn't think the interface owns the destination address).

28.25.3 Why this geography matters for driver authors

Understanding the geography gives you diagnostic power. When something breaks, you can ask focused questions. Is the counter incrementing? Step 6 in the outbound path fired. Is BPF seeing the packet? Your `BPF_MTAP()` call is present and the interface is marked running. Is the packet reaching the peer? The hardware actually transmitted it. Each question corresponds to a specific checkpoint in the geography, and each checkpoint narrows the range of possible bugs.

Production drivers extend this geography with transmit and receive rings, batch processing, hardware offloads, and interrupt moderation. Each optimisation changes the path; none of them changes the overall shape. The shape is worth memorising now, before the optimisations confuse it.

28.26 Things This Chapter Did Not Cover

An honest list of omissions helps you know what to learn next, and it sets up Chapter 29 more precisely than an artificial summary would.

Real hardware initialisation. We did not touch PCI enumeration, bus resource allocation, interrupt setup, or DMA ring construction. For that, read drivers like `/usr/src/sys/dev/e1000/if_em.c` carefully, especially `em_attach()` and `em_allocate_transmit_structures()`. You will see `bus_alloc_resource_any()`, `bus_setup_intr()`, `bus_dma_tag_create()`, and `bus_dmamap_create()` in action. Those are the functions that make a physical NIC actually move bits.

iflib. The `iflib(9)` framework abstracts most of the fiddly parts of a modern Ethernet driver. As a rough order-of-magnitude figure, a new NIC driver in FreeBSD 14.3 often consists of around 1,500 lines of hardware-specific code plus calls into `iflib`, rather than the roughly 10,000 lines of open-coded ring management a fully hand-written driver would need. We mentioned `iflib` without teaching it, because the teaching driver is simpler without it. A real driver in production in 2026 probably uses `iflib`.

Checksum offload. Modern NICs compute TCP, UDP, and IP checksums in hardware. Setting up `IFCAP_RXCSUM`, `IFCAP_TXCSUM`, and their IPv6 counterparts requires both driver support and mbuf flag manipulation (`CSUM_DATA_VALID`, `CSUM_PSEUDO_HDR`, etc.). Get it wrong and you silently corrupt traffic for some users only. The best introduction is `if_em.c`'s `em_transmit_checksum_setup()` function, paired with `ether_input()` to see how the flags propagate up.

Segmentation offload. TSO (transmit), LRO (receive), and GSO (generic segmentation offload) let the host hand the NIC multi-segment frames that the hardware (or a driver-level helper) breaks into MTU-sized fragments. For a primer, read `tcp_output()` and trace how it cooperates with `if_hwt_somax` and `IFCAP_TS04`.

Multicast filtering. Real drivers program hardware multicast hash tables based on the memberships advertised through `SIOCADDMULTI`. We stubbed the ioctls; a real implementation walks `ifp->if_multiaddrs` and pokes a hash register on the NIC.

VLAN processing. Real drivers set `IFCAP_VLAN_HWTAGGING` and allow `vlan(4)` to hand off tagging and untagging to the hardware. Without it, every VLAN-tagged frame goes through software `vlan_input()` and `vlan_output()`, slower but simpler. Our driver is VLAN-transparent: it carries tagged frames as-is.

Offload negotiation via SIOCSIFCAP. `ifconfig mynet0 -rxcsom` toggles capabilities at runtime. Real drivers must gracefully handle the capability changing mid-flight: flush rings, reconfigure hardware, then accept traffic again.

SR-IOV. Single-root I/O virtualisation lets a physical NIC present multiple virtual functions to a hypervisor. The FreeBSD support (`iovt(9)`) is non-trivial. We did not go near it.

Wireless. Wireless drivers use `net80211(4)`, a separate framework layered on top of `ifnet`. They have a rich state machine, complex rate control, encryption offload, and a completely different regulatory compliance story. Reading `/usr/src/sys/dev/ath/if_ath.c` is a worthwhile afternoon, but most of what it teaches is orthogonal to what we built here.

InfiniBand and RDMA. Out of scope entirely. These use `/usr/src/sys/ofed/` and a separate OS-agnostic verb framework.

Virtualisation-specific acceleration. `netmap(4)`, `vhost(4)`, and DPDK-style user-space fastpaths exist and matter in 2026 production environments. They are later-career topics.

We cover none of these in full. We gave you pointers to each, so that when your job demands one of them, you know where to begin reading.

28.27 Historical Context: Why ifnet Looks the Way It Does

A last stop before the bridge: a short history lesson. Understanding where `ifnet` came from makes some of its rough edges less surprising.

The earliest UNIX network stacks, in the late 1970s, had no `ifnet` structure at all. Each driver provided an ad hoc set of callbacks registered through patchy conventions. When 4.2BSD introduced the sockets API and the modern TCP/IP stack in 1983, the BSD team also introduced `struct ifnet` as a uniform interface between the protocol code and the driver code. The early version had about a dozen fields: a name, a unit number, a set of flags, an output callback, and a handful of counters. Compared to modern `struct ifnet`, it looks almost empty.

Over the next four decades, `struct ifnet` grew. BPF was added in the late 1980s. Multicast support arrived in the early 1990s. IPv6 support bolted on in the late 1990s. Interface cloning, media layer, and link-state events arrived through the 2000s. Offload capabilities, VNets, checksum offload flags, TSO, LRO, and VLAN offload appeared through the 2010s. By the time FreeBSD 11 arrived in 2016, the structure had become unwieldy enough that the project introduced the `if_t` opaque type and the `if_get*/if_set*` accessor functions, so that the structure's layout could change without breaking binary compatibility for modules.

That history explains several things. It explains why `ifnet` has both `if_ioctl1` and `if_ioctl2`; why some fields are accessed through macros and others directly; why `IFF_*` and `IFCAP_*` exist as parallel flag spaces; why the cloner API has both `if_clone_simple()` and `if_clone_advanced()`; why `ether_ifattach()` exists as a wrapper over `if_attach()`. Each addition solved a real problem. The accumulated weight is the cost of living inside a running system that never had the luxury of a clean slate.

For you, the practical takeaway is that `ifnet`'s surface area is large and a little inconsistent. Read it as geology, not as architecture. The strata record real events in the history of UNIX networking. Once you know they are strata, the inconsistencies become navigable.

28.28 Self-Assessment: Do You Really Know This Material?

Before moving on, measure your own understanding against a concrete rubric. A network driver author should be able to answer every question below without looking at the chapter. Work through them honestly. If you cannot answer one, re-read the relevant section; do not just skim until the answer looks familiar.

Conceptual questions.

1. What is the difference between `IFF_UP` and `IFF_DRV_RUNNING`, and which one decides whether a frame is actually sent?
2. Name three callbacks your driver must provide, and for each one, describe what failing to implement it correctly would cause.
3. Why does the kernel generate a random locally-administered MAC address for pseudo-interfaces, and what bit must be set to mark an address locally administered?
4. When `ether_input()` receives an Ethernet frame, which field of `m_pkthdr` tells the stack which interface the frame came from, and why does every inbound mbuf need it set correctly?
5. What does `net_epoch` protect, and why is it considered lighter weight than a traditional read lock?

Mechanical questions.

6. Write, from memory, the sequence of function calls from `if_alloc()` through `ether_ifattach()` that creates a minimum viable interface. You do not need to remember argument lists; just the names and the order.
7. Write the exact macro call that feeds an outbound mbuf to BPF. Write the exact macro call for inbound.
8. Given an mbuf chain that might be fragmented, which helper gives you a single flat buffer suitable for DMA? Which helper ensures at least the first *n* bytes are contiguous?
9. Which ioctl does `ifconfig mynet0 192.0.2.1/24` produce? Which layer in the kernel actually processes it: the generic `ifioctl()` dispatcher, `ether_ioctl()`, or your driver's `if_ioctl` callback? Why?
10. Your driver uses `callout_init_mtx(&sc->tick, &sc->mtx, 0)`. What is the purpose of the mutex argument, and what bug would arise if you passed `NULL`?

Debugging questions.

11. `ifconfig mynet0 up` returns instantly, but `netstat -in` shows the interface with zero packets after ten minutes of ping. Describe the three most likely causes and the commands you would run to distinguish between them.
12. The module loads cleanly. `ifconfig mynet create` succeeds. `ifconfig mynet0 destroy` panics with a "locking assertion" message. What is the most likely bug? How would you fix it?
13. `tcpdump -i mynet0` shows outbound packets but never inbound, even though `netstat -in` shows RX counters incrementing. Which function call is almost certainly missing, and on which code path?
14. You run `kldunload mynet` while an interface still exists. What happens? What safe sequence should the user have followed? How might a production driver refuse to unload under these conditions?

15. Running `ifconfig mynet0 up` followed immediately by `ifconfig mynet0 down` a hundred times in a loop causes the machine to panic on the fiftieth iteration with a corrupted mbuf queue. Walk through the likely class of bug and the fix.

Advanced questions.

16. Explain in your own words what `net_epoch` provides that a mutex does not, and when you would use one versus the other inside a network driver.
17. If your driver advertises `IFCAP_VLAN_HWTAGGING`, how does that change the mbufs your transmit callback sees compared to the default?
18. The kernel has two distinct delivery paths for inbound frames: one via `netisr_dispatch()`, one via direct dispatch. What are they, and when would a driver care which one is used?
19. What is the difference between `if_transmit` and the older `if_start/if_output` pair, and which should a new driver use?
20. Describe the lifecycle of a VNET on a FreeBSD 14.3 system, and explain why a cloner registered with `VNET_SYSINIT` produces a cloner in every VNET rather than a single global cloner.

If you answered every question without hesitation, you are ready for Chapter 29. If five or more questions gave you trouble, spend another session with this chapter before moving forward. The next chapter builds on the assumption that you know this material cold.

28.29 Further Reading and Source Code Study

The bibliography below is small, targeted, and ordered by usefulness for a driver author at your current stage. Treat it as a reading list for the weeks after you finish Chapter 28, not an overwhelming bookshelf.

Must read, in order.

- `/usr/src/sys/net/if.c`: the generic interface machinery. Start with `if_alloc()`, `if_attach()`, `if_detach()`, and the ioctl dispatcher `ifioctl()`. This is the file that actually runs the lifecycle functions you call in your driver.
- `/usr/src/sys/net/if_etherubr.c`: Ethernet framing. Read `ether_ifattach()`, `ether_ifdetach()`, `ether_output()`, `ether_input()`, and `ether_ioctl()`. These four functions form the contract between your driver and the Ethernet layer.
- `/usr/src/sys/net/if_disc.c`: the minimal pseudo-driver. Less than 200 lines. A reference for the absolute minimum viable `ifnet`.
- `/usr/src/sys/net/if_epair.c`: the paired pseudo-driver. The cleanest reference for writing a cloner with a shared structure between two instances.
- `/usr/src/sys/dev/virtio/network/if_vtnet.c`: a modern paravirtual driver. Small enough to read fully, realistic enough to teach you about rings, checksum offload, multiqueue, and hardware-like resource management.

Read next, when the time comes.

- `/usr/src/sys/dev/e1000/if_em.c` and the accompanying `em_txrx.c`, `if_em.h`: a production Intel NIC driver. Larger and more elaborate, but representative of real-world driver complexity.
- `/usr/src/sys/net/iflib.c` and `/usr/src/sys/net/iflib.h`: the `iflib` framework. Read after you have studied `if_em.c` so you can recognise the structures `iflib` takes over.

- `/usr/src/sys/net/if_lagg.c`: the link aggregation driver. A detailed study of multi-interface orchestration, failover, and mode selection.
- `/usr/src/sys/net/if_bridge.c`: software bridging. Excellent for learning multicast forwarding, learning bridges, and the STP state machine.

Worthwhile manual pages.

- `ifnet(9)`: the interface framework.
- `mbuf(9)`: the packet buffer system.
- `bpf(9)` and `bpf(4)`: the Berkeley Packet Filter.
- `ifmedia(9)`: the media framework.
- `ether(9)`: Ethernet helpers.
- `vnet(9)`: virtualised network stacks.
- `net_epoch(9)`: the network epoch synchronisation primitive.
- `iflib(9)`: the `iflib` framework.
- `netmap(4)`: high-speed user-space packet I/O.
- `netgraph(4)` and `netgraph(3)`: the `netgraph` framework.
- `if_clone(9)`: interface cloning.

Books and papers.

The 4.4BSD design books (particularly “The Design and Implementation of the 4.4BSD Operating System” by McKusick, Bostic, Karels, and Quarterman) remain the best long-form explanation of how the sockets and interface layers came to exist. The FreeBSD Developer’s Handbook sections on kernel programming and loadable modules are the next stop for general background. For packet processing at speed, the `netmap` papers by Luigi Rizzo are foundational; they explain both the techniques and the rationale behind modern high-performance packet pipelines.

Keep a reading journal. When you finish a file, write a paragraph summarising what surprised you, what you want to revisit, and what you think you might steal for your own drivers. Over six months of this practice, your intuition for how production drivers are structured will grow faster than you expect.

28.30 Frequently Asked Questions

New driver authors tend to ask the same questions as they work through their first `ifnet` driver. Here are the most common ones, with short, pointed answers. Each answer is a signpost, not an exhaustive treatment; follow the bread crumbs back to the relevant section of the chapter if you want more detail.

Q: Can I write an Ethernet driver without using `ether_ifattach`?

Technically yes; practically no. `ether_ifattach()` sets `if_input` to `ether_input()`, hooks BPF with `bpfattach()`, and configures a dozen small default behaviours. Skipping it means reimplementing every one of those defaults by hand. The only reason to bypass `ether_ifattach()` is if your driver is not actually Ethernet, in which case you will use `if_attach()` directly and supply your own framing callbacks.

Q: What is the difference between `if_transmit` and `if_output`?

`if_output` is the older, protocol-agnostic output callback. For Ethernet drivers, it is set to `ether_output()` by `ether_ifattach()`, and it handles ARP resolution and Ethernet framing before calling `if_transmit`.

`if_transmit` is the driver-specific callback you write. In short: `if_output` is what the stack calls; `if_transmit` is what `if_output` calls; your driver supplies the latter.

Q: Do I need to handle `SIOCSIFADDR` in my `ioctl` callback?

Not directly. `ether_ioctl()` handles address configuration for Ethernet interfaces. Your callback should delegate unrecognised `ioctls` to `ether_ioctl()` via the default: arm of its switch statement, and address-related `ioctls` will flow through that path correctly.

Q: How do I know when a frame has actually been transmitted by the hardware?

For our pseudo-driver, “transmit” is synchronous: `mynet_transmit()` frees the mbuf immediately. For a real NIC driver, the hardware signals completion via an interrupt or a ring descriptor flag; the driver’s transmit completion handler (sometimes called the “tx reaper”) walks the ring, releases the mbufs, and updates counters. Read `if_em.c`’s `em_txeof()` for a concrete example.

Q: Why does `ifconfig mynet0 delete` not call my driver?

Because address configuration lives in the protocol layer, not the interface layer. Removing an address from an Ethernet interface is handled by `in_control()` (for IPv4) or `in6_control()` (for IPv6). Your driver is unaware of these operations; it only sees them indirectly through route changes and ARP table updates.

Q: Why is my driver panicking when I call `if_inc_counter()` from a callout?

Almost certainly because you are holding a non-recursive mutex that was acquired elsewhere. `if_inc_counter()` is safe from any context on modern FreeBSD, but if your callout acquires a lock that the callout infrastructure already holds, you deadlock. The safest pattern is to call `if_inc_counter()` without holding any driver-specific locks, and to update your own counters separately inside the lock.

Q: How do I make my driver show up in `sysctl net.link.generic.ifdata.mynet0.link`?

You do not. That `sysctl` tree is populated automatically by the generic `ifnet` layer. Every interface registered via `if_attach()` (directly or via `ether_ifattach()`) receives a `sysctl` node. If yours is missing, your interface did not attach correctly.

Q: My driver works on FreeBSD 14.3 but fails to build on FreeBSD 13.x. Why?

The `if_t` opaque type and the associated accessor functions were stabilised across FreeBSD 13 and 14, but several helper APIs only arrived in 14. For example, `if_clone_simple()` has existed for years, but some of the counter accessor helpers are new. Either use `__FreeBSD_version` guards to compile cleanly on both, or state clearly in your driver that FreeBSD 14.0 or later is required.

Q: I want to write a driver that accepts packets on one interface and retransmits them on another. Is that a network driver?

Not quite. That is a bridge or a forwarder. The FreeBSD kernel has `if_bridge(4)` for bridging, `netgraph(4)` for arbitrary packet pipelines, and `pf(4)` for filtering and policy. Writing your own forwarding code from scratch is almost never the right answer in 2026; the existing frameworks are better-maintained, faster, and more flexible. Read them and configure them before writing a new driver.

Q: Do I need to worry about endianness inside my network driver?

Only at specific boundaries. Ethernet frames are network byte order (big-endian) by convention; if you parse an Ethernet header yourself, the `ether_type` field needs `ntohs()`. Inside the mbuf, the data is stored in network byte order, not the host’s native order. The `ether_input()` and `ether_output()` functions handle the conversions for you, so most driver code does not touch endianness directly.

Q: When do I use `m_pullup()` versus `m_copydata()`?

`m_pullup(m, n)` mutates the mbuf chain so that the first `n` bytes are stored contiguously in memory, making them safe to access with a pointer cast. `m_copydata(m, off, len, buf)` copies bytes out of the mbuf chain into a separate buffer you provide. Use `m_pullup()` when you want to read and potentially modify header fields in place. Use `m_copydata()` when you want a snapshot for inspection without disturbing the mbuf.

Q: Why does `netstat -I mynet0 1` sometimes show zero bytes even when packets are being exchanged?

You may be incrementing `IFCOUNTER_IPACKETS` or `IFCOUNTER_OPACKETS` without also incrementing `IFCOUNTER_IBYTES` or `IFCOUNTER_OBYTES`. The per-second display shows bytes separately; if the byte counters never move, `netstat -I` reports zero throughput. Always update packet count and byte count together.

Q: How do I destroy all cloned interfaces on module unload?

The simplest approach is to let `if_clone_detach()` do it for you; the clone detach helper walks the cloner's interface list and destroys each one. If you want to defend against leaks, you can also enumerate the interfaces belonging to the cloner and destroy them explicitly before calling `if_clone_detach()`. The shorter path is usually the better one, because the helper is tested and yours probably is not.

Q: My driver works under ping but crashes during a large iperf3 run. What is typical?

At high packet rates, all the subtle concurrency bugs in a driver become exposed. Common causes include: a counter updated outside a lock that runs on multiple CPUs, a mbuf queue that is not properly drained before free, a callout that fires during shutdown, a `BPF_MTAP()` call after the interface was detached. Run with `WITNESS` and `INVARIANTS` enabled; the locking assertions almost always catch it.

28.31 A Short Closing Note on Craft

We have spent many pages on mechanics: callbacks, locks, mbufs, ioctls, counters. The mechanics are necessary, but they are not sufficient. A good network driver is the product of a disciplined author, not merely a correct set of callbacks.

That discipline shows up in small places. It shows up in the decision to drain a callout on detach even though the test suite never catches a leak. It shows up in the decision to update a counter in the right order so that `netstat -s` adds up across a long run. It shows up in the decision to log once, clearly, when a resource cannot be allocated, rather than either staying silent or flooding the log. It shows up in the decision to use `M_ZERO` when allocating a softc, so that any future field added to the structure starts at a known zero even if the explicit initialisation is forgotten.

Each decision is small. The accumulated effect is the difference between a driver that works on the first day and a driver that works on the thousandth day. You are training a habit, not memorising a syntax. Be patient with yourself while the habit forms; it takes years.

The great FreeBSD driver authors, the ones whose names you see in the `$FreeBSD$` tags and the commit logs, did not become great by knowing the API better than you. They became great by reviewing their own work as if someone else wrote it, and fixing every small flaw they found. That practice scales. Pick it up early.

28.32 Mini-Glossary of Network Driver Terms

A short glossary follows, aimed at the reader who wants to revisit the chapter's core vocabulary in one place. Use it as a refresher, not as a replacement for the explanations in the main text.

- **ifnet.** The kernel data structure representing a network interface. Each attached interface has exactly one `ifnet`. The opaque handle `if_t` is used by most modern code.
- **ether_ifattach.** The wrapper over `if_attach()` that sets up Ethernet-specific defaults, including BPF hooks and the standard `if_input` function.
- **cloner.** A factory for pseudo-interfaces. Registered with `if_clone_simple()` or `if_clone_advanced()`. Responsible for creating and destroying interfaces in response to `ifconfig name create` and `ifconfig name0 destroy`.
- **mbuf.** The kernel's packet buffer. A small struct with metadata, an optional embedded payload, and pointers to additional buffers for chained data. Allocated with `m_gethdr()`, freed with `m_freem()`.
- **softc.** Per-instance driver state. Allocated with `malloc(M_ZERO)` in the cloner create callback and freed in the cloner destroy callback. Traditionally points to mutex, media descriptor, callout, and the interface.
- **BPF.** The Berkeley Packet Filter, a framework for userland tools like `tcpdump` to observe the traffic on an interface. Drivers hook into it with `BPF_MTAP()` on both transmit and receive paths.
- **IFF_UP.** The administrative flag set by `ifconfig name0 up`. Indicates user intent to activate the interface.
- **IFF_DRV_RUNNING.** The driver-controlled flag indicating the driver is prepared to send and receive packets. Set inside the driver after hardware (or pseudo-hardware) initialisation completes.
- **Media.** The abstraction for link speed, duplex, autonegotiation, and related physical-layer properties. Managed through the `ifmedia(9)` framework.
- **Link state.** A three-valued indicator (`LINK_STATE_UP`, `LINK_STATE_DOWN`, `LINK_STATE_UNKNOWN`) reported via `if_link_state_change()`. Used by routing daemons and userland tools.
- **VNET.** FreeBSD's virtualised network stack. Each VNET has its own interface list, routing table, and sockets. Pseudo-drivers typically use `VNET_SYSINIT` to register cloners in every VNET.
- **net_epoch.** A lightweight synchronisation primitive used to delimit read-side critical sections in the network stack. Faster than a traditional read lock.
- **IFCAP.** A bitfield of capabilities (`IFCAP_RXCSUM`, `IFCAP_TS04`, etc.) negotiated between the driver and the stack. Controls which offloads are active on a given interface.
- **IFCOUNTER.** A named counter (`IFCOUNTER_IPACKETS`, `IFCOUNTER_OBYTES`, etc.) displayed by `net-stat`. Updated by drivers via `if_inc_counter()`.
- **Ethernet type.** The 16-bit field in an Ethernet frame header identifying the encapsulated protocol. Values are defined in `net/ethernet.h`, with `ETHERTYPE_IP` and `ETHERTYPE_ARP` being the most common.
- **Jumbo frame.** An Ethernet frame larger than the standard 1500-byte MTU, typically 9000 bytes. Drivers advertise support through `ifp->if_capabilities |= IFCAP_JUMBO_MTU`.
- **Promiscuous mode.** A mode in which the interface delivers every observed frame to the stack, not only those addressed to its own MAC. Controlled via `IFF_PROMISC`. Used by network analysis tools.
- **Multicast.** Frames addressed to a group of receivers rather than a single destination. Drivers track group memberships through `SIOCADDMULTI` and `SIOCDELMULTI`, typically programming a hardware hash filter.
- **Checksum offload.** A capability where the NIC computes TCP, UDP, and IP header checksums in hardware. Negotiated through `IFCAP_RXCSUM` and `IFCAP_TXCSUM`; flagged per mbuf via `m_pkthdr.csum_flags`.

- **TSO (TCP Segmentation Offload).** A capability where the host hands the NIC a large TCP segment and the NIC splits it into MTU-sized fragments. Negotiated via `IFCAP_TS04` and `IFCAP_TS06`.
- **LRO (Large Receive Offload).** The receive-side counterpart to TSO. The NIC or software layer aggregates sequential inbound segments into a single large mbuf chain before handing it to the stack.
- **VLAN tagging.** A four-byte shim in the Ethernet frame that identifies the VLAN membership. Drivers may advertise `IFCAP_VLAN_HWTAGGING` to offload the insertion and removal to hardware.
- **MSI-X.** Message-signalled interrupts, the modern replacement for wired IRQs. Allows the NIC to raise separate interrupts per queue.
- **Interrupt moderation.** A technique where the NIC coalesces multiple completion events into fewer interrupts, reducing overhead at high packet rates.
- **Ring buffer.** A circular queue of descriptors shared between the driver and the NIC. Transmit rings feed packets to hardware; receive rings deliver packets from hardware.
- **iflib.** FreeBSD's modern NIC driver framework. Abstracts ring management, interrupt handling, and mbuf flow so the driver author can focus on hardware-specific code.
- **netmap.** A high-performance packet I/O framework that gives userland direct access to driver rings, bypassing most of the network stack.
- **netgraph.** A flexible framework for composing packet-processing pipelines out of reusable nodes. Largely orthogonal to driver writing but often relevant to network architecture.
- **pf.** FreeBSD's packet filter. A firewall and NAT engine that sits inline with `ether_input()` and `ether_output()` via `pf1l(9)` hooks. Drivers do not interact with it directly; the hooks are inserted by the generic layers.
- **pfil.** The packet filter interface through which firewalls attach to the forwarding path. Gives frameworks like `pf` and `ipfw` a stable place to observe and modify packets.
- **if_transmit.** The per-driver outbound callback, set during interface allocation. Receives an mbuf chain, is responsible for queueing it for hardware or discarding it.
- **if_input.** The per-interface inbound callback. For Ethernet drivers, set to `ether_input()` by `ether_ifattach()`. The driver calls it via the `if_input(ifp, m)` helper to hand received frames up to the stack.
- **if_ioctl.** The per-driver ioctl callback. Handles interface-level ioctls such as `SIOCSIFFLAGS`, `SIOC-SIFMTU`, and `SIOCSIFMEDIA`. Delegates unknown ioctls to `ether_ioctl()` for Ethernet drivers.

Keep this glossary nearby as you read Chapter 29 and the chapters that follow. Each term recurs often enough that a quick reference pays for itself.

28.33 Part 6 Checkpoint

Part 6 put the discipline of Parts 1 through 5 under three very different transports: USB, GEOM-backed storage, and `ifnet`-based network. Before Part 7 returns to the cumulative `myfirst` arc and starts pushing on portability, security, performance, and craft, confirm that the three transport vocabularies have settled into the same underlying model.

By the end of Part 6 you should be able to do each of the following:

- Attach to a USB device through the `usb_request_methods` framework: configure transfers for control, bulk, interrupt, and isochronous endpoints; dispatch read and write through the transfer callbacks; and survive hot-plug and hot-unplug as normal operating conditions.

- Write a storage driver that plugs into GEOM: provision a provider through `g_new_providerf`, service BIO requests in the class's start routine, walk the `g_down/g_up` threads in your head, and tear down cleanly under mounted load.
- Write a network driver that presents an `ifnet` through `ether_ifattach`: implement `if_transmit` for the outbound path, call `if_input` for the inbound path, integrate with `bpf` and media state, and clean up through `ether_ifdetach`.
- Explain why the three transports look so different at the surface but share the same underlying discipline from Parts 1 through 5: Newbus attach, softc management, resource allocation, locking, detach ordering, observability, production discipline.

If any of those still feels unsteady, the labs to revisit are:

- USB path: Lab 2 (Building and Loading the USB Driver Skeleton), Lab 3 (A Bulk Loopback Test), Lab 6 (Observing Hot-Plug Lifecycle), and Lab 7 (Building a `ucom(4)` Skeleton from Scratch) in Chapter 26.
- GEOM storage path: Lab 2 (Build the Skeleton Driver), Lab 3 (Implement the BIO Handler), Lab 4 (Increase Size and Mount UFS), and Lab 10 (Break It On Purpose) in Chapter 27.
- Network path: Lab 1 (Build and Load the Skeleton), Lab 2 (Exercise the Transmit Path), Lab 3 (Exercise the Receive Path), Lab 5 (`tcpdump` and `BPF`), and Lab 6 (Clean Detach) in Chapter 28.

Part 7 expects the following as a baseline:

- Comfort switching between `cdevsw`, GEOM, and `ifnet` as three idioms over the same Newbus-and-softc core, rather than as three disconnected subjects.
- Understanding that Part 7 returns to the single-thread `myfirst` arc for the final polish on portability, security, performance, tracing, kernel-debugger work, and the craft of engaging with the community. The transport-specific demos of Part 6 do not continue; their lessons do.
- A mental library of three real transports you have touched with your own hands, so that when Chapter 29 talks about abstraction across backends, you are drawing on experience rather than on examples you have only read about.

If those hold, Part 7 is ready for you. The final nine chapters are the part of the book that turns a capable driver author into a craftsperson; the groundwork Parts 1 through 6 have laid is what makes that transition possible.

28.34 Looking Ahead: Bridge to Chapter 29

You have just written a network driver. The next chapter, **Portability and Driver Abstraction**, zooms out from the concrete details you have mastered and asks: how do we write drivers that work well across FreeBSD's many supported architectures, and how do we structure driver code so that parts of it can be reused across different hardware backends?

That question is sharper after Chapter 28 than it was before. You have now written drivers for three very different subsystems: character devices on top of `cdevsw`, storage devices on top of GEOM, and network devices on top of `ifnet`. The three look different on the surface, but they share a surprising amount of plumbing: probe and attach, softc allocation, resource management, lifecycle control, unloading cleanliness. Chapter 29 will turn that observation into a practical refactoring: isolating the hardware-dependent code, separating backends behind a common API, preparing the driver to compile on x86, ARM, and RISC-V alike.

You will not be writing a new kind of driver in Chapter 29. You will be learning how to make the drivers you already wrote more robust, more portable, and more maintainable. That is a different kind of progress, one that matters the moment you start working on a driver that will live for years.

Before you move on, unload every module you created in this chapter, destroy every interface, and make sure `netstat -in` is back to a boring baseline. Close your lab logbook with a brief note on what worked and what puzzled you. Rest your eyes for a minute. Then, when you are ready, turn the page.

You have earned the step.

29 Portability and Driver Abstraction

29.1 Introduction

You arrive at Chapter 29 with a particular kind of experience. You have written three drivers that look very different on the surface. In Chapter 26 you connected a character device to `cdevsw` and taught `/dev` how to reach your code. In Chapter 27 you fed blocks into GEOM and let a filesystem sit on top. In Chapter 28 you shaped a network interface, registered it with the stack, and watched packets flow back and forth. Each of those drivers was cut to fit a specific subsystem, and each one taught you a specific contract with the FreeBSD kernel. If you read the three side by side, you would notice that they also share a great deal: a `softc`, a probe and attach path, resource allocation, teardown, careful handling of concurrency, and attention to the load and unload lifecycle.

This chapter is about that shared skeleton. More importantly, it is about what happens to that skeleton when the rest of the world changes around it. The machine you built your driver on today is unlikely to be the only machine that ever runs it. A driver written for an x86 workstation in 2026 might be compiled for an ARM server in 2027, for a RISC-V embedded board in 2028, and for a FreeBSD 15 release in 2029. Along the way, the hardware it talks to might change from a PCI card to a USB dongle to a simulated device in `bhyve`. The kernel API might evolve. The surrounding system might change from FreeBSD to NetBSD for one downstream consumer. The code that started small and single-purpose has to become something that survives that turbulence with minimal surgery.

That survival is what we mean by **portability**. Portability is not one property. It is a small family of related properties. It includes the ability to be compiled on a CPU architecture that you did not originally target. It includes the ability to attach to a different physical bus, such as PCI one day and USB another. It includes the ability to switch between a real device and a simulated stand-in during testing without rewriting half the file. It includes the ability to coexist with kernel API churn, so that a rename or a deprecation in a future release of FreeBSD does not force you to throw away your driver. And, occasionally, it includes the ability to share code with cousins of FreeBSD like NetBSD and OpenBSD without turning the whole codebase into a compatibility maze.

Chapter 29 teaches you how to design, structure, and refactor a driver so that each of those kinds of change becomes a local problem rather than a global rewrite. The chapter is not about a new subsystem. You have already met them. It is about the engineering discipline that allows a driver to age gracefully. It is the part of driver authorship that does not show up on the first day, when the module loads cleanly and the device responds, but that matters enormously on day one thousand, when the same module has to ship on three architectures, two buses, and four FreeBSD releases while staying readable to new contributors.

We will not invent a new driver from scratch. Instead, we will take a driver shape that is familiar to you and evolve it. You will learn how to separate hardware-dependent code from logic that does not care about hardware. You will learn how to hide a physical bus behind a backend interface, so that a PCI variant and a USB variant can share the same core. You will learn how to split a single `.c` file into a small family of files that

each have a clear responsibility. You will learn how to use the kernel's endian helpers, fixed-width types, and `bus_space/bus_dma` abstractions so that your register accesses work on little-endian and big-endian machines alike, and on 32-bit as well as 64-bit word sizes. You will learn how to use conditional compilation without turning the source into a thicket of `#ifdef`. And you will learn how to step back and version the driver so that users of your code know what it supports and what it does not.

Through all of this, the tone remains the same as in earlier chapters. Portability looks abstract when you read about it in a textbook, but it is concrete when you sit in front of a driver source file and decide which function goes where. We will stay close to that second view. Every pattern you meet will be tied to a real FreeBSD practice, and often to a real file under `/usr/src/` that you can open and read alongside the text.

Before you begin, resist the temptation to read this chapter as a set of rules. It is not. It is a set of **habits**. Each habit exists because a future version of you will thank the current version of you for having formed it. A driver that never has to be ported will still benefit from being written as if it might be, because the same habits that help portability also help readability, testability, and long-term maintenance. Do not treat this chapter as optional polish. Treat it as the part of craft that separates a driver that works from a driver that keeps working.

29.2 Reader Guidance: How to Use This Chapter

This chapter is a little different from the three that came before it. Chapters 26, 27, and 28 were each built around a concrete subsystem with a concrete API. Chapter 29 is built around a way of thinking. You will still read code, and you will still do labs, but the subject is how driver code is **organised**, not which kernel function it calls. That distinction affects the best way to study it.

If you choose the **reading-only path**, plan for roughly two to three focused hours. By the end you will recognise the common structural patterns that mature FreeBSD drivers use to stay portable and will know what to look for when you open an unfamiliar driver source tree. This is a legitimate way to use the chapter on a first pass, because the ideas here become more useful once you have seen them in action in your own work.

If you choose the **reading-plus-labs path**, plan for about five to eight hours across one or two sessions. You will take a small reference driver and progressively refactor it into a portable, multi-backend shape. You will split a single file into a core and two backend files, you will introduce a backend interface, and you will make the driver compile cleanly with and without optional backends enabled. The labs are deliberately incremental so that each step stands on its own and leaves you with a working module rather than a broken one.

If you choose the **reading-plus-labs-plus-challenges path**, plan for a weekend or a few evenings. The challenges push the refactoring further: adding a third backend, exposing version metadata to userland, simulating a big-endian environment, and writing a build matrix that proves the driver compiles in every documented configuration. Each challenge stands alone and uses only material from this chapter and the earlier ones.

You should continue to work on a throwaway FreeBSD 14.3 machine, as in earlier chapters. The labs do not require special hardware, a complicated bus, or a particular NIC. They work on a plain virtual machine with the source tree installed, a working compiler, and the usual kernel module build tools. A snapshot before you begin is cheap, and the only thing you need to undo a mistake is the ability to reboot.

A word on prerequisites. You should be comfortable with everything from Chapters 26, 27, and 28: writing a kernel module, allocating a softc, managing the probe and attach lifecycle, registering with `cdevsw`, `GEOM`, or `ifnet` as appropriate, and reasoning about the load and unload path. You should also be comfortable with `make`

and with the FreeBSD `bsd.kmod.mk` build system, since a good deal of the chapter is about how to split and compile a multi-file driver. If any of that feels uncertain, a quick skim of the earlier chapters will save you time here.

Finally, do not skip the troubleshooting section near the end. Portability refactors fail in a handful of characteristic ways, and learning to recognise those patterns early is more useful than memorising which macro lives in which header. A portable driver that mostly works is worse than a non-portable one that definitely works, because the portability gives you a false sense of confidence. Treat the troubleshooting material as part of the lesson.

29.2.1 Work Section by Section

The chapter is structured as a deliberate progression. Section 1 defines what portability really means for a driver, and why we care. Section 2 walks through isolating hardware-dependent code from logic that is not. Section 3 teaches how to hide backends behind an interface so that the core driver does not need to know whether it is talking to a PCI card or a USB dongle. Section 4 organises those pieces into a clear file layout. Section 5 deals with architectural portability: endianness, alignment, and word size. Section 6 teaches disciplined use of conditional compilation and build-time feature selection. Section 7 takes a brief look at cross-BSD compatibility, enough to be useful without turning the chapter into a NetBSD porting manual. Section 8 steps back and addresses refactoring and versioning for long-term maintenance.

You are meant to read these sections in order. Each one assumes the previous ones are fresh in your mind, and each lab builds on the results of the one before it.

29.2.2 A Gentle Warning About Over-Engineering

Before we dive in, a word of caution. Portability patterns can become an end in themselves if you are not careful. It is possible to build a driver so layered, so abstracted, and so wrapped that no one, including you, can follow what actually happens on a given code path. The goal of this chapter is not to maximise abstraction; it is to choose the right level of abstraction for the driver at hand.

For a small driver with one backend and no prospect of a second one, the simplest portable design is a single `.c` file that uses `bus_read_*`, endian helpers, and fixed-width types throughout. No backend interface. No file split. The investment stops there, and that is enough. The refactor to a multi-backend layout can happen later, when the need appears.

For a driver that already supports two or three variants, the full multi-file layout earns its keep, and the refactor to get there saves more time than it costs. For a driver that might one day support additional variants but does not yet, the middle ground is a single file with clean internal separation (a backend interface as a struct, even if only one backend is instantiated), ready to be split into multiple files when the second backend arrives.

Matching the complexity of the design to the complexity of the problem is the craft. Read this chapter with that in mind. Not every pattern in it belongs in every driver. The patterns belong where they pay for themselves.

29.2.3 Keep the Reference Driver Close

The labs in this chapter centre on a small reference driver called `portdrv`. You will find it under `examples/part-07/ch29-portability/`, staged in the same way as the examples from earlier chap-

ters. Each lab directory contains the state of the driver at that step, along with the Makefile and any support files needed to compile and test it. Do not just read the lab; type the changes, build the module, and load it. The feedback loop is tight because refactoring either compiles or it does not, and running the module after each step confirms that you did not accidentally break the driver while moving code around.

29.2.4 Open the FreeBSD Source Tree

Several sections in this chapter point to real FreeBSD drivers as concrete examples of portable structure. The files of interest include `/usr/src/sys/dev/uart/uart_core.c`, `/usr/src/sys/dev/uart/uart_bus_pci.c`, `/usr/src/sys/dev/uart/uart_bus_fdt.c`, `/usr/src/sys/dev/uart/uart_bus_acpi.c`, `/usr/src/sys/dev/uart/uart_bus.h`, `/usr/src/sys/modules/uart/Makefile`, `/usr/src/sys/sys/_endian.h`, `/usr/src/sys/sys/bus_dma.h`, and the per-architecture bus headers under `/usr/src/sys/amd64/include/_bus.h`, `/usr/src/sys/arm64/include/_bus.h`, `/usr/src/sys/arm/include/_bus.h`, and `/usr/src/sys/riscv/include/_bus.h`. Open them as the text points to them. Their shape is half the lesson.

29.2.5 Keep the Lab Logbook

Continue the lab logbook you started in Chapter 26. For this chapter, log a short entry at the end of each lab: which files you edited, which build options you enabled, which backends compiled, and any warnings the compiler produced. Portability is easy to claim and hard to verify, and a paper trail of your experiments is the fastest way to notice when you have quietly regressed.

29.2.6 Pace Yourself

If your understanding blurs during a particular section, stop. Re-read the previous subsection. Try a small experiment, for example recompiling the driver with one backend disabled and confirming that it still builds. Portability refactors reward slow, deliberate steps. A hurried split that leaves a function in the wrong file is harder to fix than an unhurried one that places it correctly the first time.

29.3 How to Get the Most Out of This Chapter

Chapter 29 is not a reference list. It is a workshop about structure. The most effective way to use it is to keep the driver under your hands while you read, and to ask of every pattern you meet: what problem does this solve, and what would happen if the problem did not exist? That question is the fastest path to internalising the habits.

29.3.1 Refactor in Small Steps

Do not try to go from a single-file driver to a fully portable, multi-backend, cross-BSD-compatible architecture in one sitting. The labs are broken into discrete steps for a reason. Each step makes exactly one change to the code and confirms that the module still loads cleanly after it. That discipline mirrors real-world practice. Professional driver refactors are rarely large, dramatic rewrites; they are a long sequence of small, reviewable moves, each of which can be reverted if something goes wrong.

29.3.2 Read the Compiler Output

The compiler is your collaborator during a portability refactor. A missed include, a stale prototype, a function moved to a file that does not list it in SRCS, these become compiler errors the moment you build, and the messages are precise if you read them carefully. When the build fails, resist the urge to guess; open the file the compiler is pointing at and fix the specific symbol it cannot find. Over a chapter's worth of labs, you will develop a feel for how the build system reacts to different kinds of reorganisation.

29.3.3 Use Version Control as a Safety Net

Commit before each refactoring step. The project's git history gives you a zero-cost way to step back to a known good state. If a split goes wrong, reverting the change is a single command, not an hour of hunting for what you broke. You do not need to push these commits anywhere; they are for you.

29.3.4 Compare Against Real FreeBSD Drivers

When the chapter points to a real file under `/usr/src/sys/`, open it. The `uart(4)` driver in particular is a superb study in backend abstraction: its core logic in `uart_core.c` knows nothing about the bus, and its backend files add PCI, FDT, and ACPI attachment with a small amount of code each. Reading those files alongside this chapter will teach you what "good" looks like far better than any synthetic example could.

29.3.5 Take Notes on Patterns, Not Syntax

The specific macros and function names in this chapter are less important than the patterns behind them. A backend interface defined as a struct of function pointers is a pattern you will use for the rest of your career; the particular spelling of the functions in `portdrv` is a one-time lesson. Write down the patterns in your logbook. Revisit them when you design your own drivers.

29.3.6 Trust the Build System

FreeBSD's `bsd.kmod.mk` is mature, and it does a great deal of work for you. A multi-file driver is not significantly more complicated to build than a single-file driver; you add a line to SRCS and the build keeps working. Lean on that. Do not write your own Makefile machinery when the existing one does the job.

29.3.7 Take Breaks

Refactoring work has a particular cognitive cost. You are holding two versions of the code in your head at once, the one before the move and the one after, and your mind gets fatigued more quickly than during new development. Two or three focused hours are usually more productive than an uninterrupted seven-hour sprint. If you notice yourself copy-pasting without reading, stand up for ten minutes.

With those habits in place, let us begin.

29.4 Section 1: What Does “Portable” Mean for a Device Driver?

Portability is one of those words that everyone uses and few people pin down. When someone says a driver is portable, they might mean any of half a dozen different things. Before we can talk about how to achieve it, we need to be clear about what we are actually trying to achieve. This section takes that step carefully, because every later section depends on sharing the same vocabulary.

29.4.1 A Working Definition

For this book, a driver is portable when changing one of the following does not force a change in the driver’s core logic:

- the CPU architecture it is compiled for
- the bus through which it reaches the hardware
- the specific revision of the hardware behind the bus
- the version of FreeBSD it is running on, within reasonable limits
- the physical deployment, such as real hardware versus a simulator

Notice that this definition is negative. It does not say that a portable driver runs everywhere without modification, because no driver does. It says that when any of these elements change, the change is **local**. The core logic stays where it was. The hardware-specific or platform-specific code absorbs the change. That locality is the practical meaning of portability, and everything in this chapter is about how to arrange the code so that change is local.

A driver that is not portable will usually reveal itself in three ways. First, a small change in the environment produces diffs scattered across the entire file. Second, the author has to read the whole file to know where to make a change, because information about the bus or the architecture is sprinkled throughout the logic. Third, adding support for a new backend requires either duplicating the entire driver or carving `#ifdef` blocks through every function. If you have ever tried to support a second variant of a device this way, you already know how fragile the result becomes.

29.4.2 Portability Across Architectures

FreeBSD runs on a wide variety of CPU architectures, and all of them can load kernel modules. The ones you are most likely to encounter in practice are `amd64` (the 64-bit x86 family), `i386` (legacy 32-bit x86), `arm64` (the 64-bit ARM family), `armv7` (the 32-bit ARM family), `riscv` (both 32-bit and 64-bit RISC-V), and `powerpc` and `powerpc64` for older and server-class PowerPC systems. A driver that is portable across architectures compiles and runs correctly on all of those without the author having to know anything specific about, say, `amd64`’s calling conventions or `arm64`’s alignment rules.

The CPU architectures differ along several axes that a driver author must take seriously. They differ in **endianness**: most of the common ones today are little-endian, but `powerpc` and `powerpc64` can be configured as big-endian and are historically big-endian. They differ in **word size**: 32-bit and 64-bit systems have different natural integer widths, which matters when you store an address in a variable. They differ in **alignment**: `amd64` will forgive an unaligned 32-bit access, but `arm` on some cores will fault. They differ in the physical layout of memory and the behaviour of memory barriers, but those are issues that the kernel’s synchronisation primitives already solve for you if you use them correctly.

A driver written carelessly will not compile at all on some of these architectures, or worse, will compile but misbehave. A driver written with a little care compiles cleanly everywhere and does the same thing everywhere. The difference between those two outcomes is almost always a matter of using the right types, the right endian helpers, and the right accessor functions, rather than a matter of deep kernel knowledge.

29.4.3 Portability Across Buses

The second axis of portability is the bus through which the driver talks to the hardware. The classical FreeBSD driver talks to a PCI device. But the same hardware function can appear behind other buses too: on USB, on a platform-specific memory-mapped region attached via Device Tree (FDT) on embedded systems, on a Low Pin Count or SMBus interface, or on nothing at all when the device is simulated in software. The `/usr/src/sys/dev/uart/` tree is a lovely example: `uart_bus_pci.c`, `uart_bus_fdt.c`, `uart_bus_acpi.c`, and several smaller files each teach the shared UART core how to be reached over a particular bus, and the core itself, in `uart_core.c`, does not care which one was used.

When a driver is portable across buses, adding a new bus is a matter of writing one small file that translates between the bus API and the driver's internal API, then listing that file in the kernel's build system. The core does not need to change. If you have a PCI-based variant of a device today and the same silicon turns up soldered to a board tomorrow, your driver picks up the new attachment in a few hundred lines rather than a few thousand.

29.4.4 Portability Across Systems

The third axis of portability is cross-system: can your driver be shared with other operating systems, especially the other BSDs? This is the form of portability that matters least for most FreeBSD-only projects and the one that consumes the most effort when it does matter, which is why we will treat it lightly in this chapter. The short answer is that NetBSD and OpenBSD share a great deal of history with FreeBSD, and a driver that is carefully written in the FreeBSD dialect can often be translated to those systems with a modest amount of work. A driver written carelessly, with FreeBSD assumptions scattered throughout, usually cannot be translated without a major rewrite.

For our purposes, cross-BSD compatibility means writing your driver so that FreeBSD-specific conventions are introduced in a small number of well-known places and can be wrapped or replaced when needed. You will see that pattern in Section 7.

29.4.5 Portability Across Time

A fourth form of portability is subtler but matters enormously over the life of a project: portability across time. FreeBSD is a living system. APIs evolve, macros are renamed, and older conventions give way to new ones. The `if_t` opaque handle for network interfaces, for example, replaced the raw `struct ifnet *` pointer across FreeBSD 13 and 14; drivers that used the opaque API continued to compile, while drivers that used the raw structure had to be updated. A driver that is careful about which APIs it uses, which headers it includes, and how it detects the kernel version it is compiled against, will survive those evolutions with little more than a version check at the top of a few files.

You will meet the `__FreeBSD_version` macro shortly. Think of it as a version stamp that the kernel itself carries, and that your driver can test against to adapt to the API surface of the specific release it is built on. Used sparingly, it is a quiet, effective tool. Used profligately, it turns the source into a patchwork.

29.4.6 Portability Across Deployments

A final, often forgotten, form of portability is portability between real hardware and a simulator. In Chapter 17 you learned how valuable it is to be able to test a driver without the real device present. A driver whose backend can be switched at build time between “the real device” and “a simulated stand-in” is much easier to develop, review, and test than one that can only run against the true hardware. This form of portability is the cheapest to achieve, because it costs only a small amount of up-front design, and it pays for itself every time a bug is caught on a CI runner that does not have the hardware.

29.4.7 Why Portability Matters

It is tempting to treat portability as a nice-to-have, something you will address “later” when the driver stabilises. In practice, portability is cheapest when it is designed in from the start, and most painful when it is bolted on after the driver has grown to ten thousand lines. There are three concrete reasons portability earns its keep.

First, hardware varies. The device you support today is rarely the only device you will ever support. Silicon vendors produce variants. A sensor that lived on I2C last year may appear over USB this year and over PCIe the year after, with the same programming model underneath. If your core logic is hardware-independent, those variants are cheap to support. If it is not, they are expensive.

Second, platforms evolve. A driver that ships on amd64 this year may need to run on arm64 next year as embedded deployments grow. A driver that assumes little-endian memory will misbehave on a big-endian target, and the bugs will be subtle because most of the time the code looks right. Arranging the driver so that endianness is handled in a small number of obvious places means the move to a new architecture is a one-afternoon exercise, not a one-month forensic investigation.

Third, time passes. A long-lived driver outlives its author. A driver that is structured cleanly can be maintained by a newcomer who has no context. A driver that is a tangle of conditional blocks and hardware assumptions cannot, and tends to be rewritten from scratch by the next maintainer. The accumulated cost of those rewrites, measured over the life of a project, is far higher than the cost of writing the original driver in a portable style.

29.4.8 Code Duplication vs Modular Design

Before we go further, it is worth naming a temptation that every driver author faces: duplication. When a second variant of a device appears, the fastest way to support it is to copy the existing driver and change the parts that are different. This works, in the short term. It is also one of the most expensive decisions a project can make, because every bug found in one copy must be found in the other, every improvement must be duplicated, and every security issue must be patched twice. After the third or fourth variant, the project has become unmaintainable.

The alternative is modular design. The parts that are the same live in one place. The parts that are different live in small, backend-specific files. The core is compiled once. Each backend is compiled separately and linked against the core. When a bug is fixed in the core, all variants benefit. When a new variant appears, only a small

file needs to be added. This is the approach FreeBSD drivers in the mainline tree overwhelmingly adopt, for the same reasons.

Chapter 29 teaches you to recognise where duplication is creeping in and how to pull it apart before it calcifies. That skill pays dividends for the rest of your career in systems programming.

29.4.9 A Tale of Two Drivers

Before closing this section, it helps to compare two hypothetical drivers that a team might write for the same device, one with portability in mind from the start and one without. The story is composite, but every scenario in it has occurred in real FreeBSD projects, usually more than once.

Driver A is written in a hurry to support a single PCI card on amd64. The author writes every register access as a direct call to `bus_read_4(sc->sc_res, offset)`. The PCI probe, the attach function, the ioctl switch, and the register dumping code all live in a single file of about twelve hundred lines. The author knows about endianness in theory but assumes the host is little-endian because the target is amd64. The driver ships, works, and passes its first round of customer testing. The team declares victory.

Six months later, a second customer requests support for a USB variant of the same device. The programming model is identical; the mechanism is different. The author opens Driver A and considers the options. Option one is to add a second attach path and scatter `if (sc->sc_is_usb)` branches through the file; the author quickly sees that this means touching more than a hundred functions, each of which would grow a new branch. Option two is to copy Driver A to Driver A'-USB and change the parts that differ. The author takes option two because it is faster. Now there are two drivers. Every bug fix must be applied twice, once to each copy, and after the third such fix the team notices they have already forgotten one of them. A maintenance tax quietly begins to accumulate.

A year after that, a third customer wants the same silicon supported on an arm64 board. The PCI driver is almost correct on arm64, but the endianness of a specific register was wrong, and the author did not notice because they had never built on a big-endian platform. The USB driver works, but only after the author realises that a structure they had cast a `volatile` pointer to was not aligned on arm64 and was producing bus faults. They add a third copy, Driver A''-USB-ARM, with fixes for the alignment and the endianness. The team now maintains three drivers. The ratio of bugs to fixes has climbed. The total lines of code have tripled, but the shared behaviour has not.

Driver B is written by a different team at the same time, for the same device. The author begins by drawing the line between hardware-dependent and hardware-independent code in the first hour, before writing any logic. They create a backend interface with four operations: `attach`, `detach`, `read_reg`, and `write_reg`. They implement a PCI backend first, and write the core against the interface. They also implement a simulation backend during development, because it is easier to test the core without needing a real card. The driver ships at about eighteen hundred lines, spread across six files.

Six months later, the USB request arrives. The author writes a `portdrv_usb.c` file of about three hundred lines, adds one line to the Makefile, and ships. The core is unchanged. The USB backend is reviewed in isolation, because it does not touch any other file. The bug fixes to the core benefit both backends automatically.

A year after that, the arm64 request arrives. The author builds on arm64, notices one warning from the compiler about a signed comparison with `uint32_t`, fixes the warning, and ships. The endianness is handled because every multi-byte value on a register boundary went through `le32toh` or `htole32` from day one. The alignment

is handled because the author used `bus_read_*` and `memcpy` rather than raw pointers. The arm64 port is a one-day job, not a month.

Two different teams, same device, same time, same customers. At the end of eighteen months, Driver A has three copies, two thousand lines of drift between them, and a steady trickle of bugs that affect one copy and not the others. Driver B has one copy, one backend interface, and a changelog that reads like a straight line. The teams that wrote them are by now worth different amounts to the project.

The moral is simple but easy to forget: **the cheap path at the start of a driver is often the expensive path over its lifetime**. Portability is an investment that pays dividends, but only if you make the investment early. Retrofitting portability into a driver that has already grown to several thousand lines is possible and sometimes necessary, but it takes more time than writing the driver from scratch in the portable style. This is not an argument for over-engineering; it is an argument for making the right small moves in the first week, before the driver has time to calcify.

29.4.10 The Cost of Deferring Portability

The “we’ll add portability later” argument sounds reasonable and is almost always wrong. There are three specific costs that a late refactor pays and an early design does not.

The detection cost. Portability bugs in a driver that was never tested portably are invisible. They lie in wait until the day someone tries to build on a new platform or a new release. By that time, other work depends on the driver, and a fix that touches a core assumption can cascade through the project. An early investment in portable patterns means the bugs are caught in the first week, not the third year.

The reviewer cost. A driver that mixes hardware-dependent code with core logic is harder to review, because every change is potentially a change to the architectural abstraction. Reviewers either slow down or, more commonly, start approving patches without fully understanding them. The project’s quality silently decays. A cleanly split driver gives reviewers a clear signal: a patch in the core deserves scrutiny because it affects every backend; a patch in a backend file is local and can be approved more quickly.

The contributor cost. New contributors are scared by tangled code. A driver that has obvious structure, where new features look like additions to an existing pattern, attracts contributions. A driver that is a wall of nested conditions repels them. Over the life of a project, this is the difference between a healthy community and a lone maintainer struggling to keep up.

None of these costs are visible in the first week. All three appear by the end of the first year. By the third year they dominate the project. That is why portability, discussed in the abstract, sounds optional and, discussed with real projects in mind, is nothing of the sort.

29.4.11 Why FreeBSD Makes This Easier Than Most

A final note, and a slightly optimistic one. Among mainstream kernels, FreeBSD is one of the easier to target for portability, because the project has taken the portable abstractions seriously from the beginning. `bus_space(9)` exists so that driver authors do not have to write architecture-specific register access. `bus_dma(9)` exists so that DMA behaves identically across platforms. Newbus exists so that bus-specific probe logic is separable from core driver logic. The endian helpers in `/usr/src/sys/sys/endian.h` exist so that a single expression works

on little-endian and big-endian hosts. The fixed-width types in `/usr/src/sys/sys/types.h` exist so that “how wide is an int on this platform” never matters.

Contrast this with projects that grew organically across architectures without a plan, and you will find drivers littered with `#ifdef CONFIG_ARM`, with raw pointer casts to device memory, and with ad hoc byte-swap code inlined at every register access. FreeBSD could have ended up that way and did not, because the core maintainers cared about portability from early on. Your drivers can inherit that discipline for free; the cost is learning which abstractions to use and using them consistently.

This is the spirit in which the rest of the chapter is written. The tools are in place. The patterns are clear. The investment is small if you make it early. Let us begin.

29.4.12 Wrapping Up Section 1

You now have a working definition of portability and a set of axes along which it varies: architecture, bus, operating system, time, and deployment. You also have a sense of why portability matters as a practical engineering concern, not a theoretical nicety, illustrated by the contrast between a careless driver and a careful one over a span of eighteen months. The next sections turn from definition to method. In Section 2, we begin the practical work of portability by isolating the parts of a driver that depend on hardware details from the parts that do not. That distinction is the first structural move every portable driver makes, and it is the move that pays the biggest dividends for the smallest cost.

29.5 Section 2: Isolating Hardware-Dependent Code

The first concrete move toward a portable driver is to separate code that depends on specific hardware details from code that does not. This sounds obvious when stated in the abstract, and it is surprisingly easy to get wrong in practice. Hardware-dependent code tends to leak into unexpected places, because the path from a piece of data to the hardware is long, and every step along the way has the potential to encode an assumption that should have stayed elsewhere.

This section introduces the idea in concrete terms, shows you the kinds of code that count as hardware-dependent, and then walks through the two most common mechanisms FreeBSD gives you for putting that code behind an abstraction: the `bus_space(9)` family of macros for register access, and the `bus_dma(9)` family for buffers that the device reads and writes directly. By the end of this section, you will recognise the pattern of hardware isolation when you see it in real drivers and know how to apply it to your own.

29.5.1 What Counts as Hardware-Dependent?

Let us be precise about what we mean. Hardware-dependent code is code whose correctness depends on facts that are specific to a particular piece of silicon or to the bus it sits on. Classic examples include:

- the numeric offset of a register within the device’s memory-mapped region
- the exact bit layout of a status word
- the width of a FIFO in bytes
- the required sequence of register writes to initialise the device
- the endianness in which the device expects to see multi-byte values

- the alignment constraints of DMA buffers for a particular DMA engine

Note that none of these facts are true across every device your driver might ever have to handle. If tomorrow the hardware vendor releases a revision with a different FIFO depth, every line that assumed the old depth must change. If the same silicon appears on USB instead of PCI, every `bus_space_read_*` call must become something else. Those are the seams where portability either holds or breaks.

By contrast, code is hardware-**independent** when it deals with the data once it is out of the device's grip. A function that walks a queue of pending requests, assigns sequence numbers, manages timeouts, or coordinates with the upper half of the kernel (like GEOM, `ifnet`, or `cdevsw`) does not need to know whether the data came in over PCI, USB, or a simulator. As long as the hardware-dependent layer has lifted the data up into a clean, uniform representation, the rest of the driver does not care.

The goal, then, is to draw a line down through the driver. Above the line lives hardware-independent logic. Below the line lives everything that knows about registers, DMA, or bus-specific APIs. The higher the line can be drawn without distorting the code, the more portable the driver becomes.

29.5.2 A Small Thought Experiment

Before we introduce tools, try a thought experiment. Imagine a driver for a made-up “widget” device that performs some simple I/O. The device has two registers: a control register at offset 0x00 and a data register at offset 0x04. Your driver writes a byte to the data register, sets a bit in the control register to start a transfer, polls another bit in the control register until it clears, and then reads a status from the data register.

Now imagine the same widget is produced in two physical forms. One form is a PCI card where both registers are reached through a memory-mapped BAR. Another form is a USB dongle where both registers are reached by USB control transfers. The programming model is identical, but the mechanism for actually reading and writing the registers is completely different.

How do you write the core of the driver, the part that performs transfers and polls for completion, so that it works with both forms?

The answer, which drives the rest of this section, is to stop writing register accesses directly. Instead, write accessors. The core logic calls `widget_read_reg(sc, offset)` and `widget_write_reg(sc, offset, value)`. Those accessors are backend-specific: the PCI backend maps them to `bus_space_read_4` and `bus_space_write_4`, and the USB backend maps them to USB control transfers. The core logic does not know, and does not need to know, which backend is in use.

This is the fundamental move of hardware isolation, and it scales. Once your driver reads and writes registers only through accessors, swapping the backend becomes a local change. Add a third backend for a simulated device? Write a third set of accessors that read and write an in-memory buffer. Nothing else in the driver changes.

29.5.3 A Closer Look at the `uart` Driver

Before we get to the FreeBSD primitives, it helps to trace the shape of a real, mature driver so you can see the ideas in situ. Open `/usr/src/sys/dev/uart/uart_core.c` and scroll slowly. You will see that the file contains the state machine for a UART: the TTY integration, the transmit and receive paths, the interrupt routing, and the lifecycle hooks. You will not find `bus_space_read_4` anywhere in this file (or rather, you will find it

only in helpers that are themselves abstracted behind class-specific methods), and you will not find any mention of PCI, FDT, or ACPI. The core does not know which bus the UART is sitting on. It knows only the programming model of a UART.

Now open `/usr/src/sys/dev/uart/uart_bus_pci.c`. This file is short. It declares a table of vendor and device IDs, implements a PCI probe that matches against the table, implements a PCI attach that allocates a BAR and wires it into the class, and registers the driver with Newbus via `DRIVER_MODULE`. It is the PCI backend for the UART. Compare with `/usr/src/sys/dev/uart/uart_bus_fdt.c`, which does the same job for Device Tree platforms, and `/usr/src/sys/dev/uart/uart_bus_acpi.c` for ACPI platforms.

Finally, open `/usr/src/sys/dev/uart/uart_bus.h`. This is the header that ties the pieces together. It defines `struct uart_class`, which is essentially the UART's backend interface: a set of function pointers plus some configuration fields. Each hardware variant of the UART provides a `struct uart_class` instance. The core calls through the class rather than calling any variant directly.

This is the same pattern we will build in Section 3, only in the specific dialect that FreeBSD uses for UARTs. Reading the UART driver alongside this chapter will cement the ideas much better than reading the chapter alone. Keep `/usr/src/sys/dev/uart/uart_bus.h` open in a second window as you read the rest of Section 2 and the whole of Section 3.

29.5.4 The `bus_space(9)` Abstraction

FreeBSD already gives you a tool that solves this problem for memory-mapped and I/O-port devices: the `bus_space(9)` API. You have encountered it before in Chapter 16, so this is a recap rather than a fresh introduction. The purpose of `bus_space` is to let you write a single register access that works correctly regardless of the CPU architecture, regardless of whether the device is reached through memory-mapped I/O or I/O ports, and regardless of the specific machine's bus topology.

At the programming model level, you hold two opaque values for each device: a `bus_space_tag_t` and a `bus_space_handle_t`. Once you have them, you access the device through calls such as:

```
uint32_t value = bus_space_read_4(sc->sc_tag, sc->sc_handle, REG_CONTROL);
bus_space_write_4(sc->sc_tag, sc->sc_handle, REG_CONTROL, value | CTRL_START);
```

On amd64, the tag and the handle are simple integers. You can confirm this by opening `/usr/src/sys/amd64/include/_bus.h`, where both are defined as `uint64_t`. On arm64, the tag is a pointer to a structure of function pointers and the handle is a `u_long`. You can see this in `/usr/src/sys/arm64/include/_bus.h`. The ARM family needs the indirection because different platforms within ARM can map memory differently, and a function-pointer dispatch handles all of them uniformly.

This architectural difference is essential to portability. If you bypass `bus_space` and use a raw volatile `uint32_t *` pointer to the device's memory, your code will work on amd64 and will fail silently or loudly on some ARM platforms. If you use `bus_space`, the same code works on both. That is the single most important portability gain you can get from this chapter with the smallest effort.

The `bus_space` functions come in a family that varies along three axes. First, the width: `bus_space_read_1`, `bus_space_read_2`, `bus_space_read_4`, and on 64-bit architectures `bus_space_read_8`. Second, the direction: read or write. Third, the multiplicity: the plain variant accesses a single value, the `_multi_` variants

access a buffer of values, and the `_region_` variants access a contiguous region with an implied increment. For most drivers the single-value variants are the workhorses, and you will reach for the others when you have FIFOs or packet buffers to move in bulk.

You can confirm the presence of these functions by opening `/usr/src/sys/sys/bus.h`. The header has a long series of macro definitions for `bus_space_read_1`, `bus_space_write_1`, `bus_space_read_2`, and so on, each defined in terms of the architecture-specific primitives.

29.5.5 Wrapping `bus_space` for Clarity

Even when you are committed to using `bus_space`, you should go one step further and wrap the calls in driver-specific accessors. Why? Because raw `bus_space_read_4(sc->sc_tag, sc->sc_handle, REG_CONTROL)` appears dozens of times in a growing driver, and each appearance is three arguments of visual noise when the only thing that matters is the offset and the value. More importantly, the raw call hardcodes the idea that the device is reached through `bus_space`. If you ever want to add a different backend, every one of those calls must change.

A driver-local wrapper gives you both a cleaner call site and a single place to change. The wrapper looks like this:

```
static inline uint32_t
widget_read_reg(struct widget_softc *sc, bus_size_t off)
{
    return (bus_space_read_4(sc->sc_btag, sc->sc_bhandle, off));
}

static inline void
widget_write_reg(struct widget_softc *sc, bus_size_t off, uint32_t val)
{
    bus_space_write_4(sc->sc_btag, sc->sc_bhandle, off, val);
}
```

Now the rest of the driver calls `widget_read_reg(sc, REG_CONTROL)` and `widget_write_reg(sc, REG_CONTROL, val)`. The call sites read like specifications of intent rather than like plumbing. And when you add a USB backend in the next section, you can change the body of these two functions to dispatch on the backend type without touching any of the hundreds of callers.

This pattern, *wrap the primitive, then call the wrapper*, is the first and most common tool of driver portability. Make it a habit. Whenever you find yourself typing `bus_space_read_*` or `bus_space_write_*` in a file other than the backend-specific one, stop and ask whether a wrapper would serve you better.

29.5.6 Conditional Register Access by Bus

Once wrappers are in place, the next question is what to do when the same driver must support multiple buses. There are two common approaches.

The first approach is **compile-time selection**. The driver is built once per bus, and the wrappers are implemented differently in each build. This is the approach `uart(4)` takes in some contexts. Each bus backend file builds its own set of attach helpers and calls the core's API. The wrappers do not need runtime dispatch because each backend compiles against its own specialisation. This approach produces the smallest, fastest binary, but it requires you to build the driver once per bus backend you want to support.

The second approach is **runtime dispatch**. The driver is built once and includes support for all enabled backends. At attach time, the driver detects which backend is actually in use and stores a function pointer in the `softc`. Each call through the wrapper costs one indirection. This approach is slightly more flexible at the cost of a small runtime overhead.

For most drivers, especially beginner-level ones, the clearest approach is a hybrid: build the core once, build each backend as a separately compiled source file, and use a small backend table per-instance that the core uses to dispatch. The wrappers are inline functions that read the backend table and call through it. You will see exactly this pattern in Section 3 when we introduce the backend interface formally.

29.5.7 The `bus_dma(9)` Abstraction

For devices that perform direct memory access (DMA) to transfer data between system memory and the device, the analogous abstraction is `bus_dma(9)`. DMA is hardware-dependent in ways that register access is not: the physical address that the device sees is not always the same as the kernel virtual address your code holds, alignment requirements vary by device and by bus, and some architectures need explicit cache-flush or cache-invalidate operations between the CPU and the DMA engine to maintain consistency.

Open `/usr/src/sys/sys/bus_dma.h` and look at the core API. Without going into the full depth of `bus_dma`, the shape of the interface is:

```
int bus_dma_tag_create(bus_dma_tag_t parent, bus_size_t alignment,
    bus_addr_t boundary, bus_addr_t lowaddr, bus_addr_t highaddr,
    bus_dma_filter_t *filtfunc, void *filtfuncarg,
    bus_size_t maxsize, int nsegments, bus_size_t maxsegsz,
    int flags, bus_dma_lock_t *lockfunc, void *lockfuncarg,
    bus_dma_tag_t *dmat);

int bus_dmamem_alloc(bus_dma_tag_t dmat, void **vaddr, int flags,
    bus_dmamap_t *mapp);

int bus_dmamap_load(bus_dma_tag_t dmat, bus_dmamap_t map, void *buf,
    bus_size_t buflen, bus_dmamap_callback_t *callback,
    void *callback_arg, int flags);

void bus_dmamap_sync(bus_dma_tag_t dmat, bus_dmamap_t dmamap,
    bus_dmasync_op_t op);

void bus_dmamap_unload(bus_dma_tag_t dmat, bus_dmamap_t dmamap);
void bus_dmamem_free(bus_dma_tag_t dmat, void *vaddr, bus_dmamap_t map);
```

The full mechanics of `bus_dma` are covered in depth in later chapters. For the purpose of this chapter you need only the portability consequence: **use `bus_dma` for all device-visible buffers**, and hide the use behind driver-local helpers just as you did for register access. The reason is identical. A direct `bus_dmamap_load` call encodes the assumption that you are using `bus_dma` and makes it hard to substitute an in-memory buffer for simulated testing, or a different DMA framework for an unusual platform. A wrapper function such as `widget_dma_load(sc, buf, len)` hides the assumption.

The callback-based nature of `bus_dmamap_load` is itself a portability feature worth pausing on. The callback is invoked with the list of physical segments the buffer was mapped into. On a system with a simple 1:1 virtual-to-physical mapping, that list will have one entry. On a system that needs bounce buffers to reach limited address ranges, the list might have several entries. Your driver writes one callback, and that callback handles both cases uniformly. That is `bus_dma`'s portability contribution: the callback sees a simple model, and the complexity lives under `bus_dma` itself.

29.5.8 A Practical Header-Per-Backend Pattern

Putting these ideas together, a portable driver often uses a small family of header files and source files to make the separation visible in the filesystem layout. The core includes `widget_common.h` for types and prototypes that are not tied to any bus. The core source, typically `widget_core.c`, contains the hardware-independent logic. Each bus backend lives in its own `.c` file, such as `widget_pci.c` or `widget_usb.c`, and includes `widget_backend.h` for the definitions of the backend interface. The Makefile lists the core and whichever backends are enabled in `SRCS`, and the kernel build does the rest.

You will see this layout applied step by step in Section 4 with the `portdrv` reference driver. Before we get there, we need to formalise what a “backend interface” looks like. That is the subject of the next section.

29.5.9 Common Mistakes When Isolating Hardware Code

A handful of mistakes recur often enough to be worth calling out. Each one is small in isolation but damaging in aggregate.

The first mistake is **hidden access**. A helper function somewhere deep in the driver reaches into the softc and performs a register access directly instead of going through the accessor. The result is a single leak, but the leak can be enough to defeat the abstraction. When you read a driver, look for calls to `bus_space_*` outside the designated backend files; they are warning signs.

The second mistake is **offset leakage**. The driver writes `bus_space_read_4(tag, handle, 0x20)` with a literal offset. Literal offsets are fine as constants named `REG_CONTROL` in a header, but they are dangerous as bare integers. A bare offset tells you nothing about what the register does, and renumbering the register on a future hardware revision becomes a search-and-replace exercise. Always define register names as macros or enumerations in the header, and use the names in the code.

The third mistake is **type confusion**. A register returns a 32-bit value, but the code stores the result in an `int` or an `unsigned long`. On a 64-bit system this often works by accident, because the value fits. On a 32-bit system with a signed `int`, a register whose top bit is set becomes negative, and comparisons break. Always match the type to the register width, using `uint8_t`, `uint16_t`, `uint32_t`, or `uint64_t` as appropriate.

The fourth mistake is **endianness assumptions at the accessor layer**. The `bus_space` functions return the value as the host sees it, in host byte order. If the device uses a specific byte order in its registers, the conversion

happens elsewhere in the code, using `htole32` or `be32toh` as needed. Do not bake the conversion into the accessor, because other parts of the driver may be accessing the same register for administrative purposes and do not want the conversion applied.

The fifth mistake is **no accessor at all**. The driver uses raw volatile `uint32_t *` pointers to map registers. This works on some architectures and fails on others. It defeats the entire point of `bus_space`. If you are reading a driver that does this, assume it is broken on at least one architecture even if you cannot immediately say which one.

29.5.10 Subregions, Barriers, and Burst Operations

Three `bus_space` features are worth a short tour, because they come up often enough in real drivers that you will meet them even in small code bases.

The **subregion** feature lets you split a mapped region into logical parts and hand each part to a different piece of code. If your device's BAR is 64 KB and the driver conceptually has three separate register banks at offsets 0, 0x4000, and 0x8000, you can create three subregions with `bus_space_subregion` and give each to the function that cares about it. Each subregion has its own tag and handle, and accesses to it naturally carry offsets from the start of the subregion, not from the start of the whole BAR. The result is code that reads `bus_space_read_4(bank_tag, bank_handle, REG_CONTROL)` rather than `bus_space_read_4(whole_tag, whole_handle, BANK_B_BASE + REG_CONTROL)`. The offsets become local to the bank and do not carry the whole BAR's address arithmetic in every call site.

The **barrier** feature is about ordering. When you write to a device register, the kernel has two separate guarantees to provide. First, that the write actually reaches the device, rather than sitting in a CPU write buffer or a bus bridge. Second, that writes happen in the order the programmer expected. On most common amd64 configurations, these guarantees come for free; on arm64 and some other platforms, they do not. The primitive is `bus_space_barrier(tag, handle, offset, length, flags)`, where `flags` is a combination of `BUS_SPACE_BARRIER_READ` and `BUS_SPACE_BARRIER_WRITE`. The call says: "all reads or writes in the specified range before this point must complete before any after it." You use it when your driver depends on the order in which the device sees register accesses, for example when arming an interrupt after setting up a DMA descriptor.

The **burst** feature handles fast moves to or from device memory. When you need to copy a packet into a FIFO or out of a DMA buffer, calling `bus_space_write_4` in a loop is correct but slow. The `bus_space_write_multi_4` and `bus_space_read_multi_4` functions do the whole burst in one call, and on architectures that have specialised move instructions they use them. For a network driver transferring frames at high rates, burst helpers can be the difference between wire speed and half wire speed, and the cost of using them is only a slightly different call signature.

None of these features is mandatory for a small driver. All of them are worth knowing about, because their absence in a growing driver leads to workarounds that are harder to read and slower to run than the primitive they were avoiding.

29.5.11 Layered Wrapping: From Primitive to Domain

The accessors shown earlier in this section are the first layer of wrapping around `bus_space`. Real drivers sometimes benefit from a second layer that lifts the accessors into a domain-specific vocabulary. The idea is

simple: once you have `portdrv_read_reg(sc, off)`, you can build on top of it with functions like `portdrv_read_status(sc)`, `portdrv_arm_interrupt(sc)`, or `portdrv_load_descriptor(sc, idx, addr, len)`. The core of the driver then speaks in the device’s vocabulary rather than in raw register accesses.

```
static inline uint32_t
portdrv_read_status(struct portdrv_softc *sc)
{
    return (portdrv_read_reg(sc, REG_STATUS));
}

static inline bool
portdrv_is_ready(struct portdrv_softc *sc)
{
    return ((portdrv_read_status(sc) & STATUS_READY) != 0);
}

static inline void
portdrv_arm_interrupt(struct portdrv_softc *sc, uint32_t mask)
{
    portdrv_write_reg(sc, REG_INTR_MASK, mask);
}
```

Code that calls `portdrv_is_ready(sc)` instead of `(portdrv_read_reg(sc, REG_STATUS) & STATUS_READY) != 0` reads like intent rather than plumbing. When the definition of “ready” changes on a newer revision of the device, the change happens in one inline function rather than at every call site. This is the same pattern as the primitive accessors, only applied to the next layer up.

Do not overdo it. A wrapper per trivial register read is wrappering for its own sake. Add a domain-level wrapper when its name is meaningfully more informative than the register access, or when the operation is non-trivial enough that the reader benefits from a name. For a single register read, the primitive wrapper is usually enough.

29.5.12 Accessor Layering Inside the FreeBSD Tree

You can see this layered approach used in mature FreeBSD drivers. Look at `/usr/src/sys/dev/e1000/` for the Intel Gigabit Ethernet driver. The file `e1000_api.c` exposes dispatch functions like `e1000_reset_hw`, `e1000_init_hw`, and `e1000_check_for_link`, each of which forwards to a chip-family helper (for example, the 82571 helpers live in `e1000_82571.c`, the 82575 helpers live in `e1000_82575.c`, and so on). Those chip-specific helpers do their register work through `E1000_READ_REG` and `E1000_WRITE_REG`, which are defined in `/usr/src/sys/dev/e1000/e1000_osdep.h` as macros that expand to `bus_space_read_4` and `bus_space_write_4` on the tag and handle carried inside the driver’s struct `e1000_osdep`. Four layers: bus primitive, register accessor macro, chip-family helper, and driver-facing API in `if_em.c`. Each layer adds a little more meaning, and each layer can be changed without disturbing the others.

For your own drivers, two layers is usually the right target: a primitive accessor that wraps `bus_read_*` or dispatches through the backend, and a set of domain helpers that lift the primitive into the vocabulary of the

device. Three layers, if the driver is large and spans multiple subsystems. More than three, and the layering is probably obscuring rather than clarifying.

29.5.13 When the Line Between HW-Dependent and HW-Independent Blurs

The line between hardware-dependent and hardware-independent code is usually clear, but there are edge cases worth naming.

One edge case is **timing logic**. Suppose the driver must wait up to 50 microseconds for a status bit to appear after writing a register. The wait itself is hardware-dependent, because the 50-microsecond figure comes from the device's datasheet. But the scaffolding, the loop that polls, the timeout that gives up, and the error return, is hardware-independent. The right design is to have a small helper in the core that takes the poll predicate as a function pointer or inline expression, and a hardware-specific constant for the timeout. The core owns the polling loop; the backend owns the constant.

```
/* Core. */
static int
portdrv_wait_for(struct portdrv_softc *sc,
    bool (*predicate)(struct portdrv_softc *sc),
    int timeout_us)
{
    int i;

    for (i = 0; i < timeout_us; i += 10) {
        if (predicate(sc))
            return (0);
        DELAY(10);
    }
    return (ETIMEDOUT);
}
```

The backend provides the predicate (which may be trivial, such as reading a status register) and the timeout. This keeps the loop structure in one place and lets the individual device details live where they belong.

Another edge case is **state machines**. The state machine of a typical request processing loop is hardware-independent: idle, pending, in-flight, complete. The transitions, however, may depend on hardware events. A common pattern is to keep the state machine in the core and invoke backend methods for the operations that might differ between backends. The core says “start the transfer”; the backend does whatever that means. The state transitions happen in the core based on the backend's return values.

A third edge case is **logging and telemetry**. You want one `device_printf` in the core when a transfer fails, not one per backend. But the `printf` may want to include backend-specific information such as the register that caused the error. The solution is to expose a backend method like `describe_error(sc, buf, buflen)` that fills a buffer with a human-readable summary, and have the core call it. The core owns the log line; the backend owns the vocabulary.

The common theme in all these edge cases is the same: **put structure in the core, and put details in the backend**. When you are unsure where a piece of code belongs, ask which category it is, structure or detail. Structure is about what happens. Details are about exactly how the hardware is touched to make it happen.

29.5.14 Wrapping Up Section 2

You have now seen the first structural move every portable driver makes: draw a clean line between hardware-dependent and hardware-independent code, and express the line in actual files and accessor functions rather than in comments. The `bus_space(9)` and `bus_dma(9)` abstractions are FreeBSD's gifts to that effort, because they already hide most architecture-specific complexity. Wrapping them in driver-local accessors gives you the final piece: a single place to change when the backend changes. Layered wrapping, disciplined use of subregions and barriers, and careful placement of edge-case code between core and backend make the line between them stable rather than fragile.

The next section takes the step beyond accessors. If the driver has to support multiple backends in the same binary, what does the interface between the core and the backends look like? That is where the idea of a **backend interface**, expressed as a struct of function pointers, becomes essential.

29.6 Section 3: Abstracting Device Behavior

In Section 2 we drew a line between hardware-dependent and hardware-independent code, and wrapped the primitive bus operations behind driver-local accessors. That move handles the simple case, where the driver talks to one kind of device over one kind of bus. In this section we handle the harder and more interesting case: a driver that supports multiple backends in the same build.

The technique that makes this manageable is straightforward to describe and profoundly useful in practice. The driver's core defines an **interface**: a struct of function pointers that describes what operations a backend must provide. Each backend provides an instance of that struct, filled in with its own implementations. The core calls through the struct rather than calling any backend directly. Adding a new backend is a matter of writing one more struct instance and one more set of functions; the core is untouched.

This pattern is everywhere in FreeBSD. The Newbus framework itself is built on a more elaborate version of it, called `kobj`, and you have been using `kobj` indirectly every time you filled in a `device_method_t` array in earlier chapters. In this chapter we introduce a lighter version that you can apply to your own drivers without the full ceremony of `kobj`.

29.6.1 Why a Formal Interface Beats If-Else Chains

The first instinct of a programmer who needs to support two backends is often to branch on a backend type at each call site. Something like this:

```
static void
widget_start_transfer(struct widget_softc *sc)
{
    if (sc->sc_backend == BACKEND_PCI) {
```

```

        bus_space_write_4(sc->sc_btag, sc->sc_bhandle,
            REG_CONTROL, CTRL_START);
    } else if (sc->sc_backend == BACKEND_USB) {
        widget_usb_ctrl_write(sc, REG_CONTROL, CTRL_START);
    } else if (sc->sc_backend == BACKEND_SIM) {
        sc->sc_sim_state.control |= CTRL_START;
    }
}

```

This works for one function. It becomes unbearable by the tenth. Every function acquires a switch statement. Every new backend adds another branch to every switch. The number of edits grows as the product of functions and backends. After the third backend the driver is barely readable, and any change to the shape of a backend operation requires visits to a dozen files.

The structural fix is to replace the chain with a single indirection. Define a struct of function pointers:

```

struct widget_backend {
    const char *name;
    int (*attach)(struct widget_softc *sc);
    void (*detach)(struct widget_softc *sc);
    uint32_t (*read_reg)(struct widget_softc *sc, bus_size_t off);
    void (*write_reg)(struct widget_softc *sc, bus_size_t off, uint32_t val);
    int (*start_transfer)(struct widget_softc *sc,
        const void *buf, size_t len);
    int (*poll_done)(struct widget_softc *sc);
};

```

Each backend provides an instance of this struct. The PCI backend defines `widget_pci_read_reg`, `widget_pci_write_reg`, and the rest, and then fills in a struct `widget_backend` with those function pointers:

```

const struct widget_backend widget_pci_backend = {
    .name          = "pci",
    .attach        = widget_pci_attach,
    .detach        = widget_pci_detach,
    .read_reg      = widget_pci_read_reg,
    .write_reg     = widget_pci_write_reg,
    .start_transfer = widget_pci_start_transfer,
    .poll_done     = widget_pci_poll_done,
};

```

The USB backend does the same, the simulation backend does the same, and so on. The `softc` carries a pointer to whichever backend this instance is using:

```

struct widget_softc {
    device_t    sc_dev;
    const struct widget_backend *sc_be;
    /* ... bus-specific fields, kept opaque to the core ... */
    void        *sc_backend_priv;
};

```

And the core calls through the pointer:

```

static void
widget_start_transfer(struct widget_softc *sc, const void *buf, size_t len)
{
    (void)sc->sc_be->start_transfer(sc, buf, len);
}

```

The core does not branch. The core does not mention pci, usb, or sim. The core calls an operation, and the operation is looked up through a pointer. This is the fundamental move. Read it a few times and let it settle; the rest of the section is elaboration and concrete detail.

29.6.2 Setting Up the Backend at Attach Time

Each attach path installs the right backend into the_softc. The PCI attach code looks roughly like this:

```

static int
widget_pci_attach(device_t dev)
{
    struct widget_softc *sc = device_get_softc(dev);
    int err;

    sc->sc_dev = dev;
    sc->sc_be = &widget_pci_backend;

    /* Allocate bus resources, store them in sc->sc_backend_priv. */
    err = widget_pci_alloc_resources(sc);
    if (err != 0)
        return (err);

    /* Hand off to the core, which will use the backend through sc->sc_be. */
    return (widget_core_attach(sc));
}

```

The USB attach is structurally identical: allocate its bus resources, install widget_usb_backend, and call widget_core_attach. The simulation attach is identical too: install widget_sim_backend, which does not allocate anything real, and call widget_core_attach.

The lesson is that **the per-backend attach path is small, and its job is to prepare the softc for the core**. Once the softc has a valid backend and valid backend-private state, the core takes over and does the universal work of wiring the driver into the rest of the kernel.

29.6.3 How the Real UART Driver Does This

Open `/usr/src/sys/dev/uart/uart_bus.h` and jump to the struct `uart_class` definition. You will see the pattern we have just described, with a FreeBSD-specific twist. The twist is that the UART driver uses the `kobj(9)` framework, which adds a layer of compile-time dispatch on top of function pointers, but the essence is the same. A struct `uart_class` carries a `uc_ops` pointer to an operations struct, along with some configuration fields, and each hardware-specific variant of the driver provides its own `uart_class` instance.

Now open `/usr/src/sys/dev/uart/uart_bus_pci.c`. The `device_method_t uart_pci_methods[]` array at the top of the file declares which functions implement the PCI-specific device methods. Compare with `/usr/src/sys/dev/uart/uart_bus_fdt.c`, which does the same job for Device Tree-based platforms. Both files are short because they do very little: they install the right class, allocate resources, and hand off to the core in `uart_core.c`.

This is exactly the pattern you are learning in this section, only dressed in the formal `kobj` idiom that FreeBSD uses for bus drivers. When you write your own lighter-weight abstraction for a smaller driver, you can use a plain struct of function pointers, and the result is easier to read and reason about than a full `kobj`. For a driver with a handful of backends and a simple method set, the plain struct is the right choice.

29.6.4 Choosing the Set of Operations

Designing a backend interface is a matter of taste and judgement. There is no mechanical formula, but a few principles help.

First, **every operation that varies by backend belongs in the interface; every operation that does not varies does not**. If the same code is correct regardless of whether the backend is PCI or USB, put that code in the core. Do not put it in the interface, because each backend would then have to implement it, and the implementations would be identical, which defeats the point.

Second, **keep the interface narrow**. The fewer operations, the less work to implement a new backend and the less surface area to maintain. If two operations always appear together at call sites, consider merging them into one operation that does both.

Third, **prefer coarse operations over fine ones**. An operation like `start_transfer(sc, buf, len)` is coarse: it captures the intent of a transfer in one call. An equivalent pair like `write_reg(sc, DATA, ...)` followed by `write_reg(sc, CONTROL, CTRL_START)` is fine-grained, and it forces the core to know the register layout. Coarse operations let the backends implement the sequence however they need to, including in ways the core cannot know. This is also the key to backend simulation: a simulator can “perform” a transfer without actually writing any registers.

Fourth, **leave room for private state**. Each backend may need its own per-instance data that the core does not care about. A `void *sc_backend_priv` in the softc, owned by the backend, is the usual pattern. The PCI backend stores its bus resources there. The USB backend stores its USB device handles. The simulation

backend stores its in-memory register map. The core never reads or writes this field; it simply passes the softc through.

Fifth, **define a versioned struct if you want to evolve the interface over time**. When the interface is small and the number of backends is also small, you can add a field and update all backends in one commit. When the backends are external to your tree or maintained separately, you may want a version field in the struct, so a backend can refuse to attach if the core expects a newer shape. This is the same trick FreeBSD itself uses in some of its KPIs.

29.6.5 Avoiding If-Else Creep

Once the interface is in place, keep watch for a subtle form of regression: the reintroduction of backend-specific branches. This can happen innocently. A new feature is added that only makes sense on one backend, and rather than extend the interface, the developer writes `if (sc->sc_be == &widget_pci_backend) { ... }` somewhere in the core. This works, once. Repeat it three or four times and the driver is no longer cleanly abstracted; the backend is leaking back into the core through tag comparisons.

The right fix for a feature that only one backend supports is to add an operation to the interface that the other backends implement trivially. If only the PCI backend supports, say, interrupt coalescing, add `set_coalesce(sc, usec)` to the interface and implement it as a no-op in the USB and simulation backends. The core calls it unconditionally. This keeps the core ignorant of backend identity while still allowing per-backend specialisation.

29.6.6 Backend Discovery and Registration

A last practical question: how does a backend get attached in the first place? The answer varies by bus. On PCI, the kernel's Newbus machinery walks the PCI tree and calls each candidate driver's probe function. You write `widget_pci_probe` and register it with `DRIVER_MODULE(widget_pci, ...)`; the kernel does the rest. On USB, the `usbd` framework performs an analogous walk over USB devices and calls each candidate driver's probe. On a purely virtual or simulated backend, there is no bus to walk, and the backend is usually attached through a module init hook that creates a single instance.

The important portability point is this: **each bus's registration machinery is part of the backend, not the core**. The core never calls `DRIVER_MODULE` or `USB_PNP_HOST_INFO` or any bus-specific registration macro. The core exports a clean entry point (`widget_core_attach`) that each backend calls when its own attach logic is ready. Everything bus-specific happens in the backend file. This is what keeps the core clean enough to move unchanged to a new bus.

29.6.7 Common Mistakes When Designing an Interface

A few pitfalls recur often enough to call out.

Over-abstracting. Not every operation belongs in the interface. An operation that only one backend ever needs should live in that backend's private code, not be forced into the struct. An interface full of stub `return 0;` implementations has grown too wide.

Under-abstracting. Conversely, an interface that forces each backend to replicate the same five lines of setup logic has grown too narrow. If three backends all begin their `start_transfer` with the same two lines, move those two lines into the core and have the interface operation receive already-validated input.

Exposing backend types. If the core has a switch on backend identity, the abstraction is broken. The core should know which operations to call, not which backend is providing them.

Mixing synchronisation policies. Each backend must honour the core's locking policy. If the core calls `start_transfer` with the softc lock held, every backend must respect that and not sleep. If the core calls `attach` without a lock, the backend must not assume one is held. An interface is not just a set of signatures; it is also a contract about the surrounding context.

Ignoring lifetime. If the core calls `backend->attach` and gets back a non-zero error code, it must not call `backend->detach` afterwards, because nothing was attached. Document the contract, and write the backends to honour it. Confusion here leads to double frees and use-after-free bugs.

29.6.8 A Closer Look at `kobj(9)`

Because Newbus itself uses `kobj(9)`, it is worth taking a moment to look at what the kernel does for you when you write `device_method_t`. The mechanism may look intimidating, but the underlying idea is the same function-pointer dispatch we have been discussing, only formalised.

Open `/usr/src/sys/kern/subr_kobj.c` and look at how classes are registered. A `kobj_class` holds a name, a list of method pointers, an ops table that the kernel builds at module load time, and a size for instance data. When you write:

```
static device_method_t portdrv_pci_methods[] = {
    DEVMETHOD(device_probe,   portdrv_pci_probe),
    DEVMETHOD(device_attach,  portdrv_pci_attach),
    DEVMETHOD(device_detach,  portdrv_pci_detach),
    DEVMETHOD_END
};
```

you are building one of these method lists. The kernel compiles the list into a dispatch table at runtime, and every call such as `DEVICE_PROBE(dev)` becomes a lookup into the table followed by a call through a function pointer. From the programmer's perspective, the call site is a single macro; from the machine's perspective, it is one indirection.

For a small backend abstraction, the plain struct-of-function-pointers pattern we showed earlier is enough and is easier to read. When the interface grows beyond about eight methods and begins to accumulate default implementations, `kobj` starts to earn its keep. The ability to have a subclass inherit methods from a base class, to default a method to a no-op when a class does not provide it, and to be statically checked via the `method-ID` header is worth the additional machinery. For most drivers in this book, you will not need `kobj` directly, but recognising that it is doing the same thing your plain struct does makes the Newbus machinery feel less magical.

29.6.9 Stacked Backends and Delegation

A related, more advanced pattern is the **stacked** or **delegating** backend. Sometimes a backend is naturally a thin wrapper over another backend, with some small transformation applied. Imagine a backend that forces all

register writes to be logged for debugging, or one that adds a delay before every read to simulate a slow bus, or one that records register traffic into a ring buffer for later replay. Each of these is functionally a wrapper around a real backend.

The pattern is straightforward. The wrapping backend's struct includes a pointer to an inner backend:

```
struct portdrv_debug_priv {
    const struct portdrv_backend *inner;
    void *inner_priv;
    /* logging state, statistics, etc. */
};

static uint32_t
portdrv_debug_read_reg(struct portdrv_softc *sc, bus_size_t off)
{
    struct portdrv_debug_priv *dp = sc->sc_backend_priv;
    void *saved_priv = sc->sc_backend_priv;
    uint32_t val;

    sc->sc_backend_priv = dp->inner_priv;
    val = dp->inner->read_reg(sc, off);
    sc->sc_backend_priv = saved_priv;

    device_printf(sc->sc_dev, "read 0x%04jx -> 0x%08x\n",
        (uintmax_t)off, val);
    return (val);
}
```

The debug backend borrows the real backend's implementation for the actual register read, and adds logging around it. The core does not know it is talking to a wrapper. At attach time, the driver can install either the plain PCI backend or the debug backend that wraps it. This is a useful pattern for development, for recording a trace of a boot failure, or for building a simulator that replays a recorded trace.

Not every driver needs stacked backends. When you do need them, the pattern above is the template. The interface stays the same; a backend instance simply has the flexibility to delegate to another instance rather than doing the work itself.

29.6.10 Making the Interface Contract Explicit

A good backend interface is not just a struct; it is a contract about what each method does, what preconditions it has, and what side effects it produces. The contract lives in a comment at the top of the header:

```
/*
 * portdrv_backend.h - backend interface contract.
 */
```

```

* The core acquires sc->sc_mtx (a MTX_DEF) before calling any method
* except attach and detach. Methods must not sleep while holding this
* lock. The core releases the lock before calling attach and detach,
* because those methods may allocate memory with M_WAITOK or perform
* other sleepable operations.
*
* read_reg and write_reg must be side-effect-free from the core's
* point of view, other than the corresponding register access.
* start_transfer may record pending work but must not block; it
* returns 0 on success or an errno on failure. poll_done returns
* non-zero when the transfer has completed and zero otherwise.
*
* Backends may return EOPNOTSUPP for an operation they do not
* support. The core treats EOPNOTSUPP as "feature not present",
* and its callers degrade gracefully.
*/

```

This kind of comment is not decoration. It is the body of knowledge that a new contributor needs to implement a new backend correctly. Without it, the author of the third backend must reverse-engineer the contract from the existing two, and usually gets it slightly wrong. With it, the third backend is a straightforward exercise.

Include in the contract at least: the locking rules (which locks the core holds at each method call), the sleep rules (which methods may sleep), the return-value conventions, and any side effects the core depends on. If the contract allows methods to be NULL, state it. If some methods are required and some are optional, state that too.

29.6.11 Minimal vs. Rich Interfaces

A recurring design question is how rich the interface should be. Should the backend expose a single `do_transfer` method that covers every transfer type, or a family of specialised methods for read, write, command, and status? Should the core do high-level request queuing and ask the backend to perform each step, or should the backend own the queue and report completions upward?

There is no universal answer, but two principles help.

If the operation differs only in the vocabulary, unify it. `read_reg` and `write_reg` differ only in whether the access is outbound or inbound. They are naturally two methods, but one could reasonably be a single `access_reg(sc, off, direction, valp)`. In practice, having two methods is clearer than a direction flag, because the call sites read better. Split when the split is obvious.

If the operations differ in content, keep them separate. A device might support both “command” transfers and “data” transfers that look completely different at the register level. Fold them into one method and you force the backend to dispatch on a flag inside. Keep them as two methods and the backend’s job becomes obvious. Split when the split is substantive.

A useful test: write the documentation for the method you are considering. If the documentation needs to describe two behaviours distinguished by a parameter, split. If it describes one behaviour with parameters that

vary the value, keep.

29.6.12 The “Soft” Contract of a Backend

Beyond the explicit contract, there is a soft contract about resource management. Each backend should be consistent in who allocates what, when each piece of state is valid, and who frees it at shutdown. The usual convention is:

- The backend’s per-attach code allocates its private state in `sc_backend_priv` and any bus resources it needs. It returns before the core begins using the interface.
- The core allocates its own state (locks, queues, character devices) after the backend is installed, in `port_drv_core_attach`.
- On detach, the core tears down its own state first, then calls the backend’s `detach` method, which releases the backend’s resources and frees `sc_backend_priv`.

This ordering matters. If the core tears down its character device before the backend stops DMA, an in-flight transfer might fire a callback into a destroyed object. If the backend frees `sc_backend_priv` before the core finishes a write, the core’s access panics with a use-after-free. Document the teardown order in the backend contract, and honour it in every backend.

Double-checking the soft contract is an excellent activity to perform when you write a new backend. Walk through the attach and detach paths mentally, keeping track of which object is alive and which is being torn down. Most bugs in a well-written backend are life-cycle bugs, and they are easiest to catch before the code runs than after.

29.6.13 Wrapping Up Section 3

The backend interface is the conceptual spine of a portable driver. Accessors handle small-scale variation like register width and architecture, but the backend interface handles large-scale variation like the entire bus. Once you have an interface, adding a backend is a one-file change, and the core is inoculated against bus-specific concerns. The first time you experience this, you will understand why the pattern is so universal in kernel code. Understanding `kobj(9)` as a more elaborate form of the same idea, and recognising when stacked backends or a careful contract are worth the effort, are the next steps toward writing drivers that scale beyond a single variant.

In Section 4 we turn the backend interface into a concrete file layout: which functions live in which file, which header each file includes, and how the build system pulls everything together. The ideas are the same; the payoff is a working multi-file driver you can build and load.

29.7 Section 4: Splitting Code Into Logical Modules

A portable driver is not usually one file. It is a small family of files, each with a clear responsibility, and a build system that knows how to assemble them. This section takes the backend interface from Section 3 and expresses it as a concrete directory layout that you can type, build, and load. The file layout is not sacred, but it is a well-established convention in FreeBSD, and following it will make your driver look familiar to anyone else who reads it.

The goal is a layout in which each file answers exactly one question. A reader picking up the driver for the first time should be able to guess which file to open based on what they are looking for, without having to `grep`. That is what “logical modules” means in this chapter: not module in the `.ko` sense, but module in the structural sense, a unit of code with one purpose.

29.7.1 The Canonical Layout

Start with the reference driver `portdrv`. After refactoring, it looks like this on disk:

```
portdrv/
├─ Makefile
├─ portdrv.h           # cross-file public prototypes and types
├─ portdrv_common.h   # types shared between core and backends
├─ portdrv_backend.h  # the backend interface struct
├─ portdrv_core.c     # hardware-independent logic
├─ portdrv_pci.c      # PCI backend
├─ portdrv_usb.c      # USB backend (optional, conditional)
├─ portdrv_sim.c      # simulation backend (optional, conditional)
├─ portdrv_sysctl.c   # sysctl tree, helpers
├─ portdrv_ioctl.c    # ioctl handlers, if the driver exposes them
└─ portdrv_dma.c      # DMA helpers, if the driver uses DMA
```

That is the target. Not every driver needs every file. A small driver that does not do DMA has no `portdrv_dma.c`. A driver that has only one backend has no separate backend files and may keep the hardware-specific code in a single `portdrv_pci.c` alongside the core. The layout scales up as the driver grows, and the key idea is that **new responsibilities get new files rather than new sections in an existing file**.

29.7.2 Responsibilities Per File

Let us walk through the intended job of each file, so that the layout feels less abstract.

- **`portdrv.h`**: the public header for the driver. This is what `portdrv_sysctl.c` and other internal files include to see the `softc` and the core entry points. It pulls in common types, forward declares the `softc`, and exposes the functions each file needs to call across file boundaries.
- **`portdrv_common.h`**: the subset of types that the backends also need. Splitting this out from `portdrv.h` is useful because the backend files should not need to see every internal detail of the core. If the `softc` has, say, a field that is only relevant to `ioctl` handling, that field’s type can stay in `portdrv.h`, while the pointer to the backend interface is exposed in `portdrv_common.h`. In smaller drivers you can merge the two headers if the split feels forced.
- **`portdrv_backend.h`**: the single authoritative definition of the backend interface: the struct of function pointers, the constants for backend identification, and the declarations of any helpers shared among backends. Every backend file includes it. The core also includes it. This is the file you open when you want to know what the core expects from a backend.

- **portdrv_core.c**: the hardware-independent logic. Attach and detach for the driver as a whole, the request queue, the softc allocation, the character device or interface registration, the callback invocation path. The core does not include any bus-specific header like `dev/pci/pcivar.h` or `dev/usb/usb.h`. If you find yourself including such a header in `portdrv_core.c`, something has gone wrong.
- **portdrv_pci.c**: everything that knows about PCI. The `device_method_t` array for PCI, the PCI probe, the PCI-specific allocator, the PCI interrupt handler if the driver uses one, and the `DRIVER_MODULE` registration. The PCI backend's implementation of each function in the backend interface lives here. This file includes `dev/pci/pcivar.h` and the other PCI headers; the core does not.
- **portdrv_usb.c**: the USB backend, with its own attach, its own registration through the USB framework, and its own implementation of the backend interface. Parallel in structure to the PCI backend, but reaching the hardware over a completely different API.
- **portdrv_sim.c**: the simulation backend. A pure software implementation that honours the same interface but stores register state in memory and synthesises completions. Useful for testing and CI; essential for development when the real hardware is not available.
- **portdrv_sysctl.c**: the driver's sysctl tree and the helper functions that read and set its variables. Keeping sysctl out of the core has two benefits. First, the core file stays focused on I/O. Second, the sysctl tree becomes easy to extend without adding noise to the core.
- **portdrv_ioctl.c**: the switch statement that dispatches ioctl commands, one helper per command. Large drivers accumulate many ioctls, and moving them to their own file keeps the main flow of the core readable. A small driver with two ioctls can keep them in the core.
- **portdrv_dma.c**: helpers for DMA setup and teardown. The functions `portdrv_dma_create_tag`, `portdrv_dma_alloc_buffer`, `portdrv_dma_free_buffer`, and so on, each of which wraps a `bus_dma` primitive. These helpers are used by the backends but do not depend on any specific bus. Isolating them into their own file makes it obvious what the DMA surface of the driver is.

29.7.3 A Minimal Core File

The core file is where most of your non-trivial logic will live. Here is a sketch of what `portdrv_core.c` looks like, showing just the structural elements. Type it in, build it, and load it; the details are filled out in the lab at the end of the chapter.

```
/*
 * portdrv_core.c - hardware-independent core for the portdrv driver.
 *
 * This file knows about the backend interface and the softc, but
 * does not include any bus-specific header. Backends are installed
 * by per-bus attach paths in portdrv_pci.c, portdrv_usb.c, etc.
 */

#include <sys/param.h>
#include <sys/system.h>
#include <sys/kernel.h>
```



```
#include <sys/module.h>
#include <sys/conf.h>
#include <sys/malloc.h>
#include <sys/mutex.h>

#include "portdrv.h"
#include "portdrv_common.h"
#include "portdrv_backend.h"

static MALLOC_DEFINE(M_PORTDRV, "portdrv", "portdrv driver state");

int
portdrv_core_attach(struct portdrv_softc *sc)
{
    KASSERT(sc != NULL, ("portdrv_core_attach: NULL softc"));
    KASSERT(sc->sc_be != NULL,
        ("portdrv_core_attach: backend not installed"));

    mtx_init(&sc->sc_mtx, "portdrv", NULL, MTX_DEF);

    /* Call the backend-specific attach step. */
    if (sc->sc_be->attach != NULL) {
        int err = sc->sc_be->attach(sc);
        if (err != 0) {
            mtx_destroy(&sc->sc_mtx);
            return (err);
        }
    }

    /* Hardware is up; register with the upper half of the kernel
     * (cdev, ifnet, GEOM, etc. as appropriate). */
    return (portdrv_core_register_cdev(sc));
}

void
portdrv_core_detach(struct portdrv_softc *sc)
{
    portdrv_core_unregister_cdev(sc);
    if (sc->sc_be->detach != NULL)
        sc->sc_be->detach(sc);
    mtx_destroy(&sc->sc_mtx);
}

int
```

```
portdrv_core_submit(struct portdrv_softc *sc, const void *buf, size_t len)
{
    /* All of this logic is hardware-independent. */
    return (sc->sc_be->start_transfer(sc, buf, len));
}
```

Notice what is not in this file. No `#include <dev/pci/pcivar.h>`. No PCI probe. No USB descriptors. No register offsets. The core is pure logic and delegation. It is small enough that a reader can hold the whole file in their head, and it does not change when a new backend is added.

29.7.4 A Minimal Backend File

The per-backend file is the only place where bus-specific headers appear. A sketch of `portdrv_pci.c`:

```
/*
 * portdrv_pci.c - PCI backend for the portdrv driver.
 */

#include <sys/param.h>
#include <sys/system.h>
#include <sys/kernel.h>
#include <sys/module.h>
#include <sys/bus.h>
#include <sys/rman.h>

#include <machine/bus.h>
#include <machine/resource.h>

#include <dev/pci/pcivar.h>
#include <dev/pci/pcireg.h>

#include "portdrv.h"
#include "portdrv_common.h"
#include "portdrv_backend.h"

struct portdrv_pci_softc {
    struct resource *pci_bar;
    int pci_bar_rid;
    struct resource *pci_irq;
    int pci_irq_rid;
    void *pci_ih;
};

static uint32_t
```

```
portdrv_pci_read_reg(struct portdrv_softc *sc, bus_size_t off)
{
    struct portdrv_pci_softc *psc = sc->sc_backend_priv;

    return (bus_read_4(psc->pci_bar, off));
}

static void
portdrv_pci_write_reg(struct portdrv_softc *sc, bus_size_t off, uint32_t val)
{
    struct portdrv_pci_softc *psc = sc->sc_backend_priv;

    bus_write_4(psc->pci_bar, off, val);
}

static int
portdrv_pci_attach_be(struct portdrv_softc *sc)
{
    /* Finish any backend-specific setup after resources are claimed. */
    return (0);
}

static void
portdrv_pci_detach_be(struct portdrv_softc *sc)
{
    /* Tear down any backend-specific state. */
}

const struct portdrv_backend portdrv_pci_backend = {
    .name          = "pci",
    .attach        = portdrv_pci_attach_be,
    .detach        = portdrv_pci_detach_be,
    .read_reg      = portdrv_pci_read_reg,
    .write_reg     = portdrv_pci_write_reg,
    .start_transfer = portdrv_pci_start_transfer,
    .poll_done     = portdrv_pci_poll_done,
};

static int
portdrv_pci_probe(device_t dev)
{
    if (pci_get_vendor(dev) != PORTDRV_VENDOR ||
        pci_get_device(dev) != PORTDRV_DEVICE)
        return (ENXIO);
}
```

```
device_set_desc(dev, "portdrv (PCI backend)");
return (BUS_PROBE_DEFAULT);
}

static int
portdrv_pci_attach(device_t dev)
{
    struct portdrv_softc *sc = device_get_softc(dev);
    struct portdrv_pci_softc *psc;
    int err;

    psc = malloc(sizeof(*psc), M_PORTDRV, M_WAITOK | M_ZERO);
    sc->sc_dev = dev;
    sc->sc_be = &portdrv_pci_backend;
    sc->sc_backend_priv = psc;

    psc->pci_bar_rid = PCIR_BAR(0);
    psc->pci_bar = bus_alloc_resource_any(dev, SYS_RES_MEMORY,
        &psc->pci_bar_rid, RF_ACTIVE);
    if (psc->pci_bar == NULL) {
        free(psc, M_PORTDRV);
        sc->sc_backend_priv = NULL;
        return (ENXIO);
    }

    err = portdrv_core_attach(sc);
    if (err != 0) {
        bus_release_resource(dev, SYS_RES_MEMORY, psc->pci_bar_rid,
            psc->pci_bar);
        free(psc, M_PORTDRV);
        sc->sc_backend_priv = NULL;
    }
    return (err);
}

static int
portdrv_pci_detach(device_t dev)
{
    struct portdrv_softc *sc = device_get_softc(dev);
    struct portdrv_pci_softc *psc = sc->sc_backend_priv;

    portdrv_core_detach(sc);
    if (psc != NULL) {
        if (psc->pci_bar != NULL)
```

```

        bus_release_resource(dev, SYS_RES_MEMORY,
            psc->pci_bar_rid, psc->pci_bar);
    free(psc, M_PORTDRV);
    sc->sc_backend_priv = NULL;
}
return (0);
}

static device_method_t portdrv_pci_methods[] = {
    DEVMETHOD(device_probe,   portdrv_pci_probe),
    DEVMETHOD(device_attach, portdrv_pci_attach),
    DEVMETHOD(device_detach, portdrv_pci_detach),
    DEVMETHOD_END
};

static driver_t portdrv_pci_driver = {
    "portdrv_pci",
    portdrv_pci_methods,
    sizeof(struct portdrv_softc)
};

DRIVER_MODULE(portdrv_pci, pci, portdrv_pci_driver, 0, 0);
MODULE_VERSION(portdrv_pci, 1);
MODULE_DEPEND(portdrv_pci, portdrv_core, 1, 1, 1);

```

Again, notice what is present and what is not. The PCI-specific header includes are all here. The PCI probe is here. The `DRIVER_MODULE` macro is here. None of it is in the core. If you are writing a USB backend next, that file looks structurally identical but includes USB headers instead and registers with the USB subsystem.

29.7.5 Header Organisation

The three headers `portdrv.h`, `portdrv_common.h`, and `portdrv_backend.h` look like this in miniature.

`portdrv.h` is the driver's public face within the driver's files:

```

#ifndef _PORTDRV_H_
#define _PORTDRV_H_

#include <sys/malloc.h>

/* Softc, opaque to anything outside the core. */
struct portdrv_softc;

/* Core entry points called by backends. */

```

```

int  portdrv_core_attach(struct portdrv_softc *sc);
void portdrv_core_detach(struct portdrv_softc *sc);
int  portdrv_core_submit(struct portdrv_softc *sc,
                        const void *buf, size_t len);

MALLOC_DECLARE(M_PORTDRV);

#endif /* !_PORTDRV_H_ */

```

portdrv_common.h contains the types that backends and the core share:

```

#ifndef _PORTDRV_COMMON_H_
#define _PORTDRV_COMMON_H_

#include <sys/param.h>
#include <sys/lock.h>
#include <sys/mutex.h>

/* Forward declaration of the backend interface. */
struct portdrv_backend;

/* The_softc, visible to the backends so they can install sc_be, etc. */
struct portdrv_softc {
    device_t          sc_dev;
    const struct portdrv_backend *sc_be;
    void             *sc_backend_priv;

    struct mtx        sc_mtx;
    struct cdev        sc_cdev;

    /* other common fields ... */
};

#endif /* !_PORTDRV_COMMON_H_ */

```

portdrv_backend.h contains the interface and the declarations of each backend's canonical instance:

```

#ifndef _PORTDRV_BACKEND_H_
#define _PORTDRV_BACKEND_H_

#include <sys/types.h>
#include <machine/bus.h>

```

```
#include "portdrv_common.h"

struct portdrv_backend {
    const char *name;
    int  (*attach)(struct portdrv_softc *sc);
    void (*detach)(struct portdrv_softc *sc);
    uint32_t (*read_reg)(struct portdrv_softc *sc, bus_size_t off);
    void     (*write_reg)(struct portdrv_softc *sc, bus_size_t off,
                        uint32_t val);
    int  (*start_transfer)(struct portdrv_softc *sc,
                        const void *buf, size_t len);
    int  (*poll_done)(struct portdrv_softc *sc);
};

extern const struct portdrv_backend portdrv_pci_backend;
extern const struct portdrv_backend portdrv_usb_backend;
extern const struct portdrv_backend portdrv_sim_backend;

#endif /* !_PORTDRV_BACKEND_H_ */
```

These headers are short on purpose. A header that takes a page to read is a header that is hiding something. Split the types where they are used, and keep each header free of unnecessary prototypes.

29.7.6 The Makefile

The build side of this layout is pleasantly simple in FreeBSD. `bsd.kmod.mk` handles multi-file modules naturally. The Makefile looks like this:

```
# Makefile for portdrv - Chapter 29 reference driver.

KMOD=    portdrv
SRCS=    portdrv_core.c \
        portdrv_sysctl.c \
        portdrv_ioctl.c \
        portdrv_dma.c

# Backends are selected at build time.
.if defined(PORTDRV_WITH_PCI) && ${PORTDRV_WITH_PCI} == "yes"
SRCS+=   portdrv_pci.c
CFLAGS+= -DPORTRV_WITH_PCI
.endif

.if defined(PORTDRV_WITH_USB) && ${PORTDRV_WITH_USB} == "yes"
SRCS+=   portdrv_usb.c
```

```
CFLAGS+= -DPORTDRV_WITH_USB
#endif

.if defined(PORTDRV_WITH_SIM) && ${PORTDRV_WITH_SIM} == "yes"
SRCS+= portdrv_sim.c
CFLAGS+= -DPORTDRV_WITH_SIM
#endif

# If no backend was explicitly selected, enable the simulation
# backend so that the driver still builds and loads.
.if !defined(PORTDRV_WITH_PCI) && \
    !defined(PORTDRV_WITH_USB) && \
    !defined(PORTDRV_WITH_SIM)
SRCS+= portdrv_sim.c
CFLAGS+= -DPORTDRV_WITH_SIM
#endif

SYSDIR?=    /usr/src/sys

#include <bsd.kmod.mk>
```

Build it with any combination of backends:

```
make clean
make PORTDRV_WITH_PCI=yes PORTDRV_WITH_SIM=yes
```

The resulting `portdrv.ko` contains only the files you asked for. Dropping a backend is as easy as removing its make variable. This is the compile-time selection approach, and it works well for drivers whose backend set is known at build time.

29.7.7 Compare with `/usr/src/sys/modules/uart/Makefile`

Open `/usr/src/sys/modules/uart/Makefile` and compare. The structure is similar in spirit: a `SRCS` list that names each file, plus conditional blocks that select backends and hardware-specific helpers based on architecture. The UART build is more elaborate than ours because the driver supports many backends and many hardware variants, but the shape is recognisable. Studying this file alongside the one in the reference driver will make both more understandable.

Take a few minutes to walk through the UART Makefile from the top. Notice how the `SRCS` list is built up through additions, not a single assignment: `SRCS+= uart_tty.c`, `SRCS+= uart_dev_ns8250.c`, and so on. Each addition can be guarded by a conditional that checks an architecture macro or a feature flag. The result is a single Makefile that produces different builds on `amd64`, `arm64`, and `riscv`, with different sets of backends enabled on each. That is the structure you want your own drivers to have once they grow to cover more than one platform.

29.7.8 Makefile Idioms for Portable Drivers

A few Makefile idioms recur so often that they deserve explicit naming. You will see each of these in real FreeBSD module Makefiles and in the reference driver.

The **unconditional core list**:

```
KMOD=    portdrv
SRCS=    portdrv_core.c portdrv_ioctl.c portdrv_sysctl.c
```

This is always the starting point: the files that are always compiled, regardless of configuration.

The **conditional backend list**:

```
.if ${MACHINE_CPUARCH} == "amd64" || ${MACHINE_CPUARCH} == "i386"
SRCS+=    portdrv_x86_helpers.c
.endif

.if ${MACHINE} == "arm64"
SRCS+=    portdrv_arm64_helpers.c
.endif
```

This pattern lets a file be included only on a specific architecture. It is less common in drivers than in kernel subsystems, but it is the right tool when genuinely architecture-specific code exists.

The **per-feature block**:

```
.if defined(PORTDRV_WITH_DMA) && ${PORTDRV_WITH_DMA} == "yes"
SRCS+=    portdrv_dma.c
CFLAGS+=    -DPORTDRV_WITH_DMA
.endif
```

Each feature that can be toggled has one block. The block adds a source file and a preprocessor flag. This is the pattern you will use most often.

The **header dependency**:

```
beforebuild: genheader
genheader:
    sh ${.CURDIR}/gen_registers.sh > ${.OBJDIR}/portdrv_regs.h
```

When a header is generated from a data file (for example, from a vendor-provided register description), this pattern ensures the header is up to date before the main build runs. Use it sparingly; each generated artefact is a maintenance point.

The **CFLAGS discipline block**:

```
CFLAGS+=      -Wall
CFLAGS+=      -Wmissing-prototypes
CFLAGS+=      -Wno-sign-compare
CFLAGS+=      -Werror=implicit-function-declaration
```

This turns on warnings that catch common portability bugs. Treat new warnings as bugs, and fix them as they appear. The `-Werror=implicit-function-declaration` flag in particular catches the class of bug where a file calls a function without including the appropriate header; on some platforms the compiler would silently assume an `int` return type and proceed, producing a latent bug.

The **include-path addition**:

```
CFLAGS+=      -I${.CURDIR}
CFLAGS+=      -I${.CURDIR}/include
CFLAGS+=      -I${SYSDIR}/contrib/portdrv_vendor_headers
```

When the driver has its own header directory, or when it depends on vendor-supplied headers checked into a contrib area, these `-I` additions are how you tell the compiler where to look. Keep the list short; each additional path increases the surface for header conflicts.

29.7.9 When Multiple Modules Share Code

A question that arises as drivers proliferate is how to share helper code among them. The two approaches are:

Library module. A separate `.ko` file provides helper functions, and the dependent drivers declare `MODULE_DEPEND` on it. This is clean when the helpers are substantial, but each module load now requires loading the helper module first.

Shared source file. Multiple drivers list the same `.c` file in their `SRCS`. Each driver gets its own copy of the functions, compiled into its own `.ko`. This is simpler but duplicates code in every driver's binary.

For helpers under a few hundred lines, the shared source approach is usually right. The duplication is small and the build is simpler. For helpers over a thousand lines, the library module pays for itself because updates to the helper affect only one module, not every driver that uses it.

A third approach is to move the helper into the base kernel and have it exported. This is the right choice when the helper is genuinely general-purpose and belongs to the kernel ABI. For driver-specific helpers, it is overkill.

29.7.10 Benefits of the Split

Why go to this much trouble? The benefits are not cosmetic.

Faster builds. A change to `portdrv_ioctl.c` recompiles one file and relinks the module. A monolithic driver recompiles everything every time.

Easier testing. The simulation backend can be loaded alone, without any hardware, and the core can be exercised completely. Unit-testing a driver this way changes what is possible.

Cleaner API surface. Each file exposes only what the others need. When a function is declared in `portdrv.h`, that declaration is a promise that the function is part of the driver's internal API. Functions that are not declared there are implicitly private to their file.

Easier review. A code reviewer who knows the layout can skim a patch by looking at which files changed. A patch that touches `portdrv_pci.c` only is doing PCI-specific work. A patch that touches `portdrv_core.c` is doing something that affects all backends, and deserves closer scrutiny.

Lower cost to add a backend. Once the layout is in place, a new backend is a single new file. In a monolithic driver, a new backend is a rewrite of half the file.

29.7.11 Common Mistakes When Splitting a Driver

Some pitfalls are easy to blunder into.

Moving too little. A split that leaves bus-specific headers in the “core” file has not moved far enough. The core should include only subsystem headers (`sys/bus.h`, `sys/conf.h`, etc.), not bus-specific ones. If you catch yourself including `dev/pci/pcivar.h` in the core, you have left PCI-specific logic behind.

Moving too much. A split that moves every helper into its own file creates a dozen tiny files and a mesh of cross-dependencies. Balance is the goal. If three functions are always used together and are meaningful only together, they belong in the same file.

Circular includes. When header A includes header B and header B includes header A, the compiler complains about incomplete types. The fix is almost always a forward declaration in one of the headers, so the other header can include it safely.

Silent API breakage. When you move a function from `portdrv.c` to `portdrv_sysctl.c`, its declaration must be visible to anyone who calls it. Check the compiler warnings; a missing declaration usually shows up as an implicit function declaration warning before it turns into a link-time error.

Forgotten entries in SRCS. A new file that is not listed in SRCS is not compiled. The driver will compile and link if you are unlucky and all its symbols are already available elsewhere, or it will fail to link with a mysterious undefined-symbol error. When you add a file, always add it to SRCS in the same commit.

29.7.12 Header Hygiene and Include Graphs

A source file's `#include` lines are a window on its dependencies. A file that includes `dev/pci/pcivar.h` is declaring, in the cleanest way a C programmer can, that it knows about PCI. A file that only includes `sys/param.h`, `sys/bus.h`, `sys/malloc.h`, and the driver's own headers is declaring independence from any specific bus. The discipline of keeping include lists small and focused is called header hygiene, and it is the quiet partner of the backend interface.

Three rules keep include graphs clean.

Rule one: include what you use, declare what you do not. If a file uses `bus_dma_tag_create`, it must include `<sys/bus.h>` and `<machine/bus.h>`. If it merely takes a pointer to a `bus_dma_tag_t` but does not dereference it, a forward declaration in a header may be enough. Over-including pulls in dependencies; under-including causes implicit function declaration warnings. Both are fixable, but the right target is exactly what is needed.

Rule two: include the smallest header that gives you what you need. `<sys/param.h>` is a fine catch-all, but it pulls in a large transitive graph. If you only need fixed-width integer types, `<sys/types.h>` may be enough. The build succeeds either way; the compilation time and the dependency graph shrink when you are disciplined.

Rule three: never rely on transitive includes. If your `.c` file uses `memset`, it must include `<sys/libkern.h>` (or `<string.h>` in userland), even if another header has already pulled it in for you. Transitive includes are not part of the contract; they change when another header is refactored. An explicit include is robust; an implicit one is a ticking bug.

When you split a driver according to the layout in this section, re-inspect every `.c` file's includes. Delete the ones you do not need; add the ones you were getting transitively. After the sweep, each file should have an includes list that matches what it actually does. The result is a source tree where a change to one header does not cascade unexpectedly through the rest.

A useful mental model is that the include graph is the **build-time equivalent of the backend interface**. Both declare what depends on what. Both are kept clean by discipline. Both pay for themselves when the driver is refactored. If the backend interface is the dependency graph of the running code, the include graph is the dependency graph of the source code, and both deserve the same care.

29.7.13 Managing CFLAGS and Symbol Visibility

A multi-file driver eventually needs some build-time knobs for its own compilation. These usually take the form of additions to CFLAGS in the Makefile:

```
CFLAGS+=      -I${.CURDIR}
CFLAGS+=      -Wall -Wmissing-prototypes
CFLAGS+=      -DPORTDRV_VERSION='"2.0"'
```

The first line makes the driver's own headers discoverable; the second turns on warnings that catch common portability mistakes; the third embeds a version string directly into the compiled module. Each flag is small; together they give the driver a small surface of control that does not leak into the source.

Be careful with symbol visibility. In a kernel module, every non-static function is a potentially exported symbol. If two drivers both define a function named `portdrv_helper`, the kernel will accept whichever loads first and reject the second, producing a confusing error. Prefix your symbols: name functions `portdrv_core_attach`, not `attach`; name functions `portdrv_pci_probe`, not `probe`. Static functions never become symbols and can be named more tersely.

This is not just a style issue. On FreeBSD, a kernel module's symbol table is shared with every other loaded module. An unprefixed name collides with everything else the kernel has ever loaded. Use a consistent prefix, and prefer `static` whenever the function does not need to be visible outside its file.

29.7.14 A Dependency Graph on Paper

A good exercise when you first split a driver is to draw its include graph on paper. A box for each file; an arrow from file A to file B when A includes B. The graph should be acyclic, flow downward from source files to the most primitive headers, and contain no surprising edges.

For the reference driver `portdrv`, the graph looks roughly like this:

```
portdrv_core.c
|
+--> portdrv.h
+--> portdrv_common.h
+--> portdrv_backend.h ---> portdrv_common.h
+--> <sys/param.h>, <sys/bus.h>, <sys/malloc.h>, ...

portdrv_pci.c
|
+--> portdrv.h
+--> portdrv_common.h
+--> portdrv_backend.h
+--> <dev/pci/pcivar.h>, <dev/pci/pci.h>
+--> <sys/bus.h>, <machine/bus.h>, <sys/rman.h>, ...

portdrv_sim.c
|
+--> portdrv.h
+--> portdrv_common.h
+--> portdrv_backend.h
+--> <sys/malloc.h>
```

The PCI backend is the only file that reaches into `dev/pci/`. The core is the only file that pulls in the universal subsystem headers. The simulation backend is the lightest of the three because it talks to no real hardware.

Draw this graph before you commit a new file layout. If an arrow goes the wrong way, fix the includes before you write any more code. A graph that flows downward is a sign of healthy layering; a graph with cycles is a sign of a design that wants rethinking.

29.7.15 Static Analysis and Continuous Reviews

Multi-file drivers benefit from a small amount of static analysis. The two tools that pay for themselves almost immediately are:

make -k -j buildkernel with warnings turned on. The FreeBSD build is usually clean, but a new file will often expose a forgotten prototype, an unused variable, or a signed/unsigned mismatch. Read every warning; fix every warning.

cppcheck or **scan-build** on the driver sources. Neither is perfect, but each catches a class of bug that the compiler does not. A missing free, a dereference of a possibly-NULL pointer, a dead branch, and a use-after-free are all common hits. Run one of them periodically, not necessarily on every commit.

For a large driver, consider running `clang-tidy` with a moderate check set. For a small driver, the compiler's warnings plus occasional human review are usually enough.

A multi-file driver is easier to review than a monolithic one precisely because each file has a small scope. Reviewers can take a file at a time, understand its responsibility, and check its conformance to the layout. A reviewer who knows the conventions can move through a large patch quickly because each file's role is obvious from its place in the layout. This is the review-cost payoff mentioned earlier, and it compounds over the life of the driver.

29.7.16 Wrapping Up Section 4

You now have a concrete file layout you can apply to any driver: a core file, a small family of backend files, a header per concern, and a Makefile that lets you select backends at build time. The layout is not mandatory, but it matches what real FreeBSD drivers use in the wild, and it scales from small drivers to large ones without reorganisation. Disciplined header hygiene, a clean include graph, and consistent symbol prefixing are the companions of the layout; together they make the driver's source tree a pleasure to navigate.

The layout also teaches the reader where to look. A new maintainer who knows the conventions can navigate the driver on the first day rather than the tenth. Over the life of a long-lived project, that is worth more than it looks.

Section 5 turns from file organisation to the topic that is easiest to get subtly wrong: supporting multiple CPU architectures. The same driver must compile on amd64, arm64, riscv, and the rest of FreeBSD's ports, and it must behave correctly on each one. The tools are almost all about endianness, alignment, and word size, and using them properly is less effort than it looks.

29.8 Section 5: Supporting Multiple Architectures

Architectural portability is the form of portability that requires the least code and the most attention. The kernel and the compiler already do most of the work for you; your job as a driver author is not to add cleverness but to avoid adding bugs. Every common architectural bug in driver code has a corresponding FreeBSD idiom that prevents it. This section introduces those idioms and shows how to use them.

The three axes along which architectures differ in ways that drivers can feel directly are endianness, alignment, and word size. We will take them in order, then look at how to test architectural portability without owning hardware for every architecture.

29.8.1 How `bus_space` Varies Across Platforms

To understand why the abstraction matters, it helps to look briefly at how it is implemented on each of the architectures FreeBSD supports. The mechanism is different on each one, but the interface you program against is the same.

On amd64, the tag is simply a type tag that distinguishes memory-mapped I/O from I/O ports, and the handle is the virtual address of the mapped region. A `bus_space_read_4` call on amd64 becomes, under the covers, a load instruction to the mapped address. The call is a thin wrapper over a raw load; there is essentially no indirection cost.

On arm64, the tag is a pointer to a table of function pointers, and the handle is the base address of the mapped region. A `bus_space_read_4` call on arm64 becomes an indirect call through the function pointer table. The

indirection is necessary because different ARM platforms can map memory with different cacheability and ordering attributes, and the function table encodes those details per platform. The cost is a single indirect call, which is measurable in tight loops but negligible in a typical driver.

On `riscv`, the mechanism is close to `arm64`, with similar rationale. On older `powerpc`, the tag and handle encode endianness configuration, so that a `bus_space_read_4` on a big-endian bus attached to a little-endian CPU does the right byte swap without the driver knowing.

All of this is invisible to the driver. Your code calls `bus_read_4(res, offset)` and gets the right value, on every architecture, at every bus configuration. The price is that you cannot shortcut the API; the moment you drop to a raw pointer, you lose all of this machinery, and your driver works on exactly one platform.

29.8.2 Endianness in Three Questions

Endianness is the order in which the bytes of a multi-byte value are stored in memory. On a little-endian system like `amd64`, `arm64`, or `riscv`, the least significant byte of a 32-bit value comes first in memory, and the most significant byte comes last. On a big-endian system like some configurations of `powerpc`, the order is reversed. The CPU does not see the bytes individually in either case; it reads the word as a whole and interprets it according to the system's endianness. The problem for drivers is that hardware does not always share the CPU's view.

There are three questions a driver author must ask, and three classes of answer.

Question 1: Does the device have a native byte order? Many devices do. A PCI device may specify that a particular register contains a 32-bit value in little-endian order. Ethernet frames, by contrast, are big-endian. When the device's byte order differs from the CPU's byte order, the driver must convert between them explicitly.

Question 2: Does the bus perform any implicit conversion? Usually not. `bus_space_read_4` returns the 32-bit value as the host sees it, in host byte order. If the device stored that value in a different byte order, the conversion is the driver's responsibility. The same is true for DMA buffers read and written through `bus_dma`. The bytes in the buffer are the bytes the device wrote, in whatever order the device prefers; your code has to interpret them correctly.

Question 3: Which direction am I converting? Going from the device to the host, use `le32toh`, `be32toh`, and their 16-bit and 64-bit companions. Going from the host to the device, use `htole32`, `htobe32`, and their companions. The name convention is simple: the first form is the source, the second form is the destination, and `h` stands for host.

29.8.3 The Endian Helpers in Detail

Open `/usr/src/sys/sys/_endian.h` and look at how the helpers are defined. On a little-endian host, `htole32(x)` is just `(uint32_t)(x)` because no swap is needed, and `htobe32(x)` is `__bswap32(x)` because the device expects the opposite order. On a big-endian host, the pattern is reversed: `htobe32(x)` is the identity and `htole32(x)` performs the swap. This means that your code can call the same helper on every architecture, and the right thing happens under the covers.

The full set of helpers you will use in driver code is:

```
htole16(x), htole32(x), htole64(x)
htobe16(x), htobe32(x), htobe64(x)
le16toh(x), le32toh(x), le64toh(x)
be16toh(x), be32toh(x), be64toh(x)
```

Plus the network-byte-order shortcuts `htons`, `htonl`, `ntohs`, `ntohl`, which are equivalent to the `htobe/betoh` family for 16-bit and 32-bit values.

Use them whenever you store a multi-byte value in a device register or DMA buffer, or whenever you read one. Never write:

```
sc->sc_regs->control = 0x12345678; /* Wrong if the device expects LE. */
```

Write instead:

```
widget_write_reg(sc, REG_CONTROL, htole32(0x12345678));
```

The cost at runtime is zero on a little-endian host, because the compiler inlines the helper to a no-op. The benefit on a big-endian host is correctness.

29.8.4 A Simple Mental Rule

When I am teaching this pattern to new driver authors, I use one rule that catches most mistakes before they happen: **every multi-byte value that leaves the CPU or enters it through hardware must pass through an endian helper**. Registers. DMA buffers. Packet headers. Descriptor rings. Every single one. If you see a `*ptr = value` or `value = *ptr` pattern on a multi-byte quantity that is device-visible, and the endpoints do not use an endian helper, that is a bug or a lucky accident, and the only way to tell which is to know the architecture you are running on.

The same rule, applied consistently, means that moving a driver to a big-endian platform becomes almost free. If the endian helpers were in place from the start, the platform change does nothing to the code. If they were not, the move is a hunt through the entire driver for every read and write that touched device memory, and every one of them is a potential bug.

29.8.5 Alignment and the `NO_STRICT_ALIGNMENT` Macro

The second architectural axis is alignment. A 32-bit value is *aligned* at an address that is a multiple of four. On amd64, unaligned accesses are permitted and only slightly slower than aligned ones. On some ARM cores, an unaligned access can produce a bus fault or silently corrupt data depending on the specific instruction and configuration.

The kernel's policy is that driver code should produce aligned accesses. If you cast a buffer pointer to `uint32_t *` and dereference it, the buffer had better be aligned to four bytes. Most kernel allocators give you naturally aligned memory, so this usually happens for free. It becomes tricky when parsing a wire protocol that does

not align its fields, like an Ethernet frame whose IP header is misaligned by two bytes because of the Ethernet header's odd size.

FreeBSD exposes the macro `__NO_STRICT_ALIGNMENT`, defined in `/usr/src/sys/x86/include/_types.h` and its architecture peers, which drivers occasionally check to see whether the platform tolerates unaligned accesses. For general-purpose driver code, you should not rely on it. Instead, use byte-at-a-time accessors, or `memcpy`, to copy multi-byte values out of unaligned memory before interpreting them. The compiler is good at optimising `memcpy(&val, ptr, sizeof(val))` into a single aligned load on architectures that support it, and a byte-at-a-time load on architectures that need it.

A typical pattern for safely reading an unaligned 32-bit value from a byte buffer is:

```
static inline uint32_t
load_unaligned_le32(const uint8_t *p)
{
    uint32_t v;
    memcpy(&v, p, sizeof(v));
    return (le32toh(v));
}
```

This works on every architecture. The `memcpy` handles alignment; the `le32toh` handles endianness. Use this kind of helper whenever you pull a multi-byte value from a wire format or an on-device buffer that may not be aligned.

29.8.6 Word Size: 32 vs 64 Bits

The third architectural axis is word size. On a 32-bit platform, a pointer is four bytes and `long` is typically four bytes. On a 64-bit platform, both are eight bytes. Most of the FreeBSD kernel and all of the common architectures today are 64-bit, but not all of them, and even 64-bit platforms sometimes run 32-bit compatibility code.

The failure mode to watch for is writing `int` or `long` where the size matters. A register value is a specific number of bits, not “whatever `int` happens to be.” If a register is 32 bits, write `uint32_t`. If it is 16 bits, write `uint16_t`. This rule is easy to remember and completely eliminates a whole class of bugs.

FreeBSD exposes these types through `/usr/src/sys/types.h` and its indirect includes. The standard fixed-width typedefs are:

```
int8_t,    uint8_t
int16_t,   uint16_t
int32_t,   uint32_t
int64_t,   uint64_t
```

And the kernel also exposes `intmax_t`, `uintmax_t`, `intptr_t`, and `uintptr_t` when you need to store an integer that must be at least as wide as any other integer, or that must be the width of a pointer.

Do not use `u_int32_t` or `u_char` in new code. They are legacy names that still exist for backward compatibility but are not the preferred style. The convention in modern FreeBSD driver code is the `uint32_t` family.

29.8.7 Bus Address Types

A related family of types governs addresses that the device and the bus use. These are defined per architecture in `/usr/src/sys/amd64/include/_bus.h`, `/usr/src/sys/arm64/include/_bus.h`, and the rest. The key types are:

- `bus_addr_t`: a bus address, typically `uint64_t` on 64-bit platforms.
- `bus_size_t`: a size on the bus, the same underlying width as `bus_addr_t`.
- `bus_space_tag_t`: the opaque tag handed to `bus_space_*` functions.
- `bus_space_handle_t`: the opaque handle into the mapped region.
- `bus_dma_tag_t`: the opaque tag used by `bus_dma`.
- `bus_dmamap_t`: the opaque DMA map handle.

Using these types instead of bare `uint64_t` or `void *` is how the kernel enforces the architectural abstraction. On `amd64`, they may be simple integers; on `arm64`, some are pointers to structures. Your code compiles the same way everywhere and does the right thing on each platform.

29.8.8 Testing on Emulated Architectures

One of the most practical questions is: if you only have an `amd64` machine, how do you test that the driver works on `arm64` or on a big-endian system? The answer is emulation.

QEMU, which FreeBSD supports well, can run `arm64` guests on an `amd64` host. The performance is slower than native, but enough for functional testing. A simple workflow is:

1. Install the FreeBSD `arm64` release into a QEMU image.
2. Boot the image under QEMU with enough disk and RAM.
3. Cross-build the driver (or build it inside the guest).
4. Load the module and run your tests.

FreeBSD's build infrastructure supports cross-building via `make buildkernel TARGET=arm64 TARGET_ARCH=aarch64`. For kernel modules, the same `TARGET` and `TARGET_ARCH` variables apply. You end up with an `arm64` module that you can copy into the `arm64` guest and test there.

For big-endian testing, QEMU's `powerpc64` and `powerpc` targets exist, and FreeBSD has releases for both. Big-endian PowerPC testing is particularly valuable because it is the fastest way to find hidden endianness bugs: the driver that works perfectly on `amd64` but mangles every register on PowerPC has almost certainly missed an `htole32` somewhere.

For everyday work, it is enough to run the driver once on a cross-architecture guest as part of your release process. You do not need to do it on every commit. But you should not skip it entirely, because the bugs it finds are the bugs your users would otherwise find for you.

29.8.9 Case Study: A DMA Descriptor Ring

A concrete example of how these idioms combine is a DMA descriptor ring. Many devices expose a ring of descriptors in shared memory: each descriptor describes one transfer, and the device walks the ring autonomously

as it services transfers. The ring is a prime example of a shared structure where endianness, alignment, word size, and cache coherency all interact.

A typical descriptor might look like this in C:

```
struct portdrv_desc {
    uint64_t addr;      /* bus address of the payload */
    uint32_t len;       /* payload length in bytes */
    uint16_t flags;     /* control bits */
    uint16_t status;    /* completion status */
};
```

Four considerations apply to this structure.

Endianness. The device reads and writes these fields. The byte order in which it interprets them comes from the datasheet. If the device expects little-endian, every write from the CPU must pass through `htole64` or `htole32` as appropriate, and every read from the device must pass through `le64toh` or `le32toh`.

Alignment. The device reads the whole descriptor in one transaction, so the descriptor must be aligned on at least its natural boundary. For this layout, that is 8 bytes because of the 64-bit `addr` field. The `bus_dma_tag_create` call that allocates the ring must set `alignment` to at least 8.

Word size. The `addr` field is 64 bits because the device may need to reach any physical address on a 64-bit system. On a 32-bit system with a 32-bit-only device, you could declare it `uint32_t`, but the portable choice is `uint64_t` with the upper bits zero when unused.

Cache coherency. After the CPU writes a descriptor and before the device reads it, the CPU's caches must be flushed to memory so the device sees the update. After the device writes the status field and before the CPU reads it, the CPU's caches for the ring memory must be invalidated so it does not read a stale value. The `bus_dmamap_sync` call with `BUS_DMASYNC_PREWRITE` handles the first case; with `BUS_DMASYNC_POSTREAD` handles the second.

Putting these together, the code to enqueue a descriptor looks roughly like this:

```
static int
portdrv_enqueue(struct portdrv_softc *sc, bus_addr_t payload_addr, size_t len)
{
    struct portdrv_desc *d;
    int idx;

    /* Pick a ring slot. */
    idx = sc->sc_tx_head;
    d = &sc->sc_tx_ring[idx];

    /* Populate the descriptor in device byte order. */
    d->addr = htole64(payload_addr);
    d->len = htole32((uint32_t)len);
    d->flags = htole16(DESC_FLAG_VALID);
}
```

```

d->status = 0;

/* Ensure the CPU's writes reach memory before the device reads. */
bus_dmamap_sync(sc->sc_ring_tag, sc->sc_ring_map,
    BUS_DMASYNC_PREWRITE);

/* Notify the device. Use a barrier to ensure the notification
 * is seen after the descriptor writes. */
bus_barrier(sc->sc_bar, 0, 0,
    BUS_SPACE_BARRIER_READ | BUS_SPACE_BARRIER_WRITE);
portdrv_write_reg(sc, REG_TX_DOORBELL, (uint32_t)idx);

sc->sc_tx_head = (idx + 1) % sc->sc_ring_size;
return (0);
}

```

Every one of the four considerations is visible in these few lines. The `htole*` calls handle endianness. The `bus_dmamap_sync` call handles cache coherency. The `bus_barrier` call handles ordering. The `uint64_t` for `addr` handles word size.

And, crucially, this code is identical on `amd64`, `arm64`, and `powerpc64`. The helpers become the right operation on each platform under the covers. The driver author writes one version and it works everywhere.

29.8.10 A Register Layout Worth Doing Carefully

Here is a register layout from a hypothetical device, written as a C structure:

```

struct widget_regs {
    uint32_t control;      /* offset 0x00 */
    uint32_t status;      /* offset 0x04 */
    uint32_t data;        /* offset 0x08 */
    uint8_t pad[4];        /* offset 0x0C: padding */
    uint64_t dma_addr;     /* offset 0x10 */
    uint16_t len;         /* offset 0x18 */
    uint16_t flags;       /* offset 0x1A */
};

```

Two things about this are worth noticing. First, padding is explicit. If the register block has a hole between `data` and `dma_addr`, write a `pad[]` member. Do not rely on the compiler to insert padding of the right size, because the compiler does not know about the device. Second, the types match the register widths exactly. A 16-bit `len` field is `uint16_t`, not `int` or `u_int`.

Do not, however, read and write these registers by casting a pointer to the structure. The pattern

```
struct widget_regs *r = (void *)sc->sc_regs_vaddr;  
r->control = htobe32(value);
```

is a portability trap. On amd64 it works. On arm64 it may work or may fault depending on the alignment of the mapping. On some ARM cores it may produce implementation-specific behaviour. The kernel provides `bus_space` for a reason; use it.

The structure above is useful for documentation and for constant names, but the code should access the registers through accessors that use `bus_space` or `bus_read_*/bus_write_*`.

29.8.11 Using the Newer `bus_read_*/bus_write_*` Helpers

FreeBSD provides a slightly friendlier family of functions, `bus_read_1`, `bus_read_2`, `bus_read_4`, `bus_read_8`, and their `bus_write_*` counterparts. They take a `struct resource *` rather than the pair (tag, handle), so they are shorter at the call site. Internally they use `bus_space` under the covers. For modern drivers they are the preferred idiom.

```
uint32_t v = bus_read_4(sc->sc_bar, REG_CONTROL);  
bus_write_4(sc->sc_bar, REG_CONTROL, v | CTRL_START);
```

These functions are defined in `/usr/src/sys/sys/bus.h`. Use them in preference to the older `bus_space_*` forms when you are writing new code; both work correctly on all supported architectures.

29.8.12 Common Mistakes When Supporting Multiple Architectures

A short list of pitfalls, each of which would bite a naive driver on at least one FreeBSD platform.

Using `int` for a register width. If the register is 32 bits, use `uint32_t`. An `int` is the right size on every FreeBSD architecture today, but it is signed, and bit patterns with the top bit set become negative numbers. That bites when you compare the value or shift it.

Casting pointers to device memory. Writing through a volatile `uint32_t *` pointer to a mapped region works on some architectures and not others. Use `bus_read_*/bus_write_*`. Always.

Packing without packing attributes. If you really must use a struct to describe a wire-format layout, use `__packed` to make sure the compiler does not insert alignment padding. But be aware that accessing fields of a packed struct is itself subject to alignment hazards on strict-alignment architectures. A safer pattern is to read fields out of a byte buffer with `memcpy` and endian helpers, as shown earlier.

Assuming `sizeof(long) == 4` or `sizeof(long) == 8`. On 32-bit platforms, `long` is 4; on 64-bit platforms, it is 8. If the size matters, use `uint32_t` or `uint64_t` explicitly.

Forgetting about 32-bit platforms. Even when the mainline FreeBSD world is largely 64-bit, there are still 32-bit ARM deployments, 32-bit MIPS deployments, and legacy i386 deployments. Testing only on amd64 is not a portability guarantee. At minimum, cross-build on a 32-bit target periodically.

29.8.13 Memory Barriers and Ordering

Register and memory accesses do not always reach their targets in the order your code writes them. Modern CPUs reorder loads and stores as an optimisation, and so do bus bridges and memory controllers. On most common amd64 configurations, the hardware's reordering is conservative enough that drivers rarely notice. On arm64, riscv, and other weakly-ordered architectures, the reordering is more aggressive, and a driver that depends on a specific order of register accesses can fail in mysterious ways if it does not tell the hardware about that dependency.

The tool for expressing ordering is the **memory barrier**. FreeBSD provides two families:

- `bus_space_barrier(tag, handle, offset, length, flags)` for ordering with respect to device accesses. The `flags` argument is a combination of `BUS_SPACE_BARRIER_READ` and `BUS_SPACE_BARRIER_WRITE`.
- `atomic_thread_fence_rel`, `atomic_thread_fence_acq`, and friends for ordering with respect to normal memory. These are found in `/usr/src/sys/sys/atomic_common.h` and the per-architecture `atomic` files.

A typical use of `bus_space_barrier` occurs when you arm an interrupt after setting up a DMA descriptor:

```
/* Program the DMA descriptor. */
portdrv_write_reg(sc, REG_DMA_ADDR_LO, htobe32((uint32_t)addr));
portdrv_write_reg(sc, REG_DMA_ADDR_HI, htobe32((uint32_t)(addr >> 32)));
portdrv_write_reg(sc, REG_DMA_LEN, htobe32(len));

/* Ensure the descriptor writes are visible before we arm the IRQ. */
bus_barrier(sc->sc_bar, 0, 0,
    BUS_SPACE_BARRIER_READ | BUS_SPACE_BARRIER_WRITE);

/* Now it is safe to tell the device to start. */
portdrv_write_reg(sc, REG_DMA_CTL, DMA_CTL_START);
```

On a strongly ordered architecture such as amd64, the `bus_barrier` call is essentially free and the barrier is implicit in most cases. On arm64, the barrier is a real instruction that prevents the CPU from reordering the register writes, and without it the device could see the `DMA_CTL_START` register written before the address registers, which is almost certainly a bug.

Use barriers whenever your code depends on a specific order of register accesses, particularly across the handoff between the CPU and the device. Read your datasheet for the cases where a specific write order is required, and bracket those sections with barriers. Omitting them is the fastest way to write a driver that works on your desk and fails in production on a different machine.

29.8.14 Cache Coherency and `bus_dmamap_sync`

The third ordering issue is cache coherency. On some architectures, the CPU has caches that are not automatically coherent with DMA. If you write to a DMA buffer from the CPU and then tell the device to read it, the

device may see stale data that is still in the CPU cache. Conversely, if the device writes into a DMA buffer, the CPU may read stale data that was cached before the write.

FreeBSD handles this through `bus_dmamap_sync`. After writing to a buffer that the device will read, you call:

```
bus_dmamap_sync(sc->sc_dmat, sc->sc_dmap, BUS_DMASYNC_PREWRITE);
```

After the device has written to a buffer that the CPU will read, you call:

```
bus_dmamap_sync(sc->sc_dmat, sc->sc_dmap, BUS_DMASYNC_POSTREAD);
```

On amd64, where the caches are coherent with DMA, these calls are usually no-ops. On architectures where they are not coherent, `bus_dmamap_sync` emits the right cache-flush or cache-invalidate operation. A driver that uses `bus_dma` but skips these sync calls works on amd64 and fails on other architectures.

As with endian helpers, the rule is uniform: every DMA buffer that is shared between the CPU and the device must pass through `bus_dmamap_sync` at the handoff. The buffer goes from CPU to device: `BUS_DMASYNC_PREWRITE`. The buffer comes back from device to CPU: `BUS_DMASYNC_POSTREAD`. The sync calls cost nothing on coherent systems and are essential on non-coherent ones. Use them consistently, and architectural portability for DMA buffers is largely free.

29.8.15 Bounce Buffers and Address Limits

Some devices cannot reach every address in the system's physical memory. A 32-bit device on a 64-bit system can only address the first four gigabytes of physical memory; anything above that is unreachable. FreeBSD's `bus_dma` handles this through **bounce buffers**: if the physical address of a buffer is outside the device's reach, `bus_dma` allocates a temporary buffer that is within reach, copies data between the real buffer and the bounce buffer, and presents the device with the bounce buffer's address.

Bounce buffers are invisible to the driver if the driver uses `bus_dma` correctly. The `lowaddr` and `highaddr` fields of `bus_dma_tag_create` tell the kernel what the device can and cannot reach. If the driver sets `lowaddr` to `BUS_SPACE_MAXADDR_32BIT`, the kernel will use bounce buffers for any buffer above 4 GB. The `bus_dmamap_sync` calls handle the copy automatically.

Bounce buffers are not free. They cost memory, they cost a copy per operation, and they cost some latency. Well-designed devices avoid them by supporting 64-bit addressing. But a driver author has no control over the device's address width; the only portability-relevant decision is to declare the limits correctly and let `bus_dma` handle the rest. A driver that lies about the device's limits will corrupt memory silently on a large system. A driver that tells the truth will work on every system, with bounce buffers when needed.

29.8.16 Architectural Smoke Tests

Architectural portability is verified by running the driver on a different architecture. Even one such test, once, catches a surprising amount of latent bugs. The minimum viable smoke test is:

1. Build or download a FreeBSD release for a different architecture, typically `arm64` or `powerpc64`.

2. Install the release into a QEMU guest.
3. Build the driver inside the guest, or cross-build it on the host.
4. Load the module.
5. Exercise the driver's main features.
6. Observe whether the driver behaves as expected.

Even this minimum is revealing. The first time you run your driver on a big-endian host, any missing `htole32` or `be32toh` will manifest as obviously wrong register values. The first time you run it on `arm64`, any raw pointer cast to device memory will usually cause an alignment fault on the first access. The first time you build it on `riscv`, any hard-coded assumption about instruction set extensions will fail at compile time. Each bug takes minutes to find because its symptom is immediate.

You do not need to do this on every commit. Once per release is enough for most drivers. The cost is low (a QEMU guest and an hour of your time), and the bugs you catch are the bugs your users would otherwise report.

29.8.17 The Realities of Non-x86 FreeBSD

A practical note on which architectures matter in the FreeBSD world today. At the time of writing, `amd64` is the overwhelmingly dominant production architecture, followed by `arm64` (growing rapidly in embedded and server deployments), `riscv` (growing in research and specific embedded niches), and `powerpc64` (used in certain HPC and legacy environments). The older `i386` platform is still supported for legacy deployments. The ARM 32-bit family (`armv7`) is supported but less common than `arm64`.

For a driver that is expected to live in the mainline tree, working correctly on `amd64`, `arm64`, and at least compiling on `riscv` is usually sufficient. The `powerpc64` port is a useful test for big-endian issues even if you do not have end users on the platform. The `i386` port is a useful test for 32-bit word-size issues. Supporting all of these is the project-wide default; dropping support for any of them is a decision that needs a specific reason.

For a driver that is maintained outside the tree, the question is whose hardware you are supporting. If your users are on `amd64` only, you can legitimately target only that platform. But the habit of writing portable code costs little and protects against surprise, and you will often find that the same driver turns out to be useful on a platform you did not originally target.

29.8.18 Aside: CHERI-BSD and Morello

One corner of the FreeBSD universe sits outside the production architectures and deserves a brief note for curious readers. If you ever encounter a board called **Morello**, or an operating system tree called **CHERI-BSD**, you are looking at a research platform rather than a mainline target. It is worth knowing what these names mean, because they change assumptions that this chapter has otherwise treated as universal.

What CHERI is. CHERI stands for Capability Hardware Enhanced RISC Instructions. It is an instruction-set extension developed jointly by the University of Cambridge Computer Laboratory and SRI International. The goal is to replace the plain pointer model that C has used since the 1970s with **capability pointers**: pointers that carry their own bounds, permissions, and a hardware-checked validity tag. A capability is not just an integer address. It is a small object that the processor knows how to dereference, and it traps if the program tries to use it outside the region it was granted access to.

The practical effect is that many classes of memory-safety bug become hardware-detectable. A buffer overflow that walks off the end of an array no longer corrupts adjacent memory; it traps. A use-after-free no longer quietly reads whatever lives at the old address; the freed capability can be revoked so its tag bit becomes invalid, and any attempt to dereference it traps. Pointer forgery in the classical sense becomes impossible, because capabilities cannot be synthesised from plain integers.

What Morello is. Morello is Arm's prototype implementation of CHERI on top of the Armv8.2-A instruction set. The Morello Program, announced by Arm together with UK Research and Innovation in 2019, produced the first widely available CHERI-capable hardware: development boards and a reference system-on-chip that researchers, universities, and some industry partners could obtain and program. The Morello board is the closest thing the world currently has to a production-grade CHERI machine, but it is explicitly a research prototype. It is not sold as a general-purpose server platform, and it is not expected to become one in its current form.

What CHERI-BSD is. CHERI-BSD is a fork of FreeBSD maintained by the Cambridge CHERI project that targets CHERI hardware, including Morello. It is the operating system in which most of the practical work of porting userland and kernel code to a capability architecture takes place. CHERI-BSD is a research platform. It tracks FreeBSD closely but is not a mainline FreeBSD release, it is not supported by the FreeBSD Project in the same sense as amd64 or arm64, and it is not a target that a typical driver author needs to ship for today. Its purpose is to let researchers and early adopters study what happens to a real operating system kernel when the pointer model changes beneath it.

What a driver author needs to know. Even if you never touch CHERI-BSD, it is useful to know the shape of the assumptions it breaks, because those assumptions appear in ordinary driver code without anyone noticing. Three points stand out.

First, **capability pointers carry sub-object bounds**. A pointer into the middle of a `struct softc` is not merely a byte address; it is bounded to whatever subrange the allocator or language construct gave it. If a driver uses pointer arithmetic to walk from one field to another without going through the enclosing pointer, that arithmetic can trap on CHERI even when it appeared harmless on amd64. The remedy is usually to derive pointers from the structure's base pointer using proper C idioms and to avoid casts that erase type information. This is the same discipline this chapter has been recommending for non-CHERI portability; CHERI simply makes the failure mode immediate rather than occasional.

Second, **freed capabilities can be revoked**. CHERI-BSD can sweep memory to invalidate capabilities whose backing allocation has been released, so that dangling pointers cannot be used even indirectly. A driver that stashes a pointer in a `sysctl` node, a device-tree entry, or some cross-subsystem handle, and then frees the underlying allocation without telling the kernel, can leave behind a revoked capability. On amd64 the use-after-free might simply read garbage; on CHERI it traps deterministically. That is generally a good thing, but it means the driver's lifecycle discipline has to be honest about every reference it hands out.

Third, **kernel APIs look similar but carry capability semantics underneath**. The `bus_space(9)` and `bus_dma(9)` interfaces look the same at the source level on CHERI-BSD as on mainline FreeBSD, but the values they return are capabilities rather than plain addresses. A driver that uses these APIs through the intended accessors will usually port with little change. A driver that casts a `bus_space_handle_t` to a `uintptr_t` and then back, or that manufactures pointers to device memory by integer arithmetic, will break, because the capability metadata is lost in the integer round-trip.

Some drivers port to CHERI-BSD with minimal changes because they already stayed inside the accessors this chapter has been recommending. Others need explicit **capability discipline**: reviewing every cast, every hand-

rolled pointer arithmetic, and every place where a pointer is stored in a non-pointer type. The CHERI-BSD project publishes porting experience reports for various subsystems, and reading one or two of them is a fast way to see what that discipline looks like in practice. The volume of change needed varies widely by driver, and no one should assume either that a given driver is trivially CHERI-clean or that it needs a total rewrite, without actually trying.

Where to look. The foundational technical description is the 2014 ISCA paper *The CHERI Capability Model: Revisiting RISC in an Age of Risk*, which is listed in the FreeBSD documentation tree at `/usr/src/share/doc/papers/bsdreferences.bib`. Subsequent papers from the same group describe the Morello implementation, capability revocation, compiler support, and real-world porting experience. For practical materials, the CHERI-BSD project and the Cambridge Computer Laboratory’s CHERI research group maintain documentation, build instructions, and porting notes; search for the project names rather than relying on a URL printed here, because research project sites move over time.

The point of mentioning CHERI-BSD in a portability chapter is not to push you toward it. It is to tell you that the habits this chapter has been teaching, using accessors rather than raw pointers, respecting the distinction between addresses and opaque handles, and avoiding gratuitous casts, pay dividends on more than one architecture. They are the same habits that make a driver survive on `arm64`, on `riscv`, and on experimental capability machines. The further you stay from clever pointer tricks, the smaller your driver’s CHERI surprise surface becomes, even if you never compile for CHERI at all.

29.8.19 Wrapping Up Section 5

You have met the three axes along which architectures differ, and the FreeBSD idioms that address each. Endianness is handled by the endian helpers; alignment by `bus_read_*` and `memcpy`; word size by fixed-width types. Memory barriers handle the ordering of accesses that are visible to the device, and `bus_dmamap_sync` handles the coherency between CPU caches and DMA buffers. Used consistently, these idioms reduce architectural portability to almost no extra work per driver and almost no extra risk. A driver that uses them well on `amd64` will usually compile cleanly and run correctly on `arm64`, `riscv`, `powerpc`, and the others with nothing more than a recompile and a smoke test.

In Section 6, we turn from architectural portability to build-time flexibility: how to use conditional compilation and kernel build options to select features, enable debugging, and switch between real and simulated backends without cluttering the source with `#ifdef` soup.

29.9 Section 6: Conditional Compilation and Build-time Flexibility

Conditional compilation is the art of making one source tree produce different binaries depending on what was requested at build time. It is tempting, it is sometimes necessary, and it is spectacularly easy to abuse. A driver that uses `#ifdef` thoughtfully is cleaner and more maintainable than one that does not. A driver that reaches for `#ifdef` at every decision becomes a thicket that no one wants to touch.

This section explains when conditional compilation is the right tool, which forms of it are preferred in FreeBSD, and how to keep the source readable in the presence of real build-time variation.

29.9.1 The Three Levels of Conditional Compilation

Conditional compilation in a FreeBSD driver happens at three distinct levels. Each has a different purpose, and choosing the right level for a given problem prevents most of the ugliness that conditional compilation is infamous for.

Level one: architecture-specific code. Sometimes you must write different code for different CPU architectures. An example is a specific byte-swapping inline assembly for a platform that lacks a general compiler builtin. FreeBSD handles this by putting the per-architecture code in a per-architecture file. Look at how `/usr/src/sys/amd64/include/_bus.h` differs from `/usr/src/sys/arm64/include/_bus.h`. Each file contains the definitions appropriate for its architecture. The core source does not use `#ifdef __amd64__` to select between them; it includes `<machine/bus.h>`, and the kernel build picks the right concrete file via include paths. This mechanism is called the **machine-specific header**, and you should prefer it whenever you need more than a few lines of per-architecture code. If you put a large block of `#ifdef __arm64__` in a driver source file, you are probably reinventing this mechanism badly.

Level two: optional features. Optional features are things that your driver can be built with or without, and whose presence is a choice, not an architectural necessity. Simulation backends, debug tracing, sysctl nodes, optional protocol support. FreeBSD handles these through the kernel options system: you add options `PORTDRV_DEBUG` to the kernel configuration or pass `-DPORTDRV_DEBUG` to make. Inside the code, you guard the relevant blocks with `#ifdef PORTDRV_DEBUG`. Because the flag is yours, you control it cleanly, and the feature is either compiled in or not.

Level three: API compatibility. FreeBSD evolves. A macro gets renamed, a function signature changes, a behaviour that used to be implicit becomes explicit. A driver that wants to compile on both an older release and a newer release uses `__FreeBSD_version` to branch on the kernel version. This is the most subtle form of conditional compilation, because the same logical operation is expressed two ways and you must maintain both until the older version is no longer supported. The right approach is to keep these branches small and to refactor them into short helper functions or macros.

29.9.2 The `__FreeBSD_version` Macro

Every FreeBSD kernel defines `__FreeBSD_version` in `/usr/src/sys/sys/param.h`. It is a single integer that encodes the release number: for example, 1403000 corresponds to FreeBSD 14.3-RELEASE. You can use it to guard code that depends on a specific API change:

```
#if __FreeBSD_version >= 1400000
    /* Use the new if_t API. */
    if_setflags(ifp, flags);
#else
    /* Use the older direct struct access. */
    ifp->if_flags = flags;
#endif
```

Two rules about `__FreeBSD_version`. First, use it sparingly; every guard is a branch that must be tested in both configurations. Second, abstract the branch into a helper as soon as it appears more than twice. If you need

`if_setflags` in ten places, write a helper macro in a compatibility header and use it everywhere, rather than duplicating the `#if` at every call site.

Compatibility is a compounding cost. Two `#if` blocks are manageable, and twenty are a maintenance burden. Make every one count, and retire them as soon as the old release drops out of support.

For a feel of what this looks like in tree, open `/usr/src/sys/dev/gve/gve_main.c`. The Google Virtual Ethernet driver is current, in-tree, and supports several releases at once, so its `__FreeBSD_version` guards document recent API migrations with minimum ceremony. Two short examples stand out. The first is in the attach path, where the driver sets the interface flags:

```
#if __FreeBSD_version >= 1400086
    if_setflags(ifp, IFF_BROADCAST | IFF_SIMPLEX | IFF_MULTICAST);
#else
    if_setflags(ifp, IFF_BROADCAST | IFF_SIMPLEX | IFF_MULTICAST | IFF_KNOWSEPOCH);
#endif
```

Older releases required drivers to set `IFF_KNOWSEPOCH` to declare that they used the network epoch correctly. From 1400086 onward the kernel no longer needs that opt-in, and drivers stop setting the flag. The branch is small, the accessor call (`if_setflags`) is the same on both sides, and only the flag list differs. This is the minimal shape these guards should take.

The second example is in the module declaration, at the end of the same file:

```
#if __FreeBSD_version < 1301503
static devclass_t gve_devclass;

DRIVER_MODULE(gve, pci, gve_driver, gve_devclass, 0, 0);
#else
DRIVER_MODULE(gve, pci, gve_driver, 0, 0);
#endif
```

At 1301503, the `DRIVER_MODULE` macro stopped taking a `devclass_t` argument; newer releases pass `0` in its place and older releases still need a real `devclass_t`. Both branches still register the driver with `DRIVER_MODULE`; the difference is a single argument and the static variable that feeds it. When your own driver needs to span this boundary, your guards should look as plain as these. If they do not, it is usually a sign that the compatibility shim wants to live in a small helper macro rather than at every call site.

29.9.3 The Kernel Options Mechanism

For driver-specific options, FreeBSD's kernel options machinery gives you a clean way to expose feature flags. The mechanism has three parts:

1. The option is declared in a per-driver options file that lives alongside the driver's source, for in-tree drivers. For out-of-tree drivers like your reference driver, you define the option via a `CFLAGS+= -DPORTDRV_SOMETHING` in the Makefile and guard code with `#ifdef PORTDRV_SOMETHING`.

2. The option is set or unset at build time. For kernel builds, it goes into the kernel configuration file (`/usr/src/sys/amd64/conf/GENERIC` or a custom file) as options `PORTDRV_SOMETHING`. For module builds, it goes into the Makefile or on the make command line as `-DPORTDRV_SOMETHING`.
3. The code reads the option with `#ifdef PORTDRV_SOMETHING`.

This is a slim mechanism, but it is enough for most needs. A driver that uses it carefully has a handful of options, each guarding a small, well-defined piece of functionality, and each with a plain-English description in the README.

29.9.4 Simulated Mode as a Build Option

A good example of the kernel-options pattern is a simulated mode. The idea is to let the driver be built and loaded without any real hardware, by selecting a software backend that mimics the device. You have already seen this pattern in Section 3; now we express it as a compile-time flag.

The Makefile adds:

```
.if defined(PORTDRV_WITH_SIM) && ${PORTDRV_WITH_SIM} == "yes"
SRCS+= portdrv_sim.c
CFLAGS+= -DPORTDRV_WITH_SIM
.endif
```

The core registers the simulated backend in its module load if the flag is set:

```
static int
portdrv_modevent(module_t mod, int type, void *arg)
{
    switch (type) {
        case MOD_LOAD:
#ifdef PORTDRV_WITH_SIM
            /* Create a single simulated instance. */
            portdrv_sim_create();
#endif
            return (0);
        case MOD_UNLOAD:
#ifdef PORTDRV_WITH_SIM
            portdrv_sim_destroy();
#endif
            return (0);
    }
    return (0);
}
```

And `portdrv_sim.c` is only compiled if the flag is set.

This pattern has several benefits at once. The driver can be developed on a machine without the real hardware. The test suite can be run on any FreeBSD VM. Reviewers can load the driver and exercise the core logic, which is usually where the interesting bugs are, without needing the hardware. And the production builds, which do not pass `PORTDRV_WITH_SIM=yes`, exclude the simulation code entirely so that no stub backend is present in shipping binaries.

29.9.5 Debug Builds

The second typical option is a debug build. Real drivers usually have a `PORTDRV_DEBUG` option that turns on verbose logging, additional assertions, and sometimes slower paths that validate invariants more aggressively. The pattern is:

```
#ifdef PORTDRV_DEBUG
#define PD_DBG(sc, fmt, ...) \
    device_printf((sc)->sc_dev, "DEBUG: " fmt "\n", ##__VA_ARGS__)
#else
#define PD_DBG(sc, fmt, ...) do { (void)(sc); } while (0)
#endif
```

Everywhere in the source that needs a conditional debug message uses `PD_DBG(sc, "got %u bytes", len)`. In a release build, the macro evaluates to nothing, so the message is compiled away entirely. In a debug build, the macro prints. The call sites are not guarded with `#ifdef`; the macro is.

This is the key trick for keeping conditional compilation readable: **hide the `#ifdef` inside a macro, and expose a uniform call at the use site**. The source reads the same way in both configurations, and the reader does not have to mentally unroll half a dozen `#if` branches to understand the code.

29.9.6 Feature Flags Through `sysctl` and `dmesg`

A related but distinct question is: how does the user or an operator discover what features this driver was built with? A production driver typically answers this with a `sysctl` tree and a printed banner at module load.

At module load, the driver prints a short message that identifies itself and the features it supports:

```
printf("portdrv: version %s loaded; backends:"
#ifdef PORTDRV_WITH_PCI
    " pci"
#endif
#ifdef PORTDRV_WITH_USB
    " usb"
#endif
#ifdef PORTDRV_WITH_SIM
    " sim"
#endif
    "\n", PORTDRV_VERSION);
```

The output in the kernel log tells an operator exactly which backends are compiled in. A build that is supposed to have USB support but whose dmesg shows only backends: `pci` reveals the misconfiguration at a glance.

At runtime, a `sysctl` tree exposes the same information machine-readably:

```
dev.portdrv.0.version: 2.0
dev.portdrv.0.backend: pci
dev.portdrv.0.features.dma: 1
dev.portdrv.0.features.debug: 0
```

A script can read these, a monitoring system can alert on them, and a bug report can include them without the reporter having to guess.

29.9.7 Avoiding `#ifdef` Sprawl

The biggest single pitfall of conditional compilation is its tendency to sprawl. A feature is added and guarded with `#ifdef FEATURE_A`. A month later another feature is added and guarded with `#ifdef FEATURE_B`. Six months later a maintainer adds a bug fix that touches both features, and it acquires `#if defined(FEATURE_A) && defined(FEATURE_B)`. Soon every function has five guards, each guarding a handful of lines, and the reader cannot trace a single path through the code without mentally resolving a cross-product of `#if` branches.

Three rules keep sprawl in check.

Keep the guards coarse. Guard an entire function, or an entire file, rather than scattered lines in the middle of a function. If you find yourself adding `#ifdef` inside the body of a function more than once, the function should probably be split: one version for `FEATURE_A`, one for `!FEATURE_A`, and a small dispatcher at the call site.

Hide the guards in macros. If you need a conditional operation in many places, wrap it in a macro or inline function that has an empty form in the `#else` branch. The call sites then do not have guards at all.

Review the guard set periodically. A feature flag that no one has disabled for two years is effectively mandatory. Delete it. A flag that guards a feature that was removed is dead code. Delete that too. A flag that means something different now than when it was introduced should be renamed.

29.9.8 Architecture-Specific Conditionals

There are situations where the right tool truly is an `#if` on the architecture macro. An example from real code, from `/usr/src/sys/dev/vnic/nicvf_queues.c`:

```
#if BYTE_ORDER == BIG_ENDIAN
    return ((i & ~3) + 3 - (i & 3));
#else
    return (i);
#endif
```

This function computes an index whose layout depends on the byte order of the host. There is no way to hide this in a macro because the two forms are genuinely different. The right thing to do is exactly what the real driver does: guard the minimum possible, comment what the two branches do, and move on. Do not try to eliminate every `#if`.

The key is discipline. Use `#if BYTE_ORDER` only where a real architectural difference exists, and keep the conditional block as small as possible. A five-line `#if` block is readable. A fifty-line one usually is not.

29.9.9 Optional Subsystem Support

Sometimes a driver supports features that depend on optional subsystems. For example, a network driver might support checksum offload only if the kernel was built with options `INET`. The idiom is:

```
#ifdef INET
    /* Set up IPv4 checksum offload. */
#endif
#ifdef INET6
    /* Set up IPv6 checksum offload. */
#endif
```

These guards are provided by the kernel build system automatically; you do not need to define them yourself. But you do need to respect them. A driver that uses `struct ip` unconditionally will fail to build when options `INET` is not set, even though the kernel supports compiling without IPv4. Check the build in the configuration your users might use; it is usually just a matter of adding a guard.

29.9.10 Avoiding Runtime Feature Checks That Should Be Compile-Time

A subtle mistake is to implement what ought to be a compile-time decision at runtime. If your driver has a simulation backend and a PCI backend, and the user picks one at build time, there is no reason to carry both in the binary. Making the choice at runtime, for example through a `sysctl` that toggles which backend is used, wastes memory and complicates testing. Make the choice at build time, exclude the unused backend from the build, and the driver is smaller and simpler.

The converse error is also possible: turning a runtime choice into a compile-time one. If the same binary must support multiple pieces of hardware simultaneously, the choice is genuinely runtime, and a compile-time flag is the wrong tool. The right question to ask each time is: “Do I need both behaviours in the same loaded driver?” If yes, runtime. If no, build-time.

29.9.11 Options Files and Kernel Configuration

For drivers that ship with the kernel tree itself, FreeBSD provides a more structured way to expose build options: the options file. Each options file sits next to the driver’s source and lists the options the driver recognises, along with a short description and a default value. The file is named `options` and follows a simple format:


```
# options - portdrv options
PORTDRV_DEBUG
PORTDRV_WITH_SIM
PORTDRV_WITH_PCI
```

Options listed this way can be enabled in the kernel configuration file, typically in `/usr/src/sys/amd64/conf/GENERIC` or a custom kernel config, as:

```
options PORTDRV_DEBUG
options PORTDRV_WITH_SIM
```

The kernel build system takes care of propagating the `-D` flags to the compiler. Inside the driver, the code guards with `#ifdef PORTDRV_DEBUG` exactly as it would for a Makefile-defined option. The difference is organisational: options files live in the source tree next to the driver, so a reader picking up an unfamiliar driver knows exactly which options it recognises.

For out-of-tree drivers, the Makefile-based approach is simpler and is what we have used in this chapter. For in-tree drivers that expect to be built as part of a kernel configuration, the options file is the canonical mechanism.

29.9.12 Module Load Hints and `device.hints`

A related but distinct topic is the `device.hints` mechanism. Hints are a way to configure devices at boot time, before the driver has fully initialised. They live in `/boot/device.hints` and take the form:

```
hint.portdrv.0.mode="simulation"
hint.portdrv.0.debug="1"
```

Inside the driver, these are read with `resource_string_value` or `resource_int_value`:

```
int mode;
const char *mode_str;

if (resource_string_value("portdrv", 0, "mode", &mode_str) == 0) {
    /* Apply the hint. */
}
```

Hints are a way to configure the driver at runtime without recompilation. They are especially useful for embedded systems where the device parameters are known at system design time but not at driver compile time. For a driver that is portable across hardware variants, hints can be the mechanism by which each variant is recognised.

Hints are not a replacement for compile-time feature selection. They are complementary. Compile-time flags decide what the driver can do; hints decide what it actually does at a given boot. A driver can have both, and most mature drivers do.

29.9.13 Feature Flags Through `MODULE_PNP_INFO`

For drivers that attach to hardware identified by vendor and device IDs, `MODULE_PNP_INFO` is a way to declare that identification in the module's metadata. The kernel and userland tools read this metadata to decide which driver to load when a specific piece of hardware is present, and `devd(8)` uses it to match devices to modules.

A typical declaration looks like this:

```
MODULE_PNP_INFO("U32:vendor;U32:device", pci, portdrv_pci,  
    portdrv_pci_ids, nitems(portdrv_pci_ids));
```

where `portdrv_pci_ids` is an array of identification records that the driver's probe function also consults. The metadata is embedded in the `.ko` file and is read by `kldxref(8)` to build an index used at boot and at device attachment.

For a portable driver with multiple backends, each backend declares its own `MODULE_PNP_INFO`. The PCI backend declares PCI IDs; the USB backend declares USB IDs; the simulation backend declares nothing because it has no hardware. The kernel picks up the right backend automatically when the matching hardware is present.

29.9.14 The Limits of `__FreeBSD_version`

`__FreeBSD_version` is valuable, but it has a limit. It tells you what release of the kernel you are compiling against, not what release is actually running. Since kernel modules are tightly coupled to the kernel they load into, these are usually the same, but there are edge cases: modules built on a slightly older tree and loaded into a slightly newer one, or modules built with a custom patch set. In these cases, `__FreeBSD_version` may mislead you.

The more robust approach for runtime decisions is to query the kernel directly through a `sysctl`:

```
int major = 0;  
size_t sz = sizeof(major);  
  
sysctlbyname("kern.osreldate", &major, &sz, NULL, 0);  
if (major < 1400000) {  
    /* Older kernel. Use the legacy code path. */  
}
```

This reads the actual running kernel's version. For most drivers it is overkill; for drivers that must adapt at runtime, it is the right tool. The usual compile-time `__FreeBSD_version` is a statement about the kernel at build time; `kern.osreldate` is a statement about the kernel at run time.

For driver code specifically, the compile-time check is almost always what you want. The module is almost always loaded into the kernel it was built against, and the kernel's ABI check via `MODULE_DEPEND` catches the rare mismatches.

29.9.15 Organising Compatibility Shims

When `__FreeBSD_version` guards begin to appear in more than a few places, the right move is to centralise them in a compatibility header. The typical layout is:

```
/* portdrv_compat.h - compatibility shims for FreeBSD API changes. */

#ifndef _PORTDRV_COMPAT_H_
#define _PORTDRV_COMPAT_H_

#include <sys/param.h>

#if __FreeBSD_version >= 1400000
#include <net/if.h>
#include <net/if_var.h>
#define PD_IF_SETFLAGS(ifp, flags)  if_setflags((ifp), (flags))
#define PD_IF_GETFLAGS(ifp)        if_getflags(ifp)
#else
#define PD_IF_SETFLAGS(ifp, flags)  ((ifp)->if_flags = (flags))
#define PD_IF_GETFLAGS(ifp)        ((ifp)->if_flags)
#endif

#if __FreeBSD_version >= 1300000
#define PD_CV_WAIT(cvp, mtxp)  cv_wait_unlock((cvp), (mtxp))
#else
#define PD_CV_WAIT(cvp, mtxp)  cv_wait((cvp), (mtxp))
#endif

#endif /* !_PORTDRV_COMPAT_H_ */
```

The core code uses `PD_IF_SETFLAGS(ifp, flags)` and `PD_CV_WAIT(cvp, mtxp)` in place of the bare kernel calls. The compatibility header is the one place where `__FreeBSD_version` appears, and the guards are close to each other rather than scattered. When the oldest supported release is dropped, you delete the `else` branches and leave the modern names in place.

This pattern is especially valuable for drivers that must support a range of kernel releases, such as out-of-tree drivers consumed by users on different systems. An in-tree driver typically only has to support the current release plus whichever earlier ones the project promises to support, which is usually one or two.

29.9.16 Build-Time Telemetry

A small but useful trick is to embed build-time telemetry in the module. A single line in the Makefile:

```
CFLAGS+=      -DPORTDRV_BUILD_DATE='"${:!date "+%Y-%m-%dT%H:%M:%S"!}'"
```

captures the build timestamp into a string the driver can print at load time. Combined with a version string and a list of enabled backends, the module's load banner becomes a small diagnostic artefact:

```
portdrv: version 2.0 built 2026-04-19T14:32:17 loaded, backends: pci sim
```

When a user reports a bug, they can paste the banner into the bug report, and you know exactly which version they are running. This is trivial to add and immensely valuable over the life of the driver.

29.9.17 Wrapping Up Section 6

Conditional compilation is a power tool. Used with restraint, it keeps optional features clean, lets you support multiple FreeBSD releases, and enables simulation backends without polluting production builds. Used carelessly, it turns source into a maze. The rules for restraint are: prefer machine-specific headers over `#ifdef __arch__`, keep option flags coarse, hide guards inside macros, and delete flags that have outlived their purpose. Options files, device hints, `MODULE_PNP_INFO`, and a compatibility header give you a small family of tools for managing build-time variation without cluttering the core logic.

Section 7 takes a brief detour outside FreeBSD. Drivers sometimes need to share code with NetBSD or OpenBSD, and while that is not a primary goal of this book, knowing the lay of the land saves you from design choices that would hurt cross-BSD portability even when you do not immediately need it.

29.10 Section 7: Adapting for Cross-BSD Compatibility

The three open-source BSDs, FreeBSD, NetBSD, and OpenBSD, share a long history. At the source-code level they diverged from a common ancestor, and although each has evolved in its own direction, they still recognise one another. A device driver that is carefully written in the FreeBSD style can often be ported to NetBSD or OpenBSD with meaningful but bounded effort, and a driver that is casually written cannot.

This section is deliberately short. This is a FreeBSD book, and turning Chapter 29 into a cross-BSD porting manual would crowd out material that belongs in later chapters. What we will do here is enough to let you recognise which parts of a FreeBSD driver travel well and which parts need translation. You will not learn how to port; you will learn what to design against, so that porting becomes a possibility rather than an impossibility.

29.10.1 Where the BSDs Agree

The areas where FreeBSD, NetBSD, and OpenBSD agree most strongly are the ones your driver leans on most heavily:

- **C language and toolchain.** All three use the same C standards and similar compilers (clang on FreeBSD and OpenBSD; gcc on NetBSD with clang available).
- **The general shape of kernel modules.** All three support loadable kernel modules with a similar life cycle: load, init, attach, detach, unload.

- **The general shape of device drivers.** Probe, attach, detach. Per-instance state. Resource allocation. Interrupt handlers. Newbus-style device trees (NetBSD has `config`; FreeBSD has `Newbus`; OpenBSD has `autoconf`).
- **Standard C library and POSIX.** Even in userland, the three systems share enough that most userland helpers port trivially.
- **Many device-specific protocols and wire formats.** An Ethernet frame is an Ethernet frame, and an NVMe command is an NVMe command, on any of the three.

A driver whose interesting logic is above the kernel API layer, parsing protocols, managing state machines, and coordinating transfers, is largely portable. The interesting logic typically doesn't know about `bus_dma` or `device_t`; it knows about the hardware's programming model.

29.10.2 Where the BSDs Differ

The areas where the three BSDs differ are concentrated in the kernel APIs:

- **Bus framework.** FreeBSD's `Newbus`, NetBSD's `autoconf(9)`, and OpenBSD's `autoconf(9)` (same name, different machinery) are not source compatible. The callbacks are similar in spirit but have different signatures.
- **DMA abstraction.** FreeBSD's `bus_dma(9)` has close cousins in NetBSD and OpenBSD called `bus_dma(9)` as well, but the types and some of the function signatures differ. The general model is shared; the exact API is not.
- **Memory allocation.** `malloc(9)` exists on all three with the same idea but slightly different function families and flags.
- **Synchronisation primitives.** Mutexes, condition variables, and epochs exist in all three, but the specific names and flags differ.
- **Network interfaces.** FreeBSD's `ifnet` and `if_t` opaque handle, NetBSD's `struct ifnet`, and OpenBSD's `struct ifnet` look similar but have diverged in the details.
- **Character devices.** FreeBSD's `cdev` framework, NetBSD's `cdevsw`, and OpenBSD's character-device machinery differ in how they register and dispatch.

This is not a complete list, but it captures the usual sources of porting pain.

29.10.3 The Cross-BSD-Friendly Style

If you want your driver to be portable across BSDs without committing to maintaining three versions, the cheapest long-term strategy is to write it in a style that isolates the FreeBSD-specific parts. The patterns are the ones you have already learned in this chapter:

- **Isolate hardware-dependent code.** A backend interface hides the bus APIs. Swapping FreeBSD's `bus_dma` for NetBSD's involves changing the backend implementation, not the core.
- **Wrap kernel primitives you use often.** If `malloc(M_TYPE, size, M_NOWAIT)` is replaced in NetBSD by a subtly different spelling, wrapping it in `portdrv_alloc(size)` means one place to change, not a hundred.
- **Keep core logic free of kernel subsystem dependencies.** The request queue, the state machine, and the protocol parsing do not need to include `sys/bus.h` or `net/if_var.h`.

- **Separate registration from logic.** The `DRIVER_MODULE` macro and its equivalents are a small, localised piece of code per driver. Putting it in a dedicated file per BSD is easier than sprinkling it through the driver.
- **Use standard types.** `uint32_t` and `size_t` exist on all three BSDs. `u_int32_t` and `int32` are older spellings that are more portable in the narrow sense but less idiomatic on FreeBSD.

If you have followed the file layout in Section 4, much of this is already in place. The core in `portdrv_core.c` is kernel-subsystem-agnostic. The backends in `portdrv_pci.c` and `portdrv_usb.c` are where the FreeBSD-specific code lives. A hypothetical NetBSD port would add `portdrv_pci_netbsd.c` and keep the core unchanged, up to small type mismatches.

29.10.4 Compatibility Wrappers

For the cases where you really do need the same source to compile on multiple BSDs, the standard technique is a compatibility header. Something like:

```
/* portdrv_os.h - OS-specific wrappers */
#ifdef __FreeBSD__
#include <sys/malloc.h>
#define PD_MALLOC(sz, flags) malloc((sz), M_PORTDRV, (flags))
#define PD_FREE(p)          free((p), M_PORTDRV)
#endif

#ifdef __NetBSD__
#include <sys/malloc.h>
#define PD_MALLOC(sz, flags) kmem_alloc((sz), (flags))
#define PD_FREE(p)          kmem_free((p), sz)
#endif

#ifdef __OpenBSD__
#include <sys/malloc.h>
#define PD_MALLOC(sz, flags) malloc((sz), M_DEVBUF, (flags))
#define PD_FREE(p)          free((p), M_DEVBUF, sz)
#endif
```

The core uses `PD_MALLOC` and `PD_FREE` everywhere. One header changes per new OS; the core never changes. Apply the same technique to locks, to DMA, to printing, to any FreeBSD primitive that differs in the other BSDs. The result is a driver whose OS-specific surface is a single file and whose core is shared.

Keep in mind the tradeoff. Writing the driver in this style costs you a little upfront: you cannot simply call `malloc` the FreeBSD way; you have to call your wrapper. If you do not actually need cross-BSD portability, the cost buys you nothing. If you do need it, the cost is far less than the alternative of maintaining three drivers.

29.10.5 Listing the FreeBSD-Specific APIs You Use

A useful exercise before you even consider cross-BSD work is to catalogue the FreeBSD-specific APIs your driver relies on. Go through the source and list every function, macro, and structure name that is unique to FreeBSD. The list typically includes:

- `bus_alloc_resource_any`, `bus_release_resource`
- `bus_setup_intr`, `bus_teardown_intr`
- `bus_read_1`, `bus_read_2`, `bus_read_4`, and the write variants
- `bus_dma_tag_create`, `bus_dmamap_load`, `bus_dmamap_sync`, `bus_dmamap_unload`
- `callout_init_mtx`, `callout_reset`, `callout_stop`, `callout_drain`
- `malloc`, `free`, `MALLOC_DEFINE`
- `mtx_init`, `mtx_lock`, `mtx_unlock`, `mtx_destroy`
- `sx_init`, `sx_slock`, `sx_xlock`
- `device_printf`, `device_get_softc`, `device_set_desc`
- `DRIVER_MODULE`, `MODULE_VERSION`, `MODULE_DEPEND`
- `DEVMETHOD`, `DEVMETHOD_END`
- `SYSCTL_ADD_NODE`, `SYSCTL_ADD_UINT`, etc.
- `if_alloc`, `if_free`, `ether_ifattach`, `if_attach`
- `cdevsw`, `make_dev`, `destroy_dev`

Each entry in this list is a candidate for a wrapper. Not every one needs to be wrapped, and a driver that wraps everything is overengineered. But the list tells you where the porting work would happen if it ever needed to, and it is a useful artifact for maintenance even without any cross-BSD plans.

29.10.6 When to Commit to Cross-BSD Support

Cross-BSD support is not free. It imposes an ongoing cost on development, because every feature must be tested on every supported BSD. It imposes a cost on readability, because wrappers replace direct API calls. It imposes a cost on performance, though usually a small one, because the wrappers add indirection.

That cost is worth paying when there is a clear reason. Examples:

- **You have users on multiple BSDs.** A device vendor that wants their product supported on both FreeBSD and NetBSD has a business reason to absorb the complexity.
- **The driver is community-developed across BSDs.** Open-source projects sometimes have contributors on all three BSDs, and unifying the code base reduces duplicate work.
- **The driver is funded by a project with cross-BSD requirements.** Some academic projects, research infrastructures, or embedded deployments have mixed-OS requirements built into their plans.

If none of these apply, do not preemptively engineer for cross-BSD support. Follow the patterns in this chapter, because they are good FreeBSD practice in their own right, and stay within FreeBSD. If the need for cross-BSD support appears later, the patterns already in place will make the eventual port cheaper than it would have been otherwise.

29.10.7 A Concrete Wrapping Example

To make the cross-BSD wrapping approach feel concrete, consider a driver that uses a callout timer. On FreeBSD, the API is the callout family. The code might look like:

```
callout_init_mtx(&sc->sc_timer, &sc->sc_mtx, 0);
callout_reset(&sc->sc_timer, hz / 10, portdrv_timer_cb, sc);
/* ... later ... */
callout_stop(&sc->sc_timer);
callout_drain(&sc->sc_timer);
```

On NetBSD, the analogous API is the callout family too, but the signatures differ slightly. On OpenBSD, timeout is the equivalent name. A cross-BSD driver wraps the usage:

```
/* portdrv_os.h */
#ifdef __FreeBSD__
typedef struct callout portdrv_timer_t;
#define PD_TIMER_INIT(t, m)    callout_init_mtx((t), (m), 0)
#define PD_TIMER_ARM(t, ticks, cb, arg) \
    callout_reset((t), (ticks), (cb), (arg))
#define PD_TIMER_STOP(t)      callout_stop(t)
#define PD_TIMER_DRAIN(t)     callout_drain(t)
#endif

#ifdef __NetBSD__
typedef struct callout portdrv_timer_t;
/* ... NetBSD-specific definitions ... */
#endif

#ifdef __OpenBSD__
typedef struct timeout portdrv_timer_t;
#define PD_TIMER_INIT(t, m)    timeout_set((t), (cb), (arg))
#define PD_TIMER_ARM(t, ticks, cb, arg) \
    timeout_add((t), (ticks))
/* ... */
#endif
```

The core code uses the PD_TIMER_* macros. The per-OS branches in the header translate between the abstract name and each OS's concrete API. This is the wrapper approach in miniature: one abstraction, many implementations.

Notice that the wrapper does more than rename calls. It also papers over genuine differences in how the APIs work. FreeBSD's callout_init_mtx binds the callout to a mutex so the callback runs with the mutex held. OpenBSD's timeout_set does not have this binding; the callback must acquire the mutex itself. The wrapper macros absorb this difference in the core's favour, by presenting the FreeBSD semantics as the canonical shape.

On OpenBSD, the implementation of `PD_TIMER_INIT` can register a shim callback that acquires the mutex before calling the user's callback.

This shows both the power and the cost of wrapping. The power is that the core code does not need to know about the difference. The cost is that the wrapper layer has non-trivial logic of its own, which must be maintained and tested. Wrapping is not free, and the wrapper layer grows as the set of wrapped APIs grows.

29.10.8 What “Porting” Actually Involves

A realistic cross-BSD port is rarely a single commit. It is a sequence of small steps, each of which makes the driver a little more portable before the final move to the target OS:

1. **Audit.** List every FreeBSD-specific API the driver uses. The list becomes the work plan.
2. **Wrap.** Introduce wrappers, one API at a time. For each wrapper, replace every call in the driver with the wrapper. Commit after each API.
3. **Test on FreeBSD.** Confirm the wrappers do not change behaviour on FreeBSD. This step matters: a bug introduced during wrapping must be caught before the port, not during it.
4. **Build on the target.** Add the target-specific header file. Implement the wrappers for the target OS. Build and fix compile errors until the driver links.
5. **Load on the target.** Load the module. Fix any missing symbols or dependencies.
6. **Test on the target.** Exercise the driver's features. Fix the bugs that appear.
7. **Iterate.** Each bug fix should flow back into the FreeBSD driver if it exposes a real issue, not a platform-specific quirk.

This is far from a one-day task. For a medium-sized driver, it is often a week or two of focused work. The cost pays off when the port is successful and the shared codebase becomes easier to maintain than two separate drivers.

29.10.9 When Cross-BSD Means Cross-Platform Generally

Some drivers benefit from running not just on other BSDs but on other operating systems entirely. Linux is the obvious target, but also macOS, Windows, and even user-space reimplementations of kernel services. A driver written with a clean backend interface and a core that speaks only in its own vocabulary is usable as a building block in any of these environments; only the backend and the wrapper layer change.

This is the deep reason the patterns in this chapter matter. They are not just about FreeBSD; they are about how to structure any driver so that the OS is a detail. The specific tools (`bus_dma`, `bus_space`, `__FreeBSD_version`) are FreeBSD's, but the patterns they embody are universal.

29.10.10 NetBSD and OpenBSD Pointers for the Curious

If you become curious about NetBSD or OpenBSD, a few pointers will save you time.

NetBSD's driver documentation lives in its tree, and the equivalent of `/usr/src/sys` is the same on NetBSD; you can compare the two trees directly. NetBSD's `bus_dma(9)` manual page explains its API with great care. The `config(8)` infrastructure is NetBSD's answer to Newbus and has its own manual.

OpenBSD's driver style is more conservative than FreeBSD's. OpenBSD values simplicity over flexibility, and its drivers tend to be smaller and more direct than FreeBSD's equivalents. If you read OpenBSD driver source, you will see fewer layers of abstraction and fewer build options. This is a design choice; it is neither inherently better nor worse than FreeBSD's style.

Both projects have excellent mailing lists and IRC channels. If you port a driver, asking for review from the community saves time. Both communities are generally welcoming to FreeBSD developers who approach them respectfully.

29.10.11 Wrapping Up Section 7

Cross-BSD portability is a deep topic, but its essentials are simple: isolate kernel-subsystem dependencies behind wrappers, keep the core logic OS-agnostic, and list the FreeBSD-specific APIs you use so you know where the borders are. If you ever need to port to NetBSD or OpenBSD, you will have only the border to translate; the core goes across unchanged. The wrapping itself is non-trivial work, but the wrapping layer grows slowly and is proportional to the breadth of APIs the driver uses, not to the driver's overall size. For most FreeBSD-only drivers, cross-BSD preparation is not justified; the patterns that support it are worth knowing because they make the driver better even when it stays FreeBSD-only.

In Section 8, we step back and address a different question: how do you package, document, and version a portable driver so that others can build, test, and consume it safely? Good code is not enough; good practice around the code matters too.

29.11 Section 8: Refactoring and Versioning for Portability

A portable driver is an artefact, but portability is also a practice. The code is one part; the way the code is organised, documented, and versioned is the rest. This section addresses that rest. It assumes you have refactored the driver into the shape laid out in Sections 2 through 7, and asks: what more should you do so that your work survives contact with other people and other systems?

The answers involve documentation, versioning, build validation, and a small amount of housekeeping. None of it is glamorous. All of it pays off the first time someone new tries to use your driver.

29.11.1 A Short README.portability.md

Every portable driver benefits from a short document that states its supported platforms, supported backends, build options, and known limitations. In the `portdrv` reference driver, that document is `README.portability.md`. A reader who picks up the driver for the first time should be able to read it in a few minutes and know whether their environment is supported.

A good portability README has four sections.

Supported platforms. List the FreeBSD versions and CPU architectures the driver is tested on, and the ones it is known or expected to work on even if not tested. For example:

Tested on:

- FreeBSD 14.3-RELEASE, amd64
- FreeBSD 14.3-RELEASE, arm64 (QEMU guest)

Expected to work but not regularly tested:

- FreeBSD 14.2-RELEASE and later, amd64
- FreeBSD 14.3-RELEASE, riscv64

Not supported:

- FreeBSD 13.x and earlier (API changes required)
- NetBSD / OpenBSD (see Section 7; see compatibility file portdrv_os.h)

Supported backends. List each backend, what hardware or software environment it targets, and any constraints. For example:

pci: PCI variant, requires a device with vendor 0x1234, device 0x5678.

usb: USB variant, requires a USB 2.0 or later host controller.

sim: Pure software simulation. No hardware required. Useful for tests.

Build options. List every -D flag that the Makefile recognises and what each one does. Mention which combinations are expected to work:

PORTDRV_WITH_PCI=yes	Enable the PCI backend.
PORTDRV_WITH_USB=yes	Enable the USB backend.
PORTDRV_WITH_SIM=yes	Enable the simulation backend.
PORTDRV_DEBUG=yes	Enable verbose debug logging.

If no backend flag is set, PORTDRV_WITH_SIM is enabled by default so that a plain "make" produces a loadable module.

Known limitations. State honestly what the driver does not support. A short, honest list is more useful than a long or evasive one.

The simulation backend does not attempt to emulate interrupt latency or DMA bandwidth limits. It completes transfers synchronously. Do not use it as a substitute for performance testing against real hardware.

A portability README lives alongside the driver's source. Update it whenever the supported set changes. If you pick up a driver that lacks one, writing one is the most valuable review activity you can perform.

29.11.2 Versioning the Driver

Kernel modules carry a version via `MODULE_VERSION`. A portable driver should version three things: the driver as a whole, each backend, and the backend interface.

The driver version is a single integer passed to `MODULE_VERSION`:

```
MODULE_VERSION(portdrv, 2);
```

Bump it whenever you change something a dependent consumer can observe, for instance when you add a new ioctl, change the semantics of an existing one, or rename a sysctl.

Each backend has its own module registration. Version each one independently:

```
MODULE_VERSION(portdrv_pci, 1);  
MODULE_DEPEND(portdrv_pci, portdrv_core, 1, 2, 2);
```

The `MODULE_DEPEND` call expresses that `portdrv_pci` depends on `portdrv_core` with a version in the range [1, 2] where 2 is the preferred value. The kernel refuses to load the backend against a core it does not understand, which prevents a mismatch from producing mysterious crashes.

The backend interface itself can be versioned in the struct. A version field at the top of struct `portdrv_backend` lets the core check that each backend was compiled against the right shape:

```
struct portdrv_backend {  
    uint32_t    version;  
    const char  *name;  
    int         (*attach)(struct portdrv_softc *sc);  
    /* ... */  
};  
  
#define PORTDRV_BACKEND_VERSION 2
```

When the core's `portdrv_core_attach` sees `sc->sc_be->version != PORTDRV_BACKEND_VERSION`, it refuses to attach and logs a clear message. This catches the case where someone updates the core but forgets to rebuild a backend, without requiring the kernel to crash to make the point.

Use these mechanisms lightly. Each one imposes a cost at maintenance time, because a version bump must be coordinated. The reward is that when something does go wrong, the error is immediate and clear instead of vague and delayed.

29.11.3 Document Supported Build Configurations

Alongside the `README.portability.md`, maintain a short **build matrix** that records which combinations of backends and options are expected to compile and which have been recently tested. This is a project-management artefact, not a runtime one, but it is invaluable when a new maintainer takes over or when a reviewer wants to know whether a specific configuration will still work after a change.

A practical build matrix looks like this:

Config	Compiles	Tested	Notes
-----	-----	-----	-----
(default: SIM only)	yes	yes	Baseline CI.
PCI only	yes	yes	
USB only	yes	yes	
PCI + USB	yes	yes	
PCI + USB + SIM	yes	yes	Full build.
PCI + DEBUG	yes	yes	
PCI + USB + DEBUG	yes	no	Should be OK.
None of PCI/USB/SIM (no backend)	yes	yes	SIM auto-enabled

Regenerate the matrix before a release. It is a small chore that saves a large amount of time later. When users file bugs against specific configurations, you can tell at a glance whether the configuration was claimed to work, and if so, whether it was in fact tested at the last release.

29.11.4 Validating the Build Before a Release

A good release discipline for a portable driver is to validate every documented configuration automatically before each release. Automation makes this cheap. A short shell script that loops over the build matrix, invokes `make clean` and `make`, and reports success or failure, is typically enough:

```
#!/bin/sh
# Validate portdrv builds in every supported configuration.

set -e

configs="
    PORTDRV_WITH_SIM=yes
    PORTDRV_WITH_PCI=yes
    PORTDRV_WITH_USB=yes
    PORTDRV_WITH_PCI=yes PORTDRV_WITH_USB=yes
    PORTDRV_WITH_PCI=yes PORTDRV_WITH_USB=yes PORTDRV_WITH_SIM=yes
    PORTDRV_WITH_PCI=yes PORTDRV_DEBUG=yes
"

OLDIFS="$IFS"
echo "$configs" | while read cfg; do
    [ -z "$cfg" ] && continue
    echo "==> Building with: $cfg"
    make clean > /dev/null
    if env $cfg make > build.log 2>&1; then
        echo "    OK"
```

```
    else
        echo "      FAIL (see build.log)"
        tail -n 20 build.log
        exit 1
    fi
done
```

Run this before you tag a release. When it passes, you know that every configuration you document as supported actually builds. When it fails, you know the exact configuration that broke and can fix it before shipping. This is the cheapest and most valuable quality practice a portable driver can adopt.

29.11.5 When to Version Bump

A frequent question from new driver authors is: “When should I bump the module version number?” The short answer is that you bump it when anything observable to consumers changes. The longer answer distinguishes three kinds of change:

Backwards-compatible additions. Adding a new ioctl command, a new sysctl, a new module option, without changing the behaviour of existing commands. Bump the minor version. Existing consumers continue to work.

Backwards-incompatible changes. Renaming an ioctl, changing the meaning of an existing sysctl, or breaking a previously documented behaviour. Bump the major version. Update the `MODULE_DEPEND` in dependent modules. Document the break in a release note.

Internal refactoring that users cannot observe. Moving functions between files, renaming private variables, reformatting. Do not bump the version. Internal changes are internal.

Keep in mind that `MODULE_VERSION` is an integer, and consumers decide how to interpret it. For a driver with a small user base, one version number per observable change is fine. For a driver with a larger community and more dependents, establishing a convention such as “major * 100 + minor” lets you encode more information in a single integer. Either way, document what your version numbers mean so that future maintainers and consumers can interpret them.

29.11.6 Refactoring for Portability as an Ongoing Discipline

A portable driver is not a finished artefact. Even after the initial refactor, the driver continues to evolve: new features, new hardware variants, new kernel APIs, new bug fixes. Keeping the driver portable is an ongoing discipline, not a one-time achievement.

Three habits help.

Review new patches for portability hazards. When a contributor adds a new register access, does it go through the accessor? When they add a new backend-specific function, does it fit the interface, or does it force an `if (sc->sc_be == pci)` in the core? When they add a new type, is it fixed-width? These questions take a few seconds per patch and catch most regressions.

Re-run the build matrix regularly. A monthly or weekly automated build of every supported configuration catches silent regressions the moment they appear. If CI resources are limited, run the matrix on pull requests and nightly on the main branch.

Re-test on a non-amd64 platform periodically. Even a single arm64 guest booted under QEMU once a quarter catches a surprising number of architectural bugs. The test does not have to be exhaustive; even just loading the driver and running the simulation backend is enough to reveal many endian and alignment issues.

29.11.7 The Final Shape of a Portable Driver

After all of this, a portable driver has a recognisable shape. The core is small and does not know about specific buses. Each backend is a thin file that implements a clean interface. The headers are short and each has a clear purpose. The Makefile uses feature flags to select backends and options. The driver uses `uint32_t` and its family for sizes, `bus_read_*` and `bus_write_*` for register access, `bus_dma_*` for DMA, and endian helpers whenever multi-byte values cross the hardware boundary. The driver documents its supported platforms and build options. The driver has a build matrix and a validation script.

A reader opening the source for the first time can find what they need in minutes. A new contributor can add a backend in an afternoon. A reviewer can audit a patch without reading the entire driver. That is the shape.

29.11.8 ABI vs. API Stability

A subtle distinction is often missed when drivers are versioned: the difference between the **application programming interface** (API) and the **application binary interface** (ABI). The API is what a programmer sees: function names, parameter types, expected behaviour. The ABI is what a linker sees: symbol names, structure layouts, calling conventions, alignment.

A change that leaves the API unchanged can still break the ABI. Adding a field to the middle of a structure is a classic example: source files that use the structure compile unchanged, but binary modules compiled against the old layout will read the wrong fields at runtime because the offsets have shifted.

For kernel modules, ABI stability matters because modules are loaded into a running kernel, and the kernel and the module must agree on the layout of the structures they share. This is the reason `MODULE_DEPEND` includes version numbers: an older module refusing to load against a newer kernel, or vice versa, is safer than silently reading wrong fields.

Two rules help preserve ABI stability over time.

Rule one: add fields only at the end of structures. The existing fields retain their offsets, and older consumers read them correctly. Newer consumers, which know about the added field, can access it at the end.

Rule two: never change the type or size of an existing field. If you need to widen a field, add a new field and leave the old one for compatibility. This is awkward, but it is the price of ABI stability.

Drivers with a small audience can ignore ABI stability most of the time, because each release rebuilds everything. Drivers whose consumers cannot rebuild everything, such as out-of-tree modules that users compile against their own kernel, must be more careful.

29.11.9 Upgrade and Downgrade Paths

A mature driver must be usable through upgrade and downgrade. Upgrades are straightforward: install the new module, unload the old one, load the new one. Downgrades are sometimes harder, because the new module may have saved state that the old module cannot read.

For most drivers, state is not persistent, so downgrade is easy. For drivers that store state in a character device's data, in filesystem metadata, or in a database the device itself maintains, downgrade paths deserve thought. Your state format should include a version field; older versions should refuse to read newer formats and produce a clear error; newer versions should read older formats when possible.

Release notes are the simplest rollout mechanism. A release note that says “this version changes the state format; downgrade is not possible without loss of data” is worth more than a sophisticated state-migration system that nobody uses. Communicate the upgrade impact clearly and let your users decide.

29.11.10 Telemetry and Observability

A portable driver that is deployed widely benefits from lightweight telemetry. Not intrusive logging that floods the kernel log, but unobtrusive counters that an operator can query to see whether the driver is healthy. The usual FreeBSD mechanism is a sysctl tree:

```
SYSCTL_ADD_UQUAD(ctx, tree, OID_AUTO, "transfers_ok",  
    CTLFLAG_RD, &sc->sc_stats.transfers_ok,  
    "Successful transfers");  
SYSCTL_ADD_UQUAD(ctx, tree, OID_AUTO, "transfers_err",  
    CTLFLAG_RD, &sc->sc_stats.transfers_err,  
    "Failed transfers");  
SYSCTL_ADD_UQUAD(ctx, tree, OID_AUTO, "queue_depth",  
    CTLFLAG_RD, &sc->sc_stats.queue_depth,  
    "Current queue depth");
```

Operators can query these with `sysctl dev.portdrv.0`, monitoring systems can scrape them at intervals, and bug reports can include their values without requiring the reporter to do anything special. The cost to the driver is one field per counter and one sysctl registration per field. For a mature driver, the telemetry tree grows naturally as questions arise.

Add the telemetry first, debug second. When a bug report arrives and the sysctl values show what happened, the debugging is much faster than when the values are missing.

29.11.11 A Post-Refactor Checklist

After a portability refactor, it is worth running through a short checklist before declaring the work complete. The checklist is deliberately terse; each item stands for a subtlety you should confirm.

1. Does the core file include any bus-specific header? If yes, move the include or the using code.

2. Does every register access go through an accessor or a backend method? Grep for `bus_read_`, `bus_write_`, `bus_space_read_`, and `bus_space_write_` in the core file; each occurrence is a candidate for attention.
3. Does the driver use fixed-width types for every value whose size matters? Look for `int`, `long`, `unsigned`, and `size_t` in register access contexts; replace with `uint32_t`, `uint64_t`, and so on as appropriate.
4. Does every multi-byte value that crosses the hardware boundary pass through an endian helper? Look for `raw *(uint32_t *)` casts on DMA buffers and register images.
5. Does the build succeed in every documented configuration? Run the build matrix script.
6. Does the module load cleanly, with the expected banner? Load each configuration in turn and check `dmesg`.
7. Does the driver detach cleanly? Unload it and confirm no panics and no leaked resources.
8. Is the documentation up to date? Re-read `README.portability.md` and ensure it matches reality.
9. Is the lab logbook current? Note what was done, what surprised you, and what you expect to change in the future.

Running the checklist takes about an hour. Finding a problem during the checklist is cheaper than finding it in production.

29.11.12 Rollouts and Rollbacks

When a refactored driver is ready to ship, consider the rollout strategy. For a small user base, a single release is enough. For a larger one, a staged rollout reduces risk:

1. **Internal staging.** Deploy to internal test systems first. Run the driver for at least a day before exposing it to external users.
2. **Early adopters.** Release to a small group of external users who have opted in. Collect feedback for a week.
3. **General release.** Ship to all users. Announce the release with a changelog.
4. **Rollback plan.** Document how to revert to the prior version. A driver that can be unloaded and replaced by the previous version is safer than one that is entangled with the rest of the system.

The granularity of the rollout depends on your project. A hobby project does not need staged rollouts. A commercial driver serving hundreds of users does. Think about the balance and choose deliberately.

29.11.13 Wrapping Up Section 8

You now know not only how to write portable code but how to package it so that others can consume it safely. Versioning, documentation, a build matrix, and a validation script are the small, unglamorous practices that separate a hobbyist driver from a production-ready one. They cost little to adopt and pay off the first time someone else touches the code, including future you. ABI stability, upgrade paths, telemetry, a post-refactor checklist, and a rollout strategy are the companion practices that mature a driver over the long term.

The chapter's remaining material gives you hands-on labs, challenge exercises, and a troubleshooting guide. The labs let you apply every technique in this chapter to the reference driver. The challenges push further, so you can practice the patterns on variations of the same problem. The troubleshooting guide catalogues the failures you are likeliest to meet along the way.

29.12 Hands-On Labs

The labs in this chapter walk you through transforming a small single-file driver into a portable, multi-backend, multi-file driver. The reference implementation, `portdrv`, lives in `examples/part-07/ch29-portability/`. Each lab is self-contained and leaves you with a working module that loads cleanly. You can do them all, or stop at any point; each one is useful on its own.

Before you begin, change into the examples directory and inspect its current state:

```
cd /path/to/FDD-book/examples/part-07/ch29-portability
ls
```

You should see a `lab01-monolithic/` directory that contains the starting-point driver, along with `lab02` through `lab07` directories that contain the successive refactoring steps. Each lab directory has its own `Makefile` and `README`. Work in `lab01-monolithic` first; when a lab is complete, move on to the next directory, which contains the state of the driver after that step.

A quick workflow tip: make a local working copy of the directory for each lab so you can compare your version against the reference afterwards.

```
cp -r lab01-monolithic mywork
cd mywork
```

Then follow the instructions.

29.12.1 Setting Up the Lab Environment

Before you start any of the labs, set up the working environment. A few minutes of preparation saves hours of friction later.

Create a working directory outside the repository, so you do not accidentally commit lab output:

```
mkdir -p ~/fdd-labs/ch29
cd ~/fdd-labs/ch29
cp -r /path/to/FDD-book/examples/part-07/ch29-portability/* .
```

Confirm that `make` works with the default lab files before making any changes:

```
cd lab01-monolithic
make clean && make
ls *.ko
```

If the build fails, stop and debug now. A lab is not useful if the starting state is broken.

Next, verify you can load and unload the reference driver:

```
sudo kldload ./portdrv.ko
dmesg | tail
sudo kldunload portdrv
```

If the load fails, check dmesg for an error message. The usual suspects are a missing module dependency or a conflict with another driver. Fix these before beginning the labs.

Finally, start a simple logbook. A plain text file or a Markdown file in your working directory is enough:

```
=== Chapter 29 Lab Logbook ===
Date: 2026-04-19
System: FreeBSD 14.3-RELEASE, amd64
```

Every lab will add an entry. The logbook is not for public consumption; it is a record for future you.

29.12.2 Lab 1: Audit the Monolithic Driver

The starting-point driver is a single file that compiles and loads but mixes every concern in one place. The goal of this lab is to **notice** the portability problems without fixing them yet. Training your eye is the first step.

```
cd lab01-monolithic
less portdrv.c
```

As you read, make notes in your lab logbook for each of the following:

1. Which lines perform register access? Mark every `bus_space_read_*`, `bus_space_write_*`, `bus_read_*`, `bus_write_*`, and any raw pointer dereferences to device memory.
2. Which lines allocate or manipulate DMA buffers? Mark every `bus_dma_*` call.
3. Which lines include bus-specific headers such as `dev/pci/pcivar.h`?
4. Which lines hardcode specific vendor or device IDs?
5. Which lines use types that might be wrong on a 32-bit platform (`int`, `long`, `size_t`)?
6. Which lines use `htons`, `htonl`, or a related endian helper?
7. Which lines are guarded with `#ifdef`?

Now build and load the driver:

```
make clean
make
sudo kldload ./portdrv.ko
dmesg | tail -5
sudo kldunload portdrv
```

Confirm that the module loads cleanly. If it does not, fix the build before moving on.

Reflection. At the end of the lab, your logbook should have a short paragraph describing the driver's current portability state. Something like: "The register accesses go through `bus_read_*`, so architectural portability is acceptable, but they are spread across the file without accessors. PCI-specific code is interleaved with core logic. No endian helpers are used anywhere. There is no simulation backend. The driver compiles on amd64 only; I have not tried arm64."

This paragraph is the baseline against which every later lab measures progress.

29.12.3 Lab 2: Introduce Register Accessors

The first structural change is to centralise register access in accessors, as described in Section 2. You are not yet splitting the file; you are only adding a layer of indirection.

Add two static inline functions at the top of `portdrv.c`:

```
static inline uint32_t
portdrv_read_reg(struct portdrv_softc *sc, bus_size_t off)
{
    return (bus_read_4(sc->sc_bar, off));
}

static inline void
portdrv_write_reg(struct portdrv_softc *sc, bus_size_t off, uint32_t val)
{
    bus_write_4(sc->sc_bar, off, val);
}
```

Then, throughout the file, replace every `bus_read_4(sc->sc_bar, X)` with `portdrv_read_reg(sc, X)`, and every `bus_write_4(sc->sc_bar, X, V)` with `portdrv_write_reg(sc, X, V)`. Do this carefully. After every small batch of replacements, rebuild:

```
make clean && make
```

And reload:

```
sudo kldunload portdrv 2>/dev/null
sudo kldload ./portdrv.ko
dmesg | tail -5
```

Confirm that the driver still works. The accessor layer is functionally invisible; the binary behaviour should be unchanged.

Checkpoint. Count the number of `bus_read_*` and `bus_write_*` occurrences in the file after the change. They should all be inside the accessors. If any are outside, find and replace them. That sweep is the whole point of the lab.

29.12.4 Lab 3: Extract the Backend Interface

Now that accessors exist, add a backend interface. Create a new file `portdrv_backend.h` with the struct definition from Section 3:

```
#ifndef _PORTDRV_BACKEND_H_
#define _PORTDRV_BACKEND_H_

struct portdrv_softc;

struct portdrv_backend {
    const char *name;
    int  (*attach)(struct portdrv_softc *sc);
    void (*detach)(struct portdrv_softc *sc);
    uint32_t (*read_reg)(struct portdrv_softc *sc, bus_size_t off);
    void      (*write_reg)(struct portdrv_softc *sc, bus_size_t off,
                          uint32_t val);
};

extern const struct portdrv_backend portdrv_pci_backend;

#endif
```

In `portdrv.c`, add the `softc` field:

```
struct portdrv_softc {
    device_t sc_dev;
    struct resource *sc_bar;
    int sc_bar_rid;
    const struct portdrv_backend *sc_be;    /* new */
    /* ... */
};
```

Rewrite the existing accessor functions to dispatch through the backend:

```
static inline uint32_t
portdrv_read_reg(struct portdrv_softc *sc, bus_size_t off)
{
    return (sc->sc_be->read_reg(sc, off));
}

static inline void
portdrv_write_reg(struct portdrv_softc *sc, bus_size_t off, uint32_t val)
{
    return (sc->sc_be->write_reg(sc, off, val));
}
```

```
    sc->sc_be->write_reg(sc, off, val);  
}
```

Define a first instance of the interface:

```
static uint32_t  
portdrv_pci_read_reg_impl(struct portdrv_softc *sc, bus_size_t off)  
{  
    return (bus_read_4(sc->sc_bar, off));  
}  
  
static void  
portdrv_pci_write_reg_impl(struct portdrv_softc *sc, bus_size_t off, uint32_t val)  
{  
    bus_write_4(sc->sc_bar, off, val);  
}  
  
const struct portdrv_backend portdrv_pci_backend = {  
    .name      = "pci",  
    .read_reg  = portdrv_pci_read_reg_impl,  
    .write_reg = portdrv_pci_write_reg_impl,  
    /* attach and detach stay NULL for now. */  
};
```

In the existing `portdrv_attach` function, install the backend:

```
sc->sc_be = &portdrv_pci_backend;
```

Rebuild, load, and confirm the driver still works.

Checkpoint. The core logic now never touches `bus_read_*` or `bus_write_*` directly. Every register access goes through the backend. Verify this by grepping for `bus_read_` and `bus_write_` in the file; they should all be inside the `_impl` functions.

29.12.5 Lab 4: Split the Core and the Backend into Separate Files

Now that the interface is in place, split the file.

Create three files:

- `portdrv_core.c`: contains the `softc` definition, the register accessors, the attach/detach logic that is not PCI-specific, and the module registration for the core.
- `portdrv_pci.c`: contains the PCI probe, the PCI-specific attach and detach, the `_impl` functions, and the backend struct.
- `portdrv_backend.h`: contains the backend interface.

The details of the split are in the reference implementation under `lab04-split`. Study it after you have done your own split, and compare.

Update the `Makefile` to list both source files:

```
SRCS= portdrv_core.c portdrv_pci.c
```

Rebuild and load. The driver should load identically to before, but the source is now organised.

Checkpoint. Open `portdrv_core.c` and search for `#include <dev/pci/pcivar.h>`. It should not appear. The core is free of PCI-specific includes. That is the milestone.

29.12.6 Lab 5: Add a Simulation Backend

Add a second backend that does not require any hardware. Create `portdrv_sim.c` with its own implementation of the backend interface:

```
/* portdrv_sim.c - simulation backend for portdrv */

#include <sys/param.h>
#include <sys/system.h>
#include <sys/kernel.h>
#include <sys/module.h>
#include <sys/bus.h>
#include <sys/malloc.h>

#include "portdrv.h"
#include "portdrv_common.h"
#include "portdrv_backend.h"

struct portdrv_sim_priv {
    uint32_t regs[256]; /* simulated register file */
};

static uint32_t
portdrv_sim_read_reg(struct portdrv_softc *sc, bus_size_t off)
{
    struct portdrv_sim_priv *psp = sc->sc_backend_priv;
    if (off / 4 >= nitems(psp->regs))
        return (0);
    return (psp->regs[off / 4]);
}

static void
portdrv_sim_write_reg(struct portdrv_softc *sc, bus_size_t off, uint32_t val)
```

```

{
    struct portdrv_sim_priv *psp = sc->sc_backend_priv;
    if (off / 4 < nitems(psp->regs))
        psp->regs[off / 4] = val;
}

static int
portdrv_sim_attach_be(struct portdrv_softc *sc)
{
    return (0);
}

static void
portdrv_sim_detach_be(struct portdrv_softc *sc)
{
}

const struct portdrv_backend portdrv_sim_backend = {
    .name      = "sim",
    .attach    = portdrv_sim_attach_be,
    .detach    = portdrv_sim_detach_be,
    .read_reg  = portdrv_sim_read_reg,
    .write_reg = portdrv_sim_write_reg,
};

```

Add a module-load hook that creates a simulated instance when the driver is loaded without any real hardware:

```

static int
portdrv_sim_modevent(module_t mod, int type, void *arg)
{
    static struct portdrv_softc *sim_sc;
    static struct portdrv_sim_priv *sim_priv;

    switch (type) {
    case MOD_LOAD:
        sim_sc = malloc(sizeof(*sim_sc), M_PORTDRV, M_WAITOK | M_ZERO);
        sim_priv = malloc(sizeof(*sim_priv), M_PORTDRV,
            M_WAITOK | M_ZERO);
        sim_sc->sc_be = &portdrv_sim_backend;
        sim_sc->sc_backend_priv = sim_priv;
        return (portdrv_core_attach(sim_sc));
    case MOD_UNLOAD:
        if (sim_sc != NULL) {
            portdrv_core_detach(sim_sc);

```



```
        free(sim_priv, M_PORTDRV);
        free(sim_sc, M_PORTDRV);
    }
    return (0);
}
return (0);
}

static moduledata_t portdrv_sim_mod = {
    "portdrv_sim",
    portdrv_sim_modevent,
    NULL
};

DECLARE_MODULE(portdrv_sim, portdrv_sim_mod, SI_SUB_DRIVERS, SI_ORDER_ANY);
MODULE_VERSION(portdrv_sim, 1);
MODULE_DEPEND(portdrv_sim, portdrv_core, 1, 1, 1);
```

Update the Makefile to include the new file:

```
SRCS= portdrv_core.c portdrv_pci.c portdrv_sim.c
```

Rebuild and load:

```
make clean && make
sudo kldload ./portdrv.ko
dmesg | tail -10
```

You should see both the PCI backend registering with the bus and a simulated instance created by the sim backend's module hook. Interact with the simulated instance through its character device (if any) and confirm that register writes and reads behave as stored integers.

Checkpoint. You now have a driver that compiles with two backends and runs both simultaneously. The PCI backend manages any real hardware the kernel discovers; the simulation backend gives you a software instance for testing.

29.12.7 Lab 6: Conditional Compilation of Backends

Make each backend conditional on a build-time flag. Edit the Makefile:

```
KMOD= portdrv
SRCS= portdrv_core.c
```

```
.if defined(PORTDRV_WITH_PCI) && ${PORTDRV_WITH_PCI} == "yes"
SRCS+= portdrv_pci.c
CFLAGS+= -DPORTDRV_WITH_PCI
.endif

.if defined(PORTDRV_WITH_SIM) && ${PORTDRV_WITH_SIM} == "yes"
SRCS+= portdrv_sim.c
CFLAGS+= -DPORTDRV_WITH_SIM
.endif

.if !defined(PORTDRV_WITH_PCI) && !defined(PORTDRV_WITH_SIM)
SRCS+= portdrv_sim.c
CFLAGS+= -DPORTDRV_WITH_SIM
.endif

SYSDIR?= /usr/src/sys
.include <bsd.kmod.mk>
```

Build with various combinations:

```
make clean && make PORTDRV_WITH_PCI=yes
make clean && make PORTDRV_WITH_SIM=yes
make clean && make PORTDRV_WITH_PCI=yes PORTDRV_WITH_SIM=yes
make clean && make
```

Confirm that each build succeeds and produces a module. Load each module in turn and watch the dmesg output. The message at load time should identify which backends are present.

Add a banner to the core's module load event:

```
static int
portdrv_core_modevent(module_t mod, int type, void *arg)
{
    switch (type) {
        case MOD_LOAD:
            printf("portdrv: version %d loaded, backends:"
#ifdef PORTDRV_WITH_PCI
                " pci"
#endif
#ifdef PORTDRV_WITH_SIM
                " sim"
#endif
                "\n", PORTDRV_VERSION);
            return (0);
    }
}
```

```
    /* ... */  
}  
return (0);  
}
```

Now `dmesg` at load time tells you exactly what this build supports.

Checkpoint. You can now select, at build time, which backends are present in the module. The source is not polluted with `#ifdef` inside function bodies; the selection happens at the Makefile level, and the banner hides its `#ifdef` chain in a controlled location.

29.12.8 Lab 7: Endian-Safe Register Access

Pick one register in the driver that represents a multi-byte value in a specific byte order. Most real devices document this in the datasheet; for the reference driver, assume that register `REG_DATA` at offset `0x08` is little-endian.

Modify the accessors to apply endian conversion explicitly. Do **not** bake the conversion into the general register accessor; the hardware registers themselves are in host order through `bus_read_4`. The conversion applies to the **interpretation** of the value.

Add a higher-level helper:

```
static uint32_t  
portdrv_read_data_le(struct portdrv_softc *sc)  
{  
    uint32_t raw = portdrv_read_reg(sc, REG_DATA);  
    return (le32toh(raw));  
}  
  
static void  
portdrv_write_data_le(struct portdrv_softc *sc, uint32_t val)  
{  
    portdrv_write_reg(sc, REG_DATA, htole32(val));  
}
```

Use these in the core whenever the code is manipulating a value that is stored in little-endian form in the register.

Checkpoint. On amd64 the endian helpers are no-ops, so behaviour does not change. On a big-endian platform, the helpers perform byte swaps that would have been missing before. You are now prepared for architectural portability even if you have not yet tested on a big-endian host.

29.12.9 Lab 8: Build Matrix Script

Write a short script that builds the driver in every supported configuration and reports success or failure. Save it as `validate-build.sh` in the chapter directory.

```
#!/bin/sh
set -e

configs="
PORTDRV_WITH_SIM=yes
PORTDRV_WITH_PCI=yes
PORTDRV_WITH_PCI=yes PORTDRV_WITH_SIM=yes
"

echo "$configs" | while read cfg; do
    [ -z "$cfg" ] && continue
    printf "==> %s ... " "$cfg"
    make clean > /dev/null 2>&1
    if env $cfg make > build.log 2>&1; then
        echo "OK"
    else
        echo "FAIL"
        tail -n 20 build.log
        exit 1
    fi
done
```

Run it:

```
chmod +x validate-build.sh
./validate-build.sh
```

If all three configurations build, you have a minimal build matrix in place. Every future change to the driver can be checked against the matrix with a single invocation.

Checkpoint. The driver builds in every configuration you have advertised. You have a machine-readable way to confirm that fact. This is the last structural piece of a portable driver.

29.12.10 Lab 9: Exercising the Simulation Backend from Userland

A refactor is only as good as the tests you can run against it. Once the simulation backend is in place, you can exercise the driver's core logic from a small userland program without needing any hardware. This lab walks through a lightweight test harness.

Create a userland program, `portdrv_test.c`, that opens `/dev/portdrv0` (or whatever character device the driver creates for the simulated instance), writes a known value, reads it back, and prints the result:

```
#include <sys/types.h>
#include <sys/stat.h>
```

```
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int
main(int argc, char *argv[])
{
    const char *dev = (argc > 1) ? argv[1] : "/dev/portdrv0";
    int fd;
    char buf[64];

    fd = open(dev, O_RDWR);
    if (fd < 0) {
        perror(dev);
        return (1);
    }

    memset(buf, 0, sizeof(buf));
    snprintf(buf, sizeof(buf), "hello from userland\n");
    if (write(fd, buf, strlen(buf)) < 0) {
        perror("write");
        close(fd);
        return (1);
    }

    memset(buf, 0, sizeof(buf));
    if (read(fd, buf, sizeof(buf) - 1) < 0) {
        perror("read");
        close(fd);
        return (1);
    }

    printf("Read back: %s", buf);
    close(fd);
    return (0);
}
```

Compile it with `cc -o portdrv_test portdrv_test.c`. Load the driver with the simulation backend enabled, then run the test program:

```
sudo kldload ./portdrv.ko
./portdrv_test
```

If the simulation backend stores and retrieves data correctly, the output should confirm a successful round trip. If the test fails, the simulation backend is the isolated place to debug: no hardware, no driver lifecycle surprises, just the core logic plus the simulated backend.

Checkpoint. You have a reproducible end-to-end test that exercises the driver's character-device path, its I/O handling, and its simulation backend without touching any real hardware. This test can run in CI, on a developer's laptop, or inside a bhyve guest with no PCI cards.

29.12.11 Lab 10: Add a Basic sysctl Tree

Introduce a sysctl tree rooted at `dev.portdrv.<unit>` that exposes a handful of runtime-visible values. The goal of this lab is not to wire every statistic into sysctl; it is to establish the structure so that later features can extend it cheaply.

In `portdrv_core.c`, add a pair of helpers:

```
static void
portdrv_sysctl_init(struct portdrv_softc *sc)
{
    struct sysctl_ctx_list *ctx;
    struct sysctl_oid *tree;

    ctx = device_get_sysctl_ctx(sc->sc_dev);
    tree = device_get_sysctl_tree(sc->sc_dev);

    SYSCTL_ADD_STRING(ctx, SYSCTL_CHILDREN(tree),
        OID_AUTO, "backend", CTLFLAG_RD,
        __DECONST(char *, sc->sc_be->name), 0,
        "Backend name");

    SYSCTL_ADD_INT(ctx, SYSCTL_CHILDREN(tree),
        OID_AUTO, "version", CTLFLAG_RD,
        NULL, PORTDRV_VERSION,
        "Driver version");

    SYSCTL_ADD_UQUAD(ctx, SYSCTL_CHILDREN(tree),
        OID_AUTO, "transfers_ok", CTLFLAG_RD,
        &sc->sc_stats.transfers_ok,
        "Successful transfers");
}
```

Call `portdrv_sysctl_init(sc)` at the end of `portdrv_core_attach`. After loading the driver, inspect the tree:

```
sysctl dev.portdrv.0
```

You should see:

```
dev.portdrv.0.%desc: portdrv (PCI backend)
dev.portdrv.0.%parent: pci0
dev.portdrv.0.backend: pci
dev.portdrv.0.version: 2
dev.portdrv.0.transfers_ok: 0
```

Exercise the driver and watch the `transfers_ok` counter increase. If it does not, the core is not updating it; that is a bug to fix.

Checkpoint. The driver's runtime state is now observable from userland without parsing `dmesg`. Monitoring systems and ad-hoc bug reports can include a snapshot of the `sysctl` tree, and you know the driver's health at a glance.

29.12.12 Lab 11: Runtime Backend Selection

For advanced use, let the driver's simulation backend be enabled or disabled at runtime via a `sysctl` or a module parameter. This lab introduces the idea of making backend selection flexible at module load time, useful for CI pipelines that want the same module to behave differently depending on the environment.

Add a tunable to the module:

```
static int portdrv_sim_instances = 0;
SYSCTL_INT(_debug_portdrv, OID_AUTO, sim_instances,
    CTLFLAG_RD, &portdrv_sim_instances, 0,
    "Number of simulation backend instances to create at load time");
TUNABLE_INT("debug.portdrv.sim_instances", &portdrv_sim_instances);
```

In the module's load hook, create the number of simulation instances requested by the tunable:

```
for (i = 0; i < portdrv_sim_instances; i++)
    portdrv_sim_create_instance(i);
```

At load time, the operator can set the tunable from `/boot/loader.conf`:

```
debug.portdrv.sim_instances="2"
```

Load the module and confirm two simulated instances appear. The PCI backend, if compiled in, still attaches to any real hardware it finds. The two kinds of instance coexist in the same module, with no conflict between them.

Checkpoint. The driver is controllable at load time via a tunable, which means the same binary can behave differently in different deployments without recompilation. This is the kind of runtime flexibility that a single-file driver would find awkward and that a well-structured multi-backend driver handles as a natural extension.

29.12.13 Lab 12: Write a Minimal CI Harness

Combine the build matrix script from Lab 8 with a basic load test. The goal is a single command that validates the driver in every supported configuration and confirms it loads cleanly in each. Automate this into a CI-like loop that you can run before committing changes.

```
#!/bin/sh
# ci-check.sh - validate portdrv in every supported configuration.
set -e

configs="
PORTDRV_WITH_SIM=yes
PORTDRV_WITH_PCI=yes
PORTDRV_WITH_PCI=yes PORTDRV_WITH_SIM=yes
"

echo "$configs" | while read cfg; do
    [ -z "$cfg" ] && continue
    printf "==> Build: %s ... " "$cfg"
    make clean > /dev/null 2>&1
    if env $cfg make > build.log 2>&1; then
        echo "OK"
    else
        echo "FAIL"
        tail -n 20 build.log
        exit 1
    fi
    printf "==> Load : %s ... " "$cfg"
    if sudo kldload ./portdrv.ko > load.log 2>&1; then
        sudo kldunload portdrv > /dev/null 2>&1
        echo "OK"
    else
        echo "FAIL"
        cat load.log
        exit 1
    fi
done
echo "All configurations passed."
```

Run it:

```
chmod +x ci-check.sh
./ci-check.sh
```

Checkpoint. A single script confirms that every configuration you document as supported actually builds and

loads. You can invoke this before every commit, or wire it into a CI system that runs it on every pull request. The cost is a minute or two per run; the reward is catching regressions within minutes of making them.

29.13 Challenge Exercises

The following challenges push the refactoring further and give you room to practice the chapter's patterns on variations of the same problem. Each challenge is independent and draws only on material from this chapter and the earlier ones. Take them at your own pace; they are designed to consolidate mastery rather than to introduce new foundations.

29.13.1 Challenge 1: Add a Third Backend

Add a third backend to the reference driver. The specifics depend on your interests:

- An MMIO backend that reaches the device through a Device Tree node on an embedded platform.
- A child of an I2C or SPI bus.
- A backend that uses bhyve's virtio framework as a guest.

The structural requirement is the same in each case: a new `portdrv_XXX.c` file, a new implementation of the backend interface, a new entry in the Makefile and a new build-time flag. After the work is done, confirm that the driver still builds in every previously supported configuration and additionally in the new one.

29.13.2 Challenge 2: Expose Version Metadata via sysctl

Create a sysctl subtree at `dev.portdrv.<unit>.version` that exposes:

- the driver version
- the backend name
- the backend version
- the flags that were set at build time

The sysctl tree should be populated in `portdrv_sysctl.c` (create it if it does not exist) and should read its values from constants defined in the individual backend files. At runtime, the operator can learn everything they need to know about the build by running `sysctl dev.portdrv.0`.

29.13.3 Challenge 3: Simulate a Big-Endian Host

You probably do not have a big-endian machine. Simulate the experience.

Write a one-time debugging mode, controlled by a build flag `PORTDRV_FAKE_BE`, in which all endian helpers are replaced by macros that assume a big-endian host. The code must still compile and run on amd64, but any multi-byte value that should have been converted will now be converted (the wrong way, for correctness, but that is the point). Load the driver with this flag on and exercise the simulation backend. If the driver works, congratulations: your use of endian helpers is consistent enough to survive an endianness flip. If it does not, the places where it fails are the places you missed an `htole` or `be32toh`.

This is a laboratory exercise, not a production pattern. The knowledge you gain, however, is exactly the knowledge that would protect you on a real big-endian platform.

29.13.4 Challenge 4: Cross-Build for arm64

Install or boot an arm64 FreeBSD guest in QEMU. Build the driver inside the guest, or cross-build it on your amd64 host using:

```
make TARGET=arm64 TARGET_ARCH=aarch64
```

Copy the resulting `.ko` into the guest and load it. Run the simulation backend and confirm the driver behaves identically. Note any warnings the arm64 compiler emits that the amd64 compiler did not; these are often alignment or size mismatches that deserve a fix.

29.13.5 Challenge 5: Write a README.portability.md

Author a `README.portability.md` that documents:

- the platforms on which the driver has been tested
- the backends it supports
- the build options the Makefile recognises
- any limitations, bugs, or known issues

This challenge is about writing, not coding. The artefact is as important to the driver's portability as the code itself. Keep it short; keep it honest.

29.13.6 Challenge 6: Create a Versioned Backend Interface

Add a version field to `struct portdrv_backend` and have the core reject a backend whose version does not match the core's expected version. Add a second version with a small additional field (for example, `configure_dma`), and confirm that a backend compiled against the older version still loads against a core that tolerates both, while a backend compiled against the newer version refuses to load against an older core.

This challenge is deeper than it looks. Versioning an interface requires careful thought about compatibility. The mechanics are worth practising once in an isolated context so that you know how to reach for them when your own drivers need them.

29.13.7 Challenge 7: Audit a Real FreeBSD Driver

Pick one of the drivers referenced in this chapter, for example `/usr/src/sys/dev/uart/`, and audit it against the patterns here. Answer:

- How is the core separated from the backends?
- Where are the register accessors, and do they use `bus_read_*`?
- What backends does the driver support, and how is each registered?

- Are there `#ifdef` chains, and are they well-justified?
- How is the Makefile structured, and what options does it expose?
- What could you change to make the driver more portable, in your judgement?

Write a short report in your lab logbook. The exercise is for you; the aim is to train your eye on real code.

29.13.8 Challenge 8: Measure the Cost of Backend Dispatch

The backend interface introduces one function-pointer indirection per register access. On a driver that performs millions of accesses per second, that indirection might or might not be measurable. Design an experiment to find out.

Use the simulation backend, so the measurement is not muddled by hardware latency. Modify the driver so it performs a tight loop of `portdrv_read_reg` calls in response to an `ioctl`. Time the loop with `clock_gettime(CLOCK_MONOTONIC)`. Compare two builds: one where the accessor dispatches through the backend, and one where the accessor calls `bus_read_4` directly.

Report your findings in your lab logbook. On a modern amd64 CPU, you will likely find the difference is in the single-digit percentage range or smaller. On an older or simpler CPU, it may be larger. Understanding the cost empirically, rather than guessing, is the right way to inform future design decisions.

29.13.9 Challenge 9: Add a Second FreeBSD Version Target

Take the driver and make it build on both FreeBSD 14.3 and FreeBSD 14.2. Identify one API difference between the two releases that your driver uses, wrap it in `__FreeBSD_version` guards, and confirm the module builds cleanly against both release headers.

Document the difference and your wrapping approach in a short note at the top of the file where the guard lives. This exercise introduces you to the practical side of multi-release compatibility, which you will need if you ever maintain an out-of-tree driver against a range of FreeBSD releases.

29.13.10 Challenge 10: Design a New Driver from Scratch

Starting from the patterns in this chapter, design a portable driver for a hypothetical device of your choice. The device should have at least two hardware variants (for example, PCI and USB), should have a simple register model, and should be amenable to simulation. Write:

- A one-page design document describing the driver's structure.
- The header file `mydev_backend.h` that defines the backend interface.
- A skeleton implementation of the core that compiles and links with placeholder functions.
- The Makefile with conditional backend selection.

You do not need to implement the full driver. The exercise is to practice the design phase, which is where most of the portability decisions are made. Share your design with a colleague or a mentor and listen for their questions; the questions reveal the parts of the design that were not yet obvious to a fresh reader.

29.14 Troubleshooting and Common Mistakes

Portability refactors fail in characteristic ways. This section catalogues the failures you are likeliest to meet, with concrete symptoms and concrete fixes.

29.14.1 Symptom: Build Fails with “Undefined Reference” After Splitting a File

A function that was static in the original file is now called from a different file. The compiler does not see it.

Fix. Remove the `static` qualifier from the function’s definition and add a declaration for it in the appropriate header. If the function should remain private to one file, find the caller in the other file and restructure so that the call crosses a proper interface rather than a private symbol.

29.14.2 Symptom: Build Fails with “Implicit Function Declaration” After Splitting

A file calls a function whose prototype it has not seen. This is the classic “I moved the definition but forgot the declaration” mistake.

Fix. Add `#include "portdrv.h"` or the appropriate header at the top of the calling file. If the header does not declare the function, add the declaration. Treat the prototype as part of the API surface; every function that crosses files must have a declaration in a shared header.

29.14.3 Symptom: Build Succeeds, but Load Fails with a Dependency Error

The module registration declares a `MODULE_DEPEND` on a module that is not loaded, or loads a version outside the expected range.

Fix. Use `kldstat` to see what is loaded. Load the required module first, then load yours. Check that the `MODULE_VERSION` in the dependency matches the range in `MODULE_DEPEND`. If you are in the middle of a refactor and the dependency chain is unstable, temporarily relax the version range or remove the dependency until the refactor settles.

29.14.4 Symptom: The Driver Loads, but `dmesg` Shows Only One Backend

The Makefile flags were not set as you expected. The banner reports only the backends whose `#ifdef` was satisfied.

Fix. Check the flags on the make command line. Look at the actual Makefile expansion:

```
make -n
```

The `-n` option prints what make would do without executing, and you can see whether `-DPORTDRV_WITH_PCI` actually made it into the compilation command. If not, something in the Makefile conditional is wrong.

29.14.5 Symptom: Register Reads Return Zero on ARM but Work on x86

You are reaching the device through raw pointer dereference instead of `bus_read_*`.

Fix. Replace every raw pointer dereference of device memory with a call to `bus_read_*` or `bus_space_read_*`. On amd64 the difference is invisible; on ARM it is the difference between working and broken.

29.14.6 Symptom: The Simulation Backend Works but the PCI Backend Hangs at Attach

The PCI resources were not allocated properly, or the backend-private data structure was not initialised correctly.

Fix. Add a `device_printf` trace at each step of the PCI attach function to narrow down where the hang happens. Check that `bus_alloc_resource_any` returned a non-NULL resource. Check that the backend's attach callback is called after `sc->sc_backend_priv` is installed; a NULL deref in the callback is easy to cause and obvious once seen.

29.14.7 Symptom: Build Fails on a Different FreeBSD Release

An API changed between the release you developed on and the release you tried to build on. The compiler complains about a missing function or a mismatched signature.

Fix. Bracket the affected code with `#if __FreeBSD_version >= N ... #else ... #endif` where N is the version at which the API took its current form. If the change is used in many places, abstract it into a helper in a compatibility header so the `#if` appears in only one place.

29.14.8 Symptom: The Driver Panics with “Locking Assertion Failed” After a Backend Split

The core and the backend are disagreeing about which lock is held at the call to the backend method.

Fix. Document, in a comment at the top of `portdrv_backend.h`, which lock the core holds at each method call. For each backend operation, either the core holds the softc mutex across the call or it does not, consistently. Audit every backend for conformance. The usual cause is a backend that acquires a lock the core already holds, or vice versa.

29.14.9 Symptom: The Driver Builds but `make clean` Leaves Stale Object Files

`bsd.kmod.mk` knows how to clean the KMOD's own files but may miss extras you added. The consequence is stale `.o` files that link into the next build and cause odd behaviour.

Fix. Delete the build directory and restart:

```
rm -f *.o *.ko
make clean
```

Then rebuild. Add the stale files to a `CLEANFILES` variable in the Makefile if they are generated by the build.

29.14.10 Symptom: Cross-Architecture Build Fails with “Cannot Find <machine/bus.h>”

The cross-compile environment is incomplete or the target architecture is not installed.

Fix. Follow the FreeBSD handbook on cross-compilation. Typically, you need to install the source tree for the target architecture and run `make TARGET=arm64 TARGET_ARCH=aarch64` from the top of the source tree. Building a single out-of-tree module requires the base system’s cross-build support, and the documentation for that is at the top of `/usr/src/Makefile` and in the “Cross-Build” handbook chapter.

29.14.11 Symptom: A `sysctl` You Just Added Does Not Appear

The `sysctl` was registered in a file that is not compiled into the module, or the registration code is not called.

Fix. Confirm that `portdrv_sysctl.c` is in `SRCS`. Confirm that the `sysctl` registration is called from the module load or attach path. Add a `device_printf` immediately before the registration call to confirm it executes.

29.14.12 Symptom: The Driver Works on Real Hardware but Not in `bhyve`

The simulated hardware in `bhyve` exposes slightly different behaviour from the real device. Often this is an MSI versus INTx difference, or a slightly different PCI capability list.

Fix. Check the capabilities your driver relies on. Use `pciconf -lvc` on both the host and the guest to see what differs. If a capability your driver needs is missing in the guest, the driver should detect this and degrade gracefully, rather than assuming the capability is always present.

29.14.13 Symptom: Performance Varies Wildly Between Architectures

You are accessing unaligned data with a direct cast. On `amd64`, unaligned access is cheap; on `arm64`, it can be several times slower if the hardware supports it at all.

Fix. Replace direct casts with `memcpy` followed by an endian helper, as shown in Section 5. The resulting code is fast on all architectures and correct on all of them.

29.14.14 Symptom: `kldunload` Hangs

Something is still using the module. Usually, a callout is still scheduled, an interrupt is still in flight, or a reference count is positive.

Fix. Add a `callout_drain` call to your detach path. Add `device_printf` traces at each step of detach to see which step is stuck. Look for open file descriptors on character devices the driver creates; `kldunload` will not complete while a device is open. Use `fstat` to identify which process is holding the device.

29.14.15 Symptom: Different Behaviour Between `make PORTDRV_WITH_SIM=yes` and `make PORTDRV_WITH_SIM=yes PORTDRV_DEBUG=yes`

Debug code is changing behaviour it should not. Often this is because a debug `printf` is calling a function that has a side effect, or because the debug path holds a lock longer than the release path.

Fix. Review every `PD_DBG` call. If any of them invokes a function rather than just formatting a value, replace the invocation with a pre-computed string. Debug printing must be an observational no-op; if the log statement has side effects, the release behaviour and the debug behaviour will diverge in ways that are hard to trace.

29.14.16 Symptom: Warning about “Incompatible Pointer Types” After Moving a Function

A function that took a pointer in the original file now takes a slightly different pointer in a new file. Usually the type has drifted between versions of a header, or a typedef has changed subtly.

Fix. Open both files and compare the declarations. If the types are genuinely the same, the declaration in a shared header is needed so both files see the same definition. If the types have drifted, pick the correct one and update the caller or the callee.

29.14.17 Symptom: `bus_dma` Fails with `ENOMEM` on amd64 but Works on arm64

Counter-intuitively, this is usually a `bus_dma_tag_create` configuration issue. The `lowaddr` field was set to a value that does not match the system’s actual memory layout.

Fix. Check the `lowaddr` and `highaddr` parameters. For a 64-bit device with no limits, use `BUS_SPACE_MAXADDR`. For a 32-bit device, use `BUS_SPACE_MAXADDR_32BIT`. Incorrect addresses produce spurious bounce-buffer allocations that can fail on systems with large memory.

29.14.18 Symptom: Userland Program Sees Different Data Than the Kernel Log Shows

The `uiomove(9)` call is moving bytes, but the endianness of the values differs between kernel and userland interpretation.

Fix. Decide what byte order your character-device interface uses. Document it in a comment on the `cdevsw` struct. Use `htole32/le32toh` or similar helpers when values cross the boundary between kernel and userland, just as you would for device boundaries.

29.14.19 Symptom: The Module Loads but `kldstat` Shows It in a Different Position Than Expected

The module’s dependencies are being loaded in an unexpected order. Usually this is a consequence of a missing `MODULE_DEPEND` declaration.

Fix. List every module your module depends on with `MODULE_DEPEND`. The kernel resolves the load order automatically; without the declarations, the order is unspecified.

29.14.20 Symptom: A Debug Build Consumes Much More Memory Than a Release Build

Debug statements are allocating memory on each invocation, or a debug-only data structure is growing without bound.

Fix. Review every `PD_DBG` call to ensure none of them perform allocations. If a debug-only data structure must grow, bound its size explicitly. A debug build that triggers out-of-memory conditions is more misleading than helpful.

29.14.21 Symptom: Tests Pass on the Development Machine but Fail in CI

The CI machine has a different FreeBSD version, a different set of loaded modules, or a different hardware configuration. The difference is exposing a latent bug.

Fix. Investigate the specific difference. If the CI environment is correct and the development machine is masking a bug, accept the bug and fix it. If the CI environment is misconfigured, fix the CI environment. Either way, the divergence between development and CI is a signal worth taking seriously.

29.14.22 Symptom: A Sysctl Can Be Read but Not Written, Even Though CTLFLAG_RW Was Set

The sysctl registration used a pointer to a read-only constant, or the handler ignores writes, or the node's parent is read-only.

Fix. Confirm the backing variable is writable. Confirm that CTLFLAG_RW is actually on the leaf node, not the parent. If a custom handler is installed, ensure it handles the write direction and not only the read direction.

29.14.23 Symptom: The Driver Panics on kldunload After Running for a Long Time

A long-running callout, taskqueue, or workqueue has been freed while still registered, or a resource reference is not released in the right order.

Fix. Call `callout_drain` and equivalent drain functions during detach. Ensure every resource allocated during attach is freed during detach, in reverse order. Walk through the attach and detach paths with a piece of paper and track every allocation; this catches the majority of shutdown panics.

29.14.24 Symptom: Cross-Compiled Module Refuses to Load on the Target

The cross-compiled module's ABI does not match the target kernel's ABI. This is almost always a version mismatch between the build tree and the target.

Fix. Ensure the `TARGET` and `TARGET_ARCH` in the cross-compile match the target exactly. If the target runs FreeBSD 14.3-RELEASE, build against the 14.3 source tree, not the 15-CURRENT tree. Module ABI compatibility across major releases is not guaranteed.

29.14.25 Symptom: A Backend's attach Method Is Called Twice on the Same Softc

Either two threads race to attach, or the backend is being re-attached without a detach in between.

Fix. Add a state flag in the softc that tracks whether the backend has been attached. The core should refuse to re-attach an already-attached softc. For symmetry, the detach should clear the flag. This catches most duplicate-attach bugs at the logic level, rather than waiting for a resource-exhaustion panic.

29.14.26 Symptom: Compiler Warns about an Unused Variable in a Conditional Block

The variable is used only when a specific `#ifdef` is active, and the build being compiled has the flag unset.

Fix. Move the variable declaration inside the `#ifdef` block if it is only used there. If the variable is used in both branches but the compiler cannot see one of them, add `(void)var;` in the unused branch to tell the compiler the variable is intentionally unused.

29.14.27 Symptom: The Build Succeeds but `kldxref` Complains About Missing Metadata

The `MODULE_PNP_INFO` declaration is malformed, or the array it points to is the wrong shape. `kldxref(8)` builds an index that maps hardware identifiers to modules, and a malformed `MODULE_PNP_INFO` breaks that index.

Fix. Check the arguments to `MODULE_PNP_INFO` against the documentation. The format string must match the layout of the identification array, and the count must be correct. A mismatch is caught at boot when `kldxref` runs, but it is easier to catch at build time by running `kldxref -v` on the directory and inspecting the output.

29.14.28 Symptom: A Backend That Worked Yesterday Fails Today After a Kernel Update

A kernel API that the backend depends on has changed in the new release. The core is fine; the backend is broken.

Fix. Check `UPDATING` in the kernel source tree for any notes about the API that changed. Check the release's `SHARED_LIBS` and module ABI notes. Wrap the affected API call in a `__FreeBSD_version` guard so that the driver supports both the old and new versions. Report the breakage to the backend's maintainer if it is not you.

29.14.29 Symptom: The Driver Rejects a Device That Should Work

The probe function's matching table is incomplete, or the device ID is being read in the wrong byte order.

Fix. Print the vendor and device IDs the probe function is seeing. Compare with the device's datasheet. If the values look byte-swapped, the probe is using the wrong accessor or the wrong mask. If the values look correct but the table does not include them, add the new IDs.

29.14.30 Symptom: A USB Backend Works with One Device but Not Another of the Same Type

USB devices have descriptors that can vary slightly between firmware revisions of the same product. The backend is matching too strictly on an optional field.

Fix. Review the USB matching logic. Match on vendor ID and product ID; ignore release numbers and serial numbers unless they are genuinely significant. The goal is to match the entire product family, not a specific build of it.

29.14.31 Symptom: After a File Split, Some `static` Functions Can No Longer Be Inlined

Functions that were `static inline` in one file are now called across files, which prevents inlining and can cause a small performance regression.

Fix. If the function is small and performance-critical, keep it in a header as a `static inline` so every file that includes the header gets its own copy. If the function is larger, move it to the appropriate `.c` file as a plain function. The compiler can still inline across translation units if link-time optimisation is enabled, but for kernel code that is rarely the case.

29.14.32 Symptom: make Recompiles Everything Even When Only One File Changed

The Makefile does not track header dependencies correctly. Changes to a shared header cause all files that include it to recompile, which is correct. But sometimes the Makefile is too aggressive and recompiles unrelated files.

Fix. Check that `bsd.kmod.mk` is generating dependency files (they usually have a `.depend` extension). If they are missing, add `.include <bsd.dep.mk>` or ensure the default include chain runs. For a small driver, full rebuilds are usually acceptable; for a large one, dependency tracking pays for itself.

29.14.33 Generic Guidance

If you are stuck on a portability problem that does not match any of the above, three habits usually help.

Narrow the configuration. If the driver fails only with certain backends enabled, build it with only one backend at a time and find the configuration in which it first breaks.

Narrow the architecture. If the driver fails on `arm64` but not on `amd64`, the bug is almost certainly endianness, alignment, or a type-size mismatch. Those are the only three common causes of architecture-specific behaviour in driver code.

Narrow the function. Add `printfs`, add `KASSERTs`, delete code until the smallest remaining driver still fails. Kernel bugs are hard to reason about, but they become tractable once you can trigger them from a tiny example.

Bisect the change. If the driver worked in an earlier commit and fails now, run `git bisect`. Kernel bugs are often specific regressions, and bisecting finds the first bad commit in $O(\log n)$ steps.

Read the compiler's warnings, even the annoying ones. A “set but not used” warning can point to a field that was renamed and forgotten. An “implicit function declaration” warning is a missed include. A “comparison between signed and unsigned” warning can be the source of the exact wraparound bug you are chasing. Compiler warnings are free advice; use them.

29.15 A Retrospective: The Patterns at a Glance

Before the closing note, here is a compact retrospective of the patterns this chapter introduced. Use it as a quick reference when you return to the chapter or when you are reviewing your own code. Each entry is the idea compressed to a sentence, with a pointer back to the section that introduces it in full.

Accessors over primitives. Wrap every register access in a driver-local function rather than calling `bus_read_*` directly in the code that cares about the device logic. The wrapper is a single point of change, it names the operation, and it lets you swap the implementation without touching callers. See Section 2.

Backend interface. Describe the variable part of the driver as a struct of function pointers. Each bus, hardware variant, or simulation becomes an instance of the struct. The core calls through the struct rather than knowing which backend is present. See Section 3.

File split by responsibility. Put core logic in one file, each backend in its own file, and each large subsystem concern (sysctl, ioctl, DMA helpers) in its own file. A reader should be able to guess which file to open from what they are looking for. See Section 4.

Fixed-width types. Use `uint8_t` through `uint64_t` for values whose size matters. Avoid `int`, `long`, unsigned, and `size_t` when the width is fixed by the device or protocol. See Section 5.

Endian helpers. Every multi-byte value that crosses the hardware boundary passes through `htole32`, `le32toh`, or their siblings. No exceptions. See Section 5.

bus_space(9) and bus_dma(9). Use the architecture-independent APIs for register access and for DMA. Never cast a raw pointer to device memory, and never compute physical addresses directly. See Sections 2 and 5.

Memory barriers. Use `bus_barrier` to enforce ordering between device-visible accesses when the order matters. Use `bus_dmamap_sync` to enforce coherency between CPU caches and DMA buffers. See Section 5.

Conditional compilation with restraint. Use build-time flags for features that should be compiled in or out. Keep `#ifdef` blocks coarse. Hide guards inside macros when they are used in many places. See Section 6.

__FreeBSD_version. Use sparingly to guard code that depends on a specific kernel release. Centralise the guards in a compatibility header if they appear more than a few times. See Section 6.

Cross-BSD wrapping. Isolate OS-specific APIs behind wrapper macros when cross-BSD support is needed. Keep the core free of OS-specific code. See Section 7.

Module versioning. Use `MODULE_VERSION` and `MODULE_DEPEND` to make upgrade and load-order issues visible to the kernel. Bump versions when observable behaviour changes. See Section 8.

Build matrix. Maintain a list of supported build configurations. Run a script that builds every configuration before a release. See Section 8.

Portability README. Document supported platforms, backends, build options, and limitations in a short readme. Update it whenever any of those change. See Section 8.

Telemetry. Expose key counters through a sysctl tree. Bug reports that include sysctl output are easier to triage than reports that do not. See Section 8.

Print this list or keep a copy in your notes. The patterns work because they reinforce one another, and seeing them together is the fastest way to internalise the shape of a portable driver.

29.16 Portability Antipatterns: What to Unlearn

Having seen the patterns that work, it helps to name the patterns that do not. Most driver portability disasters are not spectacular failures of judgement; they are small, comfortable habits that compound. A driver author who copies a working function without understanding it will be tempted, six months later, to copy it again with

a small tweak. A driver that has one `#ifdef` for one architecture will, six months later, have five. Each step feels harmless; together they produce code that nobody wants to touch.

This short catalogue names the antipatterns so that you can recognise them in your own code and in code you review. The goal is not to shame; it is to give you the vocabulary to identify the drift early and redirect it.

29.16.1 The “It Works on My Machine” Driver

A driver that only has one target, one FreeBSD version, and one bus hides portability bugs rather than lacking them. The code may compile and run cleanly for years, but the moment someone else tries to build it on a different architecture or a newer release, every shortcut is exposed at once. The antidote is not to test on every platform from day one; it is to write as if you will one day, so the eventual port is a one-week exercise rather than a three-month one. Fixed-width types, endian helpers, and `bus_read_*` are cheap insurance against the future you cannot yet see.

29.16.2 The Silent `#ifdef`

The `#ifdef` block with no comment explaining why it is there is almost always a mistake. If the block is guarding truly architecture-specific code, say so, and say what the alternative branch does. If it is guarding a workaround for a bug in a specific chip, say which chip, which bug, and when it might be removed. A driver file with undocumented `#ifdefs` is a file where future-you will not know whether the branch is still needed. Document, or delete.

29.16.3 The Casual Cast

A cast from a pointer to an integer type, or from an integer of one size to another, is almost always a bug in waiting. On amd64, a pointer is 64 bits and a `long` is 64 bits, so the cast happens to work. On an i386 system, a pointer is 32 bits and `long` is 32 bits, so the cast happens to work. On 32-bit ARM, a pointer is 32 bits but `uint64_t` is, of course, 64 bits; here the cast changes size and triggers a warning only on a subset of builds. Casts that the compiler accepts silently on one platform and warns about on another are precisely the casts you should avoid. If the logic requires a conversion, use `uintptr_t`, which is explicitly designed to hold a pointer, and document why the cast is needed.

29.16.4 The Optimistic Register

A register read or write that the driver assumes will succeed is an optimistic register. On real hardware, a read from an unmapped address can return `0xFFFFFFFF`, can machine-check the CPU, or can silently produce garbage, depending on the bus and platform. On a simulation backend, the same read returns a known constant. Drivers that never check for the pathological read are drivers that work until one day the hardware is powered off or behind a PCI bus that disconnected due to a hotplug event. Defensive coding at the accessor level (does the return value look sane?) is cheaper than debugging the panic that follows.

29.16.5 The Ad-Hoc Version Check

`if (major_version >= 14)` sprinkled across the driver is an antipattern. Version checks belong in a compatibility header, not scattered through the code. A single `#define PORTDRV_HAS_NEW_DMA_API 1` in one place, guarded by one `__FreeBSD_version` test, keeps the rest of the driver clean. Scattered version checks are also fragile: when the API changes again, you will forget one of the dozen scattered checks, and the driver will fail in a subtle way on one release.

29.16.6 The “Temporary” Hack

A comment that says `// TODO: remove before release` or `// FIXME: arm64` is a promise to yourself. Ninety percent of such promises are never kept. Before you write a temporary hack, ask whether there is a way to solve the problem properly in half an hour. Often there is. When there truly is not, the hack should come with a tracking ticket and a calendar reminder. Without those, “temporary” becomes “permanent” on the timescale of a single release cycle.

29.16.7 The Duplicated Constant

A magic number written once is a design decision. A magic number written twice is a bug waiting for one of the copies to be updated without the other. Constants for register addresses, bit masks, timeouts, and limits should live in one header and be included everywhere they are used. When you catch yourself typing the same number in two places, stop and put it in a `#define`.

29.16.8 The Growing struct softc

A `softc` that accumulates fields across many commits without any re-organisation is a code smell. It is a sign that every feature added to the driver put its state directly into the `softc` without asking whether the state belonged somewhere else. When a `softc` has more than twenty fields, consider grouping related fields into nested structs: `sc->dma.tag`, `sc->dma.map`, `sc->dma.seg` rather than `sc->dma_tag`, `sc->dma_map`, `sc->dma_seg`. The refactor is small, and the resulting structure reads better in logs and in debugger output.

29.16.9 The Uneven Test Coverage

A driver whose tests only cover the happy path is a driver whose error paths will bite someone in production. Every function that can fail should be testable, and should be tested, in at least the obvious failure mode. The simulation backend makes this cheap; use it to inject failures (out of memory, DMA timeout, device not responding) and assert the core handles them gracefully. A driver that has never been tested against a failing backend is a driver whose error paths are wishful thinking.

29.16.10 The Gratuitous Abstraction

It is possible to err in the other direction. A driver with a twelve-method backend interface, three layers of wrapping, and an abstract factory for backend instances is likely over-engineered for the concrete need. Abstractions have a cost: they obscure what the code does, they spread knowledge across files, and they make debugging

harder. The right level of abstraction is the one that lets you solve today's problem cleanly without making tomorrow's problem meaningfully easier. When in doubt, start with less abstraction and add it when a second concrete use case justifies the cost.

29.16.11 The Monolithic Commit

A single commit that introduces accessors, splits files, adds a backend interface, and creates a Makefile matrix is impossible to review, impossible to bisect, and impossible to revert cleanly. Split the work: one commit per step, each commit leaving the driver in a buildable, loadable state. Small commits also communicate design intent: the commit message for “introduce accessors” tells a reader exactly what that commit is trying to achieve, whereas a grand commit titled “refactor for portability” tells them nothing useful.

Recognising these antipatterns in your own code is a sign of maturity. The patterns in the main body of the chapter are what you should do; the antipatterns here are what to watch for. When you find one in a driver you wrote last year, do not be hard on yourself; simply fix it. Everyone writes antipatterns at some point. What separates a seasoned author is the speed at which they notice and correct them.

29.17 Portability in the Real World: Three Short Stories

Principles land better when they are tied to consequences. Here are three short stories based on patterns that recur in real FreeBSD development. No single anecdote below is one specific incident; each is a composite of issues that experienced driver authors have seen more than once. The details are illustrative, not verbatim. The lessons are exactly as stated.

29.17.1 Story 1: The Endian Bug That Slept for Five Years

A driver for a storage controller was written in 2019 on an amd64 workstation, shipped to a handful of production servers, and ran cleanly for five years. In late 2024, a team migrating to arm64 servers for power efficiency loaded the same driver on the new hardware. Reads worked. Writes succeeded. Sporadic data corruption appeared a week later, limited to a small fraction of I/O requests.

The bug was a single line in the driver's descriptor setup: a raw struct assignment that deposited an amd64-native little-endian value into a ring descriptor that the hardware interpreted in its own byte order. On amd64, the hardware byte order happened to match the host byte order, so the raw assignment worked by coincidence. On arm64, the kernel runs little-endian too, so the code did not trip immediately, but the hardware in question had a newer firmware revision that changed the descriptor byte order for certain command types. The one byte order that the driver had been relying on by coincidence disappeared, and only a specific workload (streaming writes crossing a descriptor boundary) hit the corruption.

The fix was a one-line change to use `htole32` for every descriptor field. The diagnosis took three weeks, spanning two teams, because nobody had written the first driver expecting it to encounter a second platform or a hardware revision. The lesson is that endianness bugs do not announce themselves the moment they are introduced; they can sleep for years until a change somewhere else wakes them. Writing `htole32` from the start is cheaper than paying for the sleep.

29.17.2 Story 2: The Conditional Block Nobody Remembered

A network driver that supported both PCIe-attached and embedded FDT-attached variants carried a two-line `#ifdef PCI` block in its attach path. The block enabled an error-handling feature that mattered only on the PCIe variant, because the embedded variant used a different subsystem for the same purpose. The `#ifdef` had no comment.

Three years later, the driver author moved on. The driver acquired a new maintainer. A kernel update changed the semantics of the feature toggled by the `#ifdef`, and the driver started reporting spurious errors on the embedded variant. Nobody knew whether the `#ifdef` was still needed, whether it had ever been needed, or whether it was covering up a bug in either the embedded or the PCIe path. A week of investigation revealed that yes, the `#ifdef` was still correct; it was documenting a real asymmetry between the two paths; but because it had no comment, every new reader had to re-derive that knowledge from scratch.

The fix was to add a three-line comment above the `#ifdef`, citing the subsystem difference and naming the kernel commit that introduced the need for the block. Total cost of the fix: ten minutes. Total cost of the investigation across three years: about forty engineer-hours.

The lesson is that conditional compilation blocks age into mystery very quickly. Every `#ifdef` should have a comment telling a future reader why it is there. The comment pays for itself the first time anyone else touches the file.

29.17.3 Story 3: The Simulation Backend That Saved a Release

A driver for a custom sensor had a simulation backend added at the end of its first development cycle. The addition took two days of work and was considered a nicety rather than a necessity. The real hardware was rare, expensive, and kept in a lab across the building from the main developer.

Late in the release cycle, a regression appeared: under a specific sequence of user operations, the driver would deadlock on unload. Reproducing the bug on the real hardware required booking the lab, which was in use for validation of a separate product. The developer instead used the simulation backend, reproduced the deadlock within an hour, identified the missing `mtx_unlock` that caused it, and shipped the fix the same day.

The lesson is that simulation backends are not just for testing; they are for reproducing bugs without contending for scarce hardware. The two days invested in the simulation backend saved perhaps a week of schedule pressure and arguably saved the release. The pattern is worth the cost.

These three stories illustrate what the chapter has been arguing abstractly. Endianness bugs sleep. Conditional blocks age. Simulation backends pay back quickly. The specific incidents are illustrative, but the patterns are real, and they repeat across drivers and across years. Knowing them in advance is the difference between writing a driver you will maintain and writing a driver that will embarrass you.

29.18 A Short Closing Note on Discipline

This chapter has asked you to be patient with small, structural decisions. The patterns here, accessors, backend interfaces, file splits, endian helpers, version tags, are not thrilling when you first meet them. They are the kind of thing that a new driver author learns because an older one said to. The younger self reads about them, nods, and writes them down as a checklist.

Reading them as a checklist misses the point. The patterns work because they shape the habits with which you write new code. The first driver you write with accessors in place is also the first driver in which you never forget to check for a NULL resource, because the accessor is the natural place to do the check and you wrote it once. The first driver you structure into core and backends is also the first driver in which adding a feature means touching one file. The first driver you version with `MODULE_VERSION` and `MODULE_DEPEND` is also the first driver whose upgrade path is obvious to your users. The benefits are not in the patterns; they are in the habits the patterns create.

Habit takes time. You will write the next driver faster than you wrote this one, and the one after that faster still, not because the patterns have shortcuts but because your fingers stop reaching for the long way. The cost of portability, measured per driver, decreases sharply after the first. That is why this chapter matters for the rest of your career, and not just for this book.

The great FreeBSD driver authors did not become great by memorising the patterns in this chapter. They became great by applying them every time, even on the small, boring drivers that nobody was going to read. That practice scales. Pick it up early.

29.18.1 What Success Looks Like

A signpost, since you have come this far. When the patterns in this chapter start to feel automatic, something changes about the way you write drivers. You stop writing `bus_read_4(res, 0x08)` and reach for `port_drv_read_reg(sc, REG_STATUS)` instinctively. You stop wondering which file a new function belongs in; you know. You stop thinking about whether to `htole32` or not; if the value crosses a hardware boundary, it gets the helper, full stop. You stop arguing in code review about whether a flag should be compile-time or runtime, because you have a working heuristic. These are not trivial gains. They are the difference between writing drivers and writing drivers well.

The shift does not happen on any specific day. It happens gradually, over the course of three or four drivers written in this style. One day you notice that a pattern you used to think about is now just what your fingers do. That is the sign that the habit has taken. Keep going.

29.18.2 A Note on the Path Ahead

Part 7 of this book is titled “Mastery Topics” for a reason. Chapter 29 is the first chapter where the material is about craft rather than technique. Chapters 30 through 38 will continue in this spirit: each chapter addresses a topic that separates a competent driver author from a seasoned one. Virtualisation and containerisation in Chapter 30. Interrupt handling and deferred processing in Chapter 31. Advanced DMA patterns in Chapter 32. And onward.

The patterns in this chapter will reappear throughout those chapters. A virtualisation-aware driver uses the same backend abstraction to add a `virtio` backend. An interrupt-driven driver uses the same accessors to manage interrupt enable and acknowledge registers. An advanced DMA driver uses the same `bus_dmamap_sync` calls in more sophisticated sequences. Everything you learned here is a foundation for what comes next.

Treat Chapter 29 as the hinge between the driver-subsystem chapters of Part 6 and the mastery chapters of Part 7. You have crossed it now. The rest of the book builds on what you learned here.

29.19 Mini-Glossary of Portability Terms

A short glossary follows, aimed at the reader who wants to revisit the chapter's core vocabulary in one place. Use it as a refresher, not as a replacement for the explanations in the main text.

- **Backend.** An implementation of the driver's core interface for a specific bus, hardware variant, or simulation. A driver can have multiple backends in the same module.
- **Backend interface.** A struct of function pointers that the core calls into. Each backend provides an instance of this struct with its own implementations.
- **Core.** The hardware-independent logic of a driver. Calls into the backend interface but knows nothing about specific buses.
- **Accessor.** A driver-local wrapper around a primitive operation such as `bus_read_4`. Gives the driver a single point of change when the primitive or the backend differs.
- **bus_space(9).** The architecture-independent API for memory-mapped and I/O-port device access. Defined in `/usr/src/sys/sys/bus.h` and the per-architecture `/usr/src/sys/<arch>/include/_bus.h` files.
- **bus_dma(9).** The architecture-independent API for direct memory access by devices. Defined in `/usr/src/sys/sys/bus_dma.h`.
- **Endianness.** The byte order in which a multi-byte value is stored in memory. Handled in FreeBSD through helpers declared in `/usr/src/sys/sys/endian.h`.
- **Alignment.** The requirement that a multi-byte value be stored at an address that is a multiple of its size. Relaxed on amd64 and i386; strictly enforced on some ARM cores.
- **Word size.** The native integer width of a CPU. 32-bit and 64-bit FreeBSD platforms exist. Drivers should use fixed-width types whenever the size matters.
- **Fixed-width type.** A C type whose size is guaranteed: `uint8_t`, `uint16_t`, `uint32_t`, `uint64_t`, and their signed counterparts. Declared in `/usr/src/sys/sys/types.h` and its includes.
- **__FreeBSD_version.** An integer macro that identifies the FreeBSD kernel release the driver is being compiled against. Use to guard code that depends on a specific kernel API version.
- **Kernel options.** Named compile-time flags declared in kernel configuration files. Used to enable or disable optional features. Your driver defines `PORTDRV_*` flags in the Makefile and guards code with `#ifdef PORTDRV_*`.
- **Build matrix.** A table of supported build configurations, along with whether each configuration is expected to compile and whether it has been tested. Validated by a script that iterates over the matrix.
- **Backend versioning.** A version field in the backend interface struct that lets the core reject a backend compiled against a mismatched interface.
- **Compatibility wrapper.** A macro or function that abstracts a kernel API difference between FreeBSD releases or between different BSDs. Used sparingly, because each wrapper is a maintenance cost.
- **Simulation backend.** A software-only backend that does not talk to real hardware. Used for testing the core without requiring a physical device.
- **Module dependency.** A declaration, via `MODULE_DEPEND`, that one module requires another at a specific version range. Enforced by the kernel at load time.
- **Subregion.** A named slice of a larger memory-mapped region, carved out with `bus_space_subregion`. Used to give each logical bank of registers its own tag and handle.
- **Memory barrier.** An instruction or compiler directive that enforces ordering between memory operations. Issued via `bus_barrier` or the `atomic_thread_fence_*` family.

- **Bounce buffer.** A temporary buffer used by `bus_dma` when a device cannot reach the physical address of a requested buffer directly. Transparent to the driver; made effective by the cooperation between `bus_dma_tag_create` and `bus_dmamap_sync`.
- **Include graph.** The directed graph of which source or header files include which others. A clean include graph is a sign of a well-structured driver.
- **Compile-time telemetry.** Build-time information (version, build date, enabled backends) embedded into the binary and printed at load time. Useful for debugging and bug reports.
- **API vs. ABI.** The application programming interface is what programmers see; the application binary interface is what linkers see. A change can preserve one and break the other; portability practices must consider both.
- **Stacked backend.** A backend whose implementation wraps another backend, adding logging, tracing, delaying, or other behaviour. Useful for debugging and replay tools.
- **Memory-mapped I/O (MMIO).** A technique in which device registers appear as memory addresses that the CPU can read or write. Accessed through `bus_space` in FreeBSD rather than through raw pointers.
- **Port-mapped I/O (PIO).** An older scheme, common on i386, in which device registers are accessed through a separate I/O address space. Also accessed through `bus_space`, which abstracts the difference between MMIO and PIO.
- **DMA segment.** A contiguous physical address range that a device accesses directly. `bus_dma_segment_t` describes one; a DMA transfer may involve multiple segments.
- **Coherency.** The property that a memory location presents the same value to all observers (CPU, DMA engine, caches) at the same time. On some architectures, achieving coherency requires explicit `bus_dmamap_sync` calls.
- **KASSERT.** A kernel-space assertion macro. Panics the kernel when the condition is false; compiled out in release builds. Used to check invariants during development.
- **DEVMETHOD.** A macro in a `device_method_t` array that binds a Newbus method name to an implementation. Terminated by `DEVMETHOD_END`.
- **DRIVER_MODULE.** The macro that ties together a Newbus driver's name, parent bus, driver definition, and optional event callbacks. Usually appears once per bus attachment.
- **MODULE_PNP_INFO.** A declaration that describes the device IDs a driver claims, so `devmatch(8)` can auto-load the module when matching hardware appears. Orthogonal to `MODULE_VERSION` and `MODULE_DEPEND`.
- **Loader tunable.** A variable set from `/boot/loader.conf` that a kernel module reads via `TUNABLE_INT` or similar. The most common way to parameterise a driver at boot time.
- **sysctl node.** A named variable exposed through the `sysctl(3)` interface. Can be read-only, writable, or a procedural handler. A portable driver exposes counters and options via `sysctl`.
- **device hints.** A legacy but still-useful FreeBSD mechanism for supplying per-instance configuration (IRQs, I/O addresses) from `/boot/device.hints`. Parsed by the Newbus framework.
- **Kernel options (opt_*.h).** Per-configuration header files generated by the build system, used to guard kernel features with `#ifdef`. Distinct from module-level build flags.
- **Module blacklisting.** A mechanism (via `/boot/loader.conf` or `kldxref`) for preventing a module from being auto-loaded. Useful during debugging.
- **bus_space_tag_t and bus_space_handle_t.** Opaque types that together identify a memory region. The tag says which kind of space; the handle points into it. Passed as the first two arguments to `bus_space_read_*` and `bus_space_write_*`.
- **Attach path.** The sequence of Newbus calls (`probe`, `attach`, `detach`) that binds a driver to a device instance. Portable drivers isolate bus-specific logic inside this path.

- **Softc lifetime.** The lifetime of a struct `softc` instance, bounded by `attach` (allocation) and `detach` (free). A portable driver treats the `softc` as the authoritative per-instance state, not a collection of module-global variables.
- **Hot path.** The code path executed on every I/O operation. Performance-critical code. Keep it small and free of unnecessary checks.
- **Cold path.** Code executed rarely, such as during `attach`, `detach`, or error recovery. Correctness trumps performance; explicit error handling belongs here.
- **Hotplug.** The ability of a device to be physically added or removed from a running system. PCI hotplug is the main FreeBSD use case; drivers that are expected to support it must handle `detach` correctly at arbitrary points.
- **Refactor.** A structural change to code that preserves its observable behaviour. Portability refactors typically introduce accessors, split files, and introduce backend interfaces; none of these should change what users see.
- **Invariant.** A property that a piece of code guarantees at a given point in execution. Declaring invariants with `KASSERT` makes them checkable during development and self-documenting thereafter.
- **Build configuration.** A combination of Makefile flags, kernel options, and target platform that defines a specific build. A driver's build matrix is the set of configurations it officially supports.
- **Binary drop.** A delivery of a driver as a precompiled `.ko` file rather than as source. Portable binary drops are harder than portable source drops; platform and version compatibility become runtime concerns.
- **Source drop.** A delivery of a driver as source code, typically a tarball or a git repository. Easier to port than a binary drop, because the user's kernel headers and compiler adapt to their platform.
- **Driver lifecycle.** The set of transitions a driver goes through from load to unload: `register`, `probe`, `attach`, `operate`, `detach`, `unregister`. Portable drivers document where each transition is permitted to fail.
- **Migration path.** The documented steps a user follows when upgrading from an older driver version to a newer one. A portable driver keeps migration paths short and reversible.

Keep this glossary nearby as you read Chapter 30 and the chapters that follow. Each term recurs often enough that a quick reference pays for itself.

29.20 Frequently Asked Questions

New driver authors tend to ask the same questions as they work through their first portability refactor. Here are the most common ones, with short, pointed answers. Each answer is a signpost, not an exhaustive treatment; follow the bread crumbs back to the relevant section of the chapter if you want more detail.

Q: How much of this applies if my driver only needs to run on amd64?

All of it, except the architecture-specific sections. Even if you never plan to support `arm64`, the discipline of using fixed-width types, endian helpers, and `bus_read_*` costs almost nothing and protects you against bugs that would show up on `amd64` too, just in edge cases. The backend interface and the file split help readability and maintenance even when there is only one backend. If your first driver ever becomes a second driver, you will be glad you started the habit.

Q: Should every driver have a simulation backend?

No, but many should. The question to ask is whether the driver can be meaningfully exercised without the real hardware. If yes, a simulation backend lets you run unit tests and catch regressions on CI runners without

provisioning the hardware. If no, a simulation backend is extra code with no payoff. Storage drivers, network drivers, and many sensor drivers benefit; a driver for a very specific piece of custom hardware might not.

Q: Why use a struct of function pointers instead of FreeBSD's kobj (9) framework?

For small drivers, a plain struct is simpler and easier to read. kobj (9) is a capable system that adds compile-time method resolution and interface-file processing, which is why Newbus uses it. For a driver with a handful of backends and a simple method set, the plain struct gets you most of the benefit with a fraction of the machinery. If your driver grows large enough that the ceremony of kobj pays off, switching is a local refactor.

Q: My driver has only one backend today. Should I still define a backend interface?

Probably yes, if the effort is small. A single-entry interface costs almost nothing and lets you add a second backend cheaply later. If the effort is large because the driver is tightly coupled to its current bus, consider refactoring first for separation of concerns, and add the interface when a second backend is actually needed.

Q: How do I decide where to draw the line between core and backend?

A practical test: can you imagine writing the same logic for a simulated backend without changing the code at all? If yes, that logic belongs in the core. If the logic inherently mentions PCI, USB, or any specific bus, it belongs in the backend. In doubtful cases, put the logic in the core and see whether the simulation backend can use it unchanged; if not, move it.

Q: What do I do with a driver that uses a raw volatile pointer to device memory?

Replace every access with `bus_read_*` or `bus_write_*`. This is a one-time refactor and it is the single biggest portability improvement you can make to a driver.

Q: How often should I re-run my build matrix?

On every commit to the main branch, if you can. Weekly if you cannot. The longer a regression lives unnoticed, the more work it takes to bisect. Automated matrix validation is cheap; a bisecting sleuth session is expensive.

Q: Can I use `#ifdef __amd64__` inside a driver source file?

Sparingly. Prefer machine-specific headers for non-trivial architecture differences. If you genuinely need a small amount of platform-specific code, a short `#if` block is acceptable, but if it grows beyond a few lines, consider moving the code to a separate file and including it via an architecture-selective path or a wrapper header.

Q: Why does the reference driver's Makefile fall back to SIM if no backend is selected?

So that a plain make produces a loadable module. A driver that refuses to build without a specific flag set is a friction point for new contributors and for CI. The SIM fallback means anyone who just runs make gets something they can load and explore, even without any hardware.

Q: When should I add `MODULE_VERSION` and `MODULE_DEPEND`?

Always, even for internal drivers. `MODULE_VERSION` is a one-line declaration that tells consumers which release of your driver they have. `MODULE_DEPEND` prevents incompatible combinations from loading. Both are essentially free, and they save hours of debugging when something is mismatched.

Q: How does a cross-BSD driver handle things like `malloc` that have different signatures?

Through a per-OS compatibility header that maps a driver-local wrapper (for example, `portdrv_malloc`) onto the correct OS primitive. The core calls `portdrv_malloc`, and the compatibility header contains a branch per OS. The core is never polluted with OS-specific code; only the wrapper header is.

Q: Is it worth making a production driver cross-BSD compatible if my users are all on FreeBSD?

Generally no. The upkeep cost is real, and compatibility work that is not exercised tends to rot. The patterns in this chapter still benefit the FreeBSD-only driver, but the explicit wrapper layer for other BSDs should only be introduced when there is a concrete plan to support them.

Q: How do I test endianness if my only hardware is amd64?

Use QEMU with a big-endian target such as `qemu-system-ppc64`. Boot a FreeBSD big-endian release in the guest, cross-build your driver for `powerpc64`, and run it there. This is the only reliable way to catch endianness bugs short of deploying on real big-endian hardware. Challenge 3 in this chapter gives a lighter-weight approximation using a build flag.

Q: My driver compiles on FreeBSD 14.3 and fails on 14.2. What is going on?

An API added in 14.3. Use `__FreeBSD_version` to guard the use of the new API and provide a fallback for the older release. If the new API is essential and cannot be worked around, document the minimum supported version clearly in `README.portability.md`.

Q: Why does the chapter keep talking about the UART driver?

Because `/usr/src/sys/dev/uart/` is one of the clearest examples in the FreeBSD tree of a driver with a clean core/backend separation and multiple, well-factored backends. Reading it alongside this chapter is one of the best ways to internalise the patterns, and it is why I reference it so often.

Q: Are there other FreeBSD drivers that illustrate these patterns well?

Yes. Beyond `uart(4)`, the `sdhci(4)` family in `/usr/src/sys/dev/sdhci/` is a good study: core logic in `sdhci.c`, with PCI and FDT backends in `sdhci_pci.c` and `sdhci_fdt.c`. The `ahci(4)` driver in `/usr/src/sys/dev/ahci/` also separates its core from its bus attachments cleanly. The `ixgbe` driver in `/usr/src/sys/dev/ixgbe/` shows layered accessors and a tight separation between hardware-specific helpers and the rest of the driver. Reading any of these as a study-alongside source is valuable; picking the one closest to the kind of driver you are writing is even more so.

Q: My driver already has five hundred lines. Is it too late to refactor for portability?

No. Five hundred lines is small. Even five thousand lines is refactorable, though the refactor will take a few days. The important thing is to do the refactor incrementally: introduce accessors first, then split files, then introduce the backend interface. Each step leaves the driver working, and you can stop at any point if priorities change.

Q: What if my hardware has only one physical variant and is unlikely to ever change?

Then portability across hardware variants is not a concern for you. But portability across architectures, across FreeBSD versions, and between real hardware and a simulated one are still worth the effort. Even a single-variant driver benefits from having a simulation backend for testing and from using the endian helpers so it works on every architecture FreeBSD supports. The patterns here are not only about “what if new hardware comes along”; they are also about making your one driver robust in its current form.

Q: How do I get review feedback on a portability refactor?

If you are working on an in-tree driver, send the change to `freebsd-hackers@freebsd.org` or to the relevant subsystem maintainer. For out-of-tree drivers, ask a colleague or a mentor. Either way, break the refactor into small patches: one patch per step. A reviewer can keep up with six small patches; they cannot keep up with one large one. The smaller you make each change, the better the review you will receive.

Q: Should I rename the driver when I refactor it for portability?

Usually no. The refactor is an internal change; users do not need to care. The module name, the device node name, and the sysctl tree name should all remain stable so that existing configurations continue to work. The internal file structure is a detail the user never sees.

Q: How do I justify the time spent on portability to a manager who wants features?

Talk in terms of concrete future costs. “If we support a second hardware variant later, the current driver will take three weeks to fork; a refactor now takes one week, and any subsequent fork takes two days.” “If a customer reports a bug on arm64, the current driver has to be ported before we can even reproduce; the refactored driver would reproduce in a VM.” Portability pays for itself in the second year, not the first, and communicating that horizon is how the conversation gets easier.

Q: Can two drivers share a common library module?

Yes. FreeBSD supports module-to-module dependencies via `MODULE_DEPEND`. You can build a common library as its own module and have your drivers depend on it. This is how some of the in-tree drivers share helper code. The mechanism is a little fiddly for a small project, but it is the right approach when two drivers genuinely share significant helper code.

Q: What is the minimum test I should run after a refactor before committing?

Build. Load. Run the basic use case once. Unload. Rebuild from clean. Re-load. If all these succeed, the refactor is probably safe. If the driver has a test suite, run it. If there is no test suite, the refactor is a good time to write one, because the simulation backend makes tests easy to run.

Q: How does portability relate to security?

Portable code is usually more secure than non-portable code, because the discipline that supports portability (clear interfaces, narrow responsibilities, careful type usage, strict input validation) also supports security. A driver that uses `uint32_t` consistently is less likely to overflow. A driver with a clean backend interface is less likely to have ill-defined error paths that leak memory. A driver that is tested on multiple architectures is more likely to have encountered edge cases that expose security bugs. This is not a formal relationship, but it is a real one, and the two concerns reinforce each other more than they conflict.

Q: How much of this chapter applies to drivers I download from vendors rather than write myself?

Less directly, but still some. When you review a vendor-provided driver, the patterns in this chapter let you evaluate its quality. A vendor driver that is a single file of five thousand lines with scattered `#ifdef` chains is higher risk than one that is structured into core and backend. When you consume such a driver, you can use the audit questions from Challenge 7 to form an opinion about its maintainability. If the answers are discouraging, factor that into how much you rely on the driver.

Q: What is the single most important idea in this chapter?

That the cost of portability is an investment you make once per driver, and that the return on that investment is proportional to how long the driver lives. The specific patterns (backend interfaces, endian helpers, fixed-width types, file splits) are all instances of the same underlying discipline: **separate the details that can change from the logic that should not change, and express that separation in code rather than in comments**. Learn that discipline and you will apply it everywhere, not just in FreeBSD drivers.

Q: How do I debug a driver that panics only on arm64?

Start with the panic message itself; it often names the offending function and the kind of fault (alignment, unaligned access, null pointer, invalid memory). Rebuild the driver with debug symbols (`CFLAGS+= -g`), load it on the arm64 target under `kgdb` or `ddb`, and reproduce the panic. Unaligned access faults are the most common arm64-only panic; they almost always mean a struct member was accessed at an offset that is not aligned to its type's natural alignment. Fix these by packing or reordering struct fields, or by using `memcpy` for reads and writes that must be unaligned by design.

Q: What is the right balance between runtime flexibility and compile-time flexibility?

A useful rule of thumb: if a choice varies between deployments of the same binary, make it a runtime tunable. If a choice varies between releases of the driver (because a feature is new, or because a platform is not supported), make it a compile-time flag. Tunables let users reconfigure without rebuilding; compile-time flags keep the binary small and sharp. When in doubt, start with compile-time and promote to runtime only when a concrete need arises.

Q: Is it OK to disable a backend at runtime via a sysctl?

Yes, but only for backends that support re-initialisation cleanly. Disabling a backend mid-operation can leak resources or leave the driver in an inconsistent state. The safer pattern is to expose the choice at attach time (via a loader tunable) and to treat the runtime sysctl as read-only except during a well-defined maintenance window. If you do expose runtime disable, test it thoroughly, including under I/O load.

Q: How do I contribute a portability fix upstream to the FreeBSD project?

If the fix is to an existing in-tree driver, the canonical path is to file a bug in the FreeBSD Bugzilla, attach a patch in unified diff format (or, for larger changes, as `git` commits), and either wait for a committer to pick it up or post the patch to freebsd-hackers@freebsd.org for review. Include a clear description of the bug, steps to reproduce, and the platform(s) on which the bug appears. Smaller patches get reviewed faster; if your fix is large, split it into independently reviewable parts.

Q: Does every driver need a `README.portability.md`?

Every driver that ships beyond a single developer's machine benefits from one. The file does not need to be long; even three sections (supported platforms, build configurations, known limitations) are enough to cover the essentials. The file pays for itself the first time a user asks a question that the README answers; you send a link instead of typing the answer again.

Q: What happens if I forget to call `bus_dmamap_sync` after a DMA?

On amd64, usually nothing visible, because the CPU and DMA engine share the same memory and cache hierarchy. On arm64 with a device that is not cache-coherent, you will see stale data: the CPU reads its cached copy rather than the memory the device wrote. The bug is architecture-dependent, intermittent, and hard to reproduce without explicit test cases. This is why `bus_dmamap_sync` is mandatory in portable code even when a specific target does not strictly require it.

Q: Should I use `kobj(9)` or plain function pointers for my backend interface?

For fewer than five or six methods, plain function pointers in a struct are simpler and easier to debug. For larger interfaces, especially those with inheritance or method overriding, `kobj(9)` starts to pay back the ceremony. If you are unsure, start with function pointers; migrating later is a local refactor, and you will have a clear picture of whether `kobj` is worth it by then.

Q: What is the best way to learn how an existing driver works?

Read the code top-down. Start with `DRIVER_MODULE` (the Newbus anchor) and work outward: the attach method, then the methods called from attach, then their callees. Take notes as you go: a one-line description of each function in a plain text file is enough. After an hour or two you will have a functional map of the driver. Depth comes from repeating this exercise across several drivers; no single pass will make you an expert.

Q: How do I verify that my build matrix actually covers what I think it covers?

Write a test that fails if a configuration is missing. The simplest form is a shell loop that iterates over the configurations listed in `README.portability.md` and checks that each one has a corresponding entry in the build matrix script. Less rigorously, review the README and the script side by side once a release and confirm they agree. The point is not to be perfect; it is to make the two sources of truth stay aligned over time.

Q: Can I mix clang-specific and gcc-specific code in a portable driver?

Ideally no. FreeBSD's base system uses `clang` by default, and writing portable C that compiles cleanly under both compilers is rarely difficult. When a specific feature is only available under one compiler (for example, a specific sanitizer or a compiler-specific attribute), guard it with `#ifdef __clang__` or `#ifdef __GNUC__` and provide a fallback. But such cases should be rare; most of the time, plain C11 works everywhere.

Q: What do I do when two backends need to share helper code?

Put the helper in a shared file, say `portdrv_shared.c`, and link both backends against it. Alternatively, put the helper in the core if it genuinely belongs there. Avoid the temptation to copy the helper into each backend; copy-paste is the opposite of portability. If you find yourself wanting to duplicate a function, the function wants to be shared, and the only question is where the shared copy lives.

Q: How do I version a backend interface that needs to evolve?

Add a `version` field to the backend struct. The core checks the field at registration time and rejects backends compiled against an older interface. When the interface evolves in a backward-compatible way, bump the minor version; when it evolves in a breaking way, bump the major version and require all backends to be updated. The mechanism is simple and it prevents the class of bug in which a stale backend silently misbehaves because its struct layout has changed.

Q: What makes a driver “good” beyond the patterns in this chapter?

Good drivers are reliable, fast, maintainable, and kind to their users. Reliability comes from careful error handling and thorough testing. Speed comes from a clean data path that avoids unnecessary work and locks. Maintainability comes from the patterns in this chapter. Kindness comes from clear error messages, useful logging, and documentation that respects the reader's time. A driver that does well on all four dimensions is a driver worth shipping; a driver that does well on only two is a driver that will age poorly. Aim for all four.

Q: How do I know when a driver is finished?

It never is, quite. A driver “in production” is a driver under continuous maintenance, because the hardware world keeps changing around it. A reasonable definition of “ready to ship” is: the driver builds cleanly on every supported configuration, passes the tests you have written for it, survives the reliability soak test, and has a README that reflects its current state. Beyond that, every driver is an ongoing conversation between its author, its users, and the platform. The patterns in this chapter are the shape of that conversation; the driver itself is the record of it.

Q: What should I read next, after Chapter 30?

The chapters that follow in Part 7 build on Chapter 29's foundations in specific directions: virtualisation (Chapter 30), advanced interrupt handling (Chapter 31), advanced DMA patterns (Chapter 32), and so on. Read them in order; each one assumes the patterns of this chapter as table stakes. If you want to go wider rather than deeper, read a driver from an area of FreeBSD you have not touched before (storage if you have been doing network, or vice versa), and look for how the patterns of Chapter 29 show up there. Breadth and depth reinforce each other; alternate between them.

29.21 Where the Chapter's Patterns Appear in the Real Tree

A short tour of the FreeBSD source tree, pointing out where you can see the patterns of this chapter applied. Use this as a reading list when you want to practise recognising the structures in production code.

/usr/src/sys/dev/uart/ for backend abstraction. The core in `uart_core.c` is free of bus-specific code. The backends in `uart_bus_pci.c`, `uart_bus_fdt.c`, and `uart_bus_acpi.c` each implement the same interface through different buses. The interface itself lives in `uart_bus.h` as `struct uart_class`. See also `uart_dev_*.c` files for different hardware variants, each a `struct uart_class` instance.

/usr/src/sys/dev/sdhci/ for layered bus and hardware abstraction. The `sdhci.c` core is tall enough that it has its own internal structure, with backends for PCI (`sdhci_pci.c`) and FDT (`sdhci_fdt.c`) plus SoC-specific helpers for specific platforms. The driver illustrates how a mature abstraction accommodates both bus variations and minor hardware variations within a single bus.

/usr/src/sys/dev/e1000/ for layered register accessors. The Intel Gigabit Ethernet driver stacks its register access in four layers (`bus_space_*` primitives, the `E1000_READ_REG/E1000_WRITE_REG` macros in `e1000_osdep.h`, the chip-family helpers in the per-generation files, and the driver-facing API in `if_em.c`), as discussed earlier. Reading it alongside Section 2 is a good way to understand when multiple layers are justified.

/usr/src/sys/dev/ahci/ for split by both bus and platform. AHCI supports PCI-attached storage controllers across several generations and across x86 and non-x86 platforms. The driver uses a tight backend interface and per-platform initialisation helpers.

/usr/src/sys/dev/virtio/ for the paravirtual pattern. The virtio transport is itself a portable abstraction, and the drivers that use it illustrate how a backend can be purely virtual. Each virtio driver (network, block, console, and so on) talks to the virtio transport rather than a physical bus. This is the pattern that Chapter 30 will return to.

/usr/src/sys/net/if_tuntap.c for the simulation pattern applied to a real driver. The `tuntap(4)` driver is not a backend of something else; it is a full driver that synthesises a network interface purely in software. The pattern is useful even in production code.

/usr/src/sys/sys/bus.h for the `bus_read_*` and `bus_write_*` helpers. The header is instructive to read top to bottom, because it makes clear how much architectural abstraction is layered into what looks like a simple function call.

/usr/src/sys/sys/endian.h for the endian helpers. A short header that is worth reading in its entirety at least once, because understanding what `htole32` actually expands to on each platform is the best way to remember why you should use it.

`/usr/src/sys/sys/bus_dma.h` for the DMA API. Another short header, and the starting point for understanding the `bus_dma` abstraction.

`/usr/src/sys/modules/*/Makefile` for real Makefile patterns. Browse at random; almost every subdirectory has one, and each shows a different shape of portability challenge solved by a concrete Makefile.

Reading these files is one of the best ways to internalise the chapter's ideas. Not every file is simple or small, but each one embodies lessons that are hard to convey in the abstract. Treat the tree as a library of examples, not just as a reference.

29.22 A Self-Study Schedule for the Patterns

If you want to take the chapter's lessons further before moving on to Chapter 30, a modest self-study schedule is one way to do it. The suggestions below are sized to fit into a few evenings rather than a week-long course, and each one leaves you with a small, concrete piece of code to keep.

Week 1: The reference driver end to end. Build `portdrv` in every backend configuration. Load, test, and unload each one. Walk through the source with the chapter in your other hand, and annotate the code with the section numbers that introduce each pattern. You will finish the week with a marked-up copy of a working, portable driver and a clear mental map of where each pattern appears in practice.

Week 2: Real-world reading. Pick one driver from the list in the previous section ("Where the Chapter's Patterns Appear in the Real Tree"). `uart(4)` is a good first choice; `sdhci(4)` is a good second. Read the core file and one backend file. For each pattern this chapter introduced, locate the analogous construct in the real driver and write it down. A few pages of notes from this exercise will do more for your skill than reading another chapter cold.

Week 3: A small driver of your own. Write a tiny, original driver from scratch using the chapter's patterns. It can be a pseudo-device, a simulated sensor, or a wrapper around an existing FreeBSD feature exposed as a character device. The specific subject does not matter; what matters is that you use accessors, separate core and backend, structure the Makefile for optional features, and write a `README.portability.md` file. The result will be small and possibly silly, but you will have written a driver with the chapter's patterns from the first line, not retrofitted them.

Week 4: An audit. Find any FreeBSD driver, in-tree or out-of-tree, that looks interesting to you. Apply Challenge 7 to it: go through the audit questions and grade the driver. Write your findings as a short memo, as if to the driver's original author. Do not send the memo; write it for practice. The exercise of formalising what you notice about a driver will sharpen your eye.

After four weeks, you will have built one driver, read two, written a tiny original, and audited one more. That is not a trivial amount of practice, and it is enough to carry the chapter's patterns into your own long-term work. The schedule is a suggestion, not a prescription; adjust to fit the time you have.

One caveat: do not treat this as a race. The patterns land gradually, and the benefit comes from repeated exposure rather than from any single intense session. A slow, steady schedule, an hour or two per evening, produces more durable understanding than a one-day sprint that covers the same ground.

If you finish the four-week schedule and want to keep going, choose one of the later chapters' topics (interrupts, DMA, or `virtio`) and apply the same four-week pattern to it: read one, study two, build one, audit one. The

method generalises to any topic in FreeBSD kernel development, and once you have done it twice it stops feeling like a schedule at all; it just becomes how you learn.

29.23 A Note on Collaboration and Code Review

Portable drivers are easier to review than non-portable drivers. That is not an accident; it is a consequence of the patterns in this chapter. Accessors make every register access reviewable in isolation. Backend splits make it obvious which parts of the driver are bus-independent and which are not. Fixed-width types and endian helpers let a reviewer verify correctness without running the code. A reviewer who can see the structure can focus on the logic, which is what you actually want from the review.

If you are reviewing a driver written in the style of this chapter, there are a few questions worth having ready at hand. Does every register access go through an accessor? Is the backend interface well defined, with clear method contracts? Are all multi-byte values that cross the hardware boundary passed through the endian helpers? Does the Makefile support the build configurations the README claims? Is the module version bumped when the external behaviour changes? A review that answers these questions is a review that adds value; a review that only checks for formatting is a review that wastes time.

If you are on the receiving end of a review, do not take style comments personally. A reviewer pointing out a missed accessor or a bare `volatile` cast is helping you avoid a bug before it becomes one. Take the feedback, make the change, and thank the reviewer. The fastest path to becoming a better driver author is to be reviewed often by authors who are better than you.

29.23.1 Code Review Checklists

If you find yourself reviewing many drivers, a short checklist saves time. The one below is compatible with the patterns in this chapter; print it and keep it next to your review environment.

- **Every register access goes through an accessor.** Grep for `bus_read_`, `bus_write_`, and `volatile` in files that are not the accessor file. If there are hits, flag them.
- **Every multi-byte value that crosses the hardware boundary uses endian helpers.** Look at every struct that represents a descriptor, packet, or message. Verify that reads use `le*toh/be*toh` and writes use `htole*/htobe*`.
- **Fixed-width types are used consistently.** Grep for `int`, `long`, `short`, and `unsigned` in contexts that touch device registers or DMA descriptors. If the code uses a non-fixed-width type, ask whether the size was actually unknown.
- **The backend interface contract is documented.** A reader of the backend header should understand the contract each method fulfils without reading the core source.
- **Conditional compilation blocks have comments.** `#ifdef` blocks without an explanation of why the block exists are a warning sign. Ask the author to add the justification or remove the block.
- **Module version and dependencies are declared.** `MODULE_VERSION` and, where appropriate, `MODULE_DEPEND` are present. The version matches the driver's documented version.
- **The Makefile supports documented build configurations.** The configurations the README mentions actually build when tried. If the README is out of date, say so.
- **Error paths free what they allocated.** Goto-style cleanup, if used, unwinds correctly. Each allocation is balanced by a free.

- **Locking is consistent.** The same data structure is always protected by the same lock, and the core does not call into the backend while holding a lock unless the backend's documentation allows it.
- **Tests exist and pass.** At minimum, a build-matrix script and a smoke test for each backend. Ideally a CI harness that runs them automatically.

A review that covers these items is a review that makes the driver better. Not every review will find issues in every category, but going through the list quickly confirms which categories are clean and which deserve deeper attention.

29.24 A Short Section on Pragmatism

Not every piece of advice in this chapter applies to every driver equally. The chapter has been pitched at the driver that will ship, be maintained for years, run on more than one platform, and be read by more than one author. A throwaway driver written to confirm a hardware quirk does not need any of this. A personal driver written for a single laptop does not need most of it. The patterns are proportional to the expected lifetime and reach of the driver.

How do you decide how much to apply? A few rules of thumb:

- If the driver will only ever be loaded on one machine and will be rewritten when the hardware is replaced, use fixed-width types and endian helpers, and skip the rest. Even a throwaway driver benefits from those two small disciplines.
- If the driver will be shared with colleagues but not shipped, add accessors and a simple Makefile structure. You will not regret it when a colleague asks to build it on their machine.
- If the driver will be shipped to more than one customer, apply everything in Sections 2 through 5. A backend interface, a clean file split, and careful type usage are all in play.
- If the driver will be upstreamed or maintained for five or more years, apply everything in the chapter. This is the driver where the investment pays off most clearly.

Pragmatism means knowing which of these buckets a given driver belongs in and applying the right amount of effort accordingly. Both under-investing and over-investing hurt; the former leaves you with a brittle driver, the latter burns time on abstractions that never pay back. The chapter has shown you what to do when the driver is worth doing well; the judgement about when it is worth doing well is yours.

29.25 Questions to Ask Yourself Before Calling a Driver Done

Before you consider a portable driver complete, walk through the following questions. They are a final pass, a last readiness review for the patterns this chapter introduced. The list reads quickly; going through it honestly takes more time than you expect, because most drivers find at least one item they can improve.

Would a new reader guess the file layout correctly on the first try? If someone who has never seen your driver is told that a bug exists in the PCI attach path, would they open the right file immediately? If not, the file split may be muddled or the names may be unhelpful.

Would the driver compile unchanged on arm64, riscv64, or powerpc64? You do not need to test on every platform, but the answer should be a confident yes for every platform FreeBSD supports. If you hesitate, the reason for the hesitation is something to fix.

Can a user replace the hardware with a simulation and run every user-facing code path? If the simulation backend cannot reach a given path (say, the error recovery code), the test coverage for that path is implicit. Make the simulation rich enough to reach it.

If you disappeared tomorrow, would the next maintainer know why any given `#ifdef` block exists? Not just from the conditional itself, but from a comment above it. If any `#ifdef` is unexplained, it is a hole in the driver's institutional memory.

Does the driver fail loudly when something is wrong, or fail silently? A driver that silently returns zero when it should return `ENODEV` is harder to debug than one that prints a warning. Quiet failure modes are the ones that surprise people in production.

Can you name every lock in the driver and say what it protects? If you cannot produce a one-sentence description of each lock and the data it guards, the locking design is probably less clear than you think. Fix it before you call the driver done.

Does `kldunload` always succeed? Run the driver, push some traffic through it, then try to unload. If it hangs, the driver has a reference it is not releasing. Find the reference before shipping.

Does the driver survive repeated `kldload`/`kldunload` cycles? Run ten loads and unloads in a loop. If memory usage creeps up, you have a leak. If the tenth unload fails where the first succeeded, you have a stale reference.

Does the driver handle being loaded before its dependencies? `MODULE_DEPEND` declarations are the mechanism, but it is worth manually testing that loading your driver from `/boot/loader.conf` (before full userland is up) works as you expect.

Are the error messages specific enough to be useful in a bug report? If a user sends you `portdrv0: attach failed`, can you identify the cause? If not, add context: which operation failed, with what code, on what bus.

Does the Makefile pass a clean build with `-werror`? If it does not, something in your code is generating warnings. Warnings are the compiler telling you about code smells; do not silence the signal.

Have you removed every `printf` added for debugging? Debug prints left in production code slow the driver down and clutter `dmesg`. Use `if_printf`, `device_printf`, or gated debug macros, and remove everything that is not needed.

Going through this list before every release is a habit worth developing. A driver that passes all twelve questions is a driver that is ready to ship; a driver that fails any of them has an issue that is worth addressing before the release goes out.

29.26 Resources for Further Study

The patterns in this chapter are deep enough that a single reading will not exhaust them. The resources below are the ones most likely to repay the effort of following up.

The FreeBSD source tree. No book about FreeBSD drivers can substitute for time spent reading the tree itself. The specific drivers pointed to in this chapter (`uart`, `sdhci`, `ahci`, `e1000`, `virtio`, `tuntap`) are starting points. When you find one you like, read its full commit history. The evolution of a driver often teaches as much as the current snapshot.

The FreeBSD handbook. The handbook is not primarily about driver development, but its chapters on the kernel, on jails, and on bhyve provide context that informs how drivers get deployed. A portable driver cooperates with these features; reading about them is part of writing for them.

sys/sys/bus.h, sys/sys/bus_dma.h, sys/sys/endian.h, sys/sys/systm.h. These headers are short enough to read top to bottom in a single sitting. Each one teaches a different portability concern: `bus.h` the bus abstractions, `bus_dma.h` the DMA abstractions, `endian.h` the endian helpers, `systm.h` the general kernel utilities. Read them; mark them up; come back to them.

The FreeBSD Architecture Handbook. This is the document to read when you want to understand why things are the way they are. It is less hands-on than this book but complements it by explaining the history and reasoning behind key kernel subsystems.

The style(9) manual page. Not glamorous, but the single most important reference for what your code should look like. Read it once thoroughly; skim it again every six months. Familiarity with the in-tree style makes your drivers easier for the community to read and review.

Community forums: `freebsd-hackers@freebsd.org` and `freebsd-arch@freebsd.org`. These lists are the best places to ask questions about non-trivial driver problems. Search the archives before posting; many questions have been asked before. When you do post, follow the community's conventions: clear subject, self-contained description, specific versions and platforms named.

The FreeBSD commit log (git log in the source tree). When you want to understand how a specific feature came to be, trace it through the log. You will find design discussion embedded in the commit messages, particularly for large changes. Read widely: not just commits to the driver you are studying but also the kernel changes that drive new driver requirements.

Your own changes over time. If you apply the patterns in this chapter across several drivers, keep the drivers in a version control system and read through the history occasionally. Your own evolution as a driver author is as instructive as any external reference; looking back at the choices you made six months ago is one of the best ways to see where you are growing.

These resources will sustain you beyond the boundaries of this book. The book is an introduction and a scaffolding; mastery comes from the continuous work of applying the patterns, reading more code, and refining your own taste in engineering. Pace yourself and stay curious.

29.27 Ten Principles to Carry Forward

A chapter this long deserves a closing compression. The ten principles below are what to remember if nothing else from this chapter sticks. Each one is a distillation of a section; together they are the shape of a portable driver.

1. Separate what changes from what does not. The heart of portability is knowing which parts of your driver are likely to vary and isolating them behind an interface. Buses change. Hardware variants change. Architectures change. The driver's core logic should not.

2. Write through accessors, not through primitives. Every register access goes through a named function in your driver. The name describes the operation, not the underlying primitive. One day you will be glad you did.

- 3. Use fixed-width types at hardware boundaries.** `uint32_t` for a 32-bit register field, `uint64_t` for a 64-bit descriptor word. Never `int` or `long` when the size matters.
- 4. Byte-swap at the boundary, and only at the boundary.** Every multi-byte value that crosses from CPU to hardware or vice versa passes through an endian helper. The rest of the code is blissfully unaware of byte order.
- 5. Express variation as data, not as conditionals.** A table of backends, a struct of function pointers, a registry indexed by device ID. These are easier to read, test, and extend than sprawling `if-else` chains.
- 6. Split files by responsibility, not by size.** The core in one file, each backend in its own file, each large subsystem concern in its own file. File boundaries communicate design intent.
- 7. Keep conditional compilation coarse and commented.** A `#ifdef` block with a clear comment is a feature toggle. A sea of nested `#ifdefs` with no comments is a maintenance crisis.
- 8. Version everything that can change.** `MODULE_VERSION`, backend interface version fields, `README.portability.md`. When the driver's world changes, its version numbers make the change legible.
- 9. Test with simulation, build on a matrix.** A simulation backend exercises your core without hardware. A build matrix catches compile-time portability regressions before they reach users. Both are force multipliers.
- 10. Document the invariants, not the syntax.** Comments explain why a choice was made, not what the code does. Code says what; comments say why. The combination is what makes a driver outlive its author.

These ten principles are the chapter. Everything else is elaboration, example, and exercise. Commit them to memory, apply them consistently, and the patterns will feel natural within a year. That is the payoff.

One more note on the principles. They are not a ranking; they are a set. Leaving any one out weakens the others. A driver with perfect endian handling but no file split is still hard to maintain. A driver with a clean file split but no `MODULE_VERSION` is still hard to upgrade. A driver with flawless abstractions but no tests is still fragile under change. The patterns reinforce each other, and missing one undermines the value of the rest. If you must pick a subset (say, because the driver is small or throwaway), pick based on the lifetime and reach you expect, not on which pattern is cheapest today.

The last encouragement is the simplest: start somewhere. Take one driver, one weekend, one pattern from the list, and apply it. Look at what changed. Notice that the driver got better, even if only slightly. Then do it again, the following weekend, with another pattern. After a month of this, the driver will be in better shape than you can remember it, and your sense for what the patterns feel like in practice will be solid. The path to mastery is not dramatic; it is just the patient accumulation of small improvements over time. You now have the map.

29.28 Wrapping Up

Portability is easier to internalise when you can see it on the page, so let us close the chapter with a small, concrete example of what a cross-release change looks like in the real FreeBSD tree. Open `/usr/src/sys/dev/rtsx/rtsx.c`, find `rtsx_set_tran_settings()`, and look at the short block guarded by `#if __FreeBSD_version >= 1300000`. Inside that guard, the driver handles the `MMC_VCCQ` field, which carries the I/O voltage for an MMC card and was added in the FreeBSD 13 series. On older trees the field does not exist; on 14.x it does; and the guard is the only thing that lets a single source file compile cleanly on both. That is portability expressed as a short conditional rather than as two forks of the same driver, and it is exactly the discipline this chapter has been asking you to adopt.

The lesson of that small block is larger than the block itself. Every change between releases, whether it is a new field in a structure, a renamed function, a deprecated macro, or a reorganised header, can be absorbed by the same pattern you just saw: name the variation, isolate it behind a guard or an accessor, and leave a comment that explains why the block is there. The driver keeps working on old and new trees alike, and your energy goes into the logic rather than into maintaining parallel copies of the same file for every release you care about.

A similar story, at a much larger scale, is on display in the Intel Ethernet driver at `/usr/src/sys/dev/e1000/if_em.c`. There, the attach and detach path now goes through the `iflib` framework: `em_if_attach_pre(if_ctx, ctx)` replaces the bespoke `device_t` attach that this driver once implemented directly, and the `DEVMETHOD` table near the top of the file dispatches `device_probe`, `device_attach`, and `device_detach` through `iflib_device_probe`, `iflib_device_attach`, and `iflib_device_detach`. You are not looking at a small conditional there; you are looking at the result of moving a whole family of NIC drivers onto a shared abstraction so that the per-chip code and the `iflib`-common code can evolve on different schedules. It is the same instinct as the `rtss` example, applied at a bigger scale.

If you take one thing from this chapter, take that instinct. Portability is not a property you add at the end of the project, and it is not a special kind of code you write once and forget about. It is a way of sitting down at a source file and asking: what might change, where will it change, and how do I make sure the change lands in one place rather than twenty? Answer that question consistently, and the drivers you write in the coming years will outlive several hardware generations, several kernel releases, and probably several jobs.

29.29 Looking Ahead: Bridge to Chapter 30

You have just refactored a driver for portability. The next chapter, **Virtualisation and Containerization**, turns from the driver's own internals to the environment it runs in. FreeBSD drivers run not only on bare metal but inside `bhyve` guests, under `jail(8)` with `VNET`, on `virtio`-backed storage and network, and increasingly inside OCI-style container runtimes built on top of jails.

That question is sharper after Chapter 29 than it was before. You have now learned to write drivers that absorb variations in hardware, bus, and architecture as local changes. Chapter 30 will add a new kind of variation: the environment itself is virtualised. The machine your driver sees may not be a real machine; the devices it probes may be paravirtual; the host may have removed whole classes of capability because the guest is not trusted to use them. You will learn how `virtio` guest drivers work, how `bhyve`'s `vmm(4)` and `pci_passthru(4)` fit together, how `VNET` jails isolate network stacks, and how to write drivers that cooperate with those forms of containment without assuming the usual privileges.

You will not be writing a new kind of driver in Chapter 30. You will be learning how the drivers you already know how to write must adapt when the environment around them is virtual, partitioned, or containerised. That is a different kind of step, and it matters the moment you deploy a FreeBSD system in a cloud or in a multi-tenant host.

Before you move on, unload every module you created in this chapter, clean every build directory, and make sure the driver source tree is in a tidy state. Close your lab logbook with a brief note on what worked, what surprised you, and which pattern you expect to reach for most often in your own drivers. Rest your eyes for a minute. Then, when you are ready, turn the page.

29.30 A Final Word: The Long Arc

It is worth stepping back one more time, because chapters about craft are easy to read and hard to apply. The patterns in this book are not a checklist to tick off on a single project; they are a long-running investment in your own capability. The first driver you write using them will feel like a lot of work for modest gain. The fifth driver will feel natural. The tenth driver will be faster and more reliable than any driver you could have written without the patterns, and you will no longer remember that you once found them effortful.

That arc is not unique to FreeBSD driver development. It is the arc of every skilled practice, from carpentry to music to compiler writing. The craft starts as a set of disciplines that feel heavy and deliberate; it becomes, after enough practice, the shape of your attention. When you sit down to write your next driver and find that you have already thought about the backend structure before typing the first line, that is the craft at work.

The chapter ends here, but the practice does not. Keep building. Keep reading. Keep refactoring drivers that are not yet as portable as they could be. The payoff is measurable only over years, which is the exact timescale that FreeBSD drivers tend to live on. You have picked a long game; the patterns in this chapter are the rules that let you play it well.

You have earned the step.

30 Virtualisation and Containerization

30.1 Introduction

You arrive at Chapter 30 with a new habit of mind. Chapter 29 taught you how to shape a driver so that it absorbs variation in hardware, bus, and architecture without collapsing into a maze of conditionals. You have learned to push the parts that change into small, backend-specific files and to let the core remain clean. You have met the idea that a driver is not written for a single machine; it is written for a family of machines that share a programming model. That lesson carries a great deal of weight in this chapter, because the environment around your driver can now change in a way that is more radical than anything Chapter 29 considered. The machine itself may not be real.

This chapter is about what happens when the hardware under your driver is not a physical card plugged into a physical slot, but a software simulation presented to a guest operating system by a hypervisor; or when the driver is asked to attach inside a jail that sees only part of the device tree; or when the network stack it cooperates with is one of several stacks running side by side inside a single kernel. Each of these situations is a departure from the “one kernel, one machine, one device tree” mental model that the earlier chapters built. Each one changes what your driver may assume, what it may do, and what a user of your driver may safely expect from it.

The word “virtualisation” can mean several different things depending on who is speaking. To a system administrator running a cloud fleet, it means virtual machines with full kernels of their own. To a FreeBSD user, it often means `jail(8)` and its descendants, which isolate processes and filesystems without giving each one a kernel. To a driver author, it means both, and more. It means paravirtual devices like `VirtIO` that are explicitly designed to be easy for a guest to drive. It means emulated devices that a hypervisor presents to a guest as if they were physical hardware. It means passthrough devices where a real piece of hardware is handed to a guest more or less directly, with the hypervisor stepping aside as much as it safely can. It means jails whose view of `devfs` is trimmed by a ruleset so that the containerised processes inside cannot see every device the host can see. It means `VNET` jails that own an entire network stack of their own, complete with an `ifnet` that was loaned to them by the host.

If that list already feels like a lot, take a breath. The chapter will introduce each of these pieces one at a time, with enough real FreeBSD grounding that you will be able to close the book and open the source tree and find the relevant files with your own hands. Nothing in this chapter is impossible to learn; it is a set of distinct but related ideas that share a common thread. The thread is this: a driver stops being the single most important piece of software in its particular hardware path. Above it sits a hypervisor, a jail framework, or a container runtime; beside it sit other guests or jails that share the host; below it sits a piece of silicon that the driver no longer owns exclusively. Writing drivers for that world is not harder than writing drivers for bare metal, but it is different in ways that are easy to miss if you have not been shown them.

Two distinct directions run through the chapter. The first direction is about **writing guest drivers**: code that runs inside a virtual machine and talks to devices the hypervisor presents. You will spend the bulk of your time in this direction, because it is where most of the novel programming happens. `VirtIO` is the canonical example,

and we will return to it often. The second direction is about **cooperating with FreeBSD's own virtualisation infrastructure** from the host side: understanding how your driver attaches inside a jail, how it behaves when the host moves an interface into a VNET jail, how it handles a user inside a jail trying to call an `ioctl` that only root on the host should be allowed to call, and how to test it all without ruining a running host. This direction is less about exotic new APIs and more about discipline, privilege boundaries, and a careful mental model of what is visible from where.

The chapter does not try to teach you how to write a hypervisor, how to implement a new VirtIO transport, or how to build a container runtime from scratch. Those are large topics with their own books. What it teaches is how a driver author should think about, prepare for, and work with virtualised and containerised environments, so that the drivers you write make sense everywhere they are loaded. By the end of the chapter, you will recognise a VirtIO guest driver when you see one, you will know how to detect whether your driver is running in a virtual machine, you will be able to explain what `devfs_ruleset(8)` does and why it matters for device exposure, you will be able to reason about the differences between VNET jails and ordinary jails for a network driver, and you will have written a tiny VirtIO guest driver of your own against a FreeBSD `bhyve(8)` backend.

Before we begin, a word about the tone of what follows. Some of the subjects in this chapter, especially hypervisor internals and jail security boundaries, have acquired a reputation for being exotic. A driver author who has never looked under the hood can feel intimidated by their jargon. You should not. The code is FreeBSD code; the APIs are FreeBSD APIs; the mindset is the same mindset you have been building since Chapter 1. We will move slowly, and we will keep coming back to real files you can open. Let us begin.

30.2 Reader Guidance: How to Use This Chapter

This chapter sits in a different place in the learning progression from Chapter 29. Chapter 29 was about how your driver is organised on disk. Chapter 30 is about what the world around your driver looks like at runtime. That difference matters for how you should read and practise. The patterns of Chapter 29 can be absorbed by reading carefully and typing along. The patterns of this chapter land more firmly if you also boot a virtual machine, create a jail, and watch the driver behave inside them. Plan accordingly.

If you choose the **reading-only path**, plan for roughly two to three focused hours. At the end you will understand the conceptual map: what paravirtual, emulated, and passthrough devices are; how VirtIO uses shared rings and feature negotiation; how jails and VNET isolate devices and network stacks; what `rctl(8)` controls from the driver's point of view. You will not yet have the reflexes of a driver author who has debugged a `virtqueue` mismatch at three in the morning, and that is fine. The reading pass is a legitimate first encounter, and the material has enough depth that a second pass with labs later on will extract much more value from it.

If you choose the **reading-plus-labs path**, plan for six to ten hours spread across two or three sessions. You will install `bhyve(8)` on your lab machine, boot a FreeBSD 14.3 guest on top of it with VirtIO devices, write a small VirtIO pseudo-device driver called `vtedu`, and observe it attach, service requests from the host-side device, and detach cleanly. You will also create a simple jail, attach the driver to it under different `devfs` rulesets, and watch what happens when the driver exposes a device inside the jail. The labs are structured so that each one stands on its own, leaves you with a working system, and reinforces a specific concept from the main text.

If you choose the **reading-plus-labs-plus-challenges path**, plan for a long weekend or a handful of evenings. The challenges push the baseline labs into more realistic territory: extending `vtedu` to accept multiple `virtqueues`, writing a small tool that probes the guest's hypervisor through `vm.guest` and adapts behaviour, building a VNET

jail and moving a tap interface into it, and writing a short report on how a driver that exports an `ioctl` surface should decide which `ioctls` should be accessible from inside a jail. Each challenge is invitational rather than mandatory, and each one is sized to be completable without a second weekend.

A note on the lab environment. You will continue to use the throwaway FreeBSD 14.3 machine you established in earlier chapters. That machine will act as the **host** in this chapter. On top of it you will run `bhyve(8)` guests, which are smaller FreeBSD installations managed by your host. You will also create a few jails directly on the host. This nesting sounds complicated the first time it is described, but it is genuinely simple in practice: your host runs FreeBSD, inside the host you start a `bhyve` virtual machine that also runs FreeBSD, and inside either the host or the guest you create jails. Each layer is independent, each layer has its own `dmesg`, and each layer is cheap to recreate from scratch if something goes wrong.

Make a snapshot of the host before you start. The chapter will ask you to change `/etc/devfs.rules`, to load and unload kernel modules, and to create small VMs. None of these are risky if handled carefully, but accidents happen, and the snapshot turns any mistake into a two-minute rollback. If your host is itself a VM in VirtualBox or VMware, the platform's own snapshot tool is the fastest way to do this. If your host is bare metal, a ZFS boot environment with `bectl(8)` is a good substitute.

One more note. This chapter's labs need a few packages that you may not already have installed. You will want `bhyve-firmware` for UEFI guest support, `vm-bhyve` for a more convenient front end to `bhyve` and its accessories, and `jq` for parsing JSON output during some of the more elaborate tests. The commands to install them are in the labs; do not install them now, but be aware that they are coming.

30.2.1 Prerequisites

You should be comfortable with everything from earlier chapters. In particular, this chapter assumes that you already know how to write a loadable kernel module from scratch, how `probe()` and `attach()` fit together in the driver lifecycle, how `softc` is allocated and used, and how `device_t` relates to `devclass`. It assumes fluency with the `bus_read_*` and `bus_write_*` accessors from Chapter 15, basic familiarity with interrupt handlers from Chapter 18, and the portable-driver habits from Chapter 29. If any of that feels uncertain, a brief revisit to the earlier material will save you time here.

You should also be comfortable with ordinary FreeBSD system administration: reading `dmesg`, editing `/etc/rc.conf`, using `sysctl(8)`, and creating and destroying jails with `jail(8)` or its wrapper service `jail`. You do not need prior `bhyve` experience; the labs will walk you through it. You do not need prior container experience either, since FreeBSD's container story is ultimately jails dressed in different operational clothing.

30.2.2 What This Chapter Does Not Cover

A responsible chapter tells you what it leaves out. This chapter does not teach the internals of the `bhyve(8)` hypervisor. It does not teach `libvirt(3)`, `qemu(1)`, or the Linux KVM subsystem. It does not turn you into an OCI container expert. It does not cover the `xen(4)` paravirtualisation code, since Xen has become a narrower niche in the FreeBSD ecosystem and `bhyve` is the FreeBSD-native story worth learning first. It mentions `jail(8)` at exactly the level a driver author needs, not at the depth a jail administrator would want. For the missing topics, the FreeBSD Handbook and the respective manual pages are your friends, and I will point to them when they are relevant.

Several topics that could plausibly appear in a virtualisation chapter have homes elsewhere in this book. Secure coding against hostile input is Chapter 31's subject, not this chapter's; you will meet privilege boundaries and jail visibility here, but the deeper security discipline (Capsicum, MAC framework, sanitiser-driven fuzzing) waits for the next chapter. Advanced DMA under virtualisation (IOMMU setup, mapping pages with `bus_dmamap`) is touched conceptually here and developed more fully in later chapters. Performance tuning under virtualisation (how to measure paravirtual overhead, how to decide between emulated and passthrough for a given workload) is a theme that returns in Chapter 35. We will mention these where they connect, but we will not go deep.

30.2.3 Structure and Pacing

Section 1 establishes the mental model: what virtualisation and containerization mean for a driver author, and how they differ from “hardware, but slower.” Section 2 explains the three styles of guest device (emulated, paravirtualised, passthrough) and what each one implies for the driver. Section 3 takes a careful, beginner-friendly tour of VirtIO: the shared-ring model, feature negotiation, and the `virtqueue(9)` APIs you will use. Section 4 teaches how a driver detects its runtime environment through `vm_guest` and friends, and when that detection is and is not a good idea. Section 5 turns to the host side and looks at `bhyve(8)`, `vmm(4)`, and PCI passthrough. Section 6 covers jails, `devfs`, and VNET: the FreeBSD containerisation story from the driver's perspective. Section 7 addresses resource limits and privilege boundaries. Section 8 talks about testing and refactoring. Section 9 revisits time, memory, and interrupt handling through the lens of virtualisation, the quieter topics where beginner drivers often fail in subtle ways. Section 10 widens the lens to FreeBSD on `arm64` and `riscv64`, whose virtualisation stories have their own shape. The labs and challenges follow, along with a troubleshooting appendix and a closing bridge into Chapter 31.

Read the sections in order. Each one assumes the previous one, and the labs depend on the earlier sections having been read and internalised.

30.2.4 Work Section by Section

A recurring pattern in this book is that each section does one thing. Do not try to read two sections at a time, and do not skip ahead to a “more interesting” section if something in the current one feels difficult. The interesting parts of this chapter rest on the foundational ones, and a reader who has skipped the foundation will spend more time reverse-engineering the labs than the careful reader spends on the whole chapter.

30.2.5 Keep the Reference Driver Close

Several labs in this chapter build on a small pedagogical driver called `vtedu`. You will find it under `examples/part-07/ch30-virtualisation/`, organised the same way as earlier chapters' examples. Each lab directory contains the state of the driver at that step, along with `Makefile`, `README`, and supporting scripts. Clone the directory, type along, and load the module after each change. Refactoring a VirtIO guest driver in your head is harder than refactoring one on disk; the feedback from the build system and from `dmesg` is half the lesson.

30.2.6 Open the FreeBSD Source Tree

Several sections will point to real FreeBSD files. The ones that repay careful reading in this chapter are `/usr/src/sys/dev/virtio/random/virtio_random.c` (the smallest complete VirtIO driver in the tree), `/usr/src/sys/dev/virtio/virtio.h` and `/usr/src/sys/dev/virtio/virtqueue.h` (the public API surfaces), `/usr/src/sys/dev/virtio/virtqueue.c` (the ring machinery), `/usr/src/sys/dev/virtio/p-ci/virtio_pci.c` and `/usr/src/sys/dev/virtio/mmio/virtio_mmio.c` (the two transport backends), `/usr/src/sys/sys/system.h` (for `vm_guest`), `/usr/src/sys/kern/subr_param.c` (for the `vm.guest sysctl`), `/usr/src/sys/net/vnet.h` (for VNET primitives), and `/usr/src/sys/kern/kern_jail.c` (for prison and jail APIs). Open them when the text says to, and read around the exact spot the text points at. The files are not decoration; they are the source of truth.

30.2.7 Keep a Lab Logbook

Continue the lab logbook from earlier chapters. For this chapter, log a short note for each major lab: which commands you ran, which modules loaded, what `dmesg` said, what surprised you. Virtualisation work is easy to forget the details of after a few days, and a paper trail turns every future debugging session into a one-minute lookup rather than a half-hour re-derivation.

30.2.8 Pace Yourself

Several ideas in this chapter will feel new the first time you meet them: shared-ring memory, feature negotiation, VNET's thread-local current `vnet`, the `devfs` ruleset number system. That newness is normal. If you feel your understanding blur during a particular subsection, stop. Re-read the previous paragraph, try a small experiment in the shell, and come back fresh in an hour. Consistent thirty-minute sessions produce better understanding than a single exhausted all-day effort.

30.3 How to Get the Most Out of This Chapter

Chapter 30 rewards curiosity, patience, and a willingness to experiment. The specific patterns it introduces, the ring structures, the feature bits, the prison and `vnet` scopes, are not abstract. Each one corresponds to code you can read, state you can observe with `sysctl` or `vmstat`, and behaviour you can trigger with short commands. The most valuable habit you can build while reading the chapter is to move freely between the text, the source tree, and the running system.

30.3.1 Read With the Source Tree Open

Do not read the VirtIO section without `virtio.h`, `virtqueue.h`, and `virtio_random.c` open in another window. When the text says that a driver calls `virtqueue_enqueue` to hand a buffer to the host, scroll to that function and see it in context. When the text mentions the `VIRTIO_DRIVER_MODULE` macro, open `virtio.h` and see the two `DRIVER_MODULE` lines it expands into. The FreeBSD kernel is far more approachable than its reputation suggests, and the only way to confirm that is to keep opening it.

30.3.2 Type the Labs

Every line of code in the labs is there to teach something. Typing it yourself slows you down enough to notice the structure. Copy-pasting the code often feels productive and usually is not; the finger-memory of typing kernel code is part of how you learn it. If you must copy, copy one function at a time, and read each line as you paste it.

30.3.3 Run What You Read

When the text introduces a command, run it. When the text introduces a `sysctl` name, query it. When the text introduces a kernel module, load it. The running system will surprise you sometimes (the `vm.guest` value may not be what you expect if your lab itself is a VM), and every surprise is an opportunity to learn something the text did not need to explain explicitly. A running FreeBSD system is a patient tutor.

30.3.4 Treat `dmesg` as Part of the Manuscript

A significant fraction of what this chapter teaches is visible only in the kernel's log output. A VirtIO device's negotiated feature set, the attach sequence of a guest driver, the interaction between a module and a jail, all of these surface in `dmesg`. Read it often. Tail it during the labs. Copy relevant lines into your logbook when they teach something non-obvious. Do not treat `dmesg` as noise; treat it as the ground truth of what the kernel actually did.

30.3.5 Break Things Deliberately

At three points in the chapter I will suggest deliberately breaking something to see what happens. These are the most educational moments you can give yourself. Unload a required module before a driver that depends on it and see the dependency graph complain. Remove a `devfs` rule and see the device vanish from inside the jail. Boot a guest with one CPU and then with four and compare the `virtqueue` setup. Deliberate failures teach in a way that success cannot.

30.3.6 Work in Pairs When You Can

If you have a study partner, this is a great chapter to pair on. One of you can run the host and watch `dmesg`, while the other runs the guest and drives the driver. The two perspectives teach different things, and each of you will notice what the other missed. If you are working alone, use two terminal tabs and alternate attention between them.

30.3.7 Trust the Iteration, Not the Memorisation

You will not remember every flag, every enum, every macro in this chapter on the first read. That is fine. What matters is that you remember where to look, what the overall shape of the subject is, and how to tell when your driver is doing the right thing. The specific identifiers will become second nature after you have written and debugged two or three virtualisation-aware drivers of your own; they are not a memorisation exercise.

30.3.8 Take Breaks

Virtualisation-aware debugging has a particular cognitive cost. You are tracking state on several sides at once: the host, the guest, the driver, the device model, the jail. Your mind gets fatigued faster than when you are working on a single-threaded bare-metal driver. Two hours of focused work followed by a real break is almost always more productive than four hours of grinding.

With those habits in place, let us begin.

30.4 Section 1: What Virtualisation and Containerization Mean for Driver Authors

Before we touch any code, we need to agree on what we are talking about. The words “virtualisation” and “containerization” have been worn smooth by marketing language, and they carry different meanings depending on where you encounter them. A vendor’s white paper uses “virtualisation” as a synonym for running multiple operating systems on the same hardware; a cloud provider’s documentation uses it as shorthand for any managed workload; a FreeBSD administrator uses it for anything from `bhyve(8)` to `jail(8)`. For a driver author, those meanings are not interchangeable. The shape of the problem you are solving changes depending on which kind of virtualisation your driver meets. This section anchors the vocabulary for the rest of the chapter.

30.4.1 Two Families, Not One

The first distinction to draw is between **virtual machines** and **containers**. They solve different problems and produce different constraints for driver authors. Conflating them is the single most common confusion in this territory.

A **virtual machine** is a software-emulated computer. A hypervisor, which itself runs on real hardware, creates an execution environment that looks like a complete machine to the software inside it. The machine has a BIOS or UEFI, memory, CPUs, disks, and network cards. The operating system inside the VM boots from scratch just as it would on real hardware, loads a kernel, and attaches drivers. The observation that matters is that the **guest kernel is a full kernel**, independent of the host kernel. A FreeBSD guest running under a FreeBSD host is not sharing kernel code with the host; the two kernels are peers, separated by the hypervisor and by the barrier between guest memory and host memory.

A **container**, on the FreeBSD sense of the word, is a different animal. A container is a namespaced partition of a single kernel. The host kernel is the only kernel in the picture; the containers are separate process groups, separate filesystem views, and often separate network stacks, but all of them run on the same kernel with the same drivers. On FreeBSD, the classical container is a `jail(8)`. Modern container runtimes like `ocijail` and `pot` add scripting and orchestration around that same jail primitive. The observation here is that **there is only one kernel**. A driver in the host is visible to every jail, subject to the rules the host imposes; a jail cannot load a driver of its own, and it does not have a kernel into which to load one.

These two families overlap only in superficial ways. A VM and a jail both look like “another system” from the outside. A VM and a jail both have their own `/`, their own network, and their own users. But the kernel boundary is nowhere near the same, and that matters enormously to drivers.

A VM runs a full guest kernel; the kernel sees virtualised (or passthrough) hardware; the driver you write for that hardware is a **guest driver** that attaches inside the guest. A jail shares the host's kernel; no guest driver is being written, because there is no guest kernel to load it into; instead, the question is how a **host driver** exposes its services to processes running inside the jail. The techniques you use in each case are different.

For the rest of this chapter, when the text says “virtualisation,” it usually means the VM case unless the context makes clear that jails are being discussed. When the text says “containerization,” it usually means the jail case. Read those words with the families in mind, and you will avoid most of the confusion that newcomers have.

30.4.2 Virtual Machines as an Environment for Drivers

A VM presents hardware to the guest. Exactly what that hardware looks like depends on the hypervisor's configuration. There are three broad styles of device that a guest may meet, each of which changes the driver problem.

Emulated devices are the oldest approach. The hypervisor presents to the guest a device that imitates a real, usually well-known, physical device: a ne2000 Ethernet card, for example, or a PIIX IDE controller, or a serial port compatible with the venerable 16550 UART. The hypervisor's device model implements the same register interface the real hardware would expose. The guest loads the driver it would have loaded on real hardware, and the driver runs unchanged. The price is performance: every register access in the guest traps into the hypervisor, which emulates the behaviour in software, which is much slower than real hardware. Emulated devices are perfect for compatibility and for booting unmodified guest kernels; they are poor for serious workloads.

Paravirtual devices solve the performance problem by replacing the compatibility interface with one designed to be cheap for both sides. Instead of imitating physical hardware, the device defines a new interface that is efficient when implemented in software. VirtIO is the canonical example and the one this chapter will spend most of its time on. A VirtIO device does not look like any real hardware; it looks like a VirtIO device, which is a standardised interface that exposes shared memory rings and a few control registers and very little else. The guest must have a driver for VirtIO specifically; that driver is smaller and faster than the equivalent emulated-device driver, because the interface was designed for software on both sides.

Passthrough devices go in the other direction. The hypervisor steps aside and hands the guest a real piece of hardware: a physical NIC, a physical GPU, a physical NVMe SSD. The guest's driver talks to that hardware more or less directly. The hypervisor still mediates (through an IOMMU, for example, to constrain which memory the device can reach) but it no longer emulates or paravirtualises. Passthrough is fast but brittle: the device is no longer shared with other guests, and moving the VM to a different host is no longer trivial.

Each style of device imposes a different design on your driver. An emulated-device driver is typically an old, well-tested driver for the emulated hardware, which just happens to be running on a virtualised version of the same hardware. A paravirtual driver is usually a driver written specifically for the paravirtual interface, with no expectation that the “hardware” exists in real silicon. A passthrough driver is the same driver that would have run on bare metal, with one subtlety: the guest's memory is not the same memory the device DMAs into, and the IOMMU mapping must be correct for the device to work.

The rest of this chapter leans strongly into the second style, because paravirtualisation is where most of the novel driver work in virtualised environments happens, and because it illustrates a programming model (shared rings, feature bits, descriptor-based transactions) that generalises to other parts of FreeBSD.

30.4.3 Containers as an Environment for Drivers

A jail is a different kind of environment, and it asks a different question of a driver. The driver is not running inside the jail; it is running in the single shared kernel. What the jail does is change what the processes in the jail see.

From the host's kernel, nothing changes when a jail is created. The host kernel still has all the drivers it had before; the drivers still attach to the same devices; the same `/dev/` entries still exist in the devfs mounted at the host's root. From inside the jail, a devfs mount is also performed, but that devfs is configured with a ruleset that hides some devices and exposes others. A `/dev/mem` is not normally visible inside a jail, because a process inside a jail should not be able to peek at kernel memory; a `/dev/kvm` equivalent (if FreeBSD had one) would be similarly hidden. A `/dev/null` and `/dev/zero` are visible, because there is no reason to hide them.

So what is a “driver problem” in a container? Two things, mostly. First, when your driver creates device nodes, you must decide whether those nodes should be visible to jailed processes, and if so, under what conditions. If your driver exposes an `ioctl` that allows a process to change the routing table, that `ioctl` should not be accessible to a process inside a jail that does not have the privilege to reconfigure the host's network. Second, when your driver cooperates with VNET (the FreeBSD virtual network stack framework), you must be careful about which global state is per-vnet and which is shared. An `ifnet` moved into a VNET jail should behave correctly inside that jail, which means your driver must have declared the right VNET-scoped variables and must not have cached any cross-vnet state.

These are different concerns from those of VM drivers, and they deserve their own sections. Sections 6 and 7 of this chapter work through them in detail.

30.4.4 Why This Matters for Driver Authors

It is fair to ask: why should a driver author care about any of this? Until recently, most drivers were written for physical hardware on bare metal, and the “virtualisation story” was something that happened upstream. Today, and for the foreseeable future, the majority of FreeBSD installations are virtual machines. A driver that works only on bare metal is a driver that works in a minority of deployments. A driver that works only on the host, and not inside a jail, is a driver that cannot be used in any modern containerised FreeBSD environment. Virtualisation-aware driver design is not optional anymore; it is part of writing drivers at all.

There are three concrete reasons the topic matters.

First, **most of your users are inside a guest**. The FreeBSD VMs running in public clouds, in private clouds, and in local bhyve labs outnumber the FreeBSD installations on real hardware, perhaps by a wide margin. A driver whose design fails in a virtual environment will fail for most of its users.

Second, **host-side drivers expose services to jails**. Even if your driver is about a physical device that lives in the host, the moment a jail wants to use its services, the driver faces the jail question. It may need to expose a device node through the jail's devfs, or decide that certain `ioctls` are jailed-off.

Third, **performance at scale matters more than ever**. In a virtualised environment, the difference between an emulated driver and a paravirtual one is often a factor of ten or more in throughput. Knowing how to write and recognise paravirtual patterns is worth real engineering time. A few extra paragraphs of understanding about `virtqueue(9)` can save you a day of debugging later.

30.4.5 Virtualisation Is Not Just Hardware But Slower

A common misconception is that virtualisation simply makes hardware slower. That view misses the mechanism entirely. In a well-designed virtualised environment, the guest driver does not see “slower hardware”; it sees different hardware with different access patterns. A VirtIO driver does not make one slow register read per descriptor and then complain about the overhead; it batches a dozen descriptors into a shared ring, notifies the host once, and waits for a single completion. The difference between a naive port of a physical-hardware driver to VirtIO and an idiomatic VirtIO driver is not a matter of tuning; it is a matter of architectural intent.

Similarly, containerization is not “processes, but confined.” It is a reorganisation of visibility, privilege, and global state. A driver that exports a sysctl tree has to decide whether that tree should be visible inside a jail, and whether writes from inside the jail should affect only the jail’s view or the host’s. These are not just configuration questions; they are design questions that shape the code.

If you take one thing from this section, let it be this: **the environment your driver runs in is not a transparent wrapper around a machine.** It imposes its own concerns, offers its own APIs, and rewards or punishes driver designs that respect or ignore them.

30.4.6 A Quick Taxonomy

To fix the vocabulary in your mind, here is a compact taxonomy of the environments we will discuss in this chapter. Treat it as a map you can return to when a later section mentions a term.

Environment	Kernel Boundary	What the Driver Sees	Typical FreeBSD Example
Bare metal	Full	Real hardware	Any driver
Emulated VM device	Full, at guest kernel	Imitation of real hardware	xn, em emulated by QEMU
Paravirtual VM device	Full, at guest kernel	Shared-ring interface	virtio_blk, if_vtnet
Passthrough VM device	Full, at guest kernel	Real hardware with IOMMU constraints	em on a passthrough NIC
Jail (no VNET)	Shared with host	Host-side driver; jail sees host’s devfs filtered by ruleset	devfs visibility for /dev/null
VNET jail	Shared with host	Host-side driver; jail owns its network stack; interfaces moved via if_vmove()	vnet.jail=1 in jail.conf

Notice the kernel-boundary column. The boundary is the single most important feature of each row. VMs have a boundary; jails do not. Everything else follows from that.

30.4.7 Wrapping Up

We have drawn the first map of the chapter. Virtual machines and containers are two different families. VMs present hardware (emulated, paravirtual, or passthrough) to a full guest kernel. Containers partition a single host kernel into isolated process environments, changing what they see and what they may do, but not introducing a second kernel. A driver author deals with each family differently: writing guest drivers for VMs, adapting host drivers for jails. The next section takes the first of those two worlds and looks at the three device styles a guest may meet, in enough detail to build intuition before we touch VirtIO specifically.

30.5 Section 2: Guest Drivers, Emulated Devices, and Paravirtualised Devices

A guest kernel running inside a VM must attach drivers to the devices its hypervisor has given it. Those devices fall into the three styles introduced in Section 1: emulated, paravirtual, and passthrough. Each style produces a different kind of driver, a different kind of bug, and a different kind of optimisation opportunity. This section examines them in the order you are most likely to encounter them, and begins to build the vocabulary we will need in Section 3 for VirtIO specifically.

30.5.1 Emulated Devices in Detail

An emulated device is the simplest story to tell. The hypervisor implements, in software, a faithful imitation of a known physical device. The classical FreeBSD examples include the emulated `em(4)` Ethernet NIC (the Intel 8254x family), the emulated `ahci(4)` SATA controller, the emulated 16550-compatible serial port, and the emulated VGA video adapter.

From the guest's point of view, nothing unusual is happening. The PCI device enumerator finds an Intel card; the `em(4)` driver probes it and attaches; the driver issues its usual register writes; packets eventually appear on the wire. Under the hood, each of those register writes is a trap into the hypervisor, which consults its model of the Intel chip, applies the state change, and possibly generates a virtual interrupt back into the guest.

This approach has one huge virtue and one large vice. The virtue is compatibility: the guest kernel needs no special driver at all. A stock FreeBSD install image boots on an emulated `em(4)` card as easily as it boots on a real one. The vice is cost. Every trap into the hypervisor, every software emulation of a register behaviour, every synthesised interrupt, costs CPU cycles. For a busy workload with millions of register accesses per second, emulated devices are measurably slow.

30.5.2 What Emulated Devices Imply for Driver Code

From the driver author's perspective, an emulated device looks like real hardware. The implication is that you do not normally write a "special" driver for emulated hardware; you write one driver that works on the real silicon and lean on emulation for the compatibility case. Most of the emulated-device drivers in `/usr/src/sys/dev/` are therefore the same files that drive the real cards.

There are two small exceptions that matter for our purposes. First, some drivers include a short early-boot probe that distinguishes "this is a real card" from "this is a hypervisor imitating a card." The `em(4)` driver, for example, logs a slightly different message when it detects that the environment is virtualised, because some diagnostic counters are meaningful only on real silicon. Second, a few drivers bypass hardware-specific optimisations

in virtualised environments because those optimisations would be counterproductive: prefetching blocks that the hypervisor's emulation already has in RAM is wasteful, for instance. These are rare and usually framed as conditional performance tweaks rather than architectural changes.

In practice, the first time you care about emulated-versus-real is when you are looking at a trace or a sysctl and wondering why a counter is zero. It is almost never a correctness problem.

30.5.3 Why Emulation Exists at All

It is worth pausing to ask why emulated devices exist, given that they are slower than alternatives. The answer is threefold.

First, emulation provides compatibility with existing guest kernels. A hypervisor that offered only paravirtual devices would need to persuade every guest operating system to install paravirtual drivers. That is a solved problem today, but in the early 2000s (when modern hypervisors were being designed) it was a significant barrier. Emulated devices let guests run unmodified; they gained a performance path (paravirtual) later.

Second, emulation is often good enough for low-frequency workloads. A guest that makes one disk I/O per second does not suffer measurably from emulation. A guest that makes a hundred thousand disk I/Os per second suffers badly. The majority of workloads are closer to the first than the second, and for them, emulation is a fine choice.

Third, emulation is easier to implement correctly than paravirtualisation when the guest side is an existing driver. A hypervisor author who wants to support, say, VMware-format disk images does not have to write a paravirtual disk driver for every guest OS; they write a VMware-compatible emulated disk controller, and the existing VMware guest drivers work.

In FreeBSD's bhyve(8), emulated devices include the LPC bridge (for serial ports and the RTC), PCI-IDE and AHCI controllers (for some storage cases), and the E1000 NIC (through the `e1000` backend). These are used for installers and for compatibility with older guests. For performance-oriented work, VirtIO devices are the common choice.

30.5.4 Passthrough as Seen by the Guest

Passthrough deserves a deeper look from the guest's perspective, because it is the most "hardware-like" of the three styles and introduces subtleties that a novice driver author might not expect.

When a PCI device is passed through, the guest sees an exact copy of the device's PCI configuration: vendor ID, device ID, subsystem IDs, BARs, capabilities, MSI-X table. The guest's PCI enumerator claims the device with the same driver it would have used on bare metal. The driver programs the device through register accesses; those accesses are, for the most part, direct (not trapped and emulated), because the hardware virtualisation extensions (Intel VT-x, AMD-V) support mapping a device's MMIO into a guest's address space.

DMA is where subtlety enters. The guest programs physical addresses into the device's DMA registers, but those addresses are guest-physical, not host-physical. Without help, the device would DMA to host-physical addresses that do not correspond to guest memory at all, which would be a security disaster. The help comes from the IOMMU: Intel VT-d or AMD-Vi sits between the device and the host memory bus, and it remaps device-issued addresses to the correct host memory.

From the guest driver's perspective, all of this is invisible, as long as the driver uses `bus_dma(9)` correctly. The `bus_dma(9)` framework records which physical addresses are valid for a given DMA handle, and the kernel sets up the IOMMU mappings to match. A driver that bypasses `bus_dma(9)` and programs physical addresses directly (perhaps by calling `vtophys` on a kernel pointer) is a driver that will work on bare metal but break under passthrough.

Interrupts under passthrough are handled by MSI or MSI-X exclusively; legacy pin-based interrupts cannot be passed through usefully. The guest driver configures MSI/MSI-X in the usual way, and the hypervisor sets up interrupt remapping so that the device's interrupts are delivered to the guest's virtual interrupt controller rather than to the host's.

30.5.5 Firmware and Platform Dependencies Under Passthrough

One area where passthrough can surprise driver authors is firmware. Many modern devices expect their firmware to be loaded by the host, not by the device itself. A device whose driver assumes firmware has already been loaded (perhaps by a boot-time BIOS routine on the host) may fail to initialise inside a guest, because the guest's BIOS has not done the firmware-loading work.

The solution is usually for the driver to carry firmware and load it explicitly. FreeBSD's `firmware(9)` framework makes this straightforward: the driver registers the firmware image (typically a blob compiled into a loadable module), and at attach time it calls `firmware_get` and uploads the blob to the device. A driver that works on both bare metal and passthrough is a driver that does the firmware load itself, rather than relying on platform code.

Similarly, ACPI tables and other platform tables may be different between the host and a guest. A driver that reads an ACPI table to determine board-specific routing (for example, to identify which GPIO controls a power rail) will find a different table in a guest, because the guest's virtual BIOS generates its own. The driver must either provide defaults for cases where the table is missing or treat the missing table as an unsupported platform.

30.5.6 When to Prefer Which Device Style

The three device styles are not mutually exclusive on a single VM. A `bhyve(8)` guest can have an emulated LPC bridge, a VirtIO block device, a passthrough NIC, and a VirtIO console, all at the same time. The administrator chooses which style to use for each role based on the trade-offs.

For installation and legacy guests, emulated devices are right. They require nothing from the guest and they work out of the box.

For typical FreeBSD guests doing ordinary work, VirtIO devices are right. They are fast, standardised, and well supported in FreeBSD's tree. Most production `bhyve(8)` deployments use VirtIO for disk, network, and console.

For performance-critical or hardware-specific workloads, passthrough is right. A guest that runs a GPU-accelerated workload needs a passthrough GPU; a guest that needs real network-card offload features needs a passthrough NIC.

As a driver author, the most likely device style you will write for is VirtIO, because that is where the new driver work is. Emulated and passthrough drivers are usually existing drivers; VirtIO drivers are designed from scratch

for the paravirtual interface.

30.5.7 Paravirtual Devices in Detail

Paravirtual devices are a deeper change. The hypervisor declines to imitate any physical device and instead defines an interface optimised for software on both sides. The guest must have a driver written for that interface; a stock legacy driver will not work. The benefit is performance: a paravirtual interface can batch, amortise, and simplify the host-guest interaction in ways that a realistic hardware model cannot.

The dominant paravirtual family on FreeBSD is VirtIO. VirtIO is not a FreeBSD invention; it is a cross-platform standard designed for use across Linux, FreeBSD, and other guest operating systems. FreeBSD's VirtIO implementation lives under `/usr/src/sys/dev/virtio/`, with individual drivers for block devices (`virtio_blk`), network devices (`if_vtnet`), serial consoles (`virtio_console`), SCSI (`virtio_scsi`), entropy (`virtio_random`), and memory ballooning (`virtio_balloon`). Other VirtIO device types exist (9p filesystems, input devices, sockets), and they too have FreeBSD drivers in varying states of maturity.

From the outside, a VirtIO device looks like a PCI device (when attached via PCI) or a memory-mapped device (when attached via MMIO, which is common on embedded and ARM systems). The guest kernel enumerates it the same way it would any other device: a PCI bus scan finds a device with vendor ID `0x1af4` and a device ID indicating the type, or the guest's device tree advertises a VirtIO MMIO transport with a known compatible string.

Once probed and attached, the driver talks to the device through a small set of primitives: device feature bits, device status, device configuration space, and a set of **virtqueues**. The virtqueues are the heart of the VirtIO model. Each one is a ring of descriptors in shared memory, writable by the guest and readable by the host; the host has a parallel ring of used descriptors that the guest reads to discover completions. All the actual I/O happens through the rings; the registers the driver touches are almost exclusively used for setup, notification, and feature negotiation.

This design is radically different from an emulated-device driver. A VirtIO driver almost never issues a register write during normal operation. It posts buffers into rings, notifies the host that new work is available, and reads completions from the ring when they arrive. The per-request overhead is a handful of memory accesses, not a register trap. When the system is busy, the driver can post many requests and notify once; the host can batch many completions and signal once. The result is an order-of-magnitude performance improvement over an equivalent emulated driver.

30.5.8 Paravirtual Devices and the Driver Author

For a driver author, paravirtual devices are where a lot of new work happens in FreeBSD's driver ecosystem. VirtIO is well-defined and stable, and writing a new VirtIO-backed device type is a reasonable weekend project once you have read the foundational drivers. More importantly, the patterns of VirtIO (shared rings, feature negotiation, descriptor-based transactions) echo through other FreeBSD subsystems (notably `if_netmap` and portions of the network stack), so time invested in understanding them pays dividends.

This chapter will spend all of Section 3 on VirtIO fundamentals. For now, what matters is the framing: a paravirtual driver is a first-class guest driver that was designed to live inside a VM, with the performance characteristics that design affords.

30.5.9 Passthrough Devices in Detail

Passthrough is the third style, and in some ways the most interesting because it collapses the virtualisation question back into the bare-metal question, with a twist.

In passthrough, the hypervisor declines to abstract the device at all. It hands the guest a real PCI or PCIe device: for example, a specific physical NIC in one of the host's slots, or a specific NVMe SSD, or a specific GPU. The guest's driver talks to that real hardware as if it were the host. The guest even receives the device's interrupts, through a mechanism the hypervisor provides.

There are three reasons passthrough exists. First, performance: a passthrough NIC delivers line-rate traffic to the guest with no software overhead per packet. Second, access to hardware features the hypervisor does not emulate: GPU acceleration, NVMe-specific queuing, hardware crypto engines. Third, licensing or certification: some proprietary drivers only work with real hardware and cannot be used through any emulation layer.

The cost of passthrough is isolation. Once a device is passed through to a guest, the host no longer owns it in any useful sense; it cannot use the NIC itself, and other guests cannot share it. Live migration becomes difficult or impossible, because the target host may not have the same physical device in the same slot. The guest owns the hardware until the guest stops running.

30.5.10 Passthrough and the IOMMU

Passthrough sounds simple in principle, but it requires one major piece of hardware support: an IOMMU. An IOMMU is to DMA what an MMU is to CPU memory access: a hardware-enforced translation table between the view of memory a device has (its "DMA address") and real physical memory. Without an IOMMU, a passthrough device could DMA into any physical memory on the host, including the host's kernel data, with obvious and terrifying consequences. With an IOMMU, the hypervisor can constrain the device to DMA only into the memory regions assigned to the guest, preserving the security boundary.

On amd64 systems, the IOMMU is Intel VT-d (on Intel CPUs) or AMD-Vi (on AMD CPUs). It is usually enabled through a kernel configuration option and some BIOS settings. FreeBSD's `bhyve(8)` uses the IOMMU through the `pci_passthru(4)` mechanism, which we will meet in Section 5.

For the driver author, the IOMMU is invisible most of the time. The `bus_dma(9)` API in the guest behaves the same way it does on bare metal; the guest driver's `bus_dmamem_alloc()` and `bus_dmamap_load()` calls produce DMA addresses that, from the guest's point of view, look the same as they would on bare metal. The IOMMU translation happens one layer below, between the device and the real memory, and the guest driver does not participate in it.

The one place where the IOMMU matters for the driver author is when the IOMMU is misconfigured or missing, and DMA starts to fail in mysterious ways. If you ever see "DMA timeout" messages from a passthrough driver, the IOMMU is often the first suspect. Chapter 32 will go deeper into this.

30.5.11 The Driver Author's Mental Model

If you step back from these three styles, a clear pattern emerges. The guest kernel always loads a driver; the driver always sees a device; the device's behaviour is always defined by some interface. What changes between the styles is which interface and who implements it.

In emulated devices, the interface is the real hardware's register model, and the hypervisor implements it in software. The driver is the real-hardware driver.

In paravirtual devices, the interface is a bespoke software-optimised model, and the hypervisor implements it natively. The driver is paravirtual-specific.

In passthrough devices, the interface is the real hardware's register model, and the hardware itself implements it, with the IOMMU as a gatekeeper. The driver is the real-hardware driver, running in the guest, with DMA addresses translated transparently.

The driver author is therefore working with the same skills in all three cases. The shape of the work differs only in which interface is in play. Once you understand that, the rest of this chapter becomes a study of specific interfaces, specific FreeBSD APIs, and specific deployment contexts.

30.5.12 A Note on Hybrid Models

Real systems often mix these styles. A single guest may have an emulated serial port for early boot logging, a paravirtual NIC for main network traffic, a paravirtual block device for the root filesystem, and a passthrough GPU for acceleration. Each of those devices is governed by its own driver, and the guest's driver toolkit needs to know the patterns for all three. This is not unusual; it is the common case for any non-trivial VM.

As a driver author, the implication is that you usually do not need to build a universal driver for all three styles. You build the driver for the style your device uses, and you coexist with drivers for the other styles. The per-device decisions about which style to use are made by the hypervisor administrator, not by the driver.

30.5.13 Wrapping Up

Three styles of guest device, three kinds of driver work, one underlying mental model: a driver talks to an interface, and the interface is implemented by whichever layer the hypervisor chooses. Emulated is easy but slow; paravirtual is fast and requires custom drivers; passthrough is near-native and requires an IOMMU. The rest of this chapter concentrates on the paravirtual style, with VirtIO as the example, because that is where most of the interesting new driver work in FreeBSD happens. In Section 3 we open the VirtIO toolkit and start examining its parts.

30.6 Section 3: VirtIO Fundamentals and virtqueue(9)

VirtIO is a standard. It defines a way for a guest to talk to a device that exists only in software, without reference to any physical hardware. The standard is maintained by the OASIS VirtIO Technical Committee and is implemented by a number of hypervisors, including bhyve(8), QEMU, and the Linux KVM family. Because the standard is shared, a VirtIO guest driver written for a FreeBSD guest will talk to a VirtIO device presented by any conforming hypervisor.

This section introduces the VirtIO model at the level of detail a driver author needs. It is long, because VirtIO has enough moving parts to justify the space, and short compared to the VirtIO specification itself, which spans hundreds of pages. We will focus on the parts you need to write a FreeBSD guest driver.

30.6.1 The Ingredients of a VirtIO Device

A VirtIO device exposes a small number of elements to its guest driver. Understanding each one is the foundation for everything that follows.

The first element is the **transport**. VirtIO devices can appear over several transports: VirtIO over PCI (the usual case on amd64 desktops, servers, and cloud), VirtIO over MMIO (the usual case on ARM and embedded systems), and VirtIO over channel I/O (used on IBM mainframes). From the guest's view, the transport dictates how the device is enumerated and how registers are read, but once the driver has hold of a `device_t`, the transport recedes into the background. FreeBSD's VirtIO framework provides a transport-agnostic API.

The second element is the **device type**. Every VirtIO device has a single-byte type identifier (`VIRTIO_ID_NETWORK` is 1, `VIRTIO_ID_BLOCK` is 2, `VIRTIO_ID_CONSOLE` is 3, `VIRTIO_ID_ENTROPY` is 4, `VIRTIO_ID_BALLOON` is 5, `VIRTIO_ID_SCSI` is 8, and so on, as listed in `/usr/src/sys/dev/virtio/virtio_ids.h`). The type tells the guest what the device does; the guest dispatches to the right driver based on the type.

The third element is the **device feature bits**. Each device type defines a set of optional features, each represented by a bit in a 64-bit mask. Some features are universal (the ability to use indirect descriptors, for example), and some are device-specific (VirtIO block devices may advertise write-caching, discard support, geometry information, and more). The guest driver reads the device's advertised features, selects which of them the driver knows how to use, and writes back a negotiated feature mask. The device then agrees to the negotiated set and rejects any attempt to use a feature outside it. This negotiation is the mechanism by which VirtIO devices and drivers evolve without breaking each other.

The fourth element is the **device status**. A small byte-level register reports where in the lifecycle the device sits: acknowledged by the driver, driver found for this device type, features negotiated, device set up and ready, and so on. Writing to the status register drives the device through its lifecycle; reading it tells the driver where it is.

The fifth element is the **device configuration space**. Each device type has its own small layout of bytes that carry device-specific configuration: a block device's capacity, a network device's MAC address, a console device's port count. The guest reads the configuration space to learn device-specific details and occasionally writes to it to request a configuration change.

The sixth element, and the most important for this chapter, is the **set of virtqueues**. Each VirtIO device has one or more virtqueues, and the virtqueues are where almost all the actual work happens. Think of each virtqueue as a bounded-size bidirectional conveyor belt between the guest and the host. The guest places requests on the belt; the host consumes them, acts on them, and places completions on a parallel return belt; the guest reads the completions to learn what happened. The conveyor belt is implemented as a ring of descriptors in shared memory.

30.6.2 Virtqueues and the Shared-Ring Model

At the heart of VirtIO is the virtqueue, the single most important abstraction in the whole protocol. A virtqueue is a ring of descriptors, held in memory that both the guest and the host can read and write. Its size is a power of two, chosen at device initialisation; typical values are 128 or 256 entries, though larger values are permitted.

Each descriptor in the ring describes a single scatter-gather buffer in guest memory: its guest physical address, its length, and a couple of flags (is this buffer readable or writable from the device's point of view, does this

descriptor chain to a next descriptor, does it point to an indirect descriptor table). The driver fills descriptors, chains them as needed to represent multi-buffer transactions, and writes to a small index to announce that new descriptors are available. The host's device implementation walks the descriptors, performs the work they describe, and writes to its own index to announce that the work is done.

There are three rings inside a virtqueue, not one. The **descriptor table** holds the actual descriptors. The **available ring** (driven by the guest) lists the indices of descriptor chains the driver has made available. The **used ring** (driven by the host) lists the indices of descriptor chains the device has consumed and whose results are ready. The split is subtle but important: the driver writes to the available ring and reads from the used ring, while the host does the opposite. Each side reads what the other side wrote, without locks, through careful use of memory barriers and atomic index updates.

From the driver author's point of view, you almost never manipulate the rings by hand. FreeBSD's `virtqueue(9)` API abstracts them behind a handful of functions. You call `virtqueue_enqueue()` to push a scatter-gather list onto the available ring. You call `virtqueue_notify()` to let the host know new work is available. You call `virtqueue_dequeue()` to pull the next completion off the used ring. If you are polling rather than taking interrupts, you call `virtqueue_poll()` to wait for the next completion. The API hides the index arithmetic, the memory barriers, and the flag juggling; your code deals with scatter-gather lists and cookies.

The cookie mechanism deserves a line. When you enqueue a scatter-gather list, you supply an opaque pointer, the cookie, which the API remembers. When you dequeue a completion, you receive that cookie back. This lets the driver associate each completion with the request that produced it, without the ring having to carry any driver-specific context. A driver typically passes the pointer to a softc-level request structure as the cookie; dequeue gives back the same pointer, and the driver resumes processing.

30.6.3 Feature Negotiation in Practice

Before the driver uses any of these mechanisms, it must negotiate features. The sequence is simple and looks the same in every VirtIO driver.

First, the driver reads the device's advertised feature bits with `virtio_negotiate_features()`. The argument is a mask of features the driver is prepared to use; the return value is the intersection of the driver's mask and the device's advertised bits. The device has now committed to supporting only that subset, and the driver knows exactly which features are in play.

Second, the driver calls `virtio_finalize_features()` to seal the negotiation. After this call, the device will reject any attempt to enable features outside the negotiated set. The driver's subsequent setup code can inspect the negotiated mask through `virtio_with_feature()`, which returns true if a given feature bit is present.

Third, the driver allocates virtqueues. The exact number and per-queue parameters depend on the device type. A `virtio_random` device allocates one virtqueue; a `virtio_net` device allocates at least two (one for receive, one for transmit), more if multiple queues are negotiated. Allocation uses `virtio_alloc_virtqueues()`, passing an array of `struct vq_alloc_info`, each entry describing one queue's name, callback, and maximum indirect-descriptor size.

Fourth, the driver sets up interrupts with `virtio_setup_intr()`. The interrupt handlers are the callbacks from step three. When the host has posted a completion to a queue, the guest receives a virtual interrupt, the handler runs, and it processes the completions by calling `virtqueue_dequeue()` in a loop.

This four-step sequence is the backbone of every VirtIO driver in `/usr/src/sys/dev/virtio/`. Read `vtrnd_attach()` in `/usr/src/sys/dev/virtio/random/virtio_random.c` and you will see it clearly: feature negotiation, queue allocation, and interrupt setup, in exactly that order. Read `vtblk_attach()` in `/usr/src/sys/dev/virtio/block/virtio_blk.c` and you will see the same sequence, with more moving parts because the device is more complex, but the same skeleton.

30.6.4 A Walkthrough of `virtio_random`

Because `virtio_random` is the smallest complete VirtIO driver in the FreeBSD tree, it is the best one to read first. The whole file is under four hundred lines, and every line is there for a reason. Let us trace through the important parts.

The `softc` is small:

```
struct vtrnd_softc {
    device_t      vtrnd_dev;
    uint64_t      vtrnd_features;
    struct virtqueue *vtrnd_vq;
    eventhandler_tag eh;
    bool          inactive;
    struct sglist  *vtrnd_sg;
    uint32_t       *vtrnd_value;
};
```

The fields are the device handle, the negotiated feature mask (the device has no feature bits so this will always be zero), a pointer to the single virtqueue, an event handler tag for shutdown hooks, a flag that is set to true during teardown, a scatter-gather list used for every enqueue, and a buffer in which the device will store each batch of entropy.

The device methods table is the standard newbus skeleton:

```
static device_method_t vtrnd_methods[] = {
    DEVMETHOD(device_probe,      vtrnd_probe),
    DEVMETHOD(device_attach,     vtrnd_attach),
    DEVMETHOD(device_detach,     vtrnd_detach),
    DEVMETHOD(device_shutdown,   vtrnd_shutdown),
    DEVMETHOD_END
};

static driver_t vtrnd_driver = {
    "vtrnd",
    vtrnd_methods,
    sizeof(struct vtrnd_softc)
};
```

```
VIRTIO_DRIVER_MODULE(virtio_random, vtrnd_driver, vtrnd_modevent, NULL);
MODULE_VERSION(virtio_random, 1);
MODULE_DEPEND(virtio_random, virtio, 1, 1, 1);
MODULE_DEPEND(virtio_random, random_device, 1, 1, 1);
```

VIRTIO_DRIVER_MODULE is a short macro that expands to two DRIVER_MODULE declarations, one for virtio_pci and one for virtio_mmio. This is how the same driver attaches on both transports without any transport-specific code of its own; the framework routes it to whichever transport finds a matching device.

The probe function is one line:

```
static int
vtrnd_probe(device_t dev)
{
    return (VIRTIO_SIMPLE_PROBE(dev, virtio_random));
}
```

VIRTIO_SIMPLE_PROBE consults the PNP match table that was declared above through VIRTIO_SIMPLE_PNPINFO. The match table tells the kernel that this driver wants devices whose VirtIO type is VIRTIO_ID_ENTROPY.

The attach function is more substantial. It allocates the entropy buffer and the scatter-gather list, sets up features, allocates a single virtqueue, installs a shutdown event handler, registers itself with the FreeBSD random(4) framework, and posts its first buffer into the virtqueue:

```
sc = device_get_softc(dev);
sc->vtrnd_dev = dev;
virtio_set_feature_desc(dev, vtrnd_feature_desc);

len = sizeof(*sc->vtrnd_value) * HARVESTSIZE;
sc->vtrnd_value = malloc_aligned(len, len, M_DEVBUF, M_WAITOK);
sc->vtrnd_sg = sglist_build(sc->vtrnd_value, len, M_WAITOK);

error = vtrnd_setup_features(sc);      /* feature negotiation */
error = vtrnd_alloc_virtqueue(sc);     /* allocate queue */

/* [atomic global-instance check] */

sc->eh = EVENTHANDLER_REGISTER(shutdown_post_sync,
    vtrnd_shutdown, dev, SHUTDOWN_PRI_LAST + 1);

sc->inactive = false;
random_source_register(&random_vtrnd);

vtrnd_enqueue(sc);
```

Feature negotiation is trivial because the driver advertises `VTRND_FEATURES = 0`:

```
static int
vtrnd_negotiate_features(struct vtrnd_softc *sc)
{
    device_t dev = sc->vtrnd_dev;
    uint64_t features = VTRND_FEATURES;

    sc->vtrnd_features = virtio_negotiate_features(dev, features);
    return (virtio_finalize_features(dev));
}
```

The feature mask of the device is irrelevant; the driver wants no features and gets no features, and the finalisation succeeds trivially. A more complex driver would build a non-zero mask and check it afterwards.

The virtqueue allocation asks for a single queue:

```
static int
vtrnd_alloc_virtqueue(struct vtrnd_softc *sc)
{
    device_t dev = sc->vtrnd_dev;
    struct vq_alloc_info vq_info;

    VQ_ALLOC_INFO_INIT(&vq_info, 0, NULL, sc, &sc->vtrnd_vq,
        "%s request", device_get_nameunit(dev));

    return (virtio_alloc_virtqueues(dev, 0, 1, &vq_info));
}
```

The first argument to `VQ_ALLOC_INFO_INIT` is the maximum indirect-descriptor-table size, which is zero for this driver because it does not use indirect descriptors. The second argument is the interrupt handler callback, which is `NULL` because this driver uses polling rather than interrupt-driven completion. The third is the argument to that handler. The fourth is the output pointer for the virtqueue handle. The fifth is a format string that produces the queue's name.

The enqueue function is short:

```
static void
vtrnd_enqueue(struct vtrnd_softc *sc)
{
    struct virtqueue *vq = sc->vtrnd_vq;

    KASSERT(virtqueue_empty(vq), ("%s: non-empty queue", __func__));

    error = virtqueue_enqueue(vq, sc, sc->vtrnd_sg, 0, 1);
}
```

```

    KASSERT(error == 0, ("%s: virtqueue_enqueue returned error: %d",
        __func__, error));

    virtqueue_notify(vq);
}

```

This pushes a scatter-gather list describing the entropy buffer onto the ring. The cookie is `sc` itself. The next two arguments are “readable segments” (none, the device writes the buffer) and “writable segments” (one, the entropy buffer). `virtqueue_notify()` kicks the host so it starts filling the buffer.

Because there are no interrupts, the driver uses polling to retrieve completions:

```

static int
vtrnd_harvest(struct vtrnd_softc *sc, void *buf, size_t *sz)
{
    struct virtqueue *vq = sc->vtrnd_vq;
    void *cookie;
    uint32_t rrlen;

    if (sc->inactive)
        return (EADLK);

    cookie = virtqueue_dequeue(vq, &rrlen);
    if (cookie == NULL)
        return (EAGAIN);

    *sz = MIN(rrlen, *sz);
    memcpy(buf, sc->vtrnd_value, *sz);

    vtrnd_enqueue(sc);    /* re-post the buffer */
    return (0);
}

```

The lifecycle here is the full VirtIO cycle. Enqueue posts a buffer; the host fills it with entropy and marks the descriptor “used”; the driver dequeues the used descriptor, extracts the cookie and the length, copies the result out to its caller, and re-posts the buffer for the next round. If the queue is empty (no completion yet), `virtqueue_dequeue()` returns `NULL` and the driver returns `EAGAIN`.

This is the full shape of a VirtIO driver. Everything else, from `virtio_blk` to `if_vtnet`, is elaboration on this base: more queues, more features, more work per completion, more bookkeeping per request. The skeleton is the same.

30.6.5 What the `virtqueue(9)` API Buys You

Step back for a moment and consider how much complexity the API abstracts away. The ring arithmetic, with its wraparound and its careful handling of available versus used indices, is entirely hidden. The memory barriers

that keep the guest and the host in sync, which are subtle enough to be a common source of bugs in handwritten ring code, are placed by the API. The optional features (indirect descriptors, event indexes) are turned on and off based on negotiation without the driver having to know how they work.

The cost of that abstraction is that a driver author who only ever uses the API will not understand the rings themselves in depth. That is usually fine, but on the rare occasion when you debug a ring-layer mismatch (because you posted a descriptor with the wrong writable count, say, and the host is rejecting it) you will want to know what is going on. The VirtIO specification is the reference for the underlying ring layout, and `/usr/src/sys/dev/virtio/virtio_ring.h` is the FreeBSD side of the same layout. They repay reading at least once.

30.6.6 Indirect Descriptors

A small optimisation worth knowing about is the indirect-descriptor feature (`VIRTIO_RING_F_INDIRECT_DESC`). When negotiated, it allows a driver to describe a multi-buffer transaction in an indirect descriptor table rather than a chain of descriptors on the main ring. The ring entry becomes a single descriptor that points to a block of descriptors, so a large scatter-gather list consumes only one slot on the ring rather than many.

Indirect descriptors matter in workloads that do large, scatter-gather-heavy transactions, such as network drivers sending packets with many fragments. For small drivers, they are an optional flourish. The `virtqueue(9)` API handles the mechanism transparently: if you pass a non-zero `vqai_maxindirsz` during queue allocation, the kernel can use indirects; if you pass zero, it cannot.

30.6.7 A Deeper Look at the Ring Layout

For a reader who wants to know what `virtqueue(9)` is hiding, here is a brief tour of the underlying layout. The details matter only when you are debugging a ring-layer issue; you can skip this subsection on first reading.

Each `virtqueue` has three main structures in memory, allocated contiguously (with alignment requirements that differ slightly between legacy and modern VirtIO).

The **descriptor table** is an array of `struct vring_desc`, one entry per descriptor slot:

```
struct vring_desc {
    uint64_t addr;      /* guest physical address of the buffer */
    uint32_t len;       /* length of the buffer */
    uint16_t flags;     /* VRING_DESC_F_NEXT, _WRITE, _INDIRECT */
    uint16_t next;      /* index of the next descriptor if chained */
};
```

The descriptor table has as many entries as the queue size (typically 128 or 256). Unused descriptors are linked into a free list maintained by the driver; used descriptors form chains that describe single transactions.

The **available ring** is what the driver writes and the device reads:


```

struct vring_avail {
    uint16_t flags;
    uint16_t idx;                /* producer index, monotonic */
    uint16_t ring[QUEUE_SIZE];  /* head-of-chain indices */
    uint16_t used_event;        /* for VRING_F_EVENT_IDX */
};

```

When the driver enqueues a new transaction, it picks a chain of descriptors from the free list, fills them in, places the head descriptor's index into `ring[idx % QUEUE_SIZE]`, and then increments `idx`. The increment uses a release-style memory barrier so that the device sees the new ring entry before it sees the new `idx`.

The **used ring** is what the device writes and the driver reads:

```

struct vring_used_elem {
    uint32_t id;                /* index of the head descriptor */
    uint32_t len;               /* number of bytes written */
};

struct vring_used {
    uint16_t flags;
    uint16_t idx;                /* producer index */
    struct vring_used_elem ring[QUEUE_SIZE];
    uint16_t avail_event;        /* for VRING_F_EVENT_IDX */
};

```

When the device finishes a transaction, it writes an entry to `ring[idx % QUEUE_SIZE]` with the head index and the byte count, then increments `idx`. The driver observes the increment and dequeues accordingly.

The subtlety of the format is the synchronisation. Guest and host are not synchronising through locks (they do not share locks); they are synchronising through ordered writes and atomic index updates. The spec is careful to specify the memory barriers on each side, and the `virtqueue(9)` implementation places them correctly. Getting the barriers wrong is a common bug in hand-rolled ring implementations; that is one reason to use the framework.

30.6.8 Event Indexes: A Performance Detail

The `VIRTIO_F_RING_EVENT_IDX` feature allows both sides to suppress notifications when they are not needed. The mechanism is two extra fields, `used_event` in the available ring and `avail_event` in the used ring. Each side writes a value saying “interrupt me only when the other side's producer index passes this value”.

The practical effect is that a driver producing requests at a high rate does not cause the host to be interrupted on every enqueue; instead, the host reads the current `used_event` and only delivers a notification when the producer index catches up. Similarly, a device completing transactions at a high rate does not cause the guest to be interrupted on every dequeue; the guest sets `avail_event` to suppress notifications it does not need.

This optimisation halves the notification overhead in worst-case workloads. It is a feature that most modern VirtIO drivers negotiate, and the `virtqueue(9)` API handles it transparently. As a driver author, you simply negotiate the feature and do not have to think about the details.

30.6.9 Legacy Versus Modern VirtIO

VirtIO has two major versions. Legacy (sometimes called “virtio 0.9”) was the original specification; modern (“virtio 1.0” and up) came later with a cleaner design. FreeBSD supports both.

The practical differences are in configuration-space layout, byte order, and a few feature-bit semantics. Legacy VirtIO puts configuration in native byte order; modern VirtIO is always little-endian. Legacy mandates some fields in a specific position; modern uses capability structures to describe the layout flexibly. Legacy defines a specific set of feature bits below bit 32; modern extends them above bit 32.

For driver authors, the framework hides most of these differences. Negotiating `VIRTIO_F_VERSION_1` puts the driver in modern mode; not negotiating it puts the driver in legacy mode. The `virtio_with_feature` helper checks the negotiated state. As long as your driver follows the `virtqueue(9)` API and uses `virtio_read_device_config` rather than direct configuration-space access, you can ignore the legacy/modern distinction almost entirely.

The one place where the distinction leaks is in how configuration-space fields are accessed. Legacy VirtIO uses `virtio_read_device_config` with a size argument and assumes native byte order; modern VirtIO assumes little-endian and uses helpers that byte-swap if the host is big-endian. The framework handles this, but a driver that does configuration-space access by hand (rather than through the framework) would have to be aware.

30.6.10 Feature Bits Worth Knowing

Each VirtIO device type has its own set of feature bits. A few are worth knowing in the abstract, because they show up in many drivers.

- `VIRTIO_F_VERSION_1` indicates that the device supports the modern, version 1 of the VirtIO specification. Almost all modern drivers negotiate this bit.
- `VIRTIO_F_RING_EVENT_IDX` enables a more efficient form of interrupt suppression. When both sides support it, notifications and interrupts are only sent when they are actually useful, reducing overhead under load.
- `VIRTIO_F_RING_INDIRECT_DESC`, already mentioned, allows indirect descriptors.
- `VIRTIO_F_ANY_LAYOUT` relaxes the rules about how descriptors are ordered in a chain.

Each device type has its own set on top of these. For a block device, `VIRTIO_BLK_F_WCE` indicates that the device has a write cache; `VIRTIO_BLK_F_FLUSH` provides a flush command. For a network device, `VIRTIO_NET_F_CSUM` advertises offloaded checksum calculation. For entropy, there are no device-specific feature bits.

The general rule is: read the feature bits the device advertises, decide which the driver can use, ignore the others. Negotiation is not about “demanding” features; it is about agreeing on an intersection.

30.6.11 A Second Example: Structure of `virtio_net`

It is worth a brief look at a larger VirtIO driver to see how the skeleton scales. The network driver `if_vtnet` lives in `/usr/src/sys/dev/virtio/network/if_vtnet.c`. It is roughly ten times the length of `virtio_random`, but the extra size comes from features and completeness, not from complexity in the core VirtIO interaction. Knowing how the scale-up works makes future reading more productive.

`if_vtnet` begins, like every VirtIO driver, with a module registration and a `device_method_t` table. The methods table is longer because `if_vtnet` hooks into the `ifnet(9)` framework; you will see `DEVMETHOD` entries for `device_probe`, `device_attach`, `device_detach`, `device_suspend`, `device_resume`, and `device_shutdown`, each implemented by the driver. The `VIRTIO_DRIVER_MODULE` macro at the bottom of the file registers the driver for both PCI and MMIO transports, exactly as `virtio_random` does.

The `softc`, `struct vtnet_softc`, is much larger than `vtrnd_softc`, but its role is the same: hold the state that the driver needs to serve the device. Notable additions include a pointer to the kernel's `ifnet` structure (`vtnet_ifp`), an array of receive queues (`vtnet_rxqs`), an array of transmit queues (`vtnet_txqs`), a feature mask, a cached MAC address, and statistics counters. Each queue structure contains its own `virtqueue` pointer, `softc` back-reference, taskqueue context, and a number of bookkeeping fields. A single multi-queue `virtio-net` device can have tens of `virtqueues`; each receive/transmit pair represents one queue pair that can run in parallel with the others.

The probe function uses the same `VIRTIO_SIMPLE_PROBE` pattern as `virtio_random`, but matching on `VIRTIO_ID_NETWORK` (which is 1). The attach function is substantial: it reads feature bits, negotiates them, allocates the queues, reads the device's MAC address from configuration space (`virtio_read_device_config`), initialises the `ifnet`, registers with the network stack, and sets up callouts for link-status polling. Each of these steps is a small function of its own, and the flow follows the same “negotiate, allocate, register, start” rhythm that `virtio_random` illustrates.

The receive path uses `virtqueue_dequeue` in a loop to drain completed packets. For each completed descriptor, the driver reads the packet header (`virtio_net_hdr`) that the device wrote to the first few bytes of the buffer, extracts metadata such as checksum-status flags and GSO segment sizes, and passes the packet up to `if_input`. If the packet is the last one in a batch, the driver re-posts fresh receive buffers to keep the queue primed for the next round.

The transmit path uses `virtqueue_enqueue` with a scatter-gather list covering both the packet header and the packet body. The cookie is the `mbuf(9)` pointer, so the transmit completion callback can free the `mbuf` when the device is done with it. If the queue becomes full, the driver stops accepting outbound packets until some space frees up, which is standard `ifnet` driver discipline.

Feature negotiation in `if_vtnet` is interesting because the feature set is large. Features like `VIRTIO_NET_F_CSUM` (transmit checksum offload), `VIRTIO_NET_F_GUEST_CSUM` (receive checksum offload), `VIRTIO_NET_F_GSO` (generic segmentation offload), and `VIRTIO_NET_F_MRG_RXBUF` (merged receive buffers) each change how the driver programs the `virtqueues` and how it interprets incoming data. The driver's feature-negotiation function picks a set that the driver implements, offers it for negotiation, and then queries the negotiated mask to decide which code paths to enable.

The lesson from `if_vtnet` is that a VirtIO driver scales through feature diversity and queue multiplicity, not through a fundamentally different architecture. If you understand `virtio_random`'s lifecycle, you understand `if_vtnet`'s lifecycle; the extra code is in feature-specific sub-paths and in `ifnet`-specific hooking. When the time comes to read `if_vtnet.c`, focus on `vtnet_attach` first, then walk the feature-negotiation function, then

look at `vtnet_rxq_eof` (receive) and `vtnet_txq_encap` (transmit). Those four functions explain 80% of the driver.

30.6.12 A Third Example for Intuition: `virtio_blk`

`virtio_blk`, in `/usr/src/sys/dev/virtio/block/virtio_blk.c`, is shorter than `if_vtnet` but longer than `virtio_random`. It illustrates a third common pattern: a driver that exposes a block-oriented device rather than a stream-oriented one.

The softc of `vtblk` contains a `virtqueue`, a pool of request structures, statistics, and the geometry information read from the device's configuration space. The probe and attach follow the familiar pattern. The interesting part is how requests are structured.

For every block I/O operation, the driver builds a three-segment descriptor chain: a header descriptor (containing the operation type and sector number), zero or more data descriptors (the actual payload, which is writable from the device's perspective for reads and readable from the device for writes), and a status descriptor (where the device writes a single-byte status code). The header and status are small structures; the data segments are whatever the `bio(9)` request brought in.

This three-segment layout is a common VirtIO idiom. If you see a driver building a chain that starts with a header and ends with a status, you are looking at a request-response device. `virtio_scsi` uses the same layout, as does `virtio_console` (for control messages). Recognising the pattern speeds up reading.

30.6.13 Common Mistakes When Reading VirtIO Code

When reading any VirtIO driver for the first time, a few patterns can confuse a reader who is not expecting them.

The first is the mix of probe patterns. Some drivers use `VIRTIO_SIMPLE_PROBE`; others use their own probe functions that do more complex feature checks. Both are legitimate, and the former is a shorthand for a common case.

The second is the module registration style. `VIRTIO_DRIVER_MODULE` is a macro that expands to two `DRIVER_MODULE` calls, and reading the macro definition (in `/usr/src/sys/dev/virtio/virtio.h`) makes it clear. Without the macro, you might wonder why the driver is not calling `DRIVER_MODULE` explicitly; with it, you see that it is, once per transport.

The third is the distinction between the VirtIO core functions (`virtio_*`) and the `virtqueue` functions (`virtqueue_*`). The core functions operate on the whole device; the `virtqueue` functions operate on a single queue. A driver usually uses both, and the namespace prefix is the hint.

The fourth is the polling-versus-interrupt distinction. A driver that polls (like `virtio_random`) passes `NULL` as the callback in `VQ_ALLOC_INFO_INIT` and calls `virtqueue_poll` to wait for completions. A driver that uses interrupts (like `if_vtnet`) passes a callback and lets the kernel's interrupt infrastructure schedule it. Both are legitimate; the choice depends on the workload's latency sensitivity and on whether the driver can afford to block.

30.6.14 Wrapping Up

This has been a long section, and deliberately so. VirtIO is dense, and its vocabulary is the foundation for most of the interesting work a modern guest-driver author does. You now know that a VirtIO device has a transport, a type, a feature mask, a status, a configuration space, and a set of virtqueues; that the virtqueues are shared rings with separate descriptor, available, and used components; that `virtqueue(9)` abstracts the ring mechanics behind a small API of enqueue, dequeue, notify, and poll; that feature negotiation is the mechanism for forward and backward compatibility; and that the smallest FreeBSD VirtIO driver is `virtio_random`, under four hundred lines of code that illustrate the whole skeleton. You have also seen how larger VirtIO drivers like `if_vtnet` and `virtio_blk` are structured around the same skeleton, scaled up with feature-specific code.

Section 4 takes a different turn. Now that you know what a guest driver is, we consider the question: how does the driver know it is in a guest in the first place?

30.7 Section 4: Hypervisor Detection and Environment-Aware Driver Behaviour

There are legitimate reasons for a driver to know whether it is running inside a virtual machine. A driver that is inclined to use an expensive hardware counter may skip it on a hypervisor where the counter is known to be unreliable. A driver that polls a hardware register in a tight loop may tune its polling interval differently under virtualisation, because each poll becomes a hypervisor exit. A driver that emits a warning about a suspicious interrupt rate may suppress the warning when running in a cloud where background noise is normal. A few FreeBSD drivers in the tree use this kind of adaptation.

There are also illegitimate reasons. A driver that tries to behave differently to “hide” something from a hypervisor is building anti-debugging logic, which has its place in anti-tamper software but has none in a general-purpose driver. A driver that branches on the exact hypervisor brand to gain performance is building fragile code whose behaviour will drift as hypervisors evolve. We will focus on the legitimate uses.

30.7.1 The `vm_guest` Global

The FreeBSD kernel exposes a single global variable, `vm_guest`, which records the hypervisor it detected at boot. The variable is declared in `/usr/src/sys/sys/systm.h`:

```
extern int vm_guest;

enum VM_GUEST {
    VM_GUEST_NO = 0,
    VM_GUEST_VM,
    VM_GUEST_XEN,
    VM_GUEST_HV,
    VM_GUEST_VMWARE,
    VM_GUEST_KVM,
    VM_GUEST_BHYVE,
    VM_GUEST_VBOX,
```

```
VM_GUEST_PARALLELS,  
VM_GUEST_NVMM,  
VM_LAST  
};
```

The values are self-explanatory. `VM_GUEST_NO` means the kernel is running on bare metal; the other values identify specific hypervisors. `VM_GUEST_VM` is a fallback for “a virtual machine of unknown type,” when the kernel could tell it was virtualised but could not pin down which hypervisor.

The corresponding `sysctl` is `kern.vm_guest`, which yields the human-readable form of the same information. The string values mirror the enum: “none”, “generic”, “xen”, “hv”, “vmware”, “kvm”, “bhyve”, “vbox”, “parallels”, “nvmm”. You can query it from the shell:

```
sysctl kern.vm_guest
```

On a bare-metal FreeBSD machine you will see none. On a bhyve(8) guest, you will see bhyve. On a VirtualBox guest, vbox. And so on.

30.7.2 How the Detection Works

The detection happens early in boot, on architectures where hypervisor presence is detectable. On amd64, the kernel examines the CPUID leaf for a hypervisor present bit and then probes specific leaves associated with each hypervisor brand. The code lives in `/usr/src/sys/x86/x86/identcpu.c`, in functions named `identify_hypervisor()` and `identify_hypervisor_cpuid_base()`. On arm64, a similar mechanism exists through firmware interfaces.

The driver author does not need to care about the detection code. What matters is that by the time your driver’s `attach()` runs, `vm_guest` has been set to its final value.

30.7.3 When to Consult `vm_guest`

Consulting `vm_guest` is appropriate in a driver when the driver’s correctness or performance depends on knowing the runtime environment. A few real examples from the tree illustrate the range:

- Some performance counters in `/usr/src/sys/kern/kern_resource.c` adjust their behaviour when `vm_guest == VM_GUEST_NO`, because the kernel assumes a bare-metal cost model is accurate there.
- Some time-related code in the amd64 tree may not use certain timing primitives under specific hypervisors where those primitives are known to be unreliable.
- Some informational messages in driver probe paths are suppressed under virtualisation to avoid confusing users who expect the driver to behave “differently” in a VM.

A driver you write should use `vm_guest` sparingly. A common anti-pattern is to test for `VM_GUEST_VM` as a blanket switch (“if virtualised, do X”), which is usually a sign that the driver has a bug on real hardware that it is papering over. Prefer to handle the underlying cause directly, and use `vm_guest` only when the dependency is genuinely on the environment rather than on a specific hardware quirk.

30.7.4 A Usage Example

Suppose your driver has a `sysctl` that controls how aggressively it polls a hardware register. The tight loop is fine on bare metal, where each poll is cheap, but under virtualisation each poll costs a hypervisor exit, which is expensive. You might want the default polling interval to be larger under virtualisation:

```
#include <sys/systm.h>

static int
my_poll_default(void)
{
    if (vm_guest != VM_GUEST_NO)
        return (100); /* milliseconds */
    else
        return (10);
}
```

This is a legitimate use of `vm_guest`: the driver is choosing a sensible default based on a property of the environment that genuinely affects its performance characteristics. The value remains overridable by a `sysctl` for users who know better than the default.

Contrast that with an illegitimate use:

```
/* DO NOT DO THIS */
if (vm_guest == VM_GUEST_VMWARE) {
    /* skip interrupt coalescing because VMware does it weirdly */
    sc->coalesce = false;
}
```

Here the driver is branching on a specific hypervisor brand to work around a perceived bug. This is fragile for three reasons. The specific behaviour of VMware may change in a later release; the same issue may apply to KVM or Parallels without the code noticing; and the driver now has a maintenance burden tied to a third-party product. A better approach is to address the root cause (the coalescing code is fragile, so either fix it or expose a `sysctl`), not the symptom (VMware happens to trigger it).

30.7.5 The Sysctl Path

For user-space tools and scripts, the `kern.vm_guest` `sysctl` is a better interface than poking at `/dev/mem` or similar. A small shell script can decide whether to run certain tests based on the environment:

```
if [ "$(sysctl -n kern.vm_guest)" = "none" ]; then
    echo "Running on bare metal, enabling hardware tests"
    run_hardware_tests
else
```

```
echo "Running on a hypervisor, skipping hardware tests"
fi
```

This is how the FreeBSD test suite decides which tests to run in which environment. Reusing the same variable from inside a driver keeps the whole system consistent.

30.7.6 Interaction with Subsystems

A few FreeBSD subsystems adapt their behaviour based on `vm_guest` automatically, and a driver author should be aware of the adaptations without having to touch them.

The timecounter selection code in `/usr/src/sys/kern/kern_tc.c` considers `vm_guest` when picking a default timecounter, because some hypervisors' TSC implementations are unreliable across guest migrations. The driver does not need to care; the kernel simply picks a safer timecounter.

The VirtIO transport drivers (`virtio_pci.c` and `virtio_mmio.c`) do not branch on `vm_guest` directly, because they are already guests by their own probe path. They trust that if they have a device at all, they are running in an environment that emulates VirtIO.

Some network driver code handles the case where GRO (generic receive offload) is known to interact badly with certain hypervisor configurations. These adaptations are not driver-by-driver decisions; they are made in the network stack based on observed characteristics.

As an author of a single driver, your rule of thumb is: if you feel the urge to branch on `vm_guest`, ask yourself whether the issue belongs to the kernel's environment handling rather than to your driver. Usually it does.

30.7.7 Detecting Within a VirtIO Driver

A VirtIO driver, by definition, is running inside some kind of environment that speaks VirtIO. The driver does not usually need to know which specific hypervisor is backing the VirtIO device. That is one of VirtIO's virtues: it is environment-agnostic. The same `virtio_net` driver works under bhyve, under QEMU/KVM, under Google Cloud's backend, and under AWS Nitro, because all of them implement the same standard.

There are two exceptions. One is when the driver hits a performance cliff that is hypervisor-specific, in which case `vm_guest` tells you which workaround to apply. The other is when the driver wants to print a diagnostic message that identifies the hypervisor by name. Both are rare.

30.7.8 Detecting From the Host Side

A host-side driver (one that runs in the host, not in a guest) usually does not care about `vm_guest` at all, because the host is not virtualised by definition. The `sysctl` will return `none` on bare metal, and that is the expected case.

The one subtlety is when the host is itself a guest, as in nested virtualisation scenarios. A FreeBSD host inside a VMware ESXi VM, for example, will have `vm_guest = VM_GUEST_VMWARE`. A `bhyve(8)` running inside such a host will present guests that also see virtualisation, though they see the bhyve presentation rather than the VMware one. The chain of environments can go two or three levels deep in research and test setups. Do

not assume a host is bare metal; if your driver has a host-versus-guest split, use the `sysctl` or the variable to distinguish.

30.7.9 Wrapping Up

`vm_guest` is a small, quiet API that tells your driver, and user-space tools, about the environment. It is easy to use and easy to misuse. Use it to adapt sensible defaults, not to work around hypervisor-specific bugs. Use it to inform user-space decisions through the `sysctl`. Do not make your driver's behaviour depend on the exact hypervisor brand; doing so couples your code to a third-party product's version history, and that is a maintenance tax you do not want to pay.

Section 5 looks at the other side of virtualisation for FreeBSD: the host that runs the guests, and the tools and interfaces that a driver author should understand when running or cooperating with `bhyve(8)`.

30.8 Section 5: `bhyve`, PCI Passthrough, and Host-Side Considerations

So far we have focused on what happens inside the guest. The guest is where most of the learning happens, because the guest is where most of the driver code lives. But FreeBSD has another role in the virtualisation story: it is also a hypervisor. `bhyve(8)` runs virtual machines, and understanding how `bhyve(8)` presents devices to its guests is useful both when you are the guest author (so you know what the host is doing) and when you are the host author (so you know how to share a real device with a guest).

This section steps onto the host side. We will not go deep enough to write hypervisor code, because that is a topic of its own. We will go deep enough for you to understand what the host is doing, what the host-side knobs are, and what a driver author needs to know when cooperating with `bhyve(8)`.

30.8.1 `bhyve` From the Driver Author's Perspective

`bhyve(8)` is a type-2 hypervisor that runs on a FreeBSD host and uses hardware virtualisation extensions (Intel VT-x or AMD-V) to execute guest code directly on the CPU. It is implemented as a user-space program (`/usr/sbin/bhyve`) and a kernel module (`vmm.ko`). The kernel module handles the low-level virtualisation primitives: VM entry and exit, page table management for guest memory, virtual APIC emulation, and a handful of performance-critical device backends. The user-space program handles the rest: command-line parsing, emulated device backends that do not need to live in the kernel, VirtIO backends, and the main VCPU loop.

From a driver author's perspective, three things matter about `bhyve(8)`.

First, `bhyve(8)` is a FreeBSD program, so the same kernel that runs your driver might also be running `bhyve(8)` in user space. That means host-side resources (memory, CPU, network interfaces) can be competing with `bhyve(8)` for allocation. If your driver is running on a host that is also a hypervisor, you may want to think about NUMA placement, IRQ affinity, and similar concerns.

Second, `bhyve(8)` uses a FreeBSD kernel interface called `vmm(4)` for its low-level needs. This interface is stable but niche; most driver authors never touch it directly. If you are writing a driver that needs to interact with virtual machines (for example, a driver that provides a paravirtual device to `bhyve(8)` guests from the host side), you would use `vmm(4)` or one of the higher-level libraries that wraps it.

Third, and most importantly for this chapter, `bhyve(8)` can assign a real PCI device directly to a guest. This is called PCI passthrough, and it has important implications for driver authors on both sides of the fence.

30.8.2 `vmm(4)`: The Kernel Side of `bhyve`

`vmm(4)` is a kernel module that exposes an interface for creating and managing virtual machines. It lives at `/usr/src/sys/amd64/vmm/` and related directories. The module is loaded on demand by `bhyvectl(8)` or `bhyve(8)`, and it exports a character device interface through `/dev/vmm/NAME` where `NAME` is the name of the virtual machine.

The `vmm(4)` interface is not something a beginner driver author needs to learn in depth. It is complex, specialised, and mostly of interest to people who are extending or modifying the hypervisor itself. For our purposes, it is enough to know the following. `vmm(4)` manages the virtual CPU state, including registers, page tables, and interrupt controllers. It hands off emulation of devices that are not performance-critical to user space, via a ring buffer interface. For devices that are performance-critical, such as the in-kernel virtual APIC or the virtual IOAPIC, it handles emulation in the kernel itself.

A driver author who is running inside a `bhyve(4)` guest will never interact with `vmm(4)` directly. The guest kernel sees only the virtualised devices; the mechanism by which they are emulated is invisible. The only observable trace is in `sysctl kern.vm_guest`, which will report `bhyve`.

A driver author who is writing host-side code for `bhyve(8)` will see `vmm(4)` only through the user-space libraries. Most tasks are handled by `libvmmapi(3)`, which wraps the raw `ioctl` interface. Direct `vmm(4)` work is rare outside of `bhyve(8)` development itself.

30.8.3 PCI Passthrough: Giving a Guest a Real Device

The most direct way for a guest to interact with a physical device is for the host to give the guest exclusive access to that device. This is called PCI passthrough, and `bhyve(8)` supports it through the `pci_passthru(4)` facility.

The idea is straightforward, but the mechanics are subtle. A real PCI device is normally claimed by a driver on the host. That driver programs the device, handles its interrupts, and owns its memory-mapped registers. When we do passthrough, we want the guest's driver to do all of that instead. The host must step out of the way, and the hardware must be reconfigured so that the guest's memory addresses map to the device's memory correctly and so that the device's DMA operations go to the guest's memory rather than the host's.

The host steps out of the way by detaching the driver that was attached to the device and attaching a placeholder driver (`ppt(4)`) instead. `ppt` is a minimal driver whose only purpose is to claim the device so that no one else does. Its probe function matches any PCI device whose address matches a pattern specified by the user in `/boot/loader.conf`, typically using the `pptdevs` tunable. Once the placeholder driver claims the device, `bhyve(8)` can request a passthrough through the `vmm(4)` interface, and the device becomes accessible inside the guest.

The hardware reconfiguration is handled by the IOMMU. This is the part that makes passthrough both capable and dangerous. Without an IOMMU, DMA from the device would go to physical addresses on the host's memory bus, and a misbehaving guest could program the device to read or write anywhere in host memory. That is obviously unsafe. An IOMMU (Intel VT-d or AMD-Vi on the platforms FreeBSD supports) sits between the

device and the host memory bus, remapping device-issued addresses so that they cannot escape the guest. From the device's perspective, it still does DMA to an address; from the host memory bus's perspective, that address is translated to somewhere inside the guest's memory and nowhere else.

If your host has no IOMMU, `bhyve(8)` will refuse to set up passthrough. This is an intentional safety check. Enabling passthrough without IOMMU protection would be like handing a stranger a signed blank cheque on the host kernel: one bug in the device firmware or one malicious guest, and the host is compromised.

30.8.4 How Passthrough Looks to the Guest Driver

From the guest's perspective, a PCI passthrough device looks exactly like the real hardware. The guest's PCI enumeration finds the device, with its real vendor and device IDs, its real capabilities, and its real BARs. The guest's driver attaches just as it would on bare metal. The read and write operations hit the real hardware (with some address translation in between). Interrupts are delivered to the guest through the hypervisor's virtual interrupt controller. DMA works, though the addresses the guest programs into the device are guest-physical addresses, not host-physical addresses, with the IOMMU taking care of the translation.

This has three practical consequences for a driver author.

First, your driver does not need to know it is running under passthrough. The same driver binary works on bare metal and in a passthrough guest. This is a major design goal of the whole scheme.

Second, if your driver uses DMA, make sure you are using `bus_dma(9)` properly. The bus-DMA framework handles the guest-physical versus host-physical translation transparently if you use it correctly. If you are doing clever things with physical addresses directly (which you should not be), passthrough will break those things.

Third, if your driver relies on platform-specific features (for example, special firmware on the PCI device itself, or a particular BIOS table), those features must be present in the guest too. Passthrough gives the guest the device, but it does not give it the firmware or the BIOS tables. Some passthrough setups fail because the driver assumes the presence of an ACPI table that exists on the host but not inside the guest's virtual BIOS.

The last point matters especially for devices that expect strict ordering or uncached memory access. The bus-DMA and bus-space interfaces, used correctly, handle these cases. Direct pointer manipulation to mapped memory usually does not survive passthrough.

30.8.5 Host-Side Driver Attach Under `bhyve`

When `bhyve(8)` is running on a FreeBSD host, two categories of device exist. Some devices are claimed by host drivers and shared with guests through emulation or `VirtIO`. Others are claimed by `ppt(4)` and passed through entirely.

If you are writing a driver for a device that might be used under passthrough, there are a few things to consider.

The first is whether the device should even allow passthrough. Some devices are fundamental to the host's operation (for example, the SATA controller that the host boots from, or the network interface the host is reachable on). These devices should not be marked as candidates for passthrough, because taking them away from the host would break the host. The mechanism for marking candidates is administrative, through `pptdevs` in `loader.conf`, and the host administrator is responsible. There is no per-driver lockout, but a driver author can document clearly whether passthrough is recommended for the device.

The second is whether the driver releases the device cleanly at detach time. Passthrough requires the original driver to detach, and then `ppt(4)` to attach. If your driver's `DEVICE_DETACH` method is sloppy, passthrough setup will be unstable. The detach code must stop the hardware cleanly, release IRQs, unmap resources, and free any memory that the hardware might touch. Anything that persists after detach is a risk.

The third is whether the driver tolerates being re-attached after a passthrough use. When the guest shuts down and releases the device, the host administrator might want to rebind the device to the original driver for use on the host. The driver should be able to attach to a device that was previously owned by `ppt(4)`, even though the device may have been reset and reconfigured by the guest. This means the driver's `DEVICE_ATTACH` method should not assume anything about the device's initial state; it should program the device from scratch, just as it does on first boot.

None of these are new requirements. They are all things a well-written driver does anyway. Passthrough just makes the habits more important, because the cost of getting them wrong shows up immediately.

30.8.6 IOMMU Groups and the Reality of Partial Isolation

A brief word on IOMMU groups, which come up sometimes in passthrough discussions. An IOMMU cannot always isolate a single device from another device on the same PCI bus. When two devices share a bus or a bridge without ACS (Access Control Services), the IOMMU treats them as a single group, because it cannot guarantee that one cannot see the other's DMA. FreeBSD's `dmr(4)` (Intel) and `amdvi(4)` (AMD) drivers handle this grouping internally, but the administrator sometimes has to pass through a whole group rather than a single device.

For a driver author, the practical implication is that passthrough sometimes grabs more than the device you expected. If your device sits behind a bridge shared with another device, turning on passthrough for your device may pull the other one into the guest too. The solution is usually to place devices on separate bridges in the firmware configuration, but that is an administrator concern, not a driver concern. Knowing it exists helps when diagnosing unexpected behaviour.

30.8.7 Host-Only Considerations: Memory, CPU, and the Hypervisor

A FreeBSD host that runs `bhyve(8)` guests has a few extra responsibilities beyond the usual. Memory for the guest's RAM is allocated from the host's physical memory, so a host running many guests needs correspondingly more memory. Guest VCPUs are backed by host threads, so a host running many guests needs to provision CPU capacity. And the hypervisor itself uses a small amount of memory and CPU for its bookkeeping.

A driver author on the host side does not need to manage these resources directly. They are managed by `bhyve(8)` and the host administrator. But there are two ways in which a host-side driver can interact with them.

The first is when a driver offers a device backend that `bhyve(8)` consumes. An example would be a storage driver that provides the backing store for a guest's virtual disk. If the host-side driver is slow, the guest feels it as slow disk. If the host-side driver consumes too much memory, the host runs out of memory for guests. This is a classic shared-resource problem, and it is usually solved by provisioning rather than by clever code.

The second is when a driver interacts with the hypervisor through `vmm(4)` or similar. Paravirtual device backends sometimes use kernel-side hooks to deliver notifications to guests more efficiently than going through user

space. These are rare and advanced, and outside the scope of this chapter. They are mentioned here so you are not surprised if you see them referenced later.

30.8.8 Cross-Platform Notes: bhyve on arm64

bhyve(8) runs on amd64 and, increasingly, on arm64. The arm64 port uses the ARMv8 virtualisation extensions (EL2) rather than Intel VT-x or AMD-V. The SMMU (System Memory Management Unit) takes the role of the IOMMU. The user-space interface is the same, the guest's VirtIO experience is the same, and from a driver author's perspective, the two architectures are interchangeable. The distinction matters only for the low-level hypervisor code inside vmm(4).

For a book about driver authorship, the lesson is the one we have already learned in Chapter 29: write clean, bus-dma-using, endian-correct code, and you will not notice which architecture you are running on. The hypervisor story is another good reason to follow those habits.

30.8.9 Inspecting bhyve From the Host

A driver author can inspect bhyve(8) state in a few useful ways without getting into vmm(4) internals.

bhyvectl(8) is the command-line tool for querying and controlling virtual machines. `bhyvectl --vm=NAME --get-stats` shows counters maintained by vmm(4) for a running guest, including VM exit counts, emulation counts, and similar diagnostics. This is useful when you suspect that a guest driver is generating unnecessary VM exits (a common performance pitfall).

`pciconf -lvBb` on the host shows the PCI devices and their current driver bindings. A device bound to `ppt(4)` is visible in passthrough mode; a device bound to its native driver is available to the host. This is a quick way to see what is passed through and what is not.

`vmstat -i` on the host shows interrupt counts per device. If a device is passed through to a guest, its interrupts are delivered to the guest's virtual interrupt controller, not to the host. On the host side, you will see the hypervisor's posted-interrupt or interrupt-remapping counts increase instead. This is a subtle but useful diagnostic.

None of this is required reading for driver authorship. It is mentioned here so that when you encounter bhyve(8) on a host while debugging a driver, you know where to look first.

30.8.10 A Typical bhyve Command Line

For readers who want to see what bhyve(8) actually looks like when invoked without a wrapper tool, here is a representative command line. It starts a guest named `guest0` with two VCPUs, two gigabytes of memory, a virtio-blk disk backed by a file, and a virtio-net network interface bridged to the host:

```
bhyve -c 2 -m 2G \  
  -s 0,hostbridge \  
  -s 2,virtio-blk,/vm/guest0/disk0.img \  
  -s 3,virtio-net,tap0 \  
  -s 31,lpc \  
  -l com1,stdio \  
  guest0
```

Each `-s` flag defines a PCI slot. Slot zero is the host bridge; slot two is a virtio-blk device backed by a disk image file; slot three is a virtio-net device backed by a `tap(4)` interface the host has configured; slot thirty-one is the LPC bridge used for legacy devices like the serial console. The `-l com1, stdio` redirects the guest's first serial port to the host's standard I/O, which is convenient for console access.

When this command runs, the host kernel creates a new VM through `vmm(4)`, allocates memory for the guest's RAM, and hands off VCPU execution to the guest's kernel. The virtio-blk backend in `bhyve(8)` (user-space code) services the guest's block requests by reading and writing the disk image file. The virtio-net backend sends and receives packets through the `tap0` interface, which the host's network stack handles as an ordinary interface.

A driver running inside this guest sees a PCI virtio-blk device, a PCI virtio-net device, and the usual assortment of emulated LPC devices. From the driver's perspective, there is no clue that the "hardware" is implemented by a user-space program running a few milliseconds away on the same host. The abstraction is, by design, complete.

30.8.11 Inside the `vmm(4)` Module

`vmm(4)` is the kernel-side infrastructure for `bhyve(8)`. In FreeBSD 14.3, its main source files live in `/usr/src/sys/amd64/vmm/`; an arm64 port is under active development and is expected to ship in a later release, so if you are on FreeBSD 14.3 the amd64 tree is the one to read. The module exports a small control interface through `/dev/vmmctl` and a per-VM interface through `/dev/vmm/NAME`.

The control interface is used by `bhyvectl(8)` and `bhyve(8)` to create, destroy, and enumerate virtual machines. The per-VM interface is used to read and write VCPU state, map guest memory, inject interrupts, and receive VM-exit events from the guest.

When a guest's VCPU executes code that requires emulation (reading from an I/O port, accessing a memory-mapped device register, executing a hypercall), the hardware traps the execution and the control returns to `vmm(4)`. The module either handles the exit in the kernel (for a small set of performance-critical cases, like reading the local APIC) or forwards it to `bhyve(8)` in user space (for most cases, including all VirtIO device emulation).

The kernel/user-space split is a deliberate design choice. Keeping the code in user space makes it easier to develop, debug, and audit. Keeping performance-critical paths in the kernel keeps the overhead low. For FreeBSD, the split has worked well, and `bhyve(8)` has grown from a small academic prototype to a production-quality hypervisor.

For a driver author who is not extending `bhyve(8)` itself, none of this matters in detail. What matters is the observable behaviour: a guest executes code, some operations trap, the hypervisor emulates them, and the guest continues. The driver in the guest sees consistent behaviour regardless of where the emulation happens.

30.8.12 Nested Virtualisation

A brief mention of nested virtualisation, since it comes up. FreeBSD's `bhyve(8)` on amd64 currently does not support running hardware-virtualised guests inside hardware-virtualised guests (no `VIRTUAL_VMX` or `VIRTUAL_SVM` extensions yet). If you try to run `bhyve(8)` inside a `bhyve(8)` guest, the inner hypervisor will fail

to initialise. Intel and AMD both support nested virtualisation in hardware, but the FreeBSD implementation has not yet enabled it.

This matters for labs only in the sense that you must run `bhyve(8)` on bare metal (or on a host that provides nested virtualisation, which some cloud platforms do). If your lab machine is itself a VM, you may find that `bhyve(8)` either fails to load `vmm.ko` or loads it but refuses to start guests.

On arm64, nested virtualisation is a different story; the ARM architecture has cleaner support, and the FreeBSD arm64 port is moving toward enabling it. For up-to-date information, consult the `bhyve(8)` and `vmm(4)` manual pages on the version of FreeBSD you are running.

30.8.13 An Example: Debugging a PCI Passthrough Driver

To make the host-side discussion concrete, consider a scenario that combines several of the ideas above. You have a PCI network card that your host's driver knows how to drive. You want to pass it through to a `bhyve(8)` guest and test that the same driver works unchanged in the guest.

Step 1: Identify the device. `pciconf -lvBb` shows `em0@pci0:2:0:0` with the driver `em` attached. The device is an Intel Gigabit Ethernet controller.

Step 2: Mark it for passthrough. Edit `/boot/loader.conf` and add `pptdevs="2/0/0"`. Reboot.

Step 3: Verify. After reboot, `pciconf -l` shows `ppt0@pci0:2:0:0`, with `ppt` (the placeholder driver) attached instead of `em`. The device is no longer available to the host's network stack.

Step 4: Configure the guest. In the guest's `bhyve` configuration, add `-s 4,passthru,2/0/0`. This tells `bhyve(8)` to pass through the device to the guest at PCI slot 4.

Step 5: Start the guest. Inside the guest, run `pciconf -lvBb`. The device appears, with its real Intel vendor and device IDs, attached to `em`. Check `dmesg`. The guest's `em` driver has attached to the passthrough device exactly as it would on bare metal.

Step 6: Exercise the device. Configure the interface (`ifconfig em0 10.0.0.1/24 up`), send traffic, verify it works.

Step 7: Shut down the guest. Back on the host, decide whether to keep the device in passthrough mode or return it to the host. If you want it back, edit `/boot/loader.conf` to remove `pptdevs`, reboot, and verify that `em` is attached again.

Every step in this workflow is something the administrator does; the driver itself is untouched. That is the point. If the driver is written correctly (with `bus_dma`, clean detach, no hidden platform assumptions), it works in both environments without changes.

30.8.14 Wrapping Up

The host side of virtualisation is where FreeBSD plays a slightly different role. Instead of writing a driver that consumes a hypervisor-provided device, you may find yourself writing (or at least interacting with) the infrastructure that provides devices to a hypervisor. `bhyve(8)`, `vmm(4)`, and `pci_passthru(4)` are the main interfaces to know about. PCI passthrough is the most relevant of these for a driver author, because it exercises the detach-and-reattach lifecycle that a well-written driver already supports.

With guests and hosts both covered, the next major environment in FreeBSD's virtualisation story is the one that does not use a separate kernel at all: jails. Section 6 turns to them, to devfs, and to the VNET framework that extends the jail model into the network stack.

30.9 Section 6: Jails, devfs, VNET, and Device Visibility

Virtualisation and containerization share a goal: they both let one physical machine host several workloads that appear, to their users, to be running on separate machines. The mechanisms they use are dramatically different. A virtual machine runs a complete guest kernel on top of a hypervisor; it has its own memory map, its own device tree, its own everything. A container, in the FreeBSD sense, shares the host's kernel entirely. What it has of its own is a filesystem view, a process table, and (if the administrator sets it up that way) a network stack. FreeBSD's answer to the container question is the jail, and jails have existed in some form since FreeBSD 4.0. For driver authors, jails matter because they change what a process can see and do with respect to devices, without changing the driver code at all.

This section explains how jails interact with devices. It focuses on four topics: the jail model itself, the devfs ruleset system that controls device visibility, the VNET framework that gives jails their own network stacks, and the question every container system eventually has to answer: which processes can reach which drivers.

30.9.1 What a Jail Is, and What It Is Not

A jail is a subdivision of the host kernel's resources. A process running inside a jail sees a subset of the filesystem (rooted at the jail's root directory), a subset of the process table (only processes in the same jail), a subset of the network (depending on whether the jail has its own VNET), and a subset of the devices (depending on the jail's devfs ruleset). The host kernel is a single, shared kernel; there is no separate guest kernel inside the jail. System calls made inside the jail are executed by the same kernel that executes calls from outside the jail, with jail-specific checks inserted at the right places.

Because the kernel is shared, drivers are shared too. There is no separate copy of a driver inside each jail; there is only the one driver that the kernel loaded at boot time. The jail just controls which of the driver's devices are visible to the jail's processes. A jail that has not been configured to see `/dev/null` will see no `/dev/null`; a jail that has been configured to see `/dev/null` will see the same `/dev/null` that the host sees. No new driver instance; no new softc; just a visibility rule.

This simplicity is both jails' great strength and their great limitation. It means jails are extremely cheap: starting a jail is roughly the cost of running a few system calls, whereas starting a VM is the cost of booting a whole kernel. It also means jails cannot isolate kernel-level failures: a driver bug that panics the host kernel panics every jail running on it. A jail is a policy boundary, not a crash boundary. For many workloads that trade-off is excellent; for others, a VM is the right answer.

30.9.2 The Kernel's View of Jails

Inside the kernel, a jail is represented by a struct `prison`, defined in `/usr/src/sys/sys/jail.h`. Each process has a pointer to the prison it belongs to, via `td->td_ucred->cr_prison`. Code that wants to check whether a process is inside a jail can compare this pointer against the global `prison0`, which is the root jail (the host itself). If the pointer is `prison0`, the process is on the host; otherwise, it is in some jail.

Several helper functions exist for the common checks a driver might want to do. `jailed(cred)` returns true if the credential belongs to a jail other than `prison0`. `prison_check(cred1, cred2)` returns zero if two credentials are in the same jail (or one is the parent of the other); it returns an error otherwise. `prison_priv_check(cred, priv)` is how privilege checks are extended for jails: a root user inside a jail does not have all privileges that root has on the host, and `prison_priv_check` implements the reduction.

A driver author will usually not need to call any of these directly. The framework calls them on your behalf. When a process opens a `devfs` node, for example, the `devfs` layer consults the jail's `devfs` ruleset before handing the file descriptor over. When a process tries to use a privilege-gated feature (like `bpf(4)` or `kldload(2)`), the privilege check goes through `prison_priv_check`. A driver only needs to be aware that these checks exist, and to call the framework helpers correctly when it defines its own access rules.

30.9.3 devfs: The Filesystem Through Which Devices Are Exposed

In FreeBSD, devices are exposed through a filesystem called `devfs(5)`. Every entry under `/dev` is a `devfs` node. A driver that calls `make_dev(9)` or `make_dev_s(9)` creates a `devfs` node; the name, permissions, and uid/gid are attributes of that node. The node is visible in every `devfs` instance that the kernel mounts, and FreeBSD mounts one `devfs` instance per filesystem view: one for the host's `/dev`, one for each jail's `/dev` (if the jail has its own `/dev`), and one per `chroot` that mounts its own `devfs`.

This is the first part of the visibility story. Each jail (more precisely, each filesystem view) has its own `devfs` mount, and the kernel can apply different rules to different mounts. The rules are called `devfs` rulesets, and they are the main tool for controlling which devices a jail can see.

30.9.4 devfs Rulesets: Declaring What a Jail Can See

A `devfs` ruleset is a numbered set of rules stored in the kernel. Rules can hide nodes, reveal nodes, change their permissions, or change their ownership. The ruleset is applied to a mounted `devfs` instance; every time a lookup happens in that instance, the kernel walks the ruleset and applies the matching rule to each node.

On a fresh FreeBSD system, four rulesets are predefined in `/etc/defaults/devfs.rules` (which is processed when the kernel starts). The file uses `devfs(8)` syntax, a small declarative language for ruleset construction. Let us look at a representative slice.

```
[devfsrules_hide_all=1]
add hide

[devfsrules_unhide_basic=2]
add path log unhide
add path null unhide
add path zero unhide
add path crypto unhide
add path random unhide
add path urandom unhide

[devfsrules_jail=4]
```

```
add include $devfsrules_hide_all
add include $devfsrules_unhide_basic
add include $devfsrules_unhide_login
add path zfs unhide

[devfsrules_jail_vnet=5]
add include $devfsrules_hide_all
add include $devfsrules_unhide_basic
add include $devfsrules_unhide_login
add path zfs unhide
add path 'bpf*' unhide
```

Rule 1, `devfsrules_hide_all`, hides everything. On its own it is useless, because a `devfs` mount with nothing visible is not helpful. It is the starting point for other rulesets.

Rule 2, `devfsrules_unhide_basic`, unhides a small set of essential devices: `log`, `null`, `zero`, `crypto`, `random`, `urandom`. These are the devices that essentially every program needs; without them, even basic tools fail.

Rule 4, `devfsrules_jail`, is the ruleset intended for non-VNET jails. It starts by including `devfsrules_hide_all` (so everything is hidden), then layers `devfsrules_unhide_basic` on top (so the essentials are visible), and adds ZFS devices. The result is a jail that sees a small, safe set of devices and nothing else.

Rule 5, `devfsrules_jail_vnet`, is the equivalent for VNET jails. It is the same as rule 4 with the addition that `bpf*` devices are unhidden, because a VNET jail might legitimately need `bpf(4)` (for tools like `tcpdump(8)` or `dhclient(8)`).

When creating a jail, the administrator specifies which ruleset to apply to the jail's `/dev` mount, either through `jail.conf(5)` (`devfs_ruleset = 4`) or on the command line (`jail -c ... devfs_ruleset=4`). The kernel applies the ruleset to the mount, and the jail sees only what the ruleset allows.

30.9.5 Creating a Custom `devfs` Ruleset

For most jails, the default rulesets are enough. When they are not, an administrator can define new ones. A new ruleset must have a unique number (other than the reserved defaults) and can be constructed by inclusion, addition, and override.

A classical example is a jail that needs one specific device that the default rules hide. Suppose we have a jail that runs a service that needs `/dev/tun0`, and we want to expose it without opening the whole `/dev/tun*` family. We would create a ruleset like:

```
[devfsrules_myjail=100]
add include $devfsrules_jail
add path 'tun0' unhide
```

And apply it in `jail.conf(5)`:

```
myjail {  
    path = /jails/myjail;  
    devfs_ruleset = 100;  
    ...  
}
```

The `devfs(8)` tool can load this ruleset into the running kernel with `devfs rule -s 100 add ...`, or the administrator can edit `/etc/devfs.rules` and restart `devfs`. For persistent configuration, the file is the right place.

30.9.6 What devfs Rulesets Do Not Do

It is worth noting what devfs rulesets are not. They are not a capability system. Hiding a device from a jail means the jail cannot open that specific path, but a jail that has the `allow.raw_sockets` privilege can still send arbitrary raw packets, ruleset or no ruleset. Hiding `/dev/kmem` does not prevent a determined attacker with the right privileges from reading kernel memory via other means; it just removes one obvious path.

Rulesets are a visibility policy, layered on top of the standard UNIX permissions model. The UNIX permissions still apply: a file hidden by the ruleset is not openable, but a file visible to the ruleset still respects its own permissions. A jail's root user can open `/dev/null` because the ruleset says so and the permissions say so, not because the ruleset alone grants access.

For strong isolation, combine rulesets with privilege restrictions (`allow.*` parameters in `jail.conf(5)`) and, if the workload warrants it, with a VM. Rulesets alone are one layer of defence, not the only one.

30.9.7 A Driver Author's View of devfs Rulesets

For a driver author, devfs rulesets matter for two reasons.

First, when you create a devfs node with `make_dev(9)`, you pick a default owner, group, and permission. These apply to every devfs view that the node appears in. If your device is something that jails should generally not see (for example, a low-level hardware-management interface), consider whether the name should be obvious (so administrators can easily write a rule that hides it) or whether it should be under a subdirectory (so a single rule can hide the whole subdirectory).

Second, if your driver's device is something that jails commonly need, document that fact in your driver's manual page. The administrator who writes a jail's ruleset is usually not the person who wrote the driver, and they need to know whether to unhide your node. A line like "This device is typically used inside jails; unhide it with `add path mydev unhide` in the jail's ruleset" is very helpful.

Neither of these is a code change. Both are decisions about naming and documentation. Driver authorship is not just about code; it is also about making the code usable by the administrators who will deploy it.

30.9.8 Wrapping Up the Jail and devfs Side

Jails are the lightweight containerization mechanism in FreeBSD. They share the host kernel, and thus share drivers, but they control which devices are visible through the devfs ruleset mechanism. For a driver author,

the relevant design decisions are at the naming and documentation level: pick names that make it easy to write rules, and document which jails should see the device.

The network side of jails is a story of its own, because FreeBSD has two models: single-stack jails that share the host's network, and VNET jails that have their own network stacks. The VNET model is the more interesting one for driver authors, because it has direct consequences for how network drivers are assigned and moved. We turn to that next.

30.9.9 The VNET Framework: One Kernel, Many Network Stacks

The default jail model has one network stack: the host's. Every jail sees the same routing table, the same interfaces, the same sockets. A jail can be restricted to particular IPv4 or IPv6 addresses (via `ip4.addr` and `ip6.addr` in `jail.conf(5)`), but it cannot have a genuinely independent network configuration. That is often enough for simple workloads, but it is not enough for any jail that wants to run its own firewall, use its own default gateway, or be reached by the outside world through a unique set of addresses on its own interfaces.

VNET (short for “virtual network stack”) solves this. It is a kernel feature that replicates the parts of the network stack per jail, so each VNET jail sees its own routing table, its own interface list, its own firewall state, and its own socket namespace. The stack code still belongs to the same kernel, but many of its global variables are now per-VNET instead of truly global. A network driver's per-interface state belongs to whichever VNET it is currently assigned to, and interfaces can be moved from one VNET to another.

For driver authors, VNET is interesting in three respects. It changes how global state in network subsystems is declared. It adds a lifecycle to interfaces: interfaces can be moved, and drivers must support the move cleanly. And it interacts with jail creation and destruction through VNET-specific hooks.

30.9.10 Declaring VNET State: `VNET_DEFINE` and `CURVNET_SET`

VNET's design puts the work on whoever declares global state in a VNET-aware subsystem. Instead of a plain `static int mysubsys_count;`, a VNET-aware declaration looks like:

```
VNET_DEFINE(int, mysubsys_count);  
#define V_mysubsys_count VNET(mysubsys_count)
```

The `VNET_DEFINE` macro expands to a storage declaration that places the variable in a special section of the kernel. At VNET creation time, the kernel allocates a new per-VNET region of memory and initialises it from the section. The `VNET( . . . )` macro, used via a short alias like `V_mysubsys_count`, resolves to the correct copy for the current VNET.

“The current VNET” is thread-local context. When a thread enters code that operates on a VNET, it calls `CURVNET_SET(vnet)` to establish the context, and `CURVNET_RESTORE()` to tear it down. Inside the context, `V_mysubsys_count` resolves to the right instance. Outside the context, accessing `V_mysubsys_count` is a bug; the macro depends on the thread-local current-VNET pointer, and without that pointer set the result is undefined.

Most driver authors do not need to write `VNET_DEFINE` declarations themselves. The network stack and the `ifnet` framework declare their own VNET state. Driver-level state (per-interface softcs, per-hardware private

data) is not usually VNET-scoped, because it is tied to the hardware, not to the network stack. The driver's state lives wherever the driver put it, and the framework takes care of moving the right bits between VNETs when interfaces move.

What driver authors do need to do is wrap any code that touches network-stack objects in a `CURVNET_SET` / `CURVNET_RESTORE` pair if that code is called from outside a network-stack entry point. Most driver code is called from the network stack already, so the VNET context is already set. The exception is callouts and taskqueues: a callback fired from a callout does not inherit a VNET context, and the driver must establish one before touching any `v_` variable.

A typical pattern inside a callout handler:

```
static void
mydev_callout(void *arg)
{
    struct mydev_softc *sc = arg;

    CURVNET_SET(sc->ifp->if_vnet);
    /* code that touches network-stack variables */
    CURVNET_RESTORE();
}
```

The driver stored a reference to the interface's VNET on attach (`sc->ifp->if_vnet` is filled in by the framework when the ifnet is created). On every callout, it establishes the context, does its work, and restores. This is one of the few places where driver authors encounter VNET directly.

30.9.11 `if_vmove`: Moving an Interface Between VNETs

When a VNET jail starts, it typically is given one or more network interfaces to use. There are two common mechanisms. The first is that the administrator creates a virtual interface (an `epair(4)` or `vlan(4)`) and moves one end into the jail. The second is that the administrator moves a physical interface into the jail outright, so that the jail has exclusive access to it while it runs.

The move is implemented by the kernel function `if_vmove()`. It takes an interface and a destination VNET, detaches the interface from the source VNET's network stack (without destroying it), and reattaches it to the destination VNET's network stack. The interface retains its driver, its `softc`, its hardware state, and its configured MAC address. What changes is which VNET's routing table, firewall, and socket namespace it is attached to.

For a driver author, the move imposes a lifecycle requirement. The interface must be able to survive being detached from one VNET and attached to another. The driver's `if_init` function may be called again in the new context. The driver's `if_transmit` function may receive packets from the new VNET's sockets. Any state the driver caches about the "current" network stack (for example, routing-table lookups) must be invalidated or re-established.

For a network driver written to the standard `ifnet(9)` interface, the move usually works without special handling. The ifnet framework does the heavy lifting, and the driver is largely unaware. What the driver must avoid is holding references to VNET-scoped state across entry points. Code like "grab a pointer to the current

routing table at attach and cache it” does not survive an interface move, because the interface may later belong to a different VNET with a different table.

A related primitive, `if_vmove_loan()`, is used for interfaces that should return to the host when the jail shuts down. The jail gets the interface on a loan basis, and on jail destruction the interface is moved back. This is common for `epair(4)` setups where the physical connection (if any) belongs to the host and only the logical presence belongs to the jail.

30.9.12 VNET Lifecycle Hooks

When a VNET is created or destroyed, subsystems that keep VNET state need to initialise it or release it. The `VNET_SYSINIT` and `VNET_SYSUNINIT` macros register functions to be called at those moments. A network protocol might register an `init` function that creates per-VNET hash tables, and an `uninit` function that destroys them.

Driver authors rarely need these hooks. They are relevant to protocols and stack features, not to device drivers. They are mentioned here because you will see them scattered across the network stack code, and knowing that they are VNET lifecycle hooks helps you read the source.

30.9.13 A Concrete VNET Pattern

To make the VNET abstractions concrete, consider a simplified pseudo-driver that keeps a counter of packets received per VNET. The counter must be per-VNET because the same pseudo-driver might be cloned into several VNETs simultaneously, and each clone should have its own count.

The declaration looks like:

```
#include <net/vnet.h>

VNET_DEFINE_STATIC(uint64_t, pseudo_rx_count);
#define V_pseudo_rx_count VNET(pseudo_rx_count)

/* Optional init for the counter */
static void
pseudo_vnet_init(void *unused)
{
    V_pseudo_rx_count = 0;
}
VNET_SYSINIT(pseudo_vnet_init, SI_SUB_PSEUDO, SI_ORDER_ANY,
    pseudo_vnet_init, NULL);
```

The `VNET_DEFINE_STATIC` places the counter in a per-VNET section of the kernel image. When a new VNET is created, the kernel copies the per-VNET section into fresh memory, so each VNET starts with its own zero-initialised copy of the counter. The `V_pseudo_rx_count` shorthand is a macro that expands to `VNET(pseudo_rx_count)`, which in turn dereferences the current VNET’s storage.

When a packet arrives, the receive path increments the counter:

```
static void
pseudo_receive_one(struct mbuf *m)
{
    V_pseudo_rx_count++;
    /* deliver packet to the stack */
    netisr_dispatch(NETISR_IP, m);
}
```

This looks like ordinary code, because the macro hides the per-VNET indirection. The condition for it to be correct is that the thread is already in the right VNET context when `pseudo_receive_one` is called. In a network driver's receive path that condition is automatic: the network stack calls the driver's entry point with the right context already established.

When the counter is accessed from an unusual context, the context must be established explicitly:

```
static void
pseudo_print_counter(struct vnet *vnet)
{
    uint64_t count;

    CURVNET_SET(vnet);
    count = V_pseudo_rx_count;
    CURVNET_RESTORE();
    printf("pseudo: vnet %p has received %lu packets\n", vnet, count);
}
```

Here the function is called from some administrative path that does not know the current VNET, so it sets the context manually, reads the counter, restores the context, and prints the result. This is the pattern you will see repeated in VNET-aware code.

30.9.14 Reading Real VNET Code

If you want to see the pattern in a real driver, `/usr/src/sys/net/if_tuntap.c` is a good starting point. The `tun` and `tap` cloning drivers are VNET-aware: each clone belongs to one VNET, and creating or destroying clones respects the VNET boundaries. The code is well-commented and small enough to read in a couple of evenings.

Two patterns in `if_tuntap.c` are worth noticing. The first is the use of `V_tun_cdevsw` and `V_tap_cdevsw`, per-VNET character device switch structures. Each VNET has its own copy of the switch, so `/dev/tun0` in one VNET can map to a different underlying clone than `/dev/tun0` in another VNET. This is the kind of fine-grained per-VNET duplication that the framework enables.

The second is the use of `if_clone(9)` with VNET. The `if_clone_attach` and `if_clone_detach` functions take VNET into account automatically, so a clone created in a VNET lives in that VNET until it is explicitly moved or destroyed. The cloner does not need to carry VNET state in its `softc`; the framework handles it.

Studying these patterns makes the text in this chapter concrete. Read, take notes, and come back to the text if anything is unclear.

30.9.15 Hierarchical Jails

A brief mention of hierarchical jails, which are a feature some readers will encounter. FreeBSD supports nesting jails: a jail can create child jails, and the child jails are bounded by the parent jail's restrictions. This is useful for services that want to further subdivide their environment.

From a driver author's perspective, hierarchical jails do not introduce new APIs. The `prison_priv_check` helper walks the hierarchy automatically: a privilege is granted only if every level of the hierarchy allows it. A driver that uses the framework correctly works in hierarchical jails without additional code.

The administrative side is more complex (the parent jail must allow child-jail creation, the children inherit a restricted set of privileges), but the driver-side does not need to care. Knowing that the feature exists helps when you see nested jails in a deployment.

30.9.16 Putting It All Together

A FreeBSD system with jails and VNET is a system where a single kernel serves many isolated environments. Each environment sees its own filesystem view, its own process table, its own devices (filtered by devfs ruleset), and possibly its own network stack (under VNET). The driver serving all of them is a single shared binary, but it respects the isolation because it calls the framework APIs correctly.

The framework APIs for this isolation, `priv_check`, `prison_priv_check`, `CURVNET_SET`, `VNET_DEFINE`, and the cloning helpers, are small and self-contained. A driver author who learns them once can write drivers that work correctly in every jail configuration the administrator can dream up. There is no need to special-case specific jail setups; the framework does that work.

30.9.17 Single-Stack Jails and the In-Between

Not every jail needs VNET. A jail that is running a web server that talks through a reverse proxy on the host may do perfectly well with a single-stack jail bound to a specific IPv4 address. The main cost of VNET is complexity in the stack (every protocol must be VNET-aware) and some memory overhead (each VNET has its own hash tables, caches, and counters). For lightweight jails, the single-stack model is often the better choice.

The trade-off for driver authors is worth knowing. A network driver that works correctly in a host-stack jail may still need attention for VNET jails, because the interface can be moved under VNET but not under the single-stack model. Writing the driver with VNET in mind from the start is the right approach; the additional discipline is small, and it future-proofs the driver.

30.9.18 Wrapping Up

Jails share the host's kernel and thus the host's drivers. What they do not share is visibility: devfs rulesets control which device nodes a jail can open, and VNET controls which network interfaces a jail can use. Driver authors benefit from understanding both mechanisms, because the design choices they make (how to name devfs nodes,

how to handle VNET context in callouts, how to support interface moves) affect how their driver behaves in jailed environments.

With the jail picture complete, we can now think about the companion question: once a jail has access to a device, what privileges does it have to use that device? Section 7 takes up resource limits and security boundaries, and looks at the other side of the jail policy story.

30.10 Section 7: Resource Limits, Security Boundaries, and Host-Versus-Jail Access

A driver is not an isolated object. It is a consumer of kernel resources and a provider of services to processes, and both relationships are mediated by the kernel's security and accounting frameworks. When a driver runs on a FreeBSD host that contains jails, the security boundary shifts: some privileges that are unconditional on the host are restricted for jail processes, and some resources that are unmetered on a traditional system are now subject to per-jail limits. A good driver author knows where those boundaries are, because their driver's behaviour on a host is not always its behaviour inside a jail.

This section covers three topics. First, the privilege framework and how `prison_priv_check` reshapes it for jails. Second, `rctl(8)` and how resource limits apply to kernel resources a driver might care about. Third, the practical distinction between attaching a driver from inside a jail (which is usually impossible) and making a driver's services available to a jail (which is the usual case).

30.10.1 The Privilege Framework and `prison_priv_check`

FreeBSD uses a privilege system to make fine-grained decisions about what a process can and cannot do. Traditional UNIX has a single privilege bit (root versus not-root), and that bit determines everything. FreeBSD refines this with the `priv(9)` framework, which defines a long list of named privileges. Each privilege covers a specific kind of operation. Loading a kernel module is `PRIV_KLD_LOAD`. Setting a process's root directory is `PRIV_VFS_CHROOT`. Opening a raw socket is `PRIV_NETINET_RAW`. Configuring an interface's MAC address is `PRIV_NET_SETLLADDR`. Using a BPF device for packet capture is `PRIV_NET_BPF`.

A process that is root (uid 0) has all of these privileges on the host. A process that is a jail's root has some of them, but not all. The restriction is handled by `prison_priv_check(cred, priv)`: it takes the credential and the privilege name, and returns zero if the privilege is granted and an error (usually `EPERM`) if it is denied. The kernel's privilege-checking path is structured so that for a jailed credential, `prison_priv_check` is called first; if it denies the privilege, the caller returns `EPERM` without further ado.

Which privileges a jail is allowed to exercise is determined by two things. The first is a hardcoded list inside `prison_priv_check`: some privileges are simply never granted to jails, regardless of configuration. Examples include `PRIV_KLD_LOAD` (loading kernel modules) and `PRIV_IO` (I/O port access). The second is the `allow.*` parameters in `jail.conf(5)`, which turn on or off specific categories. `allow.raw_sockets` (off by default) controls `PRIV_NETINET_RAW`. `allow.mount` (off by default) controls filesystem mounting privileges. `allow.vmm` (off by default) controls access to `vmm(4)` for running nested hypervisors. The defaults err on the side of denial: if you do not explicitly allow it, the jail does not get it.

For a driver author, the privilege framework matters whenever the driver does something that a process might or might not be allowed to do. A driver that implements a low-level hardware interface might require `PRIV_DRIVER`

(the catch-all for driver-specific privilege checks) or a more specific privilege. A driver that exposes a character device whose `ioctl`s can reconfigure the hardware will call `priv_check(td, PRIV_DRIVER)` (or a more specific name) to decide whether the caller is allowed to do the reconfiguration.

The standard pattern in driver code looks like this:

```
static int
mydev_ioctl(struct cdev *dev, u_long cmd, caddr_t data, int fflag,
            struct thread *td)
{
    int error;

    switch (cmd) {
    case MYDEV_CMD_RECONFIGURE:
        error = priv_check(td, PRIV_DRIVER);
        if (error != 0)
            return (error);
        /* do the reconfiguration */
        break;
    ...
    }
}
```

`priv_check(td, PRIV_DRIVER)` does the right thing for both host and jail callers. On a host, a root process passes; on a host, a non-root process is denied (unless the driver grants permission by other means). Inside a jail, `prison_priv_check` is consulted, and by default `PRIV_DRIVER` is denied inside jails. If the administrator has configured the jail to allow driver access (a very unusual setting), the privilege is granted and the call proceeds.

The result is that a driver that uses `priv_check` correctly gets jail safety for free. The driver does not need to know whether the caller is in a jail; it just asks whether the caller has the right privilege, and the framework takes care of the rest.

30.10.2 Named Privileges Most Relevant to Drivers

A short reference for some of the privileges a driver author will encounter.

`PRIV_IO` is for direct I/O port access on x86. It is defined in `/usr/src/sys/sys/priv.h` and is denied to jails unconditionally. Drivers that offer raw I/O port access to user space are rare (usually limited to legacy hardware like `/dev/io`), but when they exist, they use this privilege.

`PRIV_DRIVER` is the catch-all for driver-specific privileges. If a driver needs to gate an `ioctl` that only an administrator should call, `PRIV_DRIVER` is the default choice.

`PRIV_KMEM_WRITE` gates write access to `/dev/kmem`. Like `PRIV_IO`, it is denied to jails. Writing to kernel memory is the ultimate privileged operation; no reasonable container policy allows it.

`PRIV_NET_*` is a family of network-related privileges. `PRIV_NET_IFCREATE` is for creating network interfaces; `PRIV_NET_SETLLADDR` is for changing MAC addresses; `PRIV_NET_BPF` is for opening a BPF device. Each has

its own jail policy, and the combinations are how a VNET jail can (for example) run `dhclient(8)` (which needs `PRIV_NET_BPF`) without also being able to arbitrarily reconfigure interfaces.

`PRIV_VFS_MOUNT` gates filesystem mounting. Jails have a very restricted version of this by default: they can mount `nullfs` and `tmpfs` if `allow.mount` is set, but not arbitrary filesystems.

A complete list is in `/usr/src/sys/sys/priv.h`. For driver authorship, you will rarely invent new privilege categories; you will pick the existing one that fits.

30.10.3 `rctl(8)`: Per-Jail and Per-Process Resource Limits

Jails (and, indeed, processes) can be subject to resource limits beyond the traditional UNIX `ulimit(1)` model. FreeBSD's `rctl(8)` (the runtime resource control framework) lets an administrator set limits on a wide variety of resources, and enforce them with specified actions when the limits are hit.

The limits cover things like memory use, CPU time, number of processes, number of open files, I/O bandwidth, and so on. They can be applied per user, per process, per login class, or per jail. The typical use in a jail setup is to cap a jail's total memory and CPU so that a misbehaving application inside the jail cannot affect other jails on the same host.

For a driver author, `rctl(8)` matters for a subtle reason. Drivers allocate resources on behalf of processes. When a driver calls `malloc(9)` to allocate a buffer, the memory goes somewhere. When a driver creates a file descriptor by opening a file internally, the descriptor goes somewhere. When a driver spawns a kernel thread, the thread runs as part of somebody's accounting. If the "somebody" is a process inside a jail, the accounting might hit the jail's `rctl` limits.

Usually this is exactly what you want. If a jail is supposed to be limited to 100 MB of memory, and an allocation on behalf of a jail process should count against that limit, and `rctl` hits the limit, the allocation should fail with `ENOMEM`. Your driver then propagates the failure back to user space, and the well-behaved application inside the jail handles it.

Occasionally the accounting is less obvious. A driver that maintains a pool of buffers shared across all callers will, by default, charge the pool to the kernel rather than to any single process. That is fine, but it means the pool is unmetered: a very active jail can consume a larger share of the pool than it "should" under the resource limits. For most drivers this is acceptable, but for drivers whose resources are expensive (large DMA buffers, for instance) it may be worth considering whether the resource should be tracked per-process through `racct(9)`, the underlying accounting layer that `rctl` sits on top of.

The `racct(9)` framework exposes `racct_add(9)` and `racct_sub(9)` functions for drivers that want to participate in accounting. Most drivers never call these directly. Adding `racct` support is a deliberate design choice, usually made when a driver's resource consumption is large enough to matter in aggregate. For everyday character-device drivers or network drivers, the default accounting done by the kernel (per-socket buffer memory, per-process file descriptor counts, and so on) is sufficient.

30.10.4 Enforcement Actions and What They Mean for Drivers

When a resource limit is hit, `rctl(8)` can do one of several things: deny the operation, send a signal to the offending process, log the event, or throttle (for rate-based resources). The enforcement is handled by the

kernel's accounting layer, not by drivers. What drivers see is the result: an allocation fails, a signal delivers, a rate-limited operation takes longer.

For a driver, the practical implication is that every allocation and every resource acquisition must be written to handle failure. This is not specific to jails or `rctl`; it is good defensive coding anyway. A `malloc(9)` with `M_WAITOK` can wait for memory indefinitely, but in a jail with a memory limit, it may still fail (if `M_WAITOK` is not set) or it may block a long time waiting for memory that will never be freed (because no other process in the jail has any to free).

The rule of thumb: if your driver is doing allocations on behalf of a user process, consider whether `M_NOWAIT` is more appropriate than `M_WAITOK`, and whether the caller can tolerate a delayed or failed allocation. Jails (and the resource limits around them) make the consideration more than theoretical.

30.10.5 Host-Side Drivers versus Jail-Side Processes

A recurring question for driver authors is: can my driver be loaded or attached from inside a jail? The short answer is almost always no. Loading kernel modules (`kldload(2)`) requires `PRIV_KLD_LOAD`, which is never granted to jails. A jail that needs access to a driver's services must have that driver already loaded and attached on the host, and then the jail can use the driver through the usual user-space interfaces.

This is a consequence of the single-kernel model. A jail does not have its own kernel, so it cannot load drivers of its own. What it has is access (subject to the devfs ruleset) to the drivers that the host has loaded. In practice this means:

- Driver loading and unloading happen on the host. The host administrator is responsible for `kldload` and `kldunload`.
- Device attachment to devices happens on the host. The driver's `DEVICE_ATTACH` runs in host context, not in jail context.
- Device access from user space happens inside the jail, through `/dev` (if the ruleset allows) and through the driver's `open/read/write/ioctl` methods.

The separation is usually clean. The driver does not need to know whether its caller is in a jail; the kernel handles the context. Where it sometimes matters is in `ioctl` handlers that want to distinguish host and jail callers, or in drivers that allocate resources per-open whose release policy differs.

A specific caveat: when a driver creates state on `open`, that state survives until `close`. If the jail that held the open file descriptor goes away before the descriptor is closed (because the jail was destroyed while processes still had the device open), the kernel will close the descriptor on the jail's behalf. The driver's `close` method will run in a safe context. But the driver should not assume that the jail still exists during `close`; it may not, and if the driver tries to reach into the jail's state, it will encounter a freed `struct prison`. The clean rule is that `close` should only touch driver state, not jail state.

30.10.6 How Drivers Cooperate with Jails in Practice

Putting the pieces together, a typical FreeBSD setup with a driver and a jail looks like this.

1. The administrator loads the driver on the host, either at boot through `/boot/loader.conf` or at runtime through `kldload(8)`.

2. The driver's probe and attach functions run on the host, create whatever devfs nodes are appropriate, and register their `cdev_methods`.
3. The administrator creates a jail, possibly with a specific devfs ruleset and possibly with VNET.
4. The jail's processes open the driver's devices (if they are visible through the ruleset) and make `ioctl` or `read/write` calls.
5. The driver's methods execute on behalf of the jail process, with the jail's credential attached to the thread, and the driver's `priv_check` calls correctly return `EPERM` for privileges the jail does not hold.
6. When the jail is destroyed, open file descriptors are closed, the driver's `close` methods run cleanly, and the driver's state returns to its normal host-only view.

Nothing in this flow requires the driver to know about jails explicitly. The driver is a passive participant that calls the right framework functions, and the framework handles the rest. This is the cleanest possible design, and it is what you should aim for.

30.10.7 The Container Frameworks: `ocijail` and `pot`

FreeBSD's jail infrastructure is a kernel mechanism; the user-space tooling around it has multiple forms. The base system provides `jail(8)`, `jail.conf(5)`, `jls(8)`, and related tools. These are enough to manage jails by hand or with shell scripts.

Higher-level container frameworks have emerged on top. `ocijail` aims to provide an OCI (Open Container Initiative) runtime that uses jails as the isolation mechanism, letting FreeBSD participate in container ecosystems that use OCI-compliant images. `pot` (available from ports) is a more FreeBSD-native container manager that bundles a jail with a filesystem layer, a network configuration, and a lifecycle. Both are external to the base system and are installed through the ports collection or packages.

For a driver author, these frameworks do not change the fundamentals. They still use jails underneath; they still rely on devfs rulesets and VNET for isolation; they still respect the same privilege framework. What they change is how administrators describe and deploy the containers, not how the drivers interact with them. A driver that works with a hand-crafted `jail.conf(5)` will work with `ocijail` and `pot` as well.

The most a driver author usually needs to know is that these frameworks exist and are becoming common. If your driver's documentation mentions jails, mention that the recommendations apply equally to container frameworks built on top. That single sentence saves administrators a lot of guesswork.

30.10.8 Wrapping Up

Jails are a policy boundary around a shared kernel. The policy extends into three dimensions: which devices are visible (devfs rulesets), which privileges are granted (`prison_priv_check` and `allow.*` parameters), and which resources can be consumed (`rctl(8)`). Driver authors encounter each of these in small, local ways: `priv_check` for privileged operations, sensible naming of devfs nodes, graceful handling of allocation failures. There is no large new API to learn; there are small new habits to pick up.

With the security and resource picture covered, the final conceptual topic is how to actually test and develop drivers in virtualised and containerised environments. Section 8 pulls the chapter's ideas together into a development workflow.

30.11 Section 8: Testing and Refactoring Drivers for Virtualised and Containerised Environments

A driver that runs on bare metal is a driver that works on one configuration. A driver that runs across virtualisation and containerisation has been exercised under varied conditions: different bus presentations, different interrupt delivery mechanisms, different memory-mapping behaviours, different privilege contexts. The driver that makes it through all of those without changes is the driver that will survive the next new environment too. This section describes the development and testing workflow that gets you there.

The workflow has three layers. The development layer uses a VM as a disposable kernel host, so that panics and hangs cost you nothing. The integration layer uses VirtIO devices, passthrough, and jails to exercise the driver in realistic environments. The regression layer uses automation to run the whole suite repeatedly as the driver evolves.

30.11.1 Using a VM as Your Development Host

When you are writing a kernel module, the cost of a panic is your session. On a bare-metal development machine, a panic interrupts your work, possibly forces a filesystem check, and may require a reboot with manual recovery steps. On a VM, a panic is a detail: the VM stops, you restart it, and the host is untouched.

For this reason, experienced driver authors do almost all of their new-driver development inside a bhyve(8) or QEMU-based VM, not on bare metal. The workflow looks like this:

1. A FreeBSD 14.3 VM is installed in bhyve(8) with a standard disk image.
2. The source tree is either on the VM's own disk or mounted via NFS from a host-side build machine.
3. The driver is built inside the VM (`make clean && make`) or on the host and copied in.
4. `kldload(8)` loads the module. If the module panics the kernel, the VM crashes, and the VM is restarted.
5. Once the module loads cleanly, it is exercised against whatever test fixture you have: a VirtIO device, a loopback mode, or a passthrough target.

The key point is that iteration is fast. A broken module that would make a bare-metal machine unbootable is a minor inconvenience in a VM. You can try things that you would never try on a machine you rely on.

For VirtIO driver development specifically, the VM is not just convenient; it is the only sensible platform. VirtIO devices only exist inside VMs (or under qemu emulation), so the VM is where the devices are. Starting a VM with a virtio-rnd device, or a virtio-net device, or a virtio-console device, gives you a target to develop against. The hypervisor provides everything a real device would provide, including interrupts, DMA, and register access, so the driver you write inside the VM is the same driver that will run anywhere else.

30.11.2 Using VirtIO as a Test Substrate

VirtIO has a second role beyond being the target for VirtIO drivers: it is a test substrate. Because VirtIO devices are easy to define, easy to emulate, and well-documented, they are useful for building controlled test scenarios even for drivers that are not VirtIO drivers.

For example, suppose you are writing a driver for a physical PCI device, and you want to test how your driver handles a specific error condition. On the real hardware, reproducing the error may require a specific physical

fault, which is hard to arrange. On a VirtIO-based proxy, you can implement a device that always returns the error, and test your driver's error path without touching the physical hardware. The caveat is that your driver must be loosely coupled to the specific hardware (the techniques from Chapter 29 matter here); the more the driver is split along the accessor/backend lines described there, the easier it is to test the upper layers against a synthetic backend.

The mechanism for synthesizing VirtIO devices in user space is bhyve's own pluggable emulated devices. Writing a new bhyve device emulator is beyond the scope of this chapter, but the relevant code lives in `/usr/src/usr.sbin/bhyve/` and is approachable if you have basic C skills. For simpler cases, using a pre-existing virtio-blk, virtio-net, or virtio-console device configured with specific parameters is often enough.

30.11.3 Using Jails for Integration Testing

When your driver is working and you want to verify it under container-style isolation, jails are the obvious next step. The setup is simple: create a jail with an appropriate devfs ruleset that exposes your device, and run your user-space test harness inside the jail.

A typical test shape:

```
myjail {
    path = /jails/myjail;
    host.hostname = myjail;
    devfs_ruleset = 100;    # custom ruleset that unhides /dev/mydev
    ip4 = inherit;
    allow.mount = 0;
    exec.start = "/bin/sh /etc/rc";
    exec.stop = "/bin/sh /etc/rc.shutdown";
    persist;
}
```

Inside the jail, you run your test harness. It opens `/dev/mydev`, exercises the `ioctl`s or the `read/write` methods, and records results. On the host, you run the same harness and compare. If the jail-side test passes and the host-side test passes, your driver tolerates the jail environment.

If one passes and the other does not, you have a diagnostic opportunity. Possible reasons for a divergence include: a privilege check that denies the jail call (look for `priv_check` in your driver), a device node permission the ruleset does not account for, a resource limit the host happens to avoid, or a jail-specific code path in your driver that exists by mistake. Each of these is fixable once identified.

30.11.4 Using VNET Jails for Network Driver Testing

For network drivers, the test is similar but uses VNET. Create a jail with `vnet = 1` in `jail.conf(5)`, move one end of an `epair(4)` into the jail, and run traffic between the jail and the host. If your driver is a physical network driver, you can also move the physical interface into the jail for a full-isolation test.

The VNET test exercises the `if_vmove()` lifecycle: the interface is detached from the host's VNET, reattached to the jail's VNET, and eventually returned. A driver that survives this without losing state is a driver that tolerates VNET. A driver that panics, hangs, or stops delivering packets after a move has work to do.

The common failure modes in VNET testing are:

- The driver holds a pointer to the host's ifnet or VNET across the move, and uses it from a callout after the move has happened.
- The driver's `if_init` assumes it is called in the original VNET and fails when called in the new one.
- The driver cleans up incorrectly at `if_detach`, because it does not distinguish “detach to move” from “detach to destroy”.

Each of these is diagnosable with `dtrace(1)` or kernel `printfs` at the right place. The first time you see a VNET-related crash, find the point in the driver where the move happens and work backward from there.

30.11.5 Passthrough Testing

PCI passthrough is the exercise that validates your driver's detach path. Create a `bhyve(8)` guest with your device passed through, install FreeBSD inside, and load your driver there. If the driver attaches cleanly in the guest, your device setup and DMA code handle IOMMU remapping correctly. If the driver loads, the test is simple: run the driver's normal workload inside the guest.

The detach test is the harder one. Shut down the guest, rebind the device to its native driver on the host (by unloading `ppt(4)` from that device, if necessary, and letting the host's driver re-attach), and exercise the driver on the host. If the driver attaches cleanly after the guest has used the device and put it through whatever state changes, the driver's `DEVICE_ATTACH` is properly defensive. If it fails, look for assumptions about the device's initial state that should not be assumptions.

The full round-trip (host, guest, host) is the gold standard for passthrough compatibility. A driver that passes it can be handed to any administrator with confidence.

30.11.6 Hypervisor Detection in Tests

If your driver uses `vm_guest` to adjust defaults, test the adjustment. Run the driver on bare metal (if available), inside `bhyve(8)`, inside QEMU/KVM, and observe whether the defaults it picks make sense. The `kern.vm_guest sysctl` is your quick check:

```
sysctl kern.vm_guest
```

If your driver logs its environment at attach time (“attaching on bhyve host, defaulting to X”), the log makes the detection visible, which helps with debugging. Do not over-log: once at attach is usually enough.

30.11.7 Automating the Test Suite

Once the individual tests are known, the next step is to run them repeatedly as the driver evolves. FreeBSD's `kyua(1)` test runner, combined with the `atf(7)` test framework, is the standard mechanism. A test suite that

includes a “bare-metal test”, a “VirtIO guest test”, a “VNET jail test”, and a “passthrough test” covers most of what you want to verify.

The details of writing tests in `atf(7)` are outside this chapter; they are treated more fully in Chapter 32 (Debugging Drivers) and Chapter 33 (Testing and Validation). The point for now is that the test suite should exercise the driver across the environments it is expected to run in. A single test on bare metal proves very little about virtualisation; a suite of tests across environments proves something about portability.

30.11.8 Refactoring Tips Revisited

In Chapter 29 we introduced a discipline for portability: accessor layers, backend abstractions, endian helpers, no hidden hardware assumptions. Virtualisation and containerisation put that discipline to the test. A driver written with clean abstractions will survive the variety of environments described in this chapter; a driver with hidden assumptions will hit them as soon as the environment changes.

The refactoring tips most relevant to this chapter are:

- Put all register access through accessors, so the access path can be simulated or redirected in tests.
- Handle the full lifecycle: attach, detach, suspend, resume. Passthrough exercises attach and detach repeatedly; VNET exercises detach in a way the driver might not see on bare metal.
- Use `bus_dma(9)` and `bus_space(9)` correctly, never physical addresses directly. The guest-versus-host address translation under passthrough depends on correct use of these APIs.
- Use `priv_check` for privilege gating, not hardcoded uid 0 checks. Jail restrictions work correctly only if the framework is called.
- Use `CURVNET_SET` around any callout or taskqueue code that touches network-stack state. This is the one VNET-specific discipline that catches most driver authors off guard.

None of these are new concepts. They are all standard FreeBSD driver practice. What this chapter adds is the context in which each one matters: which environments exercise which disciplines. Knowing that lets you prioritise when deciding what to refactor first.

30.11.9 A Development Order That Works

Putting the pieces together, here is a development order that has proven effective.

1. Start in a `bhyve(8)` VM. Write the driver’s basic skeleton (module hooks, probe and attach, simple I/O path). Exercise it with `kldload` and a minimal test.
2. Add the accessor layer and the backend abstraction from Chapter 29. Test that the simulation backend runs, even if the real hardware is not yet plugged in.
3. If VirtIO is the target, develop against `virtio_pci.c` in the VM. You have a real device to talk to, and you can iterate quickly.
4. If real hardware is the target, begin PCI passthrough testing when the driver reaches a stable point. The round-trip (host, guest, host) becomes part of the regular test cycle.
5. Add jail-based tests when the driver exposes user-space interfaces. Start with a single-stack jail; move to VNET if the driver is a network driver.
6. Add automation with `kyua(1)` and `atf(7)` as the test count grows.

7. When a bug is found, reproduce it in the smallest environment that shows the bug, fix it, and add a regression test at that level.

This order keeps the iteration fast at the beginning (where it matters most) and adds environmental complexity only as the driver stabilises. Trying to test everything at once is a common beginner mistake; it is how projects stall. The incremental path is slower per step but much faster in aggregate.

30.11.10 An End-to-End Example: From Bare Metal to Passthrough

To illustrate the workflow, here is a complete example walkthrough for a hypothetical driver called `mydev`. The driver is a PCI-based character device; it has a small register interface, uses MSI-X interrupts, and performs DMA. The development order below is condensed into a single narrative so you can see how the steps connect.

Day 1: skeleton in a VM. You install a FreeBSD 14.3 guest in `bhyve(8)`, set up NFS so the source tree on your host is visible in the guest, and write the module skeleton. It is a `KMOD=mydev`, `SRCS=mydev.c` Makefile and a `mydev.c` with `DECLARE_MODULE`, a stub probe, a stub attach, and a stub detach. It loads and unloads cleanly. `dmesg` shows “`mydev: hello`” on load.

Day 2: accessor layer. You add the Chapter 29 accessor pattern: all register access goes through `mydev_reg_read32` and `mydev_reg_write32`, with the real backend calling `bus_read_4` and `bus_write_4`. You also add a simulation backend that stores register values in a small in-memory array. The simulation backend is selected by a module parameter. The accessor layer means the same driver can run against a real device or against the simulated backend without changes to the upper code.

Day 3: upper-layer code. You add the driver’s core logic: initialisation, the character-device interface (`open`, `close`, `read`, `write`), the `ioctl` surface, and the DMA setup. The simulation backend does not model DMA, but the upper layer is structured to treat DMA through `bus_dma(9)` handles, so the code is written correctly from the start. You exercise the code through the simulation backend in a simple test program: open the device, issue `ioctl`s, verify the responses.

Day 4: real hardware, passthrough setup. You have the target hardware in a workstation. You add `pptdevs` to `/boot/loader.conf`, reboot, and confirm that `ppt(4)` has claimed the device. You add `passthru` to the `bhyve(8)` guest configuration and boot the guest. Inside the guest, you load your driver. The driver attaches to the passed-through device. You have now demonstrated that the real hardware path works.

Day 5: interrupts and DMA testing. The driver receives interrupts; the MSI-X setup code works. You test DMA: a short DMA read works, a long DMA read works, a simultaneous read and write works. You find one bug: the driver programs a physical address computed incorrectly, but only for DMA regions crossing a page boundary. You fix it. Total time spent debugging: two hours, all in a VM that would have required a hard reboot of the workstation on bare metal.

Day 6: jail test. You exit the guest, return to the host, and configure a jail that sees `/dev/mydev0`. Your test program runs inside the jail and exercises the driver exactly as it did on the host. One `ioctl` fails with `EPERM`; you look at the driver and find that you forgot to add a `priv_check` for an operation that should require privilege. You add the check, and now the jail behaves correctly (the `ioctl` is denied for non-privileged callers; the host’s root can still run it).

Day 7: VNET test (if the driver has a network interface). You create a VNET jail and move a cloned interface into it. The interface works. You notice that one of your callouts does not set `CURVNET_SET` before accessing

a per-VNET counter; you fix it. The callout now works in both the host VNET and the jail VNET without interference.

Day 8: full round-trip. You destroy the jail, shut down the guest, unload `ppt(4)` from the device (or reboot without `pptdevs`), and wait for the host's driver to re-attach. The attach works cleanly. You exercise the device on the host. It works. The round trip host-guest-host is complete.

This eight-day cycle is a stylised version of real development; your mileage will vary. The important point is that each day's work builds on the previous one, and the test environments become more exacting as the driver stabilises. By day eight, you have exercised the driver under every environment it will see in production, and you have fixed the bugs that each environment exposes. What remains is soak testing and user-facing polish, which are topics for Chapters 33 and 34.

30.11.11 Measuring Virtualisation Overhead

A quick note on performance. Drivers running under virtualisation sometimes show measurable overhead compared to bare metal. The sources of overhead include VM exits on I/O, interrupt delivery through the hypervisor, and DMA address translation through the IOMMU.

For most drivers most of the time, the overhead is not significant. Modern hardware accelerates almost every part of the virtualisation path (posted interrupts, EPT page tables, SR-IOV), and the fraction of CPU time spent in hypervisor code is typically in the low single digits. For performance-sensitive drivers, though, measuring and understanding the overhead is essential.

The tools for measurement are the same ones you use in other performance work. `pmcstat(8)` samples hardware counters, including counters for VM exits and for translation-lookaside-buffer misses that the IOMMU may cause. `dtrace(1)` can trace specific kernel paths, and with the `fbt` provider you can measure how often each path is entered and how long it takes. `vmstat -i` shows interrupt rates.

For VirtIO drivers, the most common source of overhead is excessive notifications. Every `virtqueue_notify` potentially causes a VM exit, and a driver that notifies on every packet rather than coalescing can generate hundreds of thousands of exits per second. The `VIRTIO_F_RING_EVENT_IDX` feature, if negotiated, lets the guest and host cooperate to reduce notification frequency. Check that your driver negotiates this feature if it runs in a high-packet-rate path.

For passthrough drivers, the most common source of overhead is IOMMU translation misses. Each DMA buffer must be walked through the IOMMU's page tables, and a driver that maps and unmaps buffers many times per second spends a lot of CPU on that. The fix is usually to keep DMA mappings alive for longer (using `bus_dma(9)`'s mapping-retention features) rather than map-and-unmap every transaction.

Performance tuning is a whole chapter of its own (Chapter 34). For now, the takeaway is that virtualisation has measurable costs, those costs are usually small, and the standard FreeBSD tooling applies without modification.

30.11.12 Wrapping Up

A driver that runs correctly under virtualisation and containerisation is not the result of special virtualisation code. It is the result of standard FreeBSD driver discipline, exercised in environments that expose hidden assumptions. The test and development workflow in this section is the practical side of that discipline: a VM for fast iteration,

VirtIO for a controlled test substrate, jails for privilege and visibility checks, VNET for network-stack testing, passthrough for hardware round-trips, and automation to keep the whole thing honest.

With the conceptual and practical material in place, the rest of the chapter turns to hands-on labs that let you try these techniques yourself. Before we get there, two more sections complete the picture. Section 9 covers the quieter but equally important topics of time, memory, and interrupt handling under virtualisation, the areas where beginner drivers often fail in subtle ways that only manifest in a VM. Section 10 widens the lens to architectures beyond amd64, because FreeBSD on arm64 and riscv64 is increasingly common and the virtualisation story there has its own shape. After those two sections, we move into the labs.

30.12 Section 9: Timekeeping, Memory, and Interrupt Handling Under Virtualisation

So far the chapter has concentrated on devices, the visible artefacts that a driver binds to and talks to. But a driver also depends on three ambient services that the kernel provides transparently: time, memory, and interrupts. Under virtualisation, all three change in subtle ways that rarely break a driver outright but often make it behave strangely. A driver that ignores these differences will usually pass functional tests and then fail in production when a user notices that timeouts are wrong, throughput is lower than expected, or interrupts are being lost. This section gathers what every driver author should know.

30.12.1 Why Time Is Different Inside a VM

On bare metal, the kernel has direct access to several hardware time sources. The TSC (time-stamp counter) reads a per-CPU cycle counter; the HPET (high-precision event timer) provides a system-wide counter; the ACPI PM timer provides a lower-frequency fallback. The kernel chooses one as the current timecounter(9) source, wraps it in a small API, and uses it to derive `getbintime`, `getnanotime`, and friends.

Inside a VM, those same sources exist, but they are emulated or passed through, and each has its own quirks. The TSC, which is normally the best source, can become unreliable when the VM migrates between physical hosts, when the host's TSCs are unsynchronised, or when the hypervisor rate-limits the guest in ways that cause TSC skew. The HPET is emulated and costs a VM exit on every read, which is cheap for occasional use but expensive if a driver reads it in a tight loop. The ACPI PM timer is generally reliable but slow.

To address this, major hypervisors expose *paravirtual* clock interfaces. Linux popularised the term `kvm-clock` for the KVM paravirtual clock; Xen has `xen-clock`; VMware has `vmware-clock`; Microsoft's Hyper-V has `hyperv_tsc`. Each of these is a small protocol by which the hypervisor publishes time information in a shared memory page that the guest can read without a VM exit. FreeBSD supports several of them. You can see which the kernel has chosen as follows.

```
% sysctl kern.timecounter.choice
kern.timecounter.choice: ACPI-fast(900) i8254(0) TSC-low(1000) dummy(-1000000)

% sysctl kern.timecounter.hardware
kern.timecounter.hardware: TSC-low
```

On a guest under bhyve, the kernel may select TSC-low or one of the paravirtual options depending on the CPU flags the hypervisor advertises. The important point is that the choice is automatic and the `timecounter(9)` API is the same regardless.

30.12.2 What This Means for Drivers

For a driver that only uses the high-level time APIs (`getbintime`, `ticks`, `callout`), nothing needs to change. The `timecounter(9)` abstraction shields you from the underlying details. The driver asks “what time is it” or “how many ticks have elapsed” and the kernel answers correctly, whether on bare metal or in a VM.

Problems arise when a driver bypasses the abstraction and reads time sources directly. A driver that does `rdtsc()` inline and uses the result for timing will be wrong under virtualisation whenever the host’s TSC changes (for example, during a live migration). A driver that spins on a device register with a timeout measured in CPU cycles will consume excessive CPU inside a VM where one “CPU cycle” is not a predictable unit.

The cure is simple: use the kernel’s time primitives. `DELAY(9)` for short, bounded waits. `pause(9)` for yields that can sleep. `callout(9)` for deferred work. `getbintime(9)` or `getsbinuptime(9)` for clock readings. Each of these is correct under virtualisation because the kernel has already adapted to the environment.

A concrete pattern that breaks in VMs and is surprisingly common is the “reset and wait” sequence.

```
/* Broken pattern: busy-wait without yielding. */
bus_write_4(sc->res, RESET_REG, RESET_ASSERT);
for (i = 0; i < 1000000; i++) {
    if ((bus_read_4(sc->res, STATUS_REG) & RESET_DONE) != 0)
        break;
}
```

On bare metal, this loop might complete in a few microseconds because the device clears the status bit quickly. In a VM, the bus read and write each cost a VM exit, and the loop body runs much slower because every iteration makes a round trip through the hypervisor. The 1,000,000 iteration bound, which is instant on bare metal, can become a multi-second hang inside a VM. Worse, during a VM pause (a live migration), the guest does not execute at all, and the timeout loses its meaning.

The corrected pattern uses `DELAY(9)` and a bounded real-time wait.

```
/* Correct pattern: bounded wait with DELAY and a wallclock timeout. */
bus_write_4(sc->res, RESET_REG, RESET_ASSERT);
for (i = 0; i < RESET_TIMEOUT_MS; i++) {
    if ((bus_read_4(sc->res, STATUS_REG) & RESET_DONE) != 0)
        break;
    DELAY(1000);    /* one millisecond */
}
if (i == RESET_TIMEOUT_MS)
    return (ETIMEDOUT);
```

DELAY(9) is calibrated against the kernel's time source, so it sleeps for the intended number of microseconds regardless of how fast or slow the CPU is executing at that moment. The loop's bound is now expressed in milliseconds, which is meaningful in both bare-metal and virtualised environments.

30.12.3 Callouts and Timers Across Migration

A more subtle concern is what happens to a driver's callouts when the VM pauses. If a callout was scheduled to fire in 100 ms and the VM is paused for 5 seconds (during a live migration, for example), does the callout fire 5.1 seconds after it was scheduled, or 100 ms after the VM resumes?

The answer is that it depends on the clock source the callout(9) subsystem uses. callout uses sbt (signed binary time), which under normal circumstances is derived from the selected timecounter. For hypervisors that pause the guest's virtual TSC during migration, the callout behaves as if no time had passed during the pause; the 100 ms wait is 100 ms of guest-observed time, which may be 5.1 seconds of wallclock time. For hypervisors that do not pause the virtual TSC, the callout fires at the scheduled wallclock time, which may be "immediately" after the VM resumes.

For most drivers, either behaviour is acceptable. The callout eventually fires and the code it triggers runs. But a driver that measures real-world time (for example, a driver that talks to a hardware device whose state depends on wallclock time) may need to re-sync after a resume. FreeBSD's suspend-and-resume infrastructure provides DEVMETHOD(device_resume, ...), and a driver can detect a resume and take corrective action.

```
static int
mydrv_resume(device_t dev)
{
    struct mydrv_softc *sc = device_get_softc(dev);

    /*
     * After a resume (from ACPI suspend or VM pause), the device
     * may have lost state and the driver's view of time may no
     * longer align with the device's. Reinitialise what needs
     * reinitialising.
     */
    mydrv_reset(sc);
    mydrv_reprogram_timers(sc);

    return (0);
}
```

bhyve(8) supports guest suspend through its own suspend interface (on hypervisors that implement it); the kernel delivers a normal device_resume method call on wake-up. A driver that writes device_resume correctly works in both cases.

30.12.4 Memory Pressure, Ballooning, and Pinned Buffers

Drivers that own DMA buffers or pinned memory have a relationship with the memory subsystem that changes under virtualisation. On bare metal, the physical memory the kernel sees is the memory actually installed in the machine. Inside a VM, the guest's "physical" memory is virtual from the host's point of view: it is backed by host RAM plus possibly swap, and its extent can change during the guest's lifetime.

The `virtio-balloon` device is the mechanism hypervisors use to reclaim memory from a guest that is not using it. When the host needs memory, it asks the guest to "inflate" its balloon, which allocates pages from the guest kernel's free pool and declares them unusable. Those pages can then be unmapped from the guest and reused by the host. Conversely, when the host has memory to spare, it can "deflate" the balloon and return pages to the guest.

FreeBSD has a `virtio_balloon` driver (in `/usr/src/sys/dev/virtio/balloon/virtio_balloon.c`) that participates in this protocol. For most drivers, this is invisible: the balloon takes from the general free pool, so only drivers that pin significant amounts of memory can be affected. If your driver allocates a 256 MB DMA buffer and pins it (for a frame buffer, for example), the balloon cannot reclaim that memory. This is the correct behaviour for a pinned buffer, but it does mean that a VM running your driver cannot shrink its memory footprint as much as a VM with only non-pinning drivers.

A pragmatic guideline: avoid allocating more pinned memory than you absolutely need. For buffers that can be allocated on demand, allocate them on demand. For buffers that must be resident, size them to a reasonable working-set bound rather than a pessimistic maximum. The balloon driver will return the rest to the host when it needs to.

30.12.5 Memory Hotplug

Some hypervisors (including bhyve with appropriate support) can hot-add memory to a running guest. FreeBSD handles this through ACPI events and the generic hotplug machinery. Drivers that cache memory information at attach time must be prepared for that cache to become stale when hotplug occurs; the robust pattern is to re-read the information when needed rather than caching it indefinitely.

Hot-removal of memory is rarer and more delicate. For driver authors, it is usually enough to note that if a driver owns pinned memory, the hypervisor cannot remove it; if the driver owns unpinned memory, the kernel's memory management handles relocation. Drivers that violate this (by expecting that physical addresses they have read from the kernel remain valid forever) will break under memory hot-removal. The fix is to go through `bus_dma(9)` for every physical address that gets programmed into hardware, not to cache physical addresses outside of a DMA map.

30.12.6 DMA and the IOMMU Path

We covered IOMMU in Section 5 from the host's perspective. Here, from the driver's perspective, there are two practical consequences.

First, every address programmed into hardware must come from a `bus_dma(9)` load operation. Under bare metal without an IOMMU, a driver that programs a physical address obtained from `vmem_alloc` or a similar interface usually works, because physical-equals-bus in that environment. Under passthrough with an IOMMU, the bus address the device needs is not the physical address; it is the IOMMU-mapped address, which `bus_dma_load`

computes. A driver that programs physical addresses directly will transfer data to or from the wrong memory, sometimes corrupting unrelated data.

Second, `bus_dma` mappings have lifetimes. A typical pattern is to allocate a DMA tag at attach, allocate a DMA map per buffer, load the buffer, program the device, wait for completion, and then unload the buffer. The load and unload each cost a small amount of CPU, and under the IOMMU they also cost an IOMMU invalidation. For drivers that cycle through many small buffers per second, the invalidation cost can become significant.

The fix, when it applies, is to keep DMA mappings alive for longer. `bus_dma` supports pre-allocated maps that can be reused; a driver that needs to DMA into the same physical region repeatedly can load it once and reuse the bus address until the region is no longer needed. This is a standard optimisation and is entirely orthogonal to virtualisation, but it matters more under passthrough because the IOMMU work is bigger than the host-to-bus work on bare metal.

30.12.7 Interrupt Delivery in Virtualised Environments

Interrupts have always been the sharp edge of driver design, and under virtualisation they become sharper. The two interrupt styles a driver encounters are *INTx* (pin-based) and *MSI/MSI-X* (message-signalled).

INTx is the old-style interrupt pin. In a real machine, the pin connects through the PCI bus and an interrupt controller (APIC, IOAPIC) to the CPU. In a VM, each *INTx* delivery requires the hypervisor to intercept the device's pin assertion, map it to an internal interrupt, and inject it into the guest. The intercept and injection both cost VM exits. For low-rate interrupts (the classic “something happened” signal) this is fine. For high-rate interrupts, it can be a bottleneck.

MSI (Message-Signalled Interrupts) and its successor MSI-X avoid the pin entirely. The device writes a small message to a well-known memory-mapped address, and the interrupt controller delivers the corresponding interrupt vector. Under virtualisation, MSI-X works much better than *INTx* because the hypervisor can map the message write directly to a guest interrupt without needing to intercept every edge transition on a virtual pin. Modern hardware supports *posted interrupts*, which let the hypervisor deliver MSI-X interrupts to a running vCPU without any VM exit at all.

The driver-side implication is clear: prefer MSI-X. FreeBSD's `pci_alloc_msix` API lets a driver request MSI-X interrupts. Most modern drivers already use it. If you are writing a new driver or updating an old one, use MSI-X unless you have a specific reason not to.

```
static int
mydrv_setup_msix(struct mydrv_softc *sc)
{
    int count = 1;
    int error;

    error = pci_alloc_msix(sc->dev, &count);
    if (error != 0)
        return (error);

    sc->irq_rid = 1;
    sc->irq_res = bus_alloc_resource_any(sc->dev, SYS_RES_IRQ,
```



```
        &sc->irq_rid, RF_SHAREABLE | RF_ACTIVE));
if (sc->irq_res == NULL) {
    pci_release_msi(sc->dev);
    return (ENXIO);
}

error = bus_setup_intr(sc->dev, sc->irq_res,
    INTR_TYPE_NET | INTR_MPSAFE, NULL, mydrv_intr, sc,
    &sc->irq_handle);
if (error != 0) {
    bus_release_resource(sc->dev, SYS_RES_IRQ,
        sc->irq_rid, sc->irq_res);
    pci_release_msi(sc->dev);
    return (error);
}

return (0);
}
```

This is the standard MSI-X setup sequence. The `pci_alloc_msix` call negotiates with the kernel's PCI layer to allocate one MSI-X vector. The resource and interrupt handler are set up as usual. The cleanup path releases the MSI-X vector along with the other resources.

For drivers with multiple queues or multiple event sources, MSI-X supports up to 2048 vectors per device, and a driver can allocate one per queue to avoid lock contention. The `pci_alloc_msix` API supports requesting multiple vectors; `count` is an in-out parameter. Under virtualisation, each vector maps to a separate guest interrupt, and posted interrupts deliver them with no VM exit on modern hardware.

30.12.8 Interrupt Coalescing and Notification Suppression

Even with MSI-X, an interrupt-per-transaction model can be too expensive under virtualisation. A high-rate device that fires an interrupt for every received packet can generate hundreds of thousands of interrupts per second, and although each is cheap individually, the aggregate cost is noticeable.

Hardware interrupt coalescing addresses this on real devices: the device can be configured to deliver one interrupt for a batch of events rather than one per event. Under VirtIO, the equivalent mechanism is *notification suppression*, exposed through the `VIRTIO_F_RING_EVENT_IDX` feature.

With event indexes, the guest tells the device “do not interrupt me until you have processed up to descriptor N”, and the device honours this by checking the guest's `used_event` field before raising an interrupt. A guest that is already polling the ring does not need an interrupt at all; a guest that wants to be interrupted only after a batch can set the event index to the batch size.

FreeBSD's `virtqueue(9)` framework supports event indexes when negotiated. A driver that knows it may process multiple buffers per interrupt can enable event indexes to reduce interrupt rate. The classic pattern is:

```
static void
mydrv_intr(void *arg)
{
    struct mydrv_softc *sc = arg;
    void *cookie;
    uint32_t len;

    virtqueue_disable_intr(sc->vq);
    for (;;) {
        while ((cookie = virtqueue_dequeue(sc->vq, &len)) != NULL) {
            /* process the completed buffer */
        }
        if (virtqueue_enable_intr(sc->vq) == 0)
            break;
        /* a new buffer arrived between dequeue and enable; loop again */
    }
}
```

The `virtqueue_disable_intr` call tells the device not to interrupt again until re-enabled. The driver then drains the ring. The `virtqueue_enable_intr` call arms interrupts, but only if no new buffer has arrived in the meantime; if one has, it returns nonzero and the loop continues. This is a standard pattern that minimises interrupt rate without ever missing a completion.

30.12.9 Putting the Pieces Together

Time, memory, and interrupts are not glamorous topics, but they are where the rubber meets the road for drivers that want to be correct under virtualisation. The guidelines distil to a small set of disciplines:

- Use the kernel's time APIs rather than rolling your own.
- Treat the device's state as a thing that can disappear on resume; write `device_resume` correctly.
- Do not pin more memory than necessary, and go through `bus_dma(9)` for every DMA address.
- Prefer MSI-X over INTx.
- Use the virtqueue event-index mechanism where appropriate to reduce interrupt rate.

These are the disciplines of any well-written driver; under virtualisation they are not optional. A driver that follows them will work in a VM. A driver that violates them will work on bare metal and fail in the field on a customer's VM, sometimes in ways that are difficult to reproduce.

With that grounding in place, we can zoom out one more time and look at how these ideas travel to architectures other than amd64.

30.13 Section 10: Virtualisation on Other Architectures

FreeBSD runs well on amd64, arm64, and riscv64. The virtualisation story on amd64 is the one we have been telling: `bhyve(8)` uses Intel VT-x or AMD-V, guests see PCI-based VirtIO devices, the IOMMU is VT-d or

AMD-Vi, and the general flow is familiar. On the other architectures, the pieces are similar but the specifics differ. For a driver author, most of this is invisible, because FreeBSD's virtualisation APIs are architecture-independent. But a few practical points are worth knowing, and a few drivers have architecture-specific behaviour that only surfaces under virtualisation.

30.13.1 arm64 Virtualisation

On arm64, the hypervisor mode is called *EL2* (Exception Level 2), and the virtualisation extensions are part of the architecture's standard specification (ARMv8-A virtualisation). A guest runs in EL1 under the hypervisor in EL2. There is no direct amd64-style INTx; the interrupt controller is the *GICv3* (Generic Interrupt Controller version 3), and interrupt virtualisation is provided by the GIC's *virtual CPU interface*.

FreeBSD has an arm64 bhyve(8) host-side port under development that targets a future release; as of FreeBSD 14.3 the vmm implementation ships only for amd64, so arm64 hosts do not yet run guests natively. The guest side of FreeBSD, however, already uses the same `virtio_mmio` and `virtio_pci` transports that amd64 does, so a FreeBSD guest running on an arm64 hypervisor (for example, KVM or a future arm64 bhyve) does not know or care that the host is arm64.

For VirtIO specifically, arm64 guests more often use the MMIO transport. This is a practical consequence of how the virtual platforms are typically configured: arm64 hypervisors often expose VirtIO devices as MMIO regions rather than as emulated PCI buses. FreeBSD's `virtio_mmio.c` (in `/usr/src/sys/dev/virtio/mmio/`) provides the transport. A driver that uses `VIRTIO_DRIVER_MODULE` is automatically compatible with both transports, because the macro registers the driver with both `virtio_mmio` and `virtio_pci`.

This is one of the quiet wins of the `virtio_bus` design. A VirtIO driver written once runs across every VirtIO transport on every architecture that FreeBSD supports. No `#ifdef __amd64__` clauses, no per-architecture translation layer. The `virtio_bus` abstracts the transport; the driver talks to the abstraction.

30.13.2 riscv64 Virtualisation

On riscv64, virtualisation is provided by the *H extension* (Hypervisor extension). FreeBSD has a bhyve port for riscv64 as well, though it is less mature than the amd64 and arm64 ports as of FreeBSD 14.3. The VirtIO transports work the same way: drivers use `VIRTIO_DRIVER_MODULE` and the kernel's `virtio_bus` handles the transport specifics.

For driver authors working on riscv64, the most important thing to know is that the architecture-independent APIs all apply. `bus_dma(9)`, `callout(9)`, `mtx(9)`, `sx(9)`, and the VirtIO framework all work on riscv64. Code that is correct on amd64 usually runs unchanged on riscv64. The differences are in lower-level details (memory ordering, cache flushing, interrupt routing) that the kernel handles under the hood.

30.13.3 Cross-Architecture Considerations for Driver Authors

If you are writing a driver that should work on multiple architectures, the guidelines from Chapter 29 apply directly. Use the architecture-independent APIs. Avoid inline assembly except where strictly necessary (and then isolate it behind a portable wrapper). Do not assume a specific cache-line size or a specific page size. Do not assume an architecture's endianness unless you have explicitly used a conversion macro.

For virtualisation specifically, the main cross-architecture concern is *which VirtIO transports are present*. On amd64, VirtIO is almost always PCI. On arm64 with QEMU or Ampere hypervisors, VirtIO is often MMIO. On riscv64 with QEMU, VirtIO is often MMIO. A driver that only handles PCI VirtIO will not work in MMIO environments. The fix is to use `VIRTIO_DRIVER_MODULE`, which registers the driver with both transports, and to avoid assuming that the device's parent bus is PCI.

A concrete test: on your target architecture, run `pciconf -l` inside a guest and see whether VirtIO devices appear. If they do, the transport is PCI. If they do not (but the devices work), the transport is MMIO. On arm64 guests, you can also check `sysctl dev.virtio_mmio` to see MMIO VirtIO devices. A driver that works with both transports will not produce different output in these checks, because the `virtio_bus` API is what it interacts with.

30.13.4 When Architecture Matters for Real

Most drivers are architecture-independent if written in the FreeBSD idiom. The exceptions are:

- Drivers that deal with hardware-specific features: for example, ARM's SMMU is architecturally different from Intel's VT-d, and a driver that manipulates the IOMMU directly (rare) must handle both.
- Drivers that perform byte-order conversion: network drivers do this routinely, but they use portable helpers (`htonl`, `ntohs`, etc.) rather than architecture-specific code.
- Drivers that need specific CPU instructions: for example, cryptographic drivers that use AES-NI on amd64 or the AES extensions on arm64 need per-architecture code paths. These are rare and are already isolated by the kernel's crypto framework.

For most driver work on most architectures, the right mental model is “FreeBSD is FreeBSD”. The APIs are the same. The idioms are the same. The virtualisation framework is the same. The architecture is a detail the kernel manages for you.

With architecture considerations covered, we have the complete picture of virtualisation and containerisation as it affects a FreeBSD driver author. The remaining sections of the chapter turn to hands-on practice.

30.14 Hands-on Labs

The labs below walk you through four small exercises that put the chapter's ideas into practice. Each lab has a clear goal, a set of prerequisites, and a series of steps. Companion files are available under `examples/part-07/ch30-virtualisation/` for the ones that need more than a handful of lines of code.

Work through them in order. The first lab gets you comfortable inspecting a VirtIO guest from inside a VM. The second gets you writing a tiny VirtIO-adjacent driver. The third and fourth move into jails and VNET. Allow a couple of hours of hands-on time for the whole set; longer if the `bhyve(8)` or QEMU setup is new to you.

30.14.1 Lab 1: Exploring a VirtIO Guest

Goal: Confirm that you can start a FreeBSD 14.3 guest under `bhyve(8)`, log in, and observe the VirtIO devices it has attached. This establishes the development environment you will use for later labs.

Prerequisites: A FreeBSD 14.3 host (bare metal or nested in another hypervisor, provided the outer hypervisor supports nested virtualisation). Enough memory for a small guest (2 GB is plenty). The `vmm.ko` module loadable on the host, which it is on any standard FreeBSD 14.3 kernel.

Steps:

1. Fetch a FreeBSD 14.3 VM image. The base project provides prebuilt VM images at <https://download.freebsd.org>. Choose a bhyve-friendly image (typically the “BASIC-CI” or “VM-IMAGE” variants in the amd64 directory).
2. Install the `vm-bhyve` port or package: `pkg install vm-bhyve`. This wraps `bhyve(8)` in a friendlier management interface.
3. Configure a VM directory: `zfs create -o mountpoint=/vm zroot/vm` (or `mkdir /vm` if you are not using ZFS). In `/etc/rc.conf`, add `vm_enable="YES"` and `vm_dir="zfs:zroot/vm"` (or `vm_dir="/vm"`).
4. Initialise the directory: `vm init`. Copy a default template: `cp /usr/local/share/examples/vm-bhyve/default.conf /vm/.templates/default.conf`.
5. Create a VM: `vm create -t default -s 10G guest0`. Attach the downloaded image: `vm install guest0 /path/to/FreeBSD-14.3-RELEASE-amd64.iso`.
6. Log in once the installer finishes and reboots. The login prompt comes up on `vm console guest0`.
7. Inside the guest, run `pciconf -lvBb | head -40`. You will see `virtio-blk`, `virtio-net`, and possibly `virtio-random` devices, depending on the template. Note the driver each device is bound to.
8. Run `sysctl kern.vm_guest`. The output should be `bhyve`.
9. Run `dmesg | grep -i virtio`. Observe the attach messages for each VirtIO device.
10. Record your observations in a text file. You will refer to them in Labs 2 and 4.

Expected outcome: A running VM, a clear listing of attached VirtIO devices, and a confirmed `kern.vm_guest = bhyve` reading.

Common pitfalls: Not enabling VT-x or AMD-V in the host’s BIOS. Not having enough memory for the guest. Misconfigured network bridge preventing the guest from reaching the network (which is harmless for this lab but will matter later).

30.14.2 Lab 2: Using Hypervisor Detection in a Kernel Module

Goal: Write a small kernel module that reads `vm_guest` and logs the environment at load time. This is the smallest possible example of environment-aware driver behaviour.

Prerequisites: A working FreeBSD 14.3 guest from Lab 1. Kernel build tools installed (they come with the base system on the development kit). The `bsd.kmod.mk` build system, accessed via a simple `Makefile`.

Steps:

1. Under `/home/ebrandi/FDD-book/examples/part-07/ch30-virtualisation/lab02-detect/` in the companion examples tree, you will find a starter `detectmod.c` and `Makefile`. If you are not using the companion tree, create these files by hand.
2. The source file should define a kernel module that, on load, prints which environment it is in based on `vm_guest`.
3. Build the module: `make clean && make`.
4. Load it: `sudo kldload ./detectmod.ko`.
5. Check the `dmesg` output: `dmesg | tail`. You should see a line like `detectmod: running on bhyve`.

6. Unload: `sudo kldunload detectmod`.
7. Reboot the guest on a different hypervisor if possible (QEMU/KVM, for example) and re-run. The output should now reflect the new environment.

Expected outcome: A module that correctly identifies the hypervisor it is running on, using `vm_guest`.

Common pitfalls: Forgetting to include `<sys/systm.h>` for the `vm_guest` declaration. Linking errors from mistyped macros. Forgetting to set `KMOD=` and `SRCS=` in the Makefile.

30.14.3 Lab 3: A Minimal Character Device Driver Inside a Jail

Goal: Write a small character device driver, expose it from the host, create a jail with a custom devfs ruleset that makes the device visible inside, and verify that the jail's processes can use it while the host's privilege checks are still enforced.

Prerequisites: A FreeBSD 14.3 host. A working jail setup directory (typically `/jails/`). Basic familiarity with `make_dev(9)`, `d_read`, and `d_write`.

Steps:

1. Under `examples/part-07/ch30-virtualisation/lab03-jaildev/`, you will find `jaildev.c` and `Makefile`. The driver creates a `/dev/jaildev` character device whose read returns a fixed greeting.
2. Build and load the module on the host: `make && sudo kldload ./jaildev.ko`.
3. Verify `/dev/jaildev` exists and is readable: `cat /dev/jaildev`.
4. Create a jail root: `mkdir -p /jails/test && cp /bin/sh /jails/test/` (this is a minimal jail; for a real setup, use a proper filesystem layout).
5. Add a devfs ruleset to `/etc/devfs.rules`:

```
[devfsrules_jaildev=100]
add include $devfsrules_hide_all
add include $devfsrules_unhide_basic
add path 'jaildev' unhide
```

6. Reload the rules: `sudo service devfs restart`.
7. Start the jail: `sudo jail -c path=/jails/test devfs_ruleset=100 persist command=/bin/sh`.
8. In another terminal, enter the jail: `sudo jexec test /bin/sh`.
9. Inside the jail, run `cat /dev/jaildev`. The greeting should appear. Try `ls /dev/`: only the allowed devices (including `jaildev`) are visible.
10. Test the privilege boundary: modify the driver to require `PRIV_DRIVER` for an `ioctl`, rebuild, reload, and verify that the jail's root cannot run the `ioctl` while the host's root can.

Expected outcome: A driver visible in the jail only because the ruleset allows it, with privilege checks that behave differently for host root and jail root.

Common pitfalls: Forgetting to restart devfs after editing the rules. Not setting `persist` on the jail (without it, the jail dies as soon as the initial process exits). Misreading the ruleset syntax (whitespace is significant).

30.14.4 Lab 4: A Network Driver Inside a VNET Jail

Goal: Create a VNET jail, move one end of an `epair(4)` into it, and verify that network traffic flows between the host and the jail using only the jail's VNET. This exercises `if_vmove()` and the VNET lifecycle.

Prerequisites: A FreeBSD 14.3 host with `if_epair.ko` loadable. Root privileges.

Steps:

1. Load the `if_epair` module: `sudo kldload if_epair`.
2. Create an `epair`: `sudo ifconfig epair create`. You will get a device pair `epair0a` and `epair0b`.
3. Assign an IP to `epair0a` on the host: `sudo ifconfig epair0a 10.100.0.1/24 up`.
4. Create a jail root directory: `mkdir -p /jails/vnet-test`. Place a minimal shell and `ifconfig` binary inside (or bind-mount `/bin`, `/sbin`, `/usr/bin` for testing).
5. Create the jail with VNET enabled:

```
sudo jail -c \  
    name=vnet-test \  
    path=/jails/vnet-test \  
    host.hostname=vnet-test \  
    vnet \  
    vnet.interface=epair0b \  
    persist \  
    command=/bin/sh
```

The `vnet.interface=epair0b` parameter triggers the `if_vmove()` that moves the interface into the jail.

6. Enter the jail: `sudo jexec vnet-test /bin/sh`.
7. Inside the jail, configure the interface: `ifconfig epair0b 10.100.0.2/24 up`.
8. Still inside the jail, ping the host: `ping -c 3 10.100.0.1`. It should succeed.
9. From the host, ping the jail: `ping -c 3 10.100.0.2`. It should succeed.
10. Stop the jail: `sudo jail -r vnet-test`. The `epair0b` interface is moved back to the host (because it was moved with `vnet.interface`, which uses `if_vmove_loan()` under the hood in recent FreeBSD releases).
11. Verify the interface is back on the host: `ifconfig epair0b`. It should still exist but belong to the host's VNET again.

Expected outcome: A jail with its own network stack, moving an interface cleanly in and out.

Common pitfalls: Forgetting to enable VNET in the kernel (`options VIMAGE` is enabled in `GENERIC`, so this should be fine, but custom kernels might not have it). Trying to use a physical interface instead of `epair(4)` for the first attempt (this works but causes the host to lose that interface while the jail has it). Not giving the jail enough binaries to run a shell (the simplest workaround is a `nullfs` mount of the host's `/rescue` or a minimal bind-mount setup).

30.14.5 Lab 5: PCI Passthrough Simulation (Optional)

Goal: Observe how PCI passthrough changes a device's ownership, using a non-critical device as the target. This lab is marked optional because it requires a spare PCI device and an IOMMU-capable host. If those are not available, read through the steps; they illustrate the workflow even without running them.

Prerequisites: A FreeBSD 14.3 host with VT-d or AMD-Vi enabled in firmware. A spare PCI device that is safe to remove from the host (an unused NIC is a common choice). A bhyve(8) guest configuration.

Steps:

1. Identify the target device: `pciconf -lvBb | grep -B 1 -A 10 'Ethernet'` (or whatever type the spare device is). Note its bus, slot, and function (e.g., `pci0:5:0:0`).
2. Edit `/boot/loader.conf` and add the device to the passthrough list:

```
pptdevs="5/0/0"
```

3. Reboot the host. The device should now be bound to `ppt(4)` rather than its native driver. Confirm with `pciconf -l` (look for `ppt0`).
4. Configure a guest to pass the device through:

```
passthru0="5/0/0"
```

(Using the `vm-bhyve` configuration; the raw `bhyve` command is more verbose.)

5. Boot the guest. Inside the guest, run `pciconf -lvBb`. The device now appears in the guest with its real vendor and device IDs, attached to its native driver.
6. Exercise the device inside the guest: configure the NIC, send traffic, verify it works.
7. Shut down the guest. Edit `/boot/loader.conf` to remove the `pptdevs` line, reboot, and verify that the device returns to the host with its native driver attached.

Expected outcome: A clean round trip of a PCI device from host to guest and back, exercising the detach and reattach paths of the native driver.

Common pitfalls: The host firmware does not have VT-d or AMD-Vi enabled (look for it in the BIOS/UEFI setup). The chosen device is in the same IOMMU group as a device the host needs, forcing a multi-device passthrough. The device is attached to its native driver at boot before `ppt(4)` can claim it (usually this is fine if `pptdevs` is set early enough).

30.14.6 Lab 6: Building and Loading the vtdu Driver

Goal: Build the pedagogical `vtdu` driver from the case study, load it under a FreeBSD 14.3 guest, and observe its behaviour even without a matching backend. This lab exercises the kernel module build process in the VirtIO context and verifies the module structure.

Prerequisites: A FreeBSD 14.3 guest (the one from Lab 1 is fine). The FreeBSD source tree under `/usr/src` or the kernel headers installed (`pkg install kernel-14.3-RELEASE` works on stock systems). Root privileges inside the guest.

Steps:

1. Copy the companion files to the guest. From the book's examples tree, `examples/part-07/ch30-virtualisation/vtedu/` contains `vtedu.c`, `Makefile`, and `README.md`. Transfer them to the guest (via `scp`, `9p share`, or a shared volume).
2. Inside the guest, change into the `vtedu` directory: `cd /tmp/vtedu`.
3. Build the module: `make clean && make`. If the build fails because `/usr/src` is not installed, install it with `pkg install kernel-14.3-RELEASE` or point `SYSDIR` at an alternative kernel source tree: `make SYSDIR=/path/to/sys`.
4. A successful build produces `virtio_edu.ko` in the current directory.
5. Load the module: `sudo kldload ./virtio_edu.ko`. The load succeeds regardless of whether a matching device is present.
6. Check module status: `kldstat -v | grep -A 5 virtio_edu`. You will see the module is loaded, but no device is bound. This is expected without a backend.
7. Unload the module: `sudo kldunload virtio_edu`.
8. Inspect the module's PNP information: `kldxref -d /boot/modules` lists modules and their PNP entries. If you moved the module into `/boot/modules`, you will see the `VirtIO` simple PNP entry advertising device ID `0xffff0`.
9. Look at `dmesg` output during load and unload. There should be no errors. The absence of an attach message confirms that no device is bound, which is the expected behaviour.

Expected outcome: A successful build and load cycle that demonstrates the module is well-formed. Without a backend, the module is inert; this is a useful confirmation that the build and load plumbing work independently of the device side.

Common pitfalls: Missing kernel sources (install the `src` package). Missing `virtio.ko` dependency (load `virtio` first if it was somehow absent, though it is built into `GENERIC`). Confusion when no device attaches (re-read Section 1 of the `vtedu` `README`; this is by design).

What to do next: The real learning comes from pairing this module with a backend. Challenge 5 in the Challenge Exercises describes writing a matching backend in `bhyve(8)`. If you complete that, the driver will attach to the backend-provided device, a `/dev/vtedu0` node will appear, and you will be able to `echo hello > /dev/vtedu0 && cat /dev/vtedu0` to exercise the full `VirtIO` round-trip you studied in the case study.

30.14.7 Lab 7: Measuring VirtIO Overhead

Goal: Quantify the performance characteristics of `VirtIO` by running a simple workload under emulated, paravirtualised, and passthrough device configurations. This lab is about building intuition for what virtualisation costs, not about optimising a specific driver.

Prerequisites: A FreeBSD 14.3 host with `bhyve`, at least 8 GB of RAM, and the `vm-bhyve` tool installed. A NVMe drive is ideal but not required. The `fio(1)` benchmarking tool (available from ports as `benchmarks/fio`). Optional: a spare NIC for passthrough comparison.

Steps:

1. Create a baseline guest with `VirtIO` block and network devices (this is the default in `vm-bhyve`).
2. Inside the guest, install `fio`: `pkg install fio`.
3. Run a baseline disk benchmark: `fio --name=baseline --rw=randread --bs=4k --size=1G --numjobs=4 --iodepth=32 --runtime=30s --group_reporting`. Record the IOPS and latency.

4. On the host, measure the same workload directly (not inside the guest) using the backing storage as the target. Compare.
5. Run a baseline network benchmark: `iperf3 -c 10.0.0.1 -t 30` (server on the host, client in the guest). Record the throughput.
6. If you have a spare NIC, reconfigure the guest to use passthrough for the NIC (see Lab 5). Re-run the `iperf3` benchmark. Compare.
7. On the host, observe interrupt and VM exit counts while the benchmarks run: `vmstat -i, pmcstat -S instructions -l 10`. The most interesting counters are VM exit rates, which correlate with overhead.
8. For the disk benchmark, try varying the VirtIO features. Modify the guest configuration to disable `VIRTIO_F_RING_EVENT_IDX` (if the backend supports disabling it) and observe the change in interrupt rate.

Expected outcome: A small set of numbers that quantify the cost of virtualisation for your particular setup. Typically, VirtIO paravirtualised I/O is within 10-20% of bare metal for sustained workloads and within 30-40% for latency-sensitive random workloads; passthrough is within a few percent of bare metal but forfeits flexibility; pure emulation (for example, emulated E1000 instead of virtio-net) is 5-10x slower than VirtIO and should be avoided for any serious workload.

Common pitfalls: Benchmarks can be noisy; run each multiple times and take medians rather than single samples. The host's cache state affects disk benchmarks; run a warmup pass before the measured runs. CPU frequency scaling can skew results; pin the guest to specific cores and disable scaling on the host for reproducibility.

What this teaches: The numbers are secondary; the method is primary. Being able to measure overhead and attribute it to a specific layer (emulation vs. paravirtualisation vs. passthrough, interrupt rate vs. throughput, guest CPU vs. host CPU) is the foundation of performance work in virtualised environments. The same technique applies to any driver you write: if it has a performance requirement, you need to measure, and the measurement techniques are the ones in this lab.

30.14.8 A Note on Labs You Cannot Complete Today

Not every reader will have the hardware for every lab. Labs 1 through 4 are doable on essentially any FreeBSD 14.3 machine with enough memory and the `vmm` module. Lab 5 requires specific hardware that many readers will not have. Treat Lab 5 as a read-along walkthrough if you cannot run it; the concepts translate to any passthrough scenario, and the exact commands are documented in `pci_passthru(4)` and the `bhyve(8)` manual page.

If you are blocked on a lab, write down exactly what went wrong and come back to it. Virtualisation and containerization are areas where small environmental details can derail a whole setup, and the diagnostic skill of narrowing down the failure is as important as completing the lab.

30.15 Challenge Exercises

The labs above guide you through the standard paths. The challenges below ask you to extend the work. They are harder, less prescriptive, and designed to reward experimentation. Pick whichever one interests you most; all of them stretch different muscles.

30.15.1 Challenge 1: Extend the Detect Module

The Lab 2 module reads `vm_guest` and prints an environment label. Extend it so that it also reads the CPU vendor string (using `cpu_vendor` and `cpu_vendor_id`), the total physical memory (`realmem`), and the hypervisor signature where applicable. Produce a single structured log line that a test script can parse.

The CPU vendor string is a well-known part of CPU identification; look at `/usr/src/sys/x86/x86/identcpu.c` to see how the kernel reads it. The physical memory total is exposed through the `realmem` global and through the `hw.physmem` sysctl. The hypervisor signature lives in CPUID leaf `0x40000000` on most hypervisors; reading it requires a small piece of assembly or an intrinsic.

The interesting design question is how to expose this information to user space. You could log it at module load, add a sysctl, or create a `/dev/envinfo` device. Each has trade-offs. Think about which is most appropriate for a real production driver.

30.15.2 Challenge 2: A Simulation Backend for a Real VirtIO Driver

Chapter 29’s techniques encourage splitting a driver into accessors, backends, and a thin upper layer. Apply this to `virtio_random`. The real driver (in `/usr/src/sys/dev/virtio/random/virtio_random.c`) is small and tightly written. Can you refactor it so that the `virtqueue` operations go through an accessor layer, and a simulation backend that does not need a real VirtIO device can be selected for tests?

The refactor is subtle because the VirtIO framework provides most of the accessor layer for you: `virtqueue_enqueue`, `virtqueue_dequeue`, `virtio_notify` are already abstractions. The challenge is to find a layer above them that can be swapped out. One possibility is to move the harvest loop into a function that takes a callback for “get one buffer’s worth of data”, and implement that callback either as “call the `virtqueue`” (real) or as “read from `/dev/urandom` on the host” (simulation).

This is a design exercise more than a coding exercise. The goal is to understand how far Chapter 29’s advice can be stretched in a real driver that already has good abstractions.

30.15.3 Challenge 3: A VNET-Aware Driver Skeleton

Write a skeleton kernel module that creates a pseudo-network interface (using the `if_clone(9)` framework), supports being moved between VNETs, and reports its VNET identity in a sysctl. Verify with a VNET jail that the interface can be moved into the jail, used, and moved back.

The key subtlety is handling the VNET context correctly. Read the code in `/usr/src/sys/net/if_tuntap.c` for a reference. The `if_vmove` lifecycle requires the driver to clean up per-VNET state when the interface leaves and re-create it when the interface arrives. Pay attention to `VNET_SYSINIT` and `VNET_SYSUNINIT` if your driver needs per-VNET state.

This is a deep challenge that will take you into the VNET internals. Do not expect to complete it in a single sitting. Treat it as a multi-day project.

30.15.4 Challenge 4: A Jail-Visible Status Device

Write a driver that exposes a `/dev/status` character device. The device’s read returns different information depending on whether the caller is in a jail. If the caller is on the host, it returns system-wide status. If the caller

is in a jail, it returns jail-specific status (number of processes, current memory use, and so on).

The interesting part is how to distinguish the caller's jail. The `struct thread` passed to the `read` method has `td->td_ucred->cr_prison`, and `prison0` is the host. From the `prison` pointer you can read the jail's name, its process count (`pr_nprocs`), and so on. Be careful about locking: the `prison`'s fields are mostly read under a mutex that the driver must acquire.

This challenge is a good way to learn about the jail API without writing anything hypervisor-related. It also teaches you about `struct thread` and `struct ucred`, which are central to FreeBSD's privilege model.

30.15.5 Challenge 5: A bhyve Emulated Device in User Space

If you are ready for a bigger project, study how `bhyve(8)` emulates a simple VirtIO device (for example, `virtio-rnd`) in user space, and write a new emulated device of your own. The easiest target is a “hello” device that returns a fixed string through a VirtIO interface. The guest-side driver is your Chapter 29 or Chapter 30 work; the host-side emulator lives in `/usr/src/usr.sbin/bhyve/`.

This exercise ties together everything in the chapter: you write a VirtIO device in user space on the host, your driver inside the guest reads from it, the `virtqueue` transport between them is what you have been studying, and the whole thing exercises the full guest-host loop.

It is a substantial project, and it requires some familiarity with `bhyve(8)`'s code structure. Consider it a reach goal. If you finish it, you have genuinely learned both sides of the VirtIO story.

30.15.6 Challenge 6: Container Orchestration for Driver Testing

Write a shell script (or a more elaborate tool) that automates the Lab 3 and Lab 4 workflows. Given a driver and a test harness, the script should:

1. Build the driver.
2. Load it on the host.
3. Create a jail (or a VNET jail) with an appropriate ruleset.
4. Run the harness inside the jail.
5. Collect results.
6. Destroy the jail.
7. Unload the driver.
8. Report pass/fail with diagnostics.

The script should be idempotent (running it twice should not leave residue) and it should handle common failures gracefully. The end result is a tool you can run in CI to verify that your driver continues to work under jails as it evolves.

This is not a deep technical challenge but a practical one. Building this kind of automation is a significant fraction of real-world driver work, and building it yourself once teaches you what is involved.

30.15.7 How to Approach These Challenges

Each challenge could be a weekend project. Pick one that seems interesting and set aside time for it. Do not try to do all of them at once; your energy for unfamiliar work is limited, and you learn more from one challenge

finished than from three challenges half-done.

If you get stuck, do two things. First, re-read the relevant section of the chapter; the hints are there. Second, look at real FreeBSD drivers that do something similar. The source tree is your best teacher for the idiomatic way to solve problems that have been solved before.

30.16 Troubleshooting and Common Mistakes

This section collects the problems that driver authors most often hit when working with virtualised and containerised environments. Each entry describes the symptom, the likely cause, and the way to verify and fix it. Keep this as a reference; the first time you see a symptom, it will be new, and the next time it will be familiar.

30.16.1 VirtIO Device Not Detected in the Guest

Symptom: The guest boots, but `pciconf -l` does not show the VirtIO device, and `dmesg` contains no `virtio` attach messages.

Cause: Either the host did not configure the device (wrong `bhyve` command line, wrong `vm-bhyve` template), or the guest kernel is missing the VirtIO module. On FreeBSD 14.3, VirtIO drivers are built into `GENERIC` and auto-load; the guest side should almost never be the cause.

Fix: On the host, verify that the `bhyve` command line includes the device. For `vm-bhyve`, look at the VM's configuration file and check for lines like `disk0_type="virtio-blk"` or `network0_type="virtio-net"`. If the device is listed but still missing, check that the hypervisor has permission to access the backing resource (the disk image, the tap device).

Inside the guest, confirm the kernel has VirtIO: `kldstat -m virtio` or `kldstat | grep -i virtio`. If the module is not loaded, try `kldload virtio`.

30.16.2 virtqueue_enqueue Fails With ENOBUFS

Symptom: A VirtIO driver tries to enqueue a buffer and gets `ENOBUFS`.

Cause: The `virtqueue` is full. Either the driver has not been draining completed buffers through `virtqueue_dequeue`, or the ring size is smaller than expected and the driver is enqueueing more than it can hold.

Fix: Call `virtqueue_dequeue` in the interrupt handler to drain completed buffers and free their associated state. If the ring is genuinely too small, negotiate a larger ring size at feature-negotiation time, if the device supports `VIRTIO_F_RING_INDIRECT_DESC` or similar features.

A common beginner error is to forget that `enqueue` produces a descriptor that must be matched by a `dequeue`. Every successful `enqueue` must eventually produce a `dequeue`; otherwise the ring fills up.

30.16.3 Passthrough Device Not Available in the Guest

Symptom: The host has marked a device as passthrough-capable, the guest is configured to use it, but the guest does not see the device.

Cause: Several possible. The host did not actually bind the device to `ppt(4)` at boot (check `pciconf -l` for `pptN`). The host's firmware does not have the IOMMU enabled (check `dmesg | grep -i dmar` or `grep -i iommu`). The device is in an IOMMU group that cannot be split (so you need to pass through the whole group).

Fix: Verify each of the above in turn. `dmesg | grep -i dmar` should show initialisation messages from `dmar(4)` on Intel hosts or `amdvi(4)` on AMD hosts. If those are missing, enable VT-d or AMD-Vi in the firmware. If the device is in a shared IOMMU group, either pass through the whole group or move the device to a different PCI slot that is better isolated (an administrator task requiring chassis access).

30.16.4 Kernel Panic at Module Load Inside a Guest

Symptom: `kldload` of a new driver inside a guest causes a kernel panic.

Cause: Usually a bug in the driver, but sometimes a VirtIO-specific one: the driver assumes a device feature that the backend does not provide, or it accesses config space using the wrong layout (legacy versus modern).

Fix: Use the guest as your development platform precisely to make panics cheap. Capture the panic output (either from the serial console or from a bhyve log), narrow down the faulting function with `ddb(4)` if you have it configured, and iterate. The combination of a VM (so panics are cheap) and `printf` debugging (so you can see what happens before the panic) is usually enough to fix a VirtIO driver bug quickly.

For VirtIO-specific bugs, double-check the feature bits your driver negotiates. A driver that claims a feature but then does not implement it correctly (for example, claims `VIRTIO_F_VERSION_1` but uses legacy config space) will break in surprising ways.

30.16.5 Jail Cannot See a Device Even After Unhiding

Symptom: A devfs ruleset includes the device, the ruleset is applied to the jail, but the jail's `ls /dev` does not show it.

Cause: The devfs mount inside the jail was not re-mounted or re-ruled after the ruleset change. devfs caches the visibility decision at mount time, so later changes do not propagate until the mount is refreshed.

Fix: Run `devfs -m /jails/myjail/dev rule -s 100 applyset` (replace the path and ruleset number as appropriate) to force a reapply. Alternatively, stop the jail, restart devfs, and start the jail again. The service `devfs restart` command applies the rules in `/etc/devfs.rules` to the host's `/dev`; for jails, you generally need to restart the jail.

30.16.6 Privilege Denied Inside a Jail That Should Work

Symptom: A driver operation works on the host as root but fails with `EPERM` inside a jail's root.

Cause: The operation requires a privilege that `prison_priv_check` denies by default. This is the intended behaviour. The fix is either to use a less-privileged operation, to configure the jail with `allow.*` to grant the privilege, or (if appropriate) to change the driver to use a different privilege.

Fix: Look at the driver's `priv_check` call and identify which privilege is being checked. Consult `/usr/src/sys/sys/priv.h` and `prison_priv_check` in `/usr/src/sys/kern/kern_jail.c` to see whether the privilege is allowed inside jails. If it is not, consider whether the restriction is appropriate (usually yes) and adjust the jail's configuration accordingly (`allow.raw_sockets`, `allow.mount`, etc.).

Do not be tempted to remove the `priv_check` call just to make the jail work. The check is there for a reason; work with the framework, not against it.

30.16.7 VNET Jail Cannot Send or Receive Packets

Symptom: A VNET jail is created, an interface is moved in, but network traffic does not flow.

Cause: Several possibilities. The interface was not configured inside the jail (each VNET has its own `ifconfig` state). The default route was not set inside the jail. The host's firewall is blocking traffic between the jail's interface and the rest of the network. The interface is not in an UP state.

Fix: Inside the jail, run `ifconfig` and confirm the interface has an address and is UP. Run `netstat -rn` and confirm the routing table has the entries you expect. If you are using `pf(4)` or `ipfw(8)` on the host, check the rules: filter rules apply to the host's VNET, and the jail's packets may be rejected if the host's filter blocks them (depending on how the topology is set up).

For `epair(4)` setups specifically, remember that both ends need configuration, and that the host side stays on the host. The jail configures the end that was moved in; the host configures the end that stayed.

30.16.8 PCI Passthrough Driver Fails to Attach in the Guest

Symptom: The guest sees the passed-through device in `pciconf -l`, but the driver fails to attach, or the attach succeeds but read/write fails.

Cause: Usually one of two things. Either the driver assumes a platform-specific feature that is not present in the guest (an ACPI table, a BIOS entry), or the driver is programming physical addresses directly rather than going through `bus_dma(9)`, and the IOMMU is redirecting DMA in a way the driver does not expect.

Fix: For the first case, look at the driver's attach code for calls that read platform tables (ACPI, FDT, etc.). If the guest's firmware does not expose the expected tables, the driver must either provide defaults or refuse gracefully.

For the second case, audit the driver's DMA code. Every address programmed into the device must come from a `bus_dma(9)` load operation, not from a `vtophys` call or similar. This is standard practice anyway, but it becomes mandatory under passthrough.

30.16.9 Host Becomes Unresponsive When a Guest Starts

Symptom: Running `bhyve(8)` makes the host sluggish or unresponsive.

Cause: Resource contention. The guest is allocated more memory or more VCPUs than the host can afford to share. The host is swapping or spinning on a shared lock.

Fix: Check the guest's resource allocation. A guest with all the host's memory will starve the host; a guest with more VCPUs than the host has cores will cause thrashing. A common rule of thumb is to give guests no more than half the host's memory and no more VCPUs than `(host cores - 1)`, to leave room for the host itself.

If the sluggishness persists after sensible allocation, check `top -H` on the host for the `bhyve(8)` process and its threads. Heavy CPU use by `bhyve(8)` suggests the guest is doing something CPU-intensive; heavy CPU use by `vmm` kernel threads suggests excessive VM exits, which may indicate a guest driver that is polling too aggressively.

30.16.10 `kldunload` Hangs

Symptom: Unloading a driver module hangs the process and cannot be interrupted.

Cause: Some resource the driver owns is still in use. A file descriptor is still open on the driver's device, a callout is still scheduled, a taskqueue still has pending tasks, or a kernel thread the driver spawned has not exited.

Fix: Find and release the holder. `fstat` or `lsuf` lists open file descriptors; `procstat -kk` shows kernel threads. The driver's module unload handler must drain every async mechanism it starts: cancel callouts, drain taskqueues, wait for kernel threads to exit, close any held file descriptors. If any of these is missing, unload hangs.

For VNET-aware drivers, the unload must correctly clean up per-VNET state. A common mistake is to clean up in `mod_event(MOD_UNLOAD)` but forget that one of the VNETs the module is attached to is not the current one; accessing its state without `CURVNET_SET` leads to either a wrong-context access (fast) or a hang (slow). The correct pattern is to iterate over VNETs and clean up each explicitly.

30.16.11 Timing-Related Bugs Only in VMs

Symptom: The driver works on bare metal but hangs or loses interrupts inside a VM.

Cause: Timing assumptions that fail under virtualisation. Guest execution is sometimes paused for milliseconds at a time (during VM exits), and a driver that polls a status register in a tight loop without yielding may fail to make progress or may consume excessive CPU.

Fix: Replace tight polling with `DELAY(9)` for microsecond-scale waits, `pause(9)` for short waits, or proper sleep with `tsleep(9)` for longer waits. Use interrupt-driven designs instead of polling wherever possible. Test with `virtio-blk`'s performance counters to see whether the driver is generating an unreasonable number of VM exits.

A driver that works correctly on bare metal but fails on a VM is almost always making a timing assumption. The solution is to use the kernel's time primitives correctly; they work on both.

30.16.12 VirtIO Negotiation Fails or Returns Unexpected Features

Symptom: The driver logs that feature negotiation produced a feature set that is missing bits you expected, or the probe-and-attach path succeeds but the device behaves unexpectedly.

Cause: Two classes of issue. The first is that the device (or its backend) advertises a feature set that does not include the bit you requested. This is normal when the hypervisor's backend is older or intentionally minimal. The second is that the guest-side code is requesting a feature bit that the framework does not know about, in which case the framework may strip it silently.

Fix: Log `sc->features` immediately after `virtio_negotiate_features` returns. Compare it to the set you requested. If the device is missing a bit you thought was mandatory, your driver needs to fall back gracefully or refuse to attach with a clear error message. Never assume a bit is present without checking the post-negotiation value.

For backend-side investigation (if you are using a hypervisor whose source you can read, such as `bhyve(8)` or `QEMU`), look at the device emulator's feature advertisement. The backend holds the truth: the guest sees only

what the backend advertises. A mismatch between what you expect and what you see almost always traces back to the backend.

30.16.13 `bus_alloc_resource` Fails Inside a Guest

Symptom: The attach path calls `bus_alloc_resource_any` or `bus_alloc_resource` and receives `NULL`, causing the driver to fail attach.

Cause: Under a hypervisor, the device's resources (BARs, IRQ lines, MMIO windows) may differ from their bare-metal layout. A driver that hard-codes resource IDs or assumes specific BAR numbers can fail if the hypervisor presents a different layout.

Fix: Always use `pci_read_config(dev, PCIR_...)` to read actual BAR contents rather than assuming. Use `bus_alloc_resource_any` with the rid obtained from the resource list, not a hard-coded number. If the resource allocation still fails, compare `pciconf -lvBb` output from bare metal and from the guest to see what has changed.

A concrete example: a device that uses BAR 0 for MMIO and BAR 2 for I/O on bare metal may be configured differently by the hypervisor. Always read the BARs at runtime and allocate resources based on what is actually present.

30.16.14 `kldload` Succeeds but No Device Attaches

Symptom: `kldload` of a VirtIO driver returns success, but `kldstat -v` shows no device bound to the module, and no `dmesg` messages are produced.

Cause: The driver's PNP table does not match any device advertised by the hypervisor. This is normal when a backend is not providing the expected device. For `vtedu` in the case study, this is the expected behaviour without a matching `bhyve(8)` backend.

Fix: Run `devctl list` (or `devinfo -v`) on the host or in the guest to see which devices are present but unbound. If the device is not listed at all, the backend is not running or is misconfigured. If the device is listed but unbound, check its PNP identifiers (vendor, device for PCI, or the VirtIO type ID for VirtIO) and compare against the driver's PNP table. Mismatch is the most common cause.

A common beginner error is to believe that `kldload` success means the driver is working. It only means the module loaded. Use `kldstat -v | grep yourdriver` to see whether any device has been claimed.

30.16.15 Module Is Loaded but `/dev` Node Does Not Appear

Symptom: The driver has loaded, `kldstat -v` shows it attached to a device, but the expected `/dev` node does not appear.

Cause: Either the driver did not call `make_dev(9)`, or the `devfs` mount inside the current jail is filtered by a ruleset that hides the node, or the `make_dev` call failed silently because the device unit number clashed with an existing one.

Fix: On the host, check `ls /dev/yourdev*`. If it is missing there too, the driver did not create the node. Check the attach path for the `make_dev` call, and verify its return value. If the node is present on the host but missing

inside a jail, the devfs ruleset is the cause. Run `devfs -m /path/to/jail/dev rule show` to see the active ruleset inside the jail.

For a driver that is intended to be visible inside jails, the correct practice is to document which devfs ruleset exposes the node and to provide an example ruleset in the driver's README. Do not assume the administrator will figure it out.

30.16.16 Virtqueue Interrupts Never Fire

Symptom: The driver's interrupt handler is never called, even though the driver has submitted work to the virtqueue.

Cause: One of several possibilities. The backend never processes the work (bug in the backend). The driver did not register the interrupt handler correctly. The driver did not call `virtio_setup_intr`, so no interrupt plumbing exists. The driver disabled interrupts via `virtqueue_disable_intr` and never re-enabled them.

Fix: Systematically check each step. In the attach path, verify that `virtio_setup_intr` was called and returned 0. In the interrupt handler, verify that you are re-enabling interrupts when appropriate. Add a `printf` at the top of the handler to confirm it is never called. If the handler is genuinely never called, run `vmstat -i | grep yourdriver` to see the interrupt count; a count of zero confirms the interrupt is not arriving.

If the count is nonzero but the handler does no work, the handler is running but finding nothing in the virtqueue. This suggests the backend is acknowledging but not producing real completions; look at the backend.

30.16.17 Interrupt Storm Under Virtualisation

Symptom: Inside a VM, `vmstat -i` shows interrupt rates in the hundreds of thousands per second, and CPU utilisation is high even without real work.

Cause: An interrupt that is not being cleared, or a driver that interrupts on every event without coalescing. Under INTx specifically, a level-triggered interrupt that remains asserted causes the handler to be called in a loop.

Fix: For MSI-X, confirm that the handler acknowledges completions and that the virtqueue ring is being drained. For INTx, confirm that the handler clears the device's interrupt status register. For VirtIO specifically, negotiate `VIRTIO_F_RING_EVENT_IDX` if the backend supports it; this lets the device suppress unnecessary interrupts.

Look at the Section 9 pattern with `virtqueue_disable_intr` / `virtqueue_enable_intr`. A correct driver disables interrupts on entry, drains the ring, and only re-enables interrupts when the ring is empty. Missing this structure is a common cause of interrupt storms.

30.16.18 DMA Failures Only Under Passthrough

Symptom: The driver works correctly with an emulated device, but when the same hardware is passed through via `ppt(4)`, DMA transfers fail silently or corrupt memory.

Cause: Most often, the driver is programming physical addresses directly rather than going through `bus_dma(9)`. Under emulation, the hypervisor intercepts all I/O and translates addresses on the fly, hiding the

bug. Under passthrough, the device DMA's directly through the IOMMU, and the physical address the driver programmed is not the bus address the IOMMU expects.

Fix: Audit every place the driver computes an address to program into the device. Each must come from `bus_dma_load` or `bus_dma_load_mbuf` or similar, not from `vtophys` or a raw physical address. This is a mandatory discipline for passthrough and is strongly recommended for all drivers.

A useful diagnostic: enable IOMMU verbose logging (`sysctl hw.dmar.debug=1` on Intel, or the equivalent on AMD) and watch the kernel log for IOMMU page faults while the driver runs. A page fault reveals exactly which bus address the device attempted to access; if it does not match a mapped region, the driver's address calculation is wrong.

30.16.19 A VirtIO Device Appears but Has the Wrong Type

Symptom: The guest sees a VirtIO device at the expected PCI address, but `pciconf -lv` or `devinfo` reports a different VirtIO device type than expected.

Cause: Device-ID confusion. The PCI vendor ID for VirtIO is always `0x1af4`, but the device ID encodes the VirtIO type, and different VirtIO versions use different device-ID ranges. Legacy VirtIO uses `0x1000 + VIRTIO_ID`. Modern VirtIO uses `0x1040 + VIRTIO_ID`. A hypervisor that exposes a mix of modern and legacy devices can confuse a driver that only probes one range.

Fix: FreeBSD's `virtio_pci` transport handles both ranges transparently, so most drivers are immune. For driver authors who inspect `pciconf -lv` directly, be aware that both `0x1000-0x103f` (legacy) and `0x1040-0x107f` (modern) are VirtIO. The `VIRTIO_DRIVER_MODULE` macro registers the driver with both transports and does the right thing for both device-ID ranges.

30.16.20 Jail-Specific Ioctl Fails With ENOTTY

Symptom: An ioctl that works on the host returns `ENOTTY` when issued from inside a jail, even though the ioctl number is recognised by the driver.

Cause: The driver's ioctl handler checks jail visibility and returns `ENOTTY` to hide the ioctl's existence from jailed callers. This is a security-by-obscurity pattern used by some drivers that expose host-only administrative operations through otherwise jail-visible devices.

Fix: If the jail should be able to use the ioctl, review the driver's visibility check. The idiomatic approach is to return `EPERM` (permission denied) rather than `ENOTTY` when an operation exists but is not permitted; `ENOTTY` implies the ioctl does not exist, which can confuse callers. Consider whether the jailed caller should see the ioctl; if so, remove the hiding logic and use `priv_check` for access control instead.

30.16.21 VNET Move Leaks Per-VNET State

Symptom: After moving an interface in and out of a VNET multiple times, the kernel's memory usage grows, eventually triggering a memory pressure event.

Cause: The driver allocates per-VNET state when an interface enters a VNET but does not free it when the interface leaves. Each VNET move leaks a fixed amount of memory.

Fix: Implement the VNET-move lifecycle correctly. When an interface enters a VNET (`if_vmove` into the new VNET), allocate per-VNET state. When it leaves (`if_vmove` out), free it. The `CURVNET_SET` and `CURVNET_RESTORE` pair delineates the VNET context; use them when allocating or freeing.

Look at `/usr/src/sys/net/if_tuntap.c` for a correct VNET-move implementation. The lifecycle is subtle and easy to get wrong; a reference implementation is the best teacher.

30.16.22 Guest Kernel Panics with “Fatal Trap 12” on First I/O

Symptom: The guest boots, the driver attaches, the first user-space I/O to the driver’s device causes a “Fatal Trap 12: page fault while in kernel mode” panic.

Cause: Almost always a NULL pointer dereference in the driver’s `read`, `write`, or `ioctl` path. Under virtualisation, the fault is immediate rather than merely corrupt-then-continue, because the guest’s memory protection is exact.

Fix: Use the kernel’s debugger (`ddb(4)` or `dtrace(1)`) to find the faulting instruction. A typical cause is a `dev->si_drv1 = sc` that was forgotten in `attach`, so when user space opens `/dev/yourdriver` and calls `read`, `dev->si_drv1` is NULL. The fix is to always set `si_drv1` in `attach`, right after creating the `cdev`.

Under VMs, these panics are cheap: fix the code, rebuild, reload. On bare metal, each panic costs a reboot. One more reason to develop under virtualisation.

30.16.23 Live Migration Fails or Causes Guest Hang

Symptom: A guest under a hypervisor that supports live migration (currently limited in `bhyve(8)`, more common in Linux KVM and VMware) is migrated to a different host, and the guest hangs or corrupts after migration.

Cause: The guest driver holds state that is tied to the source host (a specific physical TSC, a specific IOMMU mapping, a specific passed-through device). Migration transfers the guest’s memory and CPU state but cannot transfer physical hardware state.

Fix: For driver authors, the advice is simple: do not cache values that are tied to the physical host. Re-read TSC frequency from `timecounter(9)` rather than storing a local copy. Do not pass through PCI devices that you need to migrate. For VirtIO devices, live migration is supported by the standard, and the guest driver needs no special code.

If you are writing a driver for a live-migration-supporting environment, the main design rule is: all state should be in the guest’s memory; anything on the host side should be recreatable after migration. Standard VirtIO drivers meet this bar because the `virtqueue` state is in guest memory.

30.16.24 Unexpected devfs Entries Appear Inside a Jail

Symptom: A jail has a minimal devfs ruleset, but entries appear that the administrator did not expect.

Cause: Either the ruleset was not applied correctly, or the jail inherited the default ruleset before the custom one was applied, or a new device appeared after the ruleset was set.

Fix: Run `devfs -m /jail/path/dev rule show` to see what rules are active. Compare with `/etc/devfs.rules`. If a later rule is adding visibility the earlier rule denied, the order is wrong. If the jail was started before the ruleset was finalised, restart the jail.

A robust practice is to always start a jail with a known ruleset specified in `/etc/jail.conf` rather than relying on the `devfs` default. The `devfs_ruleset = NNN` directive in the jail configuration ensures the jail's `devfs` mount uses the expected ruleset from the moment the jail starts.

30.16.25 Two Drivers Fight Over the Same Device

Symptom: Loading driver A works, but loading driver B after A (or vice versa) causes one of them to fail attach with a cryptic message about resource conflicts.

Cause: Two drivers are trying to claim the same device. This can happen if the device has multiple valid drivers (for example, a generic driver and a specific driver for a particular chipset variant) and the load order determines which one wins.

Fix: FreeBSD's Newbus arbitrates driver priority through `DRIVER_MODULE`'s ordering, but the exact semantics depend on which driver attached first. The general rule is that once a device is attached, another driver cannot steal it. If you need to switch drivers, detach the first one (`devctl detach yourdev0`) before loading the second.

Under virtualisation this can show up when you load a test driver after the VirtIO framework has already attached a production driver to the device. The test driver must either use a different PNP entry or explicitly detach the existing one.

30.16.26 vm_guest Shows “no” Inside an Obvious VM

Symptom: `sysctl kern.vm_guest` returns “no” inside a guest that is definitely running under a hypervisor.

Cause: The hypervisor is not setting the `CPUID` hypervisor-present bit, or is setting a vendor string that the FreeBSD kernel does not recognise. This can happen with exotic or customised hypervisors.

Fix: This is informational, not a fix. If your driver needs to detect virtualisation but `vm_guest` does not cooperate, use alternative signals: the presence of VirtIO devices, the absence of physical hardware that would be present on bare metal, specific `CPUID` leaves that expose hypervisor-specific information. But recognise that accurate hypervisor detection is hard to guarantee; design your driver to work correctly regardless of the environment, and use `vm_guest` only for non-critical defaults.

30.16.27 General Diagnostic Approach

When something breaks in a virtualised or containerised environment, the pattern for diagnosis is consistent. First, reduce the environment to the simplest setup that shows the problem: strip the VM down to a single VirtIO device, remove extra jails, disable VNET if it is not needed. Second, try the same operation at the next layer up: if the problem shows in a jail, try it on the host; if on the host, try it on a guest; if in a guest, try it on bare metal. The layer where the problem disappears is usually the layer where the problem lives.

Third, once you have localised the layer, add logging. `printf` in the kernel is still a valid debugging tool; combined with `dmesg`, it gives you a timestamped trace of what the driver does. `dtrace(1)` is more capable but has a higher setup cost. For a first diagnosis, `printf` is usually enough.

Fourth, if the problem is a race or a timing issue, simplify before you complicate. A race that happens once in ten thousand iterations can be investigated with a test loop that runs ten thousand iterations. A timing issue under VM execution can be made reproducible by pinning the VM to specific host CPUs and disabling frequency scaling.

None of this is specific to virtualisation. It is good general debugging discipline. Virtualisation just happens to be an environment where the discipline pays off quickly, because the layers are clearly separated and the layers can be swapped in and out easily.

30.17 Wrapping Up

Virtualisation and containerization are two names for what they are in FreeBSD: two different answers to the same broad question. Virtualisation multiplies kernels; containerization subdivides one. A driver lives inside a kernel, and the way it interacts with its environment depends on which of these is in use.

The virtualisation side is where the VirtIO story lives. VirtIO is the mature, standardised, high-performance paravirtual interface between a guest driver and a hypervisor-provided device. FreeBSD's VirtIO framework (in `/usr/src/sys/dev/virtio/`) implements the standard cleanly, and the `virtio_random` driver you studied in Section 3 is a minimal example of how a VirtIO driver is structured. The key concepts, feature negotiation, virtqueue management, notification, are the same whether the device is a random-number generator, a network card, or a block device. Learn them once, and every VirtIO driver becomes more approachable.

The hypervisor detection mechanism, `vm_guest`, gives a driver a small window into the environment. It is useful for adjusting defaults but dangerous when used to work around bugs. The right mindset is “this is informational”; the wrong mindset is “this is a branch target”.

The host side, where FreeBSD runs `bhyve(8)` and provides devices to guests, is where driver authorship meets hypervisor authorship. Most driver authors never touch `vmm(4)` directly, but the PCI passthrough facility is worth understanding, because it exercises the detach and reattach paths that a well-written driver already supports. A driver that survives a round trip through `ppt(4)` is a driver whose lifecycle is honest.

The containerization side is where jails, `devfs` rulesets, and VNET come together. Jails share the kernel, so drivers are not multiplied; they are filtered. The filter operates on three axes: which devices the jail can see (`devfs` rulesets), which privileges it can exercise (`prison_priv_check` and `allow.*`), and how much it can consume (`rctl(8)` and `racct(9)`). For a driver author, the design decisions are small and local: pick sensible `devfs` names, call `priv_check` correctly, handle allocation failures gracefully.

The VNET framework extends the jail model into the network stack. It is the part of jails that comes closest to requiring explicit driver cooperation. Drivers that write to per-VNET state from outside a network-stack entry point (for example, from callouts) must establish the VNET context with `CURVNET_SET`. Drivers that survive `if_vmove` cleanly are drivers that will work in VNET jails.

Together, these mechanisms shape FreeBSD as a platform for both hypervisor workloads and container workloads. A single kernel, a single driver set, several different ways for that driver to appear to its users. Understand-

ing the mechanisms is what lets you write drivers that work correctly across all of them, without special-casing any of them.

The discipline the chapter has advocated, sometimes explicitly, sometimes by example, is a continuation of Chapter 29's theme. Write clean abstractions. Use the framework APIs. Do not reach around them. Do not invent your own DMA or privilege or visibility mechanisms; FreeBSD already has well-tested ones. If you do all of this, your driver will work in environments you did not have in mind when you wrote it, and that is the best definition of portable a driver can have.

If you worked through the labs, you now have hands-on experience with starting a bhyve guest, writing a small module that uses `vm_guest`, putting a character device behind a devfs ruleset in a jail, and moving a network interface through VNET. Those four skills are all you need to start doing real work on driver code that runs in virtualised and containerised environments. Everything else in the chapter is supporting context.

If you tackled one or more of the challenges, you have stretched further: into simulation backends, into VNET-aware interfaces, into the bhyve emulator itself, or into test automation. Any of these is a genuine piece of FreeBSD driver craft, and the work compounds. Every hour you spend on these topics comes back in every driver you write afterwards.

A final note on attitude. Virtualisation and containerization can feel overwhelming because they introduce so many new pieces at once: hypervisors, paravirtual devices, jails, VNET, rulesets, privileges. But every piece has a clear purpose and a small, well-designed API. The sense of overwhelm goes away once you have seen each one in isolation, and this chapter's pacing was chosen to let you do that. If you are still feeling overwhelmed, go back to Section 3 (VirtIO) or Section 6 (jails and devfs) and re-read with a specific small question in mind. The answers are there.

30.18 Looking Ahead: Security and Privilege

Chapter 31, "Security and Privilege in Device Drivers," builds directly on the foundations laid in this chapter. Jails and virtual machines are one kind of security boundary; drivers have many others. A driver that exposes an `ioctl` is a driver that has created a new interface into the kernel, and that interface must be checked, validated, and restricted.

Chapter 31 will cover the privilege framework in depth (you saw `priv_check` here; Chapter 31 goes through the whole list), the `ucred(9)` structure and how credentials flow through the kernel, the `capsicum(4)` capability framework for finer-grained restrictions, and the MAC (Mandatory Access Control) framework for policy-based security. It will also revisit jails from a security-first perspective, complementing the container-first perspective of this chapter.

The thread that runs through Chapters 29, 30, and 31 is environment. Chapter 29 was about architectural environment: running on the same kernel with different buses or bit-widths. Chapter 30 was about operational environment: running inside a VM, a container, or on a host. Chapter 31 is about policy environment: running under the constraints a security-conscious administrator chooses to apply. A driver that handles all three kinds of environment well is a driver that can be deployed anywhere FreeBSD runs.

With Chapter 31, Part 7 approaches its midpoint. The remaining chapters of the part turn to debugging (Chapter 32), testing (Chapter 33), performance tuning (Chapter 34), and specialised driver topics in later chapters. Each builds on what you have learned so far. Take the time to let this chapter settle; the concepts will keep paying off as you continue.

30.19 Case Study: Designing a Pedagogical VirtIO Driver

This closing case study pulls together the threads of the chapter into a single design walkthrough. It does not present a complete implementation. Instead, it walks through the decisions you would make if you sat down today to design a pedagogical VirtIO driver called `vtedu`. The driver does nothing useful in production, but it exercises enough of the VirtIO surface to be a valuable teaching tool for future readers.

30.19.1 What `vtedu` Is For

Imagine that `vtedu` is meant to serve as the example driver for a future FreeBSD workshop on paravirtualised devices. Its job is to expose a single virtqueue, accept write requests from user space, pass them through the virtqueue to a backend in `bhyve(8)` (which does something simple like echoing bytes back), and deliver the echoed bytes back to user space. It must be small enough to read in an afternoon and complete enough to demonstrate the full VirtIO lifecycle.

The design choices below explain each step. A reader who finishes this section should be able to reason about any similar VirtIO driver in the same terms.

30.19.2 Choosing the Device Identifier

VirtIO defines a set of well-known device IDs in `/usr/src/sys/dev/virtio/virtio_ids.h`. For a pedagogical driver, a reserved or experimental ID is appropriate. The VirtIO specification reserves some ranges for “vendor-specific” devices, and a workshop driver would pick one of those.

For `vtedu`, we pick a hypothetical `VIRTIO_ID_EDU = 0xffff0` (chosen so it does not collide with real device IDs). The corresponding backend in `bhyve(8)` would register the same ID. A real project would coordinate with the `bhyve(8)` maintainers on the ID assignment.

30.19.3 Defining the Features

A teaching driver should negotiate a meaningful feature bit so that the reader sees feature negotiation in action. `vtedu` defines one feature:

```
#define VTEDU_F_UPPERCASE (1ULL << 0)
```

When negotiated, the backend returns the input bytes uppercased. When not negotiated, it returns them unchanged. The driver advertises the feature, the backend may or may not support it, and the negotiation produces whichever outcome both sides can support.

This kind of trivial feature is a teaching device. In a real driver, features correspond to real capabilities; in `vtedu`, the feature exists just to show how negotiation works.

30.19.4 Softc Layout

The softc is the per-instance state:


```

struct vtedu_softc {
    device_t      dev;
    struct virtqueue *vq;
    uint64_t      features;
    struct mtx      lock;
    struct cdev     *cdev;
    struct sglist    *sg;
    char          buf[VTEDU_BUF_SIZE];
    size_t        buf_len;
};

```

The device handle, the virtqueue pointer, the negotiated feature mask, a mutex protecting the driver's serialised access, a cdev for the user-space interface, a pre-allocated scatter-gather list, a buffer for the data, and its current length.

30.19.5 Transport Registration

The module uses `VIRTIO_DRIVER_MODULE` as always:

```

static device_method_t vtedu_methods[] = {
    DEVMETHOD(device_probe,      vtedu_probe),
    DEVMETHOD(device_attach,     vtedu_attach),
    DEVMETHOD(device_detach,     vtedu_detach),
    DEVMETHOD_END
};

static driver_t vtedu_driver = {
    "vtedu",
    vtedu_methods,
    sizeof(struct vtedu_softc)
};

VIRTIO_DRIVER_MODULE(vtedu, vtedu_driver, vtedu_modevent, NULL);
MODULE_VERSION(vtedu, 1);
MODULE_DEPEND(vtedu, virtio, 1, 1, 1);

```

The PNP info for `vtedu` advertises `VIRTIO_ID_EDU`, so the framework binds the driver to any VirtIO device of that type, under either PCI or MMIO transport.

30.19.6 Probe and Attach

The probe is a one-liner using `VIRTIO_SIMPLE_PROBE`. The attach sets up the device in the standard order:

```

static int
vtedu_attach(device_t dev)
{
    struct vtedu_softc *sc = device_get_softc(dev);
    int error;

    sc->dev = dev;
    mtx_init(&sc->lock, device_get_nameunit(dev), NULL, MTX_DEF);

    virtio_set_feature_desc(dev, vtedu_feature_descs);

    error = vtedu_negotiate_features(sc);
    if (error != 0)
        goto fail;

    error = vtedu_alloc_virtqueue(sc);
    if (error != 0)
        goto fail;

    error = virtio_setup_intr(dev, INTR_TYPE_MISC);
    if (error != 0)
        goto fail;

    sc->sg = sglist_alloc(2, M_WAITOK);
    sc->cdev = make_dev(&vtedu_cdevsw, device_get_unit(dev),
        UID_ROOT, GID_WHEEL, 0600, "vtedu%d", device_get_unit(dev));
    if (sc->cdev == NULL) {
        error = ENXIO;
        goto fail;
    }
    sc->cdev->si_drv1 = sc;

    device_printf(dev, "attached (features=0x%lx)\n",
        (unsigned long)sc->features);
    return (0);

fail:
    vtedu_detach(dev);
    return (error);
}

```

This follows the standard rhythm: negotiate, allocate queue, setup interrupts, set up user-space interface. It is structurally identical to `virtio_random`'s `attach`, with a `cdev` creation added because `vtedu` exposes a character device.

30.19.7 Feature Negotiation

Negotiation is straightforward:

```
static int
vtedu_negotiate_features(struct vtedu_softc *sc)
{
    uint64_t features = VIRTIO_F_VERSION_1 | VTEDU_F_UPPERCASE;

    sc->features = virtio_negotiate_features(sc->dev, features);
    return (virtio_finalize_features(sc->dev));
}
```

The driver advertises the two features, gets back an intersection, and finalises. The intersection tells the driver whether VTEDU_F_UPPERCASE is in play. Subsequent code uses `virtio_with_feature(sc->dev, VTEDU_F_UPPERCASE)` to check.

30.19.8 Queue Allocation

A single virtqueue is allocated with an interrupt callback:

```
static int
vtedu_alloc_virtqueue(struct vtedu_softc *sc)
{
    struct vq_alloc_info vq_info;

    VQ_ALLOC_INFO_INIT(&vq_info, 0, vtedu_vq_intr, sc, &sc->vq,
        "%s request", device_get_nameunit(sc->dev));

    return (virtio_alloc_virtqueues(sc->dev, 0, 1, &vq_info));
}
```

The maximum indirect-descriptor size is zero (no indirects), the interrupt callback is `vtedu_vq_intr`, and the virtqueue pointer is stored in `sc->vq`. A single queue is enough for a request-response pattern where the same queue is used for both directions.

30.19.9 The Character Device Interface

User space opens `/dev/vtedu0` and writes bytes to it. The driver accepts them, issues a VirtIO request, waits for the response, and exposes the response back to user space through `read`.

```
static int
vtedu_write(struct cdev *dev, struct uio *uio, int flags __unused)
```

```
{
    struct vtedu_softc *sc = dev->si_drv1;
    size_t n;
    int error;

    n = uio->uio_resid;
    if (n == 0 || n > VTEDU_BUF_SIZE)
        return (EINVAL);

    mtx_lock(&sc->lock);
    error = uiomove(sc->buf, n, uio);
    if (error == 0) {
        sc->buf_len = n;
        error = vtedu_submit(sc);
    }
    mtx_unlock(&sc->lock);
    return (error);
}
```

The write copies bytes into the softc buffer, sets `buf_len`, and calls `vtedu_submit`, which does the VirtIO enqueue and notify.

```
static int
vtedu_submit(struct vtedu_softc *sc)
{
    int error;

    sglist_reset(sc->sg);
    error = sglist_append(sc->sg, sc->buf, sc->buf_len);
    if (error != 0)
        return (error);

    error = virtqueue_enqueue(sc->vq, sc, sc->sg, 1, 1);
    if (error != 0)
        return (error);

    virtqueue_notify(sc->vq);
    return (0);
}
```

One readable segment (the driver's write into the buffer) and one writable segment (the device's write of the result). The cookie is `sc` itself; the interrupt handler will receive it back when the completion arrives.

30.19.10 The Interrupt Handler

Completions are processed in a taskqueue-deferred or immediate fashion. For pedagogical simplicity, vtedu processes them immediately in the interrupt callback:

```
static void
vtedu_vq_intr(void *arg)
{
    struct vtedu_softc *sc = arg;
    void *cookie;
    uint32_t len;

    mtx_lock(&sc->lock);
    while ((cookie = virtqueue_dequeue(sc->vq, &len)) != NULL) {
        /* cookie == sc; len is the length the device wrote. */
        sc->buf_len = len;
        wakeup(sc);
    }
    mtx_unlock(&sc->lock);
}
```

The handler drains all completions, updates the softc, and wakes any sleeper waiting on the buffer. For a real driver, this would be richer; for vtedu, it is all we need.

30.19.11 The Read Path

User space then reads the result:

```
static int
vtedu_read(struct cdev *dev, struct uio *uio, int flags __unused)
{
    struct vtedu_softc *sc = dev->si_drv1;
    int error;

    mtx_lock(&sc->lock);
    while (sc->buf_len == 0) {
        error = mtx_sleep(sc, &sc->lock, PCATCH, "vteduR", 0);
        if (error != 0) {
            mtx_unlock(&sc->lock);
            return (error);
        }
    }
    error = uiomove(sc->buf, sc->buf_len, uio);
    sc->buf_len = 0;
}
```

```
    mtx_unlock(&sc->lock);  
    return (error);  
}
```

If no result is available, the read sleeps on `sc`, which the interrupt handler will wake. This is the standard “block until ready” pattern for character devices with slow underlying I/O.

30.19.12 Detach

The detach reverses the attach, cleanly:

```
static int  
vtedu_detach(device_t dev)  
{  
    struct vtedu_softc *sc = device_get_softc(dev);  
  
    if (sc->cdev != NULL)  
        destroy_dev(sc->cdev);  
    if (sc->sg != NULL)  
        sglist_free(sc->sg);  
    virtio_stop(dev);  
    if (mtx_initialized(&sc->lock))  
        mtx_destroy(&sc->lock);  
    return (0);  
}
```

`virtio_stop` resets the device status so it no longer generates interrupts. The `cdev` is destroyed, the scatter-gather list freed, and the mutex destroyed.

30.19.13 Putting vtedu Together

This walkthrough has touched on every major element of a VirtIO driver:

1. Device ID selection and PNP info.
2. Feature definition and negotiation.
3. Softc layout and locking.
4. Virtqueue allocation with interrupt callback.
5. User-space interface through `cdev`.
6. Request submission through `enqueue + notify`.
7. Completion handling through the interrupt callback.
8. Clean detach.

The whole driver, fully implemented, fits in around 300 lines of C. That is less than `virtio_random` once you add in the backend plumbing. Because the user-space interface is richer than `virtio_random`’s (which hides in the `random(4)` framework), the code is slightly larger in total, but the VirtIO-specific part is no bigger.

30.19.14 Using vtedu for Teaching

A vtedu driver would be used in a workshop something like this:

1. The instructor starts by demonstrating the driver loading, attaching, and processing a write-then-read round trip.
2. Students follow along, typing the key sections from a cheat sheet.
3. The instructor introduces feature negotiation by showing what happens when `VTEDU_F_UPPERCASE` is negotiated (outputs arrive uppercased) versus when it is not.
4. Students modify the driver to add a second feature: “reverse the bytes”. They learn how features compose.
5. Finally, the instructor shows how to run the driver in a jail (it just works, as long as `vtedu0` is in the `devfs` ruleset), illustrating the containerisation side of the chapter.

This is a pedagogical design, not a production one. Its purpose is to show how the pieces fit. A reader who has followed this case study should be able to write a similar driver of their own, starting from scratch or starting from `virtio_random.c` as a template.

30.19.15 What vtedu Does Not Do

For honesty: vtedu as sketched here omits several things a production driver would include. It does not support multiple in-flight requests (the lock serialises everything). It does not handle queue-full situations gracefully (it assumes one request at a time). It does not support module-wide module-event cleanup (just per-device). It does not demonstrate indirect descriptors (because the feature is irrelevant for 256-byte messages).

Each of these is an exercise the reader could undertake after understanding the base design. The chapter’s challenge exercises hint at some of them; a dedicated workshop would develop them further.

30.19.16 A Teaching Driver Versus a Real Driver

Before leaving vtedu, a note about the difference between a teaching driver and a real one. A teaching driver is designed for readability. Its code makes every concept visible, often at the cost of clever optimisations. A real driver is designed for reliability and performance. It compresses the common-case paths, adds error handling for every edge case, and optimises for the actual workload.

The temptation when moving from learning to production is to start with the teaching driver and add features. That usually produces worse code than starting with an architectural sketch and filling it in. A teaching driver is a reference; a real driver is a system. The two are not on the same continuum.

As a driver author, your job is to understand the teaching driver well enough that the real-driver design decisions become clear. This chapter’s treatment of VirtIO is meant to get you to that understanding. The next step is yours.

30.20 Appendix: Quick Reference

The reference tables below condense the chapter’s key facts into a form you can return to during driver work. They are not a substitute for the explanatory text; think of them as the one-page cheat sheet for the day you are actually writing code and need a specific detail.

30.20.1 VirtIO Core API Functions

Function	Purpose
<code>virtio_negotiate_features(dev, mask)</code>	Advertise and negotiate feature bits.
<code>virtio_finalize_features(dev)</code>	Seal the feature negotiation.
<code>virtio_with_feature(dev, feature)</code>	Test whether a feature is negotiated.
<code>virtio_alloc_virtqueues(dev, flags, nvqs, info)</code>	Allocate a set of virtqueues.
<code>virtio_setup_intr(dev, type)</code>	Install the negotiated interrupt handlers.
<code>virtio_read_device_config(dev, offset, dst, size)</code>	Read device-specific configuration.
<code>virtio_write_device_config(dev, offset, src, size)</code>	Write device-specific configuration.

30.20.2 virtqueue(9) Functions

Function	Purpose
<code>virtqueue_enqueue(vq, cookie, sg, readable, writable)</code>	Push a scatter-gather chain onto the available ring.
<code>virtqueue_dequeue(vq, &len)</code>	Pop one completed chain from the used ring.
<code>virtqueue_notify(vq)</code>	Tell the host new work is available.
<code>virtqueue_poll(vq, &len)</code>	Wait for a completion and return it.
<code>virtqueue_empty(vq)</code>	Check whether the queue has any pending work.
<code>virtqueue_full(vq)</code>	Check whether the queue has space for another enqueue.

30.20.3 vm_guest Values

Constant	String via <code>kern.vm_guest</code>	Meaning
<code>VM_GUEST_NO</code>	<code>none</code>	Bare metal
<code>VM_GUEST_VM</code>	<code>generic</code>	Unknown hypervisor
<code>VM_GUEST_XEN</code>	<code>xen</code>	Xen
<code>VM_GUEST_HV</code>	<code>hv</code>	Microsoft Hyper-V
<code>VM_GUEST_VMWARE</code>	<code>vmware</code>	VMware ESXi / Workstation

Constant	String via kern.vm_guest	Meaning
VM_GUEST_KVM	kvm	Linux KVM
VM_GUEST_BHYVE	bhyve	FreeBSD bhyve
VM_GUEST_VBOX	vbox	Oracle VirtualBox
VM_GUEST_PARALLELS	parallels	Parallels
VM_GUEST_NVMM	nvmm	NetBSD NVMM

30.20.4 Default devfs Rulesets

Number	Name	Purpose
1	devfsrules_hide_all	Start with everything hidden.
2	devfsrules_unhide_basic	Essential devices (null, zero, random, etc.).
3	devfsrules_unhide_login	Login-related devices (pts, ttyv*).
4	devfsrules_jail	Standard non-VNET jail ruleset.
5	devfsrules_jail_vnet	Standard VNET jail ruleset.

30.20.5 Common Privilege Constants

Constant	Typical jail policy
PRIV_DRIVER	Denied (driver-private ioctls).
PRIV_IO	Denied (raw I/O port access).
PRIV_KMEM_WRITE	Denied (kernel memory writes).
PRIV_KLD_LOAD	Denied (module loading).
PRIV_NET_SETLLADDR	Denied (MAC address changes).
PRIV_NETINET_RAW	Denied unless allow.raw_sockets.
PRIV_NET_BPF	Allowed via allow.raw_sockets for BPF-needing tools.

30.20.6 VNET Macros

Macro	Purpose
VNET_DEFINE(type, name)	Declare a per-VNET variable.

Macro	Purpose
VNET(name)	Access the per-VNET variable for the current VNET.
V_name	Conventional shorthand for VNET(name).
CURVNET_SET(vnet)	Establish a VNET context on the current thread.
CURVNET_RESTORE()	Tear down the context.
VNET_SYSINIT(name, ...)	Register a per-VNET init function.
VNET_SYSUNINIT(name, ...)	Register a per-VNET uninit function.

30.20.7 bhyve and Passthrough Tools

Tool	Purpose
bhyve(8)	Run a virtual machine.
bhyvectl(8)	Query and control running VMs.
vm(8)	High-level management (via vm-bhyve port).
pciconf(8)	Show PCI devices and their driver bindings.
devctl(8)	Explicit control of driver attach/detach.
pptdevs in /boot/loader.conf	Bind devices to the passthrough placeholder.

30.20.8 Manual Pages Worth Bookmarking

- virtio(4) - VirtIO framework overview.
- vtnet(4), virtio_blk(4) - Specific VirtIO drivers.
- bhyve(8), bhyvectl(8), vmm(4) - Hypervisor user and kernel interfaces.
- pci_passthru(4) - PCI passthrough mechanism.
- jail(8), jail.conf(5), jls(8), jexec(8) - Jail management.
- devfs(5), devfs(8), devfs.rules(5) - devfs and rulesets.
- if_epair(4), vlan(4), if_tap(4) - Pseudo-interfaces useful for jails.
- rctl(8), racct(9) - Resource control.
- priv(9) - Privilege framework.

30.20.9 Top Five Things a Driver Author Should Do

If you remember nothing else from this chapter, remember these five habits:

1. Use bus_dma(9) for every DMA buffer. Never pass physical addresses directly to the hardware. This is the single most important habit for passthrough and IOMMU-protected environments.
2. Use priv_check(9) for privileged operations. Do not hardcode `cred->cr_uid == 0` checks. The framework extends your code to jails for free.

3. Keep device node names predictable. Administrators who write devfs rulesets need to know what to unhide. Document the name in your driver's manual page.
4. Handle detach cleanly. Release every resource, cancel every callout, drain every taskqueue, and never assume the softc will be re-used. Passthrough and VNET both exercise detach heavily.
5. Establish VNET context around callout and taskqueue code that touches per-VNET state. `CURVNET_SET` / `CURVNET_RESTORE` is the boilerplate, and missing it is the most common VNET-related bug.

These five habits between them cover nearly all of the “works under virtualisation and containerization” work a driver author needs to do. Everything else is refinement.

30.21 Appendix: Common Code Patterns

A small catalogue of patterns that appear repeatedly in FreeBSD drivers that run in virtualised or containerised environments. Each one is a snippet you can adapt to your own code.

30.21.1 Pattern: Environment-aware default

```
static int
mydev_default_interrupt_moderation(void)
{
    switch (vm_guest) {
    case VM_GUEST_NO:
        return (1); /* tight moderation on bare metal */
    default:
        return (4); /* loose moderation under a hypervisor */
    }
}
```

Use this pattern to seed a default that the user can override via `sysctl`. Do not branch on the specific hypervisor brand unless there is a real reason.

30.21.2 Pattern: Privilege-gated ioctl

```
case MYDEV_IOC_DANGEROUS:
    error = priv_check(td, PRIV_DRIVER);
    if (error != 0)
        return (error);
    /* perform the dangerous operation */
    return (0);
```

The default position for driver-specific privileges is to require `PRIV_DRIVER`. Consult `priv(9)` if a more specific privilege fits better.

30.21.3 Pattern: VNET-aware callout

```
static void
mydev_callout(void *arg)
{
    struct mydev_softc *sc = arg;

    CURVNET_SET(sc->ifp->if_vnet);
    /* read or write V_ variables, or call network-stack functions */
    CURVNET_RESTORE();

    callout_reset(&sc->co, hz, mydev_callout, sc);
}
```

Any callout, taskqueue function, or kernel thread that might access VNET state must establish the context. Missing this is the most common source of VNET-related bugs.

30.21.4 Pattern: VirtIO attach skeleton

```
static int
mydev_attach(device_t dev)
{
    struct mydev_softc *sc = device_get_softc(dev);
    int error;

    sc->dev = dev;
    virtio_set_feature_desc(dev, mydev_feature_descs);

    error = mydev_negotiate_features(sc);
    if (error != 0)
        goto fail;

    error = mydev_alloc_virtqueues(sc);
    if (error != 0)
        goto fail;

    error = virtio_setup_intr(dev, INTR_TYPE_MISC);
    if (error != 0)
        goto fail;

    /* post initial buffers, register with subsystem, etc. */
    return (0);
}
```

```
fail:
    mydev_detach(dev);
    return (error);
}
```

The “negotiate, allocate, setup interrupts, start” rhythm is standard for every VirtIO driver. Every VirtIO attach in `/usr/src/sys/dev/virtio/` is a variation on this skeleton.

30.21.5 Pattern: Clean detach

```
static int
mydev_detach(device_t dev)
{
    struct mydev_softc *sc = device_get_softc(dev);

    /* Stop accepting new work. */
    sc->detaching = true;

    /* Drain async mechanisms. */
    if (sc->co_initialised)
        callout_drain(&sc->co);
    if (sc->tq != NULL)
        taskqueue_drain_all(sc->tq);

    /* Release hardware resources. */
    if (sc->irq_cookie != NULL)
        bus_teardown_intr(dev, sc->irq_res, sc->irq_cookie);
    if (sc->irq_res != NULL)
        bus_release_resource(dev, SYS_RES_IRQ, 0, sc->irq_res);
    if (sc->mem_res != NULL)
        bus_release_resource(dev, SYS_RES_MEMORY, 0, sc->mem_res);

    /* Destroy devfs nodes. */
    if (sc->cdev != NULL)
        destroy_dev(sc->cdev);

    return (0);
}
```

Detach must be symmetric with attach: every resource attach allocated must be released here, in reverse order. A clean detach is what makes the driver safe for passthrough and for unload.

30.21.6 Pattern: `make_dev` with sensible defaults

```
sc->cdev = make_dev(&mydev_cdevsw, 0, UID_ROOT, GID_WHEEL, 0600, "mydev%d",
    device_get_unit(dev));
if (sc->cdev == NULL)
    return (ENXIO);
sc->cdev->si_drv1 = sc;
```

Use 0600 mode for nodes that should only be root-accessible, 0644 for read-any / write-root, 0666 for rare read-and-write-any cases. The `si_drv1` field is the conventional back-pointer from a `struct cdev` to the driver's `softc`.

30.22 Appendix: Glossary

A short glossary of the terms this chapter relies on. Refer back when a term in a later section has slipped from memory.

- **Bare metal:** A system running directly on physical hardware with no hypervisor in between.
- **bhyve:** FreeBSD's native type-2 hypervisor. Runs as a user-space program backed by the kernel's `vmm(4)` module.
- **Container:** In FreeBSD, a jail or a user-space framework built on top of jails.
- **Credential (`struct ucred`):** The per-process security context that carries uid, gid, jail pointer, and privilege-related state.
- **devfs:** The special filesystem at `/dev` where devices are exposed.
- **devfs ruleset:** A named set of rules that controls which devfs nodes are visible in a given devfs mount.
- **Emulated device:** A hypervisor-provided device that imitates a real-hardware interface.
- **Guest:** The operating system running inside a virtual machine.
- **Host:** The physical machine (or container-enclosing system) that hosts guests or jails.
- **Hypervisor:** Software that creates and manages virtual machines.
- **IOMMU:** A unit between a device and host memory that remaps DMA addresses. Enables safe passthrough.
- **Jail:** FreeBSD's lightweight containerisation mechanism.
- **Paravirtual device:** A device with an interface designed to be easy for a hypervisor to emulate and for a guest to drive. VirtIO is the canonical example.
- **Passthrough:** Giving a guest direct access to a physical device.
- **Prison:** The kernel's internal name for a jail. `struct prison` is the data structure.
- **rctl / racct:** The resource-control and resource-accounting frameworks that enforce per-jail or per-process limits.
- **Ruleset:** See *devfs ruleset*.
- **Transport (VirtIO):** The bus-level mechanism that carries VirtIO messages. Examples: PCI, MMIO.
- **Virtqueue:** The shared-ring data structure at the heart of VirtIO.
- **VirtIO:** The paravirtualisation standard used by most modern hypervisors.
- **VM:** Virtual machine.

- **VNET**: FreeBSD's virtual network stack framework, providing per-jail independent stacks.
- **vmm**: FreeBSD's kernel hypervisor core module, used by bhyve(8).

30.23 Appendix: Observing VirtIO with DTrace

DTrace is one of FreeBSD's most capable diagnostic tools, and it is especially useful for understanding how a VirtIO driver behaves at runtime. This appendix collects several concrete DTrace recipes for VirtIO observability. None of them require modifying the driver; they work against the unmodified kernel because the `fbt` (function-boundary-tracing) provider instruments every kernel function.

30.23.1 A First Probe: Counting virtqueue Enqueues

The simplest useful probe counts how often `virtqueue_enqueue` is called per second.

```
sudo dtrace -n 'fbt::virtqueue_enqueue:entry /pid == 0/ { @[probefunc] = count(); }
↵ tick-1sec { printa(@); trunc(@); }'
```

Running this on a busy VM shows numbers like:

```
virtqueue_enqueue 12340
virtqueue_enqueue 15220
virtqueue_enqueue 11890
```

Each number is enqueues per second across all virtqueues. A number in the thousands is normal for an active VM; a number in the millions suggests a pathological driver. The per-VM baseline depends on workload, but getting familiar with the baseline for your environment lets you spot anomalies later.

30.23.2 Separating Queues by Virtqueue

To see which virtqueue is being used, extend the probe to key on the virtqueue name.

```
sudo dtrace -n '
fbt::virtqueue_enqueue:entry
{
    this->vq = (struct virtqueue *)arg0;
    @[stringof(this->vq->vq_name)] = count();
}
tick-1sec
{
    printa(@);
    trunc(@);
}
'
```

Output now looks like:

vtnet0-rx	482
vtnet0-tx	430
virtio_blk0	8220

This instantly reveals which device is doing work. A disk-heavy workload shows `virtio_blk0` dominating; a network-heavy workload shows `vtnet0-tx` and `vtnet0-rx`. This kind of first-level breakdown is usually enough to localise a performance problem.

Note that the example dereferences an internal struct (`struct virtqueue`). The struct's layout is implementation-detail and can change between FreeBSD releases. Check `/usr/src/sys/dev/virtio/virtqueue.c` if the struct layout has changed and the probe fails.

30.23.3 Measuring Time Spent in Enqueue

Time-in-function is a standard DTrace recipe.

```
sudo dtrace -n '
fbt::virtqueue_enqueue:entry
{
    self->ts = vtimestamp;
}
fbt::virtqueue_enqueue:return
/ self->ts /
{
    @["virtqueue_enqueue"] = quantize(vtimestamp - self->ts);
    self->ts = 0;
}
'
```

The output is a histogram of how long each `virtqueue_enqueue` call took, in nanoseconds. For healthy VirtIO, most calls complete in the low microsecond range. A significant tail suggests lock contention (the function's mutex is held for too long), memory pressure, or expensive scatter-gather computation.

30.23.4 Watching for VM Exits

VM exits are the fundamental cost of virtualisation. DTrace's `vmm` probes (if available) let you count them.

```
sudo dtrace -n 'fbt:vmm::*:entry { @[probfunc] = count(); } tick-1sec { printa(@);
↵      trunc(@); }'
```

On a busy host, this shows dozens of `vmm` functions. The ones to watch are `vm_exit_*`, which handle different exit types (I/O, interrupt, hypercall). Seeing `vm_exit_inout` in the top results suggests the guest is doing a lot of I/O through emulated devices and would benefit from VirtIO.

30.23.5 Tracing a Specific Driver

To focus on a specific driver's functions, narrow the probe to its module name.

```
sudo dtrace -n 'fbt:virtio_blk::*:entry { @[probefunc] = count(); } tick-1sec {  
    ↪  printa(@); trunc(@); }'
```

This counts all `virtio_blk` function entries. On a quiescent VM, the output is empty; on an active VM, you see every function the driver calls, with counts. This is useful for getting a feel for a driver's internal structure.

30.23.6 Tracing a Custom Driver

For the `vtedu` driver from the case study, the same technique works as long as the module is loaded and named `virtio_edu`.

```
sudo dtrace -n 'fbt:virtio_edu::*:entry { @[probefunc] = count(); } tick-1sec {  
    ↪  printa(@); trunc(@); }'
```

If no backend is attached, the counts will be zero. If a backend is attached and the driver is exercising the virtqueue, you see `vtedu_write`, `vtedu_submit_locked`, `vtedu_vq_intr`, and `vtedu_read` in the counts, in rough proportion to usage.

30.23.7 Observing the Guest-Host Boundary

A more ambitious use of DTrace is to correlate guest-side events with host-side events. This is possible when both sides are FreeBSD and DTrace is available on both. Run DTrace on the host to count `vmm` exits and on the guest to count driver calls; the numbers should correlate one-for-one on an unloaded system, diverging as the host batches exits or as posted interrupts kick in.

This is an advanced technique and mostly interesting for performance analysis. For day-to-day debugging, single-side DTrace is usually enough.

30.23.8 Saving and Reusing Probes

The `dtrace(1)` command-line form is fine for quick investigations. For repeated use, save probes to a file and invoke with `dtrace -s`.

```
% cat > virtio_probes.d <<'EOF'  
#pragma D option quiet  
  
fbt::virtqueue_enqueue:entry  
{  
    this->vq = (struct virtqueue *)arg0;  
    @[stringof(this->vq->vq_name)] = count();  
}
```

```
}

tick-1sec
{
    printa(@);
    trunc(@);
}

EOF

% sudo dtrace -s virtio_probes.d
```

A curated set of probes in `.d` files is a good investment for anyone who spends significant time debugging VirtIO.

30.23.9 When DTrace Cannot Help

DTrace can observe the kernel and, with `pid` provider, most user-space programs. It cannot directly observe guest kernels from the host, because the guest is a process whose internal structure `dtrace` does not know. You can trace the `bhyve(8)` process itself (as a user-space program) using the `pid` provider, which shows what `bhyve` is doing but not what its guest is doing.

For guest-side tracing, DTrace runs inside the guest. If the guest is FreeBSD 14.3, the full DTrace toolkit is available. If the guest is Linux, use `bpftrace` or `perf` instead; they are different tools with similar capabilities.

30.23.10 A Closing Word

DTrace is one of FreeBSD's competitive advantages. Every driver author should be comfortable with it; the investment pays off repeatedly across years of debugging. This appendix gives you a starting point; the FreeBSD Handbook's DTrace chapter and the original *DTrace: Dynamic Tracing in Oracle Solaris* book (freely available online) are the next steps if you want to go deeper.

30.24 Appendix: A Complete bhyve Configuration Walkthrough

Readers who want to run the labs need a working `bhyve(8)` setup. This appendix walks through a complete configuration, from a bare FreeBSD 14.3 host to a guest with VirtIO devices, in enough detail for a beginner to reproduce it. If you have already built `bhyve` environments, skim this; the goal is to provide a reference for readers who have not.

30.24.1 The Host Side

Start with a FreeBSD 14.3 host. Confirm the virtualisation extensions are enabled.

```
% sysctl hw.vmm
hw.vmm.topology.cores_per_package: 0
hw.vmm.topology.threads_per_core: 0
hw.vmm.topology.sockets: 0
hw.vmm.topology.cpus: 0
...
```

If `hw.vmm` is absent entirely, the `vmm(4)` module is not loaded. Load it with `kldload vmm`. Add `vmm_load="YES"` to `/boot/loader.conf` to load it at every boot.

If `vmm` is loaded but the VT-x/AMD-V features are absent, enable them in the host firmware. The setting is usually called “VT-x”, “VMX”, “AMD-V”, or “SVM” in the BIOS/UEFI menu. After enabling, reboot.

30.24.2 Install vm-bhyve

`vm-bhyve` is a wrapper that makes `bhyve(8)` easier to use. Install it from ports or packages.

```
% sudo pkg install vm-bhyve
```

Create the VM directory. Using ZFS is convenient because it supports snapshotting.

```
% sudo zfs create -o mountpoint=/vm zroot/vm
```

Or plain UFS:

```
% sudo mkdir /vm
```

Enable `vm-bhyve` and set its directory in `/etc/rc.conf`.

```
vm_enable="YES"
vm_dir="zfs:zroot/vm"
```

For UFS, use `vm_dir="/vm"` instead. Initialise the directory.

```
% sudo vm init
% sudo cp /usr/local/share/examples/vm-bhyve/config_samples/default.conf
↪ /vm/.templates/
```

Edit `/vm/.templates/default.conf` if you want different defaults (memory size, CPU count).

30.24.3 Create and Install a Guest

Download a FreeBSD 14.3 installation image. The file `FreeBSD-14.3-RELEASE-amd64-disc1.iso` is the standard installer; for faster setup, use `FreeBSD-14.3-RELEASE-amd64.qcow2` if you prefer a prebuilt image.

```
% sudo vm iso https://download.freebsd.org/releases/amd64/amd64/ISO-  
↪ IMAGES/14.3/FreeBSD-14.3-RELEASE-amd64-disc1.iso
```

Create the guest.

```
% sudo vm create -t default -s 20G guest0
```

Start the installer.

```
% sudo vm install guest0 FreeBSD-14.3-RELEASE-amd64-disc1.iso
```

Connect to the console.

```
% sudo vm console guest0
```

The FreeBSD installer runs; follow it through to completion. At the end, reboot the guest.

30.24.4 Configure Networking

`vm-bhyve` supports two network styles: bridged and NAT. For simplicity, bridged is fine.

Create a bridge.

```
% sudo vm switch create public  
% sudo vm switch add public em0
```

Replace `em0` with your host's physical interface. Assign guests to the switch in their configuration.

```
% sudo vm configure guest0
```

In the editor that opens, ensure `network0_switch="public"` is set.

30.24.5 Start the Guest

```
% sudo vm start guest0  
% sudo vm console guest0
```

Log in, run `pciconf -lvBb`, and confirm the VirtIO devices are present.

```
# pciconf -lvBb
hostb0@pci0:0:0:0:      class=0x060000 rev=0x00 ...
virtio_pci0@pci0:0:2:0: class=0x010000 rev=0x00 vendor=0x1af4 device=0x1001 ...
    vendor      = 'Red Hat, Inc.'
    device      = 'Virtio 1.0 block device'
virtio_pci1@pci0:0:3:0: class=0x020000 rev=0x00 vendor=0x1af4 device=0x1041 ...
    vendor      = 'Red Hat, Inc.'
    device      = 'Virtio 1.0 network device'
...
```

30.24.6 Troubleshooting the Host Side

If the guest does not start, run `vm start guest0` with the verbose flag (`vm -f start guest0` keeps it in the foreground), look at the error message, and consult the `bhyve(8)` manual. The most common issues are missing resources (disk image path wrong, switch not set up) and permissions (user not in the `vm` group, or directories not readable).

If networking does not work inside the guest, check the bridge on the host (`ifconfig bridge0`), check the tap device (`ifconfig tapN`), and check that the VM's `network0` setting matches the switch name. `vm-bhyve` generates a tap device per VM and wires it into the specified switch.

30.24.7 Using This Setup for the Labs

With a working host and guest, Lab 1 is essentially done (you have a VirtIO-using guest). Lab 2 and Lab 3 can be performed inside the guest. Lab 4 needs `if_epair` on the host; Lab 5 needs a spare PCI device and IOMMU-enabled firmware. Lab 6 (building `vtedu`) is done inside the guest. Lab 7 (measuring overhead) uses the guest as the test subject.

In short: if your `vm-bhyve` setup is solid, the rest of the chapter's hands-on work is accessible.

30.25 Appendix: VirtIO Feature Bits Reference

This appendix collects the VirtIO feature bits most likely to matter to a driver author, with a brief description of each. The authoritative source is the VirtIO specification; this is a digest for quick reference.

30.25.1 Device-Independent Features

These apply across all VirtIO device types.

- `VIRTIO_F_NOTIFY_ON_EMPTY` (bit 24): The device should notify the driver when the virtqueue becomes empty, in addition to normal completion notifications. Useful for drivers that want to know when all outstanding requests have been processed.
- `VIRTIO_F_ANY_LAYOUT` (bit 27, deprecated in v1): Headers and data can be in any scatter-gather layout. Always negotiated in v1 drivers; not relevant for modern code.

- **VIRTIO_F_RING_INDIRECT_DESC** (bit 28): Indirect descriptors supported. An entry in the descriptor table can point to another table, allowing longer scatter-gather lists without expanding the main ring. Strongly recommended for drivers that handle large requests.
- **VIRTIO_F_RING_EVENT_IDX** (bit 29): Event-index interrupt suppression. Lets the driver tell the device “do not interrupt me before descriptor N is available”, reducing interrupt rate. Strongly recommended for high-rate drivers.
- **VIRTIO_F_VERSION_1** (bit 32): Modern VirtIO (version 1.0 or later). Without this, the driver is in legacy mode, with different config-space layout and conventions. New drivers should require this.
- **VIRTIO_F_ACCESS_PLATFORM** (bit 33): The device uses a platform-specific DMA address translation (for example, an IOMMU). Required for passthrough-capable deployments.
- **VIRTIO_F_RING_PACKED** (bit 34): Packed virtqueue layout. A newer, more cache-friendly layout than the classic split layout. Not supported by all backends; negotiate but do not require.
- **VIRTIO_F_IN_ORDER** (bit 35): Descriptors are used in the same order they were made available. Allows optimisations in the driver (no need to track descriptor indexes); not supported by all backends.

30.25.2 Block Device Features (**virtio_blk**)

- **VIRTIO_BLK_F_SIZE_MAX** (bit 1): The device has a maximum single-request size.
- **VIRTIO_BLK_F_SEG_MAX** (bit 2): The device has a maximum segment count per request.
- **VIRTIO_BLK_F_GEOMETRY** (bit 4): The device reports its cylinder/head/sector geometry. Mostly legacy.
- **VIRTIO_BLK_F_RO** (bit 5): The device is read-only.
- **VIRTIO_BLK_F_BLK_SIZE** (bit 6): The device reports its block size.
- **VIRTIO_BLK_F_FLUSH** (bit 9): The device supports flush (fsync) commands.
- **VIRTIO_BLK_F_TOPOLOGY** (bit 10): The device reports topology information (alignment, etc.).
- **VIRTIO_BLK_F_CONFIG_WCE** (bit 11): The driver can query and set the write-cache-enabled flag.
- **VIRTIO_BLK_F_DISCARD** (bit 13): The device supports discard (trim) commands.
- **VIRTIO_BLK_F_WRITE_ZEROES** (bit 14): The device supports write-zeroes commands.

30.25.3 Network Device Features (**virtio_net**)

- **VIRTIO_NET_F_CSUM** (bit 0): The device can offload checksum computation.
- **VIRTIO_NET_F_GUEST_CSUM** (bit 1): The driver can checksum-offload incoming packets.
- **VIRTIO_NET_F_MAC** (bit 5): The device provides a MAC address.
- **VIRTIO_NET_F_GSO** (bit 6): GSO (generic segmentation offload) supported.
- **VIRTIO_NET_F_GUEST_TSO4** (bit 7): Receive-side TSO over IPv4.
- **VIRTIO_NET_F_GUEST_TSO6** (bit 8): Receive-side TSO over IPv6.
- **VIRTIO_NET_F_GUEST_ECN** (bit 9): ECN (explicit congestion notification) supported on receive.
- **VIRTIO_NET_F_GUEST_UFO** (bit 10): Receive-side UFO (UDP fragmentation offload).
- **VIRTIO_NET_F_HOST_TSO4** (bit 11): Transmit-side TSO over IPv4.
- **VIRTIO_NET_F_HOST_TSO6** (bit 12): Transmit-side TSO over IPv6.
- **VIRTIO_NET_F_HOST_ECN** (bit 13): ECN on transmit.
- **VIRTIO_NET_F_HOST_UFO** (bit 14): Transmit-side UFO.
- **VIRTIO_NET_F_MRG_RXBUF** (bit 15): Merged receive buffers.

- `VIRTIO_NET_F_STATUS` (bit 16): Configuration status is supported.
- `VIRTIO_NET_F_CTRL_VQ` (bit 17): Control virtqueue.
- `VIRTIO_NET_F_CTRL_RX` (bit 18): Control channel for receive mode filtering.
- `VIRTIO_NET_F_CTRL_VLAN` (bit 19): Control channel for VLAN filtering.
- `VIRTIO_NET_F_MQ` (bit 22): Multiqueue support.
- `VIRTIO_NET_F_CTRL_MAC_ADDR` (bit 23): Control channel for MAC-address setting.

30.25.4 How to Read a Feature Word

Features are a 64-bit word, with bits as described. To check whether a feature is negotiated:

```
if ((sc->features & VIRTIO_F_RING_EVENT_IDX) != 0) {  
    /* event indexes are available */  
}
```

To advertise a feature for negotiation, | it into the features mask before calling `virtio_negotiate_features`. The post-negotiation value of `sc->features` is the intersection of what the driver requested and what the backend offers.

30.25.5 Common Pitfalls

- Hard-coding feature requirements. A driver that requires `VIRTIO_NET_F_MRG_RXBUF` and falls over if it is absent will not work with simpler backends. Prefer to negotiate optimistically and adapt to what you get.
- Forgetting to check the post-negotiation features. A driver that behaves as if a feature is always present, without checking `sc->features`, will mis-program the device when the feature is absent.
- Ignoring version 1 requirements. Modern code should require `VIRTIO_F_VERSION_1`; legacy mode has too many quirks to be worth supporting in new drivers.

30.25.6 A Useful Habit

Log the negotiated feature word at attach time, with each interesting bit named. The `device_printf` below does this succinctly.

```
device_printf(dev, "features: ver1=%d evt_idx=%d indirect=%d mac=%d\n",  
    (sc->features & VIRTIO_F_VERSION_1) != 0,  
    (sc->features & VIRTIO_F_RING_EVENT_IDX) != 0,  
    (sc->features & VIRTIO_F_RING_INDIRECT_DESC) != 0,  
    (sc->features & VIRTIO_NET_F_MAC) != 0);
```

This single line in `dmesg` at attach time tells you, for any bug report, what feature set the driver is operating on. It is the VirtIO equivalent of logging a hardware revision; indispensable for support.

30.26 Appendix: Further Reading

For readers who want to go deeper, here is a short curated list.

30.26.1 FreeBSD source tree

- `/usr/src/sys/dev/virtio/random/virtio_random.c` - The smallest complete VirtIO driver. Read first.
- `/usr/src/sys/dev/virtio/network/if_vtnet.c` - A larger VirtIO driver.
- `/usr/src/sys/dev/virtio/block/virtio_blk.c` - A request-response VirtIO driver.
- `/usr/src/sys/dev/virtio/virtqueue.c` - The ring machinery.
- `/usr/src/sys/amd64/vmm/` - The bhyve kernel module.
- `/usr/src/usr.sbin/bhyve/` - The bhyve user-space emulator.
- `/usr/src/sys/kern/kern_jail.c` - Jail implementation.
- `/usr/src/sys/net/vnet.h`, `/usr/src/sys/net/vnet.c` - VNET framework.
- `/usr/src/sys/net/if_tuntap.c` - A VNET-aware cloning pseudo-driver.

30.26.2 Manual pages

- `virtio(4)`, `vtnet(4)`, `virtio_blk(4)`
- `bhyve(8)`, `bhyvectl(8)`, `vmm(4)`, `vmm_dev(4)`
- `pci_passthru(4)`
- `jail(8)`, `jail.conf(5)`, `jexec(8)`, `jls(8)`
- `devfs(5)`, `devfs.rules(5)`, `devfs(8)`
- `rctl(8)`, `racct(9)`
- `priv(9)`, `ucred(9)`
- `if_epair(4)`, `vlan(4)`, `tun(4)`, `tap(4)`

30.26.3 External standards

- The VirtIO 1.2 specification (OASIS) is the authoritative reference for the protocol. Available from the OASIS VirtIO Technical Committee site.
- The PCI-SIG specifications for PCI Express, MSI-X, and ACS are relevant for passthrough.

30.26.4 FreeBSD Handbook

- Chapter on jails, which complements this chapter with an administrative perspective.
- Chapter on virtualisation, which covers bhyve(8) management in more depth than this driver-focused chapter.

These resources together form a reading programme that will carry a motivated reader from this chapter's introduction to a practical fluency in FreeBSD virtualisation and containerisation. You do not need to read them all; pick the ones that fit your current project and build from there.

30.27 Appendix: Anti-Patterns in Virtualised Drivers

A good way to learn a craft is to study its common mistakes. This appendix collects the anti-patterns we have seen throughout the chapter, in a single place, with the fix for each. When reviewing a driver (your own or someone else's), scan for these patterns; each is a reliable sign of trouble.

30.27.1 Busy-Waiting on a Status Register

```
/* Anti-pattern */
while ((bus_read_4(sc->res, STATUS) & READY) == 0)
    ;
```

Under virtualisation, each bus read is a VM exit. A tight loop consumes enormous CPU time and may never terminate if the guest is scheduled out. Use `DELAY(9)` inside the loop and a bounded iteration count, or use an interrupt-driven design that does not poll at all.

30.27.2 Caching Physical Addresses

```
/* Anti-pattern */
uint64_t phys_addr = vtophys(buffer);
bus_write_8(sc->res, DMA_ADDR, phys_addr);
/* ...later... */
bus_write_8(sc->res, DMA_ADDR, phys_addr); /* still valid? */
```

A physical address is a temporary view. Under memory compaction, live migration, or memory hotplug, the address may no longer refer to the same physical memory. Use `bus_dma(9)` and hold a DMA map; the map tracks the bus address correctly across kernel memory operations.

30.27.3 Ignoring `si_drv1`

```
/* Anti-pattern */
static int
mydrv_read(struct cdev *dev, struct uio *uio, int flags)
{
    struct mydrv_softc *sc = devclass_get_softc(mydrv_devclass, 0);
    /* what if there are multiple units? */
}
```

The `dev->si_drv1` slot is there to connect the `cdev` back to its `softc`. Setting it in `attach` and using it in `read/write/ioctl` is the idiomatic pattern. Using `devclass_get_softc` with a hardcoded unit number is a minefield as soon as more than one instance attaches.

30.27.4 Assuming INTx

```
/* Anti-pattern */
sc->irq_rid = 0;
sc->irq_res = bus_alloc_resource_any(dev, SYS_RES_IRQ, &sc->irq_rid,
    RF_SHAREABLE | RF_ACTIVE);
```

INTx is slower under virtualisation and does not scale to many-queue devices. Use `pci_alloc_msix` first, fall back to `pci_alloc_msi`, and only fall back to INTx for hardware that does not support message-signalled interrupts.

30.27.5 Hard-Coding Feature Bits

```
/* Anti-pattern */
if ((sc->features & VIRTIO_F_RING_INDIRECT_DESC) == 0)
    panic("device does not support indirect descriptors");
```

A driver that panics when a feature is missing is impossible to use with a backend that does not advertise the feature. Negotiate optimistically, check the post-negotiation result, and fall back gracefully to a less efficient path if the feature is missing.

30.27.6 Sleep-With-Spin-Lock Held

```
/* Anti-pattern */
mtx_lock(&sc->lock);
tsleep(sc, PWAIT, "wait", hz);
mtx_unlock(&sc->lock);
```

`tsleep` with an `mtx(9)` spin-lock held causes a panic on FreeBSD. Use `mtx_sleep` (which drops the mutex around the sleep) or use a blocking lock (`sx(9)`) when you need to sleep.

30.27.7 Returning ENOTTY for Unsupported Operations

```
/* Anti-pattern */
case MYDRV_PRIV_IOCTL:
    if (cred->cr_prison != &prison0)
        return (ENOTTY); /* hide from jails */
    ...
```

ENOTTY means “this ioctl does not exist”. It hides the operation from jailed callers, but it also breaks tooling that introspects available ioctls. Prefer EPERM for “this ioctl exists but you cannot use it”, leaving introspection correct.

30.27.8 Forgetting the Detach Path

```
/* Anti-pattern */
static int
mydrv_detach(device_t dev)
{
    struct mydrv_softc *sc = device_get_softc(dev);
    /* release a few things and call it done */
    bus_release_resource(dev, SYS_RES_MEMORY, sc->res_rid, sc->res);
    return (0);
}
```

Missing callout drains, missing taskqueue drains, missing cdev destruction, missing mutex destruction, missing DMA map unload, missing interrupt teardown. Each becomes a leak or a crash at kldunload. The attach path is usually clean; the detach path is where bugs hide.

30.27.9 Assuming device_printf Works Without a Device

```
/* Anti-pattern */
static int
mydrv_modevent(module_t mod, int event, void *arg)
{
    device_printf(NULL, "load event"); /* crashes */
}
```

device_printf with a NULL device_t dereferences a null pointer. Inside module event handlers, before a device has been attached, use printf (with a “mydrv:” prefix to identify the source). device_printf is for per-device events after attach.

30.27.10 Not Cleaning Up VNET State

```
/* Anti-pattern */
static int
mydrv_mod_event(module_t mod, int event, void *arg)
{
    if (event == MOD_UNLOAD)
```

```

        free(mydrv_state, M_DEVBUF); /* in which VNET? */
    }

```

Per-VNET state is allocated in a specific VNET context. Freeing it without setting that context accesses the wrong VNET's data. Use `VNET_FOREACH` and `CURVNET_SET` to walk every VNET and clean up each one's state.

30.27.11 Using `getnanouptime` for Timing

```

/* Anti-pattern */
struct timespec ts;
getnanouptime(&ts);
/* ...do work... */
struct timespec ts2;
getnanouptime(&ts2);
/* difference is arbitrary inside a VM */

```

`getnanouptime` returns a low-resolution reading that the kernel caches. For precise short-duration timing, use `binuptime(9)` or `sbinuptime(9)`, which read the high-resolution time source. Under virtualisation, the precise readings are as correct as the time source allows; the cached readings have been stale for up to a tick.

30.27.12 Probing Hardware in the Probe Method

```

/* Anti-pattern */
static int
mydrv_probe(device_t dev)
{
    /* ...read status register... */
    return (BUS_PROBE_DEFAULT);
}

```

The probe method should be purely identity-based: inspect the PNP information, return a priority, and do nothing that requires the hardware to be present and functional. Hardware interaction belongs in `attach`. Under virtualisation, a probe that pokes hardware before feature negotiation can mis-identify the device.

30.27.13 Storing Kernel Pointers Through `ioctl`

```

/* Anti-pattern */
case MYDRV_GET_PTR:

```

```
*(void **)data = sc->internal_state;  
return (0);
```

Passing kernel pointers to user space is a security bug. Under virtualisation it can even leak hypervisor-relevant information. Always copy the data the caller asks for, never the pointer.

30.27.14 Summary

These anti-patterns cover most of the ways that FreeBSD drivers go wrong under virtualisation. They are not unique to VMs, but they are *amplified* under virtualisation: the same bug that corrupts memory once a day on bare metal might corrupt it hundreds of times per second under a VM whose timing is different. Fixing these patterns is not “fixing the VM case”; it is fixing the driver to meet the kernel’s baseline expectations.

30.28 Appendix: A Checklist for a Virtualisation-Ready Driver

A concrete checklist you can apply to your own driver before declaring it virtualisation-ready. Go through each item; any “no” answer is a task.

30.28.1 Device Binding and Probe

- Does the driver use `VIRTIO_SIMPLE_PNPINFO` or `VIRTIO_DRIVER_MODULE` rather than hand-rolling PNP entries?
- Does the probe method avoid hardware access and only use PNP identity?
- Does the attach method call `virtio_negotiate_features` and log the result?
- Does attach fail cleanly (releasing every resource it has allocated) if any step after virtqueue allocation fails?

30.28.2 Resources

- Does the driver use `bus_alloc_resource_any` with a dynamic RID rather than hard-coding resource IDs?
- Does the driver use `pci_alloc_msix` (or `pci_alloc_msi`) in preference to `INTx`?
- Are all `bus_alloc_resource` calls matched by `bus_release_resource` in detach?

30.28.3 DMA

- Does every address programmed into hardware come from `bus_dma_load` (or similar) rather than `vto-phys`?
- Does the driver hold DMA tags and maps with appropriate lifetimes, not re-creating them per operation when avoidable?
- Does the driver handle `bus_dma_load_mbuf_sg` correctly for scatter-gather?
- Is the driver’s `bus_dma_tag` created with the correct alignment, boundary, and maximum-segment constraints?

30.28.4 Interrupts

- Is the interrupt handler MP-safe (declared with `INTR_MPSAFE`)?
- Does the handler handle the “no work” case gracefully (in case of spurious wake-up)?
- Does the handler disable and re-enable virtqueue interrupts correctly, using the `virtqueue_disable_intr` / `virtqueue_enable_intr` pattern?
- Is the interrupt path free of blocking operations (no `malloc(M_WAITOK)`, no `mtx_sleep`)?

30.28.5 Character Device Interface

- Does attach set `cdev->si_drv1 = sc`?
- Does detach call `destroy_dev` before freeing the `softc`?
- Do read and write handle short I/O correctly (less than the full buffer size)?
- Does the driver check the result of `uiomove` for errors?
- Does `ioctl` use `priv_check` for operations that require elevated privilege?

30.28.6 Locking

- Is every `softc` access covered by the `softc`’s mutex?
- Is the detach path drain-then-free, not free-then-drain?
- Are sleeping waits done with `mtx_sleep` or `sx(9)` rather than `tsleep` with a mutex held?
- Is there a discernible ordering of acquisitions (to avoid deadlock)?

30.28.7 Timing

- Does the driver use `DELAY(9)`, `pause(9)`, or `callout(9)` rather than busy loops?
- Does the driver avoid reading the TSC directly?
- Does every wait have a bounded timeout, with a sensible error on exceed?

30.28.8 Privilege

- Does the driver call `priv_check` for all operations that should not be available to unprivileged users?
- Does the driver use the correct privilege (`PRIV_DRIVER`, `PRIV_IO`, not `PRIV_ROOT`)?
- Does the driver consider jailed callers, using `priv_check` which also calls `prison_priv_check`?

30.28.9 Detach and Unload

- Does detach drain all callouts (`callout_drain`)?
- Does detach drain all taskqueues (`taskqueue_drain_all`)?
- Does detach stop any kernel threads the driver has created (via condition variables or similar)?
- Does detach release every resource attach allocated?
- Can the module be unloaded at any time (no hangs on `kldunload`)?

30.28.10 VNET (if applicable)

- Does the driver register VNET sysinit/sysuninit for per-VNET state?
- Does the driver use V_ prefix macros for per-VNET variables?
- Does the VNET move path allocate state on entry and free on exit?
- Does the driver use CURVNET_SET / CURVNET_RESTORE when accessing a VNET other than the current one?

30.28.11 Testing

- Does the driver have a make test target (even if it just builds and loads)?
- Has the driver been run through the attach-detach cycle at least 100 times?
- Has the driver been loaded under both VirtIO and passthrough (if applicable)?
- Has the driver been loaded inside at least one jail and one VNET jail?
- Has the driver been built on at least amd64; ideally also on arm64?

30.28.12 Documentation

- Is there a README explaining what the driver does and how to build it?
- Is there a manual page describing the driver's user-visible interface?
- Is the PNP table complete enough that kldxref finds the driver for auto-loading?

A driver that passes this checklist is well on its way to being virtualisation-ready. A driver that fails several items needs attention before it will behave well under the diverse environments of modern deployment. Run through the list for every driver you write; it is faster than debugging each issue individually when the driver hits a customer.

30.29 Appendix: Sketching a bhyve Backend for vtedu

Challenge 5 asks the reader to write a bhyve(8) backend for the vtedu driver. This appendix sketches the architecture of such a backend at a level of detail useful for planning. It is not a complete implementation; writing one is the challenge. The goal here is to demystify the backend side of the VirtIO story so the challenge becomes tractable.

30.29.1 Where the Code Lives

The bhyve(8) user-space emulator lives under `/usr/src/usr.sbin/bhyve/`. Its source files are a mixture of CPU and chipset emulation, per-device emulators for different VirtIO types, and glue code that connects bhyve(8) to the vmm(4) kernel module. The relevant per-device files for VirtIO are:

- `/usr/src/usr.sbin/bhyve/pci_virtio_rnd.c`: virtio-rnd (random number generator). The simplest VirtIO backend. Read first.
- `/usr/src/usr.sbin/bhyve/pci_virtio_block.c`: virtio-blk (block device).
- `/usr/src/usr.sbin/bhyve/pci_virtio_net.c`: virtio-net (network).

- `/usr/src/usr.sbin/bhyve/pci_virtio_9p.c`: virtio-9p (filesystem share).
- `/usr/src/usr.sbin/bhyve/pci_virtio_console.c`: virtio-console (serial).

Each of these is a few hundred to a couple of thousand lines of code. They share a common backend framework in `/usr/src/usr.sbin/bhyve/virtio.h` and `/usr/src/usr.sbin/bhyve/virtio.c`. The `pci_virtio_*.c` files above are per-device consumers of that framework; each one registers a struct `virtio_consts`, a set of virtqueue callbacks, and the device-specific config-space layout.

30.29.2 The Framework Handles the Protocol

The good news for a backend author is that `bhyve(8)` already implements the VirtIO protocol. Feature negotiation, descriptor-ring management, notification delivery, interrupt injection, all of it lives in the `virtio.c` framework. A new backend implements only the device-specific behaviour: what happens when a buffer arrives on the virtqueue, what config-space fields the device exposes, and how device-level events are generated.

The framework-facing interface is a small set of callbacks, encapsulated in a struct `virtio_consts`.

```
/* Sketch, not actual bhyve code. */
struct virtio_consts {
    const char *vc_name;
    int vc_nvq;
    size_t vc_cfgsize;
    void (*vc_reset)(void *);
    void (*vc_qnotify)(void *, struct vqueue_info *);
    int (*vc_cfgread)(void *, int, int, uint32_t *);
    int (*vc_cfgwrite)(void *, int, int, uint32_t);
    void (*vc_apply_features)(void *, uint64_t);
    uint64_t vc_hv_caps;
};
```

A new backend fills in this struct and registers it with the framework. The framework calls into the backend when the guest-side driver does interesting things (resets the device, notifies the virtqueue, reads or writes config space).

30.29.3 Sketching the vtedu Backend

For `vtedu`, the backend is simple. It has one virtqueue, no config-space fields beyond the generic ones, and one feature bit (`VTEDU_F_UPPERCASE`). Its state is:

```
struct pci_vtedu_softc {
    struct virtio_softc vsc_vs;
    struct vqueue_info vsc_vq; /* just one queue */
    pthread_mutex_t vsc_mtx;
    uint64_t vsc_features;
};
```


The callbacks are small.

```
static void
pci_vtedu_reset(void *vsc)
{
    struct pci_vtedu_softc *sc = vsc;

    pthread_mutex_lock(&sc->vsc_mtx);
    vi_reset_dev(&sc->vsc_vs);
    sc->vsc_features = 0;
    pthread_mutex_unlock(&sc->vsc_mtx);
}

static void
pci_vtedu_apply_features(void *vsc, uint64_t features)
{
    struct pci_vtedu_softc *sc = vsc;

    pthread_mutex_lock(&sc->vsc_mtx);
    sc->vsc_features = features;
    pthread_mutex_unlock(&sc->vsc_mtx);
}
```

The interesting callback is `vc_qnotify`, called when the guest notifies the virtqueue.

```
static void
pci_vtedu_qnotify(void *vsc, struct vqueue_info *vq)
{
    struct pci_vtedu_softc *sc = vsc;
    struct iovec iov[1];
    uint16_t idx;
    int n;

    while (vq_has_descs(vq)) {
        n = vq_getchain(vq, &idx, iov, 1, NULL);
        if (n < 1) {
            EPRINTLN("vtedu: empty chain");
            vq_relchain(vq, idx, 0);
            continue;
        }

        if ((sc->vsc_features & VTEDU_F_UPPERCASE) != 0) {
            for (int i = 0; i < iov[0].iov_len; i++) {
                uint8_t *b = iov[0].iov_base;
```

```
        if (b[i] >= 'a' && b[i] <= 'z')
            b[i] = b[i] - 'a' + 'A';
    }

    vq_relchain(vq, idx, iov[0].iov_len);
}

vq_endchains(vq, 1);
}
```

That is the core of it. The `vq_has_descs`, `vq_getchain`, `vq_relchain`, and `vq_endchains` calls are framework helpers that unpack the virtqueue descriptors into `iovec` structures and repackage the results.

30.29.4 Connecting to the Device Table

`bhyve(8)` maintains a table of device emulators; each backend registers itself at build time. The registration uses a `PCI_EMUL_TYPE( ... )` macro (or similar) that adds the backend's vtable to a linker-set. Once registered, the `bhyve(8)` command line can reference the backend by name:

```
bhyve ... -s 7,virtio-edu guest0
```

The `-s 7,virtio-edu` adds the `virtio-edu` device at PCI slot 7. When the guest boots, the FreeBSD kernel enumerates the PCI bus, finds the device with VirtIO vendor ID and the right device ID, and attaches `vtedu` to it.

30.29.5 What the Backend Must Verify

To make the backend correct, the author must verify:

- The VirtIO device ID matches the driver's expectation (`VIRTIO_ID_EDU = 0xfff0`).
- The feature-negotiation response matches what the driver expects (advertise `VTEDU_F_UPPERCASE` and `VIRTIO_F_VERSION_1`).
- The virtqueue sizes are large enough for the driver's workload (256 is a reasonable default).
- The config-space size matches what the driver reads (zero for `vtedu`).

30.29.6 Testing the End-to-End Loop

With both sides in place, the end-to-end test looks like this.

On the host, after building the backend and installing the modified `bhyve(8)`:

```
sudo bhyve ... -s 7,virtio-edu guest0
```

Inside the guest, after copying and building `vtedu.c`:

```
sudo kldload ./virtio_edu.ko
ls /dev/vtedu0
echo "hello world" > /dev/vtedu0
cat /dev/vtedu0
```

The expected output of `cat` is `HELLO WORLD` (uppercased by the backend).

If the output is `hello world` (not uppercased), the backend did not negotiate `VTEDU_F_UPPERCASE`. If the `echo` hangs, the virtqueue notification is not reaching the backend. If the `cat` hangs, the backend's interrupt injection is not reaching the guest. Each of these is a specific failure that the debugging techniques from the chapter can localise.

30.29.7 Why This Exercise Is Worth the Effort

Writing a backend and a driver teaches both sides of the VirtIO story. The driver side is what most authors eventually write, but the backend side tells you *why* the VirtIO protocol is shaped the way it is. Feature bits make sense when you have to decide which to advertise. Virtqueue descriptors make sense when you have to unpack them. Interrupt delivery makes sense when you have to inject one.

Completing Challenge 5 promotes you from “VirtIO user” to “VirtIO author”, which is a different level of understanding. If the challenge feels large, tackle it in stages: first get the device to appear in the guest (verify with `pciconf -lv`), then get feature negotiation to work, then handle a single virtqueue message, then polish the full pipeline. Each stage is a separate commit and a separate satisfying milestone.

This is the end of the chapter's technical material. The remaining prose ties together what you have learned and points forward to Chapter 31.

30.30 Appendix: Running the Chapter's Techniques in CI

Continuous integration is now a standard part of most driver projects. This short appendix describes how the chapter's techniques fit into a CI pipeline. The goal is to show that virtualisation and containerisation are not just runtime concerns; they are also practical tools for keeping a driver honest across changes.

30.30.1 Why CI Benefits from Virtualisation

A CI system is, among other things, a place where you need reproducible test environments. Running tests on bare metal is possible but fragile: the test machine accumulates state, different machines have different hardware, and failures are hard to separate from hardware quirks. Running tests inside a VM removes most of these problems. The VM is a clean slate at the start of each run, its “hardware” is uniform across test runs, and its failures are the driver's failures, not the host's.

For FreeBSD driver CI, the standard approach is to run a FreeBSD guest under `bhyve(8)` (if the CI host is FreeBSD) or under KVM/QEMU (if the CI host is Linux). The guest boots a FreeBSD image, loads the driver, runs the test harness, and exits. The whole cycle takes under a minute for a small driver, which means hundreds of tests per day against every commit.

30.30.2 A Minimal CI Flow for a VirtIO Driver

A reasonable flow for a VirtIO driver's CI is:

1. Check out the driver source.
2. Build the driver against the target FreeBSD kernel (often using a cross-compile on the CI host).
3. Start a FreeBSD VM with appropriate VirtIO devices.
4. Copy the built module into the VM.
5. SSH into the VM and load the module.
6. Run the test harness.
7. Capture the output.
8. Shut down the VM.
9. Report pass/fail.

Steps 1 and 2 are unchanged from a non-VM workflow. Steps 3 through 9 are what virtualisation adds.

30.30.3 Practical Tools

For step 3, `vm-bhyve` is convenient on FreeBSD hosts. For Linux CI hosts, `virt-install` from `libvirt` is a standard tool. Both produce a running VM in a few seconds with a pre-built image.

For step 4, a shared volume or a small SSH copy is usual. `virtfs (9P)` or `virtiofs` pass host directories into the guest; `scp` over a tap interface works as well.

For step 5, pre-installed SSH keys and a static IP address (or a DHCP reservation) make the connection painless.

For steps 6 and 7, the test harness is whatever the driver author writes: a shell script, a C program, a Python harness. Whatever it is, it runs inside the VM.

For step 8, `vm stop guest0 --force` (or equivalent) shuts the VM down rapidly. The image is discarded; the next run starts fresh.

For step 9, the exit code of the test harness determines pass/fail. CI systems expect zero for success and nonzero for failure; be consistent.

30.30.4 A Minimal Test Harness

A simple pass/fail harness for a VirtIO driver might look like this.

```
#!/bin/sh
set -e
```

```
# Inside the guest. Expects the module at /tmp/mydriver.ko.

kldload /tmp/mydriver.ko

# Wait for the device to attach.
for i in 1 2 3 4 5; do
    if [ -c /dev/mydev0 ]; then
        break
    fi
    sleep 1
done

if [ ! -c /dev/mydev0 ]; then
    echo "FAIL: /dev/mydev0 did not appear"
    exit 1
fi

# Exercise the device.
echo "hello" > /dev/mydev0
output=$(cat /dev/mydev0)
if [ "$output" != "hello" ]; then
    echo "FAIL: expected 'hello', got '$output'"
    exit 1
fi

# Clean up.
kldunload mydriver

echo "PASS"
exit 0
```

Short, readable, and reports a clear result. CI scales with tests like this: add one per feature, run them all on every commit.

30.30.5 Scaling to Multiple Configurations

A CI pipeline can run the same driver under multiple configurations by spinning up multiple VMs. Useful axes include:

- Kernel version (FreeBSD 14.3, 14.2, 13.5, -CURRENT).
- Architecture (amd64, arm64 with an arm64 guest).
- VirtIO feature set (force-disable certain features on the backend to exercise fallback paths).
- Hypervisor (bhyve, QEMU/KVM, VMware) where support varies.

Each configuration is an independent job that the CI system parallelises. The aggregate of “driver passes on every configuration” is a strong signal that the driver is robust.

30.30.6 A Note on Hardware-in-the-Loop CI

For drivers that talk to real hardware, CI needs a bare-metal or passthrough setup. This is more expensive and less common, but some projects maintain a small fleet of test machines for this purpose. The techniques from the chapter apply: a hardware test rig uses ppt(4) passthrough to give a guest access to a specific device, and the CI system drives the guest the same way it would drive a pure-VirtIO guest.

Hardware CI is slower to set up and more expensive to run. For most projects, pure-VirtIO CI is enough for the bulk of tests, with a small suite of hardware tests run at a slower cadence.

30.30.7 The Payoff

CI that exercises a driver under realistic conditions catches regressions quickly, while the fix is still fresh in the author’s mind. A regression caught at commit-time takes minutes to fix; a regression caught weeks later during a release candidate takes hours. Virtualisation makes the former affordable, and that is one of the strongest arguments for taking the techniques in this chapter seriously.

30.31 Appendix: Commands Cheat Sheet

A compact list of the commands a driver author uses most often when working with virtualisation and containerisation on FreeBSD. Keep this page open while working through the labs.

30.31.1 Host-Side Virtualisation

```
# Check hypervisor extensions
sysctl hw.vmm

# Load/unload vmm(4)
kldload vmm
kldunload vmm

# vm-bhyve guest management
vm list
vm start guest0
vm stop guest0
vm console guest0
vm configure guest0
vm install guest0 /path/to/iso
vm create -t default -s 20G guest0
```

```
# Direct bhyve (verbose but informative)
bhyectl --vm=guest0 --destroy
bhyectl --vm=guest0 --suspend=normal
```

30.31.2 Guest-Side Inspection

```
# Is this a guest?
sysctl kern.vm_guest

# Which devices attached?
pciconf -lvBb
devinfo -v
kldstat -v

# VirtIO-specific
dmesg | grep -i virtio
sysctl dev.virtio_pci
sysctl dev.virtqueue
```

30.31.3 PCI Passthrough

```
# Mark a device as passthrough-capable (in /boot/loader.conf)
pptdevs="5/0/0"

# Verify after reboot
pciconf -lvBb | grep ppt
dmesg | grep -i dmar      # Intel
dmesg | grep -i amdvi     # AMD
```

30.31.4 Jails

```
# Create and manage jails
jail -c name=test path=/jails/test host.hostname=test ip4=inherit persist
jls
jexec test /bin/sh
jail -r test

# devfs rulesets
```

```
devfs rule -s 100 show
devfs -m /jails/test/dev rule -s 100 applyset
```

30.31.5 VNET

```
# epair setup
kldload if_epair
ifconfig epair create
ifconfig epair0a 10.0.0.1/24 up

# Move one end into a VNET jail
jail -c name=vnet-test vnet vnet.interface=epair0b path=/jails/vnet-test persist

# Confirm
ifconfig -j vnet-test epair0b
```

30.31.6 Resource Limits

```
# rctl
rctl -a jail:test:memoryuse:deny=512M
rctl -a jail:test:pcpu:deny=50
rctl -h jail:test
rctl -l jail:test
```

30.31.7 Observability

```
# Interrupt rate
vmstat -i

# VM exit counts (needs PMC)
pmcstat -S VM_EXIT -l 10

# DTrace virtqueue activity
dtrace -n 'fbt::virtqueue_enqueue:entry { @[probefunc] = count(); }'

# Kernel trace
ktrace -i -p $(pgrep bhyve)
kdump -p $(pgrep bhyve)
```


30.31.8 Module Lifecycle

```
# Build, load, test, unload  
make clean && make  
sudo kldload ./mydriver.ko  
kldstat -v | grep mydriver  
# ...exercise driver...  
sudo kldunload mydriver
```

These are the day-to-day commands. A quick-reference card like this, pinned to a wall or kept in a terminal tab, saves hours over the course of a project.

With that, the chapter is complete.

31 Security Best Practices

31.1 Introduction

You reach Chapter 31 with an understanding of environment that few authors have even asked you to build. Chapter 29 taught you to write drivers that survive changes of bus, architecture, and word size. Chapter 30 taught you to write drivers that behave correctly when the machine under them is not a machine at all, but a virtual one, and when the process using them is not on the host but in a jail. Both chapters have been about boundaries: the boundary between hardware and kernel, the boundary between host and guest, the boundary between host and container. Chapter 31 asks you to look at a different boundary, one that sits closer to home than any of those, and one that is easier to forget precisely because it runs through the middle of your own code.

The boundary this chapter is about is the boundary between the kernel and everyone who talks to it. Userland programs, the hardware itself, other parts of the kernel, the bootloader passing you a parameter, a firmware blob that arrived from the vendor's support site last week, a device that started behaving strangely after an upgrade. Every one of them is on the other side of a trust edge from the driver you wrote. Every one of them can, deliberately or accidentally, pass the driver something that does not match its expectations. A driver that respects this boundary is a driver that can be trusted to defend the kernel. A driver that does not respect it is a driver that, the day something hostile arrives, will let that hostility reach code it should never have reached.

This chapter is about building that habit of respect. It is not a security textbook, and it will not try to turn you into a vulnerability researcher. What it will do is teach you to see your driver the way an attacker or a careless program might see it, to recognise the specific classes of mistake that turn a small bug into a full system compromise, and to reach for the right FreeBSD primitives when you want to prevent those mistakes from happening.

A few words about what this means in practice. A kernel security bug is not just a worse version of a user-space security bug. A buffer overflow in a user-space program can corrupt that program's memory; a buffer overflow in a driver can corrupt the kernel, and the kernel serves every program on the system. An off-by-one error in a user-space parser may crash the parser; the same off-by-one in a driver may hand an attacker a way to read kernel memory that holds another user's secrets, or to write arbitrary bytes into the kernel's function tables. The consequences do not scale linearly with the size of the bug. They scale with the privilege of the code where the bug lives, and the kernel sits at the top of that hierarchy.

This is why driver security is not a separate subject, added on top of the programming skills you have been building. It is those programming skills, applied with a particular discipline in mind. The `copyin()` and `copyout()` calls you saw in earlier chapters become tools for enforcing a trust boundary. The `malloc()` flags you learned become ways to control memory lifetime. The locking discipline you practised becomes a way to prevent races that an attacker could otherwise steer. The privilege checks you saw briefly in Chapter 30 become a first line of defence against unprivileged callers reaching into places they should not be able to reach. Security is, in a

real sense, the discipline of writing kernel code well, held to a higher standard and examined through the eyes of someone who wishes it to fail.

The chapter builds that view in ten steps. Section 1 motivates the subject and explains why a driver's security model is different from an application's. Section 2 works through the mechanics of buffer overflows and memory corruption in kernel code, a topic that is easy to misunderstand because the ordinary C you learned in Chapter 4 has pitfalls that grow teeth inside the kernel. Section 3 covers user input, the single largest source of exploitable bugs in drivers, and walks through safe use of `copyin(9)`, `copyout(9)`, and `copyinstr(9)`. Section 4 turns to memory allocation and lifetime, including the FreeBSD-specific flags on `malloc(9)` that matter for safety. Section 5 examines race conditions and time-of-check-to-time-of-use bugs, a class of issue that sits at the intersection of concurrency and security. Section 6 covers access control: how a driver should check whether a caller is allowed to do what they are asking for, using `priv_check(9)`, `ucred(9)`, `jailed(9)`, and the `securelevel` machinery. Section 7 addresses information leaks, the subtle class of bug in which a driver gives away data it did not intend to. Section 8 looks at logging and debugging, which can themselves become security problems if they are casual about what they print. Section 9 distils a set of design principles around secure defaults and fail-safe behaviour. Section 10 rounds out the chapter with testing, hardening, and a practical introduction to the tools FreeBSD gives you for finding these bugs before someone else does: `INVARIANTS`, `WITNESS`, the kernel sanitizers, `Capsicum(4)`, and fuzzing with `syzkaller`.

Alongside those ten sections, the chapter will touch on the `mac(4)` framework, `Capsicum`'s role in constraining `ioctl` callers, string-safety idioms such as `copyinstr(9)`, and the build pitfalls that come with modern kernel hardening features like ASLR, SSP, and PIE for loadable modules. Each of these arrives where it is most relevant, never at the cost of the core thread.

One last thing before we begin. Writing secure kernel code is not a matter of reading a checklist and ticking items off. It is a way of reading your own code, and a set of reflexes that become automatic over time. The first time you see a driver that calls `copyin` into a fixed-size stack buffer without checking the length, you may wonder what is wrong with it; the hundredth time you see that pattern, you will feel the hair on the back of your neck stand up. The goal of this chapter is not to teach you every variant of every vulnerability; that would take a shelf of books. The goal is to help you build the reflexes. Once you have them, you will write drivers that are safer by default, and you will spot the dangerous patterns in other people's code before they cause trouble.

Let us begin.

31.2 Reader Guidance: How to Use This Chapter

Chapter 31 is conceptual in a way that a few of the earlier chapters were not. The code samples are short and focused; the value is in the thinking they teach. You can work through the whole chapter by reading carefully, and you will come out of it a better driver author even if you never type a single line. The labs at the end turn the thinking into muscle memory, and the challenges push the thinking into uncomfortable corners where real bugs live, but the text itself is the main teaching surface of the chapter.

If you choose the **reading-only path**, plan for roughly three to four focused hours. At the end you will be able to recognise the major classes of driver security bug, to explain why kernel-level bugs change the whole system's trust model, to describe the FreeBSD primitives that defend against each class, and to sketch what a safe version of a given unsafe pattern should look like. That is a substantial body of knowledge, and for many readers it is where the chapter should end on the first pass.

If you choose the **reading-plus-labs path**, plan for eight to twelve hours spread across two or three sessions. The labs build on a tiny pedagogical driver called `secdev` that you will write over the course of the chapter. Each lab is a short, focused exercise: in one, you will fix a deliberately unsafe `ioctl` handler; in another, you will add `priv_check(9)` and observe what happens when unprivileged and jailed processes try to use a restricted entry point; in another, you will introduce a race condition, watch it manifest under `WITNESS`, and then fix it; and in a final lab, you will run a simple fuzzer against the driver's `ioctl` surface and read the resulting crash reports. Each lab leaves you with a working system and an entry in your lab logbook; none of them are long enough to exhaust an evening.

If you choose the **reading-plus-labs-plus-challenges path**, plan for a long weekend or two. The challenges push `secdev` into real territory: you will add MAC policy hooks so that a site-local policy can override the driver's defaults, you will tag the driver's `ioctls` with capability rights so that a Capsicum-confined process can still use the safe subset, you will write a short `syzkaller` description file for the driver's entry points, and you will run the sanitized kernel variants (`KASAN`, `KMSAN`) to see what they catch that the normal `INVARIANTS` build misses. Each challenge is self-contained; none require reading additional chapters to complete.

A note on the lab environment. You will continue to use the throwaway FreeBSD 14.3 machine from earlier chapters. The labs in this chapter do not need a second machine, do not need `bhyve(8)`, and do not need to modify the host in ways that would persist after a reboot. You will load and unload kernel modules, you will write to a test character device, you will read `dmesg` carefully, and you will edit a small tree of source files. If something goes wrong, a reboot will recover the host. A snapshot or boot environment is still a good idea, and it is cheap to create.

One piece of special advice for this chapter: **read slowly**. Security prose is sometimes deceptively gentle. The ideas look obvious on the page, but the reason a given bug is a bug can take a minute of thinking through before it clicks. Resist the temptation to skim. If a paragraph describes a race condition that you do not fully understand, stop and reread it. If a code snippet demonstrates an information leak, trace the path of the leaked bytes in your head until you can name which byte came from where. The reward for careful reading here is a set of reflexes that will outlive this book.

31.2.1 Prerequisites

You should be comfortable with everything from earlier chapters. In particular, this chapter assumes you already understand how to write a loadable kernel module, how the driver's `open`, `read`, `write`, and `ioctl` entry points connect to `/dev/` nodes, how `softc` is allocated and attached to a `device_t`, how mutexes and reference counts work at the level Chapter 14 and Chapter 21 taught, and how interrupts and callouts interact with sleeping paths. If any of that is shaky, a brief revisit before starting will make the examples land more firmly.

You should also be comfortable with ordinary FreeBSD system administration: reading `dmesg`, loading and unloading modules, running commands as an unprivileged user, creating a simple jail, and using `sysctl(8)` to observe and tune the system. The chapter will refer to these tools without walking you through each one from scratch.

No prior background in security research is needed. The chapter builds its vocabulary from the ground up.

31.2.2 What This Chapter Does Not Cover

A responsible chapter tells you what it leaves out. This chapter does not teach exploit development. It does not teach you how to write shellcode, how to build a ROP chain, or how to turn a crash into code execution. Those are legitimate subjects, but they belong to a different kind of book, and the skills they require are not the skills that help you write safer drivers.

This chapter does not turn you into a security auditor. Auditing a large codebase for every class of vulnerability is a distinct discipline with its own tools and rhythms. What the chapter does give you is the ability to audit your own driver competently, and to recognise the patterns worth flagging in someone else's.

The chapter does not replace the FreeBSD Security Advisories, the CERT C coding standard, the SEI's guidance on secure kernel programming, or the manual pages of the APIs it discusses. It points you toward those sources and expects you to consult them when a specific question goes deeper than the chapter can follow. Each major section of the chapter ends with a short pointer to the relevant manual pages so that your first port of call after the chapter is the FreeBSD documentation itself.

Finally, the chapter does not attempt to cover every possible bug class. It focuses on the classes that matter most for drivers and that can be addressed with FreeBSD primitives the reader already knows or can learn in a few pages. Some exotic classes of bug, such as Spectre-style speculative execution side channels, are mentioned only in passing; they belong to specialised hardening work that most driver authors do not and should not write from scratch.

31.2.3 Structure and Pacing

Section 1 builds the mental model: what is at risk when a driver goes wrong, and how the driver's security model differs from an application's. Section 2 addresses buffer overflows and memory corruption in kernel code, including the subtle ways they differ from their user-space cousins. Section 3 teaches safe handling of user-provided input through `copyin(9)`, `copyout(9)`, `copyinstr(9)`, and related primitives. Section 4 covers memory allocation and lifetime: the flags on `malloc(9)`, the difference between `free(9)` and `zfree(9)`, and the use-after-free patterns that kernel modules fall into. Section 5 turns to races and TOCTOU bugs, including the security-relevant ways they manifest. Section 6 covers access control and privilege enforcement, from `priv_check(9)` through `ucred(9)` and jails to the `securelevel` machinery. Section 7 addresses information leaks and the surprisingly subtle ways data escapes. Section 8 discusses logging and debugging, which can themselves become security problems. Section 9 collects the principles of secure defaults and fail-safe design. Section 10 covers testing and hardening: `INVARIANTS`, `WITNESS`, `KASAN`, `KMSAN`, `KCOV`, `Capsicum(4)`, the `mac(4)` framework, and a walkthrough of running `syzkaller` against a driver's `ioctl` surface. The labs and challenges follow, along with a closing bridge into Chapter 32.

Read the sections in order. Each one assumes the previous one, and the final two sections (Sections 9 and 10) synthesise what came before into practical advice and a workflow.

31.2.4 Work Section by Section

Each section in this chapter covers one core idea. Do not try to hold two sections in your head at the same time. If a section ends and you feel uncertain about one of its points, pause before starting the next one, re-read the

closing paragraphs, and look up the cited manual pages. A five-minute pause to consolidate is almost always faster than discovering two sections later that the foundation was not quite firm.

31.2.5 Keep the Reference Driver Close

The chapter builds a small pedagogical driver called `secdev` across its labs. You will find it, along with starter code, intentionally broken versions, and fixed variants, under `examples/part-07/ch31-security/`. Each lab directory contains the state of the driver at that step, along with its `Makefile`, a brief `README.md`, and any supporting scripts. Clone the directory, type along, and load each version after every change. Running unsafe code on your lab machine and watching what happens is part of the lesson; do not skip the live tests.

31.2.6 Open the FreeBSD Source Tree

Several sections point to real FreeBSD files. The ones that repay careful reading in this chapter are `/usr/src/sys/sys/systm.h` (for the exact signatures of `copyin`, `copyout`, `copyinstr`, `bzero`, and `explicit_bzero`), `/usr/src/sys/sys/priv.h` (for the `priv` constants and the `priv_check` prototypes), `/usr/src/sys/sys/ucrd.h` (for the credential structure), `/usr/src/sys/sys/jail.h` (for the `jailed()` macro and `prison` structure), `/usr/src/sys/sys/malloc.h` (for the allocation flags), `/usr/src/sys/sys/sbuf.h` (for the safe string builder), `/usr/src/sys/sys/capsicum.h` (for capability rights), `/usr/src/sys/sys/sysctl.h` (for the `CTLFLAG_SECURE`, `CTLFLAG_PRISON`, `CTLFLAG_CAPRD`, and `CTLFLAG_CAPWR` flags), and `/usr/src/sys/kern/kern_priv.c` (for the `priv-check` implementation). Open them when the chapter points you at them. The source is the authority; the book is a guide into it.

31.2.7 Keep a Lab Logbook

Continue the lab logbook from earlier chapters. For this chapter, log a short note for each lab: which commands you ran, which modules loaded, what `dmesg` said, what surprised you. Security work, more than most, benefits from a paper trail, because the bugs it teaches you to see are often invisible until you look for them in the right way, and a logbook entry from last week may save you an hour of re-discovery this week.

31.2.8 Pace Yourself

Several ideas in this chapter land more firmly the second time you meet them than the first. Feature bits in `virtio` made more sense in Chapter 30 after a day's rest; the same thing happens here with, say, the distinction between `copyin` error handling and TOCTOU-safe re-copying. If a subsection blurs the first time through, mark it, move on, and come back to it. Security reading rewards patience.

31.3 How to Get the Most Out of This Chapter

Chapter 31 rewards a particular kind of engagement. The specific primitives it introduces, `priv_check(9)`, `copyin(9)`, `sbuf(9)`, `zfree(9)`, `ppsratcheck(9)`, are not decorative; they are the bricks of secure driver code. The most valuable habit you can build while reading this chapter is the habit of asking two questions at every single call site: where did this data come from, and who is allowed to cause it to be here?

31.3.1 Read With a Hostile Mind

Security reading asks for a shift in how you look at code. When the chapter shows you a driver that copies `len` bytes from user space into a buffer, do not read the snippet as if the value of `len` is reasonable. Read it as if `len` is `0xFFFFFFFF`. Read it as if `len` is a carefully chosen value that passes one obvious check and fails a subtler one. Read the code the way a bored, clever, unfriendly person might read it before going to bed. That is the reading that finds bugs.

31.3.2 Run What You Read

When the chapter introduces a primitive, run a small example of it. When it shows a pattern for `priv_check`, write a two-line kernel module that calls `priv_check` with a specific constant and observe what happens when you invoke its `ioctl` from a non-root process. When it describes the effect of `CTLFLAG_SECURE` on a `sysctl`, set a dummy `sysctl` in your lab module, raise and lower the `securelevel`, and watch the behaviour change. The running system teaches what prose alone cannot.

31.3.3 Type the Labs

Every line of code in the labs is there to teach something. Typing it yourself slows you down enough to notice the structure. Copy-pasting the code often feels productive and usually is not; the finger-memory of typing kernel code is part of how you learn it. Even when a lab asks you to fix a deliberately unsafe file, type the fix yourself rather than pasting in the suggested answer.

31.3.4 Treat `dmesg` as Part of the Manuscript

Several of the bugs in this chapter surface only in kernel log output. A `KASSERT` firing, a `WITNESS` complaint about an out-of-order lock acquisition, a rate-limited warning from your own `log(9)` call, all of these appear in `dmesg` and nowhere else. Watch `dmesg` during the labs. Tail it in a second terminal. Copy relevant lines into your logbook when they teach something non-obvious.

31.3.5 Break Things Deliberately

At several points in the chapter, and explicitly in some of the labs, you will be asked to run unsafe code to see what happens. Do it. A kernel panic on your lab machine is a cheap educational experience. Unload the module after each experiment, note the symptom in your logbook, and move on. A panic in a production system is expensive; the whole point of a lab environment is to give you the freedom to learn these lessons where they are cheap.

31.3.6 Work in Pairs When You Can

If you have a study partner, this chapter is a good one to pair on. Security work benefits enormously from a second pair of eyes. One of you can read the code looking for bugs while the other reads the prose; then you can swap and compare notes. The two modes of reading find different things, and the conversation itself is educational.

31.3.7 Trust the Iteration

You will not remember every flag, every constant, every priv identifier on the first pass. That is fine. What matters is that you remember the shape of the subject, the names of the primitives, and where to look when a concrete question comes up. The specific identifiers become reflex after you have written two or three security-conscious drivers; they are not a memorisation exercise.

31.3.8 Take Breaks

Security reading is cognitively intense in a different way from performance work or bus-plumbing work. It asks you to hold a model of an adversary in your head while you read code designed to serve a friend. Two hours of focused reading followed by a real break is almost always more productive than four hours of grinding.

With those habits in place, let us begin with the question that frames everything else: why does driver security matter?

31.4 Section 1: Why Driver Security Matters

It is tempting to think of driver security as a subset of software security in general, with the same techniques and the same consequences, just applied to a different codebase. That framing is not wrong, exactly, but it misses what is distinctive about drivers. The reason drivers deserve their own chapter on security is that the consequences of a security bug in a driver are different from the consequences of the same bug in a user-space program, and the defenses look different too. This section builds the mental model that the rest of the chapter rests on.

31.4.1 What the Kernel Trusts

The kernel is the only part of the system that is trusted to do certain things. It is the only piece of software that can read or write any physical memory address. It is the only piece of software that can talk directly to hardware. It is the only piece of software that can grant or revoke privileges for user-space processes. It is the piece of software that holds the secrets of every user and the credentials of every running program. When it decides whether a given request should succeed, nothing above it can overrule that decision.

That privilege is the whole point of having a kernel. Without it, the kernel would not be able to enforce the boundaries that make a multi-user system possible. With it, the kernel carries a responsibility that no user-space program carries: every line of kernel code runs with the authority of the whole system, and every bug in kernel code can, in principle, be escalated into the authority of the whole system.

A driver is part of the kernel. Once loaded, a driver's code runs with the same privilege as the rest of the kernel. There is no finer-grained boundary inside the kernel that says "this code is only a driver, so it cannot touch the scheduler." A pointer dereference in your driver, if it lands on the wrong address, can corrupt any data structure the kernel uses. A buffer overflow in your driver, if it is large enough, can overwrite any function pointer the kernel uses. An uninitialised value in your driver, if it flows into the right place, can leak a neighbour's secrets. The kernel trusts the driver completely, because it has no mechanism to distrust it.

That asymmetry is the first thing to internalise. User-space programs run under the kernel, and the kernel can enforce rules on them. Drivers run inside the kernel, and no one enforces rules on them except the driver authors themselves.

31.4.2 A Kernel Bug Changes the Trust Model

A bug in a user-space program is a bug. A bug in the kernel, and particularly in a driver that an unprivileged process can reach, is often something worse: it is a change to the trust model of the whole system. This is the single most important idea in this chapter, and it is worth dwelling on.

Consider a tiny bug in a user-space text editor: an off-by-one error that writes one extra byte to a buffer. In the worst case, the editor crashes. Maybe the user loses a few minutes of work. Maybe the editor's sandbox catches the crash and the impact is even smaller. The consequences are bounded by what that user could already do; the editor was running with the user's privileges, so the damage cannot escape those privileges.

Now consider the same off-by-one error in a driver's `ioctl` handler. If the driver is reachable from unprivileged processes, an unprivileged user can trigger the off-by-one. The one extra byte lands in kernel memory. Depending on where it lands, it could flip a bit in a structure the kernel uses to decide who is allowed to do what. A clever attacker can arrange for that bit flip to change which process has root privileges. Now the off-by-one is not a crash; it is a privilege escalation. The unprivileged user becomes root. The system's trust model, which assumed that only authorised users were root, no longer holds.

This is not a hypothetical scaling. It is the standard way kernel bugs are turned into exploits. The kernel's data structures sit near each other in memory. An attacker who can write a single byte somewhere in the kernel can often, with enough cleverness, steer that byte into a structure that matters. A few bytes out of place in the right data structure becomes a working exploit. A few bytes in the right place can make the difference between "my editor crashed" and "the attacker now owns the machine."

This is why the mental framing of security must shift when you move from user-space to the kernel. You are not asking "what is the worst that can happen if this code goes wrong?" You are asking "what is the worst thing that someone could do to the system if they could steer this code to go wrong in exactly the way they wanted?" Those are different questions, and the second one is always the right one to ask inside the kernel.

31.4.3 A Partial Catalogue of What Is at Risk

It helps to make the abstract concrete. If a driver has a bug that can be triggered from user space, what specifically is at risk? The list is long. Here are the major categories, as a way of fixing the stakes in your mind before the chapter turns to specific classes of bug.

Privilege escalation. An unprivileged user obtains root privileges, or a jailed user obtains host-level privileges, or a user inside a capability-mode sandbox obtains privileges outside that sandbox.

Arbitrary kernel memory read. An attacker reads kernel memory they should not see. This includes cryptographic keys, password hashes, other users' file contents that happen to be in the page cache, and the kernel's own data structures that reveal where other interesting memory lives.

Arbitrary kernel memory write. An attacker writes to kernel memory they should not write. This is often the foundation for privilege escalation, because it can be used to modify credential structures, function pointers, or other security-critical state.

Denial of service. An attacker causes the kernel to panic, to hang, or to consume so many resources that the system is no longer useful. Drivers that can be made to loop indefinitely, to allocate unbounded memory, or to hit a KASSERT from user input are all sources of DoS.

Information leak. An attacker learns something they should not know: a kernel pointer (which defeats KASLR), the contents of an uninitialised buffer (which may contain previous callers' data), or metadata about other processes or devices on the system.

Persistence. An attacker installs code that survives reboots, typically by writing to a file the kernel will re-load at boot, or by corrupting a configuration structure.

Sandbox escape. An attacker who is confined to a jail, to a VM guest, or to a Capsicum capability-mode sandbox escapes their confinement by way of a driver bug.

Each of these is a plausible consequence of a single, plausible mistake in a driver. The mistake is often something that looked harmless to the author: a forgotten length check, a structure that was not zeroed before being copied out, a race between two paths that looked to be mutually exclusive. This chapter's goal is to help you see those mistakes before they become any of the items on the list.

31.4.4 Real-World Incidents

Every major kernel has a history of driver-based security incidents. FreeBSD is no exception. Without turning this into a vulnerability archaeology exercise, it is worth naming a few types of incident that are particularly instructive.

There is the classical **ioctl without privilege check**, in which a driver exposes an ioctl that does something a non-root user should not be able to do, but forgets to call `priv_check(9)` or equivalent before doing it. The fix is a one-line addition; the bug can enable arbitrary-code execution as root. This pattern has appeared in multiple kernels over the decades.

There is the **information leak through uninitialised memory**, in which a driver allocates a structure, fills in some fields, and copies the structure to user space. The fields the driver did not fill in contain whatever the allocator happened to return, which may include the last caller's data. Over time, attackers have been able to extract kernel pointers, file contents, and cryptographic keys from this class of bug.

There is the **buffer overflow in a seemingly innocent path**, in which a driver parses a structure from a firmware blob or a USB descriptor without checking the length fields the data claims. An attacker who can control the firmware (for example, by connecting a malicious USB device) can trigger the overflow. This class of bug is particularly pernicious because the attacker can be physical: plug in a USB stick, walk away.

There is the **race between open and read**, in which two threads simultaneously open and read a device, and the driver's state machine has a gap in its synchronisation. The second thread observes a half-initialised state and triggers a crash, or worse, is allowed to proceed and sees data that should have been cleared.

There is the **TOCTOU bug**, in which a driver validates a value in a user-space structure, then later trusts that the value is still the same. Between the check and the use, the user-space program has changed the value, and the driver now operates on data it never validated.

Each of these is preventable. Each has a well-known FreeBSD primitive that prevents it. The chapter is structured around teaching those primitives in the right order.

31.4.5 The Security Mindset

A recurring theme in this chapter is that secure code comes from a particular way of thinking, not from a particular set of techniques. The techniques matter; you need to know them. But the techniques without the mindset produce code that is secure against the specific bugs the author thought of, and insecure against every bug the author did not. The mindset, applied consistently, keeps producing safe code even when the techniques are imperfect.

The mindset has three habits. First, **assume the worst about every input**. Every byte you read from user space, from a device, from firmware, from a bus, from a sysctl, from a loader tunable, might be the worst byte an attacker could have chosen. Not because most inputs are adversarial, but because secure code must work correctly even when they are. Second, **assume the least about the environment**. Do not assume the caller is root just because the test setup made them root; check. Do not assume a field in a structure was zeroed just because the last writer said it was; zero it yourself. Do not assume the system is at securelevel 0; test it. Third, **fail closed rather than open**. When something is wrong, return an error. When something is missing, refuse to proceed. When a check fails, stop. A driver that errs on the side of not working when the rules are unclear is a driver that is hard to abuse; a driver that errs on the side of working anyway is a driver waiting to be exploited.

Those three habits do not need memorisation. They need internalisation. This chapter is a workshop in internalising them.

31.4.6 Even Root Is Not Trusted

A specific point that beginners sometimes miss: even when the caller is root, the driver must still validate the caller's input. This seems counterintuitive. If the caller is root, they can already do anything; what is the point of validating their input?

The point is that “the caller is root” is a statement about authorisation, not about correctness. A root user can ask your driver to do something, and the kernel will say yes. But a root user can also be a buggy program that is passing the wrong length by accident. A root user can be a compromised program that an attacker has taken over. A root user can be running a clumsy script that treats a pointer as a length. In every one of these cases, your driver must still behave sanely.

Concretely, if root passes you a `len` of `0xFFFFFFFF` in an `ioctl` argument, the correct behaviour is to return `EINVAL`, not to happily copy in four gigabytes of user memory into a kernel buffer. Root did not really want that; root was running a program that had a bug. Your driver's job is to prevent the bug from becoming a kernel bug.

This is why input validation is universal. It is not about distrust of the caller; it is about the driver protecting itself and the rest of the kernel from mistakes, whether deliberate or accidental, coming from above.

31.4.7 Where the Boundaries Are

A driver lives between several boundaries. It is worth naming them explicitly, because different classes of bug live at different boundaries, and the defenses differ.

The **user-kernel boundary** separates userland from the kernel. Data crossing from user space into the kernel must be validated; data crossing from kernel to user space must be sanitised. `copyin(9)` and `copyout(9)` are

the primary mechanism for crossing this boundary safely. Sections 3, 4, and 7 of this chapter deal with this boundary.

The **driver-bus boundary** separates the driver from the hardware. Data read from a device is not always trustworthy; a malicious device or a firmware bug can present values the driver did not expect. Length fields in descriptors, for example, must be bounded by the driver's own expectations rather than by the values the device claims. Section 2 touches on this.

The **privilege boundary** separates different levels of authority: root from non-root, the host from a jail, the kernel from a capability-mode sandbox. Privilege checks enforce this boundary. Section 6 deals with it in depth.

The **module-module boundary** separates your driver from other kernel modules. It is the least defended of the boundaries, because the kernel trusts its own modules completely by default. This is one reason the next section talks about the blast radius of a driver bug: it is almost always larger than the driver.

31.4.8 Where This Chapter Fits

Chapters 29 and 30 taught environment in two senses: architectural and operational. Chapter 29 taught you to make the driver portable across buses and architectures. Chapter 30 taught you to make it behave correctly in virtualised and containerised environments. Chapter 31 teaches a third kind of environment, which is policy: the security-relevant choices the administrator makes and the adversary attempts to violate. Taken together, the three chapters describe the environment around a FreeBSD driver at runtime, and what the driver author must do to be a responsible citizen of that environment.

The thread continues. Chapter 32 will turn to Device Tree and embedded development, which may feel like a change of subject but is in fact the same thread carried into new hardware. The security habits you learn here travel with you to every ARM board, every RISC-V system, every embedded target where a driver's privilege and resource discipline matter just as much as they do on a desktop. Later chapters will deepen the debugging story, including some of the techniques this chapter introduces at a high level. The security habits you build now will serve you for the rest of the book, and for the rest of your career as a driver author.

31.4.9 Wrapping Up Section 1

A driver is part of the kernel. Every bug in a driver is a potential kernel bug, and every kernel bug is a potential change to the system's trust model. Because the blast radius is so large, the bar for correctness in a driver is higher than the bar in a user-space program. The chapter's remaining sections walk through the specific classes of bug that matter most for drivers, and the FreeBSD primitives that defend against them.

The single sentence to remember from this section is this: **in a driver, bugs are not just mistakes; they are changes to who can do what on the system**. Keep that framing in mind as you read the rest of the chapter, and every other sentence will be easier to follow.

With the stakes in view, we turn to the first concrete class of bug: buffer overflows and memory corruption inside kernel code.

31.5 Section 2: Avoiding Buffer Overflows and Memory Corruption

Buffer overflows and their cousins, out-of-bounds reads and writes, are the oldest and still among the most common classes of security bug. They appear in user-space code, in kernel code, and in every language that does not enforce bounds at the language level. C is such a language, and kernel C is such a language with sharper edges, so drivers are fertile ground for buffer bugs.

This section explains how these bugs appear in kernel code, why they are often worse than their user-space equivalents, and how to write driver code that avoids them by construction. It assumes that the reader remembers the C material from Chapter 4 and the kernel-C material from Chapter 5 and Chapter 14. If any of that is shaky, a short revisit before reading this section will repay the investment.

31.5.1 A Short Refresher on Buffers

A buffer in C is a region of memory with a specific size. In a driver, buffers come from several places. Stack-allocated buffers are declared as local variables inside functions; they live for the duration of the function call and are cheap to allocate and free. Heap-allocated buffers come from `malloc(9)` or `uma_zalloc(9)`; they live as long as the driver keeps a pointer to them. Statically allocated buffers are declared at file scope; they live for the whole lifetime of the module. Each of these has different properties and different pitfalls.

The one thing all buffers share is a size. Writing past the end of the buffer, or reading before its beginning, or indexing into it with a value that does not fit, is a **buffer overflow** (or underflow). The overflow itself is the mechanism; what the overflow writes and where the writing lands determine the severity.

A stack overflow in a driver is the most dangerous kind, because the stack holds return addresses, saved registers, and local variables for the whole call chain. A write past the end of a stack buffer can reach into the caller's return address, and from there into arbitrary code execution. A heap overflow is less directly exploitable, but heap buffers are often adjacent to other kernel data structures, and a heap overflow that lands on the right structure is a clear path to compromise. A static-buffer overflow is the least common but can still lead to compromise if the static buffer is next to other writeable module data.

The vocabulary of “stack” and “heap” overflow should feel familiar from user-space work. The mechanism is the same. The consequences are worse, because the kernel's code and data share an address space with everything else it does.

31.5.2 How Overflows Happen in Kernel Code

Overflows do not happen because authors write to memory they did not intend to write to. They happen because the author writes to memory they did intend to write to, but the length or offset is wrong. The most common shapes of the mistake are:

Trusting a length from user space. The driver's `ioctl` argument contains a length field, and the driver uses that length to decide how much to copy in or how large a buffer to allocate. If the length is not bounded, the user can pick a length that makes the copy misbehave.

Off-by-one in a loop. A loop that iterates over an array uses `<=` where `<` was intended, or `<` where `<=` was intended. The extra iteration touches memory just past the end of the array.

Incorrect buffer size in a call. A call to `copyin`, `strncpy`, `sprintf`, or similar takes a size argument. The author passes `sizeof(buf)` for a pointer-typed `buf`, which yields the pointer's size (four or eight bytes) rather than the buffer's size. The call writes far too many bytes.

Arithmetic overflow in a length calculation. The author multiplies or adds lengths to compute a buffer size, and the multiplication overflows a 32-bit integer. The resulting "size" is small, the allocation succeeds, and the subsequent copy writes far more than was allocated.

Truncating a string without terminating it. The author uses `strncpy` or similar, but `strncpy` does not guarantee a null terminator; a later string operation reads past the end of the buffer.

Skipping a length check because the code "obviously" cannot reach a bad state. The author convinces themselves that a given path cannot produce a length greater than some bound, so no check is needed. The path can produce such a length, because the author missed a case.

Each of these is a class of bug with countermeasures. The rest of this section walks through the countermeasures.

31.5.3 Bound Everything

The simplest and most effective countermeasure is to bound every length. Before you use a length from an untrusted source, compare it against a known maximum. Before you allocate a buffer whose size comes from an untrusted source, compare the size against a known maximum. Before you copy into a buffer, confirm that the copy size fits.

Concretely, if your `ioctl` handler takes a structure with a `u_int32_t len` field, add a check like this at the very top of the handler:

```
#define SECDEV_MAX_LEN    4096

static int
secdev_ioctl_set_name(struct secdev_softc *sc, struct secdev_ioctl_args *args)
{
    char *kbuf;
    int error;

    if (args->len > SECDEV_MAX_LEN)
        return (EINVAL);

    kbuf = malloc(args->len + 1, M_SECDEV, M_WAITOK | M_ZERO);
    error = copyin(args->data, kbuf, args->len);
    if (error != 0) {
        free(kbuf, M_SECDEV);
        return (error);
    }
    kbuf[args->len] = '\0';

    /* use kbuf */
}
```

```
    free(kbuf, M_SECDEV);  
    return (0);  
}
```

The first line of the function is the bound. No matter what the caller passes, `args->len` is now at most `SECDEV_MAX_LEN`. The allocation is bounded, the copy is bounded, and the null-termination is inside the buffer. This pattern is the workhorse of safe driver code.

What should the bound be? It depends on the semantics of the argument. A name of a device might reasonably be bounded to a few hundred bytes. A configuration blob might be bounded to a few kilobytes. A firmware blob might be bounded to a few megabytes. Pick a number that is generous enough to accommodate legitimate use and small enough that its consequences, if reached, are bearable. If the bound is too small, users will complain about legitimate failures; if it is too large, an attacker can use the bound itself as an amplifier for denial of service. A generous bound is almost always the right choice.

Some drivers derive the bound from the structure of the hardware. A driver for a fixed-size register bank might bound reads and writes to the bank's size. A driver for a ring with 256 entries might bound the index to 255. Bounds derived from hardware structure are particularly robust, because they correspond to a physical constraint rather than an arbitrary choice.

31.5.4 The `sizeof(buf)` Trap

One of the most common buffer-size bugs in C code is the confusion between `sizeof(buf)` and `sizeof(*buf)`, or between `sizeof(buf)` and the length of the memory `buf` points to. The trap appears most often when a buffer is passed into a function.

Consider this unsafe function:

```
static void  
bad_copy(char *dst, const char *src)  
{  
    strcpy(dst, src, sizeof(dst));    /* WRONG */  
}
```

Here, `dst` is a `char *`, so `sizeof(dst)` is the size of a pointer: 4 on 32-bit systems, 8 on 64-bit systems. The call to `strcpy` tells it that the destination is 8 bytes long, regardless of how big the real buffer is. On a 64-bit system, the function writes up to 8 bytes and terminates, and the caller's 4096-byte buffer now contains a short string, which is probably not what anyone wanted. On any system, if the caller's buffer was less than 8 bytes, the call overflows it.

The fix is to pass the buffer size explicitly:

```
static void  
good_copy(char *dst, size_t dstlen, const char *src)  
{
```

```
    strcpy(dst, src, dstlen);  
}
```

The callers then use `sizeof(their_buf)` at the call site, where `their_buf` is known to be the array:

```
char name[64];  
good_copy(name, sizeof(name), user_input);
```

This pattern is so common in FreeBSD that many internal functions follow it: they take a `(buf, bufsize)` pair rather than a bare `buf`. When you write functions that write into a buffer, do the same. Your future self, reading the code six months later, will thank you.

31.5.5 Bounded String Functions

C's traditional string functions, `strcpy`, `strcat`, `sprintf`, were designed in an era when nobody took buffer overflows seriously. They do not take a size argument; they write until they see a null terminator. In kernel code, they are trouble, because the null terminator can be far away or absent entirely.

FreeBSD provides bounded alternatives:

- `strncpy(dst, src, dstsize)`: copy at most `dstsize - 1` bytes plus a null terminator. Returns the length of the source string. Safe to use when you know `dstsize` correctly.
- `strlcat(dst, src, dstsize)`: append `src` to `dst`, ensuring the result is at most `dstsize - 1` bytes plus a null terminator. Like `strncpy`, this is safe when `dstsize` is correct.
- `snprintf(dst, dstsize, fmt, ...)`: format into `dst`, writing at most `dstsize` bytes including the terminator. Returns the number of bytes that would have been written, which may be larger than `dstsize`. Check the return value if you need to know about truncation.

`strncpy` and `strncat` also exist, but they have surprising semantics. `strncpy` pads with nulls if the source is shorter than the destination size, and, more dangerously, it does not null-terminate if the source is longer. `strncat` is confusing in a different way. Prefer `strncpy` and `strlcat` in new code.

For longer formatted output, the `sbuf(9)` API is safer still. It manages an auto-growing buffer with a clean interface for appending strings, printing formatted output, and bounding the final size. It is overkill for small fixed-size copies but excellent for anything that builds up a longer message. Section 8 returns to `sbuf` in the context of logging.

31.5.6 Arithmetic and Overflow

A subtler class of buffer bug comes from arithmetic on sizes. The classic example is:

```
uint32_t total = count * elem_size;           /* may overflow */  
buf = malloc(total, M_SECDV, M_WAITOK);  
copyin(user_buf, buf, total);
```


If `count * elem_size` overflows a 32-bit `uint32_t`, `total` wraps around to a small number. The `malloc` succeeds with that small number. The `copyin` is asked for the same small number of bytes, which makes the allocation-and-copy pair itself safe. But a later piece of the driver may treat `count * elem_size` as if it produced the full amount, and write past the end of the buffer.

The fix is to check for overflow explicitly:

```
#include <sys/limits.h>

if (count == 0 || elem_size == 0)
    return (EINVAL);
if (count > SIZE_MAX / elem_size)
    return (EINVAL);
size_t total = count * elem_size;
```

The division is exact (no rounding) for integer types, and the test `count > SIZE_MAX / elem_size` is equivalent to “would the multiplication overflow `size_t`?” This pattern is well worth memorising. It is one of those idioms that appears unnecessary in the common case and essential in the exceptional case.

On modern compilers, FreeBSD also has `__builtin_mul_overflow` and its siblings, which perform the arithmetic and report overflow in a single operation. They are a little more convenient when you have them, but the explicit division check works everywhere.

31.5.7 Integer Types Matter

Closely related is the choice of integer types for lengths and offsets. If a length is stored as `int`, it can be negative, and a negative value sneaking into a call that expects an unsigned length can cause spectacular misbehaviour. If a length is stored as `short`, it can only represent values up to 32767, and a caller passing a value near that limit can cause truncation.

The safe types for lengths in FreeBSD are `size_t` (unsigned, at least 32 bits, often 64 on 64-bit platforms) and `ssize_t` (signed `size_t`, usually for return values that may be negative to indicate error). Use them consistently. When you take a length as input, convert it to `size_t` at the earliest opportunity. When you store a length, store it as `size_t`. When you pass a length to a FreeBSD primitive, pass a `size_t`.

If the length comes from user space and the user-facing structure uses a `uint32_t`, the conversion on a 64-bit kernel is safe (no truncation), and you should still validate the value before using it. If the user-facing structure uses `int64_t` and the kernel needs a `size_t`, check for negatives and for overflow before the conversion.

31.5.8 Stack Buffers Are Cheap but Limited

A stack buffer is a local array:

```
static int
secdev_read_name(struct secdev_softc *sc, struct uio *uio)
{
```

```
char name[64];
int error;

mtx_lock(&sc->sc_mtx);
strncpy(name, sc->sc_name, sizeof(name));
mtx_unlock(&sc->sc_mtx);

error = uiomove(name, strlen(name), uio);
return (error);
}
```

Stack buffers are allocated automatically, freed automatically when the function returns, and are essentially free to use. They are ideal for small, short-lived data that does not need to outlive the function call.

The limit on stack buffers is the size of the kernel stack itself. FreeBSD's kernel stack is small, typically 16 KiB or 32 KiB depending on the architecture, and that stack must accommodate the whole call chain, including nested calls into the VFS, the scheduler, interrupt handlers, and so on. A driver function that declares a 4 KiB local buffer is already using a quarter of the stack. A driver function that declares a 32 KiB local buffer has almost certainly blown the stack, and the kernel will panic or corrupt memory when it happens.

A safe rule of thumb: keep local buffers under 512 bytes, and preferably under 256 bytes. For anything larger, allocate on the heap. The compiler will not warn you when you declare a stack buffer that is too large; it is the author's responsibility to keep stack usage bounded.

31.5.9 Heap Buffers and Their Lifetimes

A heap buffer is allocated dynamically:

```
char *buf;

buf = malloc(size, M_SECTEV, M_WAITOK | M_ZERO);
/* use buf */
free(buf, M_SECTEV);
```

Heap buffers can be arbitrarily large (up to the available memory), can outlive the function that allocates them, and give the author explicit control over when they are freed. They come at the cost of requiring deliberate attention: every allocation must be paired with a free, every free must happen after the last use, and every free must happen only once.

The rules for heap buffers are:

1. Always check the allocation if you used `M_NOWAIT`. With `M_WAITOK`, the allocation cannot fail; with `M_NOWAIT`, it can return `NULL` and your code must handle that.
2. Pair every `malloc` with exactly one `free`. Not zero, not two.
3. After calling `free`, do not access the buffer. If the pointer may be reused, set it to `NULL` immediately after the free, so that accidental use triggers a null-pointer panic rather than a subtle corruption.

4. If the buffer held sensitive data, zero it with `explicit_bzero` or use `zfree` before freeing.

Section 4 covers these rules in more depth, including the FreeBSD-specific flags on `malloc(9)`.

31.5.10 A Worked Example: Safe and Unsafe Copy Routines

To make the patterns concrete, here is an unsafe copy routine that you might find in a first-draft driver, followed by a safe rewrite. Read the unsafe version carefully and see if you can spot all the bugs before looking at the commentary.

```
/* UNSAFE: do not use */
static int
secdev_bad_copy(struct secdev_softc *sc, struct secdev_ioctl_args *args)
{
    char buf[256];

    copyin(args->data, buf, args->len);
    buf[args->len] = '\0';
    strncpy(sc->sc_name, buf, sizeof(sc->sc_name));
    return (0);
}
```

There are at least four bugs in those four lines.

First, `copyin`'s return value is ignored. If the user supplied a bad pointer, `copyin` returns `EFAULT`, but the function continues as if the copy succeeded. The subsequent operations on `buf` operate on whatever garbage the stack happened to hold.

Second, `args->len` is not bounded. If the user supplies a `len` of 1000, `copyin` writes 1000 bytes into a 256-byte stack buffer. The stack is corrupted. The driver has just become a vehicle for privilege escalation.

Third, `buf[args->len] = '\0'` writes past the end of the buffer even in the non-malicious case. If `args->len == sizeof(buf)`, the assignment is to `buf[256]`, which is one past the end of the 256-byte array.

Fourth, the function returns 0 regardless of whether anything went wrong. A caller receives a success code and has no way to know that the driver silently dropped their input.

Here is a safe rewrite:

```
/* SAFE */
static int
secdev_copy_name(struct secdev_softc *sc, struct secdev_ioctl_args *args)
{
    char buf[256];
    int error;

    if (args->len == 0 || args->len >= sizeof(buf))
        return (EINVAL);
}
```

```
error = copyin(args->data, buf, args->len);
if (error != 0)
    return (error);

buf[args->len] = '\\0';

mtx_lock(&sc->sc_mtx);
strncpy(sc->sc_name, buf, sizeof(sc->sc_name));
mtx_unlock(&sc->sc_mtx);

return (0);
}
```

The bound is now `args->len >= sizeof(buf)`, which ensures that the terminator at `buf[args->len]` fits. The `copyin` return value is checked and propagated. The write to `sc->sc_name` happens under the mutex that protects it, ensuring that another thread reading the field at the same time sees a consistent value. The function returns the error code the caller needs to understand what happened.

The unsafe version is eight lines; the safe version is thirteen. The five extra lines are the difference between a working driver and a security incident.

31.5.11 A Second Worked Example: The Descriptor Length

Here is a different class of bug that shows up in drivers for devices that present descriptor-like data (USB, virtio, PCIe configuration):

```
/* UNSAFE */
static void
parse_descriptor(struct secdev_softc *sc, const uint8_t *buf, size_t buflen)
{
    size_t len = buf[0];
    const uint8_t *payload = &buf[1];

    /* copy the payload */
    memcpy(sc->sc_descriptor, payload, len);
}
```

The length is taken from the first byte of the buffer, which is a value the device (or an attacker impersonating it) can set arbitrarily. If `buf[0]` is 200, the `memcpy` copies 200 bytes, regardless of whether `buf` actually contains 200 bytes of valid data or whether `sc->sc_descriptor` is that large. If `buflen` is less than `buf[0] + 1`, the `memcpy` reads past the end of the caller's buffer. If `sizeof(sc->sc_descriptor)` is less than `buf[0]`, the `memcpy` writes past the end of the destination.

The safe version validates both sides of the copy:

```
/* SAFE */
static int
parse_descriptor(struct secdev_softc *sc, const uint8_t *buf, size_t buflen)
{
    if (buflen < 1)
        return (EINVAL);

    size_t len = buf[0];

    if (len + 1 > buflen)
        return (EINVAL);
    if (len > sizeof(sc->sc_descriptor))
        return (EINVAL);

    memcpy(sc->sc_descriptor, &buf[1], len);
    return (0);
}
```

Three checks, each guarding a different invariant: the buffer has at least one byte, the stated length fits in the buffer, and the stated length fits in the destination. Each check protects against a different adversarial or accidental input.

A careful reader may notice that `len + 1 > buflen` can itself overflow if `len` is `SIZE_MAX`. For a `size_t` taken from a byte, `len` is at most 255, so the overflow cannot happen here; but if you write the same code for a 32-bit length field, the check should be rearranged to `len > buflen - 1` with an explicit `buflen >= 1` check. The habit of watching for arithmetic overflow is the same habit, applied at different scales.

31.5.12 Buffer Overflow as a Class of Bug

Stepping back from the specific examples: buffer overflows are not a single bug. They are a family of bugs whose members share a structure: the code writes to or reads from a buffer with an incorrect size or offset. The concrete examples above show several members of the family, but the underlying pattern is the same: a length came from somewhere less trustworthy than the code believed it was, and the code was not prepared.

The countermeasures also share a structure. They all amount to: do not trust the length; check it against a known bound before you use it; keep the bound tight; propagate errors when the check fails; use bounded primitives (`strncpy`, `snprintf`, `sbuf(9)`) when you have a choice; watch for arithmetic overflow in length calculations; and keep stack buffers small. That short list, consistently applied, eliminates most buffer overflow bugs before they are written.

31.5.13 Memory Corruption Beyond Overflows

Not every memory-corruption bug is a buffer overflow. Drivers can corrupt memory in several other ways, and a complete treatment of safety must mention them.

Use-after-free is writing to, or reading from, a buffer after it has been freed. The allocator has almost certainly handed that memory to some other part of the kernel by now, so the write corrupts whatever that part of the kernel is doing. Section 4 covers use-after-free in depth.

Double-free is calling `free` twice on the same pointer. Depending on the allocator, this can corrupt the allocator's own data structures, leading to hard-to-diagnose panics minutes or hours later. Section 4 covers prevention.

Out-of-bounds read is the read-only cousin of buffer overflow. It does not corrupt memory directly, but it can leak information (see Section 7) and can cause the kernel to read from an unmapped page, which is a panic. It deserves the same countermeasures as overflow.

Type confusion is treating a block of memory as if it had a different type from what it actually has. For example, casting a pointer to the wrong structure type and accessing fields. In kernel C, type confusion is usually caught by the compiler, but it can still happen when a driver deals with void pointers or with structures shared across versions.

Uninitialised memory use is reading from a variable before assigning it a value. The value read is whatever happened to be in memory at that location, which may be previous callers' data. Section 7 covers this from the information-leak perspective.

Each of these has its own countermeasures, but the single most effective tool across all of them is the set of kernel sanitizers FreeBSD provides: `INVARIANTS`, `WITNESS`, `KASAN`, `KMSAN`, and `KCOV`. Section 10 covers these tools in depth. The short version: build your driver against a kernel with `INVARIANTS` and `WITNESS` always. Build it against a `KASAN`-enabled kernel during development. Run your tests under the sanitized kernel. The sanitizers will find bugs you would otherwise not find until a customer did.

31.5.14 How Compiler Protections Help, and Where They Stop

FreeBSD kernels are usually compiled with several exploit-mitigation features enabled in the compiler. Understanding what they do is part of understanding why certain defensive habits matter more than others.

Stack-smashing protection (SSP) inserts a canary value on the stack between local variables and the saved return address. When the function returns, the canary is checked against a reference value; if it has been modified (because a stack-buffer overflow clobbered it), the kernel panics. SSP does not prevent the overflow from happening, but it prevents many overflows from gaining control of execution. Without SSP, overwriting the return address would redirect execution to attacker-controlled code on return. With SSP, the overwrite is detected and the kernel stops.

SSP is heuristic. Not every function gets a canary: functions without stack-allocated buffers, for example, do not need protection. The compiler applies SSP to functions that look risky. A driver author should not assume SSP will catch any particular bug; SSP catches some stack overflows, not all, and catches them only at function return, not at the moment of the overflow.

kASLR is orthogonal to SSP. It randomizes the base address of the kernel, loadable modules, and the stack. An attacker who wants to jump to a specific kernel function (say, to bypass a check) must first learn where that function is. kASLR makes this difficult. An information leak that exposes any kernel pointer can undo kASLR for the whole kernel: once you know one function's address, you know the offsets to all the others, and you can compute every other address.

W^X enforcement ensures that memory is either writable or executable, never both at once. Historically, attackers would overflow a buffer, write shellcode into the overflowed region, and jump to it. W^X breaks this by refusing to execute from writable memory. Modern attacks therefore use return-oriented programming (ROP), which chains together small snippets of existing code rather than introducing new code. ROP is still possible under W^X, but it is harder, and it is defeated by kASLR (ROP needs to know where the snippets are).

Guard pages surround kernel stacks with unmapped pages. A write past the end of the stack hits an unmapped page, causing a page fault that the kernel catches and turns into a panic. This prevents certain stack-smashing attacks from silently corrupting memory adjacent to the stack. The cost is one unusable page per kernel stack, which is cheap.

Shadow stacks and CFI (control-flow integrity) are under discussion and partial deployment in modern kernels. They aim to prevent attackers from redirecting execution by verifying that every indirect jump lands at a legitimate target. They are not yet standard in FreeBSD, but the direction of the industry is clear: more compiler-enforced restrictions on what exploit writers can do.

The lesson for driver authors: these protections are real, and they raise the cost of exploitation. But they do not prevent bugs. A buffer overflow is still a bug, even if SSP catches it. An information leak is still a bug, even if kASLR makes it less useful. The compiler protections are a last line of defense; the first line is still your careful code.

When the first line fails, the protections buy time: time for the bug to be found and fixed before an attacker chains it into a complete exploit. An information leak that, combined with a buffer overflow, would have been trivially exploitable in 1995 now requires both bugs to exist in the same driver and several more mitigations to fall. The effect is that bug reports that once meant “this is a root exploit” now often mean “this is a pre-condition for a root exploit”. That is progress. But it is progress bought by the compiler, not by the code.

31.5.15 Wrapping Up Section 2

Buffer overflows and memory corruption are the oldest security bugs in C code, and they remain the most common way driver code goes wrong. The countermeasures are well understood: bound every length, use bounded primitives when possible, watch for arithmetic overflow, keep stack buffers small, and run under the kernel’s sanitizers during development. None of these are expensive. All of them are non-negotiable for code that lives in the kernel.

The bugs in this section all came from data of the wrong size reaching the wrong buffer. The next section turns to a closely related problem: data of the wrong shape reaching the wrong kernel function. That is the problem of user input, and it is the single largest source of driver bugs in the real world.

31.6 Section 3: Handling User Input Safely

Every driver that exports an `ioctl`, a `read`, a `write`, or a `mmap` entry point is a driver that receives user input. The shape of the input varies, but the principle does not: data from user space must cross the user-kernel boundary, and the crossing is where most driver security bugs happen.

FreeBSD gives drivers a small and well-designed set of primitives for crossing the boundary safely. The primitives are `copyin(9)`, `copyout(9)`, `copyinstr(9)`, and `uiomove(9)`. Used correctly, they make user input

almost impossible to mishandle. Used incorrectly, they turn the boundary into a gaping hole. This section teaches correct use.

31.6.1 The User-Kernel Boundary

Before the primitives, it helps to make the boundary itself vivid.

A user-space program has its own address space. The program's pointers refer to addresses that are meaningful only in that address space. A pointer that points to byte `0x7fff_1234_5678` in the program's memory has no meaning inside the kernel; the kernel's view of user memory is indirect, mediated by the virtual-memory subsystem.

When the program makes an `ioctl` call that includes a pointer (for example, a pointer to a structure the driver should fill in), the kernel does not receive kernel-space access to that memory. The kernel receives a user-space address. Dereferencing it directly from kernel code is not safe: the address may be invalid (the user sent a garbage pointer), it may point to memory that is not currently resident (paged out), it may not be mapped in the current address space at all, or it may be in a region the kernel has no business reading.

Early UNIX kernels were sometimes careless here and dereferenced user pointers directly. The result was a class of bug known as “ptrace-style” attacks, in which a user program could induce the kernel to read or write arbitrary addresses by passing cleverly crafted pointers. Modern kernels, including FreeBSD, never dereference a user pointer directly from kernel code. They always go through a dedicated primitive that validates and handles the access safely.

The primitives themselves are straightforward. Before we look at them, a note on vocabulary: when the manual pages and the kernel say “kernel address,” they mean an address that is meaningful in the kernel's address space. When they say “user address,” they mean an address supplied by a user-space caller, which is meaningful only in that caller's address space. The primitives translate between the two, with appropriate safety checks.

31.6.2 `copyin(9)` and `copyout(9)`

The two primitives at the heart of the user-kernel boundary are `copyin(9)` and `copyout(9)`:

```
int copyin(const void *udaddr, void *kaddr, size_t len);
int copyout(const void *kaddr, void *udaddr, size_t len);
```

`copyin` copies `len` bytes from user address `udaddr` to kernel address `kaddr`. `copyout` copies `len` bytes from kernel address `kaddr` to user address `udaddr`. Both return 0 on success and `EFAULT` if any part of the copy failed, typically because the user address was invalid, the memory was not resident, or the access crossed into memory the caller had no rights to.

The signatures are declared in `/usr/src/sys/sys/systm.h`. Like most kernel primitives, they are short on names and do one thing. The one thing they do, however, is essential. If a driver reads from or writes to user memory by any other means, the driver is almost certainly wrong.

Always check the return value. This is the single most common source of `copyin/copyout` bugs: a driver calls `copyin` and proceeds as if it succeeded, when in fact it might have returned `EFAULT`. If the copy failed, the destination buffer contains whatever was there before (possibly uninitialised), and operating on it is a recipe for

either a crash or an information disclosure. Every call to `copyin` or `copyout` must check the return value and either proceed with success or propagate the error:

```
error = copyin(args->data, kbuf, args->len);
if (error != 0) {
    free(kbuf, M_SECDDEV);
    return (error);
}
```

That pattern appears hundreds of times in the FreeBSD kernel. Learn it and use it at every call site.

Never reuse a pointer after a failed copy. If `copyin` returned `EFAULT`, the buffer may have been partially written. Do not try to “rescue” a partial result; do not assume that the first few bytes are valid. Discard the buffer, zero it if the remains may be sensitive, and return the error.

Always validate lengths before calling. We have seen this in Section 2, but it bears repeating here. The `len` you pass to `copyin` comes from somewhere; if it comes from the caller’s structure, it must be bounded before the call. An unbounded `len` in a `copyin` is one of the most dangerous patterns in a driver.

`copyin` and `copyout` can sleep. These primitives may cause the calling thread to sleep while waiting for a user page to be resident. This means they cannot be called from contexts where sleeping is forbidden: interrupt handlers, spin-mutex critical sections, and the like. If you need to transfer data to or from user space from such a context, defer the work to a different context (typically a taskqueue or a regular process context) and have that context do the copy.

31.6.3 `copyinstr(9)` for Strings

A string from user space is a special case. You do not know how long it is, only that it is null-terminated. You want to copy it, but you do not want to copy beyond the buffer you have prepared, and you need to handle the case where the user-provided string has no terminator within the expected range.

The primitive for this is `copyinstr(9)`:

```
int copyinstr(const void *udaddr, void *kaddr, size_t len, size_t *lencopied);
```

`copyinstr` copies bytes from `udaddr` to `kaddr` until it encounters a null byte or until `len` bytes have been copied, whichever comes first. If `lencopied` is not `NULL`, `*lencopied` is set to the number of bytes copied (including the terminator, if one was found). The return value is 0 on success, `EFAULT` on a fault, and `ENAMETOOLONG` if no terminator was found within `len` bytes.

The key safety rule is: **always pass a bounded `len`**. `copyinstr` without a bound (or with a huge bound) can cause large amounts of kernel memory to be written, and in older kernels could cause the kernel to scan huge amounts of user memory before giving up. In modern FreeBSD the scan itself is bounded by `len`, but you should still pass a tight bound appropriate to the string’s expected size. A path name might reasonably be bounded to `MAXPATHLEN` (which is `PATH_MAX`, currently 1024 on FreeBSD). A device name might be bounded to 64. A command name might be bounded to 32. Pick a bound that fits the use and pass it.

A second safety rule is: **always check the return value**, and treat ENAMETOOLONG as a distinct condition from EFAULT. The former means the user tried to pass a longer string than you were willing to accept, which is plausibly a legitimate mistake. The latter means the user's pointer was invalid, which may or may not be a legitimate mistake. Your driver may want to return a different error to user space depending on which condition occurred.

A third safety rule is: **check the copied length if you care**. The `lencopied` parameter tells you how many bytes were actually written, including the terminator. If your code depends on knowing the exact length, check it. If your buffer is exactly `len` bytes long and `copyinstr` returned 0, the terminator is at `kbuf[lencopied - 1]`, and the string is `lencopied - 1` bytes long.

A safe use of `copyinstr`:

```
static int
secdev_ioctl_set_name(struct secdev_softc *sc,
    struct secdev_ioctl_name *args)
{
    char name[SECDEV_NAME_MAX];
    size_t namelen;
    int error;

    error = copyinstr(args->name, name, sizeof(name), &namelen);
    if (error == ENAMETOOLONG)
        return (EINVAL);
    if (error != 0)
        return (error);

    /* namelen includes the terminator; the string is namelen - 1 bytes */
    KASSERT(namelen > 0, ("copyinstr returned zero-length success"));
    KASSERT(name[namelen - 1] == '\0', ("copyinstr missed terminator"));

    mtx_lock(&sc->sc_mtx);
    strlcpy(sc->sc_name, name, sizeof(sc->sc_name));
    mtx_unlock(&sc->sc_mtx);

    return (0);
}
```

The function takes a fixed-size stack buffer, calls `copyinstr` with a tight bound, handles the two error cases distinctly, asserts the invariants that `copyinstr` promises (`namelen > 0`, terminator at `name[namelen - 1]`), and copies into the `softc` under the lock. This is the canonical pattern.

31.6.4 uiomove(9) for Streams

`read` and `write` entry points do not use `copyin/copyout` directly; they use `uiomove(9)`, which is a wrapper that handles the iteration over a `struct uio` descriptor. A `uio` describes an I/O operation with potentially

multiple buffers (scatter-gather) and tracks how much has been transferred so far.

```
int uiomove(void *cp, int n, struct uio *uio);
```

`uiomove` copies up to `n` bytes between the kernel buffer `cp` and whatever is described by `uio`. If `uio->uio_rw == UIO_READ`, the copy is kernel-to-user; if `UIO_WRITE`, user-to-kernel. The function updates `uio->uio_offset`, `uio->uio_resid`, and `uio->uio_iov` to reflect the bytes transferred.

Like `copyin`, `uiomove` returns 0 on success and an error code on failure. Like `copyin`, it can sleep. Like `copyin`, the caller must check the return value.

A typical read implementation:

```
static int
secdev_read(struct cdev *dev, struct uio *uio, int flag)
{
    struct secdev_softc *sc = dev->si_drv1;
    char buf[128];
    size_t len;
    int error;

    mtx_lock(&sc->sc_mtx);
    len = strlcpy(buf, sc->sc_name, sizeof(buf));
    mtx_unlock(&sc->sc_mtx);

    if (len >= sizeof(buf))
        len = sizeof(buf) - 1;

    if (uio->uio_offset >= len)
        return (0); /* EOF */

    error = uiomove(buf + uio->uio_offset, len - uio->uio_offset, uio);
    return (error);
}
```

This handles the case where the user reads past the end of the data (returning 0 to indicate EOF), bounds the copy to the size of the kernel buffer, and propagates any error from `uiomove`. It is a safe pattern for short, fixed data; longer data typically uses `sbuf(9)` internally and copies out with `sbuf_finish/sbuf_len/uiomove` at the end.

31.6.5 Validate Every Field of Every Structure

When an `ioctl` takes a structure, the driver must validate every field before trusting any of them. A common error is to validate the fields the driver uses immediately and ignore the ones it uses later. The structure lives for the duration of the `ioctl` call, and the driver may end up using fields it did not check.

Concretely, if your `ioctl` takes this structure:

```
struct secdev_config {
    uint32_t version;      /* protocol version */
    uint32_t flags;        /* configuration flags */
    uint32_t len;          /* length of data */
    uint64_t data;         /* user pointer to data blob */
    char name[64];         /* human-readable name */
};
```

validate every field at the top of the handler:

```
static int
secdev_ioctl_config(struct secdev_softc *sc, struct secdev_config *cfg)
{
    if (cfg->version != SECDEV_CONFIG_VERSION_1)
        return (ENOTSUP);

    if ((cfg->flags & ~SECDEV_CONFIG_FLAGS_MASK) != 0)
        return (EINVAL);

    if (cfg->len > SECDEV_CONFIG_MAX_LEN)
        return (EINVAL);

    /* Name must be null-terminated within the field. */
    if (memchr(cfg->name, '\0', sizeof(cfg->name)) == NULL)
        return (EINVAL);

    /* ... proceed to use the structure ... */
    return (0);
}
```

Four invariants, each checked and enforced. The driver now knows that `version`, `flags`, `len`, and `name` are all in the expected range. It can use them without further validation. Without these checks, each use later in the function becomes another potential source of bugs.

An important subtlety: when a structure includes reserved fields or padding, the driver must decide what to do when those fields are non-zero. The safe choice is usually to require them to be zero:

```
if (cfg->reserved1 != 0 || cfg->reserved2 != 0)
    return (EINVAL);
```

This preserves the possibility of using those fields in a future version of the protocol without breaking compatibility: if every current caller passes zero, any future non-zero value is necessarily from a caller that knows about the new version. Without the check, the driver cannot later distinguish old callers (who happened to leave garbage in the reserved fields) from new callers (who are using the field for a new purpose).

31.6.6 Validate Structures That Come In Multiple Parts

Some `ioctl`s take a structure that contains a pointer to another block of data. The outer structure is copied in first; the pointer inside it then needs to be followed with a second copyin. Every field of both structures must be validated.

```
struct secdev_ioctl_args {
    uint32_t version;
    uint32_t len;
    uint64_t data;    /* user pointer to a blob of `len` bytes */
};

static int
secdev_ioctl_something(struct secdev_softc *sc,
    struct secdev_ioctl_args *args)
{
    char *blob;
    int error;

    /* Validate the outer structure. */
    if (args->version != SECDEV_IOCTL_VERSION_1)
        return (ENOTSUP);
    if (args->len > SECDEV_MAX_BLOB)
        return (EINVAL);
    if (args->len == 0)
        return (EINVAL);

    blob = malloc(args->len, M_SECDEV, M_WAITOK | M_ZERO);

    /* Copy the inner blob. */
    error = copyin((const void *)(uintptr_t)args->data, blob, args->len);
    if (error != 0) {
        free(blob, M_SECDEV);
        return (error);
    }

    /* ... now validate the inner blob, whose shape depends on the version ... */

    free(blob, M_SECDEV);
    return (0);
}
```

The `uintptr_t` cast is worth commenting on. The user pointer arrives as a `uint64_t` in the structure, to avoid portability issues between 32-bit and 64-bit userlands. The cast to `uintptr_t` and then to `const void *` converts the integer representation back into a pointer. On a 64-bit kernel, this is a no-op; on a 32-bit kernel, the

high bits of the `uint64_t` must be validated or dropped. FreeBSD runs on both, and 32-bit userland on a 64-bit kernel (via `COMPAT_FREEBSD32`) is a real case. Be explicit about the cast, and document the assumption.

31.6.7 The “freezed” Problem

Some drivers have fields in user-space structures that are pointers, and the driver’s convention is that the user-space memory stays valid until a particular operation completes. This pattern is common in drivers that do DMA directly from user memory.

The pattern is tricky because the user can, in principle, change the memory between the driver’s validation and the driver’s use. Pointer-based DMA is also the wrong idea in modern drivers; safer alternatives include:

- `mmap`, in which the driver maps kernel memory into user space for direct access, with the kernel retaining ownership of the memory and its validity.
- A copy-through-kernel approach, in which the driver always copies in, validates, and operates on the kernel copy.
- The `busdma(9)` framework, which handles user-space buffers correctly when they need to be DMA’d to hardware.

If you find yourself writing code that keeps a user-space pointer around and uses it at a later moment, stop and think. It is almost always the wrong design. Section 5 returns to this issue as a TOCTOU problem.

31.6.8 Kernel Addresses Do Not Leak Into User Pointers

A recurring class of bug is when a driver, trying to communicate a pointer to user space, copies out a kernel address. The user receives a pointer to kernel memory, which is a spectacular information leak (it reveals the kernel’s layout, defeating KASLR) and, if the user can somehow convince the kernel to treat the copied pointer as a user pointer, can become an arbitrary kernel-memory access.

The mistake is usually inadvertent. A common case is a structure that is shared between kernel and user space, and one of its fields is a pointer. If the driver fills in the field with a kernel pointer and then copies the structure to user space, the leak has happened.

The fix is structural: do not share structures between kernel and user that contain pointer fields intended to be used in either space. If a pointer field exists, make it `uint64_t` and treat it as an opaque integer. When the kernel fills in a user-visible pointer-like field, it must pick a value meaningful to user space, not reveal its own internal pointer.

A second class of leak is when a driver copies out a structure that contains uninitialised fields, and one of those fields happens to contain a kernel pointer (for example, because the allocator returned memory that was previously used for something that held a kernel pointer). Section 7 covers this in depth.

31.6.9 `compat32` and Structure Sizes

FreeBSD supports running 32-bit user-space programs on a 64-bit kernel through the `COMPAT_FREEBSD32` machinery. For a driver, this means that the structure the caller passes may be a 32-bit structure, with different layout and size from the 64-bit version. If the driver expects the 64-bit structure and the caller passed the 32-bit one, the fields the driver reads will be at the wrong offsets, and the driver will read garbage.

Handling this is outside the scope of a typical driver; the framework helps by offering `ioctl32` entry points and automatic translation for many common cases. If your driver is used from 32-bit user-space and uses custom structures, consult the `freebsd32(9)` manual page and the `sys/compat/freebsd32` subsystem for guidance. Be aware of the issue, and test your driver from a 32-bit userland in the lab environment.

31.6.10 A Larger Example: A Complete `ioctl` Handler

Combining the patterns in this section, here is what a complete, safe `ioctl` handler looks like for a hypothetical operation:

```
static int
secdev_ioctl(struct cdev *dev, u_long cmd, caddr_t data, int flag,
             struct thread *td)
{
    struct secdev_softc *sc = dev->si_drv1;
    struct secdev_ioctl_args *args;
    char *blob;
    int error;

    switch (cmd) {
    case SECDEV_IOCTL_DO_THING:
        args = (struct secdev_ioctl_args *)data;

        /* 1. Validate every field of the outer structure. */
        if (args->version != SECDEV_IOCTL_VERSION_1)
            return (ENOTSUP);
        if ((args->flags & ~SECDEV_FLAGS_MASK) != 0)
            return (EINVAL);
        if (args->len == 0 || args->len > SECDEV_MAX_BLOB)
            return (EINVAL);

        /* 2. Check that the caller has permission, if required. */
        if ((args->flags & SECDEV_FLAG_PRIVILEGED) != 0) {
            error = priv_check(td, PRIV_DRIVER);
            if (error != 0)
                return (error);
        }

        /* 3. Allocate the kernel-side buffer. */
        blob = malloc(args->len, M_SECDEV, M_WAITOK | M_ZERO);

        /* 4. Copy in the user-space blob. */
        error = copyin((const void *)(uintptr_t)args->data, blob,
                      args->len);
        if (error != 0) {
```

```
        free(blob, M_SECDEV);
        return (error);
    }

    /* 5. Do the work under the softc lock. */
    mtx_lock(&sc->sc_mtx);
    error = secdev_do_thing(sc, blob, args->len);
    mtx_unlock(&sc->sc_mtx);

    /* 6. Zero and free the kernel buffer (it held user data
     * that might be sensitive). */
    explicit_bzero(blob, args->len);
    free(blob, M_SECDEV);

    return (error);

default:
    return (ENOTTY);
}
}
```

Each numbered step is a distinct concern. Each step handles errors locally and propagates them. The allocation is bounded by the validated length; the copy is bounded by the same length; the permission check is explicit; the cleanup is symmetric with the allocation; the final return code communicates success or the specific failure. This is what a safe ioctl handler looks like. It is not short, but every line is there for a reason.

31.6.11 Common Mistakes in User-Input Handling

A short checklist of the patterns to watch for, as a reference you can return to while reviewing your own code:

- `copyin` with a length from the user, without a prior bound check.
- `copyinstr` without an explicit bound.
- Return value of `copyin`, `copyout`, or `copyinstr` ignored.
- Structure fields used before they are validated.
- Pointer field cast from `uint64_t` to `void *` without thinking about 32-bit-userland compatibility.
- String field assumed null-terminated without a `memchr` check.
- Length used in arithmetic before being bounded.
- User-space pointer kept around and used later (TOCTOU territory).
- Kernel data structure (with pointer fields) directly copied out.
- Uninitialised fields copied out to user space.

If a code review turns up any of these, pause the review, fix the pattern, and then continue.

31.6.12 A Detailed Walkthrough: Designing a Safe ioctl from Scratch

The accumulated techniques of this section can feel like a long checklist. To show how they come together in practice, let us design a single ioctl carefully, from the user-space interface down to the kernel implementation.

The problem. Our driver needs an ioctl that lets a user set a configuration parameter consisting of a name string (bounded length), a mode (enum), and an opaque data blob (variable length). It should also return the driver's interpretation of the configuration (for example, the canonicalized form of the name).

Defining the interface. The user-visible structure, defined in a header that will ship with the driver, looks like:

```
#define SECDEV_NAME_MAX    64
#define SECDEV_BLOB_MAX    (16 * 1024)

enum secdev_mode {
    SECDEV_MODE_OFF = 0,
    SECDEV_MODE_ON = 1,
    SECDEV_MODE_AUTO = 2,
};

struct secdev_config {
    char          sc_name[SECDEV_NAME_MAX];
    uint32_t      sc_mode;
    uint32_t      sc_bloblen;
    void          *sc_blob;
    /* output */
    char          sc_canonical[SECDEV_NAME_MAX];
};
```

Notes on the design:

The name is a fixed-size inline buffer, not a pointer. This is deliberate: it avoids a separate copyin for the name and makes the interface simpler. The trade-off is that the buffer is always copied even if the name is short, but for 64 bytes that is negligible.

The mode is `uint32_t` rather than `enum secdev_mode` directly, because struct members that cross the user/kernel boundary should have explicit widths. The kernel validates that the value is one of the known enum values.

The blob uses a separate pointer (`sc_blob`) and a length (`sc_bloblen`). The user sets both, and the kernel uses a second copyin to pull the data. The length is bounded by `SECDEV_BLOB_MAX`, a value we (the driver authors) choose based on what the driver is actually going to do with the data.

The canonical output is another fixed inline buffer. The user-space caller may or may not care about this output, but the kernel always fills it.

The kernel handler. Let us walk through the implementation step by step. The ioctl framework will copy the structure into the kernel for us, so by the time our handler runs, `cfg` points to kernel memory. The `sc_blob` field, however, is still a user-space pointer that we must handle ourselves.

```
static int
secdev_ioctl_config(struct secdev_softc *sc, struct secdev_config *cfg,
    struct thread *td)
{
    char kname[SECDEV_NAME_MAX];
    char canonical[SECDEV_NAME_MAX];
    void *kblob = NULL;
    size_t bloblen;
    uint32_t mode;
    int error;

    /* Step 1: Privilege check. */
    error = priv_check(td, PRIV_DRIVER);
    if (error != 0)
        return (error);

    /* Step 2: Jail check. */
    if (jailed(td->td_ucred))
        return (EPERM);

    /* Step 3: Copy and validate the name. */
    bcopy(cfg->sc_name, kname, sizeof(kname));
    kname[sizeof(kname) - 1] = '\0'; /* ensure NUL termination */
    if (strlen(kname, sizeof(kname)) == 0)
        return (EINVAL);
    if (!secdev_is_valid_name(kname))
        return (EINVAL);

    /* Step 4: Validate the mode. */
    mode = cfg->sc_mode;
    if (mode != SECDEV_MODE_OFF && mode != SECDEV_MODE_ON &&
        mode != SECDEV_MODE_AUTO)
        return (EINVAL);

    /* Step 5: Validate the blob length. */
    bloblen = cfg->sc_bloblen;
    if (bloblen > SECDEV_BLOB_MAX)
        return (EINVAL);

    /* Step 6: Copy in the blob. */
    if (bloblen > 0) {
        kblob = malloc(bloblen, M_SECDEV, M_WAITOK | M_ZERO);
        error = copyin(cfg->sc_blob, kblob, bloblen);
        if (error != 0)
```

```

        goto out;
    }

    /* Step 7: Apply the configuration under the lock. */
    mtx_lock(&sc->sc_mtx);
    if (sc->sc_blob != NULL) {
        explicit_bzero(sc->sc_blob, sc->sc_bloblen);
        free(sc->sc_blob, M_SECDEV);
    }
    sc->sc_blob = kblob;
    sc->sc_bloblen = bloblen;
    kblob = NULL; /* ownership transferred */

    strncpy(sc->sc_name, kname, sizeof(sc->sc_name));
    sc->sc_mode = mode;

    /* Produce the canonical form while still under the lock. */
    secdev_canonicalize(sc->sc_name, canonical, sizeof(canonical));
    mtx_unlock(&sc->sc_mtx);

    /* Step 8: Fill the output fields. */
    bzero(cfg->sc_canonical, sizeof(cfg->sc_canonical));
    strncpy(cfg->sc_canonical, canonical, sizeof(cfg->sc_canonical));
    /* (The ioctl framework handles copyout of cfg itself.) */

out:
    if (kblob != NULL) {
        explicit_bzero(kblob, bloblen);
        free(kblob, M_SECDEV);
    }
    return (error);
}

```

Now review this against the patterns we have discussed.

Privilege check. `priv_check(PRIV_DRIVER)` is the first line of business. No unprivileged caller ever reaches the rest.

Jail check. `jailed()` before any host-affecting work.

Name validation. The name is read from the already-copied-in `cfg`, forced NUL-terminated (defensive, in case the user did not terminate it), and whitelisted through `secdev_is_valid_name` (which presumably refuses non-alphanumeric characters).

Mode validation. An explicit whitelist of known mode values. An unknown value returns `EINVAL` immediately.

Length validation. The blob length is checked against a defined maximum before being used for allocation. Without this check, a user could request a multi-gigabyte allocation.

Allocation with `M_ZERO`. The blob buffer is zeroed so that even if `copyin` fails partway, the contents are deterministic.

Error path cleanup. The `out : label` frees `kblob` if we did not transfer ownership. The `kblob = NULL` after transfer prevents a double-free. Every path through the function reaches `out :` with `kblob` in a consistent state.

Explicit zeroing before free. The old blob (if any) is zeroed before being replaced, on the assumption that it may have contained sensitive data. The new blob on error path is also zeroed for the same reason.

Locking. The `softc` is updated under `sc_mtx`. The canonical form is computed under the lock so the name and canonical match.

Output zeroing. `cfg->sc_canonical` is zeroed before being filled, so padding and any fields the canonicalizer did not set are guaranteed zero.

This function has about forty lines of actual code and roughly a dozen security-relevant decisions. Each decision individually is small; the compound effect is a function that is defensible against nearly every pattern discussed in this chapter. This is what secure driver code looks like in practice: not flashy, not tricky, just careful.

The key insight is that the careful code is the easiest to review, the easiest to maintain, and the one that tends to keep working as the driver evolves. Clever tricks, by contrast, are where bugs hide.

31.6.13 Wrapping Up Section 3

User input is the single largest source of driver security bugs in practice. The primitives FreeBSD provides (`copyin`, `copyout`, `copyinstr`, `uiomove`) are well designed and safe, but they must be used correctly: bounded lengths, checked return values, validated fields, zeroed buffers, and properly sized destinations. A driver that consistently applies these rules at every user-kernel boundary crossing is a driver that is hard to attack from user space.

The next section turns to a closely related subject: memory allocation. The patterns in Sections 2 and 3 assumed that `malloc` and `free` are used safely. Section 4 makes that assumption explicit and shows what “safely” means for the FreeBSD allocator in particular.

31.7 Section 4: Secure Use of Memory Allocation

A driver that validates its inputs carefully but allocates memory carelessly has only done half the job. Memory allocation and deallocation are where a driver’s behaviour in bad weather (denial of service, exhaustion, hostile inputs) is most visible, and where a handful of subtle bugs, use-after-free, double-free, leaks, can turn into full system compromise. This section covers the FreeBSD allocator’s safety model and the idioms that keep a driver from becoming an allocator bug farm.

31.7.1 `malloc(9)` in the Kernel

The primary kernel allocator for general-purpose work is `malloc(9)`. Its declaration in `/usr/src/sys/sys/-malloc.h`:

```
void *malloc(size_t size, struct malloc_type *type, int flags);
void free(void *addr, struct malloc_type *type);
void zfree(void *addr, struct malloc_type *type);
```

Unlike the user-space `malloc`, the kernel version takes two extra arguments. The first, `type`, is a `struct malloc_type` tag that identifies which part of the kernel is using the memory. This allows `vmstat -m` to report, per-subsystem, how much memory each part of the kernel is using. Every driver should declare its own `malloc_type` with `MALLOC_DECLARE` and `MALLOC_DEFINE`, so that its allocations are visible in the accounting.

```
#include <sys/malloc.h>

MALLOC_DECLARE(M_SECDEV);
MALLOC_DEFINE(M_SECDEV, "secdev", "Secure example driver");
```

The first argument, `M_SECDEV`, is the identifier; the second, `"secdev"`, is the short name that appears in `vmstat -m`; the third is a longer description. Use a naming scheme that makes it easy to find the driver's allocations in system output, especially when you are diagnosing a leak.

The `flags` argument controls the allocation's behaviour. Three flags are essential:

- `M_WAITOK`: the allocator may sleep to satisfy the allocation. The call cannot fail; it either returns a valid pointer or the kernel panics (which it does only under very unusual circumstances).
- `M_NOWAIT`: the allocator must not sleep. If memory is not immediately available, the call returns `NULL`. The caller must check and handle the `NULL` case.
- `M_ZERO`: the returned memory is zeroed before being returned. Use this whenever the caller will fill in some but not all of the memory, to avoid leaking garbage.

There are others (`M_USE_RESERVE`, `M_NODUMP`, `M_NOWAIT`, `M_EXEC`), but these three are the ones a driver uses daily.

31.7.2 When to Use `M_WAITOK` and When to Use `M_NOWAIT`

The choice between `M_WAITOK` and `M_NOWAIT` is dictated by context, not preference.

Use `M_WAITOK` when the driver is in a context that can sleep. This is the case in most driver entry points: `open`, `close`, `read`, `write`, `ioctl`, `attach`, `detach`. In these contexts, sleeping is allowed, and the allocator's ability to sleep until memory is available is a significant simplification.

Use `M_NOWAIT` when the driver is in a context that cannot sleep. This is the case in interrupt handlers, inside spin-mutex critical sections, and inside certain callback paths that the kernel specifies as non-sleeping. In these contexts, `M_WAITOK` would trigger a `WITNESS` assertion and a panic. Even if `WITNESS` is not enabled, sleeping in a non-sleeping context can deadlock the system.

The rule of thumb: if you can use `M_WAITOK`, use it. It removes a whole class of error handling (the `NULL` check), and it makes the driver's behaviour more predictable under memory pressure. Only fall back to `M_NOWAIT` when the context forces it.

With `M_NOWAIT`, you must check the return value:

```
buf = malloc(size, M_SECDEV, M_NOWAIT);  
if (buf == NULL)  
    return (ENOMEM);
```

Failure to check is a null-pointer panic waiting to happen. The compiler will not warn you about it.

31.7.3 M_ZERO Is Your Friend

One of the subtlest classes of driver bug is the one where the driver allocates memory, fills in some fields, and then uses or exposes the rest. The “rest” is whatever the allocator happened to return, which in FreeBSD is whatever the allocator’s free list last had there. If that memory held another subsystem’s data before being freed, a driver that fails to clear it may accidentally expose that data (an information leak) or may behave incorrectly because a field it did not set has a non-zero value.

M_ZERO prevents both problems:

```
struct secdev_state *st;  
  
st = malloc(sizeof(*st), M_SECDEV, M_WAITOK | M_ZERO);
```

After this call, every byte of `*st` is zero. The driver can then fill in specific fields and trust that everything else is either zero or set explicitly. This is so important for safety that many FreeBSD driver authors treat M_ZERO as the default, adding it unless there is a specific reason not to.

The exception is large allocations where you are certain you will overwrite every byte before use (for example, a buffer that is immediately filled by `copyin`). In that case, M_ZERO is a small waste, and you can omit it. In all other cases, prefer M_ZERO and accept the small cost.

A particularly important case: **any structure that will be copied to user space must either have been M_ZERO’d at allocation time or have had every byte explicitly set before the copy.** Otherwise the structure may include kernel data that was there before. Section 7 returns to this.

31.7.4 uma_zone for High-Frequency Allocations

For allocations that happen many times per second with a fixed size, FreeBSD offers the UMA zone allocator:

```
uma_zone_t uma_zcreate(const char *name, size_t size, ...);  
void *uma_zalloc(uma_zone_t zone, int flags);  
void uma_zfree(uma_zone_t zone, void *item);
```

UMA zones are significantly faster than `malloc` for repeated small allocations, because they maintain per-CPU caches and avoid the global allocator lock for most operations. Drivers that handle network packets, I/O requests, or other high-frequency events typically use UMA zones instead of `malloc`.

The security properties of UMA zones are similar to those of `malloc`. You still pass `M_WAITOK` or `M_NOWAIT`. You still may pass `M_ZERO` (or you may use `uma_zcreate_arg`'s `uminit/ctor/dtor` arguments to manage initial state). You still must check `NULL` on `M_NOWAIT`.

UMA has one additional security consideration worth knowing: **items returned to a zone are not zeroed by default**. An item freed with `uma_zfree` may retain its previous contents and be handed out to a subsequent `uma_zalloc` with that same content. If the item held sensitive data, the driver must zero it before freeing, or must pass `M_ZERO` on every allocation, or must use the `uminit` constructor machinery to zero on allocation. The safest default is to use `explicit_bzero` on the item before calling `uma_zfree`.

31.7.5 Use-After-Free: What It Is and Why It Matters

A use-after-free bug occurs when a driver frees a pointer and then uses it. The allocator has, by now, almost certainly handed that memory to some other part of the kernel. Writes to the freed pointer corrupt that other part of the kernel; reads from it return whatever is now stored there.

The classic pattern:

```
/* UNSAFE */
static void
secdev_cleanup(struct secdev_softc *sc)
{
    free(sc->sc_buf, M_SECDEV);
    /* sc->sc_buf is now dangling */

    /* later, elsewhere, something calls: */
    secdev_use_buf(sc); /* crash or silent corruption */
}
```

The fix has two parts. First, set the pointer to `NULL` immediately after freeing it, so that any subsequent use is a null-pointer dereference (an immediate, diagnosable crash) rather than a dangling-pointer access (silent corruption):

```
free(sc->sc_buf, M_SECDEV);
sc->sc_buf = NULL;
```

Second, audit the code paths that might still hold references to the freed memory. The `NULL`-assignment prevents crashes at `sc->sc_buf` accesses, but a local variable or a caller's parameter that still holds the old pointer is not protected. The discipline is to free memory only when you are sure no one else holds a pointer to it. Reference counts (`refcount(9)`) are the FreeBSD primitive for this.

A variant of the bug is the **use-after-detach** pattern, in which a driver frees its `softc` during `detach` but an interrupt handler or a callback still runs and accesses the freed `softc`. The fix is to drain all asynchronous activity before freeing in `detach`: cancel outstanding callouts with `callout_drain`, drain taskqueues with `taskqueue_drain`, teardown interrupt handlers with `bus_teardown_intr`, and so on. Once all async paths are quiesced, the free is safe.

31.7.6 Double-Free: What It Is and Why It Matters

A double-free occurs when a driver calls `free` twice on the same pointer. The first `free` hands the memory back to the allocator. The second `free` corrupts the allocator's internal bookkeeping, because it tries to insert the same memory into the free list twice.

FreeBSD's allocator detects many double-frees and panics immediately (especially with `INVARIANTS` enabled). But some double-frees slip past the detection, and the consequences are subtle: a later allocation may return memory that is claimed to be available but is actually still in use somewhere.

The prevention is the same `NULL`-assignment pattern:

```
free(sc->sc_buf, M_SECDEV);  
sc->sc_buf = NULL;
```

`free(NULL, ...)` is defined to be a no-op in FreeBSD (as in most allocators), so a second call with `sc->sc_buf == NULL` does nothing. The `NULL`-assignment turns double-free into a safe no-op.

A related pattern is the **error-path double-free**, in which a function's cleanup logic frees a pointer, and then an outer function also frees the same pointer. The defence is to decide, explicitly, which function owns each allocation, and to have ownership transferred at clear moments. "Who frees this?" is a question that should have a clear answer at every line of the code.

31.7.7 Memory Leaks Are a Security Problem

A memory leak is a piece of memory that is allocated and never freed. In a long-running driver, leaks accumulate. Eventually the kernel runs out of memory, either for the driver's subsystem or for the system as a whole, and bad things happen.

Why is a leak a security problem? Two reasons. First, a leak is a denial-of-service vector: an attacker who can trigger an allocation without a corresponding free can exhaust memory. If the attacker is unprivileged but the driver's `ioctl` allocates memory on each call, the attacker can loop on `ioctl` until the kernel OOM-kills something important. Second, a leak often hides other bugs: the leak's accumulation pressure changes the behaviour of subsequent allocations (more frequent `M_NOWAIT` failures, more unpredictable page cache), which can make racy or allocation-dependent bugs surface in production.

The prevention is discipline in allocation ownership: for every `malloc`, there must be exactly one `free`, reachable on every code path. The FreeBSD `vmstat -m` tool makes leak tracking easier in practice: `vmstat -m | grep secdev` shows, per type, how many allocations are outstanding. A driver with a leak will show a steadily rising number under load; a driver without will show a stable number.

For new drivers, it is worth stress-testing the driver in the lab for leaks: open and close the device a million times in a loop, run the full `ioctl` matrix repeatedly, watch `vmstat -m` for the driver's type, and look for growth. Any sustained growth is a leak. Leaks found in the lab are a thousand times cheaper to fix than leaks found in production.

31.7.8 `explicit_bzero` and `zfree` for Sensitive Data

Some data should not be allowed to linger in memory after the driver is done with it. Cryptographic keys, user passwords, device secrets, anything whose exposure in a memory snapshot would be harmful, must be erased before the memory is freed.

The naive approach is to use `bzero` or `memset(buf, 0, len)` before the `free`. This works, but it has a subtle flaw: the optimiser may remove the `bzero` if it can prove that the memory is not read after. The optimiser's logic is correct as far as language semantics go, but it defeats the security intent.

The correct primitive is `explicit_bzero(9)`:

```
void explicit_bzero(void *buf, size_t len);
```

`explicit_bzero` is declared in `/usr/src/sys/sys/systm.h`. It performs the zeroing and is guaranteed by the compiler not to be optimised away, even if the memory is not read after. Use it for any buffer that holds sensitive data:

```
explicit_bzero(key_buf, sizeof(key_buf));  
free(key_buf, M_SECDDEV);
```

FreeBSD also provides `zfree(9)`, which zeros the memory before freeing:

```
void zfree(void *addr, struct malloc_type *type);
```

`zfree` is convenient: it combines the zero and the free into one call. It first zeros the memory using `explicit_bzero`, then frees it. Use `zfree` when you are about to free a buffer that held sensitive data. Use `explicit_bzero` followed by `free` if you need to zero the buffer without freeing it, or if you are working with memory from a source other than `malloc`.

A common question: what is “sensitive data”? The conservative answer is that any data that came from user space should be treated as sensitive, because you cannot know what it represents to the user. A more pragmatic answer is that data that is obviously a secret (a key, a password hash, a nonce, authentication material) must be zeroed, and data that might reveal information about the user's activities (file contents, network payloads, command text) should be zeroed when the driver is finished with it. When in doubt, zero. The cost is small.

31.7.9 `malloc_type` Tags and Accountability

The `malloc_type` tag on every allocation serves several purposes. It makes allocations visible in `vmstat -m`. It helps with panic debugging, because the tag is recorded in the allocator's metadata. It helps the allocator's own accounting, and in some configurations it enables per-type memory limits.

A driver that uses a single `malloc_type` for all its allocations is easier to audit than a driver that uses many. Create one tag per logical subsystem within the driver, not one per allocation site. For small drivers, a single tag is usually enough.

The declaration pattern:

```

/* At the top of the driver source file: */
MALLOC_DECLARE(M_SECDEV);
MALLOC_DEFINE(M_SECDEV, "secdev", "Secure example driver");

/* Allocations throughout the driver use M_SECDEV: */
buf = malloc(size, M_SECDEV, M_WAITOK | M_ZERO);

```

The `MALLOC_DECLARE` declares the tag for external visibility; the `MALLOC_DEFINE` actually allocates it (and registers it with the accounting system). Both are needed. Do not put `MALLOC_DEFINE` in a header, because the kernel linker will complain about duplicate definitions if multiple object files include the header.

31.7.10 The Lifetime of Softc

The softc is the driver's per-instance state. It is typically allocated during `attach` and freed during `detach`. The softc's lifetime is one of the most important things to get right in a driver.

The allocation usually happens via `device_get_softc`, which returns a pointer to a structure whose size was declared at driver-registration time. This means the softc memory is owned by the bus, not by the driver; the driver does not call `malloc` for it, and the driver does not call `free`. The bus allocates the softc when the driver is bound to the device and frees it when the driver is detached.

But the softc often contains pointers to other memory that the driver did allocate. Those pointers must be freed in `detach`, in the reverse order of their allocation. A typical pattern:

```

static int
secdev_detach(device_t dev)
{
    struct secdev_softc *sc = device_get_softc(dev);

    /* Reverse order of allocation. */

    /* 1. Stop taking new work. */
    destroy_dev(sc->sc_cdev);

    /* 2. Drain async activity. */
    callout_drain(&sc->sc_callout);
    taskqueue_drain(sc->sc_taskqueue, &sc->sc_task);

    /* 3. Free allocated resources. */
    if (sc->sc_blob != NULL) {
        explicit_bzero(sc->sc_blob, sc->sc_bloblen);
        free(sc->sc_blob, M_SECDEV);
        sc->sc_blob = NULL;
    }
}

```

```
/* 4. Destroy synchronization primitives. */
mtx_destroy(&sc->sc_mtx);

/* 5. Release bus resources. */
bus_release_resources(dev, secdev_spec, sc->sc_res);

return (0);
}
```

Each step handles a specific concern. The order matters: destroy the device node before freeing resources the device callbacks depend on; drain async activity before freeing data the async paths might touch; destroy synchronization primitives last.

A slip in any of these orderings is a bug. The wrong order can produce use-after-free or double-free patterns. The lab later in the chapter walks through a detach function that has subtle ordering bugs and asks you to fix them.

31.7.11 A Complete Allocation/Deallocation Pattern

Pulling the patterns together, here is a safe allocation and use sequence:

```
static int
secdev_load_blob(struct secdev_softc *sc, struct secdev_blob_args *args)
{
    char *blob = NULL;
    int error;

    if (args->len == 0 || args->len > SECDEV_MAX_BLOB)
        return (EINVAL);

    blob = malloc(args->len, M_SECDEV, M_WAITOK | M_ZERO);

    error = copyin((const void *)(uintptr_t)args->data, blob, args->len);
    if (error != 0)
        goto done;

    error = secdev_validate_blob(blob, args->len);
    if (error != 0)
        goto done;

    mtx_lock(&sc->sc_mtx);
    if (sc->sc_blob != NULL) {
        /* replace existing */
        explicit_bzero(sc->sc_blob, sc->sc_bloblen);
    }
}
```

```

        free(sc->sc_blob, M_SECDEV);
    }
    sc->sc_blob = blob;
    sc->sc_bloblen = args->len;
    blob = NULL; /* ownership transferred */
    mtx_unlock(&sc->sc_mtx);

done:
    if (blob != NULL) {
        explicit_bzero(blob, args->len);
        free(blob, M_SECDEV);
    }
    return (error);
}

```

The function has a single exit point via the `done` label. The `blob = NULL` after ownership transfer ensures that the cleanup at `done` sees the transfer and does not re-free. The `explicit_bzero` before every `free` zeroes the buffer in case it held sensitive data. The existing `sc->sc_blob`, if present, is zeroed and freed before being replaced, to avoid leaking the old blob's contents.

This pattern (single exit point, ownership transfer, explicit zeroing, checked allocation, checked copyin) appears in variations throughout the FreeBSD kernel. Learn it well.

31.7.12 A Closer Look at UMA Zones

`malloc(9)` is a general-purpose allocator suited to varying sizes. For fixed-size objects that are allocated and freed frequently, the UMA zone allocator is often the better choice. UMA stands for Universal Memory Allocator, and it is declared in `/usr/src/sys/vm/uma.h`.

A UMA zone is created once, at module load, and holds a pool of objects of a fixed size. `uma_zalloc(9)` returns an object from the pool (allocating a fresh one if necessary). `uma_zfree(9)` returns an object to the pool (or frees it back to the kernel if the pool is full). Because allocations come from a pre-configured pool, they are faster than general `malloc` and have better cache locality.

Creating a zone:

```

static uma_zone_t secdev_packet_zone;

static int
secdev_modevent(module_t mod, int event, void *arg)
{
    switch (event) {
    case MOD_LOAD:
        secdev_packet_zone = uma_zcreate("secdev_packet",
            sizeof(struct secdev_packet),

```

```

        NULL,    /* ctor */
        NULL,    /* dtor */
        NULL,    /* init */
        NULL,    /* fini */
        UMA_ALIGN_PTR, 0);
    return (0);

case MOD_UNLOAD:
    uma_zdestroy(secdev_packet_zone);
    return (0);
}
return (EOPNOTSUPP);
}

```

Using a zone:

```

struct secdev_packet *pkt;

pkt = uma_zalloc(secdev_packet_zone, M_WAITOK | M_ZERO);
/* ... use pkt ... */
uma_zfree(secdev_packet_zone, pkt);

```

The security advantages of a UMA zone over `malloc`:

A zone can have a constructor and destructor that initialize or finalize objects. This can guarantee that every object returned to the caller is in a known state.

A zone is named, so `vmstat -z` attributes allocations to it. This helps detect leaks and unusual memory patterns in specific subsystems.

The pool of objects can be drained under memory pressure. A `malloc` allocation is held for its lifetime; a UMA zone object can be returned to the kernel when freed if the pool is above its high-water mark.

The security pitfalls:

An object returned to the zone is not automatically zeroed. If the zone holds objects that may contain sensitive data, either add a destructor that zeros, or zero explicitly before freeing:

```

explicit_bzero(pkt, sizeof(*pkt));
uma_zfree(secdev_packet_zone, pkt);

```

Because UMA reuses objects quickly, an object you just freed may be handed to another caller almost immediately. If the other caller is a different thread in another subsystem, residual data could flow between them. The fix, again, is explicit zeroing.

A destructor function passed to `uma_zcreate` is called when an object is about to be freed back to the kernel (not when it returns to the pool). For zeroing on every free, use `M_ZERO` on `uma_zalloc` (which zeros on allocation, equivalent to `bzero` immediately after) or zero explicitly.

UMA zones are not appropriate for every driver allocation. For one-off or irregular allocations, `malloc(9)` is simpler. For high-frequency fixed-size objects, UMA wins on performance and makes memory accounting easier. Choose based on access pattern.

31.7.13 Reference Counting for Shared Objects

When an object in your driver can be held by multiple contexts (a softc that is referenced by both a callout and user-space file descriptors, for example), reference counting is the canonical tool for lifetime management. The `refcount(9)` family in `/usr/src/sys/sys/refcount.h` provides simple atomic helpers:

```
refcount_init(&obj->refcnt, 1); /* initial reference */
refcount_acquire(&obj->refcnt); /* acquire an additional reference */
if (refcount_release(&obj->refcnt)) {
    /* last reference dropped; caller frees */
    free(obj, M_SECDDEV);
}
```

The invariant is simple: each context that holds a pointer to the object also holds a reference. When it finishes, it releases. Whichever context is last to release is responsible for freeing.

Used correctly, refcounts prevent the classic “who frees it” ambiguity. Used incorrectly (unbalanced acquires and releases), they produce leaks or use-after-frees. The discipline is:

Every path that obtains a pointer to the object acquires a reference.

Every path that releases the pointer calls `refcount_release` and checks the return value.

A single “owning” reference is held by whoever created the object; the owner is the default releaser.

Even simple refcount usage catches a large class of lifetime bugs. For complex drivers with multiple concurrent contexts, refcounts are indispensable.

31.7.14 Wrapping Up Section 4

The FreeBSD allocator is safe if used correctly. The rules are simple: check `M_NOWAIT` returns, prefer `M_ZERO`, zero sensitive data before freeing, pair every `malloc` with exactly one `free` on every code path, set pointers to `NULL` after freeing, drain async activity before freeing structures those activities touch, and use a per-driver `malloc_type` for accountability. A driver that follows these rules will not have leaks, use-after-frees, or double-frees.

The next section turns to a class of bug that is related but different: races and TOCTOU bugs. These are where two threads or two moments in time interact badly, and where security consequences often hide.

31.8 Section 5: Preventing Race Conditions and TOCTOU Bugs

A race condition happens when the correctness of a driver depends on the relative timing of events it does not control. A TOCTOU bug (Time of Check to Time of Use) is a special kind of race where the driver checks

a condition at one moment, then acts on the same data at a later moment, assuming the condition is still true. In between, something changes. The check is valid. The action is valid. The combination is a bug. From a security standpoint, races and TOCTOU bugs are some of the most dangerous faults a driver can have, because they often allow an attacker to bypass checks that look correct when read in isolation.

Chapter 19 already covered concurrency, locks, and synchronization primitives. The goal there was correctness. This section revisits the same tools through a security lens. We are not asking “will my driver crash”. We are asking “can an attacker arrange timing so that a check I wrote is useless”.

31.8.1 How Races Arise in Drivers

A FreeBSD driver operates in a multi-threaded environment. Several things can be happening at once:

Two different user processes can call `read(2)`, `write(2)`, or `ioctl(2)` on the same device file. If the driver has a single `softc`, both calls run against the same state.

One thread can be running your `ioctl` handler while an interrupt handler for the same device fires on another CPU.

A user thread can be in the middle of your driver while a callout or taskqueue entry scheduled earlier also runs.

The device can be unplugged, causing `detach` to run while any of the above is still in progress.

Any shared data touched by more than one of these contexts, without proper synchronization, is a potential race. The race becomes a security problem when the shared data gates access, validates input, tracks buffer sizes, or holds lifetime information.

31.8.2 The TOCTOU Pattern

The simplest TOCTOU pattern in a driver looks like this:

```
if (sc->sc_initialized) {  
    use(sc->sc_buffer);  
}
```

Read it carefully. Nothing about it is obviously wrong. The driver checks that the buffer is initialized, then uses it. But if another thread can set `sc->sc_initialized` to false and free `sc->sc_buffer` between the check and the use, the use touches freed memory. The attacker does not need to corrupt the flag or the pointer. They only need to arrange timing.

A more subtle TOCTOU happens with user memory:

```
if (args->len > MAX_LEN)  
    return (EINVAL);  
error = copyin(args->data, kbuf, args->len);
```

Look at `args`. If it was already copied in, this is safe. But if `args` still points into user space, a second user thread can change `args->len` between the check and the `copyin`. The check validates the old length. The copy uses the new length. If the new length exceeds `MAX_LEN`, the `copyin` overruns `kbuf`.

The fix is copy-then-check, which we already covered in Section 3. The reason this technique exists is precisely because TOCTOU on user memory is a real attack vector. Always copy user data into kernel space first, then validate, then use.

31.8.3 Real-World Example: Ioctl with a Path

Imagine an ioctl that takes a path and does something with the file:

```
/* UNSAFE */
static int
secdev_open_path(struct secdev_softc *sc, struct secdev_path_arg *args)
{
    struct nameidata nd;
    int error;

    /* Check path length */
    if (strlen(args->path, sizeof(args->path)) >= sizeof(args->path))
        return (ENAMETOOLONG);

    NDINIT(&nd, LOOKUP, 0, UIO_USERSPACE, args->path);
    error = namei(&nd);
    /* ... */
}
```

This has two races. First, `args->path` is still in user space if `args` was not copied in; a user thread can change it between the `strlen` check and `namei`. Second, even if `args` was copied, using `UIO_USERSPACE` tells the VFS layer to read the path from user space, at which point the process can modify it again before VFS reads it. The fix is to copy the path into kernel space with `copyinstr(9)`, validate it as a kernel string, then pass it to VFS with `UIO_SYSSPACE`.

```
/* SAFE */
static int
secdev_open_path(struct secdev_softc *sc, struct secdev_path_arg *args)
{
    struct nameidata nd;
    char kpath[MAXPATHLEN];
    size_t done;
    int error;

    error = copyinstr(args->path, kpath, sizeof(kpath), &done);
    if (error != 0)
        return (error);

    NDINIT(&nd, LOOKUP, 0, UIO_SYSSPACE, kpath);
}
```



```

    error = namei(&nd);
    /* ... */
}

```

The corrected version copies the path into the kernel exactly once, validates it (by virtue of `copyinstr` bounding the length and guaranteeing a NUL terminator), then hands a stable kernel string to the VFS layer. The user process can change `args->path` as often as it likes; we are no longer reading from there.

31.8.4 Shared State and Locking

For races between concurrent in-kernel contexts, the tool is a lock. FreeBSD offers several. The most common in drivers are:

`mtx_t`, a mutex, created with `mtx_init(9)`. Mutexes are fast, short, and must not be held across sleeps. Use them to protect a small critical section.

`sx_t`, a shared-exclusive lock, created with `sx_init(9)`. Shared-exclusive locks can be held across sleeps. Use them when the critical section includes something like `malloc(M_WAITOK)` or a VFS call.

`struct rwlock`, a read-write lock, for the read-mostly case. Multiple readers can hold the lock in shared mode; an exclusive writer excludes all readers.

`struct mtx` paired with condition variables (`cv_init(9)`, `cv_wait(9)`, `cv_signal(9)`) for producer-consumer patterns.

The rules for safe locking are simple and absolute:

Define exactly what data each lock protects. Write it in a comment next to the `softc` field.

Acquire the lock before reading or writing the protected data. Release it afterwards.

Do not hold locks longer than necessary. Long critical sections hurt performance and increase deadlock risk.

Acquire multiple locks in a consistent order across all code paths. Inconsistent ordering leads to deadlock.

Do not sleep while holding a mutex. Convert to an `sx` lock or drop the mutex first.

Do not call into user space (`copyin`, `copyout`) while holding a mutex. Copy first, then lock. Release, then copy back.

31.8.5 A Closer Look: Fixing a Racy Driver

Consider the following minimal handler:

```

/* UNSAFE: races on sc_open */
static int
secdev_open(struct cdev *dev, int oflags, int devtype, struct thread *td)
{
    struct secdev_softc *sc = dev->si_drv1;

```

```

    if (sc->sc_open)
        return (EBUSY);
    sc->sc_open = true;
    return (0);
}

static int
secdev_close(struct cdev *dev, int fflags, int devtype, struct thread *td)
{
    struct secdev_softc *sc = dev->si_drv1;

    sc->sc_open = false;
    return (0);
}

```

The intent is that only one process can have the device open at a time. The bug is that `sc_open` is checked and set without a lock. Two concurrent `open(2)` calls can both read `sc_open == false`, both decide they are the first, and both set it to true. Both succeed. Now two processes share a device that was meant to be exclusive. This is a real-world bug class that has affected real drivers. Fix:

```

/* SAFE */
static int
secdev_open(struct cdev *dev, int oflags, int devtype, struct thread *td)
{
    struct secdev_softc *sc = dev->si_drv1;
    int error = 0;

    mtx_lock(&sc->sc_mtx);
    if (sc->sc_open)
        error = EBUSY;
    else
        sc->sc_open = true;
    mtx_unlock(&sc->sc_mtx);
    return (error);
}

static int
secdev_close(struct cdev *dev, int fflags, int devtype, struct thread *td)
{
    struct secdev_softc *sc = dev->si_drv1;

    mtx_lock(&sc->sc_mtx);
    sc->sc_open = false;
    mtx_unlock(&sc->sc_mtx);
}

```

```
    return (0);  
}
```

Now the read and the write happen inside a single critical section. Only one caller at a time can be inside that section, so the check-then-set sequence is atomic from the perspective of any other caller.

31.8.6 Lifetime Races at Detach

The hardest races in drivers are lifetime races around detach. The device goes away, but a user thread is still inside your ioctl handler, or an interrupt is in flight, or a callout is pending. If detach frees the_softc while one of these references it, you have a use-after-free.

FreeBSD gives you tools to handle this:

callout_drain(9) waits for any scheduled callout to finish before returning. Call it in detach before freeing anything the callout touches.

taskqueue_drain(9) and taskqueue_drain_all(9) wait for pending tasks to complete.

destroy_dev(9) marks a character device as gone and waits for all in-flight threads to leave the device's d_* methods before returning. After destroy_dev, no new threads can enter and no old threads remain.

bus_teardown_intr(9) removes an interrupt handler and waits for any in-flight instance of that handler to complete.

A correct detach function in a driver that has all of these resources looks roughly like:

```
static int  
secdev_detach(device_t dev)  
{  
    struct secdev_softc *sc = device_get_softc(dev);  
  
    /* 1. Prevent new user-space entries. */  
    if (sc->sc_cdev != NULL)  
        destroy_dev(sc->sc_cdev);  
  
    /* 2. Drain asynchronous activity. */  
    callout_drain(&sc->sc_callout);  
    taskqueue_drain_all(sc->sc_taskqueue);  
  
    /* 3. Tear down interrupts (if any). */  
    if (sc->sc_intr_cookie != NULL)  
        bus_teardown_intr(dev, sc->sc_irq, sc->sc_intr_cookie);  
  
    /* 4. Free resources. */  
    /* ... */  
}
```

```
/* 5. Destroy lock last. */
mtx_destroy(&sc->sc_mtx);
return (0);
}
```

The order matters. We first stop accepting new work, then drain all in-flight work, then free the resources that the in-flight work was using. If we freed resources first and drained second, a callout still running could touch freed memory. That is a classic detach-time use-after-free, and it is a security bug, not just a crash.

31.8.7 Atomics and Lock-Free Code

FreeBSD provides atomic operations (`atomic_add_int`, `atomic_cmpset_int`, and so on) in `/usr/src/sys/sys/atomic_common.h` and architecture-specific headers. Atomics are useful for counters, reference counts, and simple flags. They are not a substitute for locks when multiple related fields must change together.

A common beginner mistake is to say “I will use an atomic to avoid a lock”. Sometimes this is correct. Often it leads to a subtly broken data structure because the atomic operation only makes one field safe, while the code really needed two fields updated together.

The safe rule is: if you can express the invariant with a single atomic read or write, an atomic may be appropriate. If the invariant spans multiple fields or a compound condition, use a lock.

31.8.8 Refcounts as a Lifetime Tool

When an object can be referenced from multiple contexts, a refcount helps manage lifetime. `refcount_init`, `refcount_acquire`, and `refcount_release` (declared in `/usr/src/sys/sys/refcount.h`) give you a simple atomic refcount. The last release returns true, at which point the caller is responsible for freeing the object.

Refcounts solve the classic problem where context A and context B both hold a pointer to an object. Either can finish with it first. The one that finishes last frees it. Neither needs to know whether the other is done, because the refcount tracks that for them.

A driver that uses a refcount on its `softc`, or on per-open state, can release that state safely even under concurrent access. The cost is some care at every entry and exit point to balance acquires and releases.

31.8.9 Ordering and Memory Barriers

Modern CPUs reorder memory accesses. A write in your code may become visible to other CPUs in a different order than it was issued. This is usually invisible because locks on FreeBSD include the necessary barriers. When writing lock-free code, you may need explicit barriers (`atomic_thread_fence_acq`, `atomic_thread_fence_rel`, and variants). For almost all driver code, using a lock removes the need to think about barriers. That is another reason to prefer locks over hand-rolled lock-free constructs when you are still learning.

31.8.10 Signal and Sleep Safety

If your driver sleeps waiting for an event, using `msleep(9)`, `cv_wait(9)`, or `sx_sleep(9)`, use the interruptible variant (`msleep(..., PCATCH)`) when the wait is initiated by user space. Otherwise a stuck device can hold a process in an uninterruptible state forever, and a sufficiently patient attacker can use that to exhaust process slots. The interruptible wait lets the process be signalled.

Always check the return value of a sleep. If it returns a non-zero value, the sleep was interrupted (either by a signal or by another condition), and the driver should typically unwind and return to user space. Don't assume the condition is true just because the sleep returned.

31.8.11 Rate Limiting and Resource Exhaustion

A final race-related security concern is resource exhaustion. If an attacker can call your `ioctl` a million times per second, and each call allocates a kilobyte of kernel memory that is not freed until close, they can drive the system out of memory. This is a denial of service attack, and a careful driver defends against it.

The defenses are: cap per-open resource use, cap global resource use, rate-limit expensive operations. FreeBSD provides `eventratecheck(9)` and `ppsratecheck(9)` in `/usr/src/sys/sys/time.h` for rate limiting, and you can build your own counters where needed. The principle is that the cost to call your driver should not be wildly asymmetric: if a single call allocates megabytes of state, either the caller needs a privilege check or the driver needs a hard cap.

31.8.12 Epoch-Based Reclamation: A Lock-Free Reader Idiom

For read-mostly data structures where readers must never block and writers are rare, FreeBSD provides an epoch-based reclamation framework in `/usr/src/sys/sys/epoch.h`. Readers enter an epoch, access the shared data without taking a lock, and exit the epoch. Writers update the data (usually by replacing a pointer) and then wait for all readers currently in an epoch to exit before freeing the old data.

The idiom is useful for driver code that has frequent reads on a hot path and wants to avoid locking overhead there. For example, a network driver that looks up a rule from a routing-table-like structure on every packet may want readers to run lock-free.

```
epoch_enter(secdev_epoch);
rule = atomic_load_ptr(&sc->sc_rules);
/* use rule; must not outlive the epoch */
do_stuff(rule);
epoch_exit(secdev_epoch);
```

A writer replacing the rule set:

```
new_rules = build_new_rules();
old_rules = atomic_load_ptr(&sc->sc_rules);
atomic_store_ptr(&sc->sc_rules, new_rules);
```

```
epoch_wait(secdev_epoch);  
free(old_rules, M_SECDEV);
```

`epoch_wait` blocks until all readers that entered before the store have exited. After it returns, no reader can still be using `old_rules`, so it is safe to free.

The security considerations with epochs are subtle:

A reader inside an epoch may hold a pointer to something that is about to be replaced. The reader must finish using the pointer before exiting the epoch; any use after the exit is a use-after-free.

A reader inside an epoch cannot sleep. The epoch is an asymmetric lock: writers wait on readers, so a reader that sleeps can starve writers indefinitely.

The writer must ensure that the replacement operation is atomic from a reader's perspective. For a single pointer, an atomic store does the job. For more complex updates, two epochs or a read-copy-update sequence may be needed.

Used correctly, epochs give very high performance on read-heavy workloads. Used incorrectly (a reader that sleeps, or a writer that fails to wait), they produce races that are hard to reproduce and hard to diagnose. Beginners should prefer locks until the performance profile justifies the complexity of epoch-based code.

31.8.13 Wrapping Up Section 5

Races and TOCTOU bugs are timing-based bugs. They happen when two contexts touch shared data without coordination, or when a driver checks a condition and acts on it at two different times. The tools to prevent them are straightforward: copy user data into the kernel once and work from the copy; use a lock around every access to shared mutable state; define what each lock protects and hold it across the full check-and-act sequence; drain asynchronous work before freeing the structures it touches; use refcounts for multi-context lifetime management.

None of this is new to concurrency programming. What is new is the mindset: a race in a driver is not merely a correctness problem. It is a security problem, because an attacker can often arrange the timing they need to exploit it. The next section steps back from timing and looks at a different kind of defense: privilege checks, credentials, and access control.

31.9 Section 6: Access Control and Privilege Enforcement

Not every operation a driver exposes should be available to every user. Reading a temperature sensor may be fine for everyone. Reprogramming the device's firmware should require privilege. Writing raw bytes to a storage controller should probably require more than that. This section is about how a FreeBSD driver decides whether the caller is allowed to do what they are asking, using the kernel's credential and privilege machinery.

The tools are `struct ucred`, `priv_check(9)` and `priv_check_cred(9)`, jail-aware checks, `securelevel` checks, and the broader MAC and Capsicum frameworks.

31.9.1 The Caller's Credential: struct ucred

Every thread running in the FreeBSD kernel carries a credential, a pointer to a `struct ucred`. The credential records who the thread is running as, which jail they are confined to, which groups they belong to, and other security attributes. From inside a driver, the credential is almost always reached via `td->td_ucred`, where `td` is the `struct thread *` passed to your entry point.

The structure is declared in `/usr/src/sys/sys/ucred.h`. The fields most relevant to drivers are:

`cr_uid`, the effective user ID. Usually what you check to answer “is this root”.

`cr_ruid`, the real user ID.

`cr_gid`, the effective group ID.

`cr_prison`, a pointer to the jail the process is in. All processes have one. Unjailed processes belong to `prison0`.

`cr_flags`, a small set of flags including `CRED_FLAG_CAPMODE`, which indicates capability mode (Capsicum).

Do not check `cr_uid == 0` as your privilege gate. That is a common mistake and it is almost always wrong. The correct gate is `priv_check(9)`, which handles jails, `securelevel`, and MAC policies correctly. Checking `cr_uid` manually bypasses all of that and gives root inside a jail the same power as root on the host, which is not what jails are for.

31.9.2 priv_check and priv_check_cred

The canonical primitive for “may the caller do this privileged thing” is `priv_check(9)`. Its prototype, from `/usr/src/sys/sys/priv.h`:

```
int priv_check(struct thread *td, int priv);
int priv_check_cred(struct ucred *cred, int priv);
```

`priv_check` operates on the current thread. `priv_check_cred` operates on an arbitrary credential; you use it when the credential to check is not the running thread's, for example when validating an operation on behalf of a file that was opened earlier.

Both return 0 if the privilege is granted and an `errno` (typically `EPERM`) if not. The driver's pattern is almost always:

```
error = priv_check(td, PRIV_DRIVER);
if (error != 0)
    return (error);
```

The `priv` argument selects one of several dozen named privileges. The full list lives in `/usr/src/sys/sys/priv.h` and covers areas like filesystem, networking, virtualization, and drivers. For most device drivers, the relevant names are:

`PRIV_DRIVER`, the generic driver privilege. Grants access to operations restricted to administrators.

`PRIV_IO`, raw I/O to hardware. More restrictive than `PRIV_DRIVER`, appropriate for operations that bypass the driver's usual abstractions and talk directly to hardware.

`PRIV_KLD_LOAD`, used by the module loader. You will not typically use this from a driver.

`PRIV_NET_*`, used by network-related operations.

Several dozen more. Read the list in `priv.h` and pick the most specific match for the operation being gated. `PRIV_DRIVER` is a reasonable default when nothing more specific fits.

A real-world example from the kernel: in `/usr/src/sys/dev/mmc/mmc.c`, the driver checks `priv_check(td, PRIV_DRIVER)` before allowing certain ioctls that would let a user reprogram the storage controller. In `/usr/src/sys/dev/syscons/syscons.c`, the console driver checks `priv_check(td, PRIV_IO)` before allowing operations that manipulate the hardware directly, since those bypass the normal tty abstraction.

31.9.3 Jail Awareness

FreeBSD jails (`jail(8)` and `jail(9)`) partition the system into compartments. Processes inside a jail share the host's kernel but have a restricted view of the system: their own hostname, their own network visibility, their own filesystem root, and reduced privileges. Inside a jail, `priv_check` refuses many privileges that would otherwise be granted to root. This is one of the main reasons to use `priv_check` instead of checking `cr_uid == 0`.

Some operations, however, make no sense inside a jail at all. Reprogramming device firmware, for example, is a host operation. A jailed root user should never be able to do it. For these, add an explicit jail check:

```
if (jailed(td->td_ucred))
    return (EPERM);
error = priv_check(td, PRIV_DRIVER);
if (error != 0)
    return (error);
```

The `jailed()` macro, defined in `/usr/src/sys/sys/jail.h`, returns true if the credential's prison is anything other than `prison0`. For operations that should never be performed from within a jail, check this first.

For operations that should be allowed inside a jail but with restrictions, use the jail's own fields. `cred->cr_prison->pr_flags` carries per-jail flags; the jail framework also has helpers for checking whether certain capabilities are allowed in the jail. In most driver work you will not go beyond the simple `jailed()` check.

31.9.4 Securelevel

FreeBSD supports a systemwide setting called `securelevel`. At `securelevel 0`, the system behaves normally. At higher `securelevels`, certain operations are restricted even for root: raw disk writes may be disabled, the system time cannot be set backwards, kernel modules cannot be unloaded, and so on. The rationale is that on a well-secured server, raising the `securelevel` at boot means an attacker who later gains root cannot disable logging, install a rootkit module, or rewrite core system files.

For drivers, the relevant helpers are declared in `/usr/src/sys/sys/priv.h`:


```
int securelevel_gt(struct ucred *cr, int level);
int securelevel_ge(struct ucred *cr, int level);
```

Their return values are counterintuitive and worth studying carefully. They return 0 when the securelevel is **not** above or at the threshold (that is, the operation is allowed), and EPERM when the securelevel **is** above or at the threshold (the operation should be denied). In other words, the return value is ready to be used directly as an error code.

The usage pattern for a driver that should refuse to modify hardware at securelevel 1 or higher is:

```
error = securelevel_gt(td->td_ucred, 0);
if (error != 0)
    return (error);
```

Read carefully: this says “return an error if the securelevel is greater than 0”. When securelevel is 0 (normal), `securelevel_gt(cred, 0)` returns 0 and the check passes. When securelevel is 1 or higher, it returns EPERM and the operation is refused.

Most drivers do not need securelevel checks. They make sense for operations that are potentially system-destabilizing: reprogramming firmware, writing to raw disk sectors, lowering the system clock, and so on.

31.9.5 Layering Checks

A driver that wants to be defense-in-depth can layer these checks:

```
static int
secdev_reset_hardware(struct secdev_softc *sc, struct thread *td)
{
    int error;

    /* Not inside a jail. */
    if (jailed(td->td_ucred))
        return (EPERM);

    /* Not at elevated securelevel. */
    error = securelevel_gt(td->td_ucred, 0);
    if (error != 0)
        return (error);

    /* Must have driver privilege. */
    error = priv_check(td, PRIV_DRIVER);
    if (error != 0)
        return (error);
```

```
/* Okay, do the dangerous thing. */
return (secdev_do_reset(sc));
}
```

Each check answers a different question. `jailed()` asks whether we are in the right security domain. `securelevel_gt` asks whether the system administrator has told the kernel to refuse this kind of operation. `priv_check` asks whether this particular thread has the appropriate privilege.

In many drivers, only the `priv_check` is strictly necessary, because it handles jails and `securelevel` through the MAC framework and the privilege definitions themselves. The explicit `jailed()` and `securelevel_gt` calls are appropriate for operations with known host-wide consequences. When in doubt, start with `priv_check(td, PRIV_DRIVER)` and add more layers only when you can explain what each additional check buys.

31.9.6 The Credential on Open, Ioctl, and Other Paths

When designing privilege checks, think about where in the driver's lifecycle they live. There are two main places:

At open time. If only privileged users should be able to open the device, check privileges in `d_open`. This is simplest and gives per-open enforcement: once a user has opened the device, they are free to do what that device allows. This is the model used, for example, by `/dev/mem`, which is openable only with appropriate privilege.

At operation time. If the device supports multiple operations with different privilege requirements, check each operation independently. A storage controller might allow reading device status to any user, reading SMART data to the owner of the device file, and triggering firmware update only to users with `PRIV_DRIVER`. Each operation has its own gate.

A driver can combine both: a privilege check on open to keep unprivileged users out entirely, and additional checks on specific `ioctls` for operations that need more.

An open-time check is easy to implement:

```
static int
secdev_open(struct cdev *dev, int oflags, int devtype, struct thread *td)
{
    int error;

    error = priv_check(td, PRIV_DRIVER);
    if (error != 0)
        return (error);

    /* ... rest of open logic ... */
    return (0);
}
```

An `ioctl`-time check follows the same pattern; the `struct thread *td` argument is available in every entry point.

31.9.7 Device File Permissions

Independent of in-driver privilege checks, FreeBSD also applies the usual UNIX permission model to device files themselves. When your driver calls `make_dev_s` or `make_dev_credf` to create a device node, you choose an owner, group, and mode. Those apply at the filesystem level: a user who fails the permission check on the device node never reaches your `d_open`.

The `make_dev_args` structure, declared in `/usr/src/sys/sys/conf.h`, includes `mda_uid`, `mda_gid`, and `mda_mode` fields. The pattern is:

```
struct make_dev_args args;

make_dev_args_init(&args);
args.mda_devsw = &secdev_cdevsw;
args.mda_uid = UID_ROOT;
args.mda_gid = GID_OPERATOR;
args.mda_mode = 0640;
args.mda_si_drv1 = sc;
error = make_dev_s(&args, &sc->sc_cdev, "secdev");
```

`UID_ROOT` and `GID_OPERATOR` are conventional symbolic names. The mode `0640` means owner can read and write, group can read, others have no access. Choose these thoughtfully. A device that could expose sensitive data or cause hardware damage should not be world-readable or world-writable.

The usual pattern for a privileged device is mode `0600` (root-only) or `0660` (root and a specific group, often operator or wheel). Mode `0640` is common for devices readable by a trusted group for monitoring purposes. Modes like `0666` (world-writable) are almost never appropriate, even for simple pseudo-devices, unless the device really does nothing that should be restricted.

31.9.8 Devfs Rules

Even if your driver creates the device node with a conservative mode, the system administrator can change that through devfs rules. A devfs rule can relax or restrict permissions based on device name, jail, and other criteria. Your driver should not assume the mode it set at creation is the mode the device will have at runtime; it should continue to apply its in-kernel checks regardless. The filesystem mode and the in-kernel `priv_check` defend different attackers; keep both.

31.9.9 The MAC Framework

The FreeBSD Mandatory Access Control framework, declared in `/usr/src/sys/security/mac/`, lets policy modules hook into the kernel and make access decisions based on richer labels than UNIX permissions. A MAC policy can, for example, restrict which users can access which devices even if UNIX permissions allow it, or log every use of a sensitive operation.

For driver authors, the point is this: `priv_check` already consults the MAC framework. When you use `priv_check`, you are opting into whatever MAC policies the administrator has configured. If you bypass

`priv_check` and roll your own privilege check using `cr_uid`, you bypass MAC as well. That is one more reason to always use `priv_check`.

Writing your own MAC policy module is beyond the scope of this chapter; the MAC framework is a substantial subject and has its own documentation. The key takeaway is simply that MAC exists, `priv_check` honors it, and you should not fight it.

A brief note on MAC policies shipped with FreeBSD. The base system includes several MAC policies as loadable modules: `mac_bsdextended(4)` for file-system rule lists, `mac_portacl(4)` for network-port access control, `mac_biba(4)` for Biba integrity policy, `mac_mls(4)` for Multi-Level Security labels, and `mac_partition(4)` for partitioning processes into isolated groups. None of these need to be understood in detail by a driver author; the key point is that your driver, by using `priv_check`, gets their policy decisions for free. An administrator who enables `mac_bsdextended` gets additional filesystem-level restrictions; your driver does not need to know.

MAC and the device node. When you create a device with `make_dev_s`, the MAC framework may assign a label to the device node. Policies consult that label when access is attempted. A driver does not interact with labels directly; the framework handles it. But understanding that a label exists explains why, on a MAC-enabled system, access to your device may be refused even when UNIX permissions allow it. That is not a bug; it is MAC doing its job.

31.9.10 Capsicum and Capability Mode

Capsicum, declared in `/usr/src/sys/sys/capsicum.h`, is a capability system bolted onto FreeBSD. A process in capability mode has lost access to most global namespaces (no new file opens, no network with side effects, no arbitrary `ioctl`, and so on). It can only operate on file descriptors it already holds, and those file descriptors may themselves have limited rights (read only, write only, certain `ioctls` only, and so on).

Capsicum was introduced to FreeBSD through the work of Robert Watson and collaborators. It sits alongside the traditional UNIX permission model and adds a second, more granular layer. Where UNIX permissions ask “can this user access this resource by name”, Capsicum asks “does this process have a capability for this specific object”. The two layers work together: the user must have UNIX permission to open the file in the first place, but once the file descriptor exists, Capsicum can further restrict what the holder of the descriptor can do with it.

For a driver, the main Capsicum concern is: some of your `ioctls` may be inappropriate for a process in capability mode. The helper `IN_CAPABILITY_MODE(td)`, defined in `capsicum.h`, tells you whether the calling thread is in capability mode. A driver can check it and refuse operations that are unsafe:

```
if (IN_CAPABILITY_MODE(td))  
    return (ECAPMODE);
```

This is appropriate for operations with global side effects that a capability-mode process should not have access to. Examples might be an `ioctl` that reconfigures the global driver state, an `ioctl` that affects other processes or other file descriptors, or an `ioctl` that performs an operation that requires querying the global filesystem namespace. If your driver’s `ioctl` needs to touch something that is not already named by the file descriptor it was called on, a capability-mode check is appropriate.

For most driver operations, however, the Capsicum story is simpler: the process that holds the file descriptor was granted the rights it needed when the descriptor was given to it. The driver does not need to re-check those rights; the file-descriptor layer already did. Just make sure your driver supports the normal cap-rights flow (it almost certainly does by default) and consider which individual ioctls should be marked with `CAP_IOCTL_*` rights at the VFS layer.

Cap rights at ioctl granularity. FreeBSD allows a file descriptor to be restricted to a specific subset of ioctls via `cap_ioctls_limit(2)`. For example, a process can hold a file descriptor that allows `FIOASYNC` and `FIONBIO` but no other ioctls. The restriction is enforced by the VFS layer, not by your driver, but the set of ioctls you expose is what defines what can be selected for restriction. A driver that implements only meaningful, well-documented ioctls makes it easier for consumers to apply sensible cap-ioctl restrictions.

Examining Capsicum usage in the tree. For real-world examples of Capsicum-aware code, look at `/usr/src/sys/net/if_tuntap.c` alongside the core capability files under `/usr/src/sys/kern/sys_capability.c`. Most individual drivers rely on the VFS layer to enforce caprights, and only add an explicit `IN_CAPABILITY_MODE(td)` check on the handful of operations with global side effects. The pattern is consistent: preserve the normal behavior, add an `IN_CAPABILITY_MODE` check where operations would be unsafe, and document which ioctls are sandbox-safe.

31.9.11 Sysctls With Security Flags

Many drivers expose tunables and statistics through sysctls. A sysctl that exposes sensitive information, or that can be set to change driver behaviour, should use appropriate flags. From `/usr/src/sys/sys/sysctl.h`:

`CTLFLAG_SECURE` (value `0x08000000`) asks the sysctl framework to consult `priv_check(PRIV_SYSCTL_SECURE)` before allowing the operation. It is useful for sysctls that should not be changed at elevated securelevel.

`CTLFLAG_PRISON` allows the sysctl to be visible and writable from inside a jail (rarely wanted for drivers).

`CTLFLAG_CAPRD` and `CTLFLAG_CAPWR` allow the sysctl to be read or written from capability mode. By default, sysctls are inaccessible in capability mode.

`CTLFLAG_TUN` makes the sysctl settable as a loader tunable (from `/boot/loader.conf`).

`CTLFLAG_RD` vs `CTLFLAG_RW` determines read-only vs read-write access; prefer `CTLFLAG_RD` for anything that exposes state, and be deliberate about what you make writable.

A sysctl that exposes a driver-internal buffer for debugging should typically be `CTLFLAG_RD | CTLFLAG_SECURE` at minimum, and possibly not exist at all in production builds.

31.9.12 A Complete Privilege-Gated Ioctl

Putting the pieces together, here is what a privilege-gated ioctl looks like, end to end:

```
static int
secdev_ioctl(struct cdev *dev, u_long cmd, caddr_t data, int fflag,
             struct thread *td)
{
    struct secdev_softc *sc = dev->si_drv1;
```

```
int error;

switch (cmd) {
case SECDEV_GET_STATUS:
    /* Anyone with the device open can do this. */
    error = secdev_get_status(sc, (struct secdev_status *)data);
    break;

case SECDEV_RESET:
    /* Resetting is privileged, jail-restricted, and securelevel-sensitive. */
    if (jailed(td->td_ucred)) {
        error = EPERM;
        break;
    }
    error = securelevel_gt(td->td_ucred, 0);
    if (error != 0)
        break;
    error = priv_check(td, PRIV_DRIVER);
    if (error != 0)
        break;
    error = secdev_do_reset(sc);
    break;

default:
    error = ENOTTY;
    break;
}

return (error);
}
```

Different commands get different gates. The status command is unprivileged, since it just reads state. The reset command is the danger case, and it goes through the full layered check.

31.9.13 Wrapping Up Section 6

Access control in a FreeBSD driver is a collaboration between several layers. Filesystem permissions on the device node decide who can open it. The `priv_check(9)` family of functions decides whether a thread may perform a given privileged operation. Jail checks decide whether the operation makes sense in the caller's security domain. Securelevel checks decide whether the system administrator has allowed this class of operation at all. The MAC framework lets policy modules add their own opinions on top. Capsicum rights limit what a capability-confined process can do.

The correct use of these tools comes down to a short list of rules: check the caller's credentials at the right points, prefer `priv_check` over ad-hoc UID checks, add `jailed()` and `securelevel_gt` when the operation

has host-wide consequences, pick the most specific `PRIV_*` constant that fits the operation, and set conservative device-file modes in `make_dev_s`.

The next section looks at a different kind of leak: not a privilege escape, but an information escape. Even operations that are properly gated can inadvertently reveal kernel memory contents if they are not written carefully.

31.10 Section 7: Protecting Against Information Leaks

An information leak happens when kernel memory that should not be visible to user space is nonetheless copied out to user space. The classic form is returning the contents of a structure to the user without first initializing the structure. Any padding bytes between fields, or trailing bytes after the last field, contain whatever was on the kernel stack or in a freshly allocated page the last time that memory was used. That could be a password, a pointer that would help defeat ASLR, an encryption key, or anything else.

Information leaks are sometimes dismissed as “not that bad”. They are. In modern attack chains, an information leak is often the first step: it defeats the kernel’s address-space-layout-randomization (kASLR) and makes other exploits reliable. A bug class that starts with “just leaks a few bytes” often ends with “attacker gains kernel code execution”.

31.10.1 How Information Leaks Happen

There are three main ways a driver leaks information:

Uninitialized structure fields copied to user space. A structure has N defined fields plus padding and alignment slots. The code fills in the N fields and calls `copyout`. The padding goes along for the ride, carrying whatever uninitialized stack memory happened to be there.

Partially initialized buffers. The driver allocates a buffer, fills in some of it, and copies the whole buffer to user space. The uninitialized tail carries heap contents.

Oversized replies. The driver is asked for N bytes, but returns a buffer of size $M > N$. The extra $M - N$ bytes contain whatever was in the tail of the source buffer.

Reading beyond a NUL. For string data, the driver copies a buffer up to its allocated size instead of up to the NUL terminator. The bytes after the NUL can carry any data that happened to be in that buffer earlier.

Each of these is easy to create by accident and easy to prevent once you know the pattern.

31.10.2 The Padding Problem

Consider this structure:

```
struct secdev_info {
    uint32_t version;
    uint64_t flags;
    uint16_t id;
    char name[32];
};
```

On a 64-bit system, the compiler inserts padding to align `flags` to 8 bytes. Between `version` (4 bytes) and `flags` (8 bytes), there are 4 bytes of padding. After `id` (2 bytes) and before `name` (1-byte alignment), there are 6 more bytes of padding at the end if the structure is sized up to a multiple of 8.

If your code does:

```
struct secdev_info info;

info.version = 1;
info.flags = 0x12345678;
info.id = 42;
strncpy(info.name, "secdev0", sizeof(info.name));

error = copyout(&info, args->buf, sizeof(info));
```

then the padding bytes, which you never set, go out to user space. They contain whatever stack memory happened to be at those positions when the function was entered. That is an information leak.

The fix is universal and cheap: zero the structure first.

```
struct secdev_info info;

bzero(&info, sizeof(info));    /* or memset(&info, 0, sizeof(info)) */
info.version = 1;
info.flags = 0x12345678;
info.id = 42;
strncpy(info.name, "secdev0", sizeof(info.name));

error = copyout(&info, args->buf, sizeof(info));
```

Now the padding is zero, as is any field you forgot to set. The cost is one call to `bzero`; the benefit is that your driver cannot leak kernel memory through this structure, no matter what fields are added later. Always zero structures before copyout.

An equivalent pattern using designated initializers works when you are declaring and initializing in one step:

```
struct secdev_info info = { 0 }; /* or { } in some standards */
info.version = 1;
/* ... */
```

The `= { 0 }` zeros all bytes including padding. Combine this with setting the specific fields afterwards, and you have a clean pattern.

31.10.3 The Heap Allocation Case

When you allocate a buffer with `malloc(9)` and fill it before returning to user space, you have the same issue. Always use `M_ZERO` to zero-initialize, or explicitly zero the buffer before writing to it:


```
buf = malloc(size, M_SECTDEV, M_WAITOK | M_ZERO);
```

Even if you intend to fill every byte, using `M_ZERO` is cheap insurance: if a bug causes a partial fill, the unfilled bytes are zero rather than stale heap contents.

31.10.4 Oversized Replies

A subtle form of leak happens when the driver returns more data than the user asked for. Imagine an `ioctl` that returns a list of items:

```
/* User asks for up to user_len bytes of list data. */
if (user_len > sc->sc_list_bytes)
    user_len = sc->sc_list_bytes;

error = copyout(sc->sc_list, args->buf, sc->sc_list_bytes); /* BUG: wrong length */
```

The driver copies `sc_list_bytes` bytes regardless of what the user asked for. If `sc_list_bytes > user_len`, the driver writes past `args->buf`, which is a different bug (buffer overflow in user space). If the driver is writing to a local buffer first and then copying out, a similar error would write past the local buffer.

The correct pattern is to clamp the length and use the clamped length for the copy:

```
size_t to_copy = MIN(user_len, sc->sc_list_bytes);
error = copyout(sc->sc_list, args->buf, to_copy);
```

Information leaks through oversized replies are common when driver code evolves: the original author wrote a paired check-and-copy; a later change altered one side but not the other. Every `copyout` should use the already-validated kernel-side length, and that length should be bounded by the user's buffer size.

31.10.5 Strings and the NUL Terminator

Strings are a particularly rich source of information leaks because they have two different natural lengths: the length of the string (up to the NUL) and the size of the buffer it lives in. Suppose:

```
char name[32];
strncpy(name, "secdev0", sizeof(name)); /* copies 8 bytes, NUL-padded */

/* ... later, maybe in a different function ... */
strncpy(name, "xdev", sizeof(name)); /* copies 5 bytes, NUL-padded */

copyout(name, args->buf, sizeof(name)); /* copies all 32 bytes */
```

The second `strncpy` overwrites the first five bytes with “`xdev\0`” and then pads the rest of the buffer with NULs. That happens to be safe because `strncpy` pads with NULs when the source is shorter than the destination. But if the buffer came from `malloc(9)` without `M_ZERO`, or from a stack buffer that was written to by earlier code, bytes after the NUL may contain stale data. Copying the full buffer then leaks it.

The safe pattern is to copy only up to the NUL, or to zero the buffer before writing:

```
bzero(name, sizeof(name));
snprintf(name, sizeof(name), "%s", "secdev0");
copyout(name, args->buf, strlen(name) + 1);
```

`snprintf` guarantees NUL termination. Zeroing first ensures the bytes after the NUL are zero. The `+ 1` in the copy length includes the NUL itself.

Alternatively, copy only the string and let user space deal with its own padding:

```
copyout(name, args->buf, strlen(name) + 1);
```

The cleanest pattern is to zero first and copy exactly the valid length.

31.10.6 Sensitive Data: Explicit Zeroing Before Freeing

When a driver allocates memory to hold sensitive data (cryptographic keys, user credentials, proprietary secrets), the memory should be zeroed explicitly before being freed. Otherwise the freed memory returns to the kernel allocator’s free pool with the data still visible, and subsequent allocations from that pool may expose it.

FreeBSD provides `explicit_bzero(9)`, declared in `/usr/src/sys/sys/system.h`, which zeroes memory in a way that the compiler cannot optimize away:

```
explicit_bzero(sc->sc_secret, sc->sc_secret_len);
free(sc->sc_secret, M_SECDEV);
sc->sc_secret = NULL;
```

Ordinary `bzero` can be eliminated by the compiler if the data is not read after being zeroed, which is exactly the situation before a `free`. `explicit_bzero` is guaranteed to perform the zeroing. Use it whenever sensitive data is about to be freed or go out of scope.

There is also `zfree(9)`, declared in `/usr/src/sys/sys/malloc.h`, which zeroes and frees in one call:

```
zfree(sc->sc_secret, M_SECDEV);
sc->sc_secret = NULL;
```

`zfree` knows the allocation size from the allocator metadata and zeroes that many bytes before freeing. This is usually the cleanest pattern for cryptographic material.

For UMA zones, the equivalent is that the zone itself can be asked to zero on free, or you can `explicit_bzero` the object before calling `uma_zfree`. For stack buffers with sensitive content, `explicit_bzero` at the end of the function is the right tool.

31.10.7 Never Leak Kernel Pointers

One specific form of information leak is returning a kernel pointer to user space. The kernel address of a softc, or of an internal buffer, is useful information to an attacker trying to exploit another bug. `printf("%p")` in log messages can also leak addresses. The general rule: do not put kernel addresses in user-visible output.

For sysctls and ioctls, the simplest rule is that no field in a user-facing structure should be a raw kernel pointer. If the driver wants to expose an identifier for a kernel object, use a small integer ID (an index into a table, for example), not the address of the object. Convert from one to the other inside the driver, never expose the raw pointer.

FreeBSD's `printf(9)` supports the `%p` format, which does print a pointer, but log messages in production drivers should avoid `%p` for anything where the pointer could aid exploitation. For debugging, `%p` is fine during development; before shipping the driver, audit `printf` and `log` calls to ensure no `%p` remains in paths accessible from user space.

31.10.8 Sysctl Output

Sysctls that expose structures have the same rules as ioctls. Zero the structure before filling it, clamp the output length to the caller's buffer, and avoid pointer leaks. The `sysctl_handle_opaque` helper is often used for raw structures; make sure the structure is fully initialized before the handle returns.

A safer pattern is to expose each field as its own sysctl, using `sysctl_handle_int`, `sysctl_handle_string`, and so on. This avoids the padding problem entirely because each value is copied out as a primitive. It is also more ergonomic for users: `sysctl_secdev.stats.packets` is more useful than an opaque blob they have to decode.

31.10.9 copyout Errors

`copyout` can fail. If the user buffer becomes unmapped between the validation and the copy, `copyout` returns `EFAULT`. Your driver must handle this cleanly: typically, return the error to the user, and make sure any partial success is rolled back.

A sequence like “allocate state, fill output buffer, `copyout`, commit state” is safer than “commit state, `copyout`”. If the `copyout` fails in the second pattern, the state is committed but the user never learned what happened. If it fails in the first pattern, nothing was committed, and the user gets a clean error.

31.10.10 Deliberate Disclosure

Some sysctls and ioctls are explicitly designed to reveal information that would otherwise be private. These need an especially careful threat model. Ask: who is allowed to call this? What do they learn? Could a less-trusted attacker who obtains that information use it for something worse? A `dmesg`-style sysctl that exposes recent kernel messages is fine, but only because it has been scoped and filtered; exposing raw kernel log buffers without scoping is very different.

When in doubt, a sysctl that reveals sensitive data should be gated with `CTLFLAG_SECURE`, restricted to privileged users, and exposed only through paths that users must explicitly opt into. Default to less disclosure rather than more.

31.10.11 Kernel Pointer Hashing

Sometimes a driver legitimately needs to expose something that identifies a kernel object, for debugging or for correlating events. The raw pointer address is the wrong answer for the reasons discussed. A better answer is a hashed or masked representation that identifies the object without revealing its address.

FreeBSD provides `%p` in `printf(9)`, which prints a pointer. It also provides a related mechanism where pointers can be “obfuscated” in user-visible output using a per-boot secret, so that two pointers in the same output are consistently distinguishable but their absolute values are not leaked. The support for this varies across subsystems; when designing your own output, consider whether a dense integer ID (an index into a table) is sufficient. Often it is.

For logs, `%p` is fine during development when logs are private. Before shipping, replace any `%p` in paths reachable from user space with either a debug-only guard (so the format is present only in debug builds) or with a non-pointer identifier.

31.10.12 Wrapping Up Section 7

Information leaks are the quieter cousin of buffer overflows: they do not crash, they do not corrupt, they merely send data to user space that should have stayed in the kernel. The tools to prevent them are simple and cheap. Zero structures before filling them. Use `M_ZERO` on heap allocations that will be copied to user space. Clamp copy lengths to the smaller of the caller’s buffer and the kernel’s source buffer. Use `explicit_bzero` or `zfree` for sensitive data before freeing. Keep kernel pointers out of user-visible output. Bound strings to their actual length, not their buffer size.

A driver that applies these habits consistently will not leak information through its interfaces. The next section moves to the debugging and diagnostics side: how to log without leaking, how to debug without leaving production-hostile code behind, and how to keep the operator informed without handing an attacker a map.

31.11 Section 8: Safe Logging and Debugging

Every driver logs. `printf(9)` and `log(9)` are among the first tools a driver author reaches for, and for good reason: a well-placed log message turns a mysterious failure into a readable narrative. But logs are not free. They consume disk, they can be flooded, and they can leak sensitive data. A security-aware driver treats logging as a first-class design concern, not a debug afterthought.

This section is about writing log messages that help operators without hurting security.

31.11.1 The Logging Primitives

FreeBSD drivers have two main ways to emit messages.

`printf(9)`, the same name as the C library function but with kernel-specific semantics, writes to the kernel message buffer and, if the console is active, to the console. It is unconditional: every `printf` call results in a message.

`log(9)`, declared in `/usr/src/sys/sys/syslog.h`, writes to the kernel log ring with a syslog-compatible priority. Messages go to the in-kernel log buffer (readable by `dmesg(8)`) and, via `syslogd(8)`, to the configured log destinations. The priority is the familiar syslog scale: `LOG_EMERG`, `LOG_ALERT`, `LOG_CRIT`, `LOG_ERR`, `LOG_WARNING`, `LOG_NOTICE`, `LOG_INFO`, `LOG_DEBUG`.

Use `log(9)` when you want the message to be filtered or routed by syslog. Use `printf(9)` when you want unconditional emission, typically for very important events or for output that should always appear on the console.

`device_printf(9)` is a small wrapper over `printf` that prefixes the message with the device name (`secdev0: . . .`). Prefer it inside driver code so messages are easy to attribute.

31.11.2 What to Log and What Not to Log

A security-aware driver logs:

State transitions that matter. Attach, detach, reset, firmware update, link up, link down. These let an operator correlate driver behaviour with system events.

Errors from the hardware or from user requests. A bad ioctl argument, a DMA error, a timeout, a CRC mismatch. These let the operator diagnose problems.

Rate-limited summaries of anomalous events. If a malformed ioctl is received a million times per second, log the first, summarize the rest.

A security-aware driver does not log:

User data. The contents of buffers the user passed in. You never know what is in them.

Cryptographic material. Keys, IVs, plaintext, ciphertext. Ever.

Sensitive hardware state. On a security device, some register contents are themselves secrets.

Kernel addresses. `%p` is fine in early development; it has no place in production logs.

Details of authentication failures. A log message that says “user jane failed check X because register was 0x5d” tells an attacker what check to defeat. A log that says “authentication failed” tells the operator there was a failure without tutoring the attacker.

Think about who reads the logs. On a multi-tenant server, other users may have log-reading privileges. On a shipped appliance, the log may be exported for remote support. Treat log messages as information that could end up on any surface the system touches.

31.11.3 Rate Limiting

A noisy driver is a security problem. If an attacker can trigger a log message, they can trigger a million of them. Log flooding consumes disk space, slows the system, and buries legitimate messages. FreeBSD provides `eventratecheck(9)` and `ppsratecheck(9)` in `/usr/src/sys/sys/time.h`:

```
int eventratecheck(struct timeval *lasttime, int *cur_pps, int max_pps);
int ppsratecheck(struct timeval *lasttime, int *cur_pps, int max_pps);
```

Both return 1 if the event is allowed through and 0 if it has been rate-limited. `lasttime` and `cur_pps` are per-call state you keep in your softc. `max_pps` is the limit in events per second.

Pattern:

```
static struct timeval secdev_last_log;
static int secdev_cur_pps;

if (ppsratecheck(&secdev_last_log, &secdev_cur_pps, 5)) {
    device_printf(dev, "malformed ioctl from uid %u\n",
        td->td_ucred->cr_uid);
}
```

Now, no matter how many malformed ioctls the attacker sends, the driver emits at most 5 log messages per second. That is enough for the operator to notice something is happening without drowning the system.

Per-event rate limiting (one `lasttime/cur_pps` pair per event type) is better than a single global limit, because it prevents a flood of one event type from masking other events.

31.11.4 Log Levels in Practice

A good rule of thumb is this:

`LOG_ERR` for unexpected driver failures that require operator attention. “DMA mapping failed”, “device returned CRC error”, “firmware update aborted”.

`LOG_WARNING` for unusual but not necessarily critical situations. “Received oversized buffer, truncating”, “falling back to polled mode”.

`LOG_NOTICE` for events that are normal but worth recording. “Firmware version 2.1 loaded”, “device attached”.

`LOG_INFO` for high-volume status information that operators may filter.

`LOG_DEBUG` for debugging output. A production driver usually does not emit `LOG_DEBUG` unless the operator has enabled debug logging via a sysctl.

`LOG_EMERG` and `LOG_ALERT` are reserved for system-threatening conditions and are not typically emitted by device drivers.

Choosing the right level matters because operators configure syslog to filter by level. A driver that logs every received packet at `LOG_ERR` makes the logs useless.

31.11.5 Debug Logging and Production

During development, you will want verbose logging: every state transition, every entry and exit, every buffer allocation. That is fine. The question is how to turn it off in production without losing the ability to re-enable it when there is a bug to diagnose.

Two patterns are common:

A sysctl-controlled debug level. The driver reads a sysctl at the top of each log-worthy event and emits or suppresses the message based on the level. This allows runtime control without recompiling.

```
static int secdev_debug = 0;
SYSCTL_INT(_hw_secdev, OID_AUTO, debug, CTLFLAG_RW,
           &secdev_debug, 0, "debug level");

#define SECDEV_DBG(level, fmt, ...) do { \
    if (secdev_debug >= (level)) \
        device_printf(sc->sc_dev, fmt, ##__VA_ARGS__); \
} while (0)
```

Compile-time control. A driver can use `#ifdef SECDEV_DEBUG` to include or exclude debug blocks. This is faster (no runtime check) but requires a rebuild to change. Often the two are combined: `#ifdef SECDEV_DEBUG` wraps the infrastructure, and the sysctl controls verbosity within that.

Either way, avoid `printf` calls in hot paths that are not guarded by some kind of conditional. An uncommented `printf` in an interrupt handler or a per-packet path is a performance disaster waiting to be enabled.

31.11.6 Leaving Nothing Behind

Before committing driver changes, `grep` the driver for:

Raw `printf` calls without `device_printf` prefixes. These make log attribution harder.

`%p` format specifiers. If they appear in paths reachable from user space, replace with less sensitive formats (a sequence number, a hash, nothing).

`LOG_ERR` on user-triggerable events without rate limiting. Attackers can weaponize these.

`TODO`, `XXX`, `FIXME`, `HACK` near security-related code. Leaving these for reviewers is fine; shipping them is not.

Test-only `fprintf`-equivalents that were supposed to be removed.

31.11.7 dmesg and the Kernel Message Buffer

The kernel message buffer is a fixed-size ring buffer shared by every driver and the kernel itself. On a busy system, old messages scroll out as new ones arrive. A driver that floods the buffer pushes out useful messages from other drivers.

`dmesg(8)` shows the current contents of the buffer. Operators rely on it. Being a good citizen in the buffer means: log important things, do not log in hot paths, rate-limit everything triggerable by users, and do not flood.

The buffer size is tunable (`kern.msgbufsize` sysctl), but you cannot count on a particular size. Write as if every message is valuable and must compete with others for space.

31.11.8 KTR and Tracing

For detailed tracing without the cost of `printf`, FreeBSD provides KTR (Kernel Tracing), declared in `/usr/src/sys/sys/ktr.h`. KTR macros, when enabled, record events in a compact in-kernel ring that is separate from the message buffer. A kernel compiled with options `KTR` can be queried with `sysctl debug.ktr.buf` and with `ktrdump(8)`.

KTR events are best for per-operation tracing where a `printf` would be too heavy. They are almost free at runtime when disabled. For a security-sensitive driver, KTR gives you a way to leave tracing infrastructure in the code without paying for it in production.

Other tracing frameworks (`dtrace(1)` via SDT probes) are worth learning for deep inspection. They are out of scope for this chapter, but know that they exist.

31.11.9 Logging Privileged Operations

A specific case worth calling out: when your driver successfully performs a privileged operation, log it. This creates an audit trail. If a firmware update happens, log who triggered it. If a hardware reset is issued, log it. If a device is reconfigured, log the change.

```
log(LOG_NOTICE, "secdev: firmware update initiated by uid %u (euid %u)\n",  
    td->td_ucred->cr_ruid, td->td_ucred->cr_uid);
```

The operator can later see who did what. If there is ever a security incident, this log is the first evidence. Make it accurate and make it hard to forge.

Do not over-log legitimate privileged use; a firmware update triggered by `freebsd-update` once a month is one message, not a thousand. But the single message should carry enough detail to reconstruct what happened: who, when, what, with what arguments.

31.11.10 The audit(4) Framework

For deeper audit trails than `log(9)` provides, FreeBSD includes an audit subsystem (`audit(4)`) based on the BSM (Basic Security Module) audit format originally from Solaris. When enabled via `auditd(8)`, the kernel emits structured audit records for many security-relevant events: logins, privilege changes, file access, and, increasingly, driver-specific events when drivers instrument themselves.

A driver that handles highly sensitive operations can emit custom audit records using `AUDIT_KERNEL_*` macros declared in `/usr/src/sys/security/audit/audit.h`. This is more involved than a `log(9)` call, but it produces records that fit into the operator's existing audit workflow, are structured (machine-readable), and can be forwarded to remote audit collectors for compliance.

For most drivers, `log(9)` with `LOG_NOTICE` and a clear message is enough. For drivers that must meet specific compliance requirements (government, financial, medical), consider investing in audit integration. The infrastructure is already in the kernel; you just need to call into it.

31.11.11 Using dtrace with Your Driver

Alongside logging, `dtrace(1)` lets an operator observe driver behavior without recompiling. A driver that declares Statically Defined Trace (SDT) probes through `sys/sdt.h` exposes well-defined hook points that `dtrace` scripts can latch onto.

```
#include <sys/sdt.h>

SDT_PROVIDER_DECLARE(secdev);
SDT_PROBE_DEFINE2(secdev, , , ioctl_called, "u_long", "int");

static int
secdev_ioctl(struct cdev *dev, u_long cmd, caddr_t data, int fflag,
             struct thread *td)
{
    SDT_PROBE2(secdev, , , ioctl_called, cmd, td->td_ucred->cr_uid);
    /* ... */
}
```

An operator can then write a `dtrace` script that fires on `secdev:::ioctl_called` and counts or logs each event. The advantage over `log(9)` is that `dtrace` probes have essentially no cost when disabled, and they let the operator decide what to observe rather than forcing the driver author to anticipate every useful question.

For a security-focused driver, SDT probes on entry and exit of privileged operations let security monitoring tools observe usage patterns without the driver having to log every call. This is useful for anomaly detection: a sudden spike in `ioctl` calls from an unexpected UID, for example, can be flagged by a `dtrace`-based monitor.

31.11.12 Wrapping Up Section 8

Logging is how a driver talks to its operator. Like any communication, it can be clear or confused, honest or misleading, helpful or harmful. A security-aware driver logs important events with appropriate levels, avoids logging sensitive data, rate-limits anything an attacker can trigger, and uses debug infrastructure that can be turned on and off without recompilation. It prefers `device_printf(9)` for attribution, uses `log(9)` with thoughtful priorities, and never leaves `%p` or ungarded `printf` statements in production paths.

The next section takes a broader view. Beyond specific techniques (bounds-checking, privilege checks, safe logging), there is a design-level question: what should a driver do by default when something goes wrong? What fail-safe behavior should it exhibit? That is the subject of secure defaults.

31.12 Section 9: Secure Defaults and Fail-Safe Design

A driver's design decisions shape its security long before any individual line of code is written. Two drivers can use the same APIs, the same allocator, the same locking primitives, and end up with very different security postures, because one was designed to be open and the other was designed to be closed. This section is about the design choices that make a driver safe by default.

The central idea is summarized in a single principle: when in doubt, refuse. A driver that fails open has to be correct in every branch to be safe. A driver that fails closed only has to be correct in the narrow paths where it decides to allow something.

31.12.1 Fail Closed

The first and most important design decision is what happens when your code reaches a state it did not expect. Consider a switch statement:

```
switch (op) {
case OP_F00:
    return (do_foo());
case OP_BAR:
    return (do_bar());
}
return (0); /* fall-through: everything else succeeds! */
```

This is a fail-open design. Any operation code that is not OP_F00 or OP_BAR succeeds silently, returning 0. That is almost never what you want. A new operation code added to the API but not handled in the driver becomes a silent no-op. An attacker who discovers this can use it to bypass checks.

The fail-closed version:

```
switch (op) {
case OP_F00:
    return (do_foo());
case OP_BAR:
    return (do_bar());
default:
    return (EINVAL);
}
```

Unknown operations explicitly return an error. If a new operation is added to the API, the compiler or the tests will tell you the moment you handle it and forget to update the switch, because the new case is needed to silence the EINVAL.

The same principle applies at every decision point. When a function checks a precondition:

```
/* Fail open: if the check is inconclusive, allow the operation. */
if (bad_condition == true)
    return (EPERM);
return (0);

/* Fail closed: if the check is inconclusive, refuse. */
if (good_condition != true)
```

```
    return (EPERM);
return (0);
```

The second form fails closed: if the precondition cannot be proven good, the operation is refused. This is safer when `good_condition` has any chance of being false due to an error in setup, a race, or a bug.

31.12.2 Whitelist, Don't Blacklist

Closely related: when deciding what is allowed, whitelist the known-good rather than blacklisting the known-bad. Blacklists are always incomplete, because you cannot enumerate every bad input. Whitelists are finite by construction.

```
/* Bad: blacklist */
if (c == '\n' || c == '\r' || c == '\0')
    return (EINVAL);

/* Good: whitelist */
if (!isalnum(c) && c != '-' && c != '_')
    return (EINVAL);
```

The blacklist missed `\t`, `\x7f`, every high-bit character, and so on. The whitelist made the allowed set explicit and refused everything else.

This applies to input validation generally. A driver that accepts a set of configuration names should explicitly list them. A driver that accepts a set of operation codes should enumerate them. If a user sends something that is not on the list, refuse.

31.12.3 Smallest Useful Interface

A driver exposes functionality to user space through device nodes, `ioctl`s, `sysctl`s, and sometimes network protocols. Every exposed entry is a potential attack surface. A secure driver exposes only what users actually need.

Before shipping an `ioctl`, ask: does anyone actually use this? If a debugging `ioctl` was useful during development but has no production role, remove it or compile it out behind a debug flag. If a `sysctl` exposes internal state that only matters for engineering, hide it behind `CTLFLAG_SECURE` and consider removing it.

The cost of removing an interface now is small: a few lines of code. The cost later, when the interface has shipped and has users, is much larger. Smaller interfaces are easier to review, easier to test, and have fewer opportunities for bugs.

31.12.4 Least Privilege on Open

A device node can be created with restrictive or permissive modes. Start restrictive. A mode of `0600` or `0640` is almost always a better default than `0666`. If users complain that they cannot access the device, that is a

conversation you want to have; you can always relax the mode, and the operator can use devfs rules to do so per-site. If users silently gain access they should not have, you will not have that conversation until something breaks.

Similarly, a driver that supports jails should default to not being accessible in jails unless there is a specific reason. The reasoning is the same: it is easier to open up later than to retrofit a closed policy onto an open one.

31.12.5 Conservative Default Values

Every configurable parameter has a default. Choose conservative ones.

A driver that has a configurable “allow user X to do Y” tunable should default to X = none. If an operator wants to grant access, they can change the tunable. If the default granted access, every deployment that missed the tunable would be open.

A driver that has a timeout should default to a short timeout. If the operation usually finishes quickly, a short default is fine. If it sometimes takes longer, the operator can bump the timeout. A long default is a denial-of-service opportunity.

A driver that has a buffer size limit should default to a small limit. Again, operators can raise it; attackers cannot.

31.12.6 Defense in Depth

No single security mechanism is perfect. A defense-in-depth driver assumes any one layer can fail and builds multiple layers.

Example: suppose a driver accepts an ioctl that requires privilege. The layers of defense are:

The device node mode blocks unprivileged users from opening the device at all.

A `priv_check` at open time blocks unprivileged users even if the mode is misconfigured.

A `priv_check` on the specific ioctl catches the case where an unprivileged user somehow reached the ioctl handler.

A `jailed()` check on the ioctl blocks jailed users.

Input validation on the ioctl arguments refuses malformed requests.

A rate-limit log records repeated malformed requests.

If all five are present, a failure in any one is contained by the others. If only one is present and it fails, the driver is compromised. Defense in depth costs a little more code and a little more CPU; it buys real resilience.

31.12.7 Timeouts and Watchdogs

A driver that waits on external events should have timeouts. Hardware can fail to respond. User space can stop reading. Networks can stall. Without a timeout, a waiting driver can hold resources forever, and an attacker who controls the external event can deny service by simply not responding.

`msleep(9)` accepts a timeout argument in ticks. Use it. A sleep with no timeout is rarely the right answer in driver code.

For longer-lived operations, a watchdog timer can detect that an operation has stalled and take recovery action: abort, retry, or reset. The `callout(9)` framework is the usual mechanism.

31.12.8 Bounded Resource Use

Every resource a driver can allocate on behalf of a caller should have a cap. Buffer sizes have maximum values. Per-open resource counts have maximum values. Global resource counts have maximum values. When a cap is hit, the driver returns an error, not an attempt at “best effort”.

Without caps, a misbehaving or hostile process can exhaust resources. The exhaustion might be memory, file-descriptor-like state, interrupt-worthy events, or simply CPU time. Caps ensure that no single caller can dominate.

A reasonable default structure:

```
#define SECDEV_MAX_BUFLLEN    (1 << 20)    /* per buffer */
#define SECDEV_MAX_OPEN_BUFS  16           /* per open */
#define SECDEV_MAX_GLOBAL     256          /* driver-wide */
```

Check each cap explicitly before allocating. Return `EINVAL`, `ENOMEM`, or `ENOBUFFS` as appropriate when the cap is hit.

31.12.9 Safe Module Load and Unload

A driver that supports being unloaded must handle cleanup correctly. An unsafe unload is a security bug. If unload leaves a callback registered, or a mapping in place, or a DMA in flight, then re-loading the module (or unloading and resuming) can touch memory that is no longer owned by the driver. That is a use-after-free waiting to happen.

The rule: if any part of detach or unload fails, either propagate the error (and keep the driver loaded) or drive the cleanup to completion. Partial teardown is worse than no teardown.

A reasonable strategy: make the unload path paranoid. It checks every resource and tears down every one that was allocated, in reverse order of allocation. It uses the `callout_drain` and `taskqueue_drain` helpers to wait for async work. Only after every such resource is quiet does it free the `softc`.

If any step fails, return `EBUSY` from `detach` and document that the driver cannot currently be unloaded. That is better than half-freeing and crashing later.

31.12.10 Safe Concurrent Entries

A driver’s entry points (`open`, `close`, `read`, `write`, `ioctl`) can be called concurrently. The driver should be written as if every entry point could be called from any context at any time. Anything else is a race waiting to fire.

The practical implication: every entry point that touches shared state acquires the `softc` lock first. Every operation that uses resources from the `softc` does so under the lock. If the operation has to sleep or do user-space

work, the code drops the lock, does the work, and re-acquires carefully, checking that the state it had not changed under its feet.

Concurrency is not an afterthought. It is part of the interface.

31.12.11 Error Paths Are Normal Paths

A subtle aspect of secure design is that error paths get the same care as success paths. In a driver, error paths often free resources, release locks, and restore state. A bug on an error path is just as exploitable as a bug on a success path; often more so, because error paths are less tested.

Write every error path as if it were the happy path for a user who is trying to find bugs. Every `goto cleanup` or `out : label` is a candidate for a double-free, a missed unlock, or a left-behind mapping. Walk each error path mentally and confirm that:

Every resource allocated on the success path is freed on the error path.

No resource is freed twice.

Every lock held is released exactly once.

No error path leaves partially initialized state visible to other contexts.

A systematic pattern helps. The “single cleanup path” idiom (one label, cleanup proceeds in reverse order of allocation) catches most such bugs by construction:

```
static int
secdev_do_something(struct secdev_softc *sc, struct secdev_arg *arg)
{
    void *kbuf = NULL;
    struct secdev_item *item = NULL;
    int error;

    kbuf = malloc(arg->len, M_SECDEV, M_WAITOK | M_ZERO);

    error = copyin(arg->data, kbuf, arg->len);
    if (error != 0)
        goto done;

    item = uma_zalloc(sc->sc_zone, M_WAITOK | M_ZERO);

    error = secdev_process(sc, kbuf, arg->len, item);
    if (error != 0)
        goto done;

    mtx_lock(&sc->sc_mtx);
    LIST_INSERT_HEAD(&sc->sc_items, item, link);
    mtx_unlock(&sc->sc_mtx);
    item = NULL; /* ownership transferred */
}
```

```
done:
    if (item != NULL)
        uma_zfree(sc->sc_zone, item);
    free(kbuf, M_SECDEV);
    return (error);
}
```

Each allocation is paired with a cleanup at done. The cleanup uses NULL checks so that resources freed earlier (or never allocated) do not cause double-frees. Ownership transfers set the pointer to NULL, which suppresses the cleanup.

Consistent use of this pattern eliminates most cleanup-path bugs. The code is longer than an early-return style, but it is dramatically safer.

31.12.12 Don't Trust Yourself Either

A final aspect of fail-safe design is to assume that even your own code has bugs. Include `KASSERT(9)` checks for invariants. `KASSERT` does nothing when `INVARIANTS` is not configured (typical in release builds), but in developer kernels it checks every assertion and panics on failure. That turns a subtle corruption bug into a loud, debuggable panic.

```
KASSERT(sc != NULL, ("secdev: NULL softc"));
KASSERT(len <= SECDEV_MAX_BUFLen, ("secdev: len %zu too large", len));
```

Invariants documented as `KASSERT` help readers (future you, future colleagues) understand what the code expects. They also catch regressions that would otherwise silently corrupt state.

31.12.13 Graceful Degradation vs Full Refusal

A design choice that often arises in fail-safe work: when a non-critical part of an operation fails, should the driver continue with a degraded result, or should it refuse the operation entirely?

There is no universal answer. Each case depends on what the caller is likely to do with partial success. A driver that returns a packet with some fields uninitialized (because a subsystem call failed) is inviting the caller to trust the zero bytes as meaningful. A driver that fails the whole operation is more disruptive but less surprising.

For security-relevant operations, prefer full refusal. A privilege check that fails should not result in “most of the operation ran, but we did not do the privileged step”; it should result in the whole thing refused. A partial success that depended on the skipped step is a bug waiting to be found.

For non-security operations, graceful degradation is often the right call. If an optional statistics update fails, the main operation should still succeed. Document what the degradation looks like so callers can anticipate it.

31.12.14 Case Study: Real-World Secure Defaults in `/dev/null`

The FreeBSD `null` driver, at `/usr/src/sys/dev/null/null.c`, is worth studying as an example of secure-by-default design. It is one of the simplest drivers in the tree, yet its construction embodies most of the principles in this chapter.

It creates two device nodes, `/dev/null` and `/dev/zero`, both with world-accessible permissions (0666). This is intentional: they are meant to be used by every process, privileged or not, and neither can leak information or corrupt kernel state. The permission decision is deliberate and documented.

The read, write, and `ioctl` handlers are minimal. `null_read` returns 0 (end of file). `null_write` consumes input without touching kernel state. `zero_read` fills the user buffer with zeros using `uiomove_frombuf` against a static zero-filled buffer.

The `ioctl` handler returns `ENOIOCTL` for unknown commands, so the upper layers can translate to the proper error. A small set of specific `FIO*` commands for non-blocking and async behavior are handled, each doing only the minimal bookkeeping that makes sense for a null or zero stream.

The driver has no locking because it has no mutable state worth protecting: the zero buffer is constant, and the read/write operations do not modify any shared data. The absence of locking is not carelessness; it is a consequence of the design minimizing what is shared in the first place.

The driver's `detach` is straightforward, destroying the device nodes. Because there is no async state, no callouts, no interrupts, no taskqueues, the cleanup is correspondingly simple.

What makes this a good example of secure defaults is the discipline of not doing more than is needed. The driver does not speculatively add features, does not expose internal state, does not support `ioctls` that were not demanded by specific users. Its interface is minimal, which keeps its attack surface minimal. Its behaviour is predictable and has been exactly the same for decades.

Real drivers cannot always be this simple; most have state to manage, hardware to talk to, and operations to perform. But the design principle generalizes: the simpler the driver, the fewer the failure modes. When faced with a choice between adding functionality and leaving it out, the more secure choice is usually to leave it out.

31.12.15 Wrapping Up Section 9

Secure defaults come down to a disposition toward refusal. Default to `EINVAL` for unknown inputs. Default to restrictive modes on device nodes. Default to conservative limits on resources. Default to short timeouts. Default to strict privilege requirements. Whitelist, do not blacklist. Fail closed, not open.

None of these are exotic. They are design habits that add up. A driver built on them is not merely a driver that can be made secure; it is a driver that is secure by default, and has to be actively broken before it becomes insecure.

The next section brings the chapter to a close by looking at the other end of the development cycle: testing. How do you know your driver is as safe as you think it is? How do you hunt for the bugs that review missed?

31.13 Section 10: Testing and Hardening Your Driver

A driver is not secure because you wrote secure code. It is secure because you tested it thoroughly, including under conditions you did not design for. This section is about the tools FreeBSD gives you for finding bugs before attackers do, and the habits that make a driver stay secure as it evolves.

31.13.1 A Walkthrough: Finding a Bug with KASAN

Before the general guidance, consider a specific scenario. You have a driver that passes all your functional tests but that you suspect has a memory-safety bug. You build a kernel with options `KASAN`, boot it, load your driver, and run a stress test. The test crashes the kernel with output that looks something like:

```
=====
ERROR: KASan: use-after-free on address 0xfffffe003c180008
Read of size 8 at 0xfffffe003c180008 by thread 100123

Call stack:
  kasan_report
  secdev_callout_fn
  softclock_call_cc
  ...

Buffer of size 4096 at 0xfffffe003c180000 was allocated by thread 100089:
  kasan_alloc_mark
  malloc
  secdev_attach
  ...

The buffer was freed by thread 100089:
  kasan_free_mark
  free
  secdev_detach
  ...
=====
```

Read the output carefully. KASAN tells you the exact instruction that accessed freed memory (`secdev_callout_fn`), the exact allocation that was freed (in `secdev_attach`), and the exact free (in `secdev_detach`). Now the bug is obvious: the callout was scheduled at `attach`, but `detach` freed the buffer before draining the callout. When the callout fires after the free, it accesses the freed buffer.

The fix: add `callout_drain` to `detach` before the `free`. KASAN helped you find, in thirty seconds, a bug that might have taken hours or weeks to find by inspection, and that might never have been found in production until a customer reported a random crash.

KASAN is not free. The runtime overhead is substantial, both in CPU (perhaps 2 to 3 times slower) and in

memory (each byte of allocated memory has an accompanying shadow byte). You would not run production with it. But for developer testing, and especially for driver authors, it is one of the most effective tools available.

KMSAN works analogously for uninitialized memory reads, and KCOV powers coverage-guided fuzzing. Together they address the main classes of memory-safety bugs: use-after-free (KASAN), uninitialized memory (KMSAN), and bugs not reached by your tests (KCOV plus a fuzzer).

31.13.2 Build With Kernel Sanitizers

A stock FreeBSD kernel is optimized for production. A development kernel for driver testing should be optimized for finding bugs. The options you add to the kernel config file turn on extra checking.

options INVARIANTS enables KASSERT(9). Every assertion is checked at runtime. A failed assertion panics the kernel with a stack trace pointing to the assertion. This catches invariant violations that would otherwise corrupt data silently.

options INVARIANT_SUPPORT is implied by INVARIANTS but is sometimes needed as a separate option for modules built against an INVARIANTS kernel.

options WITNESS turns on the WITNESS lock-order checker. Every lock acquisition is recorded, and the kernel panics if a cycle is detected (A held, then B acquired; later, B held, then A acquired). This catches deadlock bugs before they deadlock.

options WITNESS_SKIPSPIN disables WITNESS for spin mutexes, which can reduce overhead at the cost of missing some checks.

options DIAGNOSTIC enables additional runtime checks in various subsystems. It is looser than INVARIANTS and catches some additional cases.

options KASAN enables the Kernel Address Sanitizer, which detects use-after-free, out-of-bounds access, and some uninitialized memory uses. It requires compiler support and a substantial memory overhead but is excellent for finding memory-safety bugs.

options KMSAN enables the Kernel Memory Sanitizer, which detects uses of uninitialized memory. This directly catches the information-leak bugs described in Section 7.

options KCOV enables kernel coverage tracking, which is what makes coverage-guided fuzzing work.

A driver-development kernel might add:

```
options INVARIANTS
options INVARIANT_SUPPORT
options WITNESS
options DIAGNOSTIC
```

and, for deeper memory-safety testing, KASAN or KMSAN on supported architectures. Build that kernel, boot it, and run your driver against it. Many bugs surface immediately.

Production builds do not typically include these options (they slow the kernel significantly). Use them as a development safety net.

31.13.3 Stress Testing

A driver that passes functional tests can still fail under stress. Stress testing exercises the driver's concurrency, its allocation patterns, and its error paths at volumes that amplify race conditions.

A simple stress harness for a character device might:

Open the device from N processes concurrently.

Each process issues M ioctls with valid and invalid arguments in a random order.

A separate process periodically detaches and re-attaches the device (or `kldunload/kldload`).

This quickly exposes races between user-space operations and detach, which are among the hardest race categories to catch by inspection.

FreeBSD's `stress2` test framework at <https://github.com/pho/stress2> has a long history of finding kernel bugs. It includes scenarios for VFS, networking, and various subsystems. A driver author can learn a lot by reading those scenarios and adapting them to the driver's interface.

31.13.4 Fuzzing

Fuzzing is the technique of generating large numbers of random or semi-random inputs and observing whether the program crashes, asserts, or misbehaves. Modern fuzzers are coverage-guided: they watch which code paths are exercised and evolve inputs that explore new paths. This is far more effective than purely random input.

For driver testing, the key fuzzer is **syzkaller**, an external project that understands syscall semantics and produces structured inputs. Syzkaller is not part of the FreeBSD base system; it is an external tool that runs on top of a FreeBSD kernel built with KCOV coverage instrumentation. Syzkaller has found many bugs in the FreeBSD kernel over the years, and a driver that wants to be exercised thoroughly benefits from being described in a syzkaller syscall description (`.txt` file under syzkaller's `sys/freebsd/`).

If your driver exposes a substantial ioctl or sysctl interface, consider writing a syzkaller description for it. The format is straightforward, and the investment pays off the first time syzkaller finds a bug no human reviewer would have spotted.

Simpler fuzzing approaches also work. A shell script that issues random ioctls with random arguments and watches `dmesg` for panics is better than no fuzzing at all. The goal is to generate inputs your design did not anticipate.

31.13.5 ASLR, PIE, and Stack Protection

Modern FreeBSD kernels use several exploit-mitigation techniques. Understanding them is part of understanding why the bugs we have discussed matter.

kASLR, kernel Address Space Layout Randomization, places the kernel's code, data, and stacks at randomized addresses at boot. An attacker who wants to jump to kernel code, or to overwrite a specific function pointer, does not know where that code or pointer is. kASLR is foundational for making many memory-safety bugs unexploitable in practice.

Information leaks (Section 7) are particularly dangerous because they can defeat kASLR. A single leaked kernel pointer gives the attacker the base address and unlocks everything else.

SSP, the Stack-Smashing Protector, places a canary value on the stack between local variables and the return address. When a function returns, the canary is checked; if it has been overwritten (because a buffer overflow clobbered it on the way to the return address), the kernel panics. SSP does not prevent overflows but it prevents many of them from gaining control of execution.

Not every function is protected. The compiler applies SSP based on heuristics: functions with local buffers, functions that take addresses of locals, and so on. Understanding this means understanding why certain buffer-overflow patterns are more exploitable than others.

PIE, Position-Independent Executables, allows the kernel (and modules) to be relocated to random addresses. Combined with kASLR, this is what makes the randomization effective.

Stack guards and guard pages surround kernel stacks with unmapped pages. An attempt to write past the stack hits an unmapped page and panics rather than silently corrupting adjacent memory.

W^X, write-xor-execute, keeps kernel memory either writable or executable, never both. This prevents many classic exploits that relied on writing shellcode into memory and then jumping to it.

A driver author does not implement any of these; they are kernel-wide protections. But a driver's bugs can undermine them. An information leak defeats kASLR. A buffer overflow that reliably hits a function pointer or vtable defeats SSP. A use-after-free that races a fresh allocation gives an attacker controlled memory at a kernel address.

In short: the point of careful driver code is not just to avoid crashes. It is to keep the kernel's defenses intact. When your driver leaks a pointer, you did not merely expose information; you downgraded the entire system's exploit-mitigation posture.

31.13.6 Reading Your Diffs

Every time you modify the driver, read the diff carefully. Look for:

New `copyin` or `copyout` calls: are the lengths clamped? Are the buffers zeroed first?

New privilege-sensitive operations: do they have `priv_check` or equivalent?

New locking: is the lock order consistent with other code?

New allocations: are they paired with frees on every path, including error paths?

New log messages: are they rate-limited? Do they leak sensitive data?

New user-visible fields in structures: are they initialized? Is the structure zeroed before the `copyout`?

A diff-review habit catches many regressions. If your project uses code review (it should), make these questions part of the checklist.

31.13.7 Static Analysis

FreeBSD kernel code can be analyzed by several static-analysis tools, including `cppcheck`, `clang-analyzer` (scan-build), and, increasingly, Coverity and GitHub CodeQL-style tools. These tools often report warnings that a human reviewer would miss: a conditional that can never be true, a pointer used after a path where it was freed, a missing null check.

Treat static-analysis warnings seriously. Most are false positives; some are real bugs. Silencing a warning should be a decision, not a reflex. When the tool is wrong, add a comment explaining why. When the tool is right, fix the code.

`syntax` check with `bmake` on the kernel source tree is a fast first pass. Running `clang --analyze` or `scan-build` against your driver is a deeper pass. Neither replaces review or testing, but both catch bugs at low cost.

31.13.8 Code Review

No tool replaces another pair of eyes. Review is especially important for security-relevant code. When proposing a change to a security-sensitive path, find someone else to look at it. Describe what the change is, what invariants it preserves, and what you checked. Be grateful when they find a problem you missed.

For open-source projects, the FreeBSD review system (reviews.freebsd.org) provides a convenient way to get external review. Use it. The community has a long tradition of thoughtful, security-aware review, and reviewers often catch things you would not.

31.13.9 Testing After a Change

When a bug is found and fixed, add a test that would have caught it. This matters because:

The same bug class often recurs in other places. A test that catches the specific instance may catch future similar bugs.

Without a test, you have no way to know that the fix worked.

Without a test, a future refactoring may re-introduce the bug.

Tests can be unit tests (in user space, exercising individual functions), integration tests (loading the driver in a VM and exercising it), or fuzz cases (inputs that used to crash and should not now). All have their place.

31.13.10 Continuous Integration

Automated testing on every change catches regressions early. A CI setup that builds the driver against a development kernel with `INVARIANTS`, `WITNESS`, and possibly `KASAN` runs the stress harness, and checks the result, is the backbone of a driver that stays safe.

For a driver in the FreeBSD tree, this is already provided by the project's CI. For out-of-tree drivers, setting up CI takes some effort but pays back quickly.

31.13.11 Treat Bug Reports Seriously

When someone reports a crash or a suspected vulnerability in your driver, treat it as real until you have evidence otherwise. Even a “harmless” bug may be exploitable in ways the reporter did not see. “I can crash the kernel with this `ioctl`” is not a minor issue; it is at minimum a denial-of-service bug, and very often a memory-safety bug that could become arbitrary code execution.

The FreeBSD security team (secteam@freebsd.org) is the right audience for vulnerability reports in the base system. For out-of-tree drivers, have a similar channel. Respond quickly, fix carefully, and credit the reporter when appropriate.

31.13.12 Hardening Over Time

A driver's security posture is not static. New classes of bugs emerge. New mitigations become available. New attack techniques make old bugs more exploitable. Budget time every release cycle to:

Re-read the security-relevant paths of the driver.

Check for newly discovered compiler warnings or static-analysis findings.

Try the latest tools (KASAN, KMSAN, syzkaller) against the driver.

Update the privilege model if FreeBSD has added new `PRIV_*` codes or more specific checks.

Remove interfaces that no user actually needs.

The discipline of regular re-examination is what distinguishes a driver that is secure on the day it ships from one that stays secure through its lifetime.

31.13.13 Post-Incident: What To Do When a Bug Becomes a CVE

A realistic chapter on security must cover the possibility that, despite all the precautions, a bug in your driver gets reported externally as a vulnerability. The pathway is typically:

A researcher or user discovers unexpected behavior in your driver.

They investigate and determine that the behavior is a security bug: information leak, privilege escalation, crash-on-untrusted-input, or similar.

They report the bug, ideally via a responsible-disclosure channel (for FreeBSD base-system drivers, this is the `secteam@freebsd.org` address).

You receive the report.

The first response matters. Even if the bug turns out to be less serious than it looks, treat the reporter as a collaborator, not an adversary. Acknowledge receipt promptly. Ask clarifying questions if needed. Do not dismiss without investigation. Do not attempt to gag the reporter. Most vulnerability researchers want the bug fixed; if you cooperate, you get a fix faster and usually get public credit that reflects well on the project.

Triage the report technically. Can you reproduce the bug? Is it a crash, an information leak, a privilege escalation, or something else? What is the attacker model: who has to have access, and what do they gain? Is it exploitable in combination with other known bugs?

If confirmed, coordinate a fix. Keep in mind that for FreeBSD base-system drivers, the fix must flow through the project's normal review process and, where appropriate, through the security advisory process. For out-of-tree drivers, you have more flexibility but still should write the fix carefully and test it thoroughly.

Prepare the disclosure. Typical disclosure practice gives the project time to fix the bug before details become public. Industry norms are usually 90 days. Within that window, the advisory is prepared, a patched version is released, and public disclosure happens simultaneously with the release. Do not leak details early; do not delay past the agreed date.

Write the commit message carefully. Security fix commits should mention the vulnerability without giving attackers a roadmap. "Fix incorrect bounds check in `secdev_write` that could allow kernel memory disclosure" is better than either "tweak write" (too vague, reviewers miss it) or "Fix CVE-2026-12345, where an attacker

can read arbitrary kernel memory by issuing a write of X bytes followed by a read, bypassing Y check” (too specific, attackers read your commit history before users can upgrade).

After the release, if details become public, be prepared to answer questions. Users want to know: am I vulnerable, how do I upgrade, and how can I tell if I was attacked? Have clear, calm answers ready.

Post-mortem the bug. Not to blame, but to learn. Why did the bug exist? Was there a pattern the review missed? Could a tool have caught it? Should the team’s process change? Write the conclusions down; apply them in future work.

Security is a continuing practice, and post-incident learning is one of its most important parts. A project that fixes the bug and moves on has learned nothing; a project that reflects on why the bug occurred makes the next bug less likely.

31.13.14 Wrapping Up Section 10

Testing and hardening are how a careful design becomes a secure one. Build your development kernel with INVARIANTS, WITNESS, and, where possible, KASAN or KMSAN. Stress-test under concurrent load. Fuzz with syzkaller or, at minimum, with a random-input harness. Use static analysis. Review diffs. Respond seriously to bug reports. Re-test after every fix. Harden over time.

A driver does not become secure by accident. It becomes secure because the author assumed bugs existed, looked for them with every tool available, and fixed them one at a time.

31.14 Hands-On Labs

The labs in this chapter build a small character device called `secdev` and guide you through making it progressively more secure. Each lab starts from a provided starter file, asks you to make specific changes, and provides a “fixed” reference to compare against. Work through them in order.

The labs are designed to be run in a FreeBSD 14.3 virtual machine or test host where kernel panics are acceptable. Do not run them on a machine with important running services; an error in the unsafe driver can crash the kernel.

If you are running these labs inside a VM, make sure the VM is configured to write crash dumps to a location you can recover after reboot. Enable `dumpon(8)` and set `/etc/fstab` appropriately so that core dumps land in `/var/crash` after a panic. See `/usr/src/sbin/savecore/savecore.8` for details. This infrastructure is how you will diagnose any panics the labs provoke.

The companion files for these labs are under `examples/part-07/ch31-security/`. Each lab has its own subfolder containing a `secdev.c` source file, a `Makefile`, a `README.md` describing the lab, and, where appropriate, a `test/` subfolder with small user-space test programs.

As you work through the labs, keep a running log in your lab logbook: which files you modified, what you observed when you loaded the broken version, what you observed with the fix in place, and any unexpected behavior. The logbook is a learning tool; it forces you to articulate what you see, which is how learning consolidates.

31.14.1 Lab 1: The Unsafe secdev

Goal. Build, load, and test the intentionally unsafe version of `secdev`, confirm that it works, and then identify at least three security issues by reading the code with a security mindset.

Prerequisites.

This lab assumes you have a FreeBSD 14.3 virtual machine or test system where you can load and unload kernel modules. You should have already completed the module-building chapters (Part 2 and onward) so that `make`, `kldload`, `kldunload`, and device node access are familiar. If you have not, pause and revisit those chapters; the rest of Chapter 31 assumes you are comfortable with module compilation.

Steps.

1. Copy `examples/part-07/ch31-security/lab01-unsafe/` to a working directory on your FreeBSD test machine. You can either clone the book's companion repository or copy the files manually if you extracted them locally.
2. Read `secdev.c` carefully. Note what it does: it provides a `/dev/secdev` character device with `read`, `write`, and `ioctl` operations. `read` returns the contents of an internal buffer. `write` copies user data into the buffer. An `ioctl` (`SECDEV_GET_INFO`) returns a structure describing the device.
3. Read `Makefile`. It should be a standard FreeBSD kernel module makefile using `bsd.kmod.mk`.
4. Build the module with `make`. Address any build errors by consulting earlier chapters on module-building. A successful build produces `secdev.ko`.
5. Load the module with `kldload ./secdev.ko`. Verify with:

```
kldstat | grep secdev
ls -l /dev/secdev
```

You should see the module listed and the device node present with whatever permissions the unsafe driver created.

6. Exercise the device as a normal functional test:

```
echo "hello" > /dev/secdev
cat /dev/secdev
```

You should see `hello` printed back. If you do not, check `dmesg` for error messages.

7. Now, review the code with the security mindset from this chapter. For each of the following categories, find at least one issue in the unsafe code:
 - Buffer overflow opportunity.
 - Information leak opportunity.
 - Missing privilege check.
 - Unchecked user input. Write down each finding in your lab logbook, including the line number and the specific concern.
8. Unload the module with `kldunload secdev` when you are done. Verify with `kldstat` that it is gone.

Observations.

The unsafe `secdev` has several issues by design. In `secdev_write`, the code calls `uiomove(sc->sc_buf, uio->uio_resid, uio)`, which copies `uio_resid` bytes regardless of `sizeof(sc->sc_buf)`. A write of 8192 bytes to a 4096-byte buffer overflows the internal buffer. Depending on what lies next to `sc_buf` in memory, this may or may not crash immediately, but it always corrupts adjacent kernel memory.

`SECDEV_GET_INFO` returns a `struct secdev_info` that is filled in field-by-field without being zeroed first. Any padding bytes between fields carry stack contents to user space. The structure likely has padding around the `uint64_t` members for alignment.

The device is created with `args.mda_mode = 0666` (or equivalent), allowing any user on the system to read and write. A user with no special privilege can corrupt the kernel buffer or leak information through the `ioctl`.

The `ioctl` handler does not check `priv_check` or similar. Any user who can open the device can issue any `ioctl`.

`secdev_read` copies `sc->sc_buflen` bytes regardless of the caller's buffer size, potentially reading beyond valid data if `sc_buflen` was ever larger than the currently valid content.

Additional exploration.

As a non-root user, try the operations that should be privileged and confirm that they succeed when they should not. Write a short C program that issues `SECDEV_GET_INFO` and prints the returned structure as a hex dump. Look for non-zero bytes in fields that were not explicitly set; those are leaked kernel data.

Wrapping up.

The goal of this lab is pattern recognition. A real driver would have subtler versions of all these bugs, buried inside hundreds of lines of code. Training yourself to see them in a simple driver makes them easier to see everywhere else. Keep `lab01-unsafe/secdev.c` as a reference for what not to do.

31.14.2 Lab 2: Bounds-Check the Buffer

Goal. Fix the buffer overflow in `write` and add a corresponding length check in `read`. Observe the difference in how the driver behaves when stress-tested.

Steps.

1. Start from `lab02-bounds/secdev.c`. This is `lab01`'s code plus some `TODO` comments marking where you will add checks.
2. In `secdev_write`, calculate how much data can safely be written to the internal buffer. Remember that `uiomove` writes at most the length you pass. Clamp `uio->uio_resid` to the remaining space before calling `uiomove`.
3. In `secdev_read`, make sure you only copy out as much data as is actually valid in the buffer, not its full allocated size.
4. Rebuild and re-test. With the fixes in place, a write of 10KB to a 4KB buffer should simply fill the buffer, not overflow it.
5. Stress-test the fixed driver:

```
dd if=/dev/zero of=/dev/secdev bs=8192 count=100
dd if=/dev/secdev of=/dev/null bs=8192 count=100
```

Neither command should crash the kernel or produce warnings in `dmesg`. If they do, your bounds check is incomplete.

6. Compare your fix with `lab02-fixed/secdev.c`. If your fix is different but correct, that is fine; multiple solutions can be valid. If yours is incorrect, study the reference fix and understand where you went wrong.

Building confidence.

Write a small C program that issues writes of various sizes (0 bytes, 1 byte, buffer size, buffer size + 1, much larger than buffer size) and verifies that each returns the expected number of bytes written or a sensible error. This kind of boundary testing is what real driver tests look like.

Wrapping up.

Bounds checking is the simplest security fix and it catches a large fraction of real-world driver bugs. Internalize the pattern: every `uiomove`, `copyin`, `copyout`, and `memcpy` bounds the length against both source and destination sizes. The compiler cannot catch this for you; it is entirely the author's responsibility.

31.14.3 Lab 3: Zero Before copyout

Goal. Fix the information leak in the `SECDEV_GET_INFO` ioctl. Observe, via a user-space test program, the difference between the broken and fixed versions.

Steps.

1. Start from `lab03-info-leak/secdev.c`. This contains the ioctl as in the original unsafe code.
2. Observe the structure definition. Note the padding between fields:

```
struct secdev_info {
    uint32_t version;
    /* 4 bytes of padding here on 64-bit systems */
    uint64_t flags;
    uint16_t id;
    /* 6 bytes of padding to align name to 8 bytes */
    char name[32];
};
```

Check the size with `pahole` or a small C program that prints `sizeof(struct secdev_info)`.

3. Before fixing, build and load the broken version. Run the test program provided in `lab03-info-leak/test/leak_check.c`. It issues the ioctl repeatedly and dumps the returned structure as a hex dump. Look at the padding bytes. You should see non-zero values that differ between runs; those are leaked kernel stack bytes.
4. In `secdev_ioctl`, before filling in the `struct secdev_info`, zero the structure with `bzero` (or use `= { 0 }` initialization).
5. Also fix the name field: use `snprintf` instead of `strncpy` to guarantee a NUL terminator, and copy only up to the NUL rather than the full buffer size.

6. Rebuild and re-test with the same `leak_check` program. The padding bytes should now be zero on every run. The visible behavior from user space is unchanged; the internal change is that padding bytes no longer carry stack contents.
7. Compare with `lab03-fixed/secdev.c`.

A deeper exploration.

If you have KMSAN built into your kernel, load the broken version of the driver and run `leak_check`. KMSAN should report an uninitialized read when the structure is copied out. This demonstrates why KMSAN is valuable: it catches information leaks that are invisible without it.

Wrapping up.

This fix costs a single `bzero` call. The benefit is that the `ioctl` cannot leak information, ever, regardless of what future changes add or remove fields. Make `bzero` (or zero-initializing declaration) part of your reflex for any structure that will touch `copyout`, `sysctl`, or similar boundaries.

31.14.4 Lab 4: Add Privilege Checks

Goal. Restrict the device to privileged users, and verify that unprivileged access is refused.

Steps.

1. Start from `lab04-privilege/secdev.c`.
2. Modify the device-node creation code in `secdev_mdevent` (or `secdev_at_tach`, depending on structure) to use a restrictive mode (`0600`) and the root user and group:

```
args.mda_uid = UID_ROOT;
args.mda_gid = GID_WHEEL;
args.mda_mode = 0600;
```

3. In `secdev_open`, add a `priv_check(td, PRIV_DRIVER)` call at the top:

```
error = priv_check(td, PRIV_DRIVER);
if (error != 0)
    return (error);
```

Return the error if the check fails.

4. Rebuild and reload the module.
5. Test from a non-root shell:

```
% cat /dev/secdev
cat: /dev/secdev: Permission denied
```

Open should fail with `EPERM` (reported as “Permission denied”). The filesystem mode blocks access before `d_open` is even reached.

6. Temporarily change the device-node mode (as root) with `chmod 0666 /dev/secdev`. Try again as non-root. This time the filesystem allows the open, but `priv_check` in `d_open` refuses:

```
% cat /dev/secdev
cat: /dev/secdev: Operation not permitted
```

This demonstrates the in-kernel layer of the defense.

7. Reset the permissions with `chmod 0600 /dev/secdev` or reload the module to restore the default.

8. As root, the device should continue to work normally. Verify:

```
# echo "hello" > /dev/secdev
# cat /dev/secdev
hello
```

9. Compare with `lab04-fixed/secdev.c`.

Digging deeper.

Try creating a jailed environment and running a shell as root inside the jail:

```
# jail -c path=/ name=testjail persist
# jexec testjail sh
# cat /dev/secdev
```

Depending on whether your driver has added a `jailed()` check, the behavior differs. If the driver does not check `jailed`, `jailed-root` can still access the device. Add `if (jailed(td->td_ucred)) return (EPERM);` at the top of `secdev_open` and verify that the jailed access is now refused.

Wrapping up.

Restricting device-node permissions is a two-layer defense: the filesystem layer, and the in-kernel `priv_check`. Both together make the driver robust against configuration mistakes. Adding `jailed()` on top blocks even root-in-jail from sensitive operations. Each layer defends against a different failure mode; do not rely on any single one.

31.14.5 Lab 5: Rate-Limited Logging

Goal. Add a rate-limited log message for malformed ioctls and verify that a flood of malformed requests does not overwhelm the log.

Steps.

1. Start from `lab05-ratelimit/secdev.c`.
2. Add a static `struct timeval` and a static `int` to hold rate-limit state. These are global per-driver, not per-softc, unless you specifically want per-device limits:

```
static struct timeval secdev_log_last;
static int secdev_log_pps;
```

3. In `secdev_ioctl`, in the default branch (the case that handles unknown ioctls), use `ppsratecheck` to decide whether to log:

```

default:
    if (ppsratecheck(&secdev_log_last, &secdev_log_pps, 5)) {
        device_printf(sc->sc_dev,
            "unknown ioctl 0x%lx from uid %u\n",
            cmd, td->td_ucred->cr_uid);
    }
    return (ENOTTY);

```

The third argument, 5, is the maximum messages per second.

4. Rebuild and reload.
5. Write a tiny test program that issues a million bad ioctls in a tight loop:

```

#include <sys/ioctl.h>
#include <fcntl.h>
int main(void) {
    int fd = open("/dev/secdev", O_RDWR);
    for (int i = 0; i < 1000000; i++)
        ioctl(fd, 0xdeadbeef, NULL);
    return (0);
}

```

6. While it runs (as root), monitor `dmesg -f`. You should see messages arriving, but at no more than about 5 per second. Without rate limiting, you would have a million messages.
7. Count the messages with something like `dmesg | grep "unknown ioctl" | wc -l`. Compare to one million (the number of attempts).
8. Compare with `lab05-fixed/secdev.c`.

Variations to try.

Replace `ppsratecheck` with `eventratecheck` and note the difference (event-based vs per-second). Experiment with different maximum rates. Add a suppressed-count summary that emits periodically (“suppressed N messages in last M seconds”) for operator visibility.

Wrapping up.

Rate-limited logging gives you visibility into suspicious activity without making the driver itself a denial-of-service vector. Apply the pattern to any log message that can be triggered by user actions. The cost is a few extra lines per log statement; the benefit is that your driver is no longer a tool attackers can use to flood the system.

31.14.6 Lab 6: Safe Detach

Goal. Make `secdev_detach` safe under concurrent activity. Observe, by deliberately racing unload with active use, how the fix prevents use-after-free panics.

Steps.

1. Start from `lab06-detach/secdev.c`. This version introduces a small callout that periodically updates an internal counter, and an `ioctl` that sleeps briefly to simulate long-running work.
2. Review the current `detach` function. Note what it frees and in what order. The starter file intentionally has a flawed `detach` that frees the `softc` without draining.
3. Test the flawed version first (build with `INVARIANTS` and `WITNESS` in the kernel):
 - Start a test program that holds `/dev/secdev` open and issues the slow `ioctl` in a loop.
 - While it runs, issue `kldunload secdev`.
 - Observe the result. You may see a kernel panic, a stuck `kldunload`, or, if you are lucky, nothing visible (the race may not fire on every run). `WITNESS` may complain about lock state.
 - Rebuild and try again until you see the problem. Concurrent races can be flaky.
4. Now fix the `detach`:
 - Use `destroy_dev` on the `cdev` before any other cleanup, so that no new user-space thread can enter the driver, and any in-flight thread finishes before `destroy_dev` returns.
 - Add a `callout_drain` call before freeing the `softc`. This ensures that any in-flight callout has finished.
 - If the driver uses a taskqueue, add `taskqueue_drain_all`.
 - Only after all draining, free resources.
5. Rebuild and re-test the same race:
 - The user program continues running uninterrupted during `kldunload`.
 - After `kldunload`, the user program's next `ioctl` receives an error (typically `ENXIO`) because the `cdev` was destroyed.
 - The kernel remains stable. No panic, no `WITNESS` complaint.
6. Compare with `lab06-fixed/secdev.c`. Confirm that the fixed version handles in-flight activity safely.

Understanding what happened.

The flawed version races because: - `destroy_dev` is called, or is not called early enough. In-flight `d_*` calls continue. - The callout is scheduled in the future and has not fired yet. - The `softc` is freed while something still references it. - The freed `softc` is reused by the allocator for some other purpose. - The callout fires, touches what it thinks is its `softc`, and corrupts whatever memory is now there.

The fix sequences the cleanup: stop accepting new entries first (`destroy_dev`), stop in-flight entries by waiting for them to leave (part of `destroy_dev`'s contract), stop scheduled work (`callout_drain`), and only then free state. Each step closes a door; nothing beyond a closed door can reach the memory being freed.

Wrapping up.

Detach-time races are among the hardest driver bugs to catch by inspection, because the bug only occurs when timing aligns. Using `destroy_dev`, `callout_drain`, and `taskqueue_drain_all` defensively in every `detach` function is one of the highest-value habits you can adopt. Do it mechanically, even if you do not think your driver has asynchronous activity. The next author to add a callout may forget; your defensive `detach` catches them.

31.14.7 Lab 7: Secure Defaults Everywhere

Goal. Apply every lesson so far to a single driver: the “secure secdev”. Build it from a skeleton, then review the finished result as if you were performing a security audit.

Steps.

1. Start from `lab07-secure/secdev.c`. This is a skeleton with TODO markers in many places.
2. Fill in each TODO, applying the lessons from Labs 1 to 6 plus any additional defenses you think appropriate. Suggested additions:

- A `MALLOC_DEFINE` for the driver’s memory.
- A softc mutex protecting all shared fields.
- `priv_check(td, PRIV_DRIVER)` in `d_open` and in each privileged ioctl.
- `jailed()` checks for operations that should not be available to jailed users.
- `securelevel_gt` for operations that should be refused at elevated `securelevel`.
- `bzero` on every structure before filling it for copyout.
- `M_ZERO` on every allocation that will be copied to user space.
- `explicit_bzero` on sensitive buffers before `free`.
- Rate-limited `device_printf` on every log message triggerable from user space.
- `destroy_dev`, `callout_drain`, and other drains in `detach` before any `free`.
- A `sysctl`-controlled `secdev_debug` flag that gates verbose logging.
- Input validation that whitelists allowed operation codes.
- Bounded copies in both directions.
- `KASSERT` statements documenting internal invariants.

3. Rebuild and load the module.

4. Run a comprehensive functional test to confirm everything still works:

- As root, open the device, read, write, ioctl.
- As non-root, confirm `/dev/secdev` is inaccessible.
- Inside a jail, confirm sensitive operations are refused.

5. Run a security stress test:

- Boundary cases (0-byte reads, exactly buffer-size writes, one-byte-over writes).
- Malformed ioctls.
- Concurrent open/read/write/close from multiple processes.
- `kldunload` during active use.

6. Compare your work with `lab07-fixed/secdev.c`. Note differences. Where your version is more defensive, ask whether the extra defense is worth the complexity. Where the reference is more defensive, ask whether you missed a defense.

A self-review.

Once your lab 7 driver builds and passes tests, put on the reviewer hat. Go through the Security Checklist section of this chapter and confirm each item. Any items you cannot confirm are gaps in your driver. Fix them now, while the code is fresh; later, finding and fixing such gaps is slower and more error-prone.

Wrapping up.

This lab is the consolidation of the chapter. Your finished driver is still a simple character device, but it is one you would not be embarrassed to see in a real FreeBSD tree. The practices you applied here are the same practices that separate amateur drivers from professional ones. Keep your lab 7 driver as a reference: when you write your first real driver, this is the skeleton you will start from.

31.15 Challenge Exercises

These challenges go beyond the labs. They are longer, more open-ended, and assume you have finished Lab 7. Take your time. None of them require new FreeBSD APIs; they require deeper application of what you have learned.

These challenges are meant to be attempted over days or weeks, not in a single sitting. They exercise judgment as much as coding: the question “is this secure” is often “secure against what threat model”. Being explicit about the threat model is part of the exercise.

31.15.1 Challenge 1: Add a Multi-Step ioctl

Design and implement an ioctl that performs a multi-step operation on secdev: first, the user uploads a blob; second, the user requests processing; third, the user downloads the result. Each step is a separate ioctl call.

The challenge is to manage per-open state correctly: the blob uploaded in step 1 must be associated with the calling file descriptor, not globally. Two concurrent users must not see each other’s blobs. State must be cleaned up when the file descriptor is closed, even if the user never reached step 3.

Security considerations: bound the blob size, validate each step of the state machine (cannot request processing without a blob; cannot download without completing processing), make sure partial state on error paths is cleaned up, and make sure a user-visible identifier (if you expose one) is not a kernel pointer.

31.15.2 Challenge 2: Write a syzkaller Description

Write a syzkaller syscall description for secdev’s ioctl interface. The format is documented in the syzkaller repository. Install syzkaller and feed it your driver; run it for at least an hour (ideally longer) and see what it finds.

If it finds bugs, fix them. Write a note about what each bug was and how the fix works. If it does not find bugs in several hours, consider whether your syzkaller description really exercises the driver. Often a description that does not find bugs is not exploring the interface thoroughly.

31.15.3 Challenge 3: Detect Double Free in Your Own Code

Intentionally introduce a double-free bug into a copy of your secure secdev. Build the module against a kernel with INVARIANTS and WITNESS. Load and exercise the module in a way that triggers the double-free. Observe what happens.

Now rebuild the kernel with KASAN. Load and exercise the same broken module. Observe the difference in how the bug is detected.

Write down what each sanitizer caught and how readable the output was. This exercise builds intuition for which sanitizer to reach for first in which situation.

31.15.4 Challenge 4: Threat-Model an Existing Driver

Pick a driver in the FreeBSD tree that you have not previously examined (something small, ideally under 2000 lines). Read it carefully. Write a threat model: who are the callers, what privileges do they need, what could go wrong, what mitigations are in place, what could be added?

The goal is not to find specific bugs. It is to practice the security mindset on real code. A good threat model is a few pages of prose that would let another engineer review the same driver efficiently.

31.15.5 Challenge 5: Compare `/dev/null` and `/dev/mem`

Open `/usr/src/sys/dev/null/null.c` and `/usr/src/sys/dev/mem/memdev.c` (or the per-architecture equivalents). Read both.

Write a short essay (a page or two) on the security differences. `/dev/null` is one of the simplest drivers in FreeBSD; what does it do, and why is it safe? `/dev/mem` is one of the most dangerous; what does it do, and how does FreeBSD keep it safe? What can you learn about the shape of secure driver code from the contrast?

31.16 Troubleshooting and Common Mistakes

A short catalogue of mistakes I have seen repeatedly in real driver code, with the symptom, the cause, and the fix.

31.16.1 “Sometimes it works, sometimes it doesn’t”

Symptom. A test passes most of the time but occasionally fails. Running under load amplifies the failure rate.

Cause. Almost always a race condition. Something is being read and written concurrently without a lock.

Fix. Identify the shared state. Add a lock. Acquire the lock for the whole check-and-act sequence. Do not trust `atomic_*` operations to solve a multi-field invariant problem.

31.16.2 “The driver crashes on unload”

Symptom. `kldunload` triggers a panic or a stuck kernel.

Cause. A callout, taskqueue task, or kernel thread is still running when `detach` frees the structure it uses. Or an in-flight `cdev` operation is still in the driver’s code when `destroy_dev` is skipped.

Fix. In detach, call `destroy_dev` before anything else. Then `callout_drain` every callout, `taskqueue_drain_all` every taskqueue, and wait for every kernel thread to exit. Only then free state. Structure the detach as a strict reverse of attach.

31.16.3 “The ioctl works from root but not from my service account”

Symptom. User reports that root can use the device, but a non-root account cannot.

Cause. Device node permissions are too restrictive, or a `priv_check` call refuses the operation.

Fix. If the operation truly should be privileged, this is working as intended; document it. If not, reconsider: was the privilege check added in error? Is the device-node mode too tight? The correct answer depends on the operation; most real answers are “yes, it should be privileged, update the docs”.

31.16.4 “dmesg is flooding”

Symptom. `dmesg` shows thousands of identical messages from the driver. Legitimate messages are being pushed out.

Cause. A log statement in a path triggerable from user space, without rate limiting.

Fix. Wrap the log in `ppsratecheck` or `eventratecheck`. Limit to a few per second. If the message is about an error, include a count of suppressed messages when the rate returns to normal (the rate helpers support this).

31.16.5 “The structure comes back with garbage bytes”

Symptom. A user-space tool reports seeing apparently random data in a field it did not expect to be set.

Cause. The driver did not zero the structure before filling and copying out. The “random” data is actually stack or heap content from before.

Fix. Add a `bzero` at the top of the function, or initialize the structure with `= { 0 }` at declaration. Never copyout an uninitialized structure.

31.16.6 “We leak memory but I don’t see where”

Symptom. `vmstat -m` shows the driver’s malloc type growing over time. Eventually the system runs out of memory.

Cause. An allocation path that does not pair with a free path, or an error path that returns without freeing.

Fix. Use a named malloc type (`MALLOC_DEFINE`). Audit every allocation. Walk every error path. Consider the single-cleanup-label pattern. Build with `INVARIANTS` and watch for allocator warnings on unload.

31.16.7 “kldload succeeds but my device doesn’t show up in /dev”

Symptom. `kldstat` shows the module loaded, but there is no `/dev/secdev` entry.

Cause. Usually an error in the attach sequence before `make_dev_s` is called, or `make_dev_s` itself failed silently.

Fix. Check the return value of `make_dev_s`. Add a `device_printf` reporting any error. Verify `attach` is being reached by adding a `device_printf` at the top.

31.16.8 “A simple C test passes, but a shell script that does the same thing in a loop fails”

Symptom. Single-shot testing works. Rapid repeated testing fails.

Cause. Likely a race between repeated operations, or a resource that is not being cleaned up between calls. Sometimes a TOCTOU bug that is timing-sensitive.

Fix. Stress-test harder. Use `dtrace` or `ktrace` to see what is happening. Look for state that persists across calls and should not.

31.16.9 “KASAN says use-after-free but my malloc/free are balanced”

Symptom. KASAN reports access to freed memory, but visual inspection of the driver shows each allocation freed exactly once.

Cause. A common subtle case: a callout or taskqueue task still holds a pointer to the freed object. The callout fires after free.

Fix. Trace the callout lifecycle. Ensure `callout_drain` (or equivalent) runs before any free. A related case is an asynchronous completion callback; ensure the operation is either completed or cancelled before the owning structure is freed.

31.16.10 “WITNESS complains about lock order”

Symptom. WITNESS reports “lock order reversal” and identifies two locks that were acquired in inconsistent order.

Cause. At one point the code acquired lock A then lock B; at another point it acquired lock B then lock A. This can deadlock.

Fix. Decide on a canonical order for your locks. Document it. Acquire them in that order everywhere. If a code path legitimately needs the reverse order, use `mtx_trylock` with a backoff-and-retry pattern.

31.16.11 “vmstat -m shows a negative free count”

Symptom. `vmstat -m` lists the driver’s malloc type with a negative number of allocations, or with an inuse count that increases over time without bound.

Cause. A mismatched `malloc/free` type, or a leak where allocations happen without corresponding frees.

Fix. A negative free count almost always means a `free` call passed the wrong type tag. Audit every `free(ptr, M_TYPE)` and confirm the type matches the `malloc`. A continuously rising inuse count is a leak; audit every path that allocates and confirm it has a matching free on every exit.

31.16.12 “The driver works on amd64 but panics on arm64”

Symptom. Functional testing on amd64 passes; the same driver panics on arm64.

Cause. Often a mismatch in structure padding or alignment. arm64 has different padding rules from amd64 for some structures. An access that is aligned on amd64 may be misaligned on arm64 and panic.

Fix. Use `__packed` carefully (it changes alignment), use `__aligned(N)` where alignment matters, and avoid assuming the size or layout of a structure matches between architectures. For fields crossing the user/kernel boundary, use explicit widths (`uint32_t` rather than `int`, `uint64_t` rather than `long`).

31.16.13 “The driver compiles without errors but dmesg shows kernel build warnings”

Symptom. The module builds, but loading it produces warnings about unresolved symbols or ABI mismatches.

Cause. The module was built against a different kernel than the one it is being loaded into. The kernel ABI is not guaranteed stable across versions, so a module built against 14.2 may not load cleanly on 14.3.

Fix. Rebuild the module against the running kernel’s source tree. `uname -r` shows the running kernel version; verify that `/usr/src` matches. If they do not, install the matching source (via `freebsd-update`, `svn`, or `git`, depending on your source distribution).

31.16.14 “The driver is intermittently slower than expected”

Symptom. Benchmarks show occasional large latency spikes even under moderate load.

Cause. Often a lock-contention issue: multiple threads queue on a single mutex. Sometimes an allocator stall: `malloc(M_WAITOK)` in a hot path waits for memory to become available.

Fix. Use `dtrace` to profile lock contention (`lockstat` provider) and identify which lock is hot. Restructure to reduce the critical section, split the lock, or use a lock-free approach. For allocator stalls, preallocate or use a UMA zone with a high-water mark.

31.17 Security Checklist for Driver Code Review

This section is a reference checklist you can keep next to your code as you review a driver, yours or someone else’s. It is not exhaustive, but if every item on the list has been consciously considered, the driver is in much better shape than the average.

31.17.1 Structural Checks

The driver’s module-load and module-unload paths are symmetric. Every resource allocated on load is freed on unload, and the order of freeing is the reverse of the order of allocation.

The driver uses `make_dev_s` or `make_dev_credf` (not the legacy `make_dev` alone) so that errors during device-node creation are reported and handled.

The device node is created with conservative permissions. Mode 0600 or 0640 is the default; anything more permissive has an explicit reason recorded in comments or commit messages.

The driver declares a named `malloc_type` via `MALLOC_DECLARE` and `MALLOC_DEFINE`. All allocations use this type.

Every lock in the driver has a comment next to its declaration saying what it protects. The comment is accurate.

31.17.2 Input and Boundary Checks

Every `copyin` call is paired with a size argument that cannot exceed the destination buffer size.

Every `copyout` call uses a length that is the minimum of the caller's buffer size and the kernel source's size.

`copyinstr` is used for strings that should be NUL-terminated. The return value (including done) is checked.

Every `ioctl` argument structure is copied into kernel space before any of its fields are read.

`uiomove` calls pass a length that is clamped to the buffer being read from or written to, not `uio->uio_resid` alone.

Every user-provided length field is validated: non-zero when required, bounded below the appropriate maximum, checked against remaining buffer space.

31.17.3 Memory Management

Every `malloc` call checks the return value if `M_NOWAIT` is used. `M_WAITOK` without `M_NULLOK` is never null-checked uselessly; the code relies on the allocator's guarantee.

Every `malloc` is paired with exactly one `free` on every code path. Success paths and error paths are both audited.

Sensitive data (keys, passwords, credentials, proprietary secrets) is zeroed with `explicit_bzero` or `zfree` before the memory is released.

Structures that will be copied to user space are zeroed before being filled.

Buffers allocated for user output use `M_ZERO` at allocation time to prevent stale-data leaks through the tail.

After a pointer is freed, it is either set to `NULL` or the scope immediately ends.

31.17.4 Privilege and Access Control

Operations that require administrative privilege call `priv_check(td, PRIV_DRIVER)` or a more specific `PRIV_*` constant.

Operations that should not be allowed inside a jail explicitly check `jailed(td->td_ucred)` and return `EPERM` if jailed.

Operations that depend on the system's `securelevel` call `securelevel_gt` or `securelevel_ge` and handle the return value correctly (note the inverted semantics: nonzero means refuse).

No operation uses `cr_uid == 0` as a privilege gate. `priv_check` is used instead.

Sysctls that expose sensitive data use CTLFLAG_SECURE or restrict themselves to privileged users via permission checks.

31.17.5 Concurrency

Every field of the softc that is accessed by more than one context is protected by a lock.

The full check-and-act sequence (including lookups that decide whether an operation is legal) is held under the appropriate lock.

No copyin, copyout, or uiomove call is made while holding a mutex. If user-space I/O is needed, the code drops the lock, does the I/O, and re-acquires, checking invariants.

detach calls destroy_dev (or equivalent) first, then drains callouts, taskqueues, and interrupts, then frees state.

Callouts, taskqueues, and kernel threads are tracked so that every one of them can be drained during unload.

31.17.6 Information Hygiene

No kernel pointer (%p or equivalent) is returned to user space through an ioctl, sysctl, or log message in a user-triggerable path.

No user-triggerable log message is uncapped; ppsratecheck or similar wraps every such message.

Logs do not include user-supplied data that could contain control characters or sensitive information.

Debug logging is wrapped in a conditional (sysctl or compile-time) so that production builds do not emit it by default.

31.17.7 Failure Modes

Every switch statement has a default: branch that returns a sensible error.

Every parser or validator whitelists what is allowed, rather than blacklisting what is not.

Every operation with resource use has a cap. The cap is documented.

Every sleep has a finite timeout unless a genuine reason requires unbounded waiting (and even then, PCATCH is used to allow signals).

Every error path frees the resources its success path would have kept.

The driver's response to unexpected input is to refuse the operation, not to guess.

31.17.8 Testing

The driver has been loaded and tested against a kernel built with INVARIANTS and WITNESS. No assertions fire and no lock-order violations are reported.

The driver has been tested under concurrent load (multiple processes, multiple open file descriptors, interleaved operations).

The driver has been tested under detach-time concurrency (a user is inside the driver while unload is attempted).

Some form of fuzzing (ideally syzkaller, at minimum a randomized shell test) has been run against the driver.

The driver has been reviewed by someone other than its author. The review was specifically for security considerations, not only functionality.

31.17.9 Evolution

The driver's security posture is re-examined at regular intervals. New compiler warnings and new sanitizer findings are triaged seriously. New FreeBSD privilege codes are considered. Unused interfaces are removed.

Bug reports against the driver are treated as possibly exploitable until proven otherwise.

Commit history shows that security-relevant changes receive careful commit messages that explain what was wrong and what the fix does.

31.18 A Closer Look at Real-World Vulnerability Patterns

The principles in this chapter are abstractions over real bugs that happened in real kernels. This section studies a few patterns that have appeared across the FreeBSD, Linux, and other open-source operating systems over the years. The goal is not to catalogue CVEs (there are whole databases for that) but to train pattern recognition.

31.18.1 The Incomplete Copy

A classic pattern: a driver receives a variable-length user buffer. It copies a fixed header, extracts a length field from the header, then copies the variable portion according to that length.

```
error = copyin(uaddr, &hdr, sizeof(hdr));
if (error != 0)
    return (error);

if (hdr.body_len > MAX_BODY)
    return (EINVAL);

error = copyin(uaddr + sizeof(hdr), body, hdr.body_len);
```

The bug is that the length check compares `body_len` against `MAX_BODY`, but `body` may be a fixed-size buffer sized differently. If `MAX_BODY` is defined carelessly, or if it was once the size of `body` but `body` has since shrunk, the copy overflows `body`.

Every time you see a pattern of “validate header, then copy body based on header”, check that the length bound actually matches the destination buffer size. Use `sizeof(body)` directly if you can, rather than a macro that might drift.

31.18.2 The Sign Confusion

A length is stored as `int` but should be non-negative. A caller passes `-1`. Your code:

```
if (len > MAX_LEN)
    return (EINVAL);

buf = malloc(len, M_FOO, M_WAITOK);
copyin(uaddr, buf, len);
```

Does the first check pass? Yes, because `-1` is less than `MAX_LEN` when compared as a signed `int`. What happens in `malloc(len, ...)` with `len = -1`? On many platforms, `-1` silently becomes a very large positive `size_t`. The allocation fails (or worse, succeeds at a huge size), or `copyin` tries to copy a huge buffer.

The fix is to use unsigned types for sizes (preferably `size_t`), or to check for negative values explicitly:

```
if (len < 0 || len > MAX_LEN)
    return (EINVAL);
```

Or, better, change the type so that negative values cannot exist:

```
size_t len = arg->len;    /* copied from user, already size_t */
if (len > MAX_LEN)
    return (EINVAL);
```

Sign confusion is one of the most common root causes of buffer overflows in kernel code. Use `size_t` for sizes. Use `ssize_t` only when negative values are meaningful. Never mix signed and unsigned in a size check.

31.18.3 The Incomplete Validation

A driver accepts a complex structure with many fields. The validation function checks some fields but forgets others:

```
if (args->type > TYPE_MAX)
    return (EINVAL);
if (args->count > COUNT_MAX)
    return (EINVAL);
/* forgot to validate args->offset */

use(args->offset); /* attacker-controlled */
```

The bug is that `args->offset` is used as an index into an array without being bounds-checked. An attacker supplies a huge offset and reads or writes kernel memory.

The fix is to treat validation as a checklist. For every field in the input structure, ask: what values are legal? Enforce them all. A helper function `is_valid_arg` that centralizes the validation and is called early is better than scattered checks.

31.18.4 The Skipped Check on the Error Path

A driver carefully validates input on the success path, but the error path cleans up based on a field that was never validated:

```
if (args->count > COUNT_MAX)
    return (EINVAL);
buf = malloc(args->count * sizeof(*buf), M_F00, M_WAITOK);
error = copyin(args->data, buf, args->count * sizeof(*buf));
if (error != 0) {
    /* error cleanup */
    if (args->free_flag)           /* untrusted field */
        some_free(args->ptr);    /* attacker-controlled */
    free(buf, M_F00);
    return (error);
}
```

The error path uses `args->free_flag` and `args->ptr`, neither of which were validated. If the attacker arranges for `copyin` to fail (say, by unmapping the memory), the error path frees an attacker-controlled pointer, corrupting the kernel heap.

The lesson: validation must cover every field that any code path reads. It is tempting to think “the error path is unusual; it is fine”. Attackers specifically aim for error paths because they are less tested.

31.18.5 The Double-Lookup

A driver looks up an object in a table by name or ID, then performs an operation. Between the lookup and the operation, the object is removed by another thread. The operation then acts on freed memory.

```
obj = lookup(id);
if (obj == NULL)
    return (ENOENT);
do_operation(obj); /* obj may have been freed in between */
```

The fix is to take a reference on the object (using a refcount) inside the lookup, hold the reference across the operation, and release it at the end. The lookup function takes the lock, increments the refcount, and releases the lock. The operation then works with a refcount-held pointer that cannot be freed out from under it. The release decrements the refcount; when it drops to zero, the last holder frees the object.

Reference counts are the FreeBSD-canonical answer to the double-lookup problem. See `/usr/src/sys/sys/refcount.h`.

31.18.6 The Buffer That Grew

A buffer was once 256 bytes. A constant `BUF_SIZE = 256` was defined. The code checked `len <= BUF_SIZE` and copied `len` bytes into the buffer. Later, someone increased the buffer to 1024 bytes but forgot to update the

constant. Or the constant was updated but an `sizeof(buf)` in one call was not, because it was not using the constant.

This class of bug is prevented by always using `sizeof` on the destination buffer directly, rather than a constant that may drift:

```
char buf[BUF_SIZE];
if (len > sizeof(buf))    /* always matches the actual buf size */
    return (EINVAL);
```

Constants are useful when multiple places need the same bound. If you use a constant, keep the definition and the array adjacent in the source code, and consider adding a `_Static_assert(sizeof(buf) == BUF_SIZE, ...)` to catch drift.

31.18.7 The Unchecked Pointer from a Structure

A driver receives a structure from user space that contains pointers. The driver uses the pointers directly:

```
error = copyin(uaddr, &cmd, sizeof(cmd));
/* cmd.data_ptr is user-space pointer */
use(cmd.data_ptr); /* treating user pointer as kernel pointer */
```

This is a catastrophic bug: the pointer is a user-space address, but the code dereferences it as if it were kernel memory. On some architectures this may access whatever memory happens to be at that address in kernel space, which is usually garbage or invalid. On others, it faults. In some specific pathological cases, it accesses sensitive kernel data.

The fix: never dereference a pointer obtained from user space. Pointers in user-supplied structures must be passed to `copyin` or `copyout`, which correctly translate user addresses. Never treat them as kernel addresses.

31.18.8 The Forgotten copyout

A driver reads a structure from user space, modifies it, but forgets to copy the modified version back:

```
error = copyin(uaddr, &cmd, sizeof(cmd));
if (error != 0)
    return (error);

cmd.status = STATUS_OK;
/* forgot to copyout */
return (0);
```

This is a functional bug, not strictly a security bug, but its mirror image is: forgetting `copyin` and assuming a field was already set. “I set `cmd.status` in `copyin`, then I read it later” is wrong if the field was actually set by user space; the user’s value is what the code reads.

Every structure that flows user-to-kernel and back needs a clear convention about when copyin and copyout happen, and what fields are authoritative in which direction. Document it and follow it.

31.18.9 The Accidental Race

A driver takes a lock, reads a field, releases the lock, and then uses the value:

```
mtx_lock(&sc->sc_mtx);
val = sc->sc_val;
mtx_unlock(&sc->sc_mtx);

/* ... some unrelated work ... */

mtx_lock(&sc->sc_mtx);
if (val == sc->sc_val) {
    /* act on val */
}
mtx_unlock(&sc->sc_mtx);
```

The driver assumes `val` is still current because it re-checks. But “act on val” uses the stale copy, not the current field. If `sc_val` is a pointer, the act may operate on a freed object. If `sc_val` is an index, the act may use a stale index.

The lesson: once you release a lock, any value you read under that lock is stale. If you need to re-act under the lock, re-read the state inside the re-acquisition. The `if (val == sc->sc_val)` check protects against changes; the act needs to use the current value, not the stored one.

31.18.10 The Silent Truncation

A driver receives a string of up to 256 bytes, stores it in a 128-byte buffer. The code uses `strncpy`:

```
strncpy(sc->sc_name, user_name, sizeof(sc->sc_name));
```

`strncpy` stops at the destination size. But `strncpy` does not guarantee a NUL terminator if the source was longer. Later code does:

```
printf("name: %s\n", sc->sc_name);
```

`printf("%s", ...)` reads until a NUL. If `sc_name` is not NUL-terminated, `printf` reads past the end of the array into adjacent memory, potentially leaking that memory in the log or crashing.

Safer options: `strncpy` (guarantees NUL termination, truncates if needed), or `snprintf` (same guarantee with formatting). `strncpy` is a landmine; it is in the standard library only for historical reasons.

31.18.11 The Over-Logged Event

A driver logs every time an event fires. The event is user-triggerable. A user sends a million events in a loop. The kernel message buffer fills and overflows; legitimate messages are lost. The user has accomplished a denial-of-service on the logging subsystem itself.

The fix, as discussed in Section 8, is rate limiting. Every user-triggerable log message should be wrapped in a rate-limit check. A suppressed-count summary (“[secdev] 1234 suppressed messages in last 5 seconds”) can be emitted periodically to inform the operator of ongoing flooding.

31.18.12 The Invisible Bug

A driver works fine for years. Then a compiler update changes how it handles a specific idiom, or a kernel API changes semantics in a new FreeBSD release, and the driver’s behaviour changes. A check that used to work silently stops working. Users do not notice until an exploit appears.

Invisible bugs are the strongest argument for KASSERT, sanitizers, and tests. A KASSERT(`p != NULL`) at the top of every function documents what that function expects. An INVARIANTS kernel catches the moment an invariant breaks. A good test suite notices when behavior changes.

The simpler the function and the clearer its contract, the fewer places invisible bugs can hide. This is one reason the FreeBSD kernel coding style described in `style(9)` values short functions with clear responsibilities: they are easier to reason about, which makes invisible bugs easier to avoid in the first place.

31.18.13 Wrapping Up the Pattern Catalogue

Each of the patterns above has been seen in real kernel code. Many have been CVEs. The defenses are:

- Use `size_t` for sizes; avoid sign confusion.
- Whitelist validation; do not forget fields.
- Treat error paths with the same rigor as success paths.
- Use refcounts to manage object lifetime under concurrency.
- Use `sizeof` directly on the buffer rather than a drift-prone constant.
- Never dereference user pointers.
- Keep the `copyin / copyout` story explicit per field.
- Remember that a value read under a lock is stale after the lock is released.
- Use `strncpy` or `snprintf`, never `strncpy`.
- Rate-limit every user-triggerable log.
- Write invariants as KASSERT so regressions are caught.

Memorize these patterns. Apply them as a mental checklist on every function you write or review.

31.19 Appendix: Headers and APIs Used in This Chapter

A short reference to the FreeBSD headers referenced throughout this chapter, grouped by topic. Each header is in `/usr/src/sys/` followed by the path listed.

31.19.1 Memory and Copy Operations

- `sys/systm.h`: declarations for `copyin`, `copyout`, `copyinstr`, `bzero`, `explicit_bzero`, `printf`, `log`, and many kernel core primitives.
- `sys/malloc.h`: `malloc(9)`, `free(9)`, `zfree(9)`, `MALLOC_DECLARE`, `MALLOC_DEFINE`, `M_*` flags.
- `sys/uio.h`: `struct uio`, `uiomove(9)`, `UIO_READ` / `UIO_WRITE` constants.
- `vm/uma.h`: UMA zone allocator (`uma_zcreate`, `uma_zalloc`, `uma_zfree`, `uma_zdestroy`).
- `sys/refcount.h`: reference-count primitives (`refcount_init`, `refcount_acquire`, `refcount_release`).

31.19.2 Privilege and Access Control

- `sys/priv.h`: `priv_check(9)`, `priv_check_cred(9)`, `PRIV_*` constants, `securelevel_gt`, `securelevel_ge`.
- `sys/ucred.h`: `struct ucred` and its fields.
- `sys/jail.h`: `struct prison`, `jailed(9)` macro, prison-related helpers.
- `sys/capsicum.h`: Capsicum capabilities, `cap_rights_t`, `IN_CAPABILITY_MODE(td)`.
- `security/mac/mac_framework.h`: MAC framework hooks (mostly for policy writers, but reference).

31.19.3 Locking and Concurrency

- `sys/mutex.h`: `struct mtx`, `mtx_init`, `mtx_lock`, `mtx_unlock`, `mtx_destroy`.
- `sys/sx.h`: shared/exclusive locks.
- `sys/rwlock.h`: read/write locks.
- `sys/condvar.h`: condition variables (`cv_init`, `cv_wait`, `cv_signal`).
- `sys/lock.h`: common lock infrastructure.
- `sys/atomic_common.h`: atomic operations (and architecture-specific headers).

31.19.4 Device Files and Dev Infrastructure

- `sys/conf.h`: `struct cdev`, `struct cdevsw`, `struct make_dev_args`, `make_dev_s`, `make_dev_credf`, `destroy_dev`.
- `sys/module.h`: `DRIVER_MODULE`, `MODULE_VERSION`, kernel module declarations.
- `sys/kernel.h`: `SYSINIT`, `SYSUNINIT`, and related kernel hook macros.
- `sys/bus.h`: `device_t`, device methods, `bus_alloc_resource`, `bus_teardown_intr`.

31.19.5 Timing, Rate Limiting, Callouts

- `sys/time.h`: `eventratecheck(9)`, `ppsratecheck(9)`, `struct timeval`.
- `sys/callout.h`: `struct callout`, `callout_init_mtx`, `callout_reset`, `callout_drain`.
- `sys/taskqueue.h`: taskqueue primitives (`taskqueue_create`, `taskqueue_enqueue`, `taskqueue_drain`).

31.19.6 Logging and Diagnostics

- `sys/syslog.h`: `LOG_*` priority constants for `log(9)`.

- `sys/kassert.h`: KASSERT, MPASS, assertion macros.
- `sys/ktr.h`: KTR tracing macros.
- `sys/sdt.h`: Statically Defined Tracing probes for `dtrace(1)`.

31.19.7 Sysctls

- `sys/sysctl.h`: `SYSCTL_*` macros, `CTLFLAG_*` flags including `CTLFLAG_SECURE`, `CTLFLAG_PRISON`, `CTLFLAG_CAPRD`, `CTLFLAG_CAPWR`.

31.19.8 Network (when applicable)

- `sys/mbuf.h`: `struct mbuf`, mbuf allocation and manipulation.
- `net/if.h`: `struct ifnet`, network interface primitives.

31.19.9 Epoch and Lock-Free

- `sys/epoch.h`: epoch-based reclamation primitives (`epoch_enter`, `epoch_exit`, `epoch_wait`).
- `sys/atomic_common.h` and architecture-specific atomic headers: memory barriers, atomic reads and writes.

31.19.10 Tracing and Observability

- `security/audit/audit.h`: kernel audit framework (when compiled in).
- `sys/sdt.h`: Statically Defined Tracing for `dtrace` integration.
- `sys/ktr.h`: KTR in-kernel tracing.

This appendix is not exhaustive; the full set of headers a driver may need is far larger. It covers the ones referenced in this chapter. When writing your own driver, `grep` through `/usr/src/sys/sys/` for the primitive you need, and read the header to understand what is available. Many of these headers are well commented and repay careful reading.

Reading the headers is itself a security practice. Every primitive has a contract: what arguments it accepts, what constraints it imposes, what it guarantees on success, what it returns on failure. A driver that uses a primitive without reading its contract is relying on assumptions that may not hold. A driver that reads the contract, and holds itself to it, is a driver that benefits from the kernel's own discipline.

Many of the headers listed above are themselves worth studying as examples of good kernel design. `sys/re-fcount.h` is small, carefully commented, and demonstrates how a simple primitive is built from atomic operations. `sys/kassert.h` shows how conditional compilation is used to build a feature that costs nothing in production but catches bugs in developer kernels. `sys/priv.h` shows how a long list of named constants can be organized by subsystem and used as the grammar of a policy. When you run out of ideas for how to structure your own driver's internals, these headers are a good place to find inspiration.

31.20 Appendix: Further Reading

A short list of resources that go deeper into FreeBSD security than this chapter can:

FreeBSD Architecture Handbook, in particular the chapters on the jail subsystem, Capsicum, and MAC framework. Available online at <https://docs.freebsd.org/en/books/arch-handbook/>.

FreeBSD Handbook security chapter, which is oriented toward administrators but includes useful context on how system-level features (jails, securelevel, MAC) interact.

Capsicum: Practical Capabilities for UNIX, the original paper by Robert Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. Explains the design rationale behind Capsicum, which helps when deciding how your driver should behave in capability mode.

“The Design and Implementation of the FreeBSD Operating System”, by Marshall Kirk McKusick, George V. Neville-Neil, and Robert N. M. Watson. The second edition covers FreeBSD 11; many security-relevant chapters remain applicable in later versions.

style(9), the FreeBSD kernel coding style guide, available as a manual page: `man 9 style`. Readable kernel code is safer kernel code; the conventions in `style(9)` are part of how the tree stays reviewable at scale.

KASAN, KMSAN, and KCOV documentation in `/usr/src/share/man/` and related sections. Reading these helps you configure and interpret sanitizer output.

syzkaller documentation, at <https://github.com/google/syzkaller>. The `sys/freebsd/` directory contains syscall descriptions that illustrate how to describe your own driver’s interface.

CVE databases such as <https://nvd.nist.gov/vuln/search> or <https://cve.mitre.org/>. Searching for “FreeBSD” or specific driver names shows real bugs that have been found and fixed. Reading a few CVE reports per month teaches a great deal about what kinds of bugs occur in practice.

FreeBSD security advisories, at <https://www.freebsd.org/security/advisories/>. These are official reports on fixed vulnerabilities. Many are kernel-side and relevant to driver authors.

The FreeBSD source tree itself is the largest and most authoritative reference. Spend time reading drivers similar to yours. Look at how they validate input, check privilege, manage locking, and handle detach. Imitating the patterns you see in well-reviewed code is one of the fastest ways to learn.

Security mailing lists, such as `freebsd-security@` and the broader `oss-security` list, carry daily traffic on kernel and driver issues across open-source projects. Subscribing passively and skimming a few posts a week builds awareness of threat trends without demanding much effort.

Formal verification literature, although specialist, has begun to touch kernel code. Projects like `seL4` demonstrate what a fully verified microkernel looks like. FreeBSD is not that, but reading about formal verification shapes how you think about invariants and contracts in your own code.

Books on secure coding practices in C such as *Secure Coding in C and C++* by Robert Seacord translate well to kernel work, since kernel C is a dialect of the same language and has the same pitfalls, plus more. Chapter-by-chapter, they provide the mental catalogue of bugs that this chapter could only sketch.

FreeBSD-specific books, notably the McKusick, Neville-Neil, and Watson book mentioned above, but also older volumes that cover the evolution of specific subsystems. Reading about how jails evolved, how Capsicum was designed, or how MAC came to be helps you understand the rationale behind the primitives rather than just their mechanics.

Conference talks from BSDCan, EuroBSDCon, and AsiaBSDCon often touch security topics. Video archives let you watch years of past talks at your own pace. Many talks are given by active FreeBSD developers and reflect current thinking.

Academic papers on operating system security from venues such as USENIX Security, IEEE S&P, and CCS provide a longer-term view. Not every paper is relevant to drivers, but the ones that are deepen your understanding of threat models, attacker capabilities, and the theoretical basis for mitigations.

The CVE feed, particularly when filtered for kernel issues, is a continuous drip of real-world examples. Reading a few each week builds intuition for what bugs look like in practice and which classes recur most often.

Your own code, six months later. Rereading your earlier work with the benefit of distance is a valuable learning tool. The bugs you will notice are the bugs you have learned to see since you wrote it. Make a habit of this; schedule time for it.

The resources above, even a small subset of them, will keep you growing for years. Security is a field of continuous learning. This chapter is one step in that learning; the next step is yours.

Every security-minded driver author should have read at least a few of these. The field moves, and staying current is part of the craft.

31.21 Wrapping Up

Security in device drivers is not a single technique. It is a way of working. Every line of code carries a little responsibility for the kernel's safety. The chapter has covered the main pillars:

The kernel trusts every driver fully. Once code runs in the kernel, there is no sandbox, no isolation, no second chance. The driver author's discipline is the system's last line of defense.

Buffer overflows and memory corruption are the classical kernel vulnerability. They are prevented by bounding every copy, preferring bounded string functions, and treating pointer arithmetic with suspicion.

User input crosses a trust boundary. Every byte from user space must be copied into the kernel with `copyin(9)`, `copyinstr(9)`, or `uiomove(9)` before it is used. Every byte going back must be copied out with `copyout(9)` or `uiomove(9)`. The user-space memory is not trustworthy; kernel memory is. Keep them cleanly separated.

Memory allocation must be checked, balanced, and accounted for. Always check `M_NOWAIT` returns. Use `M_ZERO` by default. Pair every `malloc` with exactly one `free`. Use a per-driver `malloc_type` for accountability. Use `explicit_bzero` or `zfree` for sensitive data.

Races and TOCTOU bugs are caused by inconsistent locking or by treating user-space data as stable. Fix them with locks around shared state and by copying user data before validating.

Privilege checks use `priv_check(9)` as the canonical primitive. Layer with jail awareness and `securelevel` where appropriate. Set conservative device-node permissions. Let the MAC and Capsicum frameworks work alongside.

Information leaks are prevented by zeroing structures before filling them, bounding copy lengths on both ends, and keeping kernel pointers out of user-visible output.

Logging is part of the driver's interface. Use it to help the operator without helping the attacker. Rate-limit anything triggerable from user space. Do not log sensitive data.

Secure defaults mean failing closed, whitelisting rather than blacklisting, setting conservative default values, and treating error paths with the same care as success paths.

Testing and hardening turn careful code into trustworthy code. Build with INVARIANTS, WITNESS, and the kernel sanitizers. Stress-test. Fuzz. Review. Re-test.

None of this is a one-time effort. A driver stays secure because its author keeps applying these habits, every commit, every release, for the life of the code.

The discipline is not glamorous. It is boring work: zero the structure, check the length, acquire the lock, use `priv_check`. But this boring work is exactly what keeps systems secure. An exploited kernel is a catastrophic event for users. An exploited driver is a foothold into the kernel. The person at the keyboard of that driver, deciding whether to add the bounds check or to skip it, is making a security decision that may be invisible for years and then suddenly matter very much.

Be the author who adds the bounds check.

31.21.1 One More Reflection: Security as Professional Identity

Something worth saying explicitly: the habits in this chapter are not merely techniques. They are what distinguishes a journeyman kernel author from an apprentice. Every mature kernel engineer carries this mental checklist not because they memorized it but because they have, over years, internalized a skepticism toward their own code. The skepticism is not anxiety. It is discipline.

Write code, and then read it back as if a stranger had written it. Ask what happens if the caller is hostile. Ask what happens if the value is zero, or negative, or impossibly large. Ask what happens if the other thread arrives between these two statements. Ask what happens on the error path you did not plan to test. Write the check. Write the assertion. Move on.

This is what professional kernel engineers do. It is not glamorous, it is rarely applauded, and it is what keeps the operating system we all rely on from falling apart. The kernel is not magic; it is millions of lines of carefully checked code, written and rewritten by people who treat every line as a small responsibility. Joining that profession means joining that discipline.

You have now been given the tools. The rest is practice.

31.22 Looking Ahead: Device Tree and Embedded Development

This chapter trained you to look at your driver from the outside, through the eyes of whoever might try to misuse it. The boundaries you learned to watch were invisible to the compiler but very real to the kernel: user space on one side, kernel memory on the other; one thread with privilege, another without; a length field the caller claimed, a length the driver had to verify. Chapter 31 was about *who is allowed to ask the driver to do something*, and *what the driver should check before it agrees*.

Chapter 32 shifts the perspective entirely. The question stops being *who wants this driver to run* and becomes *how does this driver find its hardware at all*. On the PC-like machines we have leaned on so far, that question had a comfortable answer. PCI devices announced themselves through standard configuration registers. ACPI-described peripherals appeared in a table the firmware handed to the kernel. The bus did the looking, the kernel

probed each candidate, and your driver's `probe()` function only had to look at an identifier and say yes or no. Discovery was mostly someone else's problem.

On embedded platforms that assumption breaks. A small ARM board does not speak PCI, does not carry an ACPI BIOS, and does not hand the kernel a neat table of devices. The SoC has an I2C controller at a fixed physical address, three UARTs at three other fixed addresses, a GPIO bank at a fourth, a timer, a watchdog, a clock tree, and a dozen other peripherals soldered onto the board in a particular arrangement. Nothing in the silicon announces itself. If the kernel is going to attach drivers to these peripherals, something has to tell the kernel where they are, what they are, and how they relate.

That something is the **Device Tree**, and Chapter 32 is where you learn to work with it. You will see how `.dts` source files describe the hardware, how the Device Tree Compiler (`dtc`) turns them into the `.dtb` blobs the bootloader hands to the kernel, and how FreeBSD's FDT support walks those blobs to decide which drivers to attach. You will meet the `ofw_bus` interfaces, the `simplebus` enumerator, and the Open Firmware helpers (`ofw_bus_search_compatible`, `ofw_bus_get_node`, the property-reading calls) that turn a Device Tree node into a working driver attachment. You will compile a small overlay, load it, and watch a pedagogical driver attach in `dmesg`.

The security habits you have built in this chapter travel with you into that territory. A driver for an embedded board is still a driver: it still runs in kernel space, still copies data across user-space boundaries, still needs bounds checks, still takes locks, still cleans up in `detach`. An ARM board does not loosen any of those requirements. If anything, embedded systems raise the stakes, because the same board image may ship to thousands of devices in the field, each one harder to patch than a server in a data center. The disposition you have just learned, skeptical of inputs, careful with memory, conservative about privilege, is exactly the disposition an embedded driver author needs.

What changes in Chapter 32 is the set of helpers you call to discover your hardware and the files you read to know where to point them. The `probe-attach-detach` shape stays. The `softc` stays. The `lifecycle` stays. A handful of new calls and a new way of thinking about hardware description are what you add. The chapter builds them up gently, from the shape of a `.dts` file to a working driver that blinks an LED on a real or emulated board.

See you there.

31.23 A Final Note on Habits

This chapter has been longer than some. The length is deliberate. Security is not a topic that can be summarized into a single punchy rule; it is a way of thinking that requires examples, practice, and repetition. A reader who finishes this chapter once will have been exposed to the patterns. A reader who returns to this chapter when starting a new driver will find new meaning in passages that seemed merely informative on the first read.

Here are the most important habits, condensed into a single list for you to carry forward. They are the reflexes that matter most in daily driver work:

Every user-space value is hostile until copied in, bounded, and validated.

Every length has a maximum. The maximum is enforced before anything uses the length.

Every structure copied to user space is zeroed first.

Every allocation is paired with a free on every code path.

Every critical section is held across the full check-and-act sequence it protects.

Every privilege-sensitive operation checks `priv_check` before acting.

Every detach path drains async work before freeing state.

Every log message triggerable from user space is rate-limited.

Every unknown input returns an error, never a silent success.

Every assumption worth making is worth writing as a `KASSERT`.

Nine lines. If these become automatic, you have the core of what this chapter teaches.

The craft grows from here. There are more patterns, more subtleties, more tools; you will encounter them as you read more FreeBSD source, as you review more code, as you write more drivers. What stays the same is the disposition: skeptical of hostile inputs, careful with memory, clear about lock boundaries, conservative about what to expose. That disposition is the one kernel engineers share across decades. You have it now. Use it well.

31.24 A Note on Evolving Threats

One further thought before the closing words. The threats we defend against today are not the threats we will defend against in ten years. Attackers evolve. Mitigations evolve. New classes of bugs are discovered, old classes are retired. A driver that was state-of-the-art in its defenses in 2020 may need updating to be considered safe in 2030.

This is not a reason for despair. It is a reason for continuous learning. Every year, a responsible driver author should read a few new security papers, try a few new sanitizers, and look at the recent CVEs affecting kernels similar to their own. Not to memorize specific vulnerabilities, but to keep a sense of where the bugs are being found today.

The patterns this chapter teaches are stable. Buffer overflows have been bugs since before UNIX. Use-after-free has been a bug since C had `malloc`. Race conditions have been bugs since kernels had multiple threads. The specific incarnations change, but the underlying defenses endure. A driver written with the disposition this chapter encourages will be mostly right in any decade; when the details shift, the author who built the disposition will adapt faster than one who merely memorized a checklist.

31.25 Closing Words

A driver is small. A driver's influence is large. The code you write runs in the most privileged part of the system, touches memory that every other process depends on, and is trusted with the secrets of users who will never see your name. That trust is not automatic; it is earned, one careful line at a time, by authors who assumed the attacker was watching and built accordingly.

The authors of FreeBSD have been writing that kind of code for decades. The FreeBSD kernel is not perfect; no kernel of its scale can be. But it has a culture of care, a set of primitives that reward diligence, and a community that treats security bugs as learning opportunities rather than embarrassments. When you write a FreeBSD driver, you are writing into that culture. Your code will be read by people who know the difference between

a buffer overflow and a buffer that happens to be large enough; who know the difference between a privilege check that catches root-outside-jail and one that catches root-inside-jail; who know that a race condition is not a rare timing fluke but a vulnerability waiting for the right attacker.

Write for those readers. Write for the user whose laptop runs your code without knowing it is there. Write for the maintainer who will inherit your work in ten years. Write for the reviewer who will spot the defensive check you added and feel quietly glad that someone thought of it.

That is what chapter 31 has been about. That is what the rest of your career as a kernel author will be about. Thank you for taking the time to work through it carefully. The chapter ends here; the practice begins tomorrow.

32 Device Tree and Embedded Development

32.1 Introduction

Chapter 31 trained you to look at your driver from the outside, through the eyes of whoever might try to misuse it. The boundaries you learned to watch were invisible to the compiler but very real to the kernel: user space on one side, kernel memory on the other; one thread with privilege, another without; a length field the caller claimed, a length the driver must verify. That chapter was about who is allowed to ask the driver to do something, and what the driver should check before it agrees.

Chapter 32 shifts the perspective entirely. The question stops being *who wants this driver to run* and becomes *how does this driver find its hardware at all*. On the machines we have leaned on so far, the answer was easy enough that you could ignore it. A PCI device announced itself through standard configuration registers. An ACPI-described peripheral appeared in a table the firmware handed to the kernel. The bus did the looking, the kernel probed each candidate, and your driver's `probe()` function only had to look at an identifier and say yes or no. Discovery was mostly someone else's problem.

On embedded platforms that assumption breaks. A small ARM board does not speak PCI, does not carry an ACPI BIOS, and does not have a firmware layer that will hand the kernel a neat table of devices. The SoC has an I2C controller at a fixed physical address, three UARTs at three other fixed addresses, a GPIO bank at a fourth, a timer, a watchdog, a clock tree, a pin multiplexer, and a dozen other peripherals that the hardware designer soldered onto the board in a particular arrangement. Nothing in the silicon announces itself. If the kernel is going to attach drivers to these peripherals, something has to tell the kernel where they are, what they are, and how they relate.

That something is the **Device Tree**, and learning to work with it is what Chapter 32 is about.

Device Tree is not a driver. It is not a subsystem in the sense that `vfs` or `devfs` is a subsystem. It is a *data structure*, a textual hardware description that the bootloader hands to the kernel at boot time, and that the kernel then walks to decide which drivers to run and where to point them. The structure has its own file format, its own compiler, its own conventions, and its own quiet conventions that embedded developers learn over time. A driver written for a Device Tree platform looks almost like the drivers you already know, with a handful of important differences in how it finds its resources. Those differences are the subject of this chapter.

We are also going to widen the map. Until now most of your drivers have run on amd64, the flavour of 64-bit x86 that powers laptops, workstations, and servers. That architecture will not be going anywhere, and your understanding of it will continue to serve you. But FreeBSD runs on more than amd64. It runs on arm64, the 64-bit ARM architecture that powers the Raspberry Pi 4, the Pine64, the HoneyComb LX2, countless industrial boards, and a growing fraction of the cloud. It runs on 32-bit ARM for older Pis, BeagleBones, and embedded appliances. It runs on RISC-V, a newer, open architecture whose first serious FreeBSD support matured over the FreeBSD 13 and 14 cycles. On every one of those architectures outside the PC-like world, Device Tree is

how drivers find their hardware. If you want to write drivers that run on anything other than a laptop, you need to know how it works.

The good news is that the shape of a driver does not change much when you cross over. Your probe and attach routines still look like probe and attach routines. Your softc still lives in the same kind of structure. Your lifecycle is still the same lifecycle, with load and unload, detach and cleanup. What changes is the set of helpers you call to discover your hardware's address, to read its interrupt specification, to turn on its clock, to pull its reset line, to request a GPIO pin. Those helpers have a family resemblance once you see a few of them. The FreeBSD source tree uses them in hundreds of places, and by the end of this chapter you will recognise them on sight and know where to look when you need one you have not used before.

The chapter proceeds in eight sections. Section 1 introduces the embedded FreeBSD world and the hardware platforms on which it runs. Section 2 explains what the Device Tree is, how its files are organised, how `.dts` source turns into `.dtb` binary blobs, and what the nodes and properties inside actually mean. Section 3 walks through FreeBSD's support for Device Tree: the `fdt(4)` framework, the `ofw_bus` interfaces, the `simplebus` enumerator, and the Open Firmware helpers that drivers use to read properties. Section 4 is the first concrete driver-writing section. You will see the shape of an FDT-aware driver from probe to attach to detach, with its `ofw_compat_data` table and its calls to `ofw_bus_search_compatible`, `ofw_bus_get_node`, and the property-reading helpers. Section 5 turns to the DTB itself: how to compile a `.dts` with `dtc`, how overlays work, how to add a custom node to an existing board description, and how FreeBSD's build tooling fits together. Section 6 is about debugging, both at boot and at runtime, using `ofwdump(8)`, `dmesg`, and the kernel log to find out why a node did not match. Section 7 brings the pieces together in a practical GPIO driver that takes its pin assignments from Device Tree and toggles an LED. Section 8 is the refactoring section: we will take the driver you built in Section 7, tighten its error handling, expose a `sysctl` for observability, and talk through what packaging an embedded image looks like.

Along the way we will touch on the peripheral frameworks that every serious embedded driver eventually leans on. The clock framework, regulator framework, and hardware-reset framework under `/usr/src/sys/dev/extracts/` are the three big ones; the pin-control framework under `/usr/src/sys/dev/fdt/` is the fourth. The GPIO framework under `/usr/src/sys/dev/gpio/` is your first stop for reading and writing pins. `interrupt-parent` chaining routes IRQs up the interrupt tree to the controller that can actually handle them. Device Tree *overlays*, files with the extension `.dtbo`, let you add or modify nodes at boot without rebuilding the base blob. And at the highest level, on arm64 in particular, a single kernel binary can run on either FDT or ACPI; the mechanism that picks between the two is worth looking at briefly, because it shows how drivers written for both are factored.

One last framing note before we begin. Embedded work can feel intimidating from a distance. The lexicon is unfamiliar, the boards are small and fiddly, the documentation is thinner than on desktop platforms, and the first time a DTB mismatch sends you into a silent boot hang it is easy to lose your nerve. None of that is a reason to stay away. The core skill you are going to build in this chapter, reading a `.dts` file and recognising what is being described, is learnable in a long afternoon. The second skill, writing a driver that matches on a compatible string and walks the properties it needs, is the same kind of driver writing you already know with three or four new helper calls. The third skill, building and loading a DTB and seeing your driver attach, is exactly the kind of feedback loop that makes kernel work enjoyable. By the end of the chapter you will have written, compiled, loaded, and observed an FDT-aware driver on either a real ARM board or an emulated one, and you will have the mental model that lets you read any FreeBSD embedded driver and understand how it finds its hardware.

The rest of the book will continue to treat drivers as drivers, but your toolkit will have grown. Let us begin.

32.2 Reader Guidance: How to Use This Chapter

Chapter 32 sits in a particular place in the book’s arc. Earlier chapters assumed a PC-like machine where buses discovered themselves and drivers bound to devices the kernel already knew about. This chapter steps sideways into a world where discovery is a problem the driver has to participate in. That step is easier than it sounds, but it does require a small shift in how you think about hardware, and a willingness to read some new kinds of files.

There are two reading paths, and one optional third path for readers who have embedded hardware at home.

If you choose the **reading-only path**, plan for three to four focused hours. You will finish the chapter with a clear mental model of what Device Tree is, how the kernel uses it, what an FDT-aware driver looks like, and where the important FreeBSD source files live. You will not have typed a driver, but you will be able to read one and understand every helper call you see. For many readers, this is the right stopping point on a first pass.

If you choose the **reading-plus-labs path**, plan for seven to ten hours spread across two or three sessions. The labs are built around a small pedagogical driver called `edled`, short for *embedded LED*. Over the chapter you will write a minimal FDT driver that matches a custom compatible string, grow it into a driver that reads its pin number and timer interval from Device Tree, and finally wrap it in a tidy detach path with a `sysctl` for runtime observability. You will compile a tiny `.dts` fragment into a `.dtb` overlay, load it with FreeBSD’s loader, and watch the driver attach in `dmesg`. None of these steps are long; none require more than a basic familiarity with `make`, `sudo`, and a shell.

If you have access to a Raspberry Pi 3 or 4, a BeagleBone, a Pine64, a Rock64, or a compatible ARM board, you can follow the **reading-plus-labs-plus-hardware path**. In that case the `edled` driver blinks a real LED attached to a real pin, and you get the satisfying experience of writing kernel code that makes a physical thing happen. If you do not have hardware, do not worry. The entire chapter can be followed with a QEMU virtual machine emulating a generic ARM platform, or even with printed simulation on a regular FreeBSD machine. You will not miss any of the conceptual material.

32.2.1 Prerequisites

You should be comfortable with the FreeBSD driver skeleton from earlier chapters: module init and exit, `probe()` and `attach()`, softc allocation and destruction, `DRIVER_MODULE()` registration, and the basics of `bus_alloc_resource_any()` and `bus_setup_intr()`. If any of those is fuzzy, a short revisit of Chapters 6 through 14 before this chapter will pay off. The helpers in this chapter sit *alongside* the familiar `bus_*` helpers; they do not replace them.

You should also be comfortable with simple FreeBSD system administration: loading and unloading kernel modules with `kldload(8)` and `kldunload(8)`, reading `dmesg(8)`, editing `/boot/loader.conf`, and running a command as root. You will use all of them, but none at any deeper level than earlier chapters have already asked.

No prior experience with embedded hardware is required. If you have never touched an ARM board, the chapter will teach you what you need. If you have used a Raspberry Pi but only as a small Linux machine, the chapter will give you a new perspective on what is going on under its covers.

32.2.2 Structure and Pacing

Section 1 sets the stage: what embedded FreeBSD looks like in practice, which architectures FreeBSD supports, what kinds of boards you are likely to encounter, and why Device Tree is the answer embedded platforms have settled on. Section 2 is the longest conceptual section: it introduces Device Tree files, their source and binary forms, their node-and-property structure, and the handful of conventions you need to read them fluently. Section 3 brings the conversation back to FreeBSD specifically, introducing the `fdt(4)` framework, the `ofw_bus` interfaces, and the `simplebus` enumerator. Section 4 is the first driver-writing section; it walks through the canonical shape of an FDT driver from probe to attach to detach. Section 5 teaches you how to build and modify `.dtb` files and how FreeBSD's overlays system works. Section 6 is the debugging section. Section 7 is the substantive worked example: the `edled` GPIO driver. Section 8 is the refactoring and finalising section.

After Section 8 you will find hands-on labs, challenge exercises, a troubleshooting reference, the Wrapping Up review, and the bridge into Chapter 33.

Read the sections in order. They build on each other, and the later sections assume the vocabulary the earlier ones establish. If you are short on time and want the smallest conceptual tour, read Sections 1, 2, and 3, then skim Section 4 for its driver skeleton; that gives you the map of the territory.

32.2.3 Work Section by Section

Each section covers one coherent piece of the subject. Read one, let it settle, then start the next. If a section ends and a point still feels blurry, pause, re-read the closing paragraphs, and open the cited FreeBSD source file. A quick look at real code often clarifies in thirty seconds what prose can only dance around.

32.2.4 Keep the Lab Driver Close

The `edled` driver used in the chapter lives under `examples/part-07/ch32-fdt-embedded/` in the book's repository. Each lab directory contains the state of the driver at that step, with its `Makefile`, a brief `README.md`, a matching DTS overlay where relevant, and any supporting scripts. Clone the directory, work in place, and load each version after every change. A kernel module that attaches in `dmesg` and toggles a pin you can observe is the single most concrete feedback loop in embedded work; use it.

32.2.5 Open the FreeBSD Source Tree

Several sections point to real FreeBSD files. The most useful ones to keep open for this chapter are `/usr/src/sys/dev/fdt/simplebus.c` and `/usr/src/sys/dev/fdt/simplebus.h`, which define the simple FDT bus enumerator every child driver binds to; `/usr/src/sys/dev/ofw/ofw_bus.h`, `/usr/src/sys/dev/ofw/ofw_bus_subr.h`, and `/usr/src/sys/dev/ofw/ofw_bus_subr.c`, which give you the compatibility helpers and property readers; `/usr/src/sys/dev/ofw/openfirm.h`, which declares the lower-level `OF_*` primitives; `/usr/src/sys/dev/gpio/gpioled_fdt.c`, a small real driver you will imitate; `/usr/src/sys/dev/gpio/gpiobusvar.h`, which defines `gpio_pin_get_by_ofw_idx()` and its siblings; and `/usr/src/sys/dev/fdt/fdt_pinctrl.h`, which defines the `pinctrl` API. Open them when the chapter points you at them. The source is the authority; the book is a guide into it.

32.2.6 Keep a Lab Logbook

Continue the lab logbook from earlier chapters. For this chapter, log a short note for each lab: which commands you ran, which DTB loaded, what `dmesg` said, which pin you used, and any surprises. Embedded debugging benefits more than most from a paper trail, because the variables that cause a driver not to attach (a missed overlay, a wrong phandle, a swapped pin number) are easy to forget and expensive to re-discover.

32.2.7 Pace Yourself

The concepts in this chapter are often easier the second time you meet them than the first. The words *phandle*, *ranges*, and *interrupt-parent* can sit uneasily for a while before clicking. If a subsection blurs, mark it, move on, and come back to it. Most readers find Section 2 (the Device Tree format itself) is the hardest of the chapter; after it, Sections 3 and 4 feel easy because they are mostly *driver code* and the shape of driver code is already familiar.

32.3 How to Get the Most Out of This Chapter

A few habits will help you turn the chapter's prose into durable intuition. They are the same habits that help with any new subsystem, tuned for the particulars of embedded work.

32.3.1 Read Real Source

The single best way to learn to read Device Tree files and FDT-aware drivers is to read real ones. The FreeBSD tree ships several hundred. Pick a peripheral you find interesting, find its driver under `/usr/src/sys/dev/`, open it, and read it. If it has an `ofw_compat_data` table near the top and a `probe()` that calls `ofw_bus_search_compatible`, you are looking at an FDT-aware driver. Notice which properties it reads. Notice how it acquires its resources. Notice what it does and does not do on detach.

The DTS side rewards the same treatment. The FreeBSD tree keeps custom board descriptions under `/usr/src/sys/dts/`, the upstream Linux-derived device trees under `/usr/src/sys/contrib/device-tree/`, and overlays under `/usr/src/sys/dts/arm/overlays/` and `/usr/src/sys/dts/arm64/overlays/`. Open a `.dts` file that describes a board you have heard of, and read it the way you would read code: top down, noticing the hierarchy, the property names, and the commentary.

32.3.2 Run What You Build

The point of the labs is that they end in something you can observe. When you load a module and nothing happens, that too is information; it usually means the driver did not match, and the chapter will teach you how to find out why. The feedback cycle is the whole point. Do not skip the loading step just because the code compiled.

32.3.3 Type the Labs

The edited driver is small on purpose. Typing it yourself slows you down enough to notice what each line does. The finger-memory of writing FDT boilerplate is worth having. Copy-pasting deprives you of that; resist the temptation even when you are sure you could reproduce the file from memory.

32.3.4 Follow the Nodes

When you read a Device Tree file or a `dmesg` boot log and you do not recognise a property, follow it. Look at the node's binding documentation in `/usr/src/sys/contrib/device-tree/Bindings/` if one exists. Search the FreeBSD source for the property name to see which drivers care about it. Embedded work is full of small conventions, and each one becomes obvious once you have seen it used in real code.

32.3.5 Treat `dmesg` as Part of the Manuscript

Almost everything interesting about FDT discovery shows up in `dmesg`, not in the shell. When a driver attaches, when a node is skipped because its status is disabled, when `simplebus` reports a child with no matching driver, you find those messages in the kernel log and nowhere else. Keep `dmesg -a` or `tail -f /var/log/messages` in a second terminal during labs. Copy relevant lines into your logbook when they teach something non-obvious.

32.3.6 Break Things on Purpose

Some of the most useful lessons in this chapter come from watching a driver *fail* to attach. A typo in a compatible string, a wrong pin number, a disabled status, a missing overlay will each produce a different flavour of silence in `dmesg`. Working through those failures in a lab environment teaches the recognition you will need when the same failure surprises you in real work. Do not only build drivers that work; break a few on purpose and look at what the kernel says.

32.3.7 Work in Pairs When You Can

Embedded debugging, like security work, benefits from a second pair of eyes. If you have a study partner, one of you can read the `.dts` while the other reads the driver; you can trade perspectives and compare what you each thought the setup was doing. The conversation tends to catch the small mistakes (a swapped cell count, a misread phandle, an interrupt cell in the wrong position) that slip past a single reader.

32.3.8 Trust the Iteration

You are not going to memorise every property, every flag, every helper on the first pass. That is fine. What you need on the first pass is the *shape* of the subject: the names of the primitives, the structure of a driver that uses them, the places to look when a concrete question comes up. The identifiers become reflex after you have written one or two FDT drivers; they are not a memorisation exercise.

32.3.9 Take Breaks

Embedded work, like security work, is cognitively dense. It asks you to hold a description of physical hardware in your head while you read software that tries to describe and control it. Two focused hours, a proper break, and another focused hour is almost always more productive than four hours of grinding through in one sitting.

With those habits in place, let us begin with the broad question: what is embedded FreeBSD, and what does it ask of a driver author that desktop FreeBSD does not?

32.4 Section 1: Introduction to Embedded FreeBSD Systems

The word *embedded* has been used so loosely over the years that it is worth pausing to say what we mean by it inside this book. For our purposes, an embedded system is a computer designed to do a specific job rather than to be a general-purpose machine. A Raspberry Pi running a thermostat control loop is embedded. A BeagleBone running a CNC controller is embedded. A small ARM box running a firewall appliance, a RISC-V development board running a dedicated sensor gateway, a router built around a MIPS SoC in the old days, all of these are embedded. A laptop is not embedded even when it is small. A server is not embedded even when it is stripped down. The word is about purpose and constraint, not size.

From a driver author's perspective, embedded systems share a handful of practical traits that shape the work. The hardware is usually an SoC with many on-chip peripherals rather than a motherboard with plug-in cards. The peripherals are fixed: they cannot be added or removed, because they are literally part of the silicon. There is usually no discoverable bus in the PCI sense; the peripherals sit at known physical addresses and the kernel must be told where they are. Power is constrained, sometimes severely. RAM is limited. Storage is limited. The boot flow is simple, often relying on a bootloader like U-Boot or a small EFI implementation. The user interface, if any, is minimal. The kernel boots, the drivers attach, the one application the machine exists to run comes up, and that is the life of the system.

FreeBSD runs on a growing family of embedded-friendly architectures. Most of this chapter assumes arm64 because it is the most widely used embedded target for FreeBSD today, but most of what you learn applies directly to 32-bit ARM, RISC-V, and, to a lesser extent, older MIPS and PowerPC platforms. The differences between these architectures matter for the compiler and the kernel's lowest-level code, but for driver writing the differences are mostly absent. The same FDT framework, the same ofw_bus helpers, the same simplebus enumerator, and the same GPIO and clock APIs work across all of them. A driver you write for a Raspberry Pi today will, with very little modification, build and run on a RISC-V SiFive HiFive Unmatched tomorrow.

32.4.1 What Embedded FreeBSD Looks Like

Picture a Raspberry Pi 4 running FreeBSD 14.3. The board has a quad-core ARM Cortex-A72 CPU, 4 GB of RAM, a PCIe root complex with a gigabit Ethernet controller and a USB host controller hanging off it, an SD card slot for storage, a Broadcom VideoCore VI GPU, an HDMI output, four USB ports, a 40-pin GPIO header, and on-chip UARTs, SPI buses, I2C buses, pulse-width modulators, and timers. Most of those peripherals sit inside the BCM2711 SoC at fixed memory-mapped addresses. A few, like the USB and the Ethernet, hang off the SoC's internal PCIe controller. The SD card is driven by a host controller inside the SoC that speaks the SD protocol.

When you power the board on, a tiny firmware on the GPU core reads the SD card, finds the FreeBSD EFI bootloader, and hands control to it. The EFI loader reads `/boot/loader.conf`, loads the FreeBSD kernel, loads the Device Tree blob that describes the hardware, loads any overlay blobs that tweak the description, and jumps to the kernel. The kernel boots, attaches a simplebus to the root node of the tree, walks the tree's children, matches drivers to nodes, attaches drivers to their hardware, and the system comes up. By the time you see a login prompt, your filesystem has been mounted from the SD card by the SD host driver, your network interface has been brought up by the Ethernet driver, your USB devices are probed and available, and the ordinary userland tools (`ps`, `dmesg`, `sysctl`, `kldload`) behave exactly as they do on a laptop.

None of that last paragraph would have been true for a PC. On a PC, the firmware is BIOS or UEFI, the peripherals are on PCI, and the kernel does not need a separate blob that describes the hardware because the PCI bus itself describes its children. The arm64 world has none of those luxuries. It has Device Tree instead.

32.4.2 Why Embedded Platforms Rely on Device Trees

The embedded world chose Device Trees because of the problem space. A small SoC has dozens of on-chip peripherals. Every SoC variant, every board revision, every integration choice the hardware engineer made shapes which peripherals are enabled, which pins they use, how their clocks are wired, and what their interrupt priorities are. There are thousands of distinct SoCs in the wild, and tens of thousands of distinct boards. A single kernel binary cannot afford to have every one of those variations compiled into it. Nor is there a magical bus protocol it could use to *ask* the hardware what is on the board; the hardware does not know, and most of it cannot answer.

The old solution, the one the embedded Linux world used before Device Trees, was called *board files*. Each board had a C source file compiled into the kernel, full of static structures that described the peripherals, their addresses, their interrupts, and their clocks. A kernel intended for five boards had five board files. A kernel intended for fifty boards had fifty, each with its own static description, each requiring the kernel to be rebuilt every time a board's hardware changed. The approach did not scale. Every board revision was a kernel release.

Device Trees are the approach that replaced board files. The core idea is elegant: rather than encode the hardware description in C inside the kernel, encode it in a separate textual file that lives outside the kernel, compile that file into a compact binary blob, and let the bootloader hand the blob to the kernel at boot. The kernel then reads the blob, decides which drivers to attach, and hands each driver the part of the blob that describes its hardware. A single kernel binary can now run on any board whose DTB it is given. A board revision that changes pin assignments or adds a peripheral needs a new DTB, not a new kernel.

The approach originated in the IBM and Apple PowerPC world of the 1990s, where the concept was called *Open Firmware* and the tree was part of the firmware itself. The modern Flattened Device Tree (FDT) format is a descendant, streamlined and reworked to suit bootloaders and kernels that do not carry a whole Forth interpreter the way Open Firmware did. In FreeBSD you will see both names. The framework is called `fdt(4)`, the helpers still live under `sys/dev/ofw/` because of that OpenFirmware lineage, and the functions that read properties are named `OF_getprop`, `OF_child`, `OF_hasprop` for the same reason. When you read *Open Firmware*, *OFW*, and *FDT* in FreeBSD source, they all refer to pieces of the same tradition.

32.4.3 Architecture Overview: FreeBSD on ARM, RISC-V, and MIPS

FreeBSD supports several embedded-friendly architectures in FreeBSD 14.3. The two most actively used are arm64 (the 64-bit ARM architecture called AArch64 in the ARM documentation) and armv7 (32-bit ARM with hardware floating point, sometimes written armv7 or armhf). RISC-V support has matured over FreeBSD 13 and 14 and now runs on real boards like the SiFive HiFive Unmatched and the VisionFive 2. MIPS support existed for a long time and powered many older routers and embedded appliances; it has been removed from the base system in recent releases, so this chapter will not dwell on it, but the skills you learn for ARM and RISC-V would transfer directly if you ever dropped back to a legacy MIPS platform.

All of those architectures use Device Tree to describe non-discoverable hardware. The boot flows differ in detail, but the shape is the same: some stage-zero firmware starts the CPU, some stage-one bootloader (U-Boot is common; on arm64 an EFI loader is increasingly standard) loads the kernel and the DTB, and the kernel takes over with the tree in hand. On arm64 the FreeBSD EFI loader handles this cleanly, and the tree can come from either a file on disk (`/boot/dtb/*.dtb`) or, on server-class hardware, from the firmware itself. On armv7 boards, U-Boot usually supplies the DTB. On RISC-V the picture is a mix of OpenSBI, U-Boot, and EFI, depending on the board.

None of these differences matter much to a driver author. The tree the kernel sees by the time your driver runs is a Device Tree, the helpers it gives you are `OF_*` and `ofw_bus_*`, and the drivers you write to run against it are portable across architectures that use the same framework.

32.4.4 Typical Limitations: No ACPI, Limited Buses, Power Constraints

It is worth enumerating the limitations of embedded platforms that shape driver work, so that they do not take you by surprise.

No ACPI, usually. ACPI is the firmware-to-OS interface that most PCs use to describe their non-discoverable hardware. It contains tables, a bytecode language called AML, and a long specification. ARM servers sometimes use ACPI, and FreeBSD supports that path on arm64 with the ACPI subsystem, but small and mid-range embedded boards almost always use FDT. A few high-end arm64 systems may ship with *both* ACPI and FDT descriptions and let the firmware pick between them; FreeBSD can handle either, and on arm64 there is a run-time switch that decides which bus to attach. The practical consequence for most embedded drivers is that you write for FDT and not worry about ACPI. We will return to the dual-support case in Section 3.

No PCI-style discovery for on-SoC peripherals. A PCI or PCIe device announces itself through vendor and device IDs in a standardised configuration space. The kernel scans the bus, finds the device, and dispatches to a driver that claims the IDs. On-SoC peripherals on an ARM chip do not have that announcement mechanism. The only way the kernel knows a UART is at address `0x7E201000` on a Raspberry Pi 4 is because the DTB says so. This shifts the driver-author's mental model: you do not wait for the bus to hand you a probed device; you wait for simplebus (or the equivalent) to find your node in the tree and dispatch your probe with that node's context.

Power constraints matter. An embedded board may run on a battery, or from a small USB adapter, or from a power-over-Ethernet feed. A driver that leaves a clock running when the device is idle, or that keeps a regulator enabled past the peripheral's needs, hurts the entire system. FreeBSD provides clock and regulator frameworks precisely so drivers can turn things off when appropriate. We will touch on them in Sections 3, 4, and 7.

Limited RAM and storage. Embedded boards commonly have between 256 MB and 8 GB of RAM, and between a few hundred megabytes and a few tens of gigabytes of storage. A driver that allocates lavishly or prints a screenful of debug output to the console on every event will consume resources the system may not have. Write for the constraints you expect to face.

A simpler boot flow, for better or worse. The boot process on an embedded board is usually shorter and less forgiving than on a PC. If the kernel fails to find the root filesystem, you may not have a handy rescue environment. If the DTB is wrong and the right interrupt controller does not attach, the system will hang silently. This is the main reason embedded work benefits from a working debug cable, a serial console, and the habit of making small, testable changes rather than heroic ones.

32.4.5 Boot Flow: From Firmware to Driver Attachment

To make the moving parts concrete, let us walk through what happens from power-on to the first driver attachment on a representative arm64 board. The details vary by board, but the shape is consistent.

1. Power-on: the first CPU core starts executing code from a boot ROM burned into the SoC. This ROM is beyond your control; its job is to load the next stage.
2. The boot ROM reads a stage-one loader from a fixed location (usually the SD card, eMMC, or SPI flash). On Raspberry Pi boards this is the VideoCore firmware; on generic arm64 boards it is usually U-Boot.
3. The stage-one loader does platform bring-up: it configures DRAM, sets up early clocks, initialises a UART for debug output, and loads the next stage. On boards with FreeBSD installed, that next stage is usually the FreeBSD EFI loader (`loader.efi`).
4. The FreeBSD EFI loader reads its configuration from the ESP partition of the boot medium, consults `/boot/loader.conf`, and loads three things: the kernel itself, a DTB from `/boot/dtb/`, and any overlays listed in the `fdt_overlays` tunable from `/boot/dtb/overlays/`.
5. The loader hands control to the kernel along with pointers to the loaded DTB.
6. The kernel starts. Its machine-dependent startup code parses the DTB to find the memory map, the CPU topology, and the root node of the tree. Based on the top-level `compatible` string of the root node, arm64 decides whether to run the FDT path or the ACPI path.
7. On the FDT path, the kernel attaches an `ofwbus` to the tree root and a `simplebus` to the `/soc` node (or the equivalent on that board). Simplebus walks its children and creates a `device_t` for each node with a valid compatible string.
8. For each of those `device_ts`, the kernel runs the usual newbus probe loop. Every driver that has registered with `DRIVER_MODULE(mydrv, simplebus, ...)` gets a chance to probe. The driver whose `probe()` returns the best match wins and has its `attach()` called. The driver then reads properties, allocates resources, and comes up.
9. The attachment process recurses for nested buses (a simple-bus child of `/soc`, an I2C controller with its own child devices, a GPIO bank with its own pin consumers), producing the full device tree visible in `dmesg` and `devinfo -r`.
10. By the time `init` runs, the devices the system needs to boot (UART, SD host, Ethernet, USB, GPIO) have all attached, and userland comes up.

That sequence is the backdrop against which every driver in this chapter operates. When you write an FDT-aware driver, you are writing the code that runs at step 8 for your particular node. The surrounding machinery runs with or without you; the part you control is the reading of your node and the allocation of your resources.

32.4.6 Where to See It for Yourself

The fastest way to see a real FDT-based boot is to run FreeBSD 14.3 on a Raspberry Pi 3 or 4. Images are available from the FreeBSD project in the standard arm64 download area, and the setup is well documented. If you do not have the hardware, the second fastest way is to use QEMU's `virt` platform, which emulates a generic arm64 machine with a small set of FDT-described peripherals. The kernel and loader from a normal FreeBSD arm64 release run inside it. A sample QEMU invocation lives in the lab notes later in this chapter.

A third option, useful even on an amd64 workstation, is to read `.dts` files and the FreeBSD drivers that consume them. Open `/usr/src/sys/contrib/device-tree/src/arm/bcm2835-rpi-b.dts` or `/usr/src/sys/contrib/device-tree/src/arm64/broadcom/bcm2711-rpi-4-b.dts`. Follow the node structure top-down. Notice how the top-level node has a `compatible` property that names the board. Notice how children describe CPUs, memory, the clock controller, the interrupt controller, the peripheral bus. Then open `/usr/src/sys/arm/broadcom/bcm2835/` in the FreeBSD tree and look at the drivers that consume those nodes. You will start to see how the descriptions and the code meet.

32.4.7 Wrapping Up This Section

Section 1 set the scene. Embedded FreeBSD is not an exotic niche inside the project; it is the project on platforms that do not look like PCs. The architecture the embedded work pushes you toward, thinking of hardware as a set of fixed peripherals on a SoC rather than discoverable devices on a bus, is exactly the architecture Device Tree supports. Device Tree is the bridge between the hardware description that the board designer knows and the driver code that the kernel runs. The rest of this chapter is about learning that bridge well enough to walk across it confidently.

In the next section we will slow down and read Device Tree files themselves. Before we can write a driver that consumes them, we need to know how they are structured, what their properties mean, and how a `.dts` source becomes the `.dtb` binary the kernel actually sees.

32.5 Section 2: What Is the Device Tree?

A Device Tree is a textual description of hardware organised as a tree of nodes, with each node representing a device or a bus, and each node carrying named properties that describe its addresses, its interrupts, its clocks, its relationships to other nodes, and anything else a driver might need to know. That description is what the embedded world uses in place of PCI enumeration or ACPI tables. In the embedded FreeBSD world you will spend your life reading, writing, and reasoning about these files. The sooner they become friendly, the easier every subsequent chapter will feel.

The best way to start is to look at one.

32.5.1 A Minimal Device Tree

Here is the smallest interesting Device Tree source file, in `.dts` form:

```
/dts-v1/;

/ {
    compatible = "acme,trivial-board";
    #address-cells = <1>;
    #size-cells = <1>;

    chosen {
        bootargs = "console=ttyS0,115200";
    };

    memory@80000000 {
        device_type = "memory";
        reg = <0x80000000 0x10000000>;
    };

    uart0: serial@10000000 {
        compatible = "ns16550a";
        reg = <0x10000000 0x100>;
        interrupts = <5>;
        clock-frequency = <24000000>;
        status = "okay";
    };
};
```

That is a complete, valid, if tiny Device Tree. It describes a fictional board called `acme, trivial-board` that has 256 MB of RAM at physical address `0x80000000`, and a 16550-compatible UART at `0x10000000` that delivers interrupt number 5 and runs at 24 MHz. Even if none of the syntax is familiar yet, the intent is legible: the file is a hardware description, written in a compact domain-specific format.

Let us unpack it piece by piece.

32.5.2 The Tree Structure

The first line, `/dts-v1/;`, is a mandatory directive declaring the DTS syntax version. Every FreeBSD DTS file you will ever write or read begins with it. Anything before it is not a valid DTS file; treat it like `#!/bin/sh` at the top of a shell script.

The rest of the file is enclosed in a single block that starts with `/ {` and ends with `};`. That outer block is the **root node** of the tree. Every Device Tree has exactly one root. Its children are peripherals and sub-buses; their children are further peripherals and sub-buses; and so on.

Nodes are identified by **names**. A name like `serial@10000000` consists of a base name (`serial`) and a **unit address** (`10000000`, which is the starting address of the node's first register region written in hexadecimal without the `0x` prefix). The unit address is a convention, not a hard requirement; it exists so that nodes with

the same base name can be distinguished (you can have multiple `serial` nodes at different addresses), and it doubles as a readable hint for what the node describes.

A **label**, like `uart0:` in `uart0: serial@10000000`, is a handle that lets other parts of the tree refer to this node. Labels let you write `&uart0` elsewhere in the file to mean *the node labelled `uart0`*. We will use labels in the overlay section.

Nodes contain **properties**. A property is a name followed by an equals sign, followed by a value, followed by a semicolon:

```
compatible = "ns16550a";
```

Some properties have no value; they exist only as flags:

```
interrupt-controller;
```

The value of a property can be a string ("`ns16550a`"), a list of strings ("`brcm,bcm2711`", "`brcm,bcm2838`"), a list of integers inside angle brackets (`<0x10000000 0x100>`), a list of references to other nodes (`<&gpio0>`), or a binary byte string (`[01 02 03]`). Most everyday properties are strings, string lists, or integer lists. The tree uses 32-bit *cells* as its fundamental integer unit; the integers inside `<...>` are cells, and 64-bit values are expressed as two consecutive cells (high, low).

Let us go back to the minimal example and read each property with fresh eyes.

32.5.3 Reading the Minimal Example

The root node has three properties:

```
compatible = "acme,trivial-board";
#address-cells = <1>;
#size-cells = <1>;
```

The **compatible** property is how the root (and any node) tells the world what it is. It is the single most important property in Device Tree. A driver matches on it. The value is a vendor-prefixed string ("`acme,trivial-board`") or, more commonly, a list of them in decreasing order of specificity. A Raspberry Pi 4 root node, for example, might be `compatible = "raspberrypi,4-model-b", "brcm,bcm2711"`; The first string says “exactly this board”; the second says “in the general family of boards using the BCM2711 chip”. Drivers that know about the specific board can match the first; drivers that only know about the chip can match the second. The DTS spec calls this a *compatibility list*, and both FreeBSD and Linux respect it.

The **#address-cells** and **#size-cells** properties at the root describe how many 32-bit cells a child node uses for its address and size in `reg` properties. At the root of a 32-bit board, both are typically 1. On a 64-bit board with more than 4 GB of addressable memory, they would both be 2. When you see `reg = <0x80000000 0x10000000>`; under the memory node, you know from the parent’s cell counts that this is one address cell and one size cell, which means the region is at `0x80000000` with size `0x10000000`. If `#address-cells` were 2, the region would instead be written `reg = <0x0 0x80000000 0x0 0x10000000>`;

The **chosen** node is a special sibling of the root's hardware children. It carries parameters the bootloader wants to pass to the kernel: the boot arguments, the console device, sometimes the initrd location. FreeBSD reads `/chosen/bootargs` and uses it to populate the kernel environment.

The **memory@80000000** node describes a region of physical memory. Memory nodes carry `device_type = "memory"` and a `reg` property giving their range. FreeBSD's early boot reads these to build its physical memory map.

The **serial@10000000** node is the interesting one. Its `compatible = "ns16550a"` tells the kernel *this is an ns16550a UART*, which is a very common PC-style serial port chip. Its `reg = <0x10000000 0x100>` says *my registers live at physical address 0x10000000 and occupy 0x100 bytes*. Its `interrupts = <5>` says *I deliver interrupt number 5 to my interrupt parent*. Its `clock-frequency = <24000000>` says *my reference clock runs at 24 MHz, so divisors should be computed from this*. Its `status = "okay"` says *I am enabled*; had it said "disabled", the driver would skip this node.

That is essentially what a Device Tree does: it describes each peripheral in a few properties whose meanings are defined by conventions called **bindings**. A binding for a UART tells you what properties a UART node should carry. A binding for an I2C controller tells you what properties an I2C controller node should carry. And so on. The bindings are documented separately, and the FreeBSD tree ships a large library of them under `/usr/src/sys/contrib/device-tree/Bindings/`.

32.5.4 Source vs Binary: .dts, .dtsi, and .dtb

Device Tree files come in three file types that are easy to confuse at first.

.dts is the main source form. A **.dts** file describes a whole board or platform and is what you feed to the compiler.

.dtsi is an include fragment, the *i* standing for *include*. A typical SoC family has a large **.dtsi** describing the SoC itself (its interrupt controller, its clock tree, its on-chip peripherals), and each board that uses that SoC has a small **.dts** that `#includes` the **.dtsi** and then describes the board-specific additions (external devices soldered to the board, pin configuration, chosen node). You will see many **.dtsi** files under `/usr/src/sys/contrib/device-tree/src/arm/` and `/usr/src/sys/contrib/device-tree/src/arm64/`.

.dtb is the compiled binary form. The kernel and bootloader deal with **.dtb** files, not **.dts** files. A **.dtb** is the output of the `dtc` compiler fed a **.dts** source. It is compact, has no whitespace or comments, and is designed to be parsed by a bootloader in a few kilobytes of code. A **.dtb** file for a Raspberry Pi 4 is typically around 30 KB.

And a fourth, less common type:

.dtbo is a compiled *overlay*. Overlays are fragments that modify an existing base **.dtb** at load time: they can enable or disable nodes, add new nodes, or change properties. They are the mechanism FreeBSD and many Linux distributions use to let users customise a stock DTB without rebuilding it. **.dtbo** files are compiled from **.dtso** (device-tree-source-overlay) files, and loaded by the bootloader via the `fdt_overlay` tunable. We will meet them in Section 5.

When you work with DTS you are almost always writing either a **.dts** or a **.dtso**. You compile either with the device tree compiler, `dtc`, which in FreeBSD lives in the `devel/dtc` port and is installed as `/usr/lo-`

cal/bin/dtc. The FreeBSD kernel build system invokes dtc through the scripts under /usr/src/sys/tools/fdt/, specifically make_dtb.sh and make_dtbo.sh.

32.5.5 Nodes, Properties, Addresses, and the Phandle

A few additional concepts recur so often that it is worth naming them once and for all.

Nodes are the hierarchical units. Every node has a name, optionally a label, possibly a unit address, and zero or more properties. Nodes nest; a node's children are described inside its enclosing braces.

Properties are key-value pairs. The keys are strings. The values are typed by convention: compatible is a string list, reg is an integer list, status is a string, interrupts is an integer list whose length depends on the interrupt parent, and so on.

Unit addresses in node names (serial@10000000) mirror the first cell of the reg property. The DTC compiler warns if they disagree. You should write both consistently.

The **reg** property describes the memory-mapped register regions of the device. Its format is <address size address size ...>, with each address-size pair being a contiguous region. Most simple peripherals have exactly one region. Some have several (a peripheral with a main register area and a separate interrupt register block, for example).

Address cells and size cells are the property pair #address-cells and #size-cells that live on a parent node and describe how reg is formatted in its children. A SoC bus with #address-cells = <1>; #size-cells = <1>; lets its children use one cell each for address and size. An I2C bus usually has #address-cells = <1>; #size-cells = <0>; because an I2C child has an address but no size.

Interrupts are described by one or more properties depending on the style used. The older style is interrupts = <...>; with the cells interpreted by the interrupt parent's convention. On ARM GIC-based platforms this is a three-cell form: interrupt type, interrupt number, interrupt flags. The newer style, mixed with the old throughout the kernel, is interrupts-extended = <&gic 0 15 4>; which explicitly names the interrupt parent. Either way, the cells tell the kernel which hardware interrupt the device raises and under what conditions.

A **phandle** is a unique integer assigned by the compiler to each node. Phandles let other nodes refer to this one. When you write <&gpio0>, the compiler substitutes the phandle of the node labelled gpio0. When you write <&gpio0 17 0>, you are passing three cells: the phandle of gpio0, the pin number 17, and a flags cell 0. The meaning of the cells after the phandle is defined by the *provider's* binding. This is the pattern by which GPIO consumers, clock consumers, reset consumers, and interrupt consumers talk to their providers: the first cell names the provider, the following cells say which resource and how.

The **status** property is a small but critical one. A node with status = "okay"; is enabled, and drivers will probe it. A node with status = "disabled"; is skipped. Overlays often toggle this property to turn a peripheral on or off without removing the node. FreeBSD's ofw_bus_status_okay() is the helper that returns true when a node's status is okay.

The **label** and **alias** mechanisms let you refer to a node by a short name rather than a path. A label like uart0: is a file-local handle; an alias defined under the special /aliases node (serial0 = &uart0;) is a kernel-visible name. FreeBSD makes use of aliases for some devices like the console.

That is most of what you need to read a typical Device Tree. A few more exotic pieces appear in specific bindings (for example, clock-names and clocks for consumers of the clock framework, reset-names and resets for

hwreset consumers, dma-names and dmas for DMA engine consumers, pinctrl-0, pinctrl-names for pin control), but they all follow the same *named index list* pattern.

32.5.6 A More Realistic Example

To give you a feel for a real SoC-level fragment, here is an abbreviated node from a BCM2711 (Raspberry Pi 4) description. The full file lives under `/usr/src/sys/contrib/device-tree/src/arm/bcm2711.dtsi`.

```
soc {
    compatible = "simple-bus";
    #address-cells = <1>;
    #size-cells = <1>;
    ranges = <0x7e000000 0x0 0xfe000000 0x01800000>,
            <0x7c000000 0x0 0xfc000000 0x02000000>,
            <0x40000000 0x0 0xff800000 0x00800000>;
    dma-ranges = <0xc0000000 0x0 0x00000000 0x40000000>;

    gpio: gpio@7e200000 {
        compatible = "brcm,bcm2711-gpio", "brcm,bcm2835-gpio";
        reg = <0x7e200000 0xb4>;
        interrupts = <GIC_SPI 113 IRQ_TYPE_LEVEL_HIGH>,
                    <GIC_SPI 114 IRQ_TYPE_LEVEL_HIGH>;
        gpio-controller;
        #gpio-cells = <2>;
        interrupt-controller;
        #interrupt-cells = <2>;
    };

    spi0: spi@7e204000 {
        compatible = "brcm,bcm2835-spi";
        reg = <0x7e204000 0x200>;
        interrupts = <GIC_SPI 118 IRQ_TYPE_LEVEL_HIGH>;
        clocks = <&clocks BCM2835_CLOCK_VPU>;
        #address-cells = <1>;
        #size-cells = <0>;
        status = "disabled";
    };
};
```

Reading this top-down:

- The soc node is the on-chip-peripherals bus. Its `compatible = "simple-bus"` is the magic token that tells FreeBSD to attach the simplebus driver here.
- Its `ranges` property defines the address translation from bus addresses (the “local” addresses the CPU’s peripheral interconnect uses internally, starting at `0x7E000000`) to CPU physical addresses (starting at

0xFE000000). FreeBSD reads this and applies it when mapping child reg properties.

- The `gpio` node is the GPIO controller. It claims two interrupts, declares itself a `gpio-controller` (so other nodes can reference it), and uses two cells per GPIO reference (the first cell is the pin number, the second is a flag word).
- The `spi0` node is the SPI controller at bus address 0x7E204000. It is `status = "disabled"` in the base description, meaning it does not attach until an overlay enables it.

Every embedded board description looks roughly like this: a tree of on-chip peripherals, each with a compatible string identifying what driver to bind, a reg for its memory-mapped registers, an interrupts for its IRQ line, and possibly references to clocks, resets, regulators, and pins.

32.5.7 How the DTB Is Loaded

For completeness, it helps to know how the DTB goes from disk to kernel at boot.

On an arm64 board running FreeBSD 14.3, the typical flow is:

1. The EFI firmware or bootloader reads the FreeBSD EFI loader from the ESP.
2. The FreeBSD loader loads the kernel from `/boot/kernel/kernel` and the DTB from `/boot/dtb/<board>.dtb`. The DTB's filename is selected based on the SoC family.
3. If `/boot/loader.conf` sets `fdt_overlays="overlay1,overlay2"`, the loader reads `/boot/dtb/overlays/overlay1.dtbo` and `/boot/dtb/overlays/overlay2.dtbo`, applies them to the base DTB in memory, and hands the merged result to the kernel.
4. The kernel takes the merged DTB as its authoritative hardware description.

On U-Boot-driven boards (common for armv7) the flow is similar but the loader is U-Boot itself. U-Boot's environment variables `fdt_file` and `fdt_addr` tell it which DTB to load and where to place it. When U-Boot finally does `bootefi` or `booti`, it passes the DTB to the FreeBSD loader or directly to the kernel.

On EFI systems that ship FDT in their firmware (rare for small boards, common for ARM servers that use the Server Base System Architecture), the firmware stores the DTB as an EFI configuration table and the kernel reads it from there.

For a driver author, the details of boot do not matter most of the time. What matters is that by the time your driver's probe is called, the tree has been loaded, parsed, and presented to the kernel; you read it with the same `OF_*` helpers no matter how it got there.

32.5.8 Wrapping Up This Section

Section 2 introduced Device Tree as a hardware-description language. You have seen a minimal example, you have met the core concepts of nodes and properties, you know the difference between `.dts`, `.dtsi`, `.dtb`, and `.dtbo` files, and you have walked through a realistic fragment of a SoC description. You also know how a DTB reaches the kernel at boot.

What you do *not* yet know is how FreeBSD's kernel consumes the tree once it is loaded: which subsystem attaches to it, which helpers drivers call to read properties, and how a driver's `probe()` and `attach()` find their node. That is Section 3.

32.6 Section 3: FreeBSD's Device Tree Support

FreeBSD handles Device Tree through a framework whose design predates FDT itself. The framework is named after Open Firmware, abbreviated **OFW** throughout the source tree, because the API it grew out of originally served PowerPC Macs and IBM systems that used the real Open Firmware specification. When the ARM world standardised on Flattened Device Tree in the late 2000s, FreeBSD mapped FDT onto the same internal API. A driver in FreeBSD 14.3 therefore calls the same `OF_getprop()` whether it is running on a PowerPC Mac, an ARM board with an FDT blob, or a RISC-V board with an FDT blob. The implementation beneath differs; the interface above is uniform.

This section introduces the pieces of that framework you need to know: the `fdt(4)` interface as it is used in practice, the `ofw_bus` helpers that drivers call on top of the raw `OF_*` primitives, the `simplebus(4)` bus driver that enumerates children, and the property-reading idioms you will use constantly. By the end of the section you will know what code already exists on the FreeBSD side and where it lives; Section 4 will then build a driver on top of it.

32.6.1 The `fdt(4)` Framework Overview

`fdt(4)` is the kernel's Flattened Device Tree support. It provides the code that parses the binary `.dtb`, walks it to find nodes, extracts properties, applies overlays, and presents the result through the `OF_*` API. You can think of `fdt(4)` as the lower half and `ofw_bus` as the upper half, with `OF_*` functions spanning both.

The code that implements the FDT side of the OFW interface lives in `/usr/src/sys/dev/ofw/ofw_fdt.c`. It is a specific instance of the `ofw_if.m kobj` interface. When the kernel calls `OF_getprop()`, the call goes through the interface and ends up in the FDT implementation, which walks the flattened blob. On a PowerPC Mac it would end up in the real Open Firmware implementation instead; the drivers above do not need to know or care.

For your purposes as a driver author, you almost never touch `ofw_fdt.c` directly. You use the helpers one layer up.

32.6.2 `OF_*`: The Raw Property Readers

The lowest-level API that drivers call is the `OF_*` family of functions declared in `/usr/src/sys/dev/ofw/openfirm.h`. The ones you will use most are a small set.

`OF_getprop(phandle_t node, const char *prop, void *buf, size_t len)` reads a property's raw bytes into a caller-supplied buffer. It returns the number of bytes read on success, or -1 on failure. The buffer must be large enough for the expected length.

`OF_getencprop(phandle_t node, const char *prop, pcell_t *buf, size_t len)` reads a property whose cells are big-endian and converts them to host byte order as it copies. Almost any property that contains integers should be read with this variant rather than `OF_getprop()`.

`OF_getprop_alloc(phandle_t node, const char *prop, void **buf)` reads a property of unknown length. The kernel allocates a buffer and returns the pointer through the third argument. When you are done, call `OF_prop_free(buf)` to release it.

`OF_hasprop(phandle_t node, const char *prop)` returns non-zero if the named property exists, zero otherwise. Useful for optional properties where the mere presence is meaningful.

`OF_child(phandle_t node)` returns the first child of a node. `OF_peer(phandle_t node)` returns the next sibling. `OF_parent(phandle_t node)` returns the parent. Combined, they let you walk the tree.

`OF_finddevice(const char *path)` returns a phandle for the node at a given path, such as `"/chosen"` or `"/soc/gpio@7e200000"`. Most drivers do not need this because the framework hands them their node already.

`OF_decode_addr(phandle_t dev, int regno, bus_space_tag_t *tag, bus_space_handle_t *hp, bus_size_t *sz)` is a convenience routine used by very early code (serial console drivers, mostly) to set up a bus-space mapping for register `regno` of a given node without going through `newbus`. Normal drivers use `bus_alloc_resource_any()` instead, which reads the `reg` property via the resource list set up during probe.

These primitives are the foundation. In practice you will call them indirectly through the slightly more convenient `ofw_bus_*` helpers, but the ones above are what those helpers use internally, and it is worth recognising them when you read real driver code.

32.6.3 ofw_bus: The Compatibility Helpers

The most common thing an FDT-aware driver does is ask: *is this device compatible with something I know how to drive?* FreeBSD provides a small layer of helpers on top of `OF_getprop` to make those checks idiomatic. They live in `/usr/src/sys/dev/ofw/ofw_bus.h` and `/usr/src/sys/dev/ofw/ofw_bus_subr.h`, with their implementations in `/usr/src/sys/dev/ofw/ofw_bus_subr.c`.

The helpers worth knowing, in the order you will meet them:

`ofw_bus_get_node(device_t dev)` returns the phandle associated with a `device_t`. It is implemented as an inline that calls the parent bus's `OFW_BUS_GET_NODE` method. For a child of `simplebus`, this returns the phandle of the DTS node that gave rise to this device.

`ofw_bus_status_okay(device_t dev)` returns 1 if the node's status property is absent, empty, or "okay"; 0 otherwise. Every FDT-aware probe should call it at the top to skip disabled nodes.

`ofw_bus_is_compatible(device_t dev, const char *string)` returns 1 if any entry of the node's compatible property exactly equals `string`. Short, precise, and the usual tool when a driver only wants one compatible string.

`ofw_bus_search_compatible(device_t dev, const struct ofw_compat_data *table)` walks a driver-provided table and returns the matching entry if any of its compatible strings is in the node's compatible list. This is the standard way a driver that supports multiple chips registers its compatibility. The table is an array of entries, each containing a string and a `uintptr_t` cookie the driver can use to remember which chip matched; the table ends with a sentinel entry whose string is `NULL`. We will see the full pattern in Section 4.

`ofw_bus_has_prop(device_t dev, const char *prop)` is a convenience wrapper over `OF_hasprop(ofw_bus_get_node(dev), prop)`.

`ofw_bus_get_name(device_t dev)`, `ofw_bus_get_compat(device_t dev)`, `ofw_bus_get_type(device_t dev)`, and `ofw_bus_get_model(device_t dev)` return the corresponding strings from the node, or `NULL` if absent.

Those are the bread-and-butter helpers. You will see them at the top of almost every FDT-aware driver's probe and attach routines.

32.6.4 simplebus: The Default Enumerator

The simplebus driver is the piece that makes all of this work in practice. It lives in `/usr/src/sys/dev/fdt/simplebus.c` and its header is at `/usr/src/sys/dev/fdt/simplebus.h`. Simplebus has two jobs.

Its first job is to enumerate children. When the kernel attaches simplebus to a node whose compatible includes "simple-bus" (or whose `device_type` is "soc", for historical reasons), simplebus walks the node's children, creates a `device_t` for each child that has a compatible property, and feeds them into newbus. This is what makes your driver's probe get called at all; simplebus is the bus your driver registers with via `DRIVER_MODULE(mydrv, simplebus, ...)`.

Its second job is to translate child addresses into CPU physical addresses. The parent node's `ranges` property encodes how the bus-local addresses that appear in children's `reg` properties map to CPU physical addresses. Simplebus reads `ranges` in `simplebus_fill_ranges()` and applies it when setting up each child's resource list, so by the time your driver asks for its memory resource, the region is already in CPU physical space.

The central probe code that decides whether simplebus should attach to a given node sits near the top of `/usr/src/sys/dev/fdt/simplebus.c`. Here it is, stripped of comments for brevity:

```
if (!ofw_bus_status_okay(dev))
    return (ENXIO);

if (ofw_bus_is_compatible(dev, "syscon") ||
    ofw_bus_is_compatible(dev, "simple-mfd"))
    return (ENXIO);

if (!(ofw_bus_is_compatible(dev, "simple-bus") &&
    ofw_bus_has_prop(dev, "ranges") &&
    (ofw_bus_get_type(dev) == NULL ||
    strcmp(ofw_bus_get_type(dev), "soc") != 0)))
    return (ENXIO);

device_set_desc(dev, "Flattened device tree simple bus");
return (BUS_PROBE_GENERIC);
```

That snippet is a compact exemplar of the entire probe style. Test status, reject the known exceptions, confirm the node looks like a simple bus, describe the device, and return a probe confidence. Every FDT-aware probe in the tree follows some variation of this shape.

Simplebus registers itself with two parent buses. On the primary ofw root, it registers via `EARLY_DRIVER_MODULE(simplebus, ofwbus, ...)`; and recursively, on itself, via `EARLY_DRIVER_MODULE(simplebus, simplebus, ...)`. The recursion is how nested simple-bus nodes get enumerated: a simplebus parent that encounters a child whose compatible is "simple-bus" attaches another simplebus instance to it, which then enumerates *its* children.

For most driver work you do not need to know more about simplebus than that it exists and that you register with it. Your driver's module registration will read `DRIVER_MODULE(mydrv, simplebus, mydrv_driver, 0, 0);` and everything downstream will happen automatically.

32.6.5 Mapping the Pieces to a Probe Call

To put the moving parts together, let us trace what happens from the DTB being loaded to your driver's probe being called.

1. The loader hands the kernel a DTB.
2. The kernel's early arm64 code parses the DTB to find memory and CPU information.
3. The `ofwbus0` pseudo-device attaches to the tree root.
4. `ofwbus0` creates a `device_t` for the `/soc` node (or whichever node has `compatible = "simple-bus"`) and dispatches the usual newbus probe loop.
5. The simplebus driver's probe runs, returns `BUS_PROBE_GENERIC`, and is picked.
6. Simplebus's attach walks the children of the `/soc` node and creates a `device_t` for each child. Each child's resource list is populated from its reg and interrupts properties, translated through the parent's ranges.
7. For each child, the newbus probe loop runs. Every driver registered to simplebus gets a chance to probe.
8. Your driver's probe is called. It calls `ofw_bus_status_okay()`, `ofw_bus_search_compatible()`, and (if it matches) returns `BUS_PROBE_DEFAULT`.
9. If your driver wins the probe contest for this node, its attach is called. At that point the `device_t` has a resource list already populated with the node's memory and interrupt information.
10. Your driver calls `bus_alloc_resource_any()` for its memory region and for its interrupt, sets up an interrupt handler if it has one, maps the memory, initialises the hardware, and returns 0 to indicate success.

From the driver author's perspective the first six steps are machinery; steps 7 through 10 are the code you write. The chapter will now zoom in on those four steps.

32.6.6 Registering a Driver with `ofw_bus`

When your driver registers with simplebus, it is implicitly opting into the OFW dispatch. The module registration line is:

```
DRIVER_MODULE(mydrv, simplebus, mydrv_driver, 0, 0);
```

That tells newbus: *attach the driver described by `mydrv_driver` as a child of `simplebus`.* Your `device_method_t` array must supply at minimum a `device_probe` and a `device_attach` method. If you have a detach, add `device_detach`. If your driver also implements OFW interface methods (rarely needed at the leaf level), add those too.

On some platforms, and for drivers that want to attach both to simplebus and to the `ofwbus` root (in case there is no simplebus between them and the root), it is common to add a second registration:

```
DRIVER_MODULE(mydrv, ofwbus, mydrv_driver, 0, 0);
```

This is what `gpioled_fdt.c` does, for example. It covers platforms where the `gpio-leds` node sits directly under the root rather than under a `simple-bus`.

32.6.7 Writing a Compatibility Table

A driver that supports more than one variant of a chip typically declares a table of compatible strings:

```
static const struct ofw_compat_data compat_data[] = {
    { "brcm,bcm2711-gpio", 1 },
    { "brcm,bcm2835-gpio", 2 },
    { NULL, 0 }
};
```

Then in `probe`:

```
static int
mydrv_probe(device_t dev)
{
    if (!ofw_bus_status_okay(dev))
        return (ENXIO);

    if (ofw_bus_search_compatible(dev, compat_data) -> ocd_str == NULL)
        return (ENXIO);

    device_set_desc(dev, "My FDT-Aware Driver");
    return (BUS_PROBE_DEFAULT);
}
```

And in `attach`, the matched entry is available for re-lookup:

```
static int
mydrv_attach(device_t dev)
{
    const struct ofw_compat_data *match;
    ...
    match = ofw_bus_search_compatible(dev, compat_data);
    if (match == NULL || match->ocd_str == NULL)
        return (ENXIO);

    sc->variant = match->ocd_data; /* 1 for BCM2711, 2 for BCM2835 */
    ...
}
```

The `ocd_data` field is the cookie you defined in the table. It is a plain `uintptr_t`, so you can use it as an integer discriminator, a pointer to a per-variant structure, or whatever fits your driver's needs.

32.6.8 Reading Properties

Once you have a `device_t`, reading its node's properties is straightforward. A typical pattern:

```
phandle_t node = ofw_bus_get_node(dev);
uint32_t val;

if (OF_getencprop(node, "clock-frequency", &val, sizeof(val)) <= 0) {
    device_printf(dev, "missing clock-frequency\n");
    return (ENXIO);
}
```

The helpers are `OF_getencprop` for integer properties (with endianness handled), `OF_getprop` for raw buffers, `OF_getprop_alloc` for strings of unknown length. For boolean properties (ones whose presence is the signal and whose value is empty), `OF_hasprop` is the idiom:

```
bool want_rts = OF_hasprop(node, "uart-has-rtscs");
```

For list properties, `OF_getprop_alloc` or the fixed-buffer variants let you pull the full list and iterate over it.

32.6.9 Acquiring Resources

Memory and interrupt resources come through the standard `bus_alloc_resource_any()` calls you already know:

```
sc->mem_rid = 0;
sc->mem_res = bus_alloc_resource_any(dev, SYS_RES_MEMORY,
    &sc->mem_rid, RF_ACTIVE);
if (sc->mem_res == NULL) {
    device_printf(dev, "cannot allocate memory\n");
    return (ENXIO);
}

sc->irq_rid = 0;
sc->irq_res = bus_alloc_resource_any(dev, SYS_RES_IRQ,
    &sc->irq_rid, RF_ACTIVE | RF_SHAREABLE);
```

This is possible because `simplebus` has already read your node's `reg` and `interrupts` properties, translated them through the parent's ranges, and stored them in the resource list. The indices 0, 1, 2 refer to the first, second, and third entries of the respective list. A device with multiple `reg` regions will have multiple memory resources at `rids` 0, 1, 2, and so on.

For anything beyond plain memory and interrupts, you reach into the peripheral frameworks.

32.6.10 Peripheral Frameworks: Clock, Regulator, Reset, Pinctrl, GPIO

Embedded peripherals usually need more than just a memory region. They need a clock turned on, a regulator enabled, a reset line de-asserted, perhaps pins multiplexed, and sometimes a GPIO to drive a chip-select or an enable. FreeBSD provides a coherent set of frameworks for each.

Clock framework. Declared in `/usr/src/sys/dev/extres/clock/clock.h`. A consumer calls `clock_get_by_ofw_index(dev, node, idx, &clk)` to acquire a handle to the N-th clock listed in the node's `clocks` property, or `clock_get_by_ofw_name(dev, node, "fck", &clk)` to acquire the clock whose `clock-names` entry is "fck". With a handle in hand, the consumer calls `clock_enable(clk)` to turn it on, `clock_get_freq(clk, &freq)` to ask its rate, and `clock_disable(clk)` to turn it off on shutdown.

Regulator framework. Declared in `/usr/src/sys/dev/extres/regulator/regulator.h`. `regulator_get_by_ofw_property(dev, node, "vdd-supply", &reg)` fetches a regulator by the named property; `regulator_enable(reg)` enables it; `regulator_disable(reg)` disables it.

Hardware-reset framework. Declared in `/usr/src/sys/dev/extres/hwreset/hwreset.h`. `hwreset_get_by_ofw_name(dev, node, "main", &rst)` fetches a reset line; `hwreset_deassert(rst)` brings the peripheral out of reset; `hwreset_assert(rst)` puts it back.

Pin control framework. Declared in `/usr/src/sys/dev/fdt/fdt_pinctrl.h`. `fdt_pinctrl_configure_by_name(dev, "default")` applies the pin configuration associated with the `pinctrl-names = "default"` slot of the node. Most drivers that need pin control simply call this once from attach.

GPIO framework. The consumer side is declared in `/usr/src/sys/dev/gpio/gpiobusvar.h`. `gpio_pin_get_by_ofw_idx(dev, node, idx, &pin)` fetches the N-th GPIO listed in the node's `gpios` property. `gpio_pin_setflags(pin, GPIO_PIN_OUTPUT)` sets its direction. `gpio_pin_set_active(pin, value)` drives its level. `gpio_pin_release(pin)` gives it back.

These frameworks are the reason embedded drivers in FreeBSD are often shorter than their Linux counterparts. You do not have to write the clock tree, the regulator logic, or the GPIO controller: you consume them through a uniform consumer API, and the provider drivers are someone else's problem. The worked example in Section 7 uses the GPIO consumer API end to end.

32.6.11 Interrupt Routing: A Quick Look at interrupt-parent

Interrupts on FDT platforms use a chained lookup scheme. A node's `interrupts` property gives the raw interrupt specifier, and the node's `interrupt-parent` property (or the nearest ancestor's) names the controller that should interpret it. That controller, in turn, may be a child of another controller (a secondary GIC redistributor, a nested PLIC, an I/O APIC-like bridge on some SoCs), which further routes the interrupt upward until it reaches a top-level controller bound to a real CPU vector.

For a driver author, you do not usually have to think about this chain. The kernel's interrupt resource already lives in your resource list as a cookie that the interrupt controller knows how to interpret, and `bus_setup_intr()` hands the cookie back to the controller when you ask for an IRQ. What matters is that your node has the right `interrupts = <...>`; for its immediate interrupt parent, and that the tree's `interrupt-parent` or `interrupts-extended` chain reaches a real controller. When it does not, your interrupt will be silently dropped at boot.

The internals live in `/usr/src/sys/dev/ofw/ofw_bus_subr.c`, with `ofw_bus_lookup_imap()`, `ofw_bus_setup_iinfo()`, and related helpers. You will probably never call them directly unless you are writing a bus driver.

32.6.12 A Brief Look at Overlays

We have mentioned overlays several times. The short version is that an overlay is a small DTB fragment that references nodes in the base tree by label (for example, `&i2c0` or `&gpio0`) and adds or modifies properties or children. The loader merges overlays into the base blob before the kernel sees it. We will return to overlays in Section 5 and use them in the worked example in Section 7; for now just note that FreeBSD supports them through the `fdt_overlays` loader tunable and the files under `/boot/dtb/overlays/`.

32.6.13 ACPI vs FDT on arm64

The arm64 port of FreeBSD supports both FDT and ACPI as discovery mechanisms. Which path the kernel takes is decided during early boot by looking at what the firmware provided. If a DTB was supplied and the top-level `compatible` does not suggest an ACPI path, the kernel attaches the FDT bus. If an ACPI RSDP was supplied and the firmware indicates SBSA compliance, the kernel attaches the ACPI bus. The relevant code is in `/usr/src/sys/arm64/arm64/nexus.c`, which handles both paths; the variable `arm64_bus_method` records which one was chosen.

The practical consequence for driver authors is that a driver written for both mechanisms has to attach to both buses. Drivers that only care about FDT (the majority of small embedded drivers) register with `simplebus` only. Drivers that serve generic hardware that can show up on either ARM servers (ACPI) or ARM embedded boards (FDT) register with both. The `ahci_generic` driver at `/usr/src/sys/dev/ahci/ahci_generic.c` is one such dual-supporting driver; its source is worth reading when you eventually need to write one. For most of this chapter we will stay on the pure FDT side.

32.6.14 Wrapping Up This Section

Section 3 gave you the map of FreeBSD's FDT support. You now know where the core code lives, which helpers to use for which job, and how the pieces connect: `fdt(4)` parses the tree, `OF_*` primitives read properties, `ofw_bus_*` helpers wrap the primitives into idiomatic checks, `simplebus` enumerates the children, and the peripheral frameworks hand you clocks, resets, regulators, pins, and GPIOs.

In the next section we will take those pieces and write a real driver with them. Section 4 walks through the full skeleton of an FDT-aware driver from top to bottom, in enough detail that you could copy the structure into your own project and start filling in the hardware-specific logic.

32.7 Section 4: Writing a Driver for an FDT-Based System

This section walks through the complete shape of an FDT-aware FreeBSD driver. The shape is simple; the reason to lay it out in full is that once you have seen it, every FDT driver in the tree becomes legible. You will start noticing the same patterns in hundreds of files under `/usr/src/sys/dev/` and `/usr/src/sys/arm/`, and each of those drivers becomes one more template you can adapt.

We will build the skeleton in six passes. First the header includes. Then the softc. Then the compatibility table. Then `probe()`. Then `attach()`. Then `detach()` and the module registration. Each pass is short; by the end you will have a complete, compilable minimal driver that prints a message when the kernel matches it to a Device Tree node.

32.7.1 The Header Includes

FDT drivers lean on a handful of headers from the `ofw` and `fdt` directories on top of the usual kernel and bus headers. A typical set:

```
#include <sys/param.h>
#include <sys/system.h>
#include <sys/bus.h>
#include <sys/kernel.h>
#include <sys/module.h>
#include <sys/lock.h>
#include <sys/mutex.h>
#include <sys/rman.h>
#include <sys/resource.h>
#include <sys/malloc.h>

#include <machine/bus.h>
#include <machine/resource.h>

#include <dev/ofw/openfirm.h>
#include <dev/ofw/ofw_bus.h>
#include <dev/ofw/ofw_bus_subr.h>
```

Nothing exotic. The three `ofw` headers bring in the property readers and the compatibility helpers. If your driver is a consumer of GPIOs, clocks, regulators, or `hwreset`, add their headers as well:

```
#include <dev/gpio/gpiobusvar.h>
#include <dev/extres/clock/clock.h>
#include <dev/extres/regulator/regulator.h>
#include <dev/extres/hwreset/hwreset.h>
```

`Pinctrl`:

```
#include <dev/fdt/fdt_pinctrl.h>
```

And `simplebus.h` if your driver actually extends `simplebus` rather than just binding to it as a leaf:

```
#include <dev/fdt/simplebus.h>
```

Most leaf drivers do not need the simplebus header. Bring it in only when you are implementing a bus-like driver that enumerates children.

32.7.2 The Softc

A softc for an FDT-aware driver is a plain softc with a few additional fields to track resources and references you acquired through the OFW helpers:

```
struct mydrv_softc {
    device_t      dev;
    struct resource *mem_res;  /* memory region (bus_alloc_resource) */
    int           mem_rid;
    struct resource *irq_res;  /* interrupt resource, if any */
    int           irq_rid;
    void          *irq_cookie;

    /* FDT-specific state. */
    phandle_t      node;
    uintptr_t      variant;    /* matched ocd_data */

    /* Example: acquired GPIO pin for driving a chip-select. */
    gpio_pin_t     cs_pin;

    /* Example: acquired clock handle. */
    clk_t          clk;

    /* Usual driver state: mutex, buffers, etc. */
    struct mtx      sc_mtx;
};
```

The fields that differ from a PCI or ISA driver are `node`, `variant`, and the consumer handles (`cs_pin`, `clk`, and others). Everything else is standard.

32.7.3 The Compatibility Table

The compatibility table is the driver's claim on a set of Device Tree nodes. It is conventionally declared file-scope and immutable:

```
static const struct ofw_compat_data mydrv_compat_data[] = {
    { "acme,trivial-timer",    1 },
    { "acme,fancy-timer",     2 },
};
```

```
    { NULL,                                0 }  
};
```

The second field, `ocd_data`, is a `uintptr_t` cookie. I like to use it as an integer discriminator (1 for the basic variant, 2 for the fancy variant); you can also use it as a pointer to a per-variant configuration structure. The table ends with a sentinel entry whose first field is `NULL`.

32.7.4 The Probe Routine

A canonical probe for an FDT-aware driver:

```
static int  
mydrv_probe(device_t dev)  
{  
  
    if (!ofw_bus_status_okay(dev))  
        return (ENXIO);  
  
    if (ofw_bus_search_compatible(dev, mydrv_compat_data)->ocd_str == NULL)  
        return (ENXIO);  
  
    device_set_desc(dev, "ACME Trivial Timer");  
    return (BUS_PROBE_DEFAULT);  
}
```

Three lines of logic. First, bail out if the node is disabled. Second, bail out if none of our compatible strings match. Third, set a descriptive name and return `BUS_PROBE_DEFAULT`. The exact return value matters when more than one driver might claim the same node; a more specialised driver can return `BUS_PROBE_SPECIFIC` to outrank a generic one, and a generic fallback can return `BUS_PROBE_GENERIC` to let anything better win. For most drivers, `BUS_PROBE_DEFAULT` is correct.

The `ofw_bus_search_compatible(dev, compat_data)` call returns a pointer to the matching entry, or to the sentinel entry if no match was found. The sentinel's `ocd_str` is `NULL`, so testing for `NULL` is the idiomatic way to say *we did not match anything*. Some drivers alternatively save the returned pointer into a local variable and re-use it; we will do that in `attach`.

32.7.5 The Attach Routine

Attach is where the real work happens. A canonical FDT attach:

```
static int  
mydrv_attach(device_t dev)  
{
```



```
struct mydrv_softc *sc;
const struct ofw_compat_data *match;
phandle_t node;
uint32_t freq;
int err;

sc = device_get_softc(dev);
sc->dev = dev;
sc->node = ofw_bus_get_node(dev);
node = sc->node;

/* Remember which variant we matched. */
match = ofw_bus_search_compatible(dev, mydrv_compat_data);
if (match == NULL || match->oed_str == NULL)
    return (ENXIO);
sc->variant = match->oed_data;

/* Pull any required properties. */
if (OF_getencprop(node, "clock-frequency", &freq, sizeof(freq)) <= 0) {
    device_printf(dev, "missing clock-frequency property\n");
    return (ENXIO);
}

/* Allocate memory and interrupt resources. */
sc->mem_rid = 0;
sc->mem_res = bus_alloc_resource_any(dev, SYS_RES_MEMORY,
    &sc->mem_rid, RF_ACTIVE);
if (sc->mem_res == NULL) {
    device_printf(dev, "cannot allocate memory resource\n");
    err = ENXIO;
    goto fail;
}

sc->irq_rid = 0;
sc->irq_res = bus_alloc_resource_any(dev, SYS_RES_IRQ,
    &sc->irq_rid, RF_ACTIVE | RF_SHAREABLE);
if (sc->irq_res == NULL) {
    device_printf(dev, "cannot allocate IRQ resource\n");
    err = ENXIO;
    goto fail;
}

/* Enable clock, if one is described. */
if (clk_get_by_ofw_index(dev, node, 0, &sc->clk) == 0) {
```

```

    err = clk_enable(sc->clk);
    if (err != 0) {
        device_printf(dev, "could not enable clock: %d\n", err);
        goto fail;
    }
}

/* Apply pinctrl default, if any. */
(void)fdt_pinctrl_configure_by_name(dev, "default");

/* Initialise locks and driver state. */
mtx_init(&sc->sc_mtx, device_get_nameunit(dev), NULL, MTX_DEF);

/* Hook up the interrupt handler. */
err = bus_setup_intr(dev, sc->irq_res,
    INTR_TYPE_MISC | INTR_MPSAFE, NULL, mydrv_intr, sc,
    &sc->irq_cookie);
if (err != 0) {
    device_printf(dev, "could not setup interrupt: %d\n", err);
    goto fail;
}

device_printf(dev, "variant %lu at %s, clock %u Hz\n",
    (unsigned long)sc->variant, device_get_nameunit(dev), freq);

return (0);

fail:
    mydrv_detach(dev);
    return (err);
}

```

That is a lot to unpack. Let us walk it.

1. `device_get_softc(dev)` returns the driver's `softc`, which FreeBSD allocated on your behalf when the driver attached.
2. `ofw_bus_get_node(dev)` returns the phandle for our DT node. We save it on the `softc` because `detach` will also want it.
3. We re-run the `compat` search and record which variant matched.
4. We read a scalar integer property with `OF_getencprop`. The call returns the number of bytes read, -1 on absence, or some smaller number if the property is too short. We treat anything non-positive as failure.
5. We allocate our memory and IRQ resources. Simplebus has already populated the resource list from the node's `reg` and `interrupts`, so indices 0 and 0 are correct.
6. We try to acquire a clock. This driver treats the clock as optional, so a missing `clocks` property is not fatal. If one is present, we enable it.

7. We apply default pin control.
8. We initialise a driver mutex.
9. We set up the interrupt handler, which will dispatch to `mydrv_intr`.
10. We log a message.
11. On any error, we goto a single cleanup path that calls `detach`.

The single cleanup path deserves its own discussion. It is an embedded-friendly pattern because embedded drivers acquire many resources from many different frameworks, and attempting to write out cleanup code at every failure point rapidly becomes unreadable. Instead, write a `detach` that handles partially-initialised softc state, and call it from the failure path. This is the pattern the FreeBSD tree uses consistently; your driver will be easier to read if you follow it.

32.7.6 The Detach Routine

A compliant `detach` tears down everything `attach` might have set up, and does so in the reverse order:

```
static int
mydrv_detach(device_t dev)
{
    struct mydrv_softc *sc;

    sc = device_get_softc(dev);

    if (sc->irq_cookie != NULL) {
        bus_tearardown_intr(dev, sc->irq_res, sc->irq_cookie);
        sc->irq_cookie = NULL;
    }

    if (sc->irq_res != NULL) {
        bus_release_resource(dev, SYS_RES_IRQ, sc->irq_rid,
            sc->irq_res);
        sc->irq_res = NULL;
    }

    if (sc->mem_res != NULL) {
        bus_release_resource(dev, SYS_RES_MEMORY, sc->mem_rid,
            sc->mem_res);
        sc->mem_res = NULL;
    }

    if (sc->clk != NULL) {
        clk_disable(sc->clk);
        clk_release(sc->clk);
        sc->clk = NULL;
    }
}
```

```
if (mtx_initialized(&sc->sc_mtx))
    mtx_destroy(&sc->sc_mtx);

return (0);
}
```

Two things to notice. First, every teardown step is guarded by a check on whether the resource was actually acquired. The check lets detach run correctly from either a normal unload path (all resources were acquired) or a failed-attach path (only some were). Second, the order is the reverse of acquisition. Interrupt handler off before interrupt resource released. Clock disabled before released. Mutex destroyed last.

32.7.7 Interrupt Handler

The interrupt handler is a normal FreeBSD interrupt routine. Nothing is FDT-specific about it:

```
static void
mydrv_intr(void *arg)
{
    struct mydrv_softc *sc = arg;

    mtx_lock(&sc->sc_mtx);
    /* Handle the hardware event. */
    mtx_unlock(&sc->sc_mtx);
}
```

What *is* FDT-specific is the way the interrupt resource was set up in attach. The resource came from the node's `interrupts` property via `simplebus`, which translated it through the interrupt-parent chain so that by the time your driver called `bus_alloc_resource_any(SYS_RES_IRQ, ...)`, the resource already represented a real hardware interrupt at a real controller.

32.7.8 Module Registration

The driver's module registration ties the device methods to the driver and registers the driver with a parent bus:

```
static device_method_t mydrv_methods[] = {
    DEVMETHOD(device_probe, mydrv_probe),
    DEVMETHOD(device_attach, mydrv_attach),
    DEVMETHOD(device_detach, mydrv_detach),

    DEVMETHOD_END
};
```

```
static driver_t mydrv_driver = {
    "mydrv",
    mydrv_methods,
    sizeof(struct mydrv_softc)
};

DRIVER_MODULE(mydrv, simplebus, mydrv_driver, 0, 0);
DRIVER_MODULE(mydrv, ofwbus, mydrv_driver, 0, 0);
MODULE_VERSION(mydrv, 1);
SIMPLEBUS_PNP_INFO(mydrv_compat_data);
```

The two `DRIVER_MODULE` calls register the driver with both `simplebus` and the `ofwbus` root. The latter covers platforms or boards whose node sits directly under the root rather than under an explicit `simple-bus`. `MODULE_VERSION` declares the driver's version for `kldstat` and dependency tracking. `SIMPLEBUS_PNP_INFO` emits a `pnpinfo` descriptor that `kldstat -v` can print; it is a small courtesy to the operator but the driver will work without it.

32.7.9 The Full Skeleton Assembled

Here is the skeleton assembled into a single minimal file that compiles as a kernel module. It does nothing useful; it only demonstrates attachment and logs a message when matched:

```
#include <sys/param.h>
#include <sys/systm.h>
#include <sys/bus.h>
#include <sys/kernel.h>
#include <sys/module.h>
#include <sys/rman.h>

#include <machine/bus.h>
#include <machine/resource.h>

#include <dev/ofw/openfirm.h>
#include <dev/ofw/ofw_bus.h>
#include <dev/ofw/ofw_bus_subr.h>

struct fdthello_softc {
    device_t dev;
    phandle_t node;
};

static const struct ofw_compat_data compat_data[] = {
    { "freebsd,fdthello", 1 },
```

```
    { NULL,                0 }
};

static int
fdthello_probe(device_t dev)
{
    if (!ofw_bus_status_okay(dev))
        return (ENXIO);

    if (ofw_bus_search_compatible(dev, compat_data)->ocd_str == NULL)
        return (ENXIO);

    device_set_desc(dev, "FDT Hello Example");
    return (BUS_PROBE_DEFAULT);
}

static int
fdthello_attach(device_t dev)
{
    struct fdthello_softc *sc;

    sc = device_get_softc(dev);
    sc->dev = dev;
    sc->node = ofw_bus_get_node(dev);

    device_printf(dev, "attached, node phandle 0x%x\n", sc->node);
    return (0);
}

static int
fdthello_detach(device_t dev)
{
    device_printf(dev, "detached\n");
    return (0);
}

static device_method_t fdthello_methods[] = {
    DEVMETHOD(device_probe, fdthello_probe),
    DEVMETHOD(device_attach, fdthello_attach),
    DEVMETHOD(device_detach, fdthello_detach),

    DEVMETHOD_END
};
```

```
static driver_t fdthello_driver = {
    "fdthello",
    fdthello_methods,
    sizeof(struct fdthello_softc)
};

DRIVER_MODULE(fdthello, simplebus, fdthello_driver, 0, 0);
DRIVER_MODULE(fdthello, ofwbus, fdthello_driver, 0, 0);
MODULE_VERSION(fdthello, 1);
```

And its matching Makefile:

```
KMOD= fdthello
SRCS= fdthello.c

SYSDIR?= /usr/src/sys

.include <bsd.kmod.mk>
```

You will find both under `examples/part-07/ch32-fdt-embedded/lab01-fdthello/`. Build them on any FreeBSD system that has the kernel sources installed. The module will only *attach* on a platform that has a Device Tree containing a node with `compatible = "freebsd,fdthello"`, but it will at least compile and load cleanly even on amd64.

32.7.10 What the Kernel Does Next

When you load that module on an arm64 system whose DTB contains a matching node, the sequence is:

1. The module registers `fdthello_driver` with `simplebus` and `ofwbus`.
2. Newbus iterates over every existing device that has `simplebus` or `ofwbus` as a parent and calls the newly registered driver's probe.
3. For each device whose node has `compatible = "freebsd,fdthello"`, the probe returns `BUS_PROBE_DEFAULT`. If no other driver is already attached (or if we outrank it), our attach is called.
4. Attach logs a message; the device is now attached.

When you unload the module, `detach` runs for each attached instance, then the module is unloaded. In the simple case, `kldunload fdthello` cleans up fully.

32.7.11 Checking Your Work

Three quick ways to tell whether your driver matched:

1. **dmesg** should show a line like:

```
fdthello0: <FDT Hello Example> on simplebus0
fdthello0: attached, node phandle 0x8f
```

2. **devinfo -r** should show your device attached somewhere under simplebus.
3. **sysctl dev.fdt.hello.0.parent** should confirm the parent bus.

If your module loads but no devices attach, the probe did not match. The most common causes are a typo in the compatible string, a missing or disabled node, or a node that sits somewhere the simplebus/ofwbus drivers did not reach. Section 6 goes into the debugging in detail.

32.7.12 A Note on Naming and Vendor Prefixes

Real FreeBSD drivers match on compatibility strings like "brcm,bcm2711-gpio", "allwinner,sun50i-a64-mmc", or "st,stm32-uart". The prefix before the comma is the vendor or community name; the rest is the specific chip or family. The convention is widely respected both in upstream Linux and in FreeBSD. When inventing a new compatible for an experiment (as we did above with "freebsd,fdthello"), follow the same vendor-slash-identifier form. Do not invent one-word compatibles; they collide with existing ones and confuse future readers.

32.7.13 Wrapping Up This Section

Section 4 walked through the shape of an FDT driver. You have seen the headers to include, the softc to define, the compatibility table to declare, the probe and attach and detach routines to write, and the module registration that ties it all together. You have a minimal, complete driver that you could compile and load right now. It does not do much, but its structure is the same structure every FDT-aware driver in FreeBSD uses.

In the next section we turn to the other half of the story. A driver is no use without a Device Tree node to match. Section 5 teaches you how to build and modify .dtb files, how FreeBSD's overlay system works, and how to add your own node to an existing board description so that your driver has something to attach to.

32.8 5. Creating and Modifying Device Tree Blobs

You now have a driver that waits patiently for a Device Tree node with `compatible = "freebsd,fdthello"`. Nothing in the running system provides such a node, so nothing probes. In this section we learn how to change that. We will look at the source-to-binary pipeline, at the overlay mechanism that lets us add nodes without rebuilding the whole .dtb, and at the loader tunables that decide which blob the kernel actually sees at boot time.

Creating Device Tree blobs is not a rite of passage for experienced kernel hackers. It is an ordinary editing task. The file is text, the compiler is standard, and the output is a small binary that lives in /boot. What makes it feel unfamiliar is only that very few hobbyist projects encounter it. On embedded FreeBSD it is routine.

32.8.1 The Source-to-Binary Pipeline

Every .dtb that boots a FreeBSD system started life as one or more source files. The pipeline is simple:


```
.dtsi  .dtsi  .dtsi
  \   |   /
  \   |   /
    .dts (top-level source)
      |
      | cpp (C preprocessor)
      v
    .dts (preprocessed)
      |
      | dtc (device tree compiler)
      v
    .dtb (binary blob)
```

The C preprocessor runs first. It expands `#include` directives, macro definitions from header files like `dt-bindings/gpio/gpio.h`, and arithmetic in property expressions. The `dtc` compiler then turns the preprocessed source into the compact flattened format the kernel can parse.

Overlay files go through the same pipeline, except their source file carries the extension `.dtsi` and their output carries `.dtbo`. The only real syntactic difference is the magic incantation at the top of overlay source files, which we look at shortly.

FreeBSD's build system wraps this pipeline in two small shell scripts you can study at `/usr/src/sys/tools/fdt/make_dtb.sh` and `/usr/src/sys/tools/fdt/make_dtbo.sh`. They chain `cpp` and `dtc` together, add the right include paths for the kernel's own `dt-bindings` headers, and write the resulting blobs into the build tree. When you make `buildkernel` for an embedded platform, these scripts are what produce the `.dtb` files that end up in `/boot/dtb/` on the installed system.

32.8.2 Installing the Tools

On FreeBSD, `dtc` is available as a port:

```
# pkg install dtc
```

The package installs the `dtc` binary together with its companion `fdtdump`, which prints the decoded structure of an existing blob. If you plan to do any overlay work, install both. The FreeBSD base tree also ships a copy of `dtc` under `/usr/src/sys/contrib/device-tree/`, but the port version is easier to reach from userspace.

To check the version:

```
$ dtc --version
Version: DTC 1.7.0
```

Any version from 1.6 onwards supports overlays. Earlier versions lack the `/plugin/;` directive, so if you inherit an old build environment, upgrade before going further.

32.8.3 Writing a Standalone .dts File

We will start with a complete, standalone Device Tree source file so the syntax has time to settle before we add overlay complications. Create a file called `tiny.dts` somewhere outside the kernel tree:

```
/dts-v1/;

/ {
    compatible = "example, tiny-board";
    model = "Tiny Example Board";
    #address-cells = <1>;
    #size-cells = <1>;

    chosen {
        bootargs = "-v";
    };

    cpus {
        #address-cells = <1>;
        #size-cells = <0>;

        cpu0: cpu@0 {
            device_type = "cpu";
            reg = <0>;
            compatible = "arm,cortex-a53";
        };
    };

    memory@0 {
        device_type = "memory";
        reg = <0x00000000 0x10000000>;
    };

    soc {
        compatible = "simple-bus";
        #address-cells = <1>;
        #size-cells = <1>;
        ranges;

        hello0: hello@10000 {
            compatible = "freebsd,fdthello";
            reg = <0x10000 0x100>;
            status = "okay";
        };
    };
};
```

```
};
```

The first line, `/dts-v1/;`, tells `dtc` which version of the source format we are using. Version 1 is the only version in current use, but the directive is still required.

After that, we have the root node, containing a handful of expected children. The `cpus` node describes the processor topology, the `memory@0` node declares a 256 MB region of DRAM at physical address zero, and the `soc` node groups on-chip peripherals under a `simple-bus`. Inside `soc`, our `hello@10000` node provides the Device Tree match for the `fdthello` driver we wrote in Section 4.

A few things are worth noticing here even in this small file.

First, `#address-cells` and `#size-cells` reappear inside the `soc` node. The values a parent sets apply only to that parent's direct children, so every level of the tree that cares about addresses has to declare them. Here the `soc` is using one cell for addresses and one for sizes, which is why `reg = <0x10000 0x100>;` inside `hello@10000` lists exactly two `u32` values.

Second, the `ranges` property on the `soc` node is empty. An empty `ranges` means “addresses inside this bus match addresses outside it one to one.” If the `soc` were, say, mapped at a different base address than its children claim, you would use a longer `ranges` list to express the translation.

Third, `status = "okay"` is explicit here. Without it, every tree implicitly defaults to `okay`, but many board files set `status = "disabled"` on optional peripherals and expect overlays or board-specific files to flip them on. Get into the habit of checking this property whenever a driver mysteriously fails to probe.

32.8.4 Compiling a .dts File

Compile the tiny example:

```
$ dtc -I dts -O dtb -o tiny.dtb tiny.dts
```

The `-I dts` flag tells `dtc` the input is textual source, and `-O dtb` requests a binary blob output. A successful compile prints nothing. A syntax error tells you the file and line.

You can verify the result with `fdtdump`:

```
$ fdtdump tiny.dtb | head -30
**** fdtdump is a low-level debugging tool, not meant for general use. ****
    Use the fdtput/fdtget/dtc tools to manipulate .dtb files.

/dts-v1/;
// magic:                0xd00dfeed
// totalsize:            0x214 (532)
// off_dt_struct:        0x38
// off_dt_strings:       0x184
// off_mem_rsvmap:       0x28
// version:              17
```

```
// last_comp_version:    16
// boot_cpuid_phys:      0x0
// boot_cpuid_phys:      0x0
// size_dt_strings:      0x90
// size_dt_strings:      0x90
// size_dt_structs:      0x14c
// size_dt_structs:      0x14c

/ {
    compatible = "example, tiny-board";
    model = "Tiny Example Board";
    ...
}
```

That round-trip confirms the blob is valid and parseable. You could now drop it into a QEMU run with `-dtb tiny.dtb` and the kernel would attempt to boot against it. In practice you rarely hand-write a complete `.dts` for a real board. You start with the vendor's own source file (for instance `/usr/src/sys/contrib/device-tree/src/arm64/broadcom/bcm2711-rpi-4-b.dts` for the Raspberry Pi 4) and modify a subset of nodes with an overlay.

32.8.5 The Role of `.dtsi` Include Files

The `.dtsi` extension is used for Device Tree *includes*. These files contain tree fragments that are meant to be pulled into another `.dts` or `.dtsi`. The compiler treats them identically to `.dts` files, but the filename suffix tells other humans (and the build system) that the file is not standalone.

A common pattern in modern SoC descriptions is:

```
arm/bcm283x.dtsi      <- SoC definition
arm/bcm2710.dtsi      <- Family refinement (Pi 3 lineage)
arm/bcm2710-rpi-3-b.dts <- Specific board top-level file, includes both
```

Each `.dtsi` adds and refines nodes. Labels declared in a lower file can be referenced from a higher file with the `&label` syntax to override properties without rewriting the node. This is the mechanism that makes it possible to support dozens of related boards from a handful of shared SoC descriptions.

32.8.6 Understanding Overlays

A full `.dts` for a real SBC like the Raspberry Pi 4 is dozens of kilobytes long. If you only want to enable SPI, or add a single GPIO-controlled peripheral, rebuilding the whole blob is wasteful and error-prone. The overlay mechanism exists exactly for this situation.

An overlay is a small, special `.dtb` that targets an existing tree. At load time, the FreeBSD loader merges the overlay into the base tree in memory, producing a combined view that the kernel sees as a single Device Tree. The base `.dtb` on disk is never modified. That means the same overlay can enable a feature across several systems with one copy on each.

The syntax of an overlay source file uses two magic directives at the top:

```
/dts-v1/;
/plugin/;
```

After those, the source refers to nodes in the base tree by label. The compiler records the references symbolically, and the loader resolves them at merge time against whatever labels the base tree actually exports. That is why the overlay can be written and compiled independently of the exact base tree it will later be merged with.

Here is a minimal overlay that attaches an `fdthello` node to an existing soc bus:

```
/dts-v1/;
/plugin/;

/ {
    compatible = "brcm,bcm2711";

    fragment@0 {
        target = <&soc>;
        __overlay__ {
            hello@20000 {
                compatible = "freebsd,fdthello";
                reg = <0x20000 0x100>;
                status = "okay";
            };
        };
    };
};
```

The outer `compatible` says this overlay is intended for a BCM2711 tree. The loader uses that string to refuse overlays that do not match the current board. Inside we see a single `fragment@0` node. Each fragment targets one existing node in the base tree through the `target` property. The content under `__overlay__` is the set of properties and children that get merged into that target.

In this example, we are adding a `hello@20000` child under whatever node `&soc` resolves to at merge time. The base tree on a Raspberry Pi 4 declares the `soc` label on the top-level SoC bus node, so after the merge the base soc will gain a new `hello@20000` child, and our driver's probe will fire.

You can also use overlays to *modify* existing nodes. If you set a property with the same name as an existing one, the overlay value replaces the original. If you add a new property, it simply appears. If you add a new child, it is grafted on. The mechanism is additive except for property value replacement.

32.8.7 Compiling and Deploying Overlays

Build the overlay with:

```
$ dtc -I dts -O dtb -@ -o fdthello.dtbo fdthello-overlay.dts
```

The `-@` flag tells the compiler to emit the symbolic label information needed at merge time. Without it, overlays that reference labels fail silently or produce unhelpful errors.

On a running FreeBSD system, overlays live under `/boot/dtb/overlays/`. The filename needs to end in `.dtbo` by convention. The loader looks in `/boot/dtb/overlays` by default; the path can be overridden through loader tunables if you want to stage overlays elsewhere.

To tell the loader which overlays to apply, add a line to `/boot/loader.conf`:

```
fdt_overlays="fdthello,sunxi-i2c1,spigen-rpi4"
```

The value is a comma-separated list of overlay basenames without the `.dtbo` extension. Order matters only when overlays interact with each other. On boot, the loader reads the list, loads each overlay, merges them into the base tree in sequence, and hands the combined blob to the kernel.

A good sanity check is to watch the loader output on a serial console or HDMI screen. When `fdt_overlays` is set, the loader prints a line like:

```
Loading DTB overlay 'fdthello' (0x1200 bytes)
```

If the file is missing or the target does not match, the loader prints a warning and continues. Your driver then fails to probe because the overlay never applied. Checking the loader's console output is the fastest way to catch this kind of failure.

32.8.8 A Walkthrough: Adding a Node to a Raspberry Pi 4 Tree

Let us put the machinery together in a realistic scenario. Imagine you are bringing up a custom daughterboard for the Raspberry Pi 4. It contains a single GPIO-controlled indicator LED on GPIO18 of the Pi's header. You want FreeBSD to drive the LED through your own `edled` driver (which we build in Section 7). You need a Device Tree node.

First, you look up what the Pi 4's base `.dtb` already declares. On a running Pi with FreeBSD installed, `ofwdump -ap | less` or `fdtdump /boot/dtb/broadcom/bcm2711-rpi-4-b.dtb | less` gives you the full tree. Your main interest is the `soc` and `gpio` nodes, where you see a label `gpio = <&gpio>;` exported from the GPIO controller.

Next, you write the overlay source:

```
/dts-v1/;
/plugin/;

/ {
    compatible = "brcm,bcm2711";
```

```
fragment@0 {
    target-path = "/soc";
    __overlay__ {
        edled@0: edled@0 {
            compatible = "example,edled";
            status = "okay";
            leds-gpios = <&gpio 18 0>;
            label = "daughter-indicator";
        };
    };
};
```

We use `target-path` instead of `target` here because the target is an existing path rather than a label. Both forms are valid; `target` takes a phandle reference, `target-path` takes a string.

The `leds-gpios` property is the conventional way to describe a GPIO reference in Device Tree. It is a phandle to the GPIO controller, followed by the GPIO number on that controller, followed by a flag word. Value 0 for the flag means active-high; 1 means active-low. Pinmux usually needs no explicit mention on the Pi because the Broadcom GPIO controller handles both direction and function through the same register set.

Compile and install the overlay:

```
$ dtc -I dts -O dtb -@ -o edled.dtbo edled-overlay.dts
$ sudo cp edled.dtbo /boot/dtb/overlays/
```

Add the overlay to the loader configuration:

```
# echo 'fdt_overlays="edled"' >> /boot/loader.conf
```

Reboot. On the way up, the loader prints its overlay-load line, the base DT gains the new `edled@0` node under `/soc`, the `edled` driver's probe fires, and the daughterboard LED comes up under software control.

32.8.9 Inspecting the Result

Once the kernel is running, three tools verify that everything landed where it should:

```
# ofwdump -p /soc/edled@0
```

prints the properties of the newly added node.

```
# sysctl dev.edled.0.%parent
```

confirms the driver attached and shows its parent bus.

```
# devinfo -r | less
```

displays the full device tree as FreeBSD sees it, with your driver in place.

If any of these disagree with the overlay content, Section 6 helps you diagnose the cause.

32.8.10 Troubleshooting Build Failures

Most DT build errors fall into a small number of categories.

Unresolved references. If the overlay refers to a label like `&gpio` that is not exported by the base tree, the loader prints `no symbol for <label>` and refuses to apply the overlay. Fix by using `target-path` with an absolute path instead, or by rebuilding the base `.dtb` with `-@` to include its symbols.

Syntax errors. These show up as `dtc` errors with a line number. Common culprits are missing semicolons at the end of property assignments, unbalanced braces, and property values that mix unit types (for example, a mix of angle-bracket integers and quoted strings on the same line).

Cell-count mismatches. If the parent declares `#address-cells = <2>` and a child's `reg` gives only one cell, the compiler tolerates it but the kernel parses the value wrong. `ofwdump -p node` and careful reading of the parent's cell counts usually reveal the mismatch.

Duplicate node names. Two nodes at the same level cannot share a name-plus-unit-address. The compiler flags this, but overlays that try to add a new node whose name collides with an existing one produce a cryptic merge failure at boot. Choose unique names or target a different path.

32.8.11 The Kernel's dtb Loading Process

For context, it helps to know what happens to the final merged blob after the loader hands it off.

On arm64 and several other platforms, the loader places the blob at a fixed physical address and passes the pointer to the kernel in its boot argument block. The earliest kernel code, in `/usr/src/sys/arm64/arm64/machdep.c`, validates the magic number and size, maps the blob into kernel virtual memory, and registers it with the FDT subsystem. By the time Newbus begins attaching devices, the blob is fully parsed and the OFW API can walk it.

On amd64 embedded systems (rare but they exist), the flow is similar: UEFI passes the blob through a configuration table, the loader discovers it, and the kernel consumes it through the same FDT API.

The blob is read-only from the kernel's perspective. You never modify it at runtime. If a property value needs to change, the right place to change it is in the source, not in a live tree.

32.8.12 Wrapping Up This Section

Section 5 taught you how to move between Device Tree source and Device Tree binary, how overlays target an existing tree, and how `fdt_overlays` wires the whole thing into the FreeBSD boot process. You can now write a `.dts`, compile it, place the result under `/boot/dtb/overlays/`, list it in `loader.conf`, and watch the kernel pick up your node. The driver you wrote in Section 4 now has something to attach to.

In Section 6 we turn the lens around and look at tools for inspecting what actually landed in the kernel. When things go wrong, as they will, good observation is the shortest path back to a working system.

32.9 6. Testing and Debugging FDT Drivers

Every driver you write will, at some point, fail to probe. The Device Tree will look right in the source, your compatibility string will read correctly, and `kldload` will complete without complaint, but `dmesg` will be silent. Debugging this class of failure is its own skill, and the sooner you build the habit the less time each problem will cost you.

This section covers the tools and techniques for inspecting the running Device Tree, for diagnosing probe failures, for observing attach behavior in detail, and for tracking down unload problems. Much of the material here is applicable to bus drivers, peripheral drivers, and pseudo-devices alike. The one thing that is truly FDT-specific is the set of tools for reading the tree itself.

32.9.1 The `ofwdump(8)` Tool

On a running FreeBSD system, `ofwdump` is your primary window into the Device Tree. It prints nodes and properties from the in-kernel tree, so what it shows is exactly what the drivers see during probe. If the tree is wrong in the kernel, it will be wrong in `ofwdump`, which saves you from compiling and booting again just to check an edit.

The simplest invocation prints the whole tree:

```
# ofwdump -a
```

Pipe through `less` on any non-trivial system; the output runs thousands of lines.

A more focused run dumps one node and its properties:

```
# ofwdump -p /soc/hello@10000
Node 0x123456: /soc/hello@10000
  compatible:  freebsd,fdthello
  reg:         00 01 00 00 00 00 01 00
  status:      okay
```

The `-p` flag prints properties alongside the node name. Integer values come out as byte strings because `ofwdump` cannot, in general, know how many cells a property is supposed to have. You interpret the bytes using the parent's `#address-cells` and `#size-cells`.

To read one specific property:

```
# ofwdump -P compatible /soc/hello@10000
```

Add `-R` to recurse into children of the given node. Add `-S` to print phandles and `-r` to print raw binary if you want to pipe the data into another tool.

Get comfortable with `ofwdump`. When someone says “check the tree,” this is the tool they mean.

32.9.2 Reading the Raw Blob Through sysctl

FreeBSD exposes the unmerged base blob through a sysctl:

```
# sysctl -b hw.fdt.dtb | fdt dump
```

The `-b` flag tells sysctl to print raw binary; piping into `fdt dump` decodes it. This is useful when you suspect an overlay has altered the tree and you want to compare the pre-merge blob with the post-merge view. `ofw dump` shows the post-merge view; `hw.fdt.dtb` shows the pre-merge base.

32.9.3 Confirming FDT Mode on arm64

FreeBSD does not expose a dedicated sysctl that says “you are running on FDT” or “you are running on ACPI.” The decision is made very early in boot by the kernel variable `arm64_bus_method`, and the easiest way to observe it from user space is by looking at the device tree dmesg prints during startup. A machine that chose the FDT path shows a root line like:

```
ofwbus0: <Open Firmware Device Tree>  
simplebus0: <Flattened device tree simple bus> on ofwbus0
```

followed by the rest of the FDT children. A machine that chose the ACPI path shows `acpi0: <...>` instead, and you will never see an `ofwbus0` line.

On a live system you can also run `devinfo -r` and look for `ofwbus0` in the hierarchy, or confirm that the sysctl `hw.fdt.dtb` is present. That sysctl is only registered when a DTB was parsed at boot, so its existence is itself a signal:

```
# sysctl -N hw.fdt.dtb 2>/dev/null && echo "FDT is active" || echo "ACPI or neither"
```

The `-N` flag asks sysctl only for the name, so it succeeds without printing the blob’s bytes.

On boards that support both mechanisms, the mechanism that chooses between them is the loader tunable `kern.cfg.order`. Setting `kern.cfg.order="fdt"` in `/boot/loader.conf` forces the kernel to try FDT first and fall back to ACPI only if no DTB is found; `kern.cfg.order="acpi"` does the opposite. On x86 platforms, `hint.acpi.0.disabled="1"` disables the ACPI attachment entirely and is sometimes useful when the firmware is misbehaving. Section 3 covered this duality in more detail; if you ever find yourself staring at an FDT driver that refuses to attach on an ARM server platform, one of the first things to verify is which bus method the kernel actually chose.

32.9.4 Debugging a Probe That Does Not Fire

The most common symptom is silence: the module loads, `kldstat` shows it, but no device attaches. The probe either never ran or returned `ENXIO`. Walk through the following checklist.

1. Is the node present in the kernel’s tree?

```
# ofwdump -p /soc/your-node
```

If the node is missing, your overlay did not apply. Review the loader's output at boot time. Re-check `/boot/loader.conf` for a `fdt_overlays=` line. Confirm the `.dtbo` file is in `/boot/dtb/overlays/`. Rebuild the overlay if you suspect a stale copy.

2. Is the status property set to okay?

```
# ofwdump -P status /soc/your-node
```

A value of "disabled" keeps the node from being probed. Base board files often declare optional peripherals as disabled and leave it to overlays to enable them.

3. Is the compatible string exactly what the driver expects?

A typo in the overlay or the driver's compat table is the single most common cause of probe failure. Compare them character by character:

```
# ofwdump -P compatible /soc/your-node
```

Against the matching line in the driver:

```
{"freebsd,fdthello", 1},
```

If even the vendor prefix differs (for example `free-bsd,` vs `freebsd,`), no match occurs.

4. Does the parent bus support probing?

FDT drivers attach to `simplebus` or `ofwbus`. If your node's parent is something else (say, an `i2c` bus node), your driver must register with that parent instead. Check the parent by looking one level up in `ofwdump`.

5. Is the driver ranked below another driver that matched?

If a more general driver returned `BUS_PROBE_GENERIC` first, your new driver needs to return something stronger, like `BUS_PROBE_DEFAULT` or `BUS_PROBE_SPECIFIC`. `devinfo -r` shows which driver actually attached.

32.9.5 Adding Temporary Debug Output

When none of the above reveals the cause, add `device_printf` calls to probe and attach to watch the flow directly. In probe:

```
static int
fdthello_probe(device_t dev)
{
    device_printf(dev, "probe: node=%ld compat=%s\n",
        ofw_bus_get_node(dev),
```

```

    ofw_bus_get_compat(dev) ? ofw_bus_get_compat(dev) : "(none)");

    if (!ofw_bus_status_okay(dev)) {
        device_printf(dev, "probe: status not okay\n");
        return (ENXIO);
    }

    if (ofw_bus_search_compatible(dev, compat_data)->oed_data == 0) {
        device_printf(dev, "probe: compat mismatch\n");
        return (ENXIO);
    }

    device_set_desc(dev, "FDT Hello Example");
    return (BUS_PROBE_DEFAULT);
}

```

This prints on every probe call, so expect noise. Remove the prints before shipping. The point is the transient visibility into what ofw_bus_* helpers are returning.

In attach, print the resource rids and addresses you allocated:

```

device_printf(dev, "attach: mem=%#jx size=%#jx\n",
    (uintmax_t)rman_get_start(sc->mem),
    (uintmax_t)rman_get_size(sc->mem));

```

This confirms that bus_alloc_resource_any handed back a valid range. A probe that matches but an attach that crashes here usually means reg in the DT is wrong.

32.9.6 Observing Attach Order and Dependencies

On embedded systems, attach order is not always intuitive. A GPIO-consuming driver has to wait for the GPIO controller to attach first. If your driver tries to acquire a GPIO line before the controller is ready, gpio_pin_get_by_ofw_idx returns ENXIO and your attach fails. FreeBSD handles the ordering through explicit dependencies expressed at driver registration time, and through the interrupt-parent walk for interrupt trees.

Use devinfo -rv to watch the order:

```
# devinfo -rv | grep -E '(gpio|edled|simplebus)'
```

If edled appears before gpio, something in the ordering needs fixing. The usual fix is a MODULE_DEPEND line in the consumer driver:

```
MODULE_DEPEND(edled, gpiobus, 1, 1, 1);
```

This forces gpiobus to load first, ensuring the GPIO controller is available when edled attaches.

32.9.7 Debugging Detach and Unload

Detach debugging is easier than probe debugging because detach runs while the system is up and `printf` output reaches `dmesg` immediately. The two problems you are most likely to meet are:

Unload returns EBUSY. Some resource is still held by the driver. The common cause is a GPIO pin or interrupt handle that was not released. Audit every `_get_` call and make sure there is a matching `_release_` in detach.

Unload succeeds but the module reattaches on next `kldload`. This is almost always because detach left a `softc` field pointing to freed memory, and the second attach followed that pointer. Treat detach as a careful teardown of everything attach built, in reverse order.

A useful trick is to add:

```
device_printf(dev, "detach: entered\n");  
...teardown...  
device_printf(dev, "detach: complete\n");
```

If the second line never appears, something in detach hung or panicked.

32.9.8 Using DTrace for Deeper Visibility

For more sophisticated investigations, DTrace can trace `device_probe` and `device_attach` across the whole kernel without touching the driver source. A one-liner that shows every attach call:

```
# dtrace -n 'fbt::device_attach:entry { printf("%s", stringof(args[0]->softc)); }'
```

The output is noisy during boot, but running it interactively while `kldload`-ing your driver filters it naturally. DTrace usage is beyond the scope of this chapter, but knowing it exists is worth the half-page it would take to set up.

32.9.9 Testing on QEMU

Not every reader has a Raspberry Pi or a BeagleBone to test against. QEMU can emulate an arm64 virt machine, boot FreeBSD on it, and let you load drivers and overlays without any real hardware. The virt machine uses its own Device Tree, which QEMU generates automatically; your overlay can target that tree in exactly the same way it would target a real board. The only caveat is that GPIOs and similar low-level peripherals are limited or absent on the virt machine. For pure DT-and-module experimentation it is perfectly adequate.

The basic invocation looks like this:

```
$ qemu-system-aarch64 \  
  -M virt \  
  -cpu cortex-a72 \  
  -m 2G \  
  -kernel /path/to/kernel \  
  -drive if=virtio,file=disk.img \  
  -serial mon:stdio \  
  -append "console=comconsole"
```

Once the system is up, `kldload` your module and watch for probe messages on the serial console.

32.9.10 When to Stop Debugging and Rebuild

Sometimes a bug is easier to fix by tearing the driver down and rebuilding from a known-good skeleton. The `fdthello` example in Section 4 is exactly that skeleton. If you find yourself chasing a probe failure for more than an hour, copy `fdthello`, rename it, add your `compat` string, and verify the trivial case attaches. Then port the real functionality over one piece at a time. You will almost always find the bug in the process.

32.9.11 Wrapping Up This Section

Section 6 armed you with the tools and habits of an embedded driver debugger. You have `ofwdump` for the tree, `hw.fdt.dtb` for the raw blob, `devinfo -r` for the attached-device view, `MODULE_DEPEND` for ordering, and `device_printf` for ad-hoc visibility. You also have a mental checklist for the common probe and detach failures.

Section 7 now puts all of the chapter's theory into a single worked example: a GPIO-backed LED driver that you build, compile, load, and drive from a `.dts` overlay. If you have worked through this chapter sequentially, the example will feel like a straightforward synthesis of the pieces we have already seen.

32.10 7. Practical Example: GPIO Driver for an Embedded Board

This section walks through the full construction of a small but real driver called `edled` (embedded LED). The driver:

1. Matches a Device Tree node with `compatible = "example,edled"`.
2. Acquires a GPIO pin listed in the node's `leds-gpios` property.
3. Exposes a `sysctl` knob the user can toggle to set the LED state.
4. Releases the GPIO cleanly on detach.

The driver is deliberately minimal. Once it works, you will be able to adapt it to drive anything that sits behind a single GPIO, and the patterns scale up when you need to handle multiple pins, interrupts, or more elaborate peripherals.

32.10.1 What You Need

To follow along you need:

- A FreeBSD system running kernel 14.3 or later.
- The kernel sources installed under `/usr/src`.
- `dtc` from the `devel/dtc` port or similar.
- A board with at least one free GPIO pin and an LED (or you can test the `sysctl` toggle without a real LED; the driver still attaches and logs state changes).

If you are on a Raspberry Pi 4 running FreeBSD, GPIO 18 is a convenient choice because it does not collide with the default console or SD card controller. A Pi 3 or Pi Zero 2 works the same way with adjusted GPIO numbers. On a BeagleBone Black, pick any of the many free pins on the 46-way header.

32.10.2 The Overall File Layout

We will produce five files:

```
edled.c          <- C source
Makefile         <- Kernel module Makefile
edled.dts        <- DT overlay source
edled.dtbo       <- Compiled overlay (output)
README           <- Notes for the reader
```

The corresponding repository layout under the `examples/` tree is:

```
examples/part-07/ch32-fdt-embedded/lab04-edled/
  edled.c
  edled.dts
  Makefile
  README.md
```

You can copy the files out of that tree once you reach the lab section at the end of the chapter.

32.10.3 The Softc

Every driver instance needs a small state block. `edled`'s `softc` holds:

- The device handle itself.
- The GPIO pin descriptor.
- The current on/off state.
- The `sysctl` oid for toggling.

```
struct edled_softc {
    device_t      sc_dev;
    gpio_pin_t    sc_pin;
    int           sc_on;
    struct sysctl_oid *sc_oid;
};
```

`gpio_pin_t` is defined in `/usr/src/sys/dev/gpio/gpiobusvar.h`. It is an opaque handle that carries the GPIO controller reference, the pin number, and the active-high/active-low flag. You never dereference it directly; you pass it to `gpio_pin_setflags`, `gpio_pin_set_active`, and `gpio_pin_release`.

32.10.4 Headers

The top of `edled.c` pulls in the definitions we need:

```
#include <sys/param.h>
#include <sys/systm.h>
#include <sys/kernel.h>
#include <sys/module.h>
#include <sys/malloc.h>
#include <sys/bus.h>
#include <sys/sysctl.h>

#include <machine/bus.h>
#include <machine/resource.h>

#include <dev/ofw/ofw_bus.h>
#include <dev/ofw/ofw_bus_subr.h>
#include <dev/ofw/openfirm.h>

#include <dev/gpio/gpiobusvar.h>
```

Compared with the `fdthello` skeleton in Section 4, we add `<sys/sysctl.h>` for the knob and `<dev/gpio/gpiobusvar.h>` for GPIO consumer APIs.

32.10.5 The Compatibility Table

A tiny table with one entry is all this driver needs:

```
static const struct ofw_compat_data compat_data[] = {
    {"example,edled", 1},
    {NULL, 0}
};
```


A real project would pick a vendor prefix it owns. Using "example," flags the compatible as illustrative. When you ship a product, replace it with your company or project prefix.

32.10.6 The Probe

Probe uses the same pattern as `fdthello`:

```
static int
edled_probe(device_t dev)
{
    if (!ofw_bus_status_okay(dev))
        return (ENXIO);

    if (ofw_bus_search_compatible(dev, compat_data)->ocd_data == 0)
        return (ENXIO);

    device_set_desc(dev, "Example embedded LED");
    return (BUS_PROBE_DEFAULT);
}
```

There is nothing new here. The only reason to copy the probe verbatim from Section 4 is to emphasize how repetitive this step is across drivers; the meaningful differences between drivers almost always live in attach, detach, and the operations layer.

32.10.7 The Attach

Attach is where the real work happens. We allocate and initialize the `softc`, acquire the GPIO pin, configure it as an output, set it to “off,” publish the `sysctl`, and print a confirmation.

```
static int
edled_attach(device_t dev)
{
    struct edled_softc *sc = device_get_softc(dev);
    phandle_t node = ofw_bus_get_node(dev);
    int error;

    sc->sc_dev = dev;
    sc->sc_on = 0;

    error = gpio_pin_get_by_ofw_property(dev, node,
        "leds-gpios", &sc->sc_pin);
    if (error != 0) {
        device_printf(dev, "cannot get GPIO pin: %d\n", error);
        return (error);
    }
}
```

```

    }

    error = gpio_pin_setflags(sc->sc_pin, GPIO_PIN_OUTPUT);
    if (error != 0) {
        device_printf(dev, "cannot set pin flags: %d\n", error);
        gpio_pin_release(sc->sc_pin);
        return (error);
    }

    error = gpio_pin_set_active(sc->sc_pin, 0);
    if (error != 0) {
        device_printf(dev, "cannot set pin state: %d\n", error);
        gpio_pin_release(sc->sc_pin);
        return (error);
    }

    sc->sc_oid = SYSCTL_ADD_PROC(device_get_sysctl_ctx(dev),
        SYSCTL_CHILDREN(device_get_sysctl_tree(dev)),
        OID_AUTO, "state",
        CTLTYPE_INT | CTLFLAG_RW | CTLFLAG_NEEDGIANT,
        sc, 0, edled_sysctl_state, "I", "LED state (0=off, 1=on)");

    device_printf(dev, "attached, GPIO pin acquired, state=0\n");
    return (0);
}

```

There are several things worth examining in this code.

The call `gpio_pin_get_by_ofw_property(dev, node, "leds-gpios", &sc->sc_pin)` parses the `leds-gpios` property from the DT node, resolves the phandle to the GPIO controller, consumes the pin number, and produces a ready-to-use handle. If the controller has not yet attached, this call returns `ENXIO`, which is why we express a `MODULE_DEPEND` on `gpio` during registration.

`gpio_pin_setflags(sc->sc_pin, GPIO_PIN_OUTPUT)` configures the pin direction. Other valid flags include `GPIO_PIN_INPUT`, `GPIO_PIN_PULLUP`, and `GPIO_PIN_PULLDOWN`. You can combine them, for example `GPIO_PIN_INPUT | GPIO_PIN_PULLUP`.

`gpio_pin_set_active(sc->sc_pin, 0)` sets the pin to its inactive level. “Active” here takes polarity into account, so on a pin configured active-low, a value of 1 drives the line low and 0 drives it high. The DT flag cell we discussed earlier is what determines this.

`SYSCTL_ADD_PROC` creates a node at `dev.edled.<unit>.state` whose handler is our own `edled_sysctl_state` function. The `CTLFLAG_NEEDGIANT` flag is appropriate for a small driver that does not yet have proper locking; a production driver would use a dedicated mutex and drop the Giant flag.

If any step fails, we release whatever we have already acquired and return the error. Leaking a GPIO pin on the error path would prevent other drivers from ever using the same line.

32.10.8 The Sysctl Handler

The sysctl handler reads or writes the LED state:

```
static int
edled_sysctl_state(SYSCTL_HANDLER_ARGS)
{
    struct edled_softc *sc = arg1;
    int val = sc->sc_on;
    int error;

    error = sysctl_handle_int(oidp, &val, 0, req);
    if (error != 0 || req->newptr == NULL)
        return (error);

    if (val != 0 && val != 1)
        return (EINVAL);

    error = gpio_pin_set_active(sc->sc_pin, val);
    if (error == 0)
        sc->sc_on = val;
    return (error);
}
```

SYSCTL_HANDLER_ARGS expands to the standard sysctl handler signature. We read the current value into a local, call `sysctl_handle_int` to do the user-space copy, and, if the user supplied a new value, we sanity-check it and apply it through the GPIO API. The current state is kept in the `softc` so a read without a write returns the last value we set.

32.10.9 The Detach

Detach must release everything attach acquired, in reverse order:

```
static int
edled_detach(device_t dev)
{
    struct edled_softc *sc = device_get_softc(dev);

    if (sc->sc_pin != NULL) {
        (void)gpio_pin_set_active(sc->sc_pin, 0);
        gpio_pin_release(sc->sc_pin);
        sc->sc_pin = NULL;
    }
    device_printf(dev, "detached\n");
}
```

```
    return (0);
}
```

We turn the LED off before releasing the pin. Leaving it on across a module unload is rude to the next driver; worse, the pin is released while asserted, and whatever it is driving stays on until something else reclaims the line. The sysctl context is owned by the newbus system through `device_get_sysctl_ctx`, so we do not free the oid explicitly; newbus tears it down for us.

32.10.10 The Method Table and Driver Registration

Nothing surprising here:

```
static device_method_t edled_methods[] = {
    DEVMETHOD(device_probe,  edled_probe),
    DEVMETHOD(device_attach, edled_attach),
    DEVMETHOD(device_detach, edled_detach),
    DEVMETHOD_END
};

static driver_t edled_driver = {
    "edled",
    edled_methods,
    sizeof(struct edled_softc)
};

DRIVER_MODULE(edled, simplebus, edled_driver, 0, 0);
DRIVER_MODULE(edled, ofwbus,    edled_driver, 0, 0);
MODULE_DEPEND(edled, gpiobus, 1, 1, 1);
MODULE_VERSION(edled, 1);
```

The only addition compared with `fdthello` is `MODULE_DEPEND(edled, gpiobus, 1, 1, 1)`. The three integer arguments are the minimum, preferred, and maximum version of `gpiobus` that `edled` can tolerate. A value triplet of 1, 1, 1 means “any version at or above 1.” In practice this is almost always what you want.

32.10.11 The Full Source

Putting it all together:

```
/*
 * edled.c - Example Embedded LED Driver
 *
 * Demonstrates a minimal FDT-driven GPIO consumer on FreeBSD 14.3.
 */
```

```
#include <sys/param.h>
#include <sys/system.h>
#include <sys/kernel.h>
#include <sys/module.h>
#include <sys/malloc.h>
#include <sys/bus.h>
#include <sys/sysctl.h>

#include <machine/bus.h>
#include <machine/resource.h>

#include <dev/ofw/ofw_bus.h>
#include <dev/ofw/ofw_bus_subr.h>
#include <dev/ofw/openfirm.h>

#include <dev/gpio/gpiobusvar.h>

struct edled_softc {
    device_t      sc_dev;
    gpio_pin_t    sc_pin;
    int           sc_on;
    struct sysctl_oid *sc_oid;
};

static const struct ofw_compat_data compat_data[] = {
    {"example,edled", 1},
    {NULL, 0}
};

static int edled_sysctl_state(SYSCTL_HANDLER_ARGS);

static int
edled_probe(device_t dev)
{
    if (!ofw_bus_status_okay(dev))
        return (ENXIO);
    if (ofw_bus_search_compatible(dev, compat_data)->oed_data == 0)
        return (ENXIO);
    device_set_desc(dev, "Example embedded LED");
    return (BUS_PROBE_DEFAULT);
}

static int
```

```
edled_attach(device_t dev)
{
    struct edled_softc *sc = device_get_softc(dev);
    phandle_t node = ofw_bus_get_node(dev);
    int error;

    sc->sc_dev = dev;
    sc->sc_on = 0;

    error = gpio_pin_get_by_ofw_property(dev, node,
        "leds-gpios", &sc->sc_pin);
    if (error != 0) {
        device_printf(dev, "cannot get GPIO pin: %d\n", error);
        return (error);
    }

    error = gpio_pin_setflags(sc->sc_pin, GPIO_PIN_OUTPUT);
    if (error != 0) {
        device_printf(dev, "cannot set pin flags: %d\n", error);
        gpio_pin_release(sc->sc_pin);
        return (error);
    }

    error = gpio_pin_set_active(sc->sc_pin, 0);
    if (error != 0) {
        device_printf(dev, "cannot set pin state: %d\n", error);
        gpio_pin_release(sc->sc_pin);
        return (error);
    }

    sc->sc_oid = SYSCTL_ADD_PROC(device_get_sysctl_ctx(dev),
        SYSCTL_CHILDREN(device_get_sysctl_tree(dev)),
        OID_AUTO, "state",
        CTLTYPE_INT | CTLFLAG_RW | CTLFLAG_NEEDGIANT,
        sc, 0, edled_sysctl_state, "I", "LED state (0=off, 1=on)");

    device_printf(dev, "attached, GPIO pin acquired, state=0\n");
    return (0);
}

static int
edled_detach(device_t dev)
{
    struct edled_softc *sc = device_get_softc(dev);
```

```
    if (sc->sc_pin != NULL) {
        (void)gpio_pin_set_active(sc->sc_pin, 0);
        gpio_pin_release(sc->sc_pin);
        sc->sc_pin = NULL;
    }
    device_printf(dev, "detached\n");
    return (0);
}

static int
edled_sysctl_state(SYSCTL_HANDLER_ARGS)
{
    struct edled_softc *sc = arg1;
    int val = sc->sc_on;
    int error;

    error = sysctl_handle_int(oidp, &val, 0, req);
    if (error != 0 || req->newptr == NULL)
        return (error);

    if (val != 0 && val != 1)
        return (EINVAL);

    error = gpio_pin_set_active(sc->sc_pin, val);
    if (error == 0)
        sc->sc_on = val;
    return (error);
}

static device_method_t edled_methods[] = {
    DEVMETHOD(device_probe, edled_probe),
    DEVMETHOD(device_attach, edled_attach),
    DEVMETHOD(device_detach, edled_detach),
    DEVMETHOD_END
};

static driver_t edled_driver = {
    "edled",
    edled_methods,
    sizeof(struct edled_softc)
};

DRIVER_MODULE(edled, simplebus, edled_driver, 0, 0);
```

```
DRIVER_MODULE(edled, ofwbus,    edled_driver, 0, 0);
MODULE_DEPEND(edled, gpiobus, 1, 1, 1);
MODULE_VERSION(edled, 1);
```

Around 140 lines of C, including headers and blank lines. That is a working, production-shaped FDT GPIO driver.

32.10.12 The Makefile

As with every kernel module in this book, the Makefile is trivial:

```
KMOD=    edled
SRCS=    edled.c

SYSDIR?= /usr/src/sys

.include <bsd.kmod.mk>
```

`bsd.kmod.mk` handles the rest. Typing `make` in the directory produces `edled.ko` and `edled.ko.debug`.

32.10.13 The Overlay Source

The companion `.dts` overlay looks like this (tuned for a Raspberry Pi 4; adjust for your board):

```
/dts-v1/;
/plugin/;

/ {
    compatible = "brcm,bcm2711";

    fragment@0 {
        target-path = "/soc";
        __overlay__ {
            edled0: edled@0 {
                compatible = "example,edled";
                status = "okay";
                leds-gpios = <&gpio 18 0>;
                label = "lab-indicator";
            };
        };
    };
};
```


Compile it with:

```
$ dtc -I dts -O dtb -@ -o edled.dtbo edled.dts
```

and copy to /boot/dtb/overlays/.

32.10.14 Building and Loading

On the target system, put all four files in a scratch directory, then:

```
$ make
$ sudo cp edled.dtbo /boot/dtb/overlays/
$ sudo sh -c 'echo fdt_overlays=\"edled\" >> /boot/loader.conf'
$ sudo reboot
```

After the reboot, you should see:

```
# dmesg | grep edled
edled0: <Example embedded LED> on simplebus0
edled0: attached, GPIO pin acquired, state=0
```

If you plugged an LED into GPIO 18, it is currently off. Verify with:

```
# sysctl dev.edled.0.state
dev.edled.0.state: 0
```

Turn it on:

```
# sysctl dev.edled.0.state=1
dev.edled.0.state: 0 -> 1
```

The LED lights. Turn it off:

```
# sysctl dev.edled.0.state=0
dev.edled.0.state: 1 -> 0
```

Done. You have a functioning embedded driver end to end: from Device Tree source through kernel module through user-space control.

32.10.15 Inspecting the Resulting Device

A few useful queries to confirm the driver is well integrated:

```
# devinfo -r | grep -A1 simplebus
# sysctl dev.edled.0
# ofwdump -p /soc/edled@0
```

The first shows your driver in the Newbus tree. The second lists all sysctls registered by your driver. The third confirms the DT node has the expected properties.

32.10.16 Pitfalls Worth Calling Out

The following mistakes are classic when writing your first GPIO-consumer driver. Each is easy to make and, once you know to look for it, easy to avoid.

Forgetting to release the pin in detach. `kldunload` succeeds, but the pin stays tied up. The next load reports “pin busy.” Always match every `gpio_pin_get_*` with a `gpio_pin_release` in detach.

Reading the DT property before the parent bus has attached. The `leds-gpios` parse returns `ENXIO`. The fix is to ensure the GPIO controller module loads first, via `MODULE_DEPEND`. During boot this happens automatically because the static kernel has both resident; during hand-loaded experiments with `kldload`, you may need to load `gpiobus` explicitly first.

Getting the active flag wrong. On boards that wire the LED so the GPIO sinks current (LED between 3V3 and the pin), “on” corresponds to a low output. In that case, `leds-gpios = <&gpio 18 1>` is correct and `gpio_pin_set_active(sc->sc_pin, 1)` will drive the pin low, lighting the LED. If the LED behaves inversely, flip the flag.

Sysctls that change state without a lock. This driver uses `CTLFLAG_NEEDGIANT` as a shortcut. In a real driver, you allocate a `struct mtx`, take it in the sysctl handler around the GPIO call, and publish the sysctl without the Giant flag. For a single-GPIO LED it does not matter much in practice, but the pattern is important once you extend the driver to handle interrupts or shared state.

32.10.17 Wrapping Up This Section

Section 7 delivered on the promise of the chapter. You have built a complete FDT-driven GPIO consumer, deployed it through an overlay, loaded it on a running system, and exercised it from user space. The components you used, compat tables, OFW helpers, resource acquisition through consumer frameworks, sysctl registration, newbus registration, are the same components every embedded driver in FreeBSD relies on.

Section 8 looks at how to turn a working driver like `edled` into a robust one. Working is the first milestone. Robust is the one that earns a driver its place in the kernel tree.

32.11 8. Refactoring and Finalizing Your Embedded Driver

The driver in Section 7 works. You load the module, flip a sysctl, and the LED behaves. That is a real accomplishment, and if the goal is a one-off experiment on a bench, you can stop there. For anything more serious, a working driver needs to be turned into a *finished* driver: one that can be read by a stranger, audited by a reviewer, and trusted in a system that stays up for months.

This section walks through the refactoring passes that take edled from working to finished. The changes are not about making it do more. They are about making it right. The same process applies to any driver you write, including drivers you adapt from other projects or port from Linux.

32.11.1 What Refactoring Means Here

“Refactor” is one of those words that often covers whatever the speaker feels like changing. For the purposes of this section, refactoring means:

1. Removing latent bugs that happen not to fire in the happy path.
2. Adding the locking and error paths a production driver requires.
3. Improving names, layout, and comments so the next reader does not need to guess.
4. Moving infrastructure out of attach into helpers when the body has grown too long.

Nothing here changes the external behavior of the driver. The sysctl still reads and writes the same integer, the LED still turns on and off, and the DT binding does not move. What changes is how reliable the driver is when something unexpected happens.

32.11.2 Pass One: Tighten the Attach Error Paths

The original attach function grew a cluster of error handlers that each call `gpio_pin_release` and return. That works, but it duplicates cleanup. A cleaner shape uses a single exit block with labels:

```
static int
edled_attach(device_t dev)
{
    struct edled_softc *sc = device_get_softc(dev);
    phandle_t node = ofw_bus_get_node(dev);
    int error;

    sc->sc_dev = dev;
    sc->sc_on = 0;

    error = gpio_pin_get_by_ofw_property(dev, node,
        "leds-gpios", &sc->sc_pin);
    if (error != 0) {
        device_printf(dev, "cannot get GPIO pin: %d\n", error);
        goto fail;
    }

    error = gpio_pin_setflags(sc->sc_pin, GPIO_PIN_OUTPUT);
    if (error != 0) {
        device_printf(dev, "cannot set pin flags: %d\n", error);
        goto fail;
    }
}
```

```

error = gpio_pin_set_active(sc->sc_pin, 0);
if (error != 0) {
    device_printf(dev, "cannot set pin state: %d\n", error);
    goto fail;
}

sc->sc_oid = SYSCTL_ADD_PROC(device_get_sysctl_ctx(dev),
    SYSCTL_CHILDREN(device_get_sysctl_tree(dev)),
    OID_AUTO, "state",
    CTLTYPE_INT | CTLFLAG_RW,
    sc, 0, edled_sysctl_state, "I", "LED state (0=off, 1=on)");

device_printf(dev, "attached, state=0\n");
return (0);

fail:
if (sc->sc_pin != NULL) {
    gpio_pin_release(sc->sc_pin);
    sc->sc_pin = NULL;
}
return (error);
}

```

The `goto fail` pattern is idiomatic FreeBSD kernel style. It collapses the cleanup logic into one place so it is impossible for a future edit to leak a resource by forgetting one of several identical `release` calls.

32.11.3 Pass Two: Add Proper Locking

`CTLFLAG_NEEDGIANT` was a shortcut. The right approach is a per-softc mutex held around the hardware access:

```

struct edled_softc {
    device_t      sc_dev;
    gpio_pin_t    sc_pin;
    int           sc_on;
    struct sysctl_oid *sc_oid;
    struct mtx     sc_mtx;
};

```

Initialize the mutex in `attach`:

```
mtx_init(&sc->sc_mtx, device_get_nameunit(dev), "edled", MTX_DEF);
```

Destroy it in detach:

```
mtx_destroy(&sc->sc_mtx);
```

Take it in the sysctl handler around the hardware call:

```
static int
edled_sysctl_state(SYSCTL_HANDLER_ARGS)
{
    struct edled_softc *sc = arg1;
    int val, error;

    mtx_lock(&sc->sc_mtx);
    val = sc->sc_on;
    mtx_unlock(&sc->sc_mtx);

    error = sysctl_handle_int(oidp, &val, 0, req);
    if (error != 0 || req->newptr == NULL)
        return (error);

    if (val != 0 && val != 1)
        return (EINVAL);

    mtx_lock(&sc->sc_mtx);
    error = gpio_pin_set_active(sc->sc_pin, val);
    if (error == 0)
        sc->sc_on = val;
    mtx_unlock(&sc->sc_mtx);

    return (error);
}
```

Notice that we drop the mutex around `sysctl_handle_int`. That call may copy data to or from user space, which can sleep, and you do not hold a mutex across a sleep. The value we hand to `sysctl_handle_int` is a local copy, so the drop is safe.

Remove `CTLFLAG_NEEDGIANT` from the `SYSCTL_ADD_PROC` call. With a real lock in place, Giant is no longer needed.

32.11.4 Pass Three: Handle the Power Rail Explicitly

On many real peripherals, the driver is responsible for turning on the power rail and the reference clock before touching the device. FreeBSD provides consumer APIs under `/usr/src/sys/dev/extres/regulator/` and

/usr/src/sys/dev/extres/clk/ for exactly this purpose. Even though a discrete LED does not need a regulator, more serious peripherals (say, a SPI-connected accelerometer) do. To keep edled a useful teaching template, we show how the machinery slots in.

Under a hypothetical DT node:

```
edled0: edled@0 {
    compatible = "example,edled";
    status = "okay";
    leds-gpios = <&gpio 18 0>;
    vled-supply = <&ldo_led>;
    clocks = <&clks 42>;
    label = "lab-indicator";
};
```

Two extra properties: vled-supply references a regulator phandle, and clocks references a clock phandle. Attach picks them up like this:

```
#include <dev/extres/clk/clk.h>
#include <dev/extres/regulator/regulator.h>

struct edled_softc {
    ...
    regulator_t    sc_reg;
    clk_t          sc_clk;
};

...

error = regulator_get_by_ofw_property(dev, node, "vled-supply",
    &sc->sc_reg);
if (error == 0) {
    error = regulator_enable(sc->sc_reg);
    if (error != 0) {
        device_printf(dev, "cannot enable regulator: %d\n",
            error);
        goto fail;
    }
} else if (error != ENOENT) {
    device_printf(dev, "regulator lookup failed: %d\n", error);
    goto fail;
}

error = clk_get_by_ofw_index(dev, node, 0, &sc->sc_clk);
if (error == 0) {
```

```

    error = clk_enable(sc->sc_clk);
    if (error != 0) {
        device_printf(dev, "cannot enable clock: %d\n", error);
        goto fail;
    }
} else if (error != ENOENT) {
    device_printf(dev, "clock lookup failed: %d\n", error);
    goto fail;
}

```

Detach releases them in reverse order:

```

if (sc->sc_clk != NULL) {
    clk_disable(sc->sc_clk);
    clk_release(sc->sc_clk);
}
if (sc->sc_reg != NULL) {
    regulator_disable(sc->sc_reg);
    regulator_release(sc->sc_reg);
}

```

The `ENOENT` check is important. If the DT does not declare a regulator or a clock, `regulator_get_by_ofw_property` and `clk_get_by_ofw_index` return `ENOENT`. A driver that supports multiple boards, some with and some without a dedicated rail, treats `ENOENT` as “not needed here” rather than as a fatal error.

32.11.5 Pass Four: Pinmux Setup

On SoCs where GPIO pins can be repurposed as UART, SPI, I2C, or other functions, the pin multiplexing controller has to be programmed before the driver can use a pin. FreeBSD handles this through the `pinctrl` framework at `/usr/src/sys/dev/fdt/fdt_pinctrl.h`. A Device Tree node requesting a specific configuration uses the `pinctrl-names` and `pinctrl-N` properties:

```

edled0: edled@0 {
    compatible = "example,edled";
    pinctrl-names = "default";
    pinctrl-0 = <&edled_pins>;
    ...
};

&pinctrl {
    edled_pins: edled_pins {
        brcm,pins = <18>;
        brcm,function = <1>; /* GPIO output */
    }
}

```

```
};  
};
```

In attach, call:

```
fdt_pinctrl_configure_by_name(dev, "default");
```

before any pin access. The framework walks the `pinctrl-0` handle, finds the referenced node, and applies its settings through the SoC-specific pinctrl driver.

The LED example does not strictly need pinmux because the Broadcom GPIO driver configures pins as part of `gpio_pin_setflags`, but on OMAP, Allwinner, and many other SoCs it is essential. Include the pattern in your teaching template so readers see where it fits.

32.11.6 Pass Five: Style and Naming Audit

Give the final source a slow read. Things to check:

- **Consistent naming.** All functions start with `edled_`, all fields with `sc_`, all constants in uppercase. A stranger reading the source should never wonder which driver a symbol belongs to.
- **No dead code.** Remove any `device_printf` or stub function that was useful during bring-up and has no production purpose.
- **No magic numbers.** If you write `sc->sc_on = 0` in ten places, define an enum or at least a `#define EDLED_OFF 0`.
- **Short comments only where the code's intent is not obvious.** Attempting to add a docstring to every function tends to clutter FreeBSD sources; brevity is the house style.
- **Correct include order.** By convention, `<sys/param.h>` comes first, followed by other `<sys/...>` headers, then `<machine/...>`, then subsystem-specific headers like `<dev/ofw/...>`.
- **Line length.** Stick to 80-column lines. Long function calls use the FreeBSD indent style for continuation.
- **License header.** Every FreeBSD source file opens with the project's BSD-style license block. For out-of-tree drivers, include your own copyright and license notice.

32.11.7 Pass Six: Static Analysis

Run the compiler with warnings turned up:

```
$ make CFLAGS="-Wall -Wextra -Werror"
```

Fix every warning. Warnings indicate either a real bug or an unclear piece of code. In both cases the fix improves the driver.

Consider running `scan-build`:


```
$ scan-build make
```

scan-build is part of the llvm clang analyzer. It catches null-pointer dereferences and use-after-free bugs the compiler misses.

32.11.8 Pass Seven: Documentation

A driver is not finished until it can be understood without reading the code. Write a one-page README that covers:

- What the driver does.
- Which DT binding it expects.
- Which module dependencies it has.
- Any known limitations or board-specific notes.
- How to build, load, and exercise it.

Include a short man page for the companion-material tree, too. Even a stub `edled(4)` page is valuable; you can refine it later.

32.11.9 Packaging and Distribution

Out-of-tree drivers live in a few canonical places:

- As an unofficial `devel/` port, for users to install on top of FreeBSD.
- As a GitHub repository following the FreeBSD project's conventional layout.
- As a `.tar.gz` archive posted alongside a README and an INSTALL file.

The FreeBSD ports tree welcomes driver packages that are known to be stable. Filing a `devel/edled-kmod` port once the driver has some miles on it is a reasonable goal.

If your driver is general enough to benefit other users, consider contributing it upstream. The review process is careful but positive, and the `freebsd-drivers@freebsd.org` mailing list is the natural starting point.

32.11.10 Reviewing Against Real FreeBSD Drivers

Once `edled` is tight, compare it against `/usr/src/sys/dev/gpio/gpioled_fdt.c`, which is the driver that inspired the example. The real driver is a bit larger because it supports multiple LEDs per parent node, but its overall shape matches yours. Note how it:

- Uses `for (child = OF_child(leds); child != 0; child = OF_peer(child))` to iterate DT children.
- Calls `OF_getprop_alloc` to read a variable-length label string.
- Registers with both `simplebus` and `ofwbus` through `DRIVER_MODULE`.
- Declares its DT binding through `SIMPLEBUS_PNP_INFO` so device-id matching works with `devmatch(8)`.

Reading real drivers in detail after finishing your own is one of the most productive things you can do in this field. You will find techniques you have never seen, and you will recognize patterns you now understand from the inside.

32.11.11 Wrapping Up This Section

Section 8 walked through the finishing passes every driver needs. Error paths tightened, locking corrected, power and clock handling made explicit, pinmux considered, style audited, analysis run, documentation written. What you have now is no longer an experiment; it is a driver you could hand to someone else with a straight face.

At this point the chapter's technical material is complete. The remaining pieces are the hands-on labs that let you run everything yourself, the challenge exercises that stretch what you have learned, a short list of common mistakes to watch for, and the wrap-up that closes the loop back to the broader arc of the book.

32.12 9. Reading Real FreeBSD FDT Drivers

We have built `fdthello` and `edled`, two drivers that exist for the purpose of teaching. They are real in the sense that you can load them on a FreeBSD system and see them attach, but they are small, and they do not carry the accumulated wisdom of drivers that have lived in the tree for years and been touched by dozens of contributors. To complete your apprenticeship as an FDT driver writer, you need to read drivers that did not start as teaching material.

This section picks a few drivers out of `/usr/src/sys` and walks through what they show. The goal is not to have you memorize their source, it is to build the habit of reading real code as a primary learning source. The book's teaching examples will fade from memory within months; real driver reading is a skill you can use for the rest of your career.

32.12.1 `gpioled_fdt.c`: A Close Relative of `edled`

Our `edled` driver was deliberately modeled on `/usr/src/sys/dev/gpio/gpioled_fdt.c`. Reading the real thing with `edled` in mind makes the comparison instructive. The real driver is about 150 lines, almost the same size as ours, but handles several details we chose to simplify.

The driver's compatibility table lists a single entry:

```
static struct ofw_compat_data compat_data[] = {
    {"gpio-leds", 1},
    {NULL,      0}
};
```

Notice that `gpio-leds` is an unprefixed string. This reflects a longstanding community binding that predates the current convention of vendor prefixes. New bindings should always use a prefix, but established ones remain as they are for the sake of compatibility.

The probe is almost identical to ours:

```
static int
gpioled_fdt_probe(device_t dev)
{
```

```

if (!ofw_bus_status_okay(dev))
    return (ENXIO);
if (ofw_bus_search_compatible(dev, compat_data)->oed_data == 0)
    return (ENXIO);
device_set_desc(dev, "OFW GPIO LEDs");
return (BUS_PROBE_DEFAULT);
}

```

The attach function is where the drivers diverge. `gpioled_fdt.c` supports multiple LEDs per DT node, following the `gpio-leds` binding that lists each LED as a child node of a single parent. The pattern is:

```

static int
gpioled_fdt_attach(device_t dev)
{
    struct gpioled_softc *sc = device_get_softc(dev);
    phandle_t leds, child;
    ...

    leds = ofw_bus_get_node(dev);
    sc->sc_dev = dev;
    sc->sc_nleds = 0;

    for (child = OF_child(leds); child != 0; child = OF_peer(child)) {
        if (!OF_hasprop(child, "gpios"))
            continue;
        ...
    }
}

```

`OF_child` and `OF_peer` are the classic walkers for iterating Device Tree children. `OF_child(parent)` returns the first child or zero. `OF_peer(node)` returns the next sibling or zero when you hit the end. This two-line iteration idiom is the backbone of every driver that processes a variable number of sub-entries.

Inside the loop, the driver reads per-LED properties:

```

name = NULL;
len = OF_getprop_alloc(child, "label", (void **)&name);
if (len <= 0) {
    OF_prop_free(name);
    len = OF_getprop_alloc(child, "name", (void **)&name);
}

```

`OF_getprop_alloc` allocates memory for a property and returns the length. The caller is responsible for releasing the buffer with `OF_prop_free`. Notice the fallback: if there is no `label` property, the driver uses the

node's name instead. This kind of graceful fallback is worth noting; it makes drivers more tolerant of binding variations.

Each GPIO is then acquired through a `gpio_pin_get_by_ofw_idx` call with an explicit index of zero, since each LED's own `gpios` property is indexed starting from zero within that child's scope. The driver calls `gpio_pin_setflags(pin, GPIO_PIN_OUTPUT)` and registers each LED with the `led(4)` framework so it shows up as `/dev/led/<name>` in user space.

32.12.2 DRIVER_MODULE Registration

The module registration lines look like this:

```
static driver_t gpioled_driver = {
    "gpioled",
    gpioled_methods,
    sizeof(struct gpioled_softc)
};

DRIVER_MODULE(gpioled, ofwbus,    gpioled_driver, 0, 0);
DRIVER_MODULE(gpioled, simplebus, gpioled_driver, 0, 0);
MODULE_VERSION(gpioled, 1);
MODULE_DEPEND(gpioled, gpiobus, 1, 1, 1);
SIMPLEBUS_PNP_INFO(compat_data);
```

Two additions stand out. `MODULE_DEPEND(gpioled, gpiobus, 1, 1, 1)` we already saw. The new line is `SIMPLEBUS_PNP_INFO(compat_data)`. This macro expands into a set of module metadata that tools like `dev-match(8)` use to decide which driver to auto-load for a given DT node. The argument is the same `compat_data` table the probe uses, so there is only one source of truth.

When you write production-grade drivers, include `SIMPLEBUS_PNP_INFO` so that auto-loading works. Without it, your driver will not be picked up automatically and the user has to add it to `loader.conf` explicitly.

32.12.3 What to Take Away from `gpioled_fdt.c`

Read it alongside `edled` and you will see:

- How to iterate multiple children in a DT node.
- How to fall back between property names.
- How to use `OF_getprop_alloc` and `OF_prop_free` for variable-length strings.
- How to register with the `led(4)` framework as well as with Newbus.
- How to declare PNP info for auto-matching.

These are five patterns that appear over and over in FreeBSD drivers. Having seen them once in a real source file, you will recognize them instantly in the next driver you open.

32.12.4 bcm2835_gpio.c: A Bus Provider

edled and `gpioled_fdt.c` are GPIO consumers. They *use* GPIO pins provided by another driver. The driver that *provides* those pins on the Raspberry Pi is `/usr/src/sys/arm/broadcom/bcm2835/bcm2835_gpio.c`. Reading it shows the other side of the transaction.

The driver's attach does considerably more than ours:

- Allocates the MMIO resource for the GPIO controller's register block.
- Allocates all of the interrupt resources (the BCM2835 routes two interrupt lines per bank).
- Initializes a mutex and a driver data structure that tracks per-pin state.
- Registers a GPIO bus child so consumers can attach to it.
- Registers pinmux functions for all the multiplex-capable pins.

The most important thing to notice, from our perspective as consumers, is the way it exposes itself. Deep in the attach:

```
if ((sc->sc_busdev = gpiobus_attach_bus(dev)) == NULL) {
    device_printf(dev, "could not attach GPIO bus\n");
    return (ENXIO);
}
```

`gpiobus_attach_bus(dev)` is what creates the `gpiobus` instance that consumers subsequently probe against. Without this call, no consumer driver could ever acquire a pin, because there would be no bus to resolve phandles against.

At the bottom of the file, `DEVMETHOD` entries map GPIO bus methods to the driver's own functions:

```
DEVMETHOD(gpio_pin_set,    bcm_gpio_pin_set),
DEVMETHOD(gpio_pin_get,    bcm_gpio_pin_get),
DEVMETHOD(gpio_pin_toggle, bcm_gpio_pin_toggle),
DEVMETHOD(gpio_pin_getcaps, bcm_gpio_pin_getcaps),
DEVMETHOD(gpio_pin_setflags, bcm_gpio_pin_setflags),
```

These are the functions our consumer ends up calling, indirectly, every time it does `gpio_pin_set_active`. The consumer API in `gpiobusvar.h` is a thin layer over this `DEVMETHOD` table.

32.12.5 ofw_iicbus.c: A Bus That Is Both Parent and Child

Many I2C controllers are attached as children of `simplebus` (their DT parent) and then themselves act as the parent of individual I2C device drivers. `/usr/src/sys/dev/iicbus/ofw_iicbus.c` is a good example to skim. It shows how a driver can both:

- Probe and attach on its own DT node like any FDT driver.
- Register its own child devices from the DT children of its node.

The iteration over children uses the same `OF_child/OF_peer` idiom, but for each child it creates a new Newbus device with `device_add_child`, sets up its own OFW metadata, and relies on Newbus to run the probe for a driver that can handle it (for example, a temperature sensor or an EEPROM).

Reading this driver gives you a feel for how bus-and-consumer relationships cascade. The FDT is a tree; so is the Newbus hierarchy. A driver in the middle of the tree plays both parent and child roles.

32.12.6 ofw_bus_subr.c: The Helpers Themselves

When you find yourself constantly looking up what exactly `ofw_bus_search_compatible` does, the answer is in `/usr/src/sys/dev/ofw/ofw_bus_subr.c`. Reading the helpers you call is an underrated way to understand what your driver is really doing.

A short tour of the helpers you will meet most often:

- `ofw_bus_is_compatible(dev, str)` returns true if the node's `compatible` list contains `str`. It iterates through all entries in the compatible list, not just the first.
- `ofw_bus_search_compatible(dev, table)` walks the same compatible list against the entries in a struct `ofw_compat_data` table and returns a pointer to the matching entry (or to a sentinel).
- `ofw_bus_status_okay(dev)` checks the status property. Missing status defaults to okay; "okay" or "ok" is okay; anything else ("disabled", "fail") is not.
- `ofw_bus_has_prop(dev, prop)` tests existence without reading.
- `ofw_bus_parse_xref_list_alloc` and related helpers read phandle reference lists (the format used by clocks, resets, gpios, etc.) and return an allocated array the caller must free.

Reading these helpers confirms that there is nothing magical in the system. They are readable C code that walks the same blob the kernel parsed at boot.

32.12.7 simplebus.c: The Driver That Runs Your Driver

If you want to understand why your FDT driver actually gets probed, open `/usr/src/sys/dev/fdt/simplebus.c`. The probe and attach of `simplebus` itself are short and, once you know what to look for, surprisingly concrete.

`simplebus_probe` checks that the node has `compatible = "simple-bus"` (or is an SoC-class node) and that it has no parent-specific peculiarities. `simplebus_attach` then walks the node's children, creates a new device for each, parses each child's reg and interrupts, and calls `device_probe_and_attach` on the new device. That last call is what kicks off your driver's probe.

The key lines are something like:

```
for (node = OF_child(parent); node > 0; node = OF_peer(node)) {
    ...
    child = simplebus_add_device(bus, node, 0, NULL, -1, NULL);
    if (child == NULL)
        continue;
}
```

This iteration is what turns a tree of DT nodes into a tree of Newbus devices. Every FDT driver in existence enters the system through this loop.

Reading `simplebus.c` demystifies the “why does my driver get called” question. You see, in plain C, exactly how the kernel walks from the blob in memory to a call into your driver’s probe. If you ever need to troubleshoot why your probe is not running, the first step is to instrument this file with `device_printf` at the right spot.

32.12.8 A Survey of Drivers Worth Reading

Beyond the specific drivers above, here is a short list of FDT drivers in `/usr/src/sys` that are worth your time as study subjects. Each is representative of a pattern you are likely to encounter.

- `/usr/src/sys/dev/gpio/gpioiic.c`: A driver that implements an I2C bus on top of GPIO pins. Shows bit-banging patterns.
- `/usr/src/sys/dev/gpio/gpiokeys.c`: Consumes GPIO inputs as a keyboard. Shows interrupt handling from GPIOs.
- `/usr/src/sys/dev/uart/uart_dev_ns8250.c`: A platform-independent UART driver with FDT hooks. Shows how a generic driver can accept FDT attach paths alongside other bus types.
- `/usr/src/sys/dev/sdhci/sdhci_fdt.c`: A large FDT driver for SD host controllers. Shows how production drivers handle clocks, resets, regulators, and pinmux together.
- `/usr/src/sys/arm/allwinner/aw_gpio.c`: A complete modern GPIO controller for the Allwinner SoC family. Worth comparing with `bcm2835_gpio.c` to see two takes on the same problem.
- `/usr/src/sys/arm/freescale/imx/imx_gpio.c`: The i.MX6/7/8 GPIO driver, another well-maintained reference.
- `/usr/src/sys/dev/extres/syscon/syscon.c`: A “system controller” pseudo-bus that exposes shared register blocks to multiple drivers. Useful to see how FreeBSD handles DT patterns that do not fit cleanly into “one node, one driver.”

You do not need to read these cover to cover. A healthy habit is to pick one every week or two, skim the structure, and then focus on whichever small detail catches your interest. Over time, these readings will build a library in your head of real code you have seen work.

32.12.9 Using `grep` as a Study Tool

When you find a new function in a driver you are reading and you are not sure what it does, a good first step is:

```
$ grep -rn "function_name" /usr/src/sys | head
```

This shows every place the function is defined and called. Often the declaration in a header is enough, combined with two or three representative call sites, to understand what the function is for. This beats searching the web, which returns outdated documentation and half-remembered forum posts.

For a particular DT binding, the same trick works:

```
$ grep -rn '"gpio-leds"' /usr/src/sys
```

The output tells you every file that references that compat string, including the driver that implements it, overlays that use it, and tests that exercise it.

32.12.10 Wrapping Up This Section

Section 9 gave you a reading list and a method. Real FreeBSD drivers are the richest resource available, and learning to read them effectively is a skill as important as writing your own. The drivers on the list above show the patterns our teaching examples simplified. They are the ones to reach for when you are stuck, when you need inspiration, and when you want to know how a production-quality driver handles the corner cases your own code has not encountered yet.

The remaining material in the chapter is practical: the labs you can run on your own hardware, the challenge exercises that stretch the teaching driver, the troubleshooting reference, and the closing bridge to the next chapter.

32.13 10. Interrupt Plumbing in FDT-Based Systems

We have mostly treated interrupts as opaque. In this section we open the box and look at how Device Tree describes interrupt connectivity, how FreeBSD's interrupt framework (`intrng`) consumes that description, and how a driver asks for an IRQ that will actually fire when the hardware wants attention.

The reason this subject deserves its own section is that interrupt wiring on modern SoCs can get complicated. Simple platforms have one controller, one set of lines, and a flat assignment. Complex platforms have a root controller, several subsidiary controllers that multiplex wider IRQ sources into narrower outputs, and pin-based controllers (like GPIOs-as-interrupts) whose lines chain through the multiplex tree. A driver writer who understands the chain can debug strange interrupt failures in minutes; one who does not can spend hours checking the wrong things.

32.13.1 The Interrupt Tree

The Device Tree expresses interrupts as a separate logical tree that runs alongside the main address tree. Each node has an interrupt parent (its controller), and the tree ascends through controllers until it reaches the root controller that the CPU actually takes exceptions from.

Three properties describe the tree:

- **interrupts**: The interrupt description for this node. Its format depends on the controller it attaches to.
- **interrupt-parent**: A phandle to the controller, if it is not the closest ancestor that is already an interrupt controller.
- **interrupt-controller**: An empty-valued property that marks a node as a controller. A consumer's `interrupt-parent` must point at such a node.

An example fragment:


```

&soc {
    gic: interrupt-controller@10000 {
        compatible = "arm,gic-v3";
        interrupt-controller;
        #interrupt-cells = <3>;
        reg = <0x10000 0x1000>, <0x11000 0x20000>;
    };

    uart0: serial@20000 {
        compatible = "arm,pl011";
        reg = <0x20000 0x100>;
        interrupts = <0 42 4>;
        interrupt-parent = <&gic>;
    };
};

```

The GIC (Generic Interrupt Controller, the standard arm64 root controller) declares `#interrupt-cells = <3>`. Every device that attaches to it must provide three cells in its `interrupts` property. For a GICv3, the three cells are *type, number, flags*: `<0 42 4>` means “shared peripheral interrupt 42, level-triggered, high.”

If `interrupt-parent` is omitted, the parent is the nearest ancestor with an `interrupt-parent` or with the `interrupt-controller` property set. This chain can be non-obvious when drivers sit several levels deep.

32.13.2 Interrupt-Parent Chaining

Consider a more realistic example. On a BCM2711 (Raspberry Pi 4), the GPIO controller is its own interrupt controller: it aggregates interrupts from individual pins into a handful of outputs that feed into the GIC. A button wired to a GPIO pin appears in the DT like this:

```

&gpio {
    button_pins: button_pins {
        brcm,pins = <23>;
        brcm,function = <0>;          /* GPIO input */
        brcm,pull = <2>;             /* pull-up */
    };
};

button_node: button {
    compatible = "gpio-keys";
    pinctrl-names = "default";
    pinctrl-0 = <&button_pins>;
    key_enter {
        label = "enter";
        linux,code = <28>;
    };
};

```

```

    gpios = <&gpio 23 0>;
    interrupt-parent = <&gpio>;
    interrupts = <23 3>;      /* edge trigger */
};
};

```

Two properties name the parent controller: `gpios = <&gpio ...>` names the GPIO controller as the pin provider, and `interrupt-parent = <&gpio>` names the same controller as the interrupt provider. The two roles are distinct and must be stated independently.

The GPIO controller then aggregates its interrupt lines and reports them to the GIC. Inside the GPIO driver, when an interrupt arrives from the GIC, it identifies which pin fired and dispatches the event to whichever driver registered a handler for that pin's IRQ resource.

When your driver requests an interrupt for this node, FreeBSD walks the chain: button driver asks for IRQ, the GPIO controller's `intrng` logic assigns a virtual IRQ number, and eventually the kernel arranges for the GIC's upstream IRQ to call into the GPIO driver's dispatcher which in turn calls the button driver's handler. You do not write any of this plumbing yourself; you just request the IRQ and handle it.

32.13.3 The `intrng` Framework

FreeBSD's `intrng` (interrupt next-generation) subsystem is what unifies all of this. An interrupt controller implements the `pic_*` methods:

```

static device_method_t gpio_methods[] = {
    ...
    DEVMETHOD(pic_map_intr,      gpio_pic_map_intr),
    DEVMETHOD(pic_setup_intr,    gpio_pic_setup_intr),
    DEVMETHOD(pic_teardown_intr, gpio_pic_teardown_intr),
    DEVMETHOD(pic_enable_intr,   gpio_pic_enable_intr),
    DEVMETHOD(pic_disable_intr,  gpio_pic_disable_intr),
    ...
};

```

`pic_map_intr` is the one that reads the DT property and returns an internal representation of the IRQ. `pic_setup_intr` attaches a handler. The remaining methods control masking and acknowledgement.

A consumer driver never calls these directly. It calls `bus_alloc_resource_any(dev, SYS_RES_IRQ, ...)`, and Newbus, together with the OFW resource code, walks the DT and the `intrng` framework to resolve the IRQ.

32.13.4 Requesting an IRQ in Practice

The full shape of interrupt handling in an FDT driver looks like this:

```

struct driver_softc {
    ...
    struct resource *irq_res;
    void *irq_cookie;
    int irq_rid;
};

static int
driver_attach(device_t dev)
{
    struct driver_softc *sc = device_get_softc(dev);
    int error;

    sc->irq_rid = 0;
    sc->irq_res = bus_alloc_resource_any(dev, SYS_RES_IRQ,
        &sc->irq_rid, RF_ACTIVE);
    if (sc->irq_res == NULL) {
        device_printf(dev, "cannot allocate IRQ\n");
        return (ENXIO);
    }

    error = bus_setup_intr(dev, sc->irq_res,
        INTR_TYPE_MISC | INTR_MPSAFE,
        NULL, driver_intr, sc, &sc->irq_cookie);
    if (error != 0) {
        device_printf(dev, "cannot setup interrupt: %d\n", error);
        bus_release_resource(dev, SYS_RES_IRQ, sc->irq_rid,
            sc->irq_res);
        return (error);
    }
    ...
}

```

The RID for interrupts starts at zero and increments for each IRQ in the node's interrupts list. A node with two IRQs would use RIDs 0 and 1 in succession.

`bus_setup_intr` registers the handler. The fourth argument is a filter function (runs in interrupt context); the fifth is a thread handler (runs in a dedicated kernel thread). You pass `NULL` for the one you are not using. The `INTR_MPSAFE` flag tells the framework that the handler does not require the Giant lock.

Teardown in detach:

```

static int
driver_detach(device_t dev)
{

```

```
struct driver_softc *sc = device_get_softc(dev);

if (sc->irq_cookie != NULL)
    bus_tearardown_intr(dev, sc->irq_res, sc->irq_cookie);
if (sc->irq_res != NULL)
    bus_release_resource(dev, SYS_RES_IRQ, sc->irq_rid,
        sc->irq_res);
return (0);
}
```

Failure to `bus_tearardown_intr` is a classic unload bug: the IRQ stays connected to freed memory, and the next time the kernel panics.

32.13.5 Filter Versus Thread Handlers

The distinction between filter and thread handlers is one of the topics that new kernel developers often find confusing. A short primer helps.

A *filter* runs in the context of the interrupt itself, at high IPL, with strict constraints on what it can call. It cannot sleep, cannot allocate memory, cannot acquire a regular sleep-capable mutex. It can only acquire spin mutexes. Its purpose is to decide whether the interrupt is for this device, acknowledge the hardware condition, and either handle the event trivially or schedule a thread handler to do the rest.

A *thread* handler runs in a dedicated kernel thread. It can sleep, allocate, and take sleep-capable locks. Many drivers do all their work in a thread handler and leave the filter empty.

For a driver as simple as `edled`, we never handle an interrupt. If we extended it to handle a pushbutton, we would start with a thread handler and only introduce a filter when profiling showed it necessary.

32.13.6 Edge Versus Level Triggering

The third cell of a GIC `interrupts` triple is the trigger type. Common values:

- 1: rising edge
- 2: falling edge
- 3: either edge
- 4: active high level
- 8: active low level

GPIO-as-interrupt nodes use a different cell count (typically two) and a similar encoding. The choice matters. Edge-triggered interrupts fire once per transition; level-triggered interrupts keep firing as long as the line is asserted. A driver that acknowledges too late on a level-triggered line can end up in an interrupt storm.

The DT binding documentation for each controller specifies the exact cell count and flag semantics. When in doubt, grep in `/usr/src/sys/contrib/device-tree/Bindings/` for the controller family.

32.13.7 Debugging Interrupts That Do Not Fire

The symptoms of misconfigured interrupts are usually clear: the hardware works the first time, and subsequent interrupts never come; or the system boots but the driver's handler never runs.

Checks, in order:

1. **Does `vmstat -i` show the interrupt being counted?** If yes, the hardware is asserting but the driver is not acknowledging it. Check your filter or thread handler.
2. **Does the DT's interrupts match the controller's expected format?** Cell count and values are the usual culprits.
3. **Is interrupt-parent pointing at the right controller?** If a pin-based controller is the source but the DT says the GIC, the request will fail because the GIC's cell format does not match.
4. **Did `bus_setup_intr` return zero?** If not, read the error code. `EINVAL` usually means the IRQ resource was not fully mapped; `ENOENT` means the IRQ number is not claimed by any controller.

The `intr_event_show` DTrace probe can help in advanced debugging, but the four-step check above catches most problems without DTrace.

32.13.8 Real Example: `gpiokeys.c`

`/usr/src/sys/dev/gpio/gpiokeys.c` is worth reading as a working example of a GPIO-consuming driver that uses interrupts. For each child node, it acquires a pin, configures it as an input, and hooks an interrupt through `gpio_alloc_intr_resource` and `bus_setup_intr`. The filter is very short: it just calls `taskqueue_enqueue` on a work item. The actual key processing runs on the kernel's taskqueue, not in interrupt context.

This is a clean pattern for a small interrupt-driven driver. A filter that just signals, a worker that does the job. When you need to implement something similar for a custom board peripheral, the `gpiokeys` driver is a good template.

32.13.9 Wrapping Up This Section

Section 10 unwrapped the interrupt machinery that our earlier examples kept hidden. You now know how the Device Tree describes interrupt connectivity, how FreeBSD's `intrng` resolves an IRQ request into a concrete handler registration, how filter and thread handlers divide work, and how to debug the class of failures that misconfigured interrupts produce.

The chapter's technical coverage is now truly complete. The labs, exercises, and troubleshooting material follow.

32.14 Hands-on Labs

Nothing in this chapter will stick without actually running it. The labs that follow are arranged in order of difficulty. Lab 1 is a warm-up you can complete on any FreeBSD system with kernel sources, even a generic amd64 laptop in QEMU. Lab 2 introduces overlays, which means you will want an arm64 target, either real

(Raspberry Pi 4, BeagleBone, Pine64) or emulated. Lab 3 is a debugging exercise in which you will deliberately break a DT and learn the symptoms. Lab 4 builds the full ed-led driver and drives an LED through it.

All lab files are published under `examples/part-07/ch32-fdt-embedded/`. Each lab has its own subdirectory with a `README.md` and all the sources you need. The following text is self-contained so you can work directly from the book, but the example tree is there as a safety net when you want to diff your work against a known-good reference.

32.14.1 Lab 1: Build and Load the fdthello Skeleton

Goal: Compile the minimal FDT-aware driver from Section 4, load it on a running FreeBSD system, and confirm it registers with the kernel even when no matching DT node exists.

What you will learn:

- How a kernel module Makefile works.
- How `kldload` and `kldunload` interact with module registration.
- How Newbus runs probes the moment a driver is introduced.

Steps:

1. Create a scratch directory named `lab01-fdthello` on a FreeBSD 14.3 system with kernel sources installed.
2. Save the full `fdthello.c` source listing from Section 4 in that directory.
3. Save a Makefile with the following content:

```
KMOD=    fdthello
SRCS=    fdthello.c

SYSDIR?= /usr/src/sys

.include <bsd.kmod.mk>
```

4. Build the module:

```
$ make
```

A clean build produces `fdthello.ko` and `fdthello.ko.debug` in the current directory.

5. Load the module:

```
# kldload ./fdthello.ko
```

On a system without a matching DT node, no probe succeeds. That is expected. The module is resident, but no `fdthello0` device appears.

6. Verify that the module is loaded:

```
# kldstat -m fdthello
```

7. Unload:

```
# kldunload fdthello
```

Expected outcome:

The build completes without warnings. The module loads and unloads cleanly. `kldstat` shows `fdthello.ko` between the two steps.

If you hit a snag:

- **kldload reports “module not found”:** Make sure you passed `./fdthello.ko` with the leading `./` so `kldload` does not try the system module path.
- **The build fails with “no such file `bsd.kmod.mk`”:** Install `/usr/src` via `pkgbase` or checkout from git.
- **The build fails because kernel symbols are missing:** Confirm `/usr/src/sys` matches the running kernel’s version. A mismatch between running kernel and source tree is the usual cause.

The starter files live at `examples/part-07/ch32-fdt-embedded/lab01-fdthello/`.

32.14.2 Lab 2: Build and Deploy an Overlay

Goal: Add a DT node that matches the `fdthello` driver, deploy it as an overlay on an arm64 FreeBSD board, and watch the driver attach.

What you will learn:

- How to write an overlay source file.
- How `dtc -@` produces overlay-ready `.dtbo` output.
- How the FreeBSD loader applies overlays through `fdt_overlays` in `loader.conf`.
- How to verify that an overlay landed correctly.

Steps:

1. On a running arm64 FreeBSD system (Raspberry Pi 4 is the reference target), install `dtc`:

```
# pkg install dtc
```

2. In a scratch directory, save the following overlay source as `fdthello-overlay.dts`:

```
/dts-v1/;
/plugin/;

/ {
    compatible = "brcm,bcm2711";

    fragment@0 {
        target-path = "/soc";
        __overlay__ {
            hello@20000 {
                compatible = "freebsd,fdthello";
                reg = <0x20000 0x100>;
                status = "okay";
            }
        }
    }
}
```

```

        };
    };
};
};

```

3. Compile the overlay:

```
$ dtc -I dts -O dtb -@ -o fdthello.dtbo fdthello-overlay.dts
```

4. Copy the result to the loader's overlay directory:

```
# cp fdthello.dtbo /boot/dtb/overlays/
```

5. Edit `/boot/loader.conf` (create it if missing) to include:

```
fdt_overlays="fdthello"
```

6. Copy your `fdthello.ko` built in Lab 1 to `/boot/modules/`:

```
# cp /path/to/fdthello.ko /boot/modules/
```

7. Ensure `fdthello_load="YES"` is in `/boot/loader.conf`:

```
fdthello_load="YES"
```

8. Reboot:

```
# reboot
```

9. After reboot, confirm:

```
# dmesg | grep fdthello
fdthello0: <FDT Hello Example> on simplebus0
fdthello0: attached, node phandle 0x...
```

Expected outcome:

The driver attaches at boot and its message shows up in `dmesg`. `ofwdump -p /soc/hello@200000` prints the node properties.

If you hit a snag:

- **Loader prints “error loading overlay”:** Usually the `.dtbo` file is missing or in the wrong directory. Confirm it is under `/boot/dtb/overlays/` and named with a `.dtbo` extension.
- **The driver does not attach:** Use the checklist from Section 6: node present, status okay, compatible exact, parent simplebus.
- **You are on a non-Pi board:** Change the top-level `compatible` in the overlay to match your board's base compatible. `ofwdump -p /` shows the current value.

The starter files live at `examples/part-07/ch32-fdt-embedded/lab02-overlay/`.

32.14.3 Lab 3: Debug a Broken Device Tree

Goal: Given a deliberately broken overlay, identify three distinct failure modes and fix each one.

What you will learn:

- How to use `dtc`, `fdtdump`, and `ofwdump` to read a blob.
- How to correlate the tree content with kernel probe behavior.
- How to use `device_printf` breadcrumbs to diagnose a probe mismatch.

Steps:

1. Copy the following broken overlay into `lab03-broken.dts`:

```
/dts-v1/;
/plugin/;

/ {
    compatible = "brcm,bcm2711";

    fragment@0 {
        target-path = "/soc";
        __overlay__ {
            hello@20000 {
                compatible = "free-bsd,fdthello";
                reg = <0x20000 0x100>;
                status = "disabled";
            };
        };
    };

    fragment@1 {
        target-path = "/soc";
        __overlay__ {
            hello@30000 {
                compatible = "freebsd,fdthello";
                reg = <0x30000>;
                status = "okay";
            };
        };
    };
};
```

2. Compile and install the overlay:

```
$ dtc -I dts -O dtb -@ -o lab03-broken.dtbo lab03-broken.dts
# cp lab03-broken.dtbo /boot/dtb/overlays/
```

3. Edit `/boot/loader.conf` to load this overlay instead of `fdthello`:

```
fdt_overlays="lab03-broken"
```

4. Reboot. Observe:

- No `fdthello0` device attaches.
- `dmesg` may be silent or may show an FDT parsing warning about `hello@300000`.

5. Diagnose. Use the following techniques in order:

a) Compare the compatible strings:

```
# ofwdump -P compatible /soc/hello@200000
# ofwdump -P compatible /soc/hello@300000
```

The first prints `free-bsd, fdthello`, which the driver does not match. The hyphen after `free` is the typo. Fix is to correct the string to `freebsd, fdthello`.

b) Check the status:

```
# ofwdump -P status /soc/hello@200000
```

Returns disabled. Even if the compatible were right, the driver would still skip this node. Fix is to set `status = "okay"`.

c) Check the reg property:

```
# ofwdump -P reg /soc/hello@300000
```

Notice that `reg` has only one cell where the parent expects address plus size. Under parents with `#address-cells = <1>` and `#size-cells = <1>`, `reg` must have two cells. The driver attaches, but if it ever allocates the resource, it will misread the size as whatever garbage follows. Fix is `reg = <0x300000 0x100>;`.

6. Apply the fixes, recompile, reinstall the overlay, and reboot. The driver should attach to one or both of the hello nodes.

Expected outcome:

After all three fixes, `dmesg | grep fdthello` shows two attached devices, `hello@200000` and `hello@300000`, each reported through `simplebus`.

If you hit a snag:

- **ofwdump reports “no such node”:** The overlay did not apply. Check the loader’s output for an overlay load message, check the `.dtbo` is where the loader expects it.
- **Only one hello device attaches:** One of the three bugs is still present.
- **The kernel panics:** You are almost certainly reading past the end of `reg` because its cell count is still wrong. Revert to the known-good overlay while you diagnose.

The starter files live at `examples/part-07/ch32-fdt-embedded/lab03-debug-broken/`.

32.14.4 Lab 4: Build the edled Driver End to End

Goal: Construct the full edled driver from Section 7, compile it, attach it through a DT overlay to GPIO18 on a Raspberry Pi 4, and toggle the LED from user space.

What you will learn:

- How to integrate GPIO resource acquisition into an FDT driver.
- How to expose a sysctl that drives hardware.
- How to verify the driver on a running system using `dmesg`, `sysctl`, `ofwdump`, and `devinfo -r`.

Steps:

1. Wire an LED between GPIO18 (pin 12 on the header) and ground through a 330-ohm resistor. If you do not have physical hardware, you can still proceed; the driver attaches and toggles its logical state, but there is nothing to light up.

2. In a scratch directory, save:

- `edled.c` from the full listing in Section 7.
- `Makefile` with `KMOD=edled`, `SRCS=edled.c`, `SYSDIR?=/usr/src/sys`, and `.include <bsd.kmod.mk>`.
- `edled.dts` overlay source from Section 7.

3. Build the module:

```
$ make
```

4. Compile the overlay:

```
$ dtc -I dts -O dtb -@ -o edled.dtbo edled.dts
```

5. Install:

```
# cp edled.ko /boot/modules/  
# cp edled.dtbo /boot/dtb/overlays/
```

6. Edit `/boot/loader.conf`:

```
edled_load="YES"  
fdt_overlays="edled"
```

7. Reboot.

8. Confirm attach:

```
# dmesg | grep edled  
edled0: <Example embedded LED> on simplebus0  
edled0: attached, GPIO pin acquired, state=0
```

9. Exercise the sysctl:

```
# sysctl dev.edled.0.state
dev.edled.0.state: 0
```

```
# sysctl dev.edled.0.state=1
dev.edled.0.state: 0 -> 1
```

The LED lights. Read back and confirm:

```
# sysctl dev.edled.0.state
dev.edled.0.state: 1
```

10. Turn the LED off and unload the driver:

```
# sysctl dev.edled.0.state=0
# kldunload edled
```

Expected outcome:

The driver loads, attaches, toggles the LED, and unloads without leaving resources in use. `gpioctl -l` shows the pin returned to the unconfigured state after unload.

If you hit a snag:

- **dmesg shows “cannot get GPIO pin”:** The GPIO controller module has not attached yet. Verify `gpio` is loaded: `kldstat -m gpio`. If it is not, `kldload gpio` before retrying.
- **The LED does not light:** Check the polarity. If the flag in the DT is 0 (active high), the pin drives 3.3V when active. If the LED’s cathode goes to the pin, you want 1 (active low).
- **kldunload fails with EBUSY:** Some process still has `dev.edled.0` open or the driver’s detach path left a resource acquired. Audit detach.

The starter files live at `examples/part-07/ch32-fdt-embedded/lab04-edled/`.

32.14.5 Lab 5: Extend edled to Consume a GPIO Interrupt

Goal: Modify the `edled` driver from Lab 4 so that a second GPIO, configured as an input with a pull-up resistor, becomes an interrupt source. When the pin is grounded (a push button pulling it low), the handler toggles the LED.

What you will learn:

- How to acquire a GPIO interrupt resource through `gpio_alloc_intr_resource`.
- How to set up a thread-context handler with `bus_setup_intr`.
- How to coordinate the interrupt path and the `sysctl` path through shared state.
- How to tear down an interrupt handler cleanly in detach.

Steps:

1. Start from `edled.c` from Lab 4. Copy it to `edledi.c` in a fresh scratch directory.
2. Add a second GPIO to the softc and to the DT binding. The new DT node looks like:

```
edledi0: edledi@0 {
    compatible = "example,edledi";
    status = "okay";
    leds-gpios = <&gpio 18 0>;
    button-gpios = <&gpio 23 1>;
    interrupt-parent = <&gpio>;
    interrupts = <23 3>;
};
```

The button uses GPIO 23, wired with the usual pushbutton arrangement: one leg to the pin, the other to ground, with a pull-up to 3.3V.

3. Update the compat string table to "example,edledi" so the driver matches the new binding.
4. In the softc, add:

```
gpio_pin_t      sc_button;
struct resource *sc_irq;
void            *sc_irq_cookie;
int            sc_irq_rid;
```

5. In attach, after acquiring the LED pin, acquire the button pin and its interrupt:

```
error = gpio_pin_get_by_ofw_property(dev, node,
    "button-gpios", &sc->sc_button);
if (error != 0) {
    device_printf(dev, "cannot get button pin: %d\n", error);
    goto fail;
}

error = gpio_pin_setflags(sc->sc_button,
    GPIO_PIN_INPUT | GPIO_PIN_PULLUP);
if (error != 0) {
    device_printf(dev, "cannot configure button: %d\n", error);
    goto fail;
}

sc->sc_irq_rid = 0;
sc->sc_irq = bus_alloc_resource_any(dev, SYS_RES_IRQ,
    &sc->sc_irq_rid, RF_ACTIVE);
if (sc->sc_irq == NULL) {
    device_printf(dev, "cannot allocate IRQ\n");
    goto fail;
}

error = bus_setup_intr(dev, sc->sc_irq,
    INTR_TYPE_MISC | INTR_MPSAFE,
    NULL, edledi_intr, sc, &sc->sc_irq_cookie);
```

```

if (error != 0) {
    device_printf(dev, "cannot setup interrupt: %d\n", error);
    goto fail;
}

```

6. Write the interrupt handler:

```

static void
edledi_intr(void *arg)
{
    struct edled_softc *sc = arg;

    mtx_lock(&sc->sc_mtx);
    sc->sc_on = !sc->sc_on;
    (void)gpio_pin_set_active(sc->sc_pin, sc->sc_on);
    mtx_unlock(&sc->sc_mtx);
}

```

This is a thread handler (passed as the fifth argument to `bus_setup_intr`, with `NULL` as the fourth for the filter). It is safe to take a mutex and to call into the GPIO framework.

7. In detach, add teardown in reverse order:

```

if (sc->sc_irq_cookie != NULL)
    bus_teardown_intr(dev, sc->sc_irq, sc->sc_irq_cookie);
if (sc->sc_irq != NULL)
    bus_release_resource(dev, SYS_RES_IRQ, sc->sc_irq_rid,
        sc->sc_irq);
if (sc->sc_button != NULL)
    gpio_pin_release(sc->sc_button);

```

8. Rebuild, redeploy the overlay, reboot, and press the button.

Expected outcome:

Each press toggles the LED. The `sysctl` still works for programmatic control. The driver unloads cleanly.

If you hit a snag:

- **Interrupt never fires:** Confirm the pull-up is actually pulling the pin high when idle; check the DT trigger cell (3 = either edge); look at `vmstat -i` to see if any IRQ is being counted for your device.
- **Repeated interrupts on a single press (bouncing):** Mechanical buttons bounce. A simple software debounce can be done by ignoring interrupts within a short window after the first. Use `sbintime()` and a state field in the `softc`.
- **kldunload fails with EBUSY:** You missed a `bus_teardown_intr` or a `gpio_pin_release`.

The starter files live at `examples/part-07/ch32-fdt-embedded/lab05-edledi/`.

32.14.6 After the Labs

At the end of Lab 4 you have run through the whole arc of embedded driver work: source, overlay, kernel module, user-space access, teardown. The remaining sections of the chapter give you ways to stretch what you have built and a last look at common pitfalls.

32.15 Challenge Exercises

The exercises below go beyond the guided labs. They do not include step-by-step instructions, because the point is for you to apply what you have learned to unstructured problems. If you get stuck, the reference material in Sections 5 through 8 and the real drivers under `/usr/src/sys/dev/gpio/` and `/usr/src/sys/dev/fdt/` are your strongest resources.

32.15.1 Challenge 1: Multiple LEDs Per Node

Modify `edled` to accept a DT node that declares several GPIOs, like the real `gpio-led-fdt.c` does. The binding should look like:

```
edled0: edled@0 {
    compatible = "example,edled-multi";
    led-red = <&gpio 18 0>;
    led-amber = <&gpio 19 0>;
    led-green = <&gpio 20 0>;
};
```

Expose one sysctl per LED: `dev.edled.0.red`, `dev.edled.0.amber`, `dev.edled.0.green`. Each should behave independently.

Hint: Walk the DT properties in `attach`. For a cleaner structure, store an array of pin handles in the `softc` and iterate over it in both `attach` and `detach`. `gpio_pin_get_by_ofw_property` takes the property name as its third argument, so the same driver can handle different property names with a small lookup table.

32.15.2 Challenge 2: Support a Blink Timer

Extend `edled` with a second sysctl `dev.edled.0.blink_ms` that, when set to a non-zero value, starts a kernel callout that toggles the pin every `blink_ms` milliseconds. Writing 0 stops the blink and leaves the LED in its current state.

Hint: Use `callout_init_mtx` to associate the callout with the per-`softc` mutex, and `callout_reset` to schedule it. Remember to `callout_drain` in `detach` so the system does not leave a scheduled event pointing at freed memory.

32.15.3 Challenge 3: Generalize to an Arbitrary GPIO Output

Rename and generalize `edled` into `edoutput`, a driver that can drive any GPIO output line with a `sysctl` interface. Accept a `label` property from the DT and use it as part of the `sysctl` path so multiple instances do not collide. Add a `dev.edoutput.0.pulse_ms` `sysctl` that drives the line active for the given number of milliseconds and then returns to inactive.

Hint: `device_get_unit(dev)` gives you the unit number; use `device_get_nameunit(dev)` for a combined name+unit string when you need one.

32.15.4 Challenge 4: Consume an Interrupt

If your board exposes a button wired to a GPIO input (or you can use a pull-up resistor and a jumper wire as a one-shot simulation), modify the driver to watch an input pin for edge transitions and log them through `device_printf`. You will need to acquire the IRQ resource with `bus_alloc_resource_any(dev, SYS_RES_IRQ, ...)`, set up the interrupt handler with `bus_setup_intr`, and release it cleanly in `detach`.

Hint: Look at `/usr/src/sys/dev/gpio/gpiokeys.c` as a reference for a driver that consumes GPIO-triggered interrupts from FDT.

32.15.5 Challenge 5: Produce a Custom Device Tree for QEMU

Write a complete `.dts` for a hypothetical embedded board that includes:

- A single ARM Cortex-A53 core.
- 256 MB of RAM.
- One `simplebus`.
- One UART at a chosen address.
- One `edled` node under `simplebus`, referencing a GPIO controller.
- A minimal GPIO controller node you invent.

Compile the result, boot a FreeBSD arm64 kernel against it under QEMU with `-dtb`, and watch the driver attach. The GPIO controller will fail because nothing drives the invented hardware, but you will see the full path from DT to Newbus with your own source.

Hint: Use `/usr/src/sys/contrib/device-tree/src/arm64/arm/juno.dts` as a structural reference.

32.15.6 Challenge 6: Port a Real Driver's DT Binding to a New Board

Pick any existing FreeBSD FDT driver (for example, `sys/dev/iicbus/pmic/act8846.c`), read its DT binding by studying the driver source, and write a complete DT fragment that would attach it on a Raspberry Pi 4. You do not need to actually run the driver; the exercise is in reading the binding from the source and producing a correct `.dtsi` fragment.

Hint: Read the driver's `compat` table, its `probe`, and any `of_` calls to discover which properties it expects. Vendor kernel source trees often document DT bindings in comments at the top of the file.

32.15.7 Challenge 7: Hand-Write a Full DT for an Embedded Target Under QEMU

Challenge 5 invited you to write a partial DT for a hypothetical board. Challenge 7 goes further: write a complete .dts for a QEMU arm64 target that you define entirely yourself. Include memory, timer, a PL011 UART, a GIC, one PL061-style GPIO controller, and one instance of a peripheral of your own design. Boot an unmodified FreeBSD arm64 kernel under QEMU with your .dtb. Verify that the console comes up on your chosen UART and that the kernel's device probe walks the tree.

Hint: The qemu-system-aarch64 command supports -dtb plus -machine virt,dumpdtb=out.dtb to emit a reference DT you can study and adapt.

32.15.8 Challenge 8: Implement a Simple MMIO Peripheral Driver

Write a driver for a hypothetical MMIO peripheral you also simulate under QEMU. The peripheral exposes one 32-bit register at a fixed address. Reading the register returns a free-running counter; writing zero resets the counter. Your driver should expose one sysctl dev.counter.0.value that reads and writes this register. Verify that bus_read_4 and bus_write_4 work as expected. Simulate the hardware by writing a small QEMU device model, or by repurposing an existing simulated region whose value you can observe.

Hint: bus_read_4(sc->mem, 0) returns the u32 at offset 0 within your allocated memory resource. The bus_space(9) man page is the authoritative reference.

32.15.9 After the Challenges

These exercises are intentionally open-ended. If you finish any one of them, you have internalized the material of this chapter. If you find yourself wanting more, /usr/src/sys/dev/ has dozens of FDT drivers at all sizes. Reading a driver a week is one of the best habits an embedded FreeBSD developer can build.

32.16 Common Mistakes and Troubleshooting

This section is a concentrated reference for the issues that most often trip up writers of FDT drivers. Everything here has already been mentioned in Sections 3 through 8, but collecting the points in one place gives you something you can skim when a particular symptom appears.

32.16.1 Module Loads but No Device Attaches

Most probe failures come down to one of five causes:

1. **Typo in compatible.** Either in the DT source or in the driver's compat table. The string must match byte for byte.
2. **Node has status = "disabled".** Either fix the base tree or write an overlay that sets status to okay.
3. **Wrong parent bus.** If the node sits under an I2C or SPI controller node, the driver must register with that controller's driver type, not with simplebus.
4. **Overlay did not apply.** Check the loader output at boot for an error message. Confirm the .dtbo is in /boot/dtb/overlays/ and listed in loader.conf.

5. **Driver outranked by another driver.** Use `devinfo -r` to see which driver actually attached. Increase the probe's return value (`BUS_PROBE_DEFAULT` is the common baseline; `BUS_PROBE_SPECIFIC` signals a more exact match).

32.16.2 Overlay Fails to Apply at Boot

The loader prints its attempts on the console. Watch for lines like:

```
Loading DTB overlay 'edled' (0x1200 bytes)
```

If that line is missing, the loader either did not find the file or skipped it. Possible causes:

- Filename ends in `.dtbo` but `fdt_over` lays misspells it.
- File is in the wrong directory. The default is `/boot/dtb/overlays/`.
- Base blob and overlay disagree on the top-level compatible. The loader refuses to apply an overlay whose top-level compatible does not match the base.
- The `.dtbo` was compiled without `-@` and references labels the loader cannot resolve.

32.16.3 Cannot Allocate Resources

If `attach` calls `bus_alloc_resource_any(dev, SYS_RES_MEMORY, ...)` and it returns `NULL`, the most likely causes are:

- `reg` missing or malformed in the DT node.
- Cell count mismatch between the node's `reg` and the parent's `#address-cells/#size-cells`.
- Another driver already claimed the same region.
- The parent bus's ranges does not cover the requested address.

Print the resource start and size in `attach` while debugging:

```
if (sc->mem == NULL) {
    device_printf(dev, "cannot allocate memory resource (rid=%d)\n",
        sc->mem_rid);
    goto fail;
}
device_printf(dev, "memory at %#jx len %#jx\n",
    (uintmax_t)rman_get_start(sc->mem),
    (uintmax_t)rman_get_size(sc->mem));
```

32.16.4 GPIO Acquisition Fails

`gpio_pin_get_by_ofw_*` returns `ENXIO` when:

- The GPIO controller referenced in the DT property has not attached yet.
- The pin number is out of range for that controller.

- The phandle in the DT is wrong.

The first cause is by far the most common. The fix is `MODULE_DEPEND(your_driver, gpiobus, 1, 1, 1)` so the dynamic loader brings up gpiobus first.

32.16.5 Interrupt Handler Does Not Fire

If the hardware should raise an interrupt and nothing happens:

- Confirm the DT interrupts property is correct. The format depends on the parent interrupt controller's `#interrupt-cells`.
- Confirm `bus_alloc_resource_any(SYS_RES_IRQ, ...)` returned a valid resource.
- Confirm `bus_setup_intr` returned zero.
- Confirm your handler's return value is `FILTER_HANDLED` or `FILTER_STRAY` for a filter, or `FILTER_SCHEDULE_THREAD` if you are using a threaded handler.

Use `vmstat -i` to see whether your interrupt is being counted. If the count stays at zero, the interrupt is not even being routed to your handler.

32.16.6 Unload Returns EBUSY

Detach forgot to release something. Walk your attach carefully and confirm every `_get_` has a matching `_release_` call in detach. Common culprits are:

- GPIO pins retrieved with `gpio_pin_get_by_*`.
- Interrupt handlers set up with `bus_setup_intr`.
- Clock handles from `clk_get_by_*`.
- Regulator handles from `regulator_get_by_*`.
- Memory resources from `bus_alloc_resource_any`.

Print a breadcrumb on each release:

```
device_printf(dev, "detach: releasing GPIO pin\n");
gpio_pin_release(sc->sc_pin);
```

If the breadcrumbs stop before the expected end, the resource that goes uncleared is the one after the last printed line.

32.16.7 Panic During Boot

Panics that fire during FDT parsing usually mean the blob itself is malformed, or a driver is dereferencing an `OF_getprop` result it did not check. Two safeguards:

- Always check the return value of `OF_getprop`, `OF_getencprop`, and `OF_getprop_alloc`. A missing property returns `-1` or `ENOENT`; treating it as present leads to reads of whatever is next on the stack.
- Use `OF_hasprop(node, "prop")` before calling `OF_getprop` when a property is optional.

32.16.8 DT Compile Errors

dtc error messages are reasonably clear. A few patterns to recognize:

- **syntax error**: Missing semicolon, unbalanced brace, or bad property value syntax.
- **Warning (simple_bus_reg)**: A node under simple-bus has reg but no ranges translation, or its reg does not match the parent's cell counts.
- **FATAL ERROR: Unable to parse input tree**: The file is syntactically broken at a coarse level. Check for missing `/dts-v1/`; or mis-quoted strings.
- **ERROR (phandle_references): Reference to non-existent node or label**: An `&label` reference that the compiler cannot resolve. This is where `-@` matters; without it, overlays that depend on base-tree labels cannot be validated.

32.16.9 The Kernel Sees the Wrong Hardware

If your driver attaches but reads garbage from registers:

- Double-check the reg value in the DT.
- Confirm `#address-cells` and `#size-cells` match what you expect at the parent's level.
- Use hexdump `/dev/mem` style techniques only if you are certain the address is safe; reading the wrong MMIO range can hang the bus.

32.16.10 You Changed the Driver But Nothing Changed

Double-check:

- Did you run `make` after editing?
- Did you copy the new `.ko` to `/boot/modules/` (or load the local one explicitly)?
- Did you unload the old module before loading the new one? `kldstat -m driver` shows currently resident modules.

A simple habit that avoids the last pitfall is to always `kldunload` explicitly before `kldload`, or to use `kldload -f` to force replacement.

32.16.11 Quick Reference: Most-Used OFW Calls

For convenience, here is a compact table of the OFW and `ofw_bus` helpers that FDT drivers use most often. Each is declared in either `<dev/ofw/openfirm.h>` or `<dev/ofw/ofw_bus_subr.h>`.

Call	What it does
<code>ofw_bus_get_node(dev)</code>	Return the phandle of this device's DT node.
<code>ofw_bus_get_compat(dev)</code>	Return the first compatible string of the node, or NULL.
<code>ofw_bus_get_name(dev)</code>	Return the node's name part (before the '@').
<code>ofw_bus_status_okay(dev)</code>	True if status is missing, "okay", or "ok".

Call	What it does
<code>ofw_bus_is_compatible(dev, s)</code>	True if one of the compatible entries matches <code>s</code> .
<code>ofw_bus_search_compatible(dev, tbl)</code>	Return the matching entry in a <code>compat_data</code> table.
<code>ofw_bus_has_prop(dev, s)</code>	True if the property is present on the node.
<code>OF_getprop(node, name, buf, len)</code>	Copy raw property bytes.
<code>OF_getencprop(node, name, buf, len)</code>	Copy property, byte-swapping u32 cells to host endianness.
<code>OF_getprop_alloc(node, name, bufp)</code>	Allocate and return property; caller frees with <code>OF_prop_free</code> .
<code>OF_hasprop(node, name)</code>	Return non-zero if the property exists.
<code>OF_child(node)</code>	First child phandle, or 0.
<code>OF_peer(node)</code>	Next sibling phandle, or 0.
<code>OF_parent(node)</code>	Parent phandle, or 0.
<code>OF_finddevice(path)</code>	Look up a node by absolute path.

32.16.12 Quick Reference: Most-Used Peripheral Calls

From `<dev/gpio/gpiobusvar.h>`:

Call	What it does
<code>gpio_pin_get_by_ofw_idx(dev, node, idx, &pin)</code>	Acquire pin by <code>gpios</code> index.
<code>gpio_pin_get_by_ofw_name(dev, node, n, &pin)</code>	Acquire pin by named reference (e.g., <code>led-gpios</code>).
<code>gpio_pin_get_by_ofw_property(dev, n, p, &pin)</code>	Acquire pin from named DT property.
<code>gpio_pin_setflags(pin, flags)</code>	Configure direction, pulls, and similar.
<code>gpio_pin_set_active(pin, val)</code>	Drive output to active or inactive state.
<code>gpio_pin_get_active(pin, &val)</code>	Read current input or output level.
<code>gpio_pin_release(pin)</code>	Return pin to the unowned pool.

From `<dev/extres/clk/clk.h>`:

Call	What it does
<code>clk_get_by_ofw_index(dev, node, i, &c)</code>	Acquire <code>n</code> th clock listed in <code>clocks</code> property.

Call	What it does
<code>clk_get_by_ofw_name(dev, node, n, &c)</code>	Acquire clock by name.
<code>clk_enable(c) / clk_disable(c)</code>	Gate the clock on or off.
<code>clk_get_freq(c, &f)</code>	Read current frequency in Hz.
<code>clk_release(c)</code>	Release clock handle.

From `<dev/extres/regulator/regulator.h>`:

Call	What it does
<code>regulator_get_by_ofw_property(dev, node, p, &r)</code>	Acquire regulator from DT property.
<code>regulator_enable(r) / regulator_disable(r)</code>	Turn rail on or off.
<code>regulator_set_voltage(r, min, max)</code>	Ask for voltage within min..max range.
<code>regulator_release(r)</code>	Release regulator handle.

From `<dev/extres/hwreset/hwreset.h>`:

Call	What it does
<code>hwreset_get_by_ofw_name(dev, node, n, &h)</code>	Acquire reset line by name.
<code>hwreset_get_by_ofw_idx(dev, node, i, &h)</code>	Acquire reset line by index.
<code>hwreset_assert(h) / hwreset_deassert(h)</code>	Put peripheral into or out of reset.
<code>hwreset_release(h)</code>	Release reset handle.

From `<dev/fdt/fdt_pinctrl.h>`:

Call	What it does
<code>fdt_pinctrl_configure_by_name(dev, name)</code>	Apply the named pinctrl state.
<code>fdt_pinctrl_configure_tree(dev)</code>	Apply <code>pinctrl-0</code> recursively to children.
<code>fdt_pinctrl_register(dev, mapper)</code>	Register a new pinctrl provider.

Print this table, pin it near your workstation, and refer to it whenever you start a new driver. The calls become second nature within a few projects.

32.16.13 Summary Checklist

Before declaring a driver finished, run through this list:

- ☐ Module builds cleanly with no warnings under `-Wall -Wextra`.
- ☐ `kldload` produces the expected attach message.
- ☐ `kldunload` succeeds without `EBUSY`.
- ☐ Repeating load-unload cycles a dozen times does not leak resources.
- ☐ `devinfo -r` shows the driver in the expected position in the tree.
- ☐ `sysctls` are present, readable, and writable where intended.
- ☐ All DT properties the driver cares about are documented in the README.
- ☐ A companion overlay source compiles with `dtc -@`.
- ☐ A fresh reader could read the source and understand what it does.

32.17 Glossary of Device Tree and Embedded Terms

This glossary collects the terms this chapter uses, defined briefly so you can check a term on first encounter without searching the prose. Cross-references to the relevant section appear in parentheses where helpful.

ACPI: Advanced Configuration and Power Interface. An alternative to FDT used by PC-class and some arm64 servers. A FreeBSD kernel picks one or the other at boot. (Section 3.)

amd64: FreeBSD's 64-bit x86 architecture. Usually uses ACPI rather than FDT, though FDT can be used in specialized embedded x86 cases.

arm64: FreeBSD's 64-bit ARM architecture. Uses FDT by default on embedded boards; uses ACPI on SBSA-compliant servers.

Bindings: Documented conventions for how a peripheral's properties are spelled in Device Tree. For instance, the `gpio-leds` binding documents what properties an LED controller's DT node should carry.

Blob: Informal term for a `.dtb` file, since it is an opaque binary chunk from the perspective of anything other than an FDT parser.

BSP: Board Support Package. The collection of files (kernel config, Device Tree, loader hints, sometimes drivers) needed to run an operating system on a specific board.

Cell: A 32-bit big-endian value that is the atomic unit of a DT property. Property values are sequences of cells.

Compatible string: The identifier a driver's probe matches against, stored in the `compatible` property of a DT node. Usually has vendor-prefix-slash-model form: `"brcm,bcm2711-gpio"`.

Compat data table: An array of `struct ofw_compat_data` entries that a driver walks in probe via `ofw_bus_search_compatible`. (Section 4.)

dtb: Compiled Device Tree binary. Output of `dtc`; the format the kernel parses at boot.

dtbo: Compiled Device Tree overlay. A small binary that the loader merges into the main `.dtb` before handing it to the kernel.

dtc: The Device Tree Compiler. Translates source to binary.

dts: Device Tree Source. Text input to `dtc`.

dtsti: Device Tree Source Include. A source fragment meant to be `#include-d`.

Edge triggered: An interrupt that fires on a level transition (rising, falling, or both). Contrast with level-triggered.

FDT: Flattened Device Tree. The binary format and framework FreeBSD uses for DT-based systems. Also used colloquially to refer to the whole concept.

fdt_overlays: A `loader.conf` tunable that lists overlay names to apply at boot.

fdtdump: A utility that decodes a `.dtb` file to a readable approximation of its source.

Fragment: A top-level entry in an overlay that names a target and declares the content to merge. (Section 5.)

GPIO: General-Purpose Input/Output. A programmable digital pin that can drive or read a line.

intrng: FreeBSD's interrupt next-generation framework. Unifies interrupt controllers and consumers. (Section 10.)

kldload: The command that loads a kernel module into a running system.

kldunload: The command that unloads a loaded kernel module.

Level triggered: An interrupt that asserts as long as a condition is true. Must be cleared at the source to stop firing.

Loader: FreeBSD's boot loader. Reads configuration, loads modules, merges DT overlays, hands control to the kernel.

MMIO: Memory-Mapped IO. A hardware register set exposed through a range of physical addresses.

Newbus: FreeBSD's device driver framework. Every driver registers into a tree of Newbus parent-child relationships.

Node: A point in the Device Tree. Has a name (and optional unit address) and a set of properties.

OFW: Open Firmware. A historical standard that FreeBSD's FDT code reuses the API of.

ofwbus: FreeBSD's top-level bus for Open Firmware-derived device enumeration.

ofwdump: A userspace utility that prints nodes and properties from the running kernel's DT. (Section 6.)

Overlay: A partial `.dtb` that modifies an existing tree. Targets nodes by label or path and merges content under them. (Section 5.)

phandle: Phantom handle. A 32-bit integer identifier for a DT node, used to cross-reference nodes within the tree.

pinctrl: Pin control framework. Handles multiplexing of SoC pins between their possible functions. (Section 8.)

PNP info: Metadata a driver publishes to identify the DT compatible strings it supports. Used by `devmatch(8)` for auto-loading. (Section 9.)

Probe: The driver method that inspects a candidate device and reports whether it can drive it. Returns a strength score or an error.

Property: A named value in a DT node. Values can be strings, cell lists, or opaque byte strings.

Reg: A property listing one or more (address, size) pairs that describe the MMIO ranges the peripheral occupies.

Root controller: The topmost interrupt controller. On arm64 systems, usually a GIC.

SBC: Single-Board Computer. An embedded board with CPU, memory, and peripherals on one PCB. Examples: Raspberry Pi, BeagleBone.

SIMPLEBUS_PNP_INFO: A macro that exports a driver's compat table as module metadata. (Section 9.)

Simplebus: The FreeBSD driver that probes DT children whose parent has `compatible = "simple-bus"`. It turns DT nodes into Newbus devices.

softc: Short for “soft context.” A per-device state structure allocated by Newbus and passed to driver methods. (Section 4.)

SoC: System on Chip. An integrated circuit containing CPU, memory controller, and many peripheral blocks.

Status: A property on a DT node that indicates whether the device is enabled ("okay") or not. Missing status defaults to okay.

sysctl: FreeBSD's system control interface. A driver can publish tunable parameters that userspace reads and writes.

Target: In an overlay, the node in the base tree that a fragment modifies.

Unit address: The numeric part after the @ in a node name. Indicates where in the parent's address space the node lives.

Vendor prefix: The part of a compatible string before the comma. Identifies the organization responsible for the binding.

32.18 Frequently Asked Questions

These are questions that come up repeatedly when people first write FDT drivers for FreeBSD. Most are already answered somewhere in the chapter; the FAQ format just puts the short answer in one place.

Do I need to know ARM assembly to write an FDT driver?

No. The whole point of the driver framework is that you work in C against a uniform API. You might read disassembly if you are debugging a very low-level crash, but that is the exception, not the rule.

Can I write FDT drivers on amd64, or do I need arm64 hardware?

You can develop on amd64 and cross-build for arm64. You can also run arm64 FreeBSD under QEMU on an amd64 host, which is the most common workflow for anyone who does not want to wait on a Pi to reboot. For the final validation, you eventually want real hardware or a faithful emulator, but day-to-day iteration fits on a laptop.

What is the difference between simplebus and ofwbus?

ofwbus is the top-level root bus for Open Firmware-derived device enumeration. simplebus is a generic bus driver that covers DT nodes compatible with "simple-bus" and similar simple enumeration. Most of your drivers will register with both; ofwbus handles the root and specialized cases, simplebus handles the overwhelming majority of peripheral buses.

Why does my driver need to register with both ofwbus and simplebus?

Some nodes appear under simplebus, some directly under ofwbus (particularly on systems where the tree structure is unusual). Registering with both means the driver attaches wherever the node happens to end up.

Why is my overlay not applying?

Run through the checklist in Section 6. The most common cause, in order: wrong filename or directory, typo in `fdt_overlays`, base compatible mismatch, missing `-@` at overlay compile time.

Can a driver span multiple DT nodes?

Yes. A single driver instance usually matches one node, but the attach function can walk children or phandle references to collect state from several nodes. See Section 9's discussion of `gpioled_fdt.c` for the children case.

How do I handle a device that has both ACPI and FDT descriptions?

Write two compat paths. Most large FreeBSD drivers that support both platforms do exactly this: separate probe functions register with each bus, and the shared code lives in the common attach. Look at `sdhci_acpi.c` and `sdhci_fdt.c` for a worked example.

What about DT bindings I do not find in the FreeBSD tree?

The source of truth for DT bindings is the upstream device-tree specification plus the Linux kernel's documentation. FreeBSD uses the same bindings where practical. If you need a binding FreeBSD does not yet support, you can typically port the relevant Linux driver or write a native FreeBSD driver that consumes the same binding.

Do I need to modify the kernel to add a new driver?

No. Out-of-tree modules compile against `/usr/src/sys` headers and load at runtime. You only edit the kernel tree itself when you contribute a driver upstream or when you need to change a generic piece of infrastructure.

How do I cross-compile for arm64 from an amd64 host?

Use the cross toolchain included in FreeBSD's build system:

```
$ make TARGET=arm64 TARGET_ARCH=aarch64 buildworld buildkernel
```

This builds a full arm64 system image on your amd64 workstation. Module-only builds follow the same pattern with narrower targets.

Is there a way to auto-load my driver when a matching DT node appears?

Yes, through `devmatch(8)` and the `SIMPLEBUS_PNP_INFO` macro. Declare your compat table, include `SIMPLEBUS_PNP_INFO(compat_data)` in the driver source, and `devmatch` picks it up.

Can I use C++ for an FDT driver?

No. FreeBSD kernel code is strictly C (and a very small amount of assembly). Other languages are not supported, and the kernel API assumes C conventions.

How do I debug a crash during DT parsing at boot?

Early crashes are hard. The usual techniques: enable verbose boot messages (`-v` in `loader.conf`), compile the kernel with KDB and DDB, use a serial console, and attach a JTAG debugger if you have one. Also consider adding temporary `printf` statements directly in the FDT parsing code under `/usr/src/sys/dev/ofw/`.

Do modules for drivers that share FDT bindings interfere with each other?

No. Each module registers its compat table, and the probe strength of matching drivers decides which one wins. If two drivers claim the same compatible with the same strength, the order they were loaded determines the winner. Give each driver a distinctive strength to avoid surprises.

How do I preserve state across kldload/kldunload cycles?

You cannot. A module that unloads loses all its state. If your driver needs persistence, write to a file, a sysctl's tunable form, a kernel tunable, or a location in NVRAM or EEPROM that outlives the module. For debugging, printing state to dmesg before unload and parsing it back after the next load is a workable shortcut.

Is the compatible list order significant?

Yes. A node may list multiple compatible strings, from most specific to least specific. `compatible = "brcm,bcm2711-gpio", "brcm,bcm2835-gpio"`; declares that the node is primarily a 2711-variant but is compatible with the older 2835 binding as a fallback. A driver that claims the 2835 compatible will match this node if no driver for 2711 is loaded. The order lets firmware describe a device in multiple levels of detail so newer kernels can take advantage of refinements without breaking older ones.

Why does FreeBSD sometimes use different property names than Linux?

Most DT properties are shared, but a few are intentionally different where FreeBSD's behavior diverges from Linux's expectations. When you port a driver, read the FreeBSD side's existing bindings carefully; silently assuming the Linux semantics is a common source of porting bugs.

What is the relationship between Newbus and intrng?

Newbus handles device probe, attach, and resource allocation. intrng handles interrupt controller registration and IRQ routing. They interoperate: Newbus resource allocation for `SYS_RES_IRQ` goes through intrng to find the right controller, and intrng dispatches interrupts back into the driver handler that Newbus registered.

32.19 Wrapping Up

This chapter took you from “FreeBSD is a general-purpose operating system” to “I can write a driver that fits into an embedded system's Device Tree.” The two tasks are not the same. The first is about using the kernel you already know; the second is about understanding a whole new vocabulary for describing the hardware the kernel runs on.

We began by looking at what embedded FreeBSD actually is: a lean, capable system running on SBCs, industrial boards, and purpose-built hardware. We saw how those systems describe themselves not through PCI enumeration or ACPI tables but through a static Device Tree that the firmware hands to the kernel at boot.

We then learned the Device Tree language itself: nodes with names and unit addresses, properties with cell-structured values, phandles for cross-references, and the `/plugin/` overlay syntax that lets us add or modify nodes without rebuilding the base blob. The language takes an afternoon to get used to; the habits it builds, about thinking of hardware as a parented, addressed, typed tree, last for a career.

With the language in hand, we looked at FreeBSD's machinery for consuming a DT. The `fdt(4)` subsystem loads the blob. The OFW API walks it. The `ofw_bus_*` helpers expose the walk in terms of compat strings and status checks. The `simplebus(4)` driver enumerates children. And the consumer frameworks (`clk`, `regulator`,

hwreset, pinctrl, GPIO) all integrate with DT phandle references so that drivers can acquire their resources through a uniform pattern.

Armed with the machinery, we built a driver. First `fdthello`, the minimal skeleton, which showed the required shape of an FDT driver in its purest form. Then `edled`, a complete GPIO-driven LED driver that illustrates attach, detach, `sysctl`, and proper resource management. Along the way we looked at how to compile overlays, deploy them through `loader.conf`, inspect the tree at runtime with `ofwdump`, and debug the class of failures that embedded drivers uniquely exhibit.

Finally, we took `edled` through the refactoring passes that turn a working driver into a finished one: tightened error paths, a real mutex, optional power and clock handling, `pinctrl` awareness, a style audit. This is the work that distinguishes a toy from a driver you would be comfortable running in production.

The chapter's labs gave you a chance to run everything yourself. The challenge exercises gave you room to extend. The troubleshooting section gave you somewhere to look when things break.

32.19.1 What You Should Be Able to Do Now

If you worked through the labs and read the explanations carefully, you can now:

- Read an unfamiliar `.dts` and explain what hardware it describes.
- Write a new `.dts` or `.dtsi` that adds a peripheral to an existing board.
- Compile that source into a binary and deploy it to a running system.
- Write an FDT-aware FreeBSD driver from scratch.
- Acquire memory, IRQ, GPIO, clock, and regulator resources through the standard consumer APIs.
- Debug the probe-did-not-fire, attach-failed, and detach-stuck classes of problems.
- Identify when a board is running in FDT mode versus ACPI mode on arm64, and what that means for your driver.

32.19.2 What This Chapter Leaves Implicit

There are three topics the chapter leaves for other sources. They are not book-shaped; they are reference-shaped, and the book would bog down if it tried to cover them fully.

First, the full DT binding specification. We covered the properties you are most likely to use. The complete binding catalog, maintained upstream by the device-tree community, is browsable at the online documentation for Linux kernel's `Documentation/devicetree/bindings/` tree. FreeBSD follows most of those bindings, with exceptions noted in `/usr/src/sys/contrib/device-tree/Bindings/` for the subset FreeBSD relies on.

Second, interrupt routing on complex SoCs. We touched on interrupt-parent chaining. A board with multiple GIC-style controllers, `pinctrl`-managed interrupts, and nested `gpio-as-interrupt` nodes can get intricate. The FreeBSD `intrng` subsystem is where to look when the simple case is no longer enough.

Third, the FDT support for peripherals that do not go through `simplebus`: USB phys, clock trees on SoCs with hierarchical PLLs, voltage scaling for DVFS. Each is its own subtopic. The book's appendix points at the canonical sources.

32.19.3 Key Takeaways

If the chapter can be distilled into a single page, here are the points worth remembering:

1. **Embedded systems describe themselves through Device Tree, not through runtime enumeration.** The blob the firmware hands the kernel is the authoritative description of what hardware is present.
2. **A driver matches through compatible strings.** Your probe consults a compat table, which compares against the node's compatible property. Get this string exactly right.
3. **simplebus(4) is the enumerator.** Every FDT driver's parent is either simplebus or ofwbus. Register with both at module load time.
4. **Resources come from the framework.** `bus_alloc_resource_any` for MMIO and IRQ, `gpio_pin_get_by_ofw_*` for GPIO, and the matching `clk_*`, `regulator_*`, `hwreset_*` calls for power and reset. Release each in detach.
5. **Overlays modify the tree without rebuilding it.** A small `.dtbo` placed under `/boot/dtb/overlays/` and listed in `fdt_overlays` is a clean way to add or enable peripherals on a given board.
6. **Interrupt wiring follows a parent chain.** A node's interrupt-parent and interrupts properties connect through intrng to the root controller. Understanding the chain is essential for debugging silent interrupts.
7. **Real drivers are the best reference.** `/usr/src/sys/dev/gpio/`, `/usr/src/sys/dev/fdt/`, and each SoC's platform tree contain dozens of FDT drivers that demonstrate every pattern you are likely to need.
8. **Error paths and teardown matter.** A driver that loads and works once is the easy case. A driver that loads, unloads, and reloads cleanly a hundred times is the robust case.
9. **The tooling is simple but effective.** `dtc`, `ofwdump`, `fdtdump`, `devinfo -r`, `sysctl dev.<name>.<unit>`, and `kldstat` together cover almost every inspection you need.
10. **FDT is just one of several hardware-description systems.** On arm64 servers, ACPI takes the same role. The driver patterns you learned here transfer, but the match layer changes.

32.19.4 Before You Move On

Before you treat this chapter as completed and move to Chapter 33, take a moment to verify the following. These are the kinds of checks that separate a learner who has seen the material from a learner who owns it.

- You can sketch, without looking, the skeleton of an FDT-aware driver, including the compat table, probe, attach, detach, method table, and `DRIVER_MODULE` registration.
- You can take a paragraph of DT source and narrate what it describes, node by node, property by property.
- You can distinguish a phandle from a path reference and explain when you would use each.
- You can explain why `MODULE_DEPEND(your_driver, gpiobus, 1, 1, 1)` matters and what goes wrong without it.
- You know which loader tunable controls overlay loading and where the `.dtbo` files live.
- Given a probe that does not fire, you have a mental checklist of four or five things to verify before reaching for a debugger.
- You know what `SIMPLEBUS_PNP_INFO` does and why it matters for production drivers.

- You can name at least three real FDT drivers in `/usr/src/sys` and describe, in a sentence each, what they demonstrate.

If any of these still feel shaky, loop back to the relevant section and re-read it. The chapter's material compounds; Chapter 33 will assume you have internalized most of what is here.

32.19.5 A Note on Sustained Practice

Embedded FreeBSD rewards regular practice. The first time you read a `.dts` for a real board, it looks overwhelming. The tenth time, you skim for the nodes you care about. The hundredth time, you edit in place. If you have a spare SBC on your desk, put it to work. Pick a sensor, write a driver, add a line to a `loader.conf`. The skill compounds. Every small driver you finish is a template for the next one, and the patterns carry across boards and vendors.

32.19.6 Connecting Back to the Rest of the Book

Chapter 32 sits late in Part 7, and the material here rests on layers you met earlier. A short tour of what you now see in a new light:

From Part 1 and Part 2, the kernel module skeleton, the `DRIVER_MODULE` registration, and the `kld-load/kldunload` lifecycle. The shape of an FDT driver is the same shape, with a different probe strategy.

From Part 3, the Newbus framework. `simplebus` is a bus driver that happens to source its children from a Device Tree rather than from a bus protocol. Every Newbus pattern you learned earlier applies, unchanged, in this context.

From Part 4, the driver-to-userspace interfaces: `cdev`, `sysctl`, `ioctl`. Our edited driver used `sysctl` as its control interface; in a larger project it might add a character device or even a netlink socket.

From Part 5 and Part 6, the practical chapters on concurrency, testing, and kernel debugging. Those tools all apply to embedded drivers in full. The only difference is that the hardware is often harder to reach, so the tools matter more.

From Part 7, Chapter 29 on 32-bit platforms and Chapter 30 on virtualization both touch on embedded angles. Chapter 31's security chapter applies directly: a driver on an embedded device is often exposed to whatever runs in the product's userspace, and the same defensive patterns apply.

The picture after Chapter 32 is that you have a complete starter toolkit for most classes of FreeBSD device work. You can write character drivers, bus drivers, network drivers, and now FDT-based embedded drivers. The remaining chapters of Part 7 refine and finalize those skills.

32.19.7 Looking Ahead

The next chapter, Chapter 33, turns from “does it work” to “how well does it work.” Performance tuning and profiling is the art of measuring what the kernel actually does, under real load, with real data. On an embedded system, the question has extra force. The hardware is usually small, the workload is often fixed, and the margin between “runs well” and “runs badly” is measured in microseconds. In Chapter 33 we look at the tools FreeBSD gives you to measure: `hwpmc`, `pmcstat`, `DTrace`, flame graphs, and the kernel's own

performance-counter probes. We also look at the mental model you need to interpret what the tools say, and at the pitfalls of tuning the wrong thing.

Beyond Chapter 33, Chapters 34 through 38 complete the mastery arc: coding style and readability, contributing to FreeBSD, porting drivers between platforms, documenting your work, and a capstone project that brings several earlier threads together. Each builds on what you have done here. The `ed1ed` driver you wrote in this chapter, extended through the challenge exercises, is a perfectly reasonable candidate to carry forward into those final chapters as a running example.

The driver work is done. The measurement work begins.

32.19.8 A Parting Word

One last encouragement before we close. Embedded FreeBSD has a quiet, steady community. The boards are inexpensive, the source is open, the mailing lists are patient. If you stick with it, you will find that each driver you write teaches you something the next one needs. The `ed1ed` driver in this chapter is a small thing, yet in writing it you touched the FDT parser, the OFW API, the GPIO consumer framework, the Newbus tree, and the `sysctl` mechanism. That is not nothing. It is the backbone of every FDT driver in the tree, exercised in miniature.

Keep the practice up. The field rewards curiosity with steady progress. The next chapter gives you the tools to make sure that progress is progress in the right direction.

When you eventually sit down in front of an unfamiliar board, the view will not feel unfamiliar anymore. You will know to ask for its `.dts`, to skim the peripherals listed under `soc`, to check which GPIO controller hosts the pin you care about, to look up the compatible string you need to match. The habits you have just built travel with you. They apply to a Raspberry Pi, to a BeagleBone, to a custom industrial board, to a QEMU virt machine. They apply in ten years from now on boards that do not yet exist, because the underlying language of hardware description is stable.

That portability, in the end, is what the Device Tree was designed for, and what this chapter was designed to teach. Welcome to embedded FreeBSD.

With Chapter 32 in hand, your toolkit is nearly complete. On to Chapter 33, and to the measurement work that will tell you whether the driver you just wrote is as efficient as it needs to be.

Good reading, and good driver writing.

33 Performance Tuning and Profiling

33.1 Introduction

Chapter 32 ended with a driver that attaches on an embedded board, reads its pin assignments from a Device Tree, toggles an LED, and detaches cleanly. That chapter answered one question: *does the driver work?* The present chapter answers a different one: *does it work well?* The two questions sound similar and are profoundly distinct.

A driver that works is one whose code path produces the expected result for each valid input. A driver that works well is one whose code path produces that result with the right throughput, the right latency, the right CPU cost, and the right memory footprint for the machine it runs on. You can write a driver that passes every functional test and still makes the system sluggish, miss deadlines, drop packets, or overheat a small board. In kernel work, correctness is the floor. Performance is the view above it.

We are going to spend this chapter building the habits, tools, and mental model that turn an observation like *the system feels slow* into a diagnosis like *the driver's write path is acquiring a contended mutex twice per byte, which on this workload adds forty microseconds per call*. The diagnosis, not the fix, is the hard part. The fix is usually small. What takes experience is the discipline to resist guessing and the skill to measure the right thing.

There is a joke among performance engineers that every beginner optimisation starts with the words *I thought it would be faster if....* That sentence is the single most reliable source of regressions in kernel code. A cache-friendly layout in the abstract can become a cache-hostile layout on a particular processor. A lock-free queue can be slower than a mutex-guarded one at low contention. A carefully tuned interrupt coalescing window can ruin latency for a small class of workloads you did not have in mind. The only antidote to *I thought it would be faster* is *I measured that it is faster*, and the whole point of this chapter is to give you a toolkit with which *I measured that* becomes a habit.

FreeBSD gives you an unusually rich set of measurement tools. DTrace, originally brought into the tree from Solaris, lets you probe almost any function in the kernel without recompiling anything. The `hwpmc(4)` subsystem and the `pmcstat(8)` userland tool expose the CPU's hardware performance counters: cycles, instructions, cache misses, branch mispredictions, stalls. The `sysctl(9)` interface and the `counter(9)` framework give you cheap, safe, always-on counters you can leave in production. The `log(9)` kernel logging facility gives you a rate-limited channel for informational messages. The `ktrace(1)` subsystem traces system-call boundaries. The `top(1)` and `systat(1)` tools let you see where CPU time is going in aggregate. Each of these tools has a particular strength, and the chapter will spend careful time showing you which tool to reach for when.

We are also going to spend time on the *habits* of measurement rather than only the tools. Measurement is easy to do badly. A driver that is instrumented heavily may not behave like the uninstrumented version; the probes change timing. A counter that is written from every CPU to a single cache line can become the bottleneck the driver is trying to measure. A DTrace script that prints on every probe turns into a denial-of-service attack

on `dmesg`. A `sysctl` that reads unaligned memory may appear fast because the compiler silently inlines it on x86-64 and slow on arm64. The tools you use to measure *affect* what you measure, and you need the habit of remembering this while you work.

The chapter has a clear arc. We start in Section 1 with what performance is, why it matters, when it is worth pursuing, and how to set measurable goals. Section 2 introduces the kernel's timing and measurement primitives: the different `nano*` and `bintime` calls, the precision-versus-cost trade-off between them, and the habits for counting operations and timing them without wrecking the code path you are trying to observe. Section 3 is the DTrace chapter-within-a-chapter: we look at the providers most useful to driver authors, the form of a useful one-liner, the shape of a longer script, and the `lockstat` provider that specialises in lock contention. Section 4 introduces hardware PMCs, explains what cycles, cache misses, and branches actually measure, and walks through a `pmcstat(8)` session from sample collection to a flame-graph-style view of a driver's hot functions. Section 5 turns inward, toward the driver: cache-line alignment, buffer pre-allocation, UMA zones, per-CPU counters via `DPCPU_DEFINE(9)` and `counter(9)`. Section 6 looks at interrupt handling and taskqueue use, because interrupts are often where the first performance cliff sits. Section 7 covers the production-grade runtime metrics you leave in the driver after tuning ends: `sysctl` trees, rate-limited logging, and the kinds of debug modes that belong in a shipped driver. Section 8 is the discipline section: how to remove your scaffolding, document what you measured, and ship the optimised driver as a version the rest of the system can rely on.

After Section 8 you will find hands-on labs, challenge exercises, a troubleshooting reference, a Wrapping Up section, and a bridge into Chapter 34.

One practical note before we begin. Performance work in a kernel is easier than it was in the 1990s and harder than it appears today. The 1990s problem was that the tools were few and the documentation was thin. The modern problem is the opposite: the tools are many, each is capable, and it is easy to use the wrong one or to chase a metric that does not match the workload. Pick the simplest tool that answers your question. A DTrace one-liner that shows you a function runs twice as often as you expected is worth more than a six-hour `pmcstat` session that produces a flame graph of noise. The most valuable skill is taste in picking the question, not virtuosity with any single tool.

Let us begin.

33.2 Reader Guidance: How to Use This Chapter

Chapter 33 sits at a particular place in the book's arc. Earlier chapters taught you how to build drivers that do their job, survive bad input, integrate with the kernel's frameworks, and run on diverse platforms. This chapter teaches you how to look at those drivers with measuring instruments and decide whether they are doing their job *well*. The orientation is inward, toward code you have already written, and outward, toward the tools FreeBSD gives you to look at that code while it runs.

There are two reading paths. The reading-only path takes about three to four focused hours. You finish with a clear mental model of what the FreeBSD performance tools do, when to use each, and how a disciplined tuning cycle looks from goal to fix. You will not have produced a tuned driver, but you will be able to read a `pmcstat` output, a DTrace aggregation, a `lockstat` summary, or a `sysctl` metric from a colleague and understand what it says.

The reading-plus-labs path takes seven to ten hours spread across two or three sittings. The labs are built around a small pedagogical driver called `perfdemo`, a character device that synthesises reads from an in-kernel generator

at a configurable rate. Across the chapter you will instrument `perfdemo` with counters and `sysctl` nodes, trace it with `DTrace`, sample it with `pmcstat`, tighten its memory and interrupt paths, expose runtime metrics, and finally strip the scaffolding and ship a v2.3-optimized version. Every lab ends in something you can observe on a running FreeBSD system. You do not need embedded hardware; an ordinary amd64 or arm64 FreeBSD machine, physical or virtual, is enough.

If you happen to have a machine with a hardware PMU and privileges to load the `hwpmc(4)` kernel module, you will gain the most from Section 4. Most consumer x86 and arm64 systems do have useable counters. Virtual machines expose a subset; some cloud environments expose none. When a lab requires PMC support, we will flag it and offer an alternative path using `profile` `DTrace` sampling that works everywhere.

33.2.1 Prerequisites

You should be comfortable with the driver material from Parts 1 through 6 and the embedded-focused Chapter 32. In particular you should know:

- What `probe()`, `attach()`, and `detach()` do and when each runs.
- How to declare a softc, how `device_t` and the `kobj` method table fit together, and how `DRIVER_MODULE(9)` registers a driver with `newbus`.
- How to allocate and release a mutex via `mtx_init(9)`, `mtx_lock(9)`, `mtx_unlock(9)`, and `mtx_destroy(9)`.
- How `bus_setup_intr(9)` registers an interrupt filter or handler, and the difference between filter and `ithread` handlers.
- How to allocate memory with `malloc(9)`, release it with `free(9)`, and when `M_WAITOK` versus `M_NOWAIT` is appropriate.
- How to declare a `sysctl` node with the static macros (`SYSCTL_DECL`, `SYSCTL_NODE`, `SYSCTL_INT`, and siblings) and the dynamic context (`sysctl_ctx_init(9)`).
- How to run `kldload(8)`, `kldunload(8)`, and `sysctl(8)` on a FreeBSD system.

If any of these feel shaky, the relevant earlier chapter is a short revisit away. Chapter 7 covers module and lifecycle basics, Chapters 11 through 13 cover locking and interrupt handling, Chapter 15 covers `sysctl`, Chapter 17 covers taskqueues, Chapter 21 covers DMA and buffering, and Chapter 32 the `newbus` resource-acquisition patterns you will build on in Section 6.

You do not need prior experience with `DTrace`, `pmcstat`, or flame graphs. The chapter introduces each from scratch at the level a driver author needs, and points you at the manual pages when you want to go deeper.

33.2.2 Structure and Pacing

Sections 1 and 2 are foundational. They introduce the vocabulary of performance and the kernel's measurement primitives. They are short by the standards of this chapter, because the book's earlier material already assumes the reader knows how to read a `sysctl` and how to reason about mutexes.

Sections 3 and 4 are the tool-heavy sections. `DTrace` and `pmcstat` deserve thorough treatment because they are the two tools you will reach for most often. Expect to spend an hour on each if you are reading carefully.

Sections 5 and 6 are the tuning sections. They are the ones that tell you what to *change* once you have measured. They are also the sections most likely to tempt a reader into over-engineering; read them with the mindset that

you will apply their techniques selectively, only after evidence demands it.

Section 7 is about the metrics that stay in the driver after tuning. This section is the one that turns a sprinter into a marathon runner; the drivers that age well are the ones that expose the right numbers, not the ones that went fastest on the benchmark day.

Section 8 closes the loop. It teaches you how to remove the temporary instrumentation that helped during tuning, document the benchmarks you ran, update the manual page with the tuning knobs the driver now exposes, and publish the driver at its new version.

Read the sections in order on a first pass. Each is written to stand alone enough for later reference, but the book's progressive teaching model depends on the order.

33.2.3 Work Section by Section

Each section covers one clear piece of the subject. Read one, let it sit for a while, then move on. If the end of a section feels blurry, pause and re-read the final paragraphs; they are designed to consolidate the material before the next section builds on it.

33.2.4 Keep the Lab Driver Close

The lab driver lives under `examples/part-07/ch33-performance/` in the book's repository. Each lab directory holds a self-contained stage of the driver with its `Makefile`, `README.md`, and any DTrace scripts or helper shell scripts. Clone the directory, work in place, build it with `make`, load it with `kldload`, and run the relevant measurements. The feedback loop between a kernel module and a `sysctl` or DTrace reading is the most educational feature of this chapter; use it.

33.2.5 Open the FreeBSD Source Tree

Several sections refer to real FreeBSD files. The most useful ones to keep open for this chapter are `/usr/src/sys/sys/counter.h` and `/usr/src/sys/sys/pcpu.h`, which define the counter and per-CPU primitives; `/usr/src/sys/sys/time.h`, which declares the kernel's time functions; `/usr/src/sys/sys/lockstat.h`, which defines the lockstat DTrace probes; `/usr/src/sys/vm/uma.h`, which declares the UMA zone allocator; `/usr/src/sys/dev/hwpmc/`, the PMC driver tree; and `/usr/src/sys/kern/subr_taskqueue.c`, where `taskqueue_start_threads_cpuset()` lives. Open them when the chapter points you at them. The manual pages (`counter(9)`, `sysctl(9)`, `mutex(9)`, `dtrace(1)`, `pmc(3)`, `pmcstat(8)`, `hwpmc(4)`, `uma(9)`, `taskqueue(9)`) are the second-best references after the source.

A note on line numbers. Each of those files will still define the symbols the chapter references: `sbin_uptime` in `time.h`, the lockstat probe macros in `lockstat.h`, `taskqueue_start_threads_cpuset` in `subr_taskqueue.c`. Those names persist across FreeBSD 14.x point releases. The line each one sits on in your tree may have moved since this chapter was written, so search for the symbol rather than scrolling to a specific number.

33.2.6 Keep a Lab Logbook

Continue the lab logbook from earlier chapters. For this chapter, note the *numbers* in particular. A log entry like *perfdemo v1.0, read() at 1000 Hz, median 14.2 us, P99 85 us* is the kind of evidence that lets you compare a later version honestly. Without the logbook, you will find yourself guessing a week from now whether a change made things faster; with it, you can check.

33.2.7 Pace Yourself

Performance work is cognitively demanding. A reader who does two focused hours, takes a proper break, and does two more hours will almost always go further than a reader who grinds through five hours in one sitting. The tools and the data benefit from a rested mind.

33.3 How to Get the Most Out of This Chapter

A few habits compound throughout the chapter. They are the same habits that experienced performance engineers use; the only trick is to start early.

33.3.1 Measure Before You Change

This is the rule. Every optimisation should begin with a measurement that shows the current behaviour, end with a measurement that shows the new behaviour, and compare the two against a goal stated in advance. An optimisation without a before-and-after number is a guess that happened to pass review.

If you remember nothing else from this chapter, remember this.

33.3.2 State the Goal in Numbers

A goal like *I want the read path to be faster* is not a goal. A goal like *I want median read() latency below 20 microseconds and P99 below 80 microseconds, on this workload* is a goal. You can measure it, you can know when you have achieved it, and you can stop optimising once you have. Most performance work runs long because the goal was vague.

33.3.3 Prefer the Simplest Tool That Answers the Question

The temptation to reach for `pmcstat` with a callgraph and a flame graph is strong. Usually a DTrace one-liner is enough. The simplest tool that tells you which function dominates, which counter is climbing, or which lock is contended is the right tool. Escalate only when the simple tool runs out.

33.3.4 Never Instrument the Hot Path at Production Cost

The act of measuring adds latency. A counter increment costs a few nanoseconds on modern hardware. A DTrace probe, when enabled, costs tens to hundreds of nanoseconds depending on the probe. A `printf` in the

hot path costs tens of microseconds. Know these rough numbers before you sprinkle measurement code around. If the driver's budget is fifty microseconds, a single `printf` will eat the budget.

33.3.5 Read the Manual Pages You Use

`dtrace(1)`, `pmcstat(8)`, `hwpmc(4)`, `sysctl(8)`, `sysctl(9)`, `counter(9)`, `uma(9)`, `taskqueue(9)`, `mutex(9)`, `timer(9)`, and `logger(1)` are the manual pages that matter for this chapter. Each is short, precise, and written by people who know the tool deeply. The book cannot replace them, only orient you.

33.3.6 Type the Lab Code

The `perfdemo` driver in the labs is small for the same reason `edled` was small in Chapter 32: so that you can type it. Type it. The finger-memory of writing a `sysctl` tree, a DTrace probe definition, a counter increment, and a lock acquisition path is worth more than reading the same code ten times.

33.3.7 Tag Every Measurement with Its Context

A number without context is useless. Every measurement should be recorded with the workload that produced it, the machine it ran on, the kernel version, the driver version, and what the system was also doing at the time. A driver that does 1.2 million operations per second on an idle server may do 400,000 on the same server while a backup is running. If you do not record the context, you will be mystified when the number changes and will blame the driver.

33.3.8 Do Not Tune What Is Not Slow

Every section of this chapter will describe a technique you could apply. Apply them only when measurement demands. Cache-line alignment matters when false sharing is a real cost; on a low-concurrency driver it is clutter. Per-CPU counters matter when counter contention is real; on a low-throughput driver they are premature. Resist the temptation to sprinkle all the techniques because you have just learned them; the book has taught you the options, and the measurements tell you which to choose.

33.3.9 Follow the Numbers Between Sections

As you work through the chapter, keep the `perfdemo` driver's numbers in your logbook. Each section should produce a small improvement that moves the numbers in the right direction. If a section's technique does not move your numbers, write that down too; it is equally valuable to know what *did not* help.

With those habits in hand, let us look at the reason any of this matters in the first place: the performance behaviour of the driver itself.

33.4 Section 1: Why Performance Matters in Device Drivers

Every driver is a performance contract whether its author writes one or not. The contract says: *when you ask me to do X, I will do X in this amount of time, using this much CPU, with this peak memory, and with*

this variability. A driver that keeps its contract is a well-behaved part of the system. A driver that breaks it produces the symptoms everyone on the system feels: audio skipping, video tearing, packets dropped, storage queues piling up, shells hanging on a write, interrupt-driven sensors missing events, a UI stuttering even though the machine has spare CPU. Most user-visible performance problems on any non-trivial system have a driver somewhere in their call path.

The contract has four axes. Learning to see them as separate is the first real step of this chapter.

33.4.1 The Four Axes of Driver Performance

Throughput is the amount of work the driver can do per unit time. For a network driver it is packets per second, or bytes per second. For a storage driver it is I/O operations per second (IOPS) or megabytes per second. For a character device it is reads or writes per second. For a GPIO driver it is pin toggles per second. Throughput answers the question *how much*?

Latency is the time between a request and its completion. For a network driver it is the microseconds between a packet arriving and the stack handing it to the protocol layer. For a storage driver it is the milliseconds (or now microseconds) between a `read()` call and the data being available. For an input driver it is the time between a key press and the userland process seeing the event. Latency answers the question *how fast?*, and unlike throughput it is usually measured as a distribution. A median (P50) of 10 microseconds with a P99 of 500 microseconds is not the same driver as a median of 50 microseconds with a P99 of 60 microseconds.

Responsiveness is a related but distinct axis: how quickly the driver wakes up and does its work after an event, from the perspective of other threads. A driver may have fine latency on its hot path but still be unresponsive if it holds a lock for ten milliseconds, if it busy-waits on a register poll, or if it queues work on a taskqueue that shares threads with some other slow subsystem. Responsiveness is what users and schedulers experience; latency is what the driver itself would report.

CPU cost is how much CPU time each unit of work consumes. A driver doing 100,000 operations per second at 1% CPU is cheaper than one doing the same work at 10% CPU. On a small embedded board this is the axis that decides whether a sensor loop is feasible at all; on a larger server it is the axis that decides how much else can share the machine.

These four axes are not independent. A driver can trade throughput for latency: batch more work per interrupt and you raise throughput but also raise the latency each packet sees. It can trade latency for CPU: poll more often and you lower latency but raise CPU cost. It can trade responsiveness for throughput: hold the lock longer per call and each call is cheaper but contention rises. You cannot minimise all four at once; performance work is about choosing the trade-off that matches the workload.

That, in a sentence, is why measurement matters. Without numbers, you cannot know which axis is actually the problem, and you cannot know whether your change improved the right one.

33.4.2 A Worked Example: Where Do These Axes Show Up?

Consider a network driver handling a small flow of 100 packet-per-second control traffic and a large flow of 1 gigabit-per-second bulk transfer at the same time. Each flow needs a different performance profile.

The small flow wants low latency. A 1 ms spike on a control packet may miss a deadline that the higher layer cares about. For this flow, you would rather handle each packet immediately than batch. The driver's RX path

should wake up the protocol layer on every packet, the interrupt should be serviced with low delay, and any locks the packet passes through should be held briefly.

The large flow wants high throughput and is relatively insensitive to latency. Each packet in the bulk transfer can sit in a receive queue for tens of microseconds without anyone noticing, and the system benefits if the driver batches interrupt processing to reduce per-packet overhead. For this flow, interrupt coalescing is your friend and a dedicated taskqueue thread makes sense.

A driver serving only one of these flows can tune for it. A driver serving both has to pick a compromise, or maintain multiple queues with different policies, or provide a tuning knob that lets the operator choose. In all three cases, the driver author needs to know which axis matters for the workload the driver will actually face.

Real FreeBSD drivers do make this kind of distinction. The `iflib(9)` framework that many modern network drivers use provides both a fast-path receive for low-latency delivery and a `rxeof` batching loop for throughput. The `cam(4)` storage subsystem distinguishes between synchronous I/O that blocks a thread and asynchronous I/O that does not. The `hwpmc(4)` module distinguishes between counting mode (low-overhead, always on) and sampling mode (higher overhead, diagnostic). Every mature driver tree in FreeBSD has these distinctions built in somewhere; noticing them is part of learning to read kernel code as a performance engineer.

33.4.3 When Optimisation Is Worth Doing

Not every driver needs tuning. A GPIO driver that toggles a pin a few times a second is doing a fine job even if every operation takes a millisecond. A debug pseudo-device that logs one line per hour is as fast as it needs to be. An ACPI-hotplug-notification driver that fires once per week spends almost no time in any state. For drivers like these, spending a day profiling would be a day wasted.

The drivers that usually merit tuning share some common features. They sit in a *hot path*, a sequence of calls that a high-rate or latency-sensitive workload passes through many times per second. They are in the ladder from the userland syscall to the hardware, with enough layers above and below them to make the driver a measurable fraction of total cost. They are on devices whose hardware is fast enough that the driver, not the device, becomes the bottleneck. And they are visible to users or to tests, so that a regression will be noticed and a speedup will be valued.

The classic examples are high-speed network drivers (10G, 25G, 40G, 100G), NVMe storage drivers, virtualisation I/O drivers (`virtio-net`, `virtio-blk`), audio drivers with tight latency budgets, and on embedded systems any driver in the main control loop of the product. If a driver is on this list, measuring it and tuning it pays back many times over. If it is not, tune only when a measurement (or a complaint from a user) points at the driver as the problem.

33.4.4 When Optimisation Is Premature

Knuth's remark that *premature optimisation is the root of all evil* was about the middle layer of a program, not its outermost design decisions. The saying applies to drivers in a particular way: it is almost always wrong to spend effort optimising code before it works correctly, before it is measured, or before a goal has been set. The reason is not that optimisation is never warranted; it is that early optimisation tends to make code harder to reason about, harder to debug, and harder to change when the real bottleneck turns out to be somewhere else.

Concretely, the following are red flags for premature optimisation in a driver:

- Hand-written SIMD code in a driver whose throughput is measured in thousands, not billions, of operations per second.
- A carefully cache-line-aligned softc in a driver that has at most one outstanding operation per CPU at a time.
- A lock-free ring buffer in a driver whose real bottleneck is a `malloc(9)` call per operation.
- A per-CPU counter farm in a driver that runs one operation per second.
- An elaborate taskqueue scheme in a driver whose interrupt handler runs in 2 microseconds.

In each case the technique is real and legitimate, but it has been chosen without evidence. The effort costs something (complexity, reviewability, risk of bugs), and it produces a benefit only when evidence shows it is needed. The discipline is not *never optimise*; it is *measure, then decide*.

33.4.5 Setting Measurable Goals

A performance goal has four parts: the **metric**, the **target**, the **workload**, and the **environment**.

The metric names what you are measuring. *Median read() latency, packets per second forwarded, interrupts per second serviced, CPU percentage at full load, peak kernel memory used, or P99 worst-case latency of the probe-to-attach path* are all valid metrics. Each is a single number produced by a single procedure.

The target names the value you want the metric to have. *Below 20 microseconds, above 1 million packets per second, under 5% CPU, under 4 MB of kernel memory at peak*. The target should be concrete and measurable.

The workload names the conditions under which the metric is produced. *read() calls issued at 10,000 Hz with M_WAITOK allocation, one TCP stream at the line rate on a 10 Gbps NIC, a 4K random-read test against a 16 GB file*. The workload should be reproducible; if no one can rerun it, the measurement is not useful.

The environment names the machine, the kernel, and the surrounding processes. *amd64, 8-core Xeon at 3.0 GHz, FreeBSD 14.3 GENERIC kernel, otherwise idle*. Two different machines will give different numbers for the same driver, and a measurement that is valid on one environment may be invalid on another.

A goal of the form *the perfdemo driver's median read() latency, measured across 100,000 calls on an idle amd64 Xeon E5-2680 at 3.0 GHz running FreeBSD 14.3 GENERIC, with one userland reader thread pinned to CPU 1, should be below 20 microseconds* is a complete performance goal. It has a metric, a target, a workload, and an environment. You can run it, record the result, compare a tuning change, and decide whether you are done.

A goal of the form *make the driver faster* is useless. It has no metric, no target, no workload, no environment. You can work on it for a year and still not know when to stop.

The discipline of writing the goal down before you start is where most of the quality in a performance project comes from. If you cannot write the goal, you are not ready to optimise. The rest of the chapter will assume you have one, and will teach you the tools that let you check it.

33.4.6 A Note on “Fast” and “Correct”

Two failures of performance work deserve to be named early.

The first is that a driver that is fast in one dimension can be bad overall. A driver that achieves blistering throughput by dropping occasional packets silently is not faster; it is broken. A driver that halves its latency

by skipping a safety check in the fast path is not faster; it is risky. A driver that reduces CPU by deferring free until unload is not faster; it is leaking. Every optimisation must preserve correctness, including the correctness of error paths and the behaviour under pathological input. An optimisation that trades correctness for speed is a regression, full stop.

The second failure is that a driver that is fast today can be a debugging nightmare tomorrow. Performance work often introduces complexity: pre-allocation pools, multiple locks, batched operations, inline assembly for atomics, careful memory layouts. Each of these is a future debugging cost. Before you commit to a complex optimisation, ask whether a simpler one is good enough. The driver that lives in the tree for years is not usually the one that was fastest in the benchmark; it is the one a maintainer can still read and reason about after the original author has moved on.

We will return to both points in Section 8 when we talk about shipping a tuned driver.

33.4.7 Exercise 33.1: Define Performance Goals for a Driver You Have

Pick a driver you have written in earlier chapters. `perfdemo` will appear in Section 2, so for now choose an earlier one: `nullchar` from Chapter 7, the ring-buffer character driver from Chapter 12, or the `ed` from Chapter 32. For that driver, write performance goals for at least two of the four axes (throughput, latency, responsiveness, CPU cost). For each goal, include a metric, a target, a workload, and an environment. Keep the goals in your logbook.

You do not have to have measured the driver yet. The exercise is the writing of the goal. If you cannot decide on a target, write *unknown, to be measured in Section 2* and come back.

The act of writing a specific goal sharpens what the rest of the chapter is trying to teach. Readers who skip this exercise often find Sections 2 through 7 feel abstract; readers who do it find that each new tool has an obvious use.

33.4.8 Wrapping Up Section 1

We opened with the claim that every driver is a performance contract whether the author writes one or not. The four axes of throughput, latency, responsiveness, and CPU cost give you a vocabulary for writing that contract down. The discipline of measurable goals, pinned to a metric, target, workload, and environment, gives you a way to know whether the contract is being kept. The reminders about premature optimisation and about the relationship between speed and correctness keep the work honest.

The rest of the chapter is the toolkit. In the next section we look at the kernel's measurement primitives: timing functions, counters, `sysctl` nodes, and the FreeBSD tools (`sysctl`, `dtrace`, `pmcstat`, `ktrace`, `top`, `sysstat`) that expose the metrics your goal depends on.

33.5 Section 2: Measuring Performance in the Kernel

Measurement in userland is the world most programmers grow up in. You wrap a function call with a timer, you run it a million times, you print the average. Measurement in the kernel is not so different in shape, but it is more sensitive in practice. Kernel code runs at the boundary of the system; a careless measurement can slow down the very thing it is trying to measure, or worse, change the behaviour of the system in ways that make the

measurement meaningless. The goal of this section is to teach you how to measure inside the kernel without contaminating the measurement.

We will move through four things: the kernel's time functions, the counter primitives that let you accumulate events cheaply, the tools that read those counters from userland, and the habits of instrumentation that keep the measurement honest.

33.5.1 Kernel Time Functions

The kernel exposes several functions that return the current time. They differ in precision, cost, and whether they advance monotonically. The declarations are in `/usr/src/sys/sys/time.h`.

There are two orthogonal choices when you pick one of these. The first choice is **format**:

- `*time` variants return a `struct timespec` (seconds and nanoseconds) or `struct timeval` (seconds and microseconds).
- `*uptime` variants return the same format, but measured from the system's boot rather than from the wall clock. Uptime does not jump when the administrator adjusts the wall clock; wall time does.
- `*bintime` variants return a `struct bintime` (a fixed-point binary fraction of a second) and `sbintuptime()` returns a `sbintime_t`, a signed 64-bit fixed-point value. These are the kernel's internal, highest-resolution formats.

The second choice is **precision versus cost**. FreeBSD distinguishes between a *fast but imprecise* path and a *precise but more expensive* path:

- Functions with the **get** prefix (`getnanotime()`, `getnanouptime()`, `getbinuptime()`, `getmicrotime()`, `getmicrouptime()`, `getsbinuptime()`) return a value cached at the last timer tick. They are very fast, typically just a few loads from a cache-resident global. Their precision is on the order of 1/hz, which on a default FreeBSD system is about 1 millisecond.
- Functions without the **get** prefix (`nanotime()`, `nanouptime()`, `binuptime()`, `microtime()`, `microuptime()`, `sbintuptime()`) call into the selected timecounter hardware and return a value accurate to the timecounter's resolution, often tens of nanoseconds. They cost more, typically tens to hundreds of nanoseconds, and on some hardware may serialise the CPU pipeline.

The golden rule: use the `get*` variants whenever 1 millisecond of precision is enough, and the non-`get` variants when you need real precision. A driver measuring its own hot-path latency generally needs the non-`get` variants. A driver timestamping a rare event, or recording the time of a state change, can almost always use the `get*` variants.

Here is the short version of the decision tree:

- Need wall-clock time, millisecond precision: `getnanotime()`.
- Need uptime (monotonic), millisecond precision: `getnanouptime()`.
- Need uptime, nanosecond precision: `nanouptime()`.
- Need uptime, highest resolution, minimal cost per call: `sbintuptime()` (or `getsbinuptime()` when millisecond precision is fine).
- Need to compute a duration: subtract two `sbintuptime()` values and convert to microseconds or nanoseconds at the end.

A representative timing in a driver read path might look like this:

```
#include <sys/time.h>
#include <sys/sysctl.h>

static uint64_t perfdemo_read_ns_total;
static uint64_t perfdemo_read_count;

static int
perfdemo_read(struct cdev *dev, struct uio *uio, int ioflag)
{
    sbintime_t t0, t1;
    int error;

    t0 = sbinuptime();

    /* ... the real work of reading ... */
    error = do_the_read(uio);

    t1 = sbinuptime();

    /* Convert sbintime_t difference to nanoseconds.
     * sbt2ns() is defined in /usr/src/sys/sys/time.h. */
    atomic_add_64(&perfdemo_read_ns_total, sbttons(t1 - t0));
    atomic_add_64(&perfdemo_read_count, 1);

    return (error);
}
```

A few things are worth noticing. The timestamps are captured with `sbinuptime()`, not `getsbinuptime()`, because we are measuring microsecond-scale durations. The accumulator and counter are 64-bit and updated with `atomic_add_64()`, so concurrent readers do not lose updates. The conversion from `sbintime_t` to nanoseconds uses `sbttons()`, a macro declared in `/usr/src/sys/sys/time.h`; we do not do the division on the hot path. And neither the timestamps nor the accumulation print anything; the data goes into counters that a `sysctl` reads on demand.

The pattern generalises. Timestamp at the boundaries of the operation you care about, accumulate the difference, expose the accumulator through `sysctl`, and compute derived metrics (average, throughput) in userland.

33.5.2 Time Stamp Counters and Other High-Precision Sources

Below the `nanotime` and `sbinuptime` family sits the hardware that actually feeds them: the processor's Time Stamp Counter (TSC) on x86, the Generic Timer on arm64, and their equivalents on other architectures. FreeBSD abstracts these sources behind a *timecounter* interface; the kernel selects one at boot

and the `sbnuptime` path reads it. You can see which timecounter your system is using with `sysctl kern.timecounter`:

```
# sysctl kern.timecounter
kern.timecounter.tick: 1
kern.timecounter.choice: ACPI-fast(900) HPET(950) i8254(0) TSC-low(1000)
↪ dummy(-1000000)
kern.timecounter.hardware: TSC-low
```

On a modern amd64 machine the TSC is almost always the chosen source. It advances at a constant rate regardless of CPU frequency (on processors that support invariant TSC, which is essentially everything since the mid-2000s), it is cheap to read (a single instruction), and its resolution is on the order of the CPU clock period, roughly 0.3 nanoseconds at 3 GHz.

A driver rarely needs to read the TSC directly. `sbnuptime()` already wraps it. But when you are debugging the kernel's own timing code, or when you need a timestamp that is *only* a load from a register with no further arithmetic, the kernel provides `rdtsc()` as a static `__inline` function in `/usr/src/sys/amd64/include/cpufunc.h`. Its use is almost always a mistake in driver code: you lose the kernel's unit conversions, you lose portability across architectures, and you gain a few nanoseconds. Reach for `sbnuptime()` instead; the portability and abstraction pay for themselves.

On arm64, the equivalent of the TSC is the Generic Timer's counter register, read via the ARM-specific `READ_SPECIALREG` macros in `/usr/src/sys/arm64/include/cpu.h`. The kernel exposes it through the same `sbnuptime()` abstraction, so a driver written to `sbnuptime()` is portable across the two architectures without change. This is one of the small but meaningful benefits of staying on the abstractions FreeBSD provides.

A subtle point: the different timecounter choices have different costs and different precisions. On the Intel and AMD generations currently in service, HPET is slow (on the order of hundreds of nanoseconds to read) but high-precision, ACPI-fast is fast but lower-precision, and TSC is fast and precise, which is why the kernel prefers it when available. If `kern.timecounter.hardware` shows anything other than TSC on an amd64 machine, something on the system disabled it, and `sbnuptime()` calls will be more expensive than you expect. Check `dmesg | grep timecounter` early in any performance investigation. See Appendix F for a reproducible benchmark that rotates through each timecounter source.

33.5.3 Composition: Timing, Counting, and Aggregating Together

The examples above showed timing (a pair of `sbnuptime()` calls) and counting (a `counter_u64_add()` increment) separately. Real driver measurement almost always composes the two: you want to know *how many* of each kind of operation happened *and* what the *latency distribution* of each kind looked like. The composition is simple once the primitives are in hand.

A pattern that appears in many FreeBSD drivers is the counter-and-histogram pair. For every operation, you increment a counter and add its latency to a histogram bucket. The histogram is represented as an array of `counter_u64_t` values, one per bucket, with bucket boundaries chosen for the expected range. On the hot path, you do one counter increment for total count and one counter increment for the bucket the latency fell in:

```

#define PD_HIST_BUCKETS 8

static const uint64_t perfdemo_hist_bounds_ns[PD_HIST_BUCKETS] = {
    1000,          /* <1us    */
    10000,         /* <10us   */
    100000,        /* <100us  */
    1000000,       /* <1ms    */
    10000000,      /* <10ms   */
    100000000,     /* <100ms  */
    1000000000,    /* <1s     */
    UINT64_MAX,    /* >=1s    */
};

static counter_u64_t perfdemo_read_hist[PD_HIST_BUCKETS];
static counter_u64_t perfdemo_read_count;

static int
perfdemo_read(struct cdev *dev, struct uio *uio, int ioflag)
{
    sbintime_t t0, t1;
    uint64_t ns;
    int error, i;

    t0 = sbinuptime();
    error = do_the_read(uio);
    t1 = sbinuptime();

    ns = sbttens(t1 - t0);
    counter_u64_add(perfdemo_read_count, 1);
    for (i = 0; i < PD_HIST_BUCKETS; i++) {
        if (ns < perfdemo_hist_bounds_ns[i]) {
            counter_u64_add(perfdemo_read_hist[i], 1);
            break;
        }
    }

    return (error);
}

```

The hot path does one `sbintuptime` pair (on the order of a few tens of nanoseconds each on typical FreeBSD 14.3-amd64 hardware), one multiplication to convert to nanoseconds, a linear scan of eight bucket bounds (branch-predicted well after warm-up), and two counter increments. As a rough order-of-magnitude figure on the same class of machine, the total overhead stays comfortably under a microsecond, which is acceptable for most read paths and small enough to leave on in production. See Appendix F for a reproducible benchmark of

the underlying `sbinuptime()` cost on your own hardware.

Exposing the histogram via `sysctl` is a small extension of the procedural-`sysctl` pattern from Section 7. You fetch each bucket's counter, copy them into a local array, and hand the array to `SYSCTL_OUT()`. Userland reads the array and plots it.

If the expected latency range is known, the linear bucket search can be replaced by a constant-time mapping. For power-of-ten buckets, $\log_{10}$ of the latency lands you in the right bucket; for power-of-two buckets, a single `fls` instruction suffices. The constant-time variants are only worth the complexity when the linear search appears as a real cost in a profile, which in practice happens only for very hot paths with many buckets.

33.5.4 A Complete Instrumented Read Path

Putting the above together, a fully instrumented `perfdemo_read()` that captures call count, error count, total bytes, total latency, and a latency histogram looks like this:

```
#include <sys/counter.h>
#include <sys/time.h>

struct perfdemo_stats {
    counter_u64_t reads;
    counter_u64_t errors;
    counter_u64_t bytes;
    counter_u64_t lat_ns_total;
    counter_u64_t hist[PD_HIST_BUCKETS];
};

static struct perfdemo_stats perfdemo_stats;

static int
perfdemo_read(struct cdev *dev, struct uio *uio, int ioflag)
{
    sbintime_t t0, t1;
    uint64_t ns, bytes_read;
    int error, i;

    t0 = sbinuptime();
    bytes_read = uio->uio_resid;
    error = do_the_read(uio);
    bytes_read -= uio->uio_resid;
    t1 = sbinuptime();

    ns = sbttions(t1 - t0);
    counter_u64_add(perfdemo_stats.reads, 1);
    counter_u64_add(perfdemo_stats.lat_ns_total, ns);
    counter_u64_add(perfdemo_stats.bytes, bytes_read);
```

```

    if (error)
        counter_u64_add(perfdemo_stats.errors, 1);
    for (i = 0; i < PD_HIST_BUCKETS; i++) {
        if (ns < perfdemo_hist_bounds_ns[i]) {
            counter_u64_add(perfdemo_stats.hist[i], 1);
            break;
        }
    }

    return (error);
}

```

This is the shape of a production-grade measurement. It counts, it times, it records distributions, and it costs a handful of nanoseconds per call on the hot path. The derived metrics (mean latency, P50, P95, P99 estimated from the histogram) are computed in userland on demand. The counter(9) values scale to tens of cores with no contention; the linear bucket search is cache-friendly; the conditional errors counter adds no cost on the success path.

In the sections that follow, we will tune other parts of the driver, but the measurement scaffolding above is the baseline we measure *against*. Every change we make will be judged against the numbers the scaffold produces.

33.5.5 Counters: Simple, Atomic, Per-CPU

Kernel code often needs to count things: operations completed, errors, retries, interrupt storms. FreeBSD offers three grades of counter primitive, in order of increasing complexity and increasing scalability.

A plain `uint64_t` updated with `atomic_add_64()`. This is the simplest counter. It is correct under concurrency, it is cheap on modern hardware, and it works everywhere. At low concurrency (tens of thousands of updates per second from a handful of CPUs), it is a fine default. At high concurrency (hundreds of thousands of updates per second from many CPUs), the cache line holding the counter becomes a contention point and the atomic operation starts to show up in profiles. For most driver counters, the plain atomic is the right choice.

A counter(9) value (`counter_u64_t`). This is a per-CPU counter with a simple fetch interface. You allocate it with `counter_u64_alloc()`, update it with `counter_u64_add()`, and read it with `counter_u64_fetch()` (which sums across all CPUs). Because each CPU updates its own memory, there is no contention on the update path. The trade-off is that the fetch iterates over all CPUs, so reading the counter is slightly more expensive. For drivers whose counters are updated on fast paths from many CPUs, counter(9) is a better fit than a plain atomic. The declarations are in `/usr/src/sys/sys/counter.h`.

A `DPCPU_DEFINE(9)` variable. DPCPU stands for *dynamic per-CPU*. It lets you define any type of variable, not just a counter, with per-CPU storage. You declare the variable with `DPCPU_DEFINE(type, name)`, access the current CPU's copy with `DPCPU_GET(name)` and `DPCPU_SET(name, v)`, and sum across all CPUs with `DPCPU_SUM(name)`. It is the most flexible of the three primitives, and it is what counter(9) is implemented on top of. Reach for it when you need per-CPU state that is not just a counter, or when you need a counter and have a specific reason to bypass the counter(9) abstraction. The declarations are in `/usr/src/sys/sys/pcpu.h`.

Here is a practical example. Suppose `perfdemo` has three counters: total reads, total errors, and total bytes delivered. The counter(9) form is:

```
#include <sys/counter.h>

static counter_u64_t perfdemo_reads;
static counter_u64_t perfdemo_errors;
static counter_u64_t perfdemo_bytes;

/* In module init: */
perfdemo_reads = counter_u64_alloc(M_WAITOK);
perfdemo_errors = counter_u64_alloc(M_WAITOK);
perfdemo_bytes = counter_u64_alloc(M_WAITOK);

/* On the fast path: */
counter_u64_add(perfdemo_reads, 1);
counter_u64_add(perfdemo_bytes, bytes_delivered);
if (error)
    counter_u64_add(perfdemo_errors, 1);

/* In module fini: */
counter_u64_free(perfdemo_reads);
counter_u64_free(perfdemo_errors);
counter_u64_free(perfdemo_bytes);

/* In a sysctl handler that reports the values: */
uint64_t v = counter_u64_fetch(perfdemo_reads);
```

The counter(9) API keeps the hot-path cost small and constant regardless of how many CPUs your driver serves. On a 64-core server it is a transformational improvement over a single atomic. On a 4-core laptop it is a small but real improvement. On an embedded single-core board it is no better than an atomic. Use it when there will be contention, or when you want to be ready for the day a driver graduates from a 2-core test machine to a 64-core production one.

33.5.6 Exposing Metrics via sysctl

A counter that nobody can read is a counter that does not exist. FreeBSD's `sysctl(9)` subsystem is the standard way to expose driver metrics to userland, and the chapter covers the production patterns in detail in Section 7. For now, the simplest pattern is:

```
SYSCCTL_NODE(_hw, OID_AUTO, perfdemo, CTLFLAG_RD, 0,
    "perfdemo driver");

SYSCCTL_U64(_hw_perfdemo, OID_AUTO, reads, CTLFLAG_RD,
```



```
&perfdemo_reads_value, 0,  
"Total read() calls");
```

For a counter(9) variable, the idiomatic way is to use a procedural `sysctl` that calls `counter_u64_fetch()` on each read. We will write that pattern out in Section 7. For now, the `SYSCTL_U64` macro above is enough for a plain `uint64_t` or for the `counter_u64_fetch()` result cached into a regular variable during a periodic callout.

From userland, `sysctl hw.perfdemo.reads` returns the value, and scripts can poll at intervals, compute rates, and produce plots.

33.5.7 The Measurement Tools FreeBSD Gives You

On top of the primitives above, FreeBSD ships several userland tools that read the kernel's exposed metrics and present them in useful ways. A short tour follows; we will return to the important ones in later sections.

sysctl(8). Reads any `sysctl` node. The most basic measurement interface. It is the tool for polling a counter every second to compute a rate, for checking a one-shot metric, and for dumping a subtree for later comparison. A script like `while sleep 1; do sysctl hw.perfdemo; done | awk ...` is a surprisingly common first step.

top(1). Shows processes, threads, CPU use, memory use. With the `-H` flag it shows kernel threads, including interrupt and taskqueue threads. Useful when a driver has its own thread whose CPU use you want to watch, or when you want to see an `ithread` spinning on interrupts.

sysstat(1). An older but still-useful family of monitoring views. `sysstat -vmstat` shows CPU, disk, memory, and interrupt rates. `sysstat -iostat` focuses on storage. `sysstat -netstat` focuses on the network. When you want a live monitor without writing a script, `sysstat` is often faster to reach for than anything more elaborate.

vmstat(8). A per-second view of system-wide statistics. `vmstat -i` lists interrupt rates by vector. `vmstat -z` lists UMA zone activity, which is invaluable when tracking down kernel memory pressure.

ktrace(1). Traces system-call entries and exits for specific processes. It is lower-level than `DTrace` and predates it, but remains useful when the interaction you want to see crosses the userland-kernel boundary and does not require kernel-side custom probes.

dtrace(1). The Swiss army knife of FreeBSD kernel observation. Covered in depth in Section 3.

pmcstat(8). The hardware-performance-counter tool. Covered in depth in Section 4.

The important insight is that these tools are not competitors. Each one is best at a particular kind of measurement. You learn performance work by learning which tool answers which question, and reaching for that one first.

33.5.8 The Heisenberg Effect, Practically

The joking reference in the chapter's introduction was to a real problem: the act of measuring changes what is measured. In kernel code the problem has teeth, and you need to know its shape.

The first form of the problem is **direct overhead**. Every measurement has a cost. If you add a `printf` inside a read path that runs at 100,000 reads per second, you have added tens of microseconds per read, and the

driver's effective rate drops to a fraction of what it would be without the print. If you time every operation with `sbinuptime()` and increment an atomic counter, you have added perhaps 50 nanoseconds per call; for most drivers this is acceptable, but in a nanosecond-budget hot path even this is too much. The rule of thumb: estimate the cost of each measurement probe and decide whether the driver can afford it on the path you are probing.

The second form is **cache and lock contention**. A single atomic counter updated by all CPUs is a cache line bouncing between them. On low-contention paths this costs a few nanoseconds. On high-contention paths it can cost hundreds of nanoseconds per update and dominate the operation. A counter that is supposed to be cheap can become the bottleneck it is trying to measure. The fix, as we saw above, is `counter(9)` or a DPCPU variable, both of which update per-CPU memory.

The third form is **observer-induced behaviour change**. This is the most insidious. Adding instrumentation can change what the system does beyond the direct cost. A `printf` may cause a timer that was about to fire to miss its deadline. A DTrace probe that turns out to use a slower code path can disable a fast-path optimisation. A breakpoint set in a debugger can hide a race that the unprobed code exhibits. Good measurement design tries to minimise observer effects, but the awareness that they exist is what separates a disciplined measurement practice from a superstitious one.

The practical consequence is that you should always *disable* measurement code when measuring the baseline, unless the measurement is itself what you are evaluating. If you want to know how fast a driver can go, run it with the minimum instrumentation needed to report the result. If the minimum instrumentation is expensive, reduce it. This is also why `counter(9)` is preferred over atomics for hot-path counting: not just because it is faster, but because it perturbs the system less.

33.5.9 Instrumenting Without Contamination

The practical advice on adding measurement code distils into a small checklist. Before you add a probe, ask:

1. **What metric is this probe producing?** Every probe must correspond to a concrete line in the goals document. If you cannot name it, do not add it.
2. **On what path does the probe live?** A probe on the module-load path is free. A probe on a path that runs once per operation on a high-throughput driver must be cheap enough to fit the budget.
3. **What does the probe cost?** An atomic counter is a few nanoseconds. A `counter(9)` is about the same, with better scaling. A `sbinuptime()` timestamp is tens of nanoseconds. A `printf` is tens of microseconds. A DTrace probe, when enabled, is hundreds of nanoseconds.
4. **Is the probe always on, or only during an instrumentation mode?** A counter that stays in production is a feature. A detailed trace is a tool for a tuning session.
5. **Does the probe cause contention?** Writing to a single cache line from many CPUs is a contention problem; reading it is not.
6. **Does the probe change ordering?** An added memory barrier can hide a race; an added sleep can hide a contention problem.

Most of this is common sense once you have seen one or two measurements go wrong. Writing it down as a checklist early saves the pain of re-discovery.

33.5.10 Exercise 33.2: Time the Read Path

Take a simple driver you have written earlier, or use the `perfdemo` scaffold we will introduce in the lab section. Add two `counter_u64_t` variables: `reads` and `read_ns_total`. In the read handler, timestamp with `sbinuptime()` at entry, do the work, timestamp again, add the nanosecond difference to `read_ns_total`, and increment `reads` by 1. Expose both via `sysctl`.

Build the driver. Load it. Run a userland loop that reads from the device 100,000 times. Then compute the average latency as `read_ns_total / reads` and record the number in your logbook. Next, remove the counter updates, build again, and rerun the test; if the timing tools allow, compare the wall-clock time of the 100,000-read loop with and without the counters. The difference is the probe overhead.

You will find that `counter(9)` updates are cheap enough to leave in production. That fact is one of the most useful practical takeaways of the chapter.

33.5.11 Wrapping Up Section 2

Kernel measurement uses the same primitives as userland measurement but with tighter constraints. The `get*` time functions are for cheap, low-precision timestamps; the non-`get*` functions are for precise, more expensive ones. Counter primitives come in three grades, from plain atomics to `counter(9)` to DPCPU variables, with increasing scalability and flexibility. Exposure to userland is through `sysctl(9)`. The tools that read the exposed state (`sysctl(8)`, `top`, `systat`, `vmstat`, `ktrace`, `dtrace`, `pmcstat`) each specialise. And the discipline of measuring without contaminating the measurement is as important as the primitives themselves.

The next section introduces DTrace, which is the most general and capable of the kernel measurement tools FreeBSD gives you. DTrace deserves a whole section of its own because it answers the *what is happening right now* question better than anything else.

33.6 Section 3: Using DTrace to Analyze Driver Behavior

DTrace is the single most useful tool in a kernel performance engineer's box. Its combination of expressiveness, low overhead when disabled, and ability to see almost any function in the kernel without recompiling has no peer. Section 3 is the longest section of the chapter for this reason; the time you spend learning DTrace pays back on every driver you ever measure.

The section assumes no prior DTrace experience. Readers who have used DTrace on Solaris or macOS will find FreeBSD's implementation familiar in substance, with a few differences in provider availability and in how to enable it.

33.6.1 What DTrace Is

DTrace is a kernel facility that lets you attach small scripts to probes. A probe is a named point in the kernel at which you can observe what is happening: a function entry, a function return, a scheduler switch, an I/O completion, a packet arrival, a user-defined event in a driver. A script says *when this probe fires, do X*. The script is compiled into a safe bytecode that runs in the kernel's context, aggregated in memory, and read back into the userland `dtrace` process.

The design has four features that matter for driver work.

First, the probes are **pervasive**. Every non-static function in the kernel is a probe. So are scheduler events, every system call boundary, every lock operation, every I/O dispatch, and thousands of other points. You rarely need to add probes to find one you need.

Second, probes have **near-zero cost when disabled**. An unused probe is a NOP in the instruction stream; it does not slow the kernel. The overhead appears only when you enable the probe, and even then the enabled overhead is typically a few hundred nanoseconds per fire. You can leave your driver production-ready with a dozen SDT probes in it, and when DTrace is not running, the driver is exactly as fast as if the probes were not there.

Third, DTrace is **safe**. The scripts run in a constrained language that prohibits loops, arbitrary memory access, and any operation that could crash the kernel. A buggy script returns an error; it does not panic the system.

Fourth, DTrace **aggregates in the kernel**. You can keep per-thread counts, quantiles, average values, histograms, and similar summaries in the kernel and pull the summary out periodically. You do not need to emit every event to userland; most DTrace scripts emit aggregated summaries, which is why they scale to millions of events per second.

33.6.2 Enabling DTrace on FreeBSD

DTrace is part of the base system and works out of the box on most FreeBSD installations, but it requires that the kernel have DTrace support compiled in and that the relevant modules be loaded. On a GENERIC kernel, the support is present. To use it, load the provider modules:

```
# kldload dtraceall
```

The `dtraceall` module is a convenience that loads all the DTrace providers. If you only need a specific provider (for example, `fbt` for function-boundary tracing or `lockstat` for lock contention), you can load just that one:

```
# kldload fbt
# kldload lockstat
```

After loading, `dtrace -l` lists the available probes. Expect the list to be long; a typical GENERIC kernel has tens of thousands of probes available.

33.6.3 The Probe Name Format

Every DTrace probe has a four-tuple name of the form:

```
provider:module:function:name
```

The **provider** is the source of the probe: `fbt` for function-boundary tracing, `sdt` for statically defined tracepoints, `profile` for timed sampling, `sched` for scheduler events, `io` for I/O completion, `lockstat` for lock events, and several others.

The **module** is the kernel module the probe lives in: `kernel` for core kernel code, `your_driver_name` for your own module, `geom_base` for GEOM, and so on.

The **function** is the function the probe is in.

The **name** is the name of the probe within the function: for `fbt`, always `entry` or `return`.

So a probe like `fbt:kernel:malloc:entry` means *the entry point of the malloc function, in the kernel module, via the fbt provider*. A probe like `fbt:perfdemo:perfdemo_read:return` means *the return point of perfdemo_read, in the perfdemo module*.

Wildcards are allowed. `fbt:perfdemo::entry` means *all entry points in the perfdemo module*. `fbt:::entry` means *all function entries in the kernel*. Be careful with the latter; it is a lot of probes.

33.6.4 Your First DTrace One-Liner

The canonical DTrace introduction is to count how often a function is called. For `perfdemo`, once the driver is loaded:

```
# dtrace -n 'fbt:perfdemo:perfdemo_read:entry { @[probefunc] = count(); }'
```

This says: on every entry to any `perfdemo` function named like `perfdemo_read`, increment an aggregation keyed by function name. When you press Ctrl-C, DTrace prints the aggregation:

<code>perfdemo_read</code>	10000

Ten thousand calls were made to `perfdemo_read` during the trace window. You now know the call rate; divide by the wall-clock seconds you ran the trace for.

A slightly richer one-liner that measures the time spent in the function:

```
# dtrace -n 'fbt:perfdemo:perfdemo_read:entry { self->t = timestamp; }
           fbt:perfdemo:perfdemo_read:return /self->t/ {
               @t = quantize(timestamp - self->t); self->t = 0; }'
```

This says: on entry, store the timestamp in a thread-local variable `self->t`. On return, if we have a saved timestamp (the `/self->t/` predicate), compute the difference and add it to a quantised histogram. When you stop the trace, DTrace prints the histogram:

value	----- Distribution -----	count
512		0
1024	@@@@	1234
2048	@@@@@@@@@@@@@@@@@@@@	6012
4096	@@@@@@@@@@@@@@@@	4581
8192	@@@@	1198
16384	@	145

32768	29
65536	8
131072	2
262144	0

The histogram says the function mostly runs in the 1 to 8 microsecond range, with a long tail out to a quarter of a millisecond. You now have the distribution, not just the average, of the function's latency. This is already better information than most published benchmarks publish.

33.6.5 Aggregations: the DTrace Superpower

The quantize aggregator is one of several that DTrace provides. The ones you will use most often are:

- `count()`: how many times the event fired.
- `sum(value)`: total of value across all fires.
- `avg(value)`: mean of value.
- `min(value), max(value)`: extremes.
- `quantize(value)`: power-of-two histogram.
- `lquantize(value, lower, upper, step)`: linear histogram with custom bounds.
- `stddev(value)`: standard deviation.

Aggregations can be indexed. `@[probefunc] = count();` keys by function name. `@[pid] = count();` keys by process id. `@[execname] = sum(arg0);` sums a value by executable name. Indexed aggregations are how DTrace produces *top-N* style summaries without ever sending per-event data to userland.

33.6.6 Useful DTrace Patterns for Driver Work

Here are a handful of patterns that keep coming back in driver performance work. Each is a fill-in-the-blank template.

Count calls to a function by caller:

```
# dtrace -n 'fbt:perfdemo:perfdemo_read:entry {
    @[stack(5)] = count();
}'
```

The `stack(5)` argument captures the five-frame user stack at the time of the probe. The aggregation tells you who the typical caller is.

Measure the time a function spends in itself versus in its callees:

```
# dtrace -n '
fbt:perfdemo:perfdemo_read:entry { self->s = timestamp; }
fbt:perfdemo:perfdemo_read:return / self->s / {
    @total = quantize(timestamp - self->s);
}
```

```
    self->s = 0;
}
fbt:perfdemo:perfdemo_do_work:entry { self->w = timestamp; }
fbt:perfdemo:perfdemo_do_work:return / self->w / {
    @worktime = quantize(timestamp - self->w);
    self->w = 0;
}'
```

Running this with a concurrent read loop tells you how much of the read time is inside `perfdemo_do_work` versus in the rest of `perfdemo_read`.

Count errors by location:

```
# dtrace -n 'fbt:perfdemo:perfdemo_read:return / arg1 != 0 / {
    @errors[probefunc, arg1] = count();
}'
```

The `arg1` on a return probe is the function's return value. If the function returns an `errno` on failure, this aggregation shows you which errors happen how often.

Watch memory allocation:

```
# dtrace -n 'fbt:kernel:malloc:entry / execname == "perfdemo" / {
    @sizes = quantize(arg0);
}'
```

This aggregates the size argument (`arg0`) passed to `malloc(9)` by contexts running in the `perfdemo` module. It answers questions like *how big are the allocations my driver does?*

These patterns are small variations on a theme. Learn them by writing them, not by reading them. The chapter's lab in Section 3 will give you a concrete one to try.

33.6.7 Profile Sampling with the profile Provider

The `fbt` provider fires on every function entry or return, which is thorough but noisy. For *where is CPU time going* questions, the `profile` provider is often better. It fires at a regular rate (for example, 997 Hz, a prime so it does not synchronise with the timer interrupt) on every CPU, independent of what is running. A script keyed by kernel stack gives you a statistical profile of where the kernel is spending time.

```
# dtrace -n 'profile-997 / arg0 != 0 / {
    @[stack()] = count();
}'
```

The `/arg0 != 0/` predicate filters out idle CPUs (where `arg0` is the user-space PC, which is zero for kernel-thread idling). Ctrl-C after a minute, and you get a list of kernel stacks with counts. The top stack is where the kernel spent most time. Send the output to a flame-graph renderer (the `FlameGraph` tools are installable as a port), and you have a visual, hierarchical view of kernel CPU use.

33.6.8 Static Probes with SDT

fbt is broad but not always the right answer. The fbt provider has a probe for every non-static function in the kernel, which means you can observe any function you please, but it also means the probe names are compiler-generated and may change as the code does. For stable, named observation points, FreeBSD provides the **SDT** (Statically Defined Tracepoint) provider. An SDT probe is declared in the driver's source, has a stable name, and can be inspected by DTrace scripts just like any other probe. It fires only when enabled, and when disabled is a NOP.

A driver adds SDT probes like this:

```
#include <sys/sdt.h>

SDT_PROVIDER_DEFINE(perfdemo);
SDT_PROBE_DEFINE2(perfdemo, , , read_start,
    "struct perfdemo_softc *", "size_t");
SDT_PROBE_DEFINE3(perfdemo, , , read_done,
    "struct perfdemo_softc *", "size_t", "int");
```

The first argument is the provider name (perfdemo). The next two are the module and function grouping; we leave them empty to get perfdemo::`read_start` as the probe name. Then the probe name itself. Then the argument types, one per fired argument.

In the driver code, you fire the probe where it matters:

```
static int
perfdemo_read(struct cdev *dev, struct uio *uio, int ioflag)
{
    struct perfdemo_softc *sc = dev->si_drv1;
    size_t want = uio->uio_resid;
    int error;

    SDT_PROBE2(perfdemo, , , read_start, sc, want);

    error = do_the_read(sc, uio);

    SDT_PROBE3(perfdemo, , , read_done, sc, uio->uio_resid, error);

    return (error);
}
```

From a DTrace script, the probes now have readable names:

```
# dtrace -n 'perfdemo::read_start { @starts = count(); }
           perfdemo::read_done { @done = count(); }'
```


The stable names are the point. A change to `perfdemo_read`'s internal structure does not break the script; a DTrace user can write `perfdemo:::read_done` and know exactly what it means, which they could not say of `fbt:perfdemo:perfdemo_read:return`.

For drivers that are going to be observed often, a modest set of SDT probes at the operational boundaries (read, write, ioctl, interrupt, error, overflow, underflow) is a very small code cost and a large observability gain.

33.6.9 The lockstat Provider

One of DTrace's more specialised providers, `lockstat`, deserves a separate introduction because lock contention is one of the most common performance problems in concurrent drivers. Its probes are declared in `/usr/src/sys/sys/lockstat.h`. They fire on every acquire, release, block, and spin of every kernel mutex, rwlock, sxlock, and lockmgr lock.

The two probes you will use most often are:

- `lockstat:::adaptive-acquire` and `lockstat:::adaptive-release` for plain mutex (mtx) operations.
- `lockstat:::adaptive-block` when a thread had to sleep waiting for a contended mutex.
- `lockstat:::spin-acquire`, `lockstat:::spin-release`, `lockstat:::spin-spin` for spin mutexes.

The argument `arg0` to a `lockstat` probe is a `struct lock_object *` pointer. To get the lock's name, `((struct lock_object *) arg0)->lo_name`. A useful one-liner that shows the lock names with the most acquires over a measurement window:

```
# dtrace -n 'lockstat:::adaptive-acquire {
    @[((struct lock_object *)arg0)->lo_name] = count();
}'
```

A more useful one that measures how long the lock was *held*:

```
# dtrace -n '
lockstat:::adaptive-acquire {
    self->s[arg0] = timestamp;
}
lockstat:::adaptive-release / self->s[arg0] / {
    @[((struct lock_object *)arg0)->lo_name] = sum(timestamp - self->s[arg0]);
    self->s[arg0] = 0;
}'
```

Expect thread-local storage to grow if many distinct locks are acquired; the kernel caps it and DTrace will stop the script if it overflows. For most drivers, this script runs comfortably for a few minutes before it runs out.

A third useful one tracks **contention**, not just acquisition. An adaptive mutex that blocks because some other thread holds it fires `lockstat:::adaptive-block`. Counting those by lock name shows you which locks are actually contended:

```
# dtrace -n 'lockstat:::adaptive-block {
    @[((struct lock_object *)arg0)->lo_name] = count();
}'
```

If your driver's mutex shows up at the top of this list, you have found a real contention problem. If it does not, locking is not your problem and you can focus elsewhere. The data is more valuable than most of the advice you could give yourself about locking in the abstract.

33.6.10 A Longer DTrace Script

DTrace scripts can live in files (.d) and be invoked with `dtrace -s script.d`. Here is a longer script for measuring `perfdemo`'s read behaviour, with per-process aggregation and a predicate to ignore a specific process that is also running on the system:

```
/*
 * perfdemo-reads.d
 *
 * Aggregate perfdemo_read() timings per userland process.
 * Requires the perfdemo module to be loaded.
 */

#pragma D option quiet

fbt:perfdemo:perfdemo_read:entry
{
    self->start = timestamp;
    self->size = args[1]->uio_resid;
}

fbt:perfdemo:perfdemo_read:return
/ self->start && execname != "dtrace" /
{
    this->dur = timestamp - self->start;
    @durations[execname] = quantize(this->dur);
    @sizes[execname] = quantize(self->size);
    @count[execname] = count();
    self->start = 0;
    self->size = 0;
}

END
{
    printa("\nRead counts per process:\n%-20s %u\n", @count);
    printa("\nRead durations (ns) per process:\n%s\n%d\n", @durations);
}
```

```
    printa("\nRead request sizes per process:\n%s\n%d\n", @sizes);
}
```

The script uses three DTrace idioms that are worth calling out. First, `args[1]->uio_resid` accesses the second argument of `perfdemo_read` (the struct `uio *`) and reads its `uio_resid` field; DTrace understands kernel structures with their field names. Second, `self->...` is thread-local storage that carries data between an entry probe and the matching return probe. Third, the predicate `/ self->start && execname != "dtrace" /` filters out probes where we did not see the entry (for example, because the script started mid-call) and excludes DTrace's own reads of the driver (which would otherwise skew the results).

Invoke the script while a workload runs:

```
# dtrace -s perfdemo-reads.d
```

Let it run for a minute, then Ctrl-C. The three aggregated printouts at the end give you counts, duration histograms, and size histograms per process, all from a single script with essentially no effect on the workload's performance.

Scripts like this are where DTrace really shines. They answer questions a developer could not have answered without recompiling the kernel two decades ago.

33.6.11 The sched Provider for Scheduler Events

One of DTrace's less-advertised providers is `sched`, which fires on scheduler events: a thread being put on a run queue, a thread being taken off a run queue, a context switch, a thread being woken up, and related events. For driver performance work, `sched` answers questions about *latency in the scheduler*, which is the layer of the system that gets between your driver's wake-up call and the userland thread actually running.

The probes you will reach for most often are:

- `sched:::enqueue`: a thread is put on a run queue.
- `sched:::dequeue`: a thread is taken off a run queue.
- `sched:::wakeup`: a thread is awoken by another thread (often from an interrupt or a driver).
- `sched:::on-cpu`: a thread starts running on a CPU.
- `sched:::off-cpu`: a thread stops running on a CPU.

A common use in driver work is to measure the *wake-up-to-run latency*: how long from the moment the driver calls `wakeup(9)` on a sleep channel to the moment the user-land thread actually runs. A simple one-liner:

```
# dtrace -n '
sched:::wakeup / curthread->td_proc->p_comm == "mydriver_reader" / {
    self->w = timestamp;
}
sched:::on-cpu / self->w / {
    @runlat = quantize(timestamp - self->w);
    self->w = 0;
}'
```

This script records the timestamp when the reader process is awoken and again when it starts running on a CPU. The aggregation is the distribution of the delay between the two. On typical FreeBSD 14.3-amd64 hardware in our lab environment, an idle system usually shows this latency in the sub-microsecond range, while a contended one pushes it into the low tens of microseconds. If your driver's median read latency is under 10 microseconds but its P99 is in the hundreds, the scheduler is often where the tail comes from, and `sched:::wakeup` and `sched:::on-cpu` are the probes that prove it. See Appendix F for notes on reproducing this measurement across kernel configurations.

Another useful `sched` pattern is counting how often a thread is preempted off a CPU, keyed by the thread's name:

```
# dtrace -n '
sched:::off-cpu / curthread->td_flags & TDF_NEEDRESCHED / {
    @preempt[execname] = count();
}'
```

This tells you which threads are being preempted away by a higher-priority thread, which in an interrupt-heavy system points at the drivers whose handlers are forcing context switches. On a well-behaved system this count is low.

33.6.12 The io Provider for Storage Drivers

For drivers in the storage stack, the `io` provider is invaluable. It fires on buffer-cache and bio events: `io:::start` when a bio is dispatched, `io:::done` when it completes, `io:::wait-start` when a thread begins waiting, `io:::wait-done` when the wait finishes. The combination gives you end-to-end latency for every storage operation.

A classic one-liner to measure storage latency:

```
# dtrace -n 'io:::start { self->s = timestamp; }
io:::done / self->s / {
    @lat = quantize(timestamp - self->s);
    self->s = 0; }'
```

The histogram it produces is the latency distribution of every bio the system completed during the measurement window, in nanoseconds. For a NVMe-backed file system the median is tens of microseconds; for a SATA SSD it is hundreds; for a rotating disk it is milliseconds. If a driver you are investigating shows much higher latency than its peer, you have narrowed the problem.

A richer version aggregates by the device and operation:

```
# dtrace -n '
io:::start { self->s = timestamp; }
io:::done / self->s / {
    @lat[args[0]->bio_dev, args[0]->bio_cmd == BIO_READ ? "READ" : "WRITE"]
```

```
        = quantize(timestamp - self->s);
    self->s = 0;
}'
```

The script accesses the `bio` structure's fields through `args[0]`, which DTrace understands because the probe arguments are typed in the kernel. It partitions the latency distribution by device and operation direction, so you can see whether reads and writes have different distributions, or whether one device is dragging down the average.

The `io` provider's real strength is that it answers the *total* latency question: the time the application saw, not just the time the driver contributed. If your driver is fast but the system is slow, `io` helps localise the problem.

33.6.13 Combining Multiple Providers in One Script

A script is not limited to a single provider. DTrace's power is most evident when you combine providers to answer a question that no single provider could.

Consider this question: *on a system where the `perfdemo` driver is used by two processes, which process's reads cause the most time to be spent in the kernel?*

A script:

```
/*
 * perfdemo-by-process.d
 *
 * Measures cumulative kernel time spent handling perfdemo_read()
 * per userland process, using sched + fbt providers.
 */

#pragma D option quiet

fbt:perfdemo:perfdemo_read:entry
{
    self->start = timestamp;
    self->pid   = pid;
    self->exec  = execname;
}

fbt:perfdemo:perfdemo_read:return / self->start /
{
    @total_time[self->exec] = sum(timestamp - self->start);
    @count[self->exec]      = count();
    self->start = 0;
}
```

```

END
{
    printf("\nPer-process perfdemo_read() summary:\n");
    printf("%-20s %10s %15s\n", "PROCESS", "CALLS", "TOTAL_NS");
    printa("%-20s %@10d %@15d\n", @count, @total_time);
}

```

Run this script while the workload runs, Ctrl-C, and see a two-column summary of which process spent the most total time in your driver's read path. That is data a plain `top(1)` cannot give you; it attributes kernel time *per process*, not *per thread*, and only for your specific function.

33.6.14 Putting It All Together: Driver Trace Session

A typical DTrace session for a driver you are tuning follows a predictable sequence of scripts, each building on the previous one. A concrete workflow:

1. **Who is calling the driver?** Run `dtrace -n 'fbt:perfdemo:perfdemo_read:entry { @[execname, pid] = count(); }'` for thirty seconds, Ctrl-C. This tells you which userland process and PID is exercising the driver, which is useful context for the rest of the session.
2. **How long do calls take?** Run the `quantize` script from the chapter. The distribution tells you the expected range and whether there is a tail.
3. **If there is a tail, what causes it?** Run a script that keys the histogram by stack trace. Different stack traces take different paths; seeing them split shows you which path is slow.
4. **Which lock, if any, is contended?** Run the `lockstat::adaptive-block` script. If your mutex is in the top entries, lock contention is real.
5. **Where is the CPU time going?** Run the `profile-997` script for a minute. The top stacks tell you which functions dominate.
6. **What is the scheduler doing?** If wake-up latency matters, run the `sched::wakeup / sched::on-cpu` one-liner. If the distribution has a long tail, the scheduler is where the user-visible delays come from.

At the end of this sequence, you have a coherent picture: who calls, how fast, where time goes, what blocks, and how the scheduler handles the results. Each script is a few lines. The sequence is the whole DTrace-for-drivers workflow.

33.6.15 A Note on DTrace's Limitations

DTrace is capable but not infinite. A few limitations to keep in mind:

- The DTrace language has no loops. This is by design; it is what guarantees probes terminate. If you need something that feels like a loop, use aggregations instead.
- You cannot modify kernel state from a DTrace script. It is an observer, not a debugger. To intervene, you need different tools.

- Probes fire in a safe but constrained context. You cannot, for example, call arbitrary kernel functions from inside a probe.
- Large amounts of thread-local storage can overflow. The kernel reclaims stale TLS conservatively; a script that stashes data in `self->foo` without clearing it will eventually run out.
- Aggregations are kept in per-CPU buffers and merged lazily. If you print an aggregation and immediately exit, the last few events may not appear.
- Some providers are unavailable in VMs, in jails, or when MAC policies restrict tracing.

The last point is practically important: if your driver is under test in a bhyve guest or on a cloud VM, some providers may simply not work. The `fbt` and `sdt` providers usually work; the `profile` provider depends on kernel support; the `lockstat` provider depends on the kernel being compiled with the probes in place.

33.6.16 Exercise 33.3: DTrace Your Driver

Take the `perfdemo` driver and write a DTrace script (or a one-liner) that answers each of these questions:

1. How many reads per second is the driver doing over a one-minute window?
2. What is the P50, P95, and P99 read latency, in nanoseconds?
3. Which userland process is doing the reads?
4. Does the driver ever sleep on its mutex, and if so, for how long?
5. Of the CPU time in the kernel during the measurement, how much is spent inside `perfdemo` and how much elsewhere?

Save your scripts in the lab directory. The act of writing them is where DTrace becomes second-nature.

33.6.17 Wrapping Up Section 3

DTrace is the kernel performance engineer's default tool. Its probes are everywhere, its enabled cost is small, its aggregations scale, and its scripts are expressive without being dangerous. We looked at the probe name format, at one-liners and aggregations, at SDT probes for stable observation points, at the `lockstat` provider for lock contention, at profile sampling for CPU profiling, and at the shape of a longer `.d` script.

The next section turns from function-level observation to CPU-level observation. `pmcstat` and `hwpmc(4)` give you hardware-counter data that DTrace cannot: cycles, cache misses, branch mispredicts. They are the right tool when you know which function is hot and need to understand *why* the hardware is taking so long to run it.

33.7 Section 4: Using `pmcstat` and CPU Counters

Section 3 gave you a way to measure time and event counts at the function level. Section 4 is about measuring what the CPU hardware is doing while it runs those functions: instructions retired, cycles consumed, cache misses, branch mispredictions, memory stalls. These hardware events are the layer below function-level timing, and they explain the *why* behind a surprising `pmcstat` result. A function may be slow because it runs too many instructions, because it waits on memory, because it mispredicts branches, or because the CPU is stalled on a dependency. `pmcstat` tells you which.

The section introduces hardware counters, explains how `pmcstat(8)` uses them, walks through a sampling session from setup to interpretation, and closes with the limitations of the tool.

33.7.1 Hardware Performance Counters in One Page

Modern CPUs include a small set of hardware counters that the processor increments on specific events. The counters are programmable: you choose which event to count, start the counter, and read the value. Intel calls the subsystem *Performance Monitoring Counters* (PMCs). AMD calls it the same thing. ARM calls it the *Performance Monitor Unit* (PMU). They differ in detail but share a model.

On an x86-64 Intel or AMD CPU you typically have:

- A small number of fixed-function counters, each tied to a specific event (for instance, *instructions retired* or *unhalted core cycles*).
- A larger number of programmable counters, which can be configured to count any of hundreds of events.

Each counter is a 48-bit or 64-bit register that increments on every event. The events are vendor-specific and vary by processor generation. Common events include:

- **Cycles**: how many CPU cycles have passed. `cpu_clk_unhalted.thread` on Intel.
- **Instructions retired**: how many instructions completed. `inst_retired.any` on Intel.
- **Cache references** and **cache misses**: at various levels of the memory hierarchy. `llc-misses` is a commonly useful shorthand.
- **Branches** and **branch mispredictions**.
- **Memory stalls**: cycles where the pipeline was waiting on memory.
- **TLB misses**: cycles where virtual-to-physical translation missed the TLB.

The counters can be used in two modes. **Counting mode** just reads the counter at the end of a workload and gives you a total. **Sampling mode** configures the counter to fire an interrupt every N events; the interrupt handler captures the current PC and stack, and after the run you have a statistical sample of where in the code the events happened. Counting mode tells you the total; sampling mode tells you the distribution. Both are useful.

33.7.2 `hwpmc(4)` and `pmcstat(8)`

FreeBSD exposes the CPU counters through the `hwpmc(4)` kernel module and the `pmcstat(8)` userland tool. The module drives the hardware; the tool collects and presents the data. To use them:

```
# kldload hwpmc
# pmcstat -L
```

The first command loads the module. The second asks the tool to list the event names available on this machine. On an Intel Core i7 laptop the list is hundreds of entries long; on an arm64 board it is shorter but still substantive.

The event names are the main complication. Intel has its own naming, AMD has a different one, ARM has a third. The `pmcstat -L` command lists the names for your CPU. FreeBSD also provides a set of portable mnemonic

events that work on any supported processor: `CPU_CLK_UNHALTED`, `INSTRUCTIONS_RETIRED`, `LLC_MISSES`, and a few others. Prefer the portable mnemonics when your measurement does not depend on a vendor-specific event.

33.7.3 A First Sampling Session

The simplest `pmcstat` invocation runs a command under a sampled counter:

```
# pmcstat -S instructions -O /tmp/perfdemo.pmc sleep 30 &
# dd if=/dev/perfdemo of=/dev/null bs=4096 count=1000000 &
```

The `-S instructions` flag configures the counter to sample on instructions (a portable mnemonic for retired instructions). `-O /tmp/perfdemo.pmc` tells it to write the raw samples to a file. `sleep 30` is the workload; for thirty seconds the sampler runs. The `dd` runs in parallel and drives load on the `perfdemo` device.

When `sleep 30` finishes, `pmcstat` stops and the file `/tmp/perfdemo.pmc` holds the raw samples. You turn them into a summary with:

```
# pmcstat -R /tmp/perfdemo.pmc -G /tmp/perfdemo.graph
```

The `-R` flag reads the raw file; `-G` writes a callgraph summary. The callgraph is a text file in a format that a flame-graph renderer or a simple `sort | uniq -c` pipeline can consume.

You can also ask for a top-style view of the hottest functions:

```
# pmcstat -R /tmp/perfdemo.pmc -T
```

which prints a list sorted by sample count:

%SAMP	CUM	IMAGE	FUNCTION
12.5	12.5	kernel	perfdemo_read
8.3	20.8	kernel	uiomove
6.9	27.7	kernel	copyout
5.1	32.8	kernel	_mtx_lock_sleep
...			

The `%SAMP` column is the fraction of samples the function received. `perfdemo_read` dominated with 12.5%. `uiomove`, `copyout`, and `_mtx_lock_sleep` followed. Now you know where to focus. If `perfdemo_read` is doing most of the work and `_mtx_lock_sleep` is in the top five, your driver is likely contending on its mutex.

33.7.4 Sampling Event Choice

`instructions` is the default because it is portable and usually useful, but other events change the question you are asking. A few useful variants:

- **-S cycles**: samples on unhalted cycles. Tells you where the CPU spends wall-clock time. Usually the best starting event.
- **-S LLC-misses** (Intel): samples on last-level-cache misses. Tells you which functions are suffering from main-memory accesses.
- **-S branches** or **-S branch-misses**: tells you which functions are hot in the branch predictor.
- **-S mem-any-ops** (Intel): memory operation rate.

A common workflow is to run a session with `-S cycles` first, identify a suspect function, then rerun with `-S LLC-misses` to see whether that function is hot because of instructions or because of memory.

33.7.5 Callgraphs and Flame Graphs

A top-N function list tells you where the hot function is; a callgraph tells you *how you got there*. `pmcstat -G` writes a callgraph file, and the conventional format can be fed to Brendan Gregg's FlameGraph scripts to produce an SVG flame graph. A flame graph shows the stack traces that led to each sampled function, sized by sample count. It is the single most useful visualisation of a CPU profile I know of, and learning to read one is worth a couple of hours.

A practical invocation on FreeBSD, assuming you have installed the FlameGraph tools from the `sysutils/flamegraph` port:

```
# pmcstat -R /tmp/perfdemo.pmc -g -k /boot/kernel > /tmp/perfdemo.stacks
# stackcollapse-pmc.pl /tmp/perfdemo.stacks > /tmp/perfdemo.folded
# flamegraph.pl /tmp/perfdemo.folded > /tmp/perfdemo.svg
```

The `-g` flag tells `pmcstat` to include kernel stacks; `-k /boot/kernel` tells it to resolve kernel symbols against the running kernel. The `stackcollapse-pmc.pl` and `flamegraph.pl` scripts come from the FlameGraph tools. Open the resulting SVG in a browser; each box is a function, its width is the fraction of time spent in it, and the stack of boxes below it shows how execution reached it.

33.7.6 Interpreting a PMC Result

A `pmcstat` result is raw data, not a conclusion. You have to reason about what it means in the context of the workload and the goal. A few patterns are common enough to recognise.

A function at the top of %SAMP but with low IPC (instructions per cycle) is probably memory-bound. Compare the cycles count to the instructions count; if cycles are much higher than instructions, the CPU is stalling. Look at cache misses and TLB misses to confirm.

A function at the top with high IPC is doing a lot of work in a tight loop. That is either a legitimate hot loop you should leave alone, or an opportunity for algorithmic improvement. Run the LLC-misses and branches counters to see whether the hardware is happy.

_mtx_lock_sleep or turnstile_wait in the top five is a sign that a mutex is being contended on. Run `lockstat` (Section 3) to find out which one.

Many functions below the 1% threshold, and no clear hot function usually means the overhead is spread across the whole driver path. Look at the total CPU cost of the operation and decide whether the driver is doing too much per call, rather than doing one thing slowly.

A function that does not appear at all but that you expected to see may be compiled inline, may have been absorbed by a macro, or may simply not be on the hot path. Check whether the compiler optimised it away; a quick `objdump -d` on the module can confirm.

These patterns get easier with practice. The first few `pmcstat` sessions leave most readers confused. By the tenth, the patterns are familiar.

33.7.7 Reading a Callgraph in Detail

When you run `pmcstat -R output.pmc -G callgraph.txt`, the resulting file is a text-format callgraph: every sample's full stack trace, one per line, in reverse order (innermost frame last). A small excerpt:

```
Callgraph for event instructions
@ 100.0% 12345 total samples
perfdemo_read    <- devfs_read_f <- dofileread <- sys_read <- amd64_syscall <-
↳ fast_syscall_common
    at 45.2% 5581 samples
perfdemo_do_work <- perfdemo_read <- devfs_read_f <- dofileread <- sys_read <-
↳ amd64_syscall <- fast_syscall_common
    at 20.1% 2481 samples
_mtx_lock_sleep  <- perfdemo_read <- devfs_read_f <- dofileread <- sys_read <-
↳ amd64_syscall <- fast_syscall_common
    at 12.8% 1580 samples
```

Each entry shows the *leaf* function on the left, followed by the chain of callers. The percentage is the fraction of total samples in which that exact call chain was seen. The top entry shows where most of the CPU time ended up; the chain of callers shows *how* the profiler got there.

Three habits make callgraph reading productive.

First, **do not trust the percentages at face value for short sessions**. A sample count under a few hundred is noisy; percentages based on such counts can swing by ten or more between runs. Collect samples for long enough to get into the thousands before drawing conclusions.

Second, **follow the chain, not just the leaf**. A function that shows up at the top may be a common subroutine called from many places. The interesting question is often *which caller spent the most time calling this subroutine?* The callgraph answers that directly; the function list does not.

Third, **treat the root samples with caution**. The top of the stack is usually a syscall entry or an ithread wrapper. It is the caller of interest that tells you whose code is hot, not the common boilerplate at the stack's bottom.

33.7.8 Cross-Referencing `pmcstat` and DTrace

`pmcstat` gives you where the CPU is spending time. DTrace gives you what the functions are doing. A disciplined investigation uses both. A typical workflow:

1. Run `pmcstat -S cycles` for a minute while the workload runs. Identify the top three functions.
2. For each top function, run a DTrace script on its `fbt:::entry` and `fbt:::return` probes to get a call-rate and latency histogram.
3. Multiply call rate by mean latency to estimate the total CPU time the function receives. This number should roughly match the `pmcstat` fraction; if it does not, one of the two tools is lying (usually your predicate).
4. Pick the function whose impact is largest and instrument its body with more granular DTrace probes.

The two tools complement each other. `pmcstat` has the hardware-counter granularity; DTrace has the function-body granularity and the ability to aggregate by context (process, thread, syscall, lock). Used alone, each tells half a story. Used together, they triangulate the performance picture.

33.7.9 A Top-Down Performance Analysis Method

For amd64 CPUs, Intel publishes a *top-down microarchitecture analysis* (TMA) method that organises CPU performance into a tree of categories: front-end bound, back-end bound, bad speculation, retiring. Each category has sub-categories that narrow the diagnosis: memory-bound vs core-bound, bandwidth vs latency, branch mispredicts vs machine clears. The method is useful because it turns a list of hundreds of PMC events into a small hierarchy that points at the bottleneck.

FreeBSD's `pmcstat` does not produce a top-down report directly, but you can compute the relevant ratios by collecting the right events together:

- **Retiring rate:** `UOPS_RETIRED.RETIRE_SLOTS` divided by total issue slots.
- **Front-end bound:** stall cycles in the front end divided by total cycles.
- **Back-end bound:** stall cycles in the back end divided by total cycles.
- **Bad speculation:** branch mispredict cost divided by total cycles.

A `pmcstat` invocation that counts these events together:

```
# pmcstat -s cpu_clk_unhalted.thread -s uops_retired.retire_slots \
-s idq_uops_not_delivered.core -s int_misc.recovery_cycles \
./run-workload.sh 100000
```

The exact event names vary between CPU generations; `pmcstat -L` lists what your CPU supports. Compute the ratios manually from the `pmcstat` output. If retiring is under 30% of slots, your code is stalling; the other categories narrow the cause.

For most driver work this level of detail is overkill. The simpler `-S cycles` sampling session identifies the hot function, and a look at the function's source tells you whether it is dominated by memory accesses, arithmetic, branches, or locks. But when the simpler analysis runs out (you see a hot function, you cannot tell *why* it is hot), the top-down method is the systematic next step.

33.7.10 Counting Mode

`pmcstat -P` (counting, per-process) and `pmcstat -s` (system-wide counting) are the counting counterparts of `-S`. Counting mode is ideal when you have a short benchmark and want a single number. A typical invocation:

```
# pmcstat -s instructions -s cycles dd if=/dev/perfdemo of=/dev/null bs=4096
↪ count=10000
```

When `dd` finishes, `pmcstat` prints:

```
p/instructions p/cycles
  2.5e9         9.8e9
```

This tells you the `dd` ran 2.5 billion instructions in 9.8 billion cycles, an IPC of about 0.25. That is a low IPC; a modern CPU can sustain 3 or 4 instructions per cycle on friendly code. An IPC of 0.25 suggests the code is stalling heavily, usually on memory. A rerun with `-s LLC-misses` confirms:

```
p/LLC-misses p/cycles
  1.2e7         9.8e9
```

12 million cache misses over 9.8 billion cycles. Divide 9.8 billion by 1.2 million: about 8000 cycles per miss on average, which is roughly the cost of a DRAM fetch. That is consistent with memory-bound behaviour.

The workflow is: run a counting session to get the system-wide numbers, run a sampling session to find which function is responsible, then reason about why. Each session answers a different question.

33.7.11 Process-Attached vs System-Wide

`pmcstat -P` attaches to a process and reports its counters exclusively. `-s` counts system-wide. The distinction matters when you want to exclude other workloads. A process-attached measurement of `dd` gives you just `dd`'s cycles and instructions, not the whole system's. A system-wide measurement includes everything.

For driver work, system-wide sampling is often what you want, because the driver's CPU time is counted in kernel threads that are not the benchmark process. Use `pmcstat -s` with kernel-stack collection when you want to see kernel functions, and `pmcstat -P` when you want to focus on a single userland process.

33.7.12 pmcstat and Virtualisation

Hardware PMCs in virtual machines are complicated. Some hypervisors expose the host's PMCs to guests (KVM with `vPMC` support, Xen with PMC passthrough); some expose a filtered subset; some expose nothing at all. FreeBSD's `hwpmc(4)` will report errors on events that the underlying hardware does not expose. If `pmcstat -L` produces a short list on your VM, or if `pmcstat -s cycles` fails with an error, you are probably in a PMC-restricted environment.

The fallback is DTrace's `profile` provider. It samples at a fixed rate instead of on hardware events, so it works anywhere the kernel does, including in heavily virtualised environments. Its results are coarser (you cannot sample on cache misses, for example), but it tells you where CPU time is going, which is the most common question anyway.

33.7.13 Exercise 33.4: `pmcstat perfdemo`

On a FreeBSD machine with the `perfdemo` driver loaded and the `hwpmc` module available, run these three sessions and record the results in your logbook:

1. Counting mode: `pmcstat -s instructions -s cycles <your dd invocation>`. Note the IPC.
2. Sampling mode: `pmcstat -S cycles -O /tmp/perfdemo.pmc <your workload>`, then `pmcstat -R /tmp/perfdemo.pmc -T`. Note the top five functions.
3. If your system supports it, `pmcstat -S LLC-misses -O /tmp/perfdemo-miss.pmc <your workload>`, then `pmcstat -R /tmp/perfdemo-miss.pmc -T`. Compare to the cycle samples.

If `hwpmc` is not available, substitute DTrace's `profile-997` with `@[stack()] = count();` for the sampling sessions. The results will be less precise but still instructive.

33.7.14 Wrapping Up Section 4

Hardware performance counters are the layer below function-level timing. `hwpmc(4)` exposes them, `pmcstat(8)` drives them, and DTrace's profile provider is the portable fallback. Counting mode gives totals, sampling mode gives distributions, and a flame graph turns the samples into a shape the human eye can read. The data is raw; interpretation takes practice. But once the patterns are familiar, a `pmcstat` session answers questions no other tool can.

We have now finished the measurement half of the chapter. Sections 1 through 4 taught you how to know what your driver is doing. Sections 5 through 8 turn to what to do with that knowledge: how to buffer, align, and allocate for throughput; how to keep interrupt handlers fast; how to expose runtime metrics; and how to ship the optimised result.

33.8 Section 5: Buffering and Memory Optimization

Memory is where a surprising number of driver performance problems hide. A function that looks fine on paper can spend most of its cycles waiting for memory, thrashing the cache, fighting with another CPU for a shared line, or churning through the memory allocator because it frees and reallocates on every call. The techniques that fix these problems are well understood and well supported in FreeBSD; what takes experience is recognising when each one applies.

This section covers five memory topics, in order of increasing specificity: cache lines and false sharing, alignment for DMA and cache efficiency, pre-allocated buffers, UMA zones, and per-CPU counters.

33.8.1 Cache Lines and False Sharing

A modern CPU does not read memory one byte at a time. It reads in units of a **cache line**, typically 64 bytes on x86-64 and on arm64 (some arm64 implementations use 128-byte lines; check `CACHE_LINE_SIZE` in `/usr/src/sys/arm64/include/param.h` for your target). Every time the CPU reads a byte, it reads the whole cache line into its L1 data cache. Every time it writes a byte, it modifies the whole cache line.

This is almost always a win: spatial locality means the next few bytes you read are often in the same line. But it also creates a subtle problem when two fields in a single cache line are written by different CPUs. The CPUs'

caches have to coordinate; each time one CPU writes the line, the other CPU's copy is invalidated, and the line bounces between them. The cost is measurable and, at high concurrency, dominant. The phenomenon is called **false sharing**, because the two CPUs are not actually sharing data (they write different fields), but the cache coherence protocol treats them as if they were.

A driver's `softc` structure is a common place for false sharing to show up. If a `softc` has two counters, one updated by the read path and one updated by the write path, and both sit in the same cache line, every read and every write causes a cache bounce. At low concurrency this is invisible. At high concurrency it can halve your throughput.

The fix is explicit cache-line alignment. FreeBSD's `CACHE_LINE_SIZE` macro names the cache line size for the current architecture, and the `__aligned` compiler attribute places a variable at the right alignment. To isolate a field in its own cache line:

```
struct perfdemo_softc {
    struct mtx          mtx;
    /* ... other fields ... */

    uint64_t            read_ops __aligned(CACHE_LINE_SIZE);
    char                pad1[CACHE_LINE_SIZE - sizeof(uint64_t)];

    uint64_t            write_ops __aligned(CACHE_LINE_SIZE);
    char                pad2[CACHE_LINE_SIZE - sizeof(uint64_t)];
};
```

This is one way to do it, and it makes the isolation explicit to the reader. A cleaner way, when the struct is going to be allocated individually rather than as an array, is to put each hot counter in its own aligned sub-structure, or to use `counter(9)` which handles per-CPU placement internally and avoids the issue.

You do not have to pad every field in every struct. In most cases the `softc` is allocated once per device and accessed by one CPU at a time, and alignment does not matter. Reach for cache-line alignment when:

- A profile shows false-sharing-like behaviour (high LLC-misses on a function that does not look memory-heavy, or low IPC on a function that should be CPU-bound).
- Multiple CPUs are known to write different fields of the same struct concurrently.
- You measured a difference. As always.

33.8.2 Alignment for DMA

DMA engines usually require buffers aligned to a boundary larger than a cache line: 512 bytes for disk DMA, 4 KB for some network hardware, sometimes larger. If you allocate a buffer with `malloc(9)` and hand it to DMA without checking alignment, you are relying on the allocator's default alignment, which is usually large enough on amd64 but not guaranteed on all architectures.

For DMA, FreeBSD's `bus_dma(9)` API is the right interface. It handles alignment, bounce buffers, and scatter-gather for you. We covered it in Chapter 21. Here the relevant note is just that `bus_dma` handles alignment explicitly, and a driver should not allocate DMA-able memory with plain `malloc(9)` and hope.

A related memory concern is **alignment for SIMD or vectorised code**, where the compiler or the ISA may require 16-byte or 32-byte alignment on certain arguments. Again, `__aligned` is the tool. For buffers you allocate yourself, `contigmalloc(9)` or the UMA zone allocator can produce aligned memory.

33.8.3 Pre-Allocated Buffers

Every driver has a hot path. The cardinal rule for a hot path is that it should not allocate memory. Memory allocation is expensive on every kernel allocator: `malloc(9)` takes a lock and walks lists; `uma(9)` caches things per CPU but still has a fallback path to the slab allocator; `contigmalloc(9)` may block on page-level operations. None of these belong in a function that runs thousands of times per second.

The fix is **pre-allocation**. Allocate the memory your hot path needs in `attach()`, not in the hot path. If you need a fixed number of buffers for a ring, allocate them once. If you need a pool of buffers that get reused, allocate the pool at `attach` and keep a free list. If you need an occasional larger buffer, allocate it in a cold path.

For `perfdemo`, pre-allocation might look like this:

```
#define PERFDEMO_RING_SIZE      64
#define PERFDEMO_BUFFER_SIZE    4096

struct perfdemo_softc {
    struct mtx      mtx;
    char            *ring[PERFDEMO_RING_SIZE];
    int             ring_head;
    int             ring_tail;
    /* ... */
};

static int
perfdemo_attach(device_t dev)
{
    struct perfdemo_softc *sc = device_get_softc(dev);
    int i;

    for (i = 0; i < PERFDEMO_RING_SIZE; i++) {
        sc->ring[i] = malloc(PERFDEMO_BUFFER_SIZE, M_PERFDEMO, M_WAITOK);
    }
    /* ... */
    return (0);
}

static int
perfdemo_detach(device_t dev)
{
    struct perfdemo_softc *sc = device_get_softc(dev);
```



```

int i;

for (i = 0; i < PERFDEMO_RING_SIZE; i++) {
    if (sc->ring[i] != NULL) {
        free(sc->ring[i], M_PERFDEMO);
        sc->ring[i] = NULL;
    }
}
return (0);
}

```

On the hot path, the driver takes a buffer from the ring, uses it, returns it. No allocator, no lock beyond the ring's own, no risk of allocation failure under load.

This is the pattern that separates drivers that perform well under stress from drivers that fall off a cliff when memory pressure rises. A driver that allocates on the hot path will eventually fail to allocate, and its error path will be exercised in the most stressful moment. A driver that pre-allocates takes its failure at attach time, where it can be reported cleanly.

33.8.4 UMA Zones

`malloc(9)` is a good general-purpose allocator, but for frequently allocated and freed fixed-size objects it is not optimal. FreeBSD provides the UMA (Universal Memory Allocator) zone framework for this case. UMA gives you per-CPU caches, so a `uma_zalloc()` followed by a `uma_zfree()` within one CPU's thread is usually a handful of pointer swaps with no locks. The declarations are in `/usr/src/sys/vm/uma.h`.

A zone is created once per driver at module load, destroyed at unload:

```

#include <vm/uma.h>

static uma_zone_t perfdemo_buffer_zone;

static int
perfdemo_modevent(module_t mod, int what, void *arg)
{
    switch (what) {
        case MOD_LOAD:
            perfdemo_buffer_zone = uma_zcreate("perfdemo_buffer",
                PERFDEMO_BUFFER_SIZE, NULL, NULL, NULL, NULL,
                UMA_ALIGN_CACHE, 0);
            if (perfdemo_buffer_zone == NULL)
                return (ENOMEM);
            break;
        case MOD_UNLOAD:
            uma_zdestroy(perfdemo_buffer_zone);
    }
}

```

```
        break;
    }
    return (0);
}
```

Note the `UMA_ALIGN_CACHE` flag. It asks UMA to align each item in the zone to a cache-line boundary, which matters when items are used from multiple CPUs. The macro is defined in `/usr/src/sys/vm/uma.h`.

On the hot path, allocation and free look like this:

```
void *buf;

buf = uma_zalloc(perfdemo_buffer_zone, M_WAITOK);
/* use buf */
uma_zfree(perfdemo_buffer_zone, buf);
```

The `M_WAITOK` flag says “this call may sleep waiting for memory”. On a non-sleeping path, use `M_NOWAIT` and handle a `NULL` return. For a hot path that absolutely must not block, you can also do `M_NOWAIT | M_ZERO` and keep a fallback pool.

UMA is the right tool when the driver repeatedly allocates and frees objects of the same size. For one-off, variable-size allocations, `malloc(9)` remains fine. For rings of pre-allocated buffers, UMA is overkill; a plain array is simpler.

The rule of thumb: if you are allocating the same-size thing hundreds of times per second, it deserves a UMA zone.

33.8.5 A Tour of UMA Internals

UMA’s behaviour under the hood is worth understanding because it explains both its performance advantages and the cases where it does *not* help. The zone allocator has three layers: per-CPU caches, zone-wide lists, and the slab allocator below everything.

The per-CPU cache is a small array of free items, one per CPU. When a CPU calls `uma_zalloc`, UMA first checks its own cache; if there is an item, the call is a quick pointer swap with no lock. The default cache size is tuned by UMA based on the item size and the machine’s CPU count, and can be overridden with `uma_zone_set_maxcache()` when a particular zone needs a different ceiling.

When the per-CPU cache is empty, UMA refills it from the zone-wide free list. This path takes a lock, pulls several items at once, and returns. It is more expensive than the cache path but still much cheaper than a full allocator call.

When the zone-wide free list is empty, UMA calls into the slab allocator to carve more items from a fresh page. This is the most expensive path and involves allocating a physical page, partitioning it into slab-sized items, and populating the zone’s free list. Most of a zone’s hot-path calls never reach this layer; the per-CPU cache and the zone-wide list are hit first.

Three consequences follow.

First, **UMA is fastest when allocations and frees happen on the same CPU**. A pair of `uma_zalloc` / `uma_zfree` calls on one CPU is a handful of pointer swaps. A `uma_zalloc` on CPU 0 followed by a `uma_zfree` on CPU 1 returns the item to CPU 1's cache, which may or may not benefit CPU 0's future allocations. If your driver allocates on one CPU and frees on another routinely, the per-CPU advantage erodes.

Second, **zones can be tuned with `uma_zone_set_max()` and `uma_prealloc()`**. Pre-allocation reserves a fixed number of items at zone creation, so the first few hundred allocations never hit the slab path. Max-size caps the zone and fails subsequent allocations with `M_NOWAIT` instead of allowing unbounded growth. Both are useful for drivers that need predictable memory behaviour.

Third, **`vmstat -z` is your window into UMA**. It lists every zone, the per-item size, the current count, the cumulative count, and the failure count. A zone whose current count is growing without a matching free path is a leak. A zone whose failure count is non-zero is under memory pressure. Learn to read `vmstat -z` fluently; it is the single best tool for diagnosing UMA issues.

The internals are in `/usr/src/sys/vm/uma_core.c` for the main allocator and `/usr/src/sys/vm/uma.h` for the public interface. Reading the interface file is enough for most driver work; the core file is where to look when a failure mode does not match your expectation.

33.8.6 Bounce Buffers and DMA Memory Mapping

DMA-capable devices need physically contiguous, DMA-safe memory. On amd64 all memory is DMA-capable by default (assuming the device is not IOMMU-constrained), but on other architectures and on IOMMU-protected systems the story is more complex. FreeBSD's `bus_dma(9)` API hides the complexity, but a driver author needs to know when bounce buffers come into play.

A **bounce buffer** is an intermediate buffer the kernel uses when a user-provided buffer is not DMA-safe. If the device's DMA engine cannot reach a given physical address (for example, the device is 32-bit and the buffer is above 4 GB), the kernel allocates a bounce buffer in reachable memory, copies the data in or out, and points the DMA engine at the bounce buffer. This is transparent to the driver but not free: every bounced operation includes a memory copy.

The performance impact shows up in two places. First, the copy doubles the memory traffic for the operation, which matters when the operation is bandwidth-sensitive. Second, the bounce-buffer pool is bounded; under pressure, allocations fail, and the driver must wait. A `bus_dma(9)` driver on a 64-bit system with plenty of memory rarely sees bounce buffers; a 32-bit driver on a system with memory above 4 GB sees them constantly.

The tuning knobs, when bounce buffers become a measurable cost:

- Ask the device if it supports 64-bit DMA. Many 32-bit-era devices actually implement 64-bit addressing and the driver can enable it with the right `bus_dma_tag_create()` flags.
- Use `BUS_SPACE_UNRESTRICTED` in the tag's `lowaddr` field if the device really is unrestricted, to tell `bus_dma` not to bounce.
- For truly 32-bit devices that must work on a 64-bit system, consider pre-allocating reachable buffers in the driver and copying there on the dispatch path, which at least moves the bounce cost to a known place.

This topic deserves its own chapter; Chapter 21 covers `bus_dma(9)` in depth. Here the point is that bounce buffers are a performance feature you may not have chosen consciously, and when they appear in a profile you now know what they mean.

33.8.7 Per-CPU Counters Revisited

Section 2 introduced `counter(9)` and `DPCPU_DEFINE(9)`. This section is where those primitives matter most. On a driver with many CPUs all writing to the same counter, the counter's cache line becomes a contention point. `counter(9)` avoids it by keeping a per-CPU copy and summing on read.

A before-and-after example makes the point concrete. Suppose `perfdemo` has a single `atomic_add_64()` counter for total reads, updated on every read. On an 8-core system doing a parallel-reader benchmark, the counter cache line bounces between eight L1 caches. A `pmcstat` profile shows `atomic_add_64` surprisingly high in the sampled functions. Switching to `counter_u64_add()`:

- Each CPU updates its own copy. No contention.
- The cache line each CPU writes is in its own L1 exclusively.
- The fetch path sums all CPUs' copies, which happens only when someone reads the `sysctl`.

The result is that the counter's cost on the hot path drops from hundreds of nanoseconds per update (the cache miss plus the atomic) to a few nanoseconds (a direct write to a CPU-local cache line). On an 8-core system this can be a 10x reduction in counter cost.

A practical template for a driver with several per-CPU counters:

```
#include <sys/counter.h>

struct perfdemo_stats {
    counter_u64_t reads;
    counter_u64_t writes;
    counter_u64_t bytes;
    counter_u64_t errors;
};

static struct perfdemo_stats perfdemo_stats;

static void
perfdemo_stats_init(void)
{
    perfdemo_stats.reads = counter_u64_alloc(M_WAITOK);
    perfdemo_stats.writes = counter_u64_alloc(M_WAITOK);
    perfdemo_stats.bytes = counter_u64_alloc(M_WAITOK);
    perfdemo_stats.errors = counter_u64_alloc(M_WAITOK);
}

static void
perfdemo_stats_free(void)
{
    counter_u64_free(perfdemo_stats.reads);
    counter_u64_free(perfdemo_stats.writes);
    counter_u64_free(perfdemo_stats.bytes);
}
```

```
    counter_u64_free(perfdemo_stats.errors);
}
```

Each field is a `counter_u64_t`, which is internally a pointer to per-CPU storage. Each increment is a per-CPU add. Each read is a sum across all CPUs.

For state that is not a counter, DPCPU is the tool:

```
#include <sys/pcpu.h>

DPCPU_DEFINE_STATIC(struct perfdemo_cpu_state, perfdemo_cpu);

/* In a fast path, on the current CPU: */
struct perfdemo_cpu_state *s = DPCPU_PTR(perfdemo_cpu);
s->last_read_ns = now;
s->read_count++;
```

DPCPU is slightly less convenient than `counter(9)` because you have to define and initialise the per-CPU structure yourself, but it gives you the flexibility of per-CPU arbitrary state rather than just a counter.

33.8.8 Per-CPU Data Patterns Beyond Counters

`counter(9)` handles the common case of per-CPU integer accumulation. DPCPU handles the general case of per-CPU anything. In practice there are four patterns that come up repeatedly.

Pattern 1: Per-CPU cache of a scratch buffer. When a driver needs a small scratch buffer on a hot path, and the buffer is always used on one CPU at a time, a DPCPU pointer to a per-CPU pre-allocated buffer avoids any allocation at all:

```
DPCPU_DEFINE_STATIC(char *, perfdemo_scratch);

static int
perfdemo_init_scratch(void)
{
    int cpu;
    CPU_FOREACH(cpu) {
        char *p = malloc(PERFDEMO_SCRATCH_SIZE, M_PERFDEMO, M_WAITOK);
        if (p == NULL)
            return (ENOMEM);
        DPCPU_ID_SET(cpu, perfdemo_scratch, p);
    }
    return (0);
}
```

```

/* On the hot path: */
critical_enter();
char *s = DPCPU_GET(perfdemo_scratch);
/* ... use s ... */
critical_exit();

```

The `critical_enter()` / `critical_exit()` pair is necessary because the scheduler could preempt the thread between the `DPCPU_GET` and the use, and the thread could resume on a different CPU. Staying in a critical section prevents the migration. The cost is small (a few nanoseconds) but non-zero; use it whenever you access DPCPU data that must be stable across an operation.

Pattern 2: Per-CPU state for lockless statistics. Some drivers maintain rolling windows, last-timestamp values, or other state per CPU. DPCPU gives you the per-CPU storage without any synchronisation: each CPU reads and writes its own copy, and a global summary is computed on demand.

```

struct perfdemo_cpu_stats {
    uint64_t last_read_ns;
    uint64_t cpu_time_ns;
    uint32_t current_queue_depth;
};

DPCPU_DEFINE_STATIC(struct perfdemo_cpu_stats, perfdemo_cpu_stats);

/* Fast path: */
critical_enter();
struct perfdemo_cpu_stats *s = DPCPU_PTR(perfdemo_cpu_stats);
s->last_read_ns = sbtttons(sbinuptime());
s->cpu_time_ns += elapsed;
critical_exit();

```

The summary for reporting:

```

static uint64_t
perfdemo_total_cpu_time(void)
{
    uint64_t total = 0;
    int cpu;
    CPU_FOREACH(cpu) {
        struct perfdemo_cpu_stats *s =
            DPCPU_ID_PTR(cpu, perfdemo_cpu_stats);
        total += s->cpu_time_ns;
    }
    return (total);
}

```

Pattern 3: Per-CPU queues, one-way. A driver that defers work to a taskqueue can maintain a per-CPU pending list that the producing CPU appends to and the consumer periodically drains. Because each producer is the only writer of its per-CPU list, no locks are needed on the producer side. The consumer still needs locking or atomic swaps to drain safely, but the hot path is lock-free.

Pattern 4: Per-CPU configuration. Some drivers parameterise their behaviour per-CPU for scaling reasons: each CPU has its own buffer size, its own retry count, its own soft limits. DPCPU makes this natural. The trade-off is that configuration values are scattered across CPUs; reporting them requires a loop that visits every CPU.

These patterns share a common theme: per-CPU state eliminates inter-CPU synchronisation on the hot path, in exchange for a small cost on the aggregation path. When the hot path is fast and the aggregation is infrequent, the trade-off is obviously favourable.

33.8.9 When Not to Align, Allocate, or Cache

Every technique above has a cost. Alignment pads structures and wastes memory. Pre-allocation ties up buffers that may go unused. UMA zones consume kernel memory. Per-CPU counters use $N * \text{CPU\_COUNT}$ bytes of memory where the driver might have got away with $N * 1$. None of these costs is fatal, but in the aggregate they add up, especially on small systems.

The rule of thumb is the one you will hear throughout the chapter: do not apply these techniques until measurement demands. A driver that runs at 1000 operations per second on an idle desktop does not benefit from cache alignment, UMA zones, or per-CPU counters. A driver on a 64-core storage server processing 10 million operations per second may need all three.

A useful self-check before adding any of these techniques: *can I point to a specific pmcstat result, DTrace aggregation, or benchmark number that shows the current code is slow in the way the technique would fix?* If yes, apply it. If no, do not.

33.8.10 Exercise 33.5: Cache-Line and Per-CPU Tuning

Using the `perfdemo` driver, do the following:

1. Baseline: measure the throughput of `perfdemo_read()` with the current atomic-counter implementation, using a multi-threaded reader that runs one thread per CPU. Record the throughput in ops/sec.
2. Add `__aligned(CACHE_LINE_SIZE)` to the atomic counters in the `softc` and pad to isolate them. Rebuild, reload, rerun. Record the new throughput.
3. Replace the atomic counters with `counter(9)` counters. Rebuild, reload, rerun. Record the new throughput.
4. Compare the three numbers.

The point is not that one approach is always best. The point is that the three approaches produce different numbers on different hardware, and the evidence tells you which is right for your target.

33.8.11 Wrapping Up Section 5

Memory tuning is about understanding the memory hierarchy the CPU is working against. Cache-line alignment avoids false sharing. DMA alignment works through `bus_dma(9)`. Pre-allocation keeps the hot path out of the allocator. UMA zones specialise fixed-size frequent allocations. Per-CPU counters and per-CPU state eliminate cache-line contention on shared metrics. Each technique is a tool in the toolkit, and each deserves evidence before being applied.

The next section looks at the other major source of driver performance bottlenecks: interrupt handling and deferred work via taskqueues.

33.9 Section 6: Interrupt and Taskqueue Optimization

Interrupts are where the driver meets the hardware, and for many drivers they are where the first performance cliff sits. An interrupt handler runs in a constrained context. It cannot sleep. It cannot acquire most sleepable locks. It competes with other threads for CPU time, but it also preempts them. A handler that takes too long holds up the whole system. A handler that fires too often consumes CPU the rest of the system needs. The techniques in this section are about keeping the interrupt handler fast and moving the rest of the work to a place where it can be done calmly.

The section covers four topics: measuring interrupt behaviour, filter versus ithread handlers (a brief recap), moving work to taskqueues, and interrupt coalescing.

33.9.1 Measuring Interrupt Behaviour

Before you change anything, you need to know how often interrupts fire and how long the handler runs. FreeBSD gives you both metrics through standard tools.

Interrupt rate is reported by `vmstat -i`:

```
# vmstat -i
interrupt                total      rate
irq1: atkbd0              10         0
irq9: acpi0              1000000     500
irq11: em0 ehci0+         44000       22
irq16: ohci0 uhci1+       1000        0
cpu0:timer                2000000     1000
cpu1:timer                2000000     1000
cpu2:timer                2000000     1000
cpu3:timer                2000000     1000
Total                    8148010     4547
```

The `total` column is cumulative since boot; the `rate` column is per second over the sampling window. If your driver's interrupt is firing at thousands of hertz and you did not expect it, something is wrong. If it is firing at single digits per second and you did expect thousands, something else is wrong.

Interrupt handler latency is reported by DTrace. The fbt provider has probes on every non-static function, including the driver's interrupt handler. A one-liner that measures how long the handler runs:

```
# dtrace -n 'fbt:perfdemo:perfdemo_intr:entry { self->t = timestamp; }
           fbt:perfdemo:perfdemo_intr:return /self->t/ {
               @ = quantize(timestamp - self->t);
               self->t = 0; }'
```

The resulting histogram tells you the latency distribution of your interrupt handler. A handler with a median of one or two microseconds and a P99 of under ten is well behaved. A handler with a median of 100 microseconds and a P99 in the milliseconds is doing too much in the wrong context.

Interrupt storm detection is built into FreeBSD. When an interrupt fires much too often (the default threshold is 100,000 per second), the kernel disables the interrupt source temporarily and logs a warning:

```
interrupt storm detected on "irq18:"; throttling interrupt source
```

If you see this message in `dmesg`, your driver is either processing the interrupt incorrectly (not acknowledging it, letting it re-fire immediately) or the hardware is generating at a rate the driver cannot keep up with. Either is a bug; it is not a normal state.

33.9.2 Filter Handlers vs Ithread Handlers

Chapter 13 covered the two shapes of interrupt handler in detail. This section relies on them; here is a short recap for context.

A **filter handler** runs in the interrupt context of the hardware interrupt itself. It has very strict constraints: it cannot sleep, it cannot acquire most locks, and it should do the minimum work needed to quiet the hardware and tell the kernel what to do next. It returns one of `FILTER_HANDLED`, `FILTER_STRAY`, or `FILTER_SCHEDULE_THREAD`. The last tells the kernel to run the associated ithread handler.

An **ithread handler** runs in a dedicated kernel thread, scheduled shortly after the filter returns `FILTER_SCHEDULE_THREAD`. It can acquire sleepable locks, do more complex work, and take as long as is reasonable without stalling the system. It is still preemptible but it is not the interrupt context.

The two-level design lets you split interrupt handling cleanly. The filter does the minimum: read the status register, figure out whether the interrupt is ours, acknowledge it so the hardware stops asserting it, and schedule the ithread if work remains. The ithread does the rest: process the data, wake up userland, deallocate buffers, update counters.

For performance, the question is: do you need both levels? The answer is almost always yes for drivers handling real hardware, and often no for simple devices. A filter-only design is fast: the handler runs once per interrupt, does its work, and returns. A filter-plus-ithread design adds the thread-scheduling cost (a few microseconds) but lets the main work run with fewer context constraints.

If your filter handler is fast and complete, keep the design filter-only. If it is large and you need sleepable locks, split into filter and ithread. If the ithread is itself fast and the split is just adding overhead, consolidate. The measurements will tell you which case you are in.

33.9.3 Moving Work to Taskqueues

A common pattern for long-running driver work is to move it out of interrupt context entirely, into a taskqueue. A taskqueue is a simple, named queue of functions to run, scheduled by a dedicated kernel thread. You enqueue a task from the interrupt handler; the thread dequeues and runs it; the interrupt returns quickly.

The basic pattern:

```
#include <sys/taskqueue.h>

struct perfdemo_softc {
    struct task perfdemo_task;
    /* ... */
};

static void
perfdemo_task_handler(void *arg, int pending)
{
    struct perfdemo_softc *sc = arg;
    /* long-running work here; can acquire sleepable locks */
}

static int
perfdemo_attach(device_t dev)
{
    struct perfdemo_softc *sc = device_get_softc(dev);

    TASK_INIT(&sc->perfdemo_task, 0, perfdemo_task_handler, sc);
    /* ... */
    return (0);
}

static void
perfdemo_intr(void *arg)
{
    struct perfdemo_softc *sc = arg;

    /* Quick filter work */

    taskqueue_enqueue(taskqueue_fast, &sc->perfdemo_task);
}
```

The filter handler does its minimal work and then schedules a task on `taskqueue_fast`, the kernel's default fast-taskqueue. The task runs on the taskqueue's thread, which is preemptible and can sleep. The interrupt handler returns as quickly as it can.

The `taskqueue_fast` queue is shared by many subsystems. On a heavily loaded system, this can mean your task waits behind others. If you need a taskqueue of your own, create one:

```
struct taskqueue *my_tq;

/* In module init: */
my_tq = taskqueue_create("perfdemo_tq", M_WAITOK,
    taskqueue_thread_enqueue, &my_tq);
taskqueue_start_threads(&my_tq, 1, PI_NET, "perfdemo tq thread");
```

`taskqueue_start_threads` creates one thread bound to the queue. For a driver with CPU-bound work and a many-core system, you can use `taskqueue_start_threads_cpuset()` to pin threads to specific CPUs, which helps cache locality:

```
#include <sys/cpuset.h>

cpuset_t cpus;

CPU_ZERO(&cpus);
CPU_SET(1, &cpus); /* bind to CPU 1 */
taskqueue_start_threads_cpuset(&my_tq, 1, PI_NET, &cpus,
    "perfdemo tq thread");
```

The advantage of a dedicated taskqueue is predictable scheduling. Your task is not competing with the whole system for thread time on `taskqueue_fast`. The disadvantage is more threads and more memory, so do this only when the shared taskqueue is genuinely causing a problem.

33.9.4 Interrupt Coalescing

Hardware interrupts have a fixed cost: saving CPU state, transitioning to kernel context, dispatching the handler, and returning. On modern hardware this is under a microsecond, but it is not zero. When a device fires very many interrupts per second, the cumulative cost becomes noticeable. Network NICs on a 10 Gbps link can generate hundreds of thousands of interrupts per second at peak; storage controllers on NVMe can generate more.

Interrupt coalescing is the technique of asking the hardware to batch multiple events into a single interrupt. Instead of interrupting once per packet, the NIC interrupts once per millisecond, and the driver handles all packets received in that millisecond. The throughput goes up because the per-event overhead drops; the latency goes up too, because events wait longer in the batch.

Not every driver supports coalescing, and it is a hardware feature. When the hardware provides it (most modern NICs and NVMe controllers do), the driver exposes knobs through `sysctl` or `ioctl`. The trade-off must be tuned to the workload: a latency-sensitive workload wants short coalescing windows; a throughput-sensitive workload wants long ones. Sometimes a driver exposes separate knobs for the two directions (RX and TX), allowing fine-grained tuning.

As a driver author, you have three options relating to coalescing:

1. **Use hardware coalescing.** If the hardware has it, expose a sysctl knob. The operator tunes the driver for their workload.
2. **Do software-side batching.** If the hardware does not, and the driver can usefully process events in groups, the driver can keep work pending until a threshold is reached and then dispatch the whole batch. `iflib(9)` does this for many network drivers.
3. **Do nothing.** If the event rate is low enough that coalescing would not help, or the latency budget is tight enough that coalescing would hurt, leave it alone.

The right choice is, again, the one the measurements point to. If `vmstat -i` shows your driver interrupting at hundreds of thousands of hertz and CPU time in the interrupt path is a noticeable fraction of total CPU, coalescing is worth trying. If the interrupt is at single digits per second, coalescing is irrelevant.

33.9.5 Debouncing

A related concept is **debouncing**: filtering out redundant or too-close-together interrupts. GPIO input buttons are the classic example; a mechanical switch produces many rapid interrupts when pressed or released, and the driver must filter out the ones that are not real state changes. The technique is:

1. On interrupt, read the input and timestamp.
2. If the input matches the last-reported state and the timestamp is within the debounce window, ignore.
3. Otherwise, report the change and update the last-reported state.

A `sbinuptime()` timestamp and a 20-millisecond debounce window are sufficient for most mechanical switches. For other event sources, the threshold is domain-specific.

Debouncing is not usually called performance tuning, but it has the same effect: fewer interrupts handled, less CPU consumed, fewer spurious events reported. It is a small but real technique.

33.9.6 Dedicated Threads for Long Work

The kernel allows a driver to have its own long-lived thread, distinct from both taskqueue threads and interrupt threads. Use `kproc_create(9)` or `kthread_add(9)` to create one. A dedicated thread makes sense when:

- The driver has a long-running loop (for example, a polling loop that replaces interrupts for deterministic timing).
- The work must not contend with other subsystems on a shared taskqueue.
- The work has its own cadence that does not match the interrupt cadence (for example, a periodic flush, a watchdog, or a state-machine driver).

For most drivers, a taskqueue is simpler and sufficient. A dedicated thread is a step up in complexity and should be justified by the work's nature.

33.9.7 MSI and MSI-X: Per-Vector Interrupts

A classic device uses a single interrupt line: every event fires the same interrupt, the handler must distinguish them by reading a status register, and the handler runs on whichever CPU the kernel's interrupt balancer chose.

A modern device supports Message-Signaled Interrupts (MSI) or the extended MSI-X variant, which lets the device send many distinct interrupts, each with its own vector and each configurable to a specific CPU.

For a driver author, MSI-X brings three performance advantages over legacy line-based interrupts.

First, **per-vector interrupts eliminate shared handlers**. With legacy interrupts, a driver's filter handler might fire for other drivers' events and must return `FILTER_STRAY` when the event is not its own. With MSI-X, each vector belongs to one driver; the filter handler only runs when its device has work.

Second, **per-vector CPU pinning reduces cache-line bouncing**. A multi-queue network driver can dedicate one RX vector to CPU 0, one to CPU 1, and so on. Each CPU processes its own queue, touching its own data structures, without sharing cache lines with the other CPUs. On a high-throughput driver, this scales linearly with core count.

Third, **interrupt routing reduces latency**. If a driver knows which CPU will consume a packet's data, it can request an interrupt on that CPU directly, so the data arrives with the cache already warm.

In FreeBSD, MSI-X is requested via `pci_alloc_msix(9)` during `attach()`. The driver asks for `N` vectors; the PCI subsystem allocates up to `N` (or fewer if the hardware supports fewer). Each vector is a standard `bus_alloc_resource()` IRQ that you set up with `bus_setup_intr(9)` as usual. The `iflib(9)` framework wraps the machinery for network drivers; for custom drivers, the raw API is only a few lines longer.

A small excerpt from a driver that allocates four MSI-X vectors:

```
static int
mydrv_attach(device_t dev)
{
    struct mydrv_softc *sc = device_get_softc(dev);
    int n = 4;

    if (pci_alloc_msix(dev, &n) != 0 || n < 4) {
        device_printf(dev, "cannot allocate 4 MSI-X vectors\n");
        return (ENXIO);
    }

    for (int i = 0; i < 4; i++) {
        sc->irq[i] = bus_alloc_resource_any(dev, SYS_RES_IRQ,
            &(int){i + 1}, RF_ACTIVE);
        if (sc->irq[i] == NULL)
            return (ENXIO);
        if (bus_setup_intr(dev, sc->irq[i], INTR_TYPE_NET | INTR_MPSAFE,
            NULL, mydrv_intr, &sc->queue[i], &sc->ih[i]) != 0)
            return (ENXIO);
    }

    /* Optionally, pin each IRQ to a CPU: */
    for (int i = 0; i < 4; i++) {
        int cpu = i;
        bus_bind_intr(dev, sc->irq[i], cpu);
    }
}
```

```
    }  
  
    return (0);  
}
```

The per-vector handler `mydrv_intr` runs only when *its* queue has work; it gets the per-queue `softc` pointer as its argument. The `bus_bind_intr()` call pins each vector to a specific CPU; on a system with more CPUs than vectors, you might pin every vector to a different CPU.

MSI-X is not always a win. On a single-queue device there is no benefit, just extra setup. On a low-interrupt-rate device the difference is negligible. But on any modern high-throughput device (10 Gbps NIC, NVMe drive), MSI-X is the standard interrupt model and using anything else leaves performance on the table.

33.9.8 Polling as an Alternative to Interrupts

For the extreme upper end of throughput, some drivers abandon interrupts entirely in favour of *polling*. A polling driver runs a tight loop that checks the hardware for work, processes it immediately, and goes around again. The trade-off is CPU cost: a polling thread consumes a CPU continuously, even when idle. The benefit is interrupt cost elimination: at high event rates, the interrupt and context-switch overhead per event disappears.

FreeBSD's network stack supports polling via `ifconfig polling`. Storage drivers typically do not poll. For custom drivers, polling is worth considering only when:

- The event rate is extremely high (millions per second).
- Latency is critical enough that interrupt latency (under a microsecond) is too much.
- CPU is cheap (a dedicated core for the poll loop is acceptable).

Most drivers fit none of these criteria, and polling is the wrong choice. But knowing it exists lets you recognise when a profile shows interrupt overhead dominating, and gives you a last-resort alternative.

A middle ground is *adaptive polling*: switch between interrupt-driven and polling mode based on load. NAPI-style batching (named after the Linux subsystem that pioneered it, but widely used in FreeBSD's `iflib`) takes the first interrupt, disables further interrupts, polls until the queue is drained, then re-enables. This captures most of polling's efficiency at high rates while keeping idle-time cost low.

33.9.9 NAPI-Style Batching in Practice

The `iflib(9)` framework implements NAPI-style batching automatically. A driver that uses `iflib` receives packets through a callback that polls the hardware queue until empty. For non-`iflib` drivers, the pattern is straightforward to implement by hand:

```
static int  
mydrv_filter(void *arg)  
{  
    struct mydrv_softc *sc = arg;
```

```

    /* Disable hardware interrupts for this queue. */
    mydrv_hw_disable_intr(sc);

    /* Schedule the ithread or taskqueue to drain. */
    return (FILTER_SCHEDULE_THREAD);
}

static void
mydrv_ithread(void *arg)
{
    struct mydrv_softc *sc = arg;
    int drained = 0;

    while (mydrv_hw_has_work(sc) && drained < MYDRV_POLL_BUDGET) {
        mydrv_process_one(sc);
        drained++;
    }

    if (!mydrv_hw_has_work(sc)) {
        /* Queue is empty; re-enable interrupts. */
        mydrv_hw_enable_intr(sc);
    } else {
        /* Budget hit with work remaining; reschedule. */
        taskqueue_enqueue(sc->tq, &sc->task);
    }
}

```

The filter disables the interrupt and schedules the ithread. The ithread polls up to `POLL_BUDGET` events, then checks whether the queue is empty. If yes, it re-enables the interrupt. If no, it reschedules itself to continue draining on the next pass. The budget prevents a single burst from monopolising the CPU; the empty-queue check prevents perpetually-polling when traffic stops.

NAPI-style batching is a good fit for medium-to-high-rate drivers where neither pure interrupts nor pure polling is ideal. The budget and the re-enable logic are the two places where mistakes happen; the budget must be large enough to amortise the interrupt re-enable cost but small enough not to stall other drivers, and the re-enable must happen before the hardware can forget about pending work.

33.9.10 Budget Allocation Across Stages

A useful exercise for any performance-critical driver is to write down a *latency budget* across the driver's stages. The total budget is the operation's deadline (for example, 100 microseconds for a low-latency network packet). Subtract each stage's expected cost, and what remains is the headroom.

For a network driver receiving a packet:

- Interrupt dispatch: 1 us.

- Filter handler: 2 us.
- Ithread scheduling + start: 3 us.
- Packet processing (protocol layer): 20 us.
- Wakeup of userland reader: 5 us.
- Scheduler dispatch: 5 us.
- Userland copy and acknowledgment: 10 us.

Total: 46 us. On a 100 us budget, there are 54 us of headroom. If the driver starts missing its deadline, the budget tells you which stage is most likely to have slipped, and where to measure first.

The numbers above are illustrative; the actual numbers are hardware-specific and workload-specific. The habit of writing the budget before measuring is what makes measurement efficient. You start with a hypothesis about where time is going, confirm or refute it with the tools in Sections 2 through 4, and refine the budget as the driver evolves.

33.9.11 Exercise 33.6: Interrupt vs Taskqueue Latency

With the `perfdemo` driver, add a second variant that simulates interrupt-like processing using a high-resolution callout. Compare two configurations:

1. All processing done directly in the callout's handler (interrupt-like, in a privileged context).
2. Callout enqueues a task on a taskqueue; the taskqueue thread does the processing.

Measure the end-to-end latency (the time from the simulated interrupt to when the processed data is visible to userland) in both cases. A DTrace script can record the timestamps; a userland reader with `nanosleep()` on an `select()` can see the delivery.

Configuration 1 will usually have lower latency and higher CPU concentration in the callout context. Configuration 2 will have slightly higher latency but smoother CPU use across threads. Which is better depends on the workload; the exercise is to see the trade-off with numbers, not to declare one universally correct.

33.9.12 Wrapping Up Section 6

Interrupts are where the driver meets the hardware, and they are where performance cliffs first appear. Measuring interrupt rate and handler latency is easy with `vmstat -i` and DTrace. The filter-plus-ithread split keeps the interrupt context small; taskqueues move work to a calmer environment; dedicated threads offer isolation for special cases; coalescing and debouncing control the interrupt rate itself. Each technique has a cost, and each deserves evidence before it is applied.

The next section turns to the metrics and logging you leave in the driver after tuning ends: the `sysctl` tree that exposes your runtime state, the `log(9)` calls that report the conditions worth reporting, and the patterns that distinguish a shipped driver from a benchmark-only one.

33.10 Section 7: Using `sysctl` and Logging for Runtime Metrics

The measurement techniques in Sections 2 through 6 are instruments you reach for when a specific question arises. The metrics in Section 7 are the ones that stay in the driver forever. They are the dashboard lights on

a car: always visible, always current, always there to tell a future operator what the driver is doing. A driver without a good sysctl tree is a driver that an operator cannot diagnose after the original author has moved on. A driver with a good one is a driver that telegraphs its state whenever anyone asks.

This section covers sysctl tree design, the common metric shapes, rate-limited logging via `log(9)`, and the art of putting the right amount of observability into a shipped driver.

33.10.1 Designing a sysctl Tree

The sysctl tree is hierarchical. Each node has a name, a type, a flag set, and usually a value. The top-level nodes (`kern`, `net`, `vm`, `hw`, `dev`, and a few others) are fixed by the kernel; a driver creates a subtree under one of them. For a hardware driver, `dev.<driver_name>.<unit>.<metric>` is the conventional placement. For a pseudo-device or non-hardware driver, `hw.<driver_name>.<metric>` is common.

The declarations use the macros in `/usr/src/sys/sys/sysctl.h`. A minimal driver subtree:

```
SYSCALL_NODE(_dev, OID_AUTO, perfdemo, CTLFLAG_RD, 0,
    "perfdemo driver");

SYSCALL_NODE(_dev_perfdemo, OID_AUTO, stats, CTLFLAG_RD, 0,
    "Statistics");

SYSCALL_U64(_dev_perfdemo_stats, OID_AUTO, reads, CTLFLAG_RD,
    &perfdemo_reads_cached, 0, "Total read() calls");
```

The first `SYSCALL_NODE` creates `dev.perfdemo`. The second creates `dev.perfdemo.stats`. The `SYSCALL_U64` creates `dev.perfdemo.stats.reads` and points it at a `uint64_t` variable. From userland:

```
# sysctl dev.perfdemo.stats.reads
dev.perfdemo.stats.reads: 12345
```

For a device with multiple instances, the more idiomatic path is to use `device_get_sysctl_ctx(9)` and `device_get_sysctl_tree(9)` inside the `attach()` method. FreeBSD will have already created `dev.perfdemo.0` and `dev.perfdemo.1` for two instances, and these helpers give you the handle to add children under each:

```
static int
perfdemo_attach(device_t dev)
{
    struct perfdemo_softc *sc = device_get_softc(dev);
    struct sysctl_ctx_list *ctx = device_get_sysctl_ctx(dev);
    struct sysctl_oid *tree = device_get_sysctl_tree(dev);

    SYSCALL_ADD_U64(ctx, SYSCALL_CHILDREN(tree), OID_AUTO, "reads",
        CTLFLAG_RD, &sc->stats.reads_cached, 0,
```

```

        "Total read() calls");
    SYSCTL_ADD_U64(ctx, SYSCTL_CHILDREN(tree), OID_AUTO, "errors",
        CTLFLAG_RD, &sc->stats.errors_cached, 0,
        "Total errors");
    /* ... */
    return (0);
}

```

The `ctx` and `tree` are released automatically when the device detaches, so you do not have to do cleanup by hand. This is the standard pattern for per-device metrics.

33.10.2 Exposing counter (9) Values via sysctl

A `counter_u64_t` is a per-CPU counter, and exposing it through `sysctl` requires a little more work than a plain `uint64_t`. The pattern is a *procedural sysctl*: a `sysctl` that runs a function when read. The function fetches the counter sum and writes it to the `sysctl` buffer.

```

static int
perfdemo_sysctl_counter(SYSCTL_HANDLER_ARGS)
{
    counter_u64_t *cntp = arg1;
    uint64_t val = counter_u64_fetch(*cntp);

    return (sysctl_handle_64(oidp, &val, 0, req));
}

SYSCTL_PROC(_dev_perfdemo_stats, OID_AUTO, reads,
    CTLTYPE_U64 | CTLFLAG_RD | CTLFLAG_MPSAFE,
    &perfdemo_stats.reads, 0, perfdemo_sysctl_counter, "QU",
    "Total read() calls");

```

The `CTLFLAG_MPSAFE` flag says the handler does not need to be serialised by the `sysctl` subsystem's lock. "QU" is the type-string: Q for quad (64-bit) and U for unsigned. The handler is called on every read, so the sum is current.

Several modern drivers encapsulate this in a helper macro; see `/usr/src/sys/net/if.c` for an example of `SYSCTL_ADD_COUNTER_U64`. If you add many counter `sysctls`, it is worth defining a similar helper for your driver.

33.10.3 What to Expose

A driver's `sysctl` tree is an interface. Like any interface, it should be thoughtful: expose enough to be useful, but not so much that the tree becomes cluttered and its values lose meaning. A good starting set for a data-plane driver:

- **Total operations:** reads, writes, ioctls, interrupts. Cumulative counters.
- **Total errors:** transient errors (retries), permanent errors (failures), partial-transfer errors.
- **Bytes moved:** total bytes transferred in each direction.
- **Current state:** queue depth, busy flag, mode.
- **Configuration:** interrupt coalescing threshold, buffer size, debug level.

For each, think about whether an operator would want the value to diagnose a problem. If yes, expose it. If no, keep it internal.

Avoid exposing implementation details the driver might change. The sysctl tree is an API; renaming a node breaks scripts and monitoring dashboards. Choose the names carefully and keep them stable.

33.10.4 Read-Only, Read-Write, and Tunable

The CTLFLAG_RD flag makes a sysctl read-only from userland. The CTLFLAG_RW flag makes it writable, so an operator can change the driver's behaviour at runtime. CTLFLAG_TUN marks a sysctl as a **tunable**, which means its initial value can be set in `/boot/loader.conf` before the module loads.

Writable sysctls are flexible and dangerous. An operator can set a debug level, change a buffer size, toggle a feature flag. The driver must validate the written value carefully; an out-of-range buffer size can corrupt state. For most metrics, read-only is the right choice. For configuration, read-write with validation in a procedural handler is the pattern.

Here is a validated tunable that accepts values between 16 and 4096:

```
static int perfdemo_buffer_size = 1024;

static int
perfdemo_sysctl_bufsize(SYSCTL_HANDLER_ARGS)
{
    int new_val = perfdemo_buffer_size;
    int error;

    error = sysctl_handle_int(oidp, &new_val, 0, req);
    if (error != 0 || req->newptr == NULL)
        return (error);
    if (new_val < 16 || new_val > 4096)
        return (EINVAL);
    perfdemo_buffer_size = new_val;
    return (0);
}

SYSCTL_PROC(_dev_perfdemo, OID_AUTO, buffer_size,
    CTLTYPE_INT | CTLFLAG_RWTUN | CTLFLAG_MPSAFE,
    NULL, 0, perfdemo_sysctl_bufsize, "I",
    "Buffer size in bytes (16..4096)");
```

The handler checks `req->newptr != NULL` to distinguish a write from a read (a read has a non-null `newptr` only if a value was supplied). It validates the range and updates the variable. The `CTLFLAG_RWTUN` flag combines `CTLFLAG_RW` and `CTLFLAG_TUN`: runtime-writable and loader-tunable.

A rule worth absorbing: *any sysctl that changes driver state must validate its input*. The alternative is a user-configurable crash.

33.10.5 Rate-Limited Logging with `log(9)`

The kernel's `printf(9)` is fast but undisciplined: every call produces a line in `dmesg` regardless of rate. For informational messages that might fire often, the kernel provides `log(9)`, which tags the message with a priority level (`LOG_DEBUG`, `LOG_INFO`, `LOG_NOTICE`, `LOG_WARNING`, `LOG_ERR`, `LOG_CRIT`, `LOG_ALERT`, `LOG_EMERG`) and lets userland's `syslogd(8)` filter it. A message at `LOG_DEBUG` is only logged if `syslogd` is configured to accept debug messages; the default configuration drops them.

```
#include <sys/syslog.h>
```

```
log(LOG_DEBUG, "perfdemo: read of %zu bytes returning %d\n",  
    size, error);
```

The line still goes through the kernel's log buffer and costs the same as a `printf(9)` to produce, but it does not appear in `dmesg` unless `syslogd` is told to include debug messages. It is the right tool for diagnostic messages that an operator might sometimes want but that should not pollute the default log.

For messages that really should be rate-limited, FreeBSD has `ppsratecheck(9)` and `ratecheck(9)`. They return non-zero at most `N` times per second; use them to gate a print:

```
static struct timeval perfdemo_last_err;  
  
if (error != 0 && ratecheck(&perfdemo_last_err,  
    &(struct timeval){1, 0})) {  
    device_printf(dev, "transient error %d\n", error);  
}
```

This rate-limits the print to once per second. The `struct timeval` in the second argument is the interval; `{1, 0}` means one second. If the error rate is a thousand per second, you get one log line per second instead of a thousand.

Rate-limited logging is the right pattern for any message that could plausibly fire on a hot path. A driver that logs every transient error at full rate can DoS itself through `dmesg` alone.

33.10.6 `device_printf(9)` and Its Friends

For messages that identify the specific device, use `device_printf(9)`:

```
device_printf(sc->dev, "attached at %p\n", sc);
```

It prepends the device name and unit number automatically: `perfdemo0: attached at 0xfffffe00c...`. This is the convention for any message that would otherwise need to include `sc->dev` in its format string. Every FreeBSD driver uses `device_printf(9)` for its attach and detach messages; emulate that pattern.

For messages that do not belong to a specific device, plain `printf(9)` is fine but should still be tagged with the module name: `printf("perfdemo: %s: ...\n", __func__, ...)`. The `__func__` identifier is a C99 built-in that expands to the current function's name; it makes logs much easier to trace to their source.

33.10.7 Debug Modes

A common pattern in mature drivers is a **debug mode**: a writable sysctl that controls the verbosity of logging. At debug level 0 the driver logs only attach, detach, and real errors. At level 1 it logs transient errors. At level 2 it logs every operation. The pattern is cheap (a compare-to-zero on the hot path) and gives operators a safe way to turn on detailed logging when diagnosing a problem.

```
static int perfdemo_debug = 0;
SYSCTL_INT(_dev_perfdemo, OID_AUTO, debug, CTLFLAG_RWTUN,
    &perfdemo_debug, 0,
    "Debug level (0=errors, 1=transient, 2=verbose)");

#define PD_DEBUG(level, sc, ...) do {                                \
    if (perfdemo_debug >= (level))                                  \
        device_printf((sc)->dev, __VA_ARGS__);                    \
} while (0)
```

Used as:

```
PD_DEBUG(2, sc, "read %zu bytes\n", bytes);
PD_DEBUG(1, sc, "transient error %d\n", error);
```

At the default `perfdemo_debug = 0`, both lines are a single compare-to-zero that predicts well in the branch predictor. Neither message is produced. At `perfdemo_debug = 1`, only the transient-error line is produced. At `perfdemo_debug = 2`, both are.

The operator turns this on with:

```
# sysctl dev.perfdemo.debug=2
```

and off again when done. This is a convention found in many FreeBSD drivers; reuse it.

33.10.8 Tracking Behaviour Over Time

A counter that is polled every second gives you a rate. A sysctl that returns a histogram of recent latencies gives you a distribution. For longer-term tracking, you can keep a rolling window of values inside the driver, exposed through a sysctl that returns the array:

```
#define PD_LAT_SAMPLES 60

static uint64_t perfdemo_recent_lat[PD_LAT_SAMPLES];
static int perfdemo_lat_idx;

static void
perfdemo_lat_record(uint64_t ns)
{
    perfdemo_recent_lat[perfdemo_lat_idx] = ns;
    perfdemo_lat_idx = (perfdemo_lat_idx + 1) % PD_LAT_SAMPLES;
}

static int
perfdemo_sysctl_recent_lat(SYSCTL_HANDLER_ARGS)
{
    return (SYSCTL_OUT(req, perfdemo_recent_lat,
        sizeof(perfdemo_recent_lat)));
}

SYSCTL_PROC(_dev_perfdemo, OID_AUTO, recent_lat,
    CTLTYPE_OPAQUE | CTLFLAG_RD | CTLFLAG_MPSAFE,
    NULL, 0, perfdemo_sysctl_recent_lat, "S",
    "Last 60 samples of read latency (ns)");
```

The pattern works for any rolling window: latencies, queue depths, interrupt rates, whatever the operator might want to see in time order.

More sophisticated forms of the same idea store percentile estimates (P50, P95, P99) using streaming algorithms like exponential-weighted moving averages or the t-digest. For most drivers this is overkill; a simple rolling window, or even just a cumulative histogram stored as a bucket array, is enough.

33.10.9 Exposing Histograms via sysctl

A cumulative histogram is one of the most useful long-lived metrics a driver can expose. Userland can poll it, subtract the previous snapshot from the current one, and compute the rate per bucket. Plotting the result over time gives an immediate view of the driver's latency distribution.

The pattern: declare an array of `counter_u64_t`, one per bucket. Update the correct bucket in the hot path (linear scan or constant-time bucketing). Expose the array through a single procedural sysctl.

```
static int
perfdemo_sysctl_hist(SYSCTL_HANDLER_ARGS)
{
    uint64_t values[PD_HIST_BUCKETS];
    int i, error;

    for (i = 0; i < PD_HIST_BUCKETS; i++)
        values[i] = counter_u64_fetch(perfdemo_stats.hist[i]);

    error = SYSCTL_OUT(req, values, sizeof(values));
    return (error);
}

SYSCTL_PROC(_dev_perfdemo_stats, OID_AUTO, lat_hist,
    CTLTYPE_OPAQUE | CTLFLAG_RD | CTLFLAG_MPSAFE,
    NULL, 0, perfdemo_sysctl_hist, "S",
    "Read latency histogram (buckets: <1us, <10us, <100us, "
    "<1ms, <10ms, <100ms, <1s, >=1s)");
```

From userland, a small script reads the array and prints it:

```
#!/bin/sh
sysctl -x dev.perfdemo.stats.lat_hist | awk -F= '{
    split($2, bytes, " ");
    for (i = 1; i <= length(bytes); i += 8) {
        val = 0;
        for (j = 7; j >= 0; j--) {
            val = val * 256 + strtonum("0x" bytes[i + j]);
        }
        printf "Bucket %d: %d\n", (i - 1) / 8, val;
    }
}'
```

The script is a few lines and gives you an immediately useful dashboard. Over time, extensions become obvious: compare two snapshots to get rates, plot the deltas over time, alert when a high-latency bucket crosses a threshold.

33.10.10 Per-CPU Sysctl Presentation

For a driver with DPCPU state, there is a choice between aggregating across CPUs (the standard approach: compute a single number from DPCPU_SUM or a manual loop) and presenting per-CPU values separately. Per-CPU presentation is useful when the operator needs to diagnose CPU imbalance: one CPU handling most of the work, another idle, or queue-depth variance between CPUs.

A procedural sysctl that returns a per-CPU array:

```
static int
perfdemo_sysctl_percpu(SYSCTL_HANDLER_ARGS)
{
    uint64_t values[MAXCPU];
    int cpu;

    for (cpu = 0; cpu < mp_ncpus; cpu++) {
        struct perfdemo_cpu_stats *s =
            DPCPU_ID_PTR(cpu, perfdemo_cpu_stats);
        values[cpu] = s->cpu_time_ns;
    }
    return (SYSCTL_OUT(req, values, mp_ncpus * sizeof(uint64_t)));
}
```

The handler visits each active CPU, reads its DPCPU state, and copies the values into a contiguous buffer. The userland reader sees an array of `uint64_t` sized `mp_ncpus`. For exporting to a monitoring tool, per-CPU presentation gives a detailed picture that aggregation obscures.

33.10.11 Stable Interfaces for Operators

A driver's sysctl tree is an interface; once operators write scripts or dashboards against it, changing the names breaks things. A few rules help keep the tree stable over years.

First, **name things for what they are, not for how they are implemented**. A sysctl called `reads` describes a user-observable concept; a sysctl called `atomic_counter_0` describes an implementation detail. The former is stable across refactors; the latter forces a rename whenever the implementation changes.

Second, **version the interface if it has to change**. If you add a new field to an existing structure-valued sysctl, older scripts will still read the old size correctly. If you rename a node, add the new name as an alias first, then deprecate the old one over at least one release cycle.

Third, **document every node in the descr string**. The last argument to `SYSCTL_*` macros is a short description that appears in `sysctl -d`. Keep it accurate and useful; it is the only inline documentation an operator has when diagnosing at three in the morning.

Fourth, **avoid driver-internal concepts in names**. A counter named `uma_zone_free_count` requires the operator to know what UMA is; a counter named `free_buffers` describes what it counts in terms any operator can understand.

A driver that follows these rules produces a sysctl tree that ages well. The exercise in Section 7 is a chance to practise.

33.10.12 Handling Sysctl Writes Safely

A writable sysctl is a public API for changing driver state at runtime. Every path that accepts user input must validate, synchronise, and report cleanly on failure. The pattern in production drivers is:


```
static int
perfdemo_sysctl_mode(SYSCTL_HANDLER_ARGS)
{
    struct perfdemo_softc *sc = arg1;
    int new_val, error;

    new_val = sc->mode;
    error = sysctl_handle_int(oidp, &new_val, 0, req);
    if (error != 0 || req->newptr == NULL)
        return (error);

    if (new_val < 0 || new_val > 2)
        return (EINVAL);

    PD_LOCK(sc);
    if (sc->state != PD_STATE_READY) {
        PD_UNLOCK(sc);
        return (EBUSY);
    }
    sc->mode = new_val;
    perfdemo_apply_mode(sc);
    PD_UNLOCK(sc);

    return (0);
}
```

Three things are happening. First, the handler validates the value; out-of-range values return `EINVAL` rather than corrupting the driver. Second, the handler takes the driver's lock before mutating state; another CPU reading or writing the mode concurrently does not see half-updates. Third, the handler checks the driver's lifecycle state; changes are rejected if the driver is detaching or in error. Each of these is a small addition to the simplest possible handler, and each prevents a real class of bug.

A related consideration is whether the sysctl change is idempotent. If the new value equals the current value, the handler should do nothing (or at most confirm the current state). If the new value differs, the handler must change state atomically so that no one sees a partial update. The lock-then-validate-then-apply pattern above satisfies both constraints.

33.10.13 Exercise 33.7: A Reporting sysctl Tree

Build out the `perfdemo` driver's sysctl tree with at least the following nodes:

- `dev.perfdemo.<unit>.stats.reads`: total reads.
- `dev.perfdemo.<unit>.stats.errors`: total errors.
- `dev.perfdemo.<unit>.stats.bytes`: total bytes.
- `dev.perfdemo.<unit>.stats.latency_avg_ns`: average latency in nanoseconds.

- `dev.perfdemo.<unit>.config.buffer_size`: buffer size (tunable, 16..4096).
- `dev.perfdemo.<unit>.config.debug`: debug level (0..2).

Use `counter(9)` for the counters; use a procedural `sysctl` for the derived latency average; use `CTLFLAG_RWTUN` for the tunables. Build and load the driver; verify each `sysctl` returns a sensible value; change a tunable at runtime and confirm the driver respects it.

The exercise produces an observable interface the rest of the chapter can rely on. It is also a nice self-contained piece of work that is worth adding to a portfolio.

33.10.14 Wrapping Up Section 7

The `sysctl` tree is the driver's always-on observability interface. A thoughtful tree exposes totals, rates, states, and configuration. `counter(9)` values need procedural `sysctls` to fetch their summed totals. Rate-limited logging via `log(9)` and `ratecheck(9)` keeps informational messages from drowning the log. Debug modes give operators a safe way to enable verbose tracing when diagnosing problems. Each of these is a small investment; the cumulative effect is a driver that is auditable from the command line, diagnosed without rebooting, and maintained confidently after the author has moved on.

The last instructional section, Section 8, is the discipline section. It teaches you how to clean up after a tuning project, document the benchmarks that produced the result, update the manual page with the knobs the driver now exposes, and ship the optimised driver as a version the system can rely on.

33.11 Section 8: Refactoring and Versioning After Tuning

Tuning adds scaffolding. Counters that measured specific hot paths, DTrace probes that answered a question once, print statements from an all-nighter, commented-out variants you want to keep handy: all of this accumulates during a performance project. The same discipline that produced the measurements produces the final cleanup. A driver that lives in the tree after tuning is the one whose author knew when to stop and how to leave the code in a state a maintainer can still read.

This section is about that cleanup. It covers what to remove, what to keep, how to document the work, and how to version the result.

33.11.1 Removing Temporary Measurement Code

The most important cleanup is the removal of temporary measurement code. During tuning you may have added:

- `printf()` calls that trace specific operations.
- Counters that helped you find one bottleneck but do not belong in production.
- Timings that accumulated into global arrays.
- DTrace-style probes in the middle of hot paths that are not formally SDT-declared.
- Commented-out versions of optimisations you tried and discarded.

Each of these is a future maintenance cost. The rule is simple: if a piece of code was only useful for the tuning session, delete it. Version control keeps the history; the commit message explains what you tried. The working-tree code should read as if the measurements had never been there.

A useful discipline is to **mark temporary code with a comment when you add it**, so you know what to remove later:

```
/* PERF: added for v2.3 tuning; remove before ship. */
printf("perfdemo: read entered at %ju\n", (uintmax_t)sbinuptime());
```

When the tuning project is done, `grep` for `PERF:` and remove every line. A driver with no `PERF:` markers is one that has been cleaned up; a driver with a dozen is one where the author forgot.

33.11.2 What to Keep

Not every piece of measurement code should go. Some of it has lasting value and belongs in the shipped driver. The criteria:

- **Keep counters that tell operators what the driver is doing.** Total reads, writes, errors, bytes: these belong in the `sysctl` tree forever.
- **Keep SDT probes at operational boundaries.** They cost nothing when disabled and give any future engineer an instant way to measure the driver.
- **Keep configuration knobs that expose meaningful trade-offs.** The buffer size, the coalescing threshold, the debug level: these are interfaces to the operator.
- **Remove one-off prints and hand-written timings.** These were for one tuning session and have no place in shipped code.
- **Remove counters that are too specific to the question you were answering.** A counter named `reads_between_cache_miss_A_and_cache_miss_B` probably has no lasting value; a counter named `reads` does.

The test is: *would a future engineer benefit from this information six months from now?* If yes, keep it. If no, remove it.

33.11.3 Benchmarking the Final Driver

After the scaffolding is gone, benchmark the final driver one more time. This benchmark is the one that becomes the *published* performance of the driver; record it in your logbook and in a plain-text document in the driver's source tree. A benchmark report should include:

- The driver version (for example, `perfdemo 2.3`).
- The machine: CPU model, core count, RAM size, FreeBSD version.
- The workload: what `dd` command, what file sizes, what userland threads, what coalescing window.
- The result: throughput, latency (P50, P95, P99), CPU usage.
- Context: what else was running, any relevant `sysctl` settings.

A reader who picks up the driver months later should be able to reproduce the benchmark without asking you. That reproducibility is the purpose of the report.

33.11.4 Documenting Tuning Knobs

If the driver now exposes tunable sysctls that affect performance, document them. The right place is the driver's manual page (an .4 page under /usr/share/man/man4/) or a section at the top of the driver's source file. A manual page section looks like:

```
.Sh TUNABLES
The following tunables can be set at the
.Xr loader.conf 5
prompt or at runtime via
.Xr sysctl 8 :
.Bl -tag -width "dev.perfdemo.buffer_size"
.It Va dev.perfdemo.buffer_size
Size of the internal read buffer, in bytes.
Default 1024; valid range 16 to 4096.
Larger values increase throughput for bulk reads
but raise per-call latency.
.It Va dev.perfdemo.debug
Verbosity of debug logging.
0 logs errors only.
1 adds transient-error notes.
2 logs every operation.
Default 0; higher values should only be used
during diagnostic sessions.
.El
```

The format is arcane on a first look, but every FreeBSD manual page uses the same pattern. The existing .4 pages under /usr/src/share/man/man4/ are excellent templates; copy one whose style matches your driver and adapt.

For drivers that do not ship a manual page yet, this is the moment to write one. A driver without a manual page is underdocumented; one with a manual page is at a credibility threshold expected of base-system code.

33.11.5 Versioning

A tuned driver is a new version of the driver. Mark it with `MODULE_VERSION( )` in the driver source:

```
MODULE_VERSION(perfdemo, 3);  /* was 2 before this tuning pass */
```

The version number is consumed by `kldstat -v` and by other modules that declare `MODULE_DEPEND( )` against yours. Bumping it signals that the driver's behaviour has changed enough for consumers to care.

For a shipped driver, the convention is a major version for breaking changes, a minor version (or version suffix) for compatible additions, and a patch version for bug fixes. A pure performance tuning pass that adds no new functionality is a patch; a tuning pass that adds new sysctls is a minor; a tuning pass that changes the semantics of

existing interfaces is a major. The book's exercises call a tuned driver `v2.3-optimized`; in reality, the version scheme is the one your project uses.

Updating the changelog is part of the versioning work. A `CHANGELOG.md` or a comment in the driver source is the right place. Each entry should include:

- The version and date.
- What changed at a high level (tuning, bug fix, new feature).
- The benchmark numbers if they changed.
- Any backward-incompatible changes operators need to know about.

The habit of keeping the changelog current is the habit that makes long-lived code maintainable. Drivers without changelogs accumulate lore; drivers with them accumulate history.

33.11.6 Reviewing the Diff

Before shipping, read the whole diff of the tuning work as if you were a new maintainer reviewing the change. Look for:

- Code paths that have become harder to reason about.
- Comments that no longer match the code.
- Temporary code that survived the cleanup.
- Error paths that were changed but not retested.
- Locking that was added or removed.

This review catches the problems your own measurements did not. A piece of code that is correct but unreadable is a problem waiting to happen.

33.11.7 The Performance Report

The final deliverable of a tuning project is a short written report. Not a blog post; not a presentation; a plain document that lives with the driver. A useful template:

```
perfdemo v2.3-optimized - Performance Report
=====
```

```
Summary: v2.3 is a pure-tuning release that reduces median read
latency from 40 us to 12 us and triples throughput on a 4-core
amd64 test machine, without changing the driver's interface.
```

```
Goals (set before tuning):
```

- ```
- Median read() latency under 20 us.
- P99 read() latency under 100 us.
- Single-thread throughput above 500K ops/sec.
- Multi-thread throughput scaling linearly up to 4 CPUs.
```

```
Before (v2.2):
```

- Median read: 40 us.
- P99 read: 180 us.
- Single-thread: 180K ops/sec.
- 4-thread: 480K ops/sec (2.7x; sublinear).

After (v2.3):

- Median read: 12 us.
- P99 read: 65 us.
- Single-thread: 520K ops/sec.
- 4-thread: 1.95M ops/sec (3.75x; near-linear).

Changes applied:

1. Replaced single atomic counter with counter(9).
2. Cache-line aligned hot softc fields.
3. Pre-allocated read buffers in attach().
4. Switched to UMA zone for per-request scratch buffers.
5. Added sysctl nodes for stats and config (non-breaking).

Measurements:

All numbers from DTrace aggregations over 100,000-sample windows, 4-core amd64, 3.0 GHz Xeon, FreeBSD 14.3 GENERIC, otherwise idle. See `benchmark/v2.3/` for raw data and scripts.

Tunables introduced:

- `dev.perfdemo.buffer_size`: runtime buffer size (default 1024).
- `dev.perfdemo.debug`: runtime debug verbosity (default 0).

Risks:

Cache-line alignment increases softc memory by approximately 200 bytes per instance. Unlikely to matter on modern systems but worth noting for memory-constrained embedded targets.

Remaining work:

None for v2.3. Future tuning may investigate reducing P99 latency further if workload analysis shows a specific cause.

A report like this becomes institutional knowledge. A maintainer two years from now reading this document has everything needed to understand what was done, why, and how to reproduce it.

### 33.11.8 A/B Testing Tuning Changes

Before committing a tuning change to the tree, the responsible thing is to test it against the previous version under the same workload. An A/B test compares the two versions against a single benchmark, multiple times,

under the same system state. If the new version is measurably better *and* the difference survives the noise of multiple runs, the change is worth keeping.

A simple A/B harness:

```
#!/bin/sh
ab-test.sh <module-a-name> <module-b-name> <runs>

MODULE_A=$1
MODULE_B=$2
RUNS=$3

echo "Module A: $MODULE_A"
echo "Module B: $MODULE_B"
echo "Runs per module: $RUNS"
echo

for i in $(seq 1 $RUNS); do
 sudo kldload ./"$MODULE_A".ko
 time ./run-workload.sh 100000
 sudo kldunload "$MODULE_A"
 echo "A$i done"

 sudo kldload ./"$MODULE_B".ko
 time ./run-workload.sh 100000
 sudo kldunload "$MODULE_B"
 echo "B$i done"
done
```

Run this with `./ab-test.sh perfdemo-v22 perfdemo-v23 10`. The script alternates modules to prevent warm-up effects from biasing one in the other's favour. Ten runs each is usually enough to distinguish a 5% difference from noise.

A/B testing matters for two reasons. First, it forces you to state the comparison as *version A vs version B under workload X*, which is the shape a performance claim should have. Second, it catches regressions: if v2.3 is slower than v2.2 on some axis you did not measure, the A/B run shows it. A tuning change that improves one metric while hurting another is surprisingly common; only a disciplined comparison surfaces them.

### 33.11.9 The Benchmark Harness

A sophisticated project eventually accumulates a proper benchmark harness: a reusable script that runs a known workload, collects a known set of metrics, and writes the results in a known format. A harness is worth building when:

- You run the same benchmark more than three or four times.
- Multiple people need to reproduce the benchmark.

- The results are used in performance reports and must be comparable across runs.

The harness typically includes:

- A setup script that loads the driver, configures its sysctls, and verifies the system is idle.
- A runner script that invokes the workload for a fixed duration or a fixed number of iterations.
- A collector that captures metrics before and after, computes deltas, and writes a structured report.
- A teardown script that unloads the driver and restores any system state.
- A results archive that preserves raw and processed outputs with timestamps and run identifiers.

For `perfdemo`, the harness lives in the `lab` directory as `final-benchmark.sh`. For a production driver, it lives in the source tree alongside the driver, so anyone can reproduce the results.

The details of the harness depend on the driver. What matters is that there is *some* harness; ad-hoc benchmarks that someone has to remember how to run are ad-hoc evidence that someone has to remember how to trust.

### 33.11.10 Sharing Performance Numbers With the Project

For drivers that go into the FreeBSD base system or a significant port, the performance numbers are not a private deliverable. They become part of the project record: commit messages, mailing-list threads, release notes. The conventions for sharing are worth knowing.

**Commit messages** should summarise the change, the benchmark, and the result in the body. A good performance-related commit message looks like:

```
perfdemo: switch counters to counter(9) for better scaling
```

```
Single counters in the softc were contended on multi-CPU systems.
Switching to counter(9) reduces per-call overhead and allows the
driver to scale to higher throughput under parallel load.
```

```
Measured on an 8-core amd64 Xeon at 3.0 GHz, FreeBSD 14.3-STABLE:
Before: 480K ops/sec at 4-thread peak.
After: 1.95M ops/sec at 4-thread peak (4x).
```

```
Benchmark data and scripts are in tools/perfdemo-bench/.
```

The first line is a short summary. The body expands on what changed, why, and what the measurement showed. A reader scanning commit history can understand the significance of the change in 60 seconds.

**Mailing-list threads** announcing the change follow the same structure but can include more context: the goal of the tuning, the alternatives considered, any caveats. Link to benchmark scripts and raw data if the scale of the change warrants discussion.

**Release notes** are terser. One or two lines: *The perfdemo(4) driver now uses counter(9) for its internal statistics. This reduces overhead on multi-CPU systems and allows higher throughput under parallel load.*

Each audience gets what it needs. The pattern is that every performance claim, at every level of detail, points at a reproducible measurement.



### 33.11.11 Exercise 33.8: Ship perfdemo v2.3

Using the state of perfdemo you built across Sections 2 through 7, produce a v2.3-optimized release:

1. Remove every PERF: marker and the scaffolding it denoted.
2. Decide what stays in the sysctl tree and what was only useful during tuning.
3. Update MODULE\_VERSION( ) to 3 (or your chosen new number).
4. Update the driver's manual page (or write one if it does not exist) with the TUNABLES section.
5. Run the final benchmark and record the numbers.
6. Write the performance report.

The exercise is the entire chapter's output in a concrete form. When you finish, you have a driver that would pass review in any serious FreeBSD project, with the performance measurements that justify its state.

### 33.11.12 Wrapping Up Section 8

Tuning is only half a project. The other half is the discipline of cleaning up, documenting, and shipping. A driver with clean measurement primitives, a thoughtful sysctl tree, honest manual-page documentation, an updated version number, and a written performance report is the driver that ages well. The shortcut of skipping any of these steps is tempting, but the cost is paid by the next maintainer, which might be your future self.

The instructional portion of the chapter is complete. The next sections are the hands-on labs, the challenge exercises, the troubleshooting reference, the Wrapping Up, and the bridge into Chapter 34.

## 33.12 Putting It All Together: A Complete Tuning Session

Before the labs, a worked walkthrough that ties the eight sections into a single story. This is the shape a real tuning session takes, compressed into a few pages. The session is not fictional; it is the same pattern that produces most of the performance improvements in the FreeBSD tree.

**The driver.** perfdemo v2.0 is a working character device that produces synthetic data on read(). Its hot path takes a mutex, allocates a temporary buffer via malloc(9), fills it with pseudo-random data, copies it to userland via uiomove(9), frees the buffer, releases the mutex, and returns. It has a single atomic counter for total reads and no sysctl tree beyond what FreeBSD's device\_t gives it for free.

**The goal.** A user reports that perfdemo is slow in their production workload: they run forty concurrent reader threads, and throughput saturates at about 600,000 ops/sec across all threads on their 16-core amd64 server. They want 2 million or better. The stated goal is *median read( ) latency under 20 us and aggregate throughput above 2 million ops/sec on a 16-core amd64 Xeon running FreeBSD 14.3 GENERIC, with forty reader threads pinned across CPUs 0-15.*

**Measurement, round 1.** Before changing anything, we need numbers. We load the driver, run the user's workload, and collect baseline data.

vmstat -i does not show any unusual interrupt rates, so we rule out interrupt problems early. top -H shows the system's CPU at 65%; the driver is clearly not the only consumer, but it is a significant one. We instrument the driver minimally: one counter(9) for total reads, one for total nanoseconds of latency, both exposed via

a procedural sysctl. We run the workload for sixty seconds and record the average latency as 52 microseconds. The goal is under 20. We are 32 microseconds off.

**DTrace, round 1.** With the baseline in hand, we reach for DTrace. A one-liner to see the function-level latency distribution:

```
dtrace -n '
fbt:perfdemo:perfdemo_read:entry { self->s = timestamp; }
fbt:perfdemo:perfdemo_read:return /self->s/ {
 @lat = quantize(timestamp - self->s);
 self->s = 0;
}'
```

After a minute the histogram shows a clear shape. P50 is around 30 us; P95 is 150 us; P99 is a surprising 1.5 ms. The long tail is where the user's pain comes from.

We widen the investigation. Another one-liner:

```
dtrace -n 'lockstat:::adaptive-block {
 @[((struct lock_object *)arg0)->lo_name] = count();
}'
```

The top entry: *perfdemo\_softc\_mtx* with about 30,000 blocks during the one-minute window. Our driver's mutex is contended; forty reader threads are serialising on it.

**The first fix.** The mutex protects a counter and a pool pointer; in principle the counter could be per-CPU and the pool pointer could be lockless. We replace the atomic counter with `counter_u64_add()`, which does not require the driver's lock, and we rework the pool into a per-CPU array of buffers (one per CPU). The hot path no longer needs to take the mutex for the common case; the mutex only guards the pool's administrative paths (init, fini, resize).

We rebuild, reload, rerun. Average latency drops from 52 us to 14 us. Throughput climbs from 600K to 1.8M ops/sec. We are close to the goal but not there.

**DTrace, round 2.** With the mutex out of the way, we profile again:

```
dtrace -n 'profile-997 /arg0 != 0/ { @[stack()] = count(); }'
```

The top stack now shows most of the time in `uiomove`, `malloc`, and `free`. The allocator is the next bottleneck; every read is allocating and freeing a temporary buffer. The counter-level cost is negligible.

**The second fix.** We switch from `malloc(9)` on the hot path to a UMA zone created at attach, with `UMA_ALIGN_CACHE` for per-CPU-aligned items. The zone is sized for the expected working set; `vmstat -z` confirms the zone population stabilises a few seconds after workload start.

Rebuild, reload, rerun. Average latency drops from 14 us to 9 us. Throughput climbs from 1.8M to 2.3M ops/sec. We have met the goal: median under 20 us and throughput above 2 million.

**pmcstat, round 1.** We are inside the goal, but before declaring victory we want to know whether the driver is now well-balanced across CPUs. We run a sampling session:

```
pmcstat -S cycles -O /tmp/perfdemo.pmc ./run-workload.sh
pmcstat -R /tmp/perfdemo.pmc -T
```

The top functions are `uiomove`, `perfdemo_read`, and a couple of cache-coherence primitives from the kernel. No single bottleneck; the driver is now spending its time on the work it should be doing. Good.

A counting session:

```
pmcstat -s cycles -s instructions ./run-workload.sh
```

Prints an IPC of about 1.8. That is healthy for mostly-memory-moving code; we are not leaving hardware throughput on the table.

**Cleanup.** We read the diff. Four `PERF:` markers remain from the investigation; we remove them. Two commented-out variants exist from the second fix; we remove them. A debug-level `sysctl` that we added at one point but never used finds its way in; we remove it too.

**Sysctl tree.** The tree now has `dev.perfdemo.0.stats.reads`, `dev.perfdemo.0.stats.errors`, `dev.perfdemo.0.stats.bytes`, `dev.perfdemo.0.stats.lat_hist` (the histogram), and `dev.perfdemo.0.config.debug` (a level-0-to-2 debug verbosity). Each is documented in the `descr` string.

**Manual page.** A paragraph in the existing `.4` page describes the new tunables. The benchmark-report document describes the tuning and the numbers.

**Final numbers.** On the 16-core Xeon under the user's workload:

- Median `read()` latency: 9 us.
- P95 `read()` latency: 35 us.
- P99 `read()` latency: 95 us.
- Throughput: 2.3 million ops/sec.

**Time invested.** Roughly one engineer-day from first measurement to final report. The two fixes are small; finding them took longer than writing them. That ratio is typical.

**Lessons.** Three points stand out.

First, the investigation did not start with code changes. It started with measurement, moved through DTrace, and only then changed code. Every change was motivated by a specific observation.

Second, two fixes were enough. The mutex and the allocator were the two bottlenecks; the third-order optimisations (cache alignment on small fields, NAPI-style batching, MSI-X) were unnecessary. The chapter described them all because you have to know they exist; the session showed that you apply them selectively.

Third, the scaffolding was temporary. `PERF:` markers, commented-out variants, an unused debug level: all removed before ship. The cleaned-up driver looks simple because the tuning left little trace. That simplicity is the goal.

The labs below walk through the same shape of session, with concrete code and commands, at each stage.

## 33.13 Hands-On Labs

This chapter's labs are built around a small pedagogical driver called `perfdemo`. The driver is a character device that synthesises data on `read()` at a controlled rate, and exposes a `sysctl` tree. It is deliberately uncomplicated; the interesting part is what the labs do *with* it. Each lab takes the driver through one stage of the performance workflow: baseline measurement, DTrace tracing, PMC sampling, memory tuning, interrupt tuning, and final cleanup.

All files for the labs live under `examples/part-07/ch33-performance/`. Each lab directory has its own `README.md` with build and run instructions. Clone the directory, follow the `README` for each lab, and work through them in order.

### 33.13.1 Lab 1: Baseline `perfdemo`, Counters, and `sysctl`

**Objective:** Build the baseline `perfdemo` driver, load it, and exercise its `sysctl`-based counters. This lab is the foundation for every other lab in the chapter; take the time to get it right.

**Directory:** `examples/part-07/ch33-performance/lab01-perfcounters/`.

**Prerequisites:** A FreeBSD 14.3 system (physical, virtual, or jail with kernel build tools), the FreeBSD source tree at `/usr/src`, and root access to load kernel modules.

**Steps:**

1. Change into the lab directory: `cd examples/part-07/ch33-performance/lab01-perfcounters/`.
2. Read `perfdemo.c` to familiarise yourself with the baseline driver. Note the structure: a `module_event_handler`, a `probe/attach/detach` set, a `cdevsw` with a `read` method, a `softc`, and the `sysctl` tree under `hw.perfdemo`.
3. Build the module with `make`. If the build fails, check that `/usr/src` is populated and that you can build other kernel modules (the easiest test: `cd /usr/src/sys/modules/nullfs && make`).
4. Load the module: `sudo kldload ./perfdemo.ko`. Confirm it is loaded: `kldstat | grep perfdemo`.
5. Check the `sysctl` tree: `sysctl hw.perfdemo`. You should see:

```
hw.perfdemo.reads: 0
hw.perfdemo.writes: 0
hw.perfdemo.errors: 0
hw.perfdemo.bytes: 0
```
6. Verify the device node exists: `ls -l /dev/perfdemo`.
7. In one terminal, start a watch: `watch -n 1 'sysctl hw.perfdemo'`.
8. In another terminal, run the workload: `./run-workload.sh 100000`. The script issues 100,000 reads against `/dev/perfdemo`. It prints its wall-clock time on completion.
9. Verify the counters moved as expected. After the workload, `hw.perfdemo.reads` should be around 100,000 (a few extras from ad-hoc reads are normal).
10. Run the workload at a larger size: `./run-workload.sh 1000000`. Record the wall-clock time again. Divide `1000000 / wallclock_seconds` to get ops/sec.

11. Unload the driver: `sudo kldunload perfdemo`.

**What you should see:**

- The sysctl counters increase as the workload runs.
- The final values should match the workload's request count (within one or two operations of variance).
- The wall-clock time of the 1M-read run should be a few seconds on a modern machine.
- No kernel warnings in `dmesg`.

**What you should record in your logbook:**

- The wall-clock time and ops/sec of both workload runs.
- The machine: CPU model, core count, RAM, FreeBSD version.
- Any `dmesg` output from the module-load and module-unload events.
- The final sysctl values.

**Expected baseline numbers (for comparison):**

On an 8-core amd64 Xeon at 3.0 GHz running FreeBSD 14.3-RELEASE with the baseline driver, a 1M-read single-threaded workload completes in roughly 3 seconds, giving about 330,000 ops/sec. Your numbers will differ based on hardware, but the order of magnitude should be similar.

**Common mistakes:**

- Forgetting to load the driver before running the workload. The loader script will fail with `ENODEV` if `/dev/perfdemo` is missing.
- Running the loader script with a very small count (for example, 100) and expecting stable numbers. The noise of measurement starts to dominate below a few thousand iterations. Use 100,000 or more for baseline runs.
- Confusing `hw.perfdemo.reads` with `hw.perfdemo.bytes`. The counter for total reads is distinct from the counter for total bytes. A 100-byte read increments reads by one and bytes by 100.
- Forgetting to unload the driver before rebuilding. A stale `.ko` in the kernel can cause confusing symptoms on the next load.

**Troubleshooting:** If the module fails to load with *module version mismatch*, your kernel version does not match the one `perfdemo` was built against. Rebuild after ensuring `/usr/obj` and `/usr/src` match your running kernel.

### 33.13.2 Lab 2: DTrace perfdemo

**Objective:** Use DTrace to understand what `perfdemo`'s read path is doing, measured from outside without modifying the driver.

**Directory:** `examples/part-07/ch33-performance/lab02-dtrace-scripts/`.

**Prerequisites:** Lab 1 complete; the `perfdemo` driver loaded.

**Steps:**

1. Ensure the `perfdemo` driver is loaded (from Lab 1). If not, `cd` to `lab01-perfcounters/` and `sudo kldload ./perfdemo.ko`.

2. Load the DTrace providers: `sudo kldload dtraceall`. You can verify with `sudo dtrace -l | head` (it will list a few probes).
3. In one terminal, run the workload continuously: `while :; do ./run-workload.sh 10000; done`. Let it churn.
4. In another terminal, run the simple count one-liner:

```
sudo dtrace -n 'fbt:perfdemo:perfdemo_read:entry { @ = count(); }'
```

Let it run for 30 seconds, then Ctrl-C. You should see a number in the tens of thousands. Divide by 30 to get reads/sec, which should match what `hw.perfdemo.reads delta` shows.

5. Run the latency histogram script `read-latency.d` in this lab directory:

```
sudo dtrace -s read-latency.d
```

Let it run for 30 seconds. The histogram will look something like:

| value  | ----- Distribution ----- | count |
|--------|--------------------------|-------|
| 512    |                          | 0     |
| 1024   | @                        | 125   |
| 2048   | @@@@@                    | 520   |
| 4096   | @@@@@@@@@@@@@@@@@@@@@@@@ | 1850  |
| 8192   | @@@@@@@@@@@@@@@@@@@@     | 1320  |
| 16384  | @@@                      | 290   |
| 32768  |                          | 35    |
| 65536  |                          | 8     |
| 131072 |                          | 2     |
| 262144 |                          | 0     |

Record the P50 (the bucket that contains the median) and the P99 (the bucket that crosses 99% cumulative).

6. Run the lock-contention script `lockstat-simple.d`:

```
sudo dtrace -s lockstat-simple.d
```

Let it run for 30 seconds. Look for `perfdemo` or `perfdemo_softc_mtx` in the output. The baseline driver *will* show contention on its mutex if you run the workload with concurrency.

7. Run the profile-sampling script `profile-sample.d`:

```
sudo dtrace -s profile-sample.d
```

Let it run for 60 seconds. The output lists the top kernel stacks by sample count. `perfdemo_read` should appear among the top entries.

8. Stop the workload (Ctrl-C in its terminal).

### What you should see:

- `read-latency.d` produces a histogram with a bell-like peak around the median, and a tail to the right.

- `lockstat-simple.d` produces a list of locks touched during the measurement; if the workload is concurrent, `perfdemo_softc_mtx` appears.
- `profile-sample.d` produces a list of function stacks; `perfdemo_read`, `uiomove`, `copyout`, and `_mtx_lock_sleep` are typical entries.

**What you should record in your logbook:**

- P50 and P99 latency.
- Top three functions from `profile-sample.d`.
- The contended lock names from `lockstat-simple.d`.

**Expected results (for comparison):**

On the same 8-core Xeon, the baseline driver under 4-thread concurrent read load shows a median around 8 us, P99 around 60 us, heavy contention on `perfdemo_softc_mtx`, and top profile entries dominated by `perfdemo_read`, `uiomove`, and `_mtx_lock_sleep`.

**Common mistakes:**

- Running DTrace before the driver is loaded. The `fbt:perfdemo: probes` do not exist until `perfdemo.ko` is in the kernel.
- Writing scripts that print per-event. The DTrace buffer can drop events; always aggregate and print at the end.
- Forgetting to Ctrl-C the DTrace session at the right moment. Some aggregations only print on explicit stop.
- Running the workload at too low a rate to produce interesting data. DTrace latency histograms need at least tens of thousands of samples for clean percentile estimates.
- Forgetting to load `dtraceall`. A bare DTrace install without providers will complain about unknown providers.

**Troubleshooting:** If `dtrace -l | grep perfdemo` is empty after the driver is loaded, check that `dtraceall` is loaded (`kldstat | grep dtrace`). If the probes still do not appear, the driver may have been compiled without debug symbols; check that make used the default flags and did not strip `-g`.

### 33.13.3 Lab 3: `pmcstat perfdemo` (optional, requires `hwpmc`)

**Objective:** Sample `perfdemo` with hardware performance counters and interpret the output. This lab is richer than the others because `pmcstat` has more knobs than DTrace; plan on spending extra time.

**Directory:** `examples/part-07/ch33-performance/lab03-pmcstat/`.

**Prerequisites:** A physical or fully paravirtualised machine where PMCs are available. On most cloud VMs and on shared hypervisors, PMCs are restricted; if `pmcstat -L` lists only a handful of events or the tool refuses to start, follow the alternative path with DTrace's `profile` provider.

**Steps:**

1. Ensure `hwpmc` is loaded: `sudo kldload hwpmc` (or confirm it is already loaded with `kldstat | grep hwpmc`). Check `dmesg | tail` after loading; you should see a line like `hwpmc: SOFT/16/64/0x67<REA,WRI,INV,LOG,EV1,EV2,CAS>`.

2. Verify the tool sees events: `pmcstat -L | head -30`. You should see a mix of portable event names (cycles, instructions, cache-references, cache-misses) and vendor-specific names (for Intel, entries starting with `cpu_clk_`, `inst_retired`, `uops_issued`, and so on).
3. Run a counting session during a workload. The `-s` flag (lowercase) is for system-wide counting; use this to see the driver's contribution plus everything else the kernel is doing:

```
sudo pmcstat -s instructions -s cycles -O /tmp/pd.pmc \
 ./run-workload.sh 100000
```

After the workload finishes, `pmcstat` prints the totals:

```
p/instructions p/cycles
 3.2e9 9.5e9
```

Divide instructions by cycles to compute IPC: 3.2 billion / 9.5 billion = 0.34. An IPC below 1 on a modern CPU usually indicates stalls; likely memory or branch mispredicts.

4. Run a sampling session. The `-S` flag (uppercase) configures sampling:

```
sudo pmcstat -S cycles -O /tmp/pd-cycles.pmc \
 ./run-workload.sh 100000
pmcstat -R /tmp/pd-cycles.pmc -T
```

The `-T` output is the top-N function list. You should see something like:

| %SAMP | CUM  | IMAGE  | FUNCTION          |
|-------|------|--------|-------------------|
| 13.2  | 13.2 | kernel | perfdemo_read     |
| 9.1   | 22.3 | kernel | uiomove_faultflag |
| 8.5   | 30.8 | kernel | copyout           |
| 6.2   | 37.0 | kernel | _mtx_lock_sleep   |
| 4.9   | 41.9 | kernel | _mtx_unlock_sleep |
| ...   |      |        |                   |

Note the top five functions and their percentages.

5. Run a second sampling session, this time with LLC-misses to see where cache misses happen (requires Intel CPU):

```
sudo pmcstat -S LLC-misses -O /tmp/pd-llc.pmc \
 ./run-workload.sh 100000
pmcstat -R /tmp/pd-llc.pmc -T
```

Compare the top entries to the cycle samples. If the same functions appear in both, the time is memory-bound; if different functions appear, the cycle-hot functions are CPU-bound (arithmetic, branches).

6. If you have the FlameGraph tools installed (install them from the `sysutils/flamegraph` port: `sudo pkg install flamegraph`), generate an SVG:

```
sudo pmcstat -R /tmp/pd-cycles.pmc -g -k /boot/kernel > pd.stacks
stackcollapse-pmc.pl pd.stacks > pd.folded
flamegraph.pl pd.folded > pd.svg
```



Open `pd.svg` in a browser. The SVG is interactive; click any box to zoom into that call stack.

7. Inspect the flame graph. The bottom boxes (system call entry) should be wide and boring. Above them, your driver's functions appear as narrower stacks. The width of each box is its fraction of CPU time; the height shows call depth.

**Alternative if `hwpmc` is unavailable:** Use DTrace's profile provider:

```
sudo dtrace -n 'profile-997 /arg0 != 0/ { @[stack()] = count(); }' \
-o /tmp/pd-prof.txt
```

Let it run for a minute while the workload runs, Ctrl-C, then examine `/tmp/pd-prof.txt`. The output is similar to `pmcstat`'s callgraph, though coarser because the profile sample rate is fixed.

**What you should record in your logbook:**

- The IPC from the counting session.
- The top five functions from the sampling session, with their percentages.
- The top three functions in the LLC-misses session (if available) and whether they matched the cycle-samples list.
- A screenshot or description of the flame graph's dominant stacks.

**Expected baseline numbers (for comparison):**

On the baseline driver on an 8-core Xeon, IPC is typically 0.3 to 0.5 (memory-bound). The top function is usually `perfdemo_read` at 10-15% of cycles, with `uiomove`, `copyout`, and `_mtx_lock_sleep` following. After tuning (Labs 4 and 6), IPC should climb above 1.0 and `_mtx_lock_sleep` should disappear from the top entries.

**Common mistakes:**

- Forgetting to load `hwpmc` before running `pmcstat`. The tool will report *no device* or *no PMCs available*.
- Running the sampling session for too short a window. A few seconds of samples gives noisy data; a minute is usually enough.
- Misreading `pmcstat -T`'s output. The `%SAMP` column is the fraction of samples a function received across all CPUs; a 10% value means 10% of time, not 10% of instructions.
- Using the wrong flag: `-s` is counting, `-S` is sampling. They produce different outputs and the difference is easy to miss when reading someone else's scripts.
- Trying to sample a function that got inlined. `inst_retired.any` counts instructions retired; an inlined function's instructions are counted against the caller, not the inlined one. If you expect to see function X in the samples and it does not appear, check with `objdump -d perfdemo.ko | grep X` to see whether X was compiled as a real function.

**Troubleshooting:** If `pmcstat` hangs or prints *ENOENT*, the event name is wrong. `pmcstat -L` lists every event name the kernel knows about on your CPU; pick one from that list. If the tool crashes with a signal, `hwpmc` may not be fully initialised; unload and reload the module.

### 33.13.4 Lab 4: Cache Alignment and Per-CPU Counters

**Objective:** Measure the performance impact of cache-line alignment and the counter (9) API compared to a plain atomic, under concurrent multi-CPU load.

**Directory:** examples/part-07/ch33-performance/lab04-cache-aligned/.

**Prerequisites:** Labs 1 and 2 complete, so you know the baseline driver's behaviour and how to use DTrace.

**Steps:**

1. Build three variants of the driver, each in its own subdirectory:

- v1-atomic: single atomic counter, default layout.
- v2-aligned: single atomic counter with `__aligned(CACHE_LINE_SIZE)` and padding around it to isolate the cache line.
- v3-counter9: `counter(9)` counter instead of an atomic.

Each subdirectory has its own Makefile. Run `make` in each. The only differences between the three sources are in the `softc` layout and the counter increment lines.

2. Identify your CPU count: `NCPU=$(sysctl -n hw.ncpu)`.
3. For each variant, run the multi-threaded reader script `./run-parallel.sh <N>` with `N` equal to the number of CPUs:

```
cd v1-atomic && sudo kldload ./perfdemo.ko && ./run-parallel.sh $NCPU
sudo kldunload perfdemo
```

```
cd ../v2-aligned && sudo kldload ./perfdemo.ko && ./run-parallel.sh $NCPU
sudo kldunload perfdemo
```

```
cd ../v3-counter9 && sudo kldload ./perfdemo.ko && ./run-parallel.sh $NCPU
sudo kldunload perfdemo
```

The script spawns `N` reader threads, each pinned to a different CPU, each issuing reads as fast as it can for a fixed duration. It prints the aggregate throughput at the end.

4. Record the throughput (ops/sec) for each variant.
5. Now repeat with half and double the thread count:

```
./run-parallel.sh $((NCPU / 2))
./run-parallel.sh $((NCPU * 2))
```

Compare the three throughputs at each thread count. Half-CPU runs usually show less contention; double-CPU runs show heavy contention and scheduler effects.

6. For the clearest demonstration, run a DTrace script on each variant while the workload runs:

```
sudo dtrace -n 'lockstat:::adaptive-block /
 ((struct lock_object *)arg0)->lo_name == "perfdemo_mtx" / {
 @ = count();
 }'
```

v1-atomic should show significant blocking; v3-counter9 should show almost none.

**What you should see:**

- `v1-atomic` is the baseline. Its throughput does not scale well beyond about half the CPUs; the atomic counter's cache-line bouncing becomes a bottleneck.
- `v2-aligned` improves if the counters were previously packed with other hot fields on the same cache line. If the original softc had the atomic counter on its own line already, alignment is a no-op.
- `v3-counter9` scales near-linearly to the CPU count. Each CPU updates its own per-CPU copy; no cache-line bouncing on the counter.

**Expected results (for comparison):**

On an 8-core Xeon with 8 threads running this workload:

- `v1-atomic`: around 600K ops/sec aggregate.
- `v2-aligned`: around 680K ops/sec aggregate.
- `v3-counter9`: around 2.0M ops/sec aggregate.

At 16 threads (oversubscribed):

- `v1-atomic`: around 550K (starts dropping).
- `v2-aligned`: around 650K (also drops).
- `v3-counter9`: around 1.8M (held up by scheduler overhead, not contention).

The per-CPU counter scales; the atomics do not.

**What you should record in your logbook:**

- Throughput numbers for each variant and each thread count.
- The lock-block count for each variant from the DTrace script.
- A short note on what the numbers tell you about your hardware (how contended the CPU-interconnect is, whether hyper-threading helps or hurts).

**Common mistakes:**

- Running the parallel script with only one thread. The point of the experiment is contention; a single-thread run eliminates it and hides the effect.
- Concluding from a small difference. If the three numbers differ by less than 10%, the measurement noise could explain the difference; rerun at least three times and check variance.
- Forgetting that `UMA_ALIGN_CACHE` in UMA zones does similar work for you. The counter (9) primitive is a clean solution; you rarely need to hand-align for counters.
- Expecting linear scaling from `v1-atomic`. Atomic counters *cannot* scale past about the memory system's coherence bandwidth; once that bandwidth is saturated, adding CPUs makes throughput worse, not better.
- Forgetting to unload between runs. Loading a new variant on top of a running one does not replace the running one; `kldunload perfdemo` then `kldload ./perfdemo.ko`.

**Troubleshooting:** If the throughput numbers are inconsistent between runs, CPU frequency scaling is probably the culprit. Pin the frequency with `sysctl dev.cpu0.freq=$(sysctl -n dev.cpu0.freq_levels | awk '{print $1}' | cut -d/ -f1)` (adjust for your CPU). On systems where this is not available, disable dynamic frequency scaling in the BIOS/firmware.

### 33.13.5 Lab 5: Interrupt and Taskqueue Split

**Objective:** Compare in-interrupt-context work with taskqueue-deferred work, including behaviour under load.

**Directory:** `examples/part-07/ch33-performance/lab05-interrupt-tq/`.

**Prerequisites:** Lab 1 complete.

**Steps:**

1. Inspect the two variants in the lab directory:
  - `v1-in-callout`: a callout fires every millisecond and does its processing inline. The “interrupt handler” is the callout’s function; all work happens in that context.
  - `v2-taskqueue`: the callout enqueues a task; a taskqueue thread dequeues and processes the work. The callout itself does nothing else.
2. Build each: `make` in each subdirectory.
3. Load the first variant: `cd v1-in-callout && sudo kldload ./perfdemo.ko`.
4. Run the measurement script: `./measure-latency.sh`. It spawns a userland reader that waits for the callout’s processed data, then records the wall-clock time between the callout firing and the reader seeing the result. The script prints the median and P99 latency over 10,000 iterations.
5. Unload and load the second variant: `sudo kldunload perfdemo && cd ../v2-taskqueue && sudo kldload ./perfdemo.ko`.
6. Run the same measurement script.
7. Record median and P99 latency for each variant at idle.
8. Now repeat with artificial load on the system. In a second shell, start a CPU-intensive workload: `make -j$(sysctl -n hw.ncpu) buildworld` (if you have the world source checked out) or a simpler stressor: `for i in $(seq 1 $(sysctl -n hw.ncpu)); do yes > /dev/null & done`.
9. Rerun the measurement for each variant. The scheduler is now busy; the two variants should differ more sharply.
10. Stop the artificial load (`killall yes` or equivalent). Check that CPU usage returns to idle before declaring the test complete.

**What you should see:**

- `v1-in-callout` at idle has lower median latency but spikes when the callout’s context is preempted (which happens at taskqueue boundary).
- `v2-taskqueue` at idle has a few microseconds of added latency (the callout-to-taskqueue dispatch) but smoother behaviour.
- Under load, `v1-in-callout` latency becomes highly variable; the callout context competes with the stressor for CPU.
- Under load, `v2-taskqueue` latency is higher but smoother; the taskqueue thread’s scheduling class is more stable.

**Expected results (for comparison):**

On an 8-core Xeon:

- v1-in-callout at idle: median 8 us, P99 30 us.
- v2-taskqueue at idle: median 12 us, P99 40 us.
- v1-in-callout under load: median 15 us, P99 2000 us (spikes due to preemption).
- v2-taskqueue under load: median 20 us, P99 250 us (smoother).

The P99 improvement under load is the main reason production drivers prefer taskqueues for any significant work.

**What you should record:**

- Median and P99 latency at idle for each variant.
- Median and P99 latency under load for each variant.
- A short note on when you would use each in practice.

**Common mistakes:**

- Benchmarking only at idle. The two variants behave most differently under system load; test both states.
- Confusing callouts with real interrupts. A callout is a software timer; the lab uses it to stand in for an interrupt because real interrupt timing is hardware-specific. The conclusions transfer, but the absolute numbers depend on your scheduler.
- Forgetting to stop the stressor before declaring the test complete. Leaving yes running is rude to the next test.
- Drawing conclusions from a single measurement. Latency is distributional; a single run is a single point, not a distribution. Collect at least 10,000 measurements per configuration.

### 33.13.6 Lab 6: Ship v2.3-optimized

**Objective:** Apply the full tuning work to produce a finished `perfdemo` v2.3-optimized, complete with `sysctl` tree, manual page, and benchmark report.

**Directory:** `examples/part-07/ch33-performance/lab06-v23-final/`.

**Prerequisites:** Labs 1 through 5 complete. You should have the baseline numbers from Lab 1 and familiarity with the per-CPU counter work from Lab 4.

**Steps:**

1. Inspect the starting state of the driver in the lab directory. This is the baseline `perfdemo` with no tuning applied; comments throughout the source mark where each tuning pass will go.
2. Apply the three changes we worked through in the chapter:

**Change 1: counter(9) for all statistics.** Replace the atomic counters with `counter_u64_t`. Update the `sysctl` handlers to use procedural `sysctls` that call `counter_u64_fetch`. Mark each change with a short comment describing what was changed.

**Change 2: Cache-line alignment on hot softc fields.** Identify the fields in the `softc` that are written frequently from multiple CPUs. For each, add `__aligned(CACHE_LINE_SIZE)` and appropriate padding.

Note: if you use counter(9) for all counters, most of the cache-alignment work is already done; only non-counter hot fields (like pool pointers or state flags) need manual alignment.

**Change 3: Pre-allocated buffers.** Create a UMA zone at MOD\_LOAD time with UMA\_ALIGN\_CACHE. Use `uma_zalloc/uma_zfree` on the hot path instead of `malloc/free`. Destroy the zone at MOD\_UNLOAD.

3. Build and load: `make && sudo kldload ./perfdemo.ko`. Run a quick smoke test with `./run-workload.sh 10000` to confirm the driver works.
4. Clean up any PERF: markers from the source. Grep for them: `grep -n 'PERF:' perfdemo.c`. Each line should either be removed or the marker replaced with a permanent comment if the measurement is staying.
5. Update the `MODULE_VERSION()` macro to 3. Find the line and change:

```
MODULE_VERSION(perfdemo, 2); /* before */
MODULE_VERSION(perfdemo, 3); /* after */
```

6. Update the driver's manual page. If one does not exist, copy `/usr/src/share/man/man4/null.4` as a template and adapt it. The page should have:

- `.Nm perfdemo`
- A one-line description in `.Nd`
- A `.Sh DESCRIPTION` explaining what the driver does.
- A `.Sh TUNABLES` section documenting the sysctl knobs.
- A `.Sh SEE ALSO` referencing related pages.

Lint the page: `mandoc -Tlint perfdemo.4`. Render it: `mandoc -Tascii perfdemo.4 | less`.

7. Run the final benchmark: `./final-benchmark.sh`. The script exercises the driver across several workloads (single-thread sequential, single-thread random, multi-thread parallel) and records median and P99 latency plus throughput for each. Copy the output to your logbook.
8. Write the performance report. Use the template from Section 8 as a starting point. Fill in:
  - The “Before” numbers from Lab 1.
  - The “After” numbers from step 7.
  - The three changes applied.
  - The machine details.
  - The tunables introduced (if any).

Save the report as `PERFORMANCE.md` in the lab directory.

9. Unload the driver: `sudo kldunload perfdemo`. Confirm it unloaded cleanly with no hung threads or leaked memory (`vmstat -z | grep perfdemo` should show zero allocated, or the zone should not appear if it was destroyed).

### What you should produce:

- `perfdemo.c` with the three tuning changes applied and no PERF: markers.
- `perfdemo.4` (manual page) with a TUNABLES section.
- `PERFORMANCE.md` (benchmark report).
- Clean build, clean load, clean unload.

**Expected results (for comparison):**

The v2.3 driver on an 8-core Xeon:

- Single-thread sequential reads: median 9 us, P99 60 us, throughput 400K ops/sec.
- Multi-thread (8-thread) parallel reads: median 11 us, P99 85 us, throughput 2.8M ops/sec.

Compare to the baseline (roughly 30 us median, 330K ops/sec, 600K under 8 threads). The v2.3 is 3x on single-thread and nearly 5x on parallel.

**Common mistakes:**

- Skipping the cleanup. A driver with `PERF:` markers is not a shipped driver.
- Reporting benchmark numbers without context. Every benchmark number needs the environment it was produced in.
- Updating the version number without the matching changelog entry. The version change signals behaviour change; the changelog documents what changed.
- Writing the manual page in vague prose. The `TUNABLES` section in particular should name exact value ranges, defaults, and trade-offs.
- Not running `mandoc -Tlint`. A broken manual page is as bad as no manual page.

**33.13.7 Wrapping Up the Labs**

The labs work through one complete tuning cycle: baseline, DTrace, PMC, memory, interrupts, ship. By the time you finish Lab 6, you will have exercised every tool in the chapter at least once and have a concrete `perfdemo` artefact you can carry forward. If you have time, the challenge exercises below extend the driver in directions that are too specific for the main text but that teach the habits the chapter pointed at.

**33.13.8 A Lab Summary Table**

For quick reference, the labs at a glance:

| Lab | What it teaches              | Primary tool            | Output                           |
|-----|------------------------------|-------------------------|----------------------------------|
| 1   | Baseline measurement         | <code>sysctl(8)</code>  | Ops/sec number                   |
| 2   | Function-level profiling     | <code>dtrace(1)</code>  | Latency histogram, hot functions |
| 3   | Hardware-level profiling     | <code>pmcstat(8)</code> | IPC, PMC-hot functions           |
| 4   | Counter-contention tuning    | <code>counter(9)</code> | Scaling comparison               |
| 5   | Interrupt-context trade-offs | <code>taskqueue</code>  | Latency under load               |

| Lab | What it teaches | Primary tool                | Output              |
|-----|-----------------|-----------------------------|---------------------|
| 6   | Ship discipline | MODULE_VERSION, manual page | A clean v2.3 driver |

Each lab is scoped to one hour if you are practised with the tools, two hours if you are learning them for the first time. The full set is a solid afternoon's work; the challenge exercises extend it into a weekend's work for readers who want deeper practice.

### 33.13.9 Lab Dependencies

The labs are sequential. Lab 1 establishes the baseline numbers. Lab 2 uses those numbers as the before-and-after comparison. Lab 3 is optional (hwpmc may not be available), but its concepts flow into Lab 4. Lab 4 is where the main memory-tuning lesson is. Lab 5 is independent of the others but uses the same `perfdemo` driver. Lab 6 assumes you have absorbed Labs 1-5 and are ready to apply the full sequence.

If you are short on time, do Labs 1, 2, 4, and 6 in order. Those four are the core of the chapter.

## 33.14 Challenge Exercises

These exercises go beyond the main labs. Each is optional and can be done in any order. They are designed to stretch the ideas from the chapter in ways that generalise to real driver work.

### 33.14.1 Challenge 1: Latency Budget for a Real Driver

Pick a real FreeBSD driver that interests you. Good candidates include `/usr/src/sys/dev/e1000/` for the Intel e1000 family of Ethernet drivers (including `em` and `igb`), `/usr/src/sys/dev/nvme/` for the NVMe storage driver, `/usr/src/sys/dev/sound/pcm/` for the audio core, or `/usr/src/sys/dev/virtio/block/` for the virtio block device.

Read the driver's source and write a latency budget. For each phase of its hot path (interrupt filter, `ithread`, taskqueue handler, `softclock` callout, userland wakeup), estimate the maximum latency the driver can afford and still meet its workload's needs. A reasonable budget entry looks like:

| phase                         | target  | justification                              |
|-------------------------------|---------|--------------------------------------------|
| -----                         | -----   | -----                                      |
| PCIe read of status           | < 500ns | register read on local bus                 |
| interrupt filter              | < 2us   | must not steal <code>ithread</code> time   |
| <code>ithread</code> hand-off | < 10us  | <code>softclock</code> expects prompt wake |
| taskqueue deferred            | < 100us | reasonable for RX processing               |
| wakeup to userland            | < 50us  | scheduler-dependent                        |

Then audit the actual code: does the code on each phase justify your estimate? If the `ithread` handler takes a `MTX_DEF` sleep mutex that might block, is your budget honest?



The exercise is not about precision; the numbers above are illustrative. It is about looking at a real driver with performance eyes and reasoning about how its design fits a workload. Record your budget, your audit, and any discrepancies in your logbook. This habit, applied over many drivers, builds intuition faster than any other practice.

**Hint:** Look for explicit comments about timing in the source. Real drivers occasionally have comments of the form `/* this must complete within N us because ... */`. Those comments are a gift. Read them.

### 33.14.2 Challenge 2: Compare Three Allocators

For a driver hot path that allocates fixed-size buffers (say, 64-byte buffers), benchmark three implementations:

1. `malloc(9)` with a per-call `malloc(M_TEMP, M_WAITOK)` followed by `free(M_TEMP)`.
2. A UMA zone created with `uma_zcreate("myzone", 64, ..., UMA_ALIGN_CACHE, 0)`, with `uma_zalloc()/uma_zfree()` on the hot path.
3. A pre-allocated array of 1024 buffers plus a freelist, allocated once at attach, reused on the hot path with a simple atomic pop/push.

Wire each into a clone of `perfdemo` and run the same read workload against all three. Measure:

- Throughput (reads/second).
- Median and P99 latency (from a counter-backed histogram).
- Cache miss rate (from `pmcstat -S LLC_MISSES` or the DTrace profile provider's `LLC_MISSES` alias if available).

On a modern amd64 machine, you should see a meaningful step from 1 to 2 (roughly 1.5x to 3x throughput, depending on contention), and a smaller but real step from 2 to 3 (perhaps 10% to 30%). The shape of the histogram is often more interesting than the average: `malloc(9)` has a long tail driven by VM pressure, UMA is tighter, the preallocated ring has the tightest tail.

Write a short note explaining the trade-offs. The preallocated ring is the fastest but the most rigid (the buffer count is fixed at attach). UMA is the right default for variable workloads. `malloc(9)` is fine for rare, large, or non-hot-path allocations. Do not conclude from this exercise that `malloc(9)` is bad; it is often the correct choice for control paths.

### 33.14.3 Challenge 3: Write a Useful DTrace Script

Write a DTrace script that answers a question you actually have about a driver. Some examples:

- *How much time does my driver spend waiting on its own mutex?* Use `lockstat:::adaptive-spin` and `lockstat:::adaptive-block` filtered by the lock address.
- *What is the distribution of request sizes coming into my driver?* Capture `args[0]->uio_resid` from `fbt:myfs:myfs_read:entry` into a `quantize()` aggregation.
- *Which userland process is the heaviest consumer of my driver?* Aggregate `@[execname] = count()` on the driver's entry point.
- *What is the 99th percentile latency of the write path, broken out by request size?* Use a two-dimensional aggregation `@[bucket(size)] = quantize(latency)` where `bucket()` collapses sizes into a handful of ranges.

- *Is my driver ever called at interrupt context?* Use `self->in_intr = curthread->td_intr_nesting_level` and aggregate by that flag.

Save the script as a `.d` file in your lab directory. Annotate it with a header comment that names:

- The script's purpose (one sentence).
- The probes it uses.
- The invocation command, including any arguments.
- Sample output and how to interpret it.

A DTrace script that has the answer to a question a colleague asks is more valuable than almost anything else in an engineer's notebook. Keep a growing folder of such scripts over the years; they compound.

**Hint:** Before committing to a complex aggregation, run the probe with a `printf()` on a handful of events and confirm the variables you want to capture are actually available and contain the values you expect. Add the aggregation only once the raw probe is known to fire correctly.

### 33.14.4 Challenge 4: Instrument a Read Path Without Contaminating It

Start with a `perfdemo`-style driver where the read path runs in a tight loop at high throughput. Add enough instrumentation to answer two questions:

1. What is the P50 and P99 latency of this read path?
2. What is the distribution of request sizes?

Measure the throughput before and after instrumentation. Aim for under 5% throughput loss from instrumentation. The exercise rewards thinking about the cost of each probe. Rough cost hierarchy on modern amd64, from cheapest to most expensive:

- Static SDT probe at compile time with DTrace disabled: effectively zero cost.
- A counter(9) increment on a per-CPU slot: a few cycles.
- An `atomic_add_int()` on a shared counter: tens to hundreds of cycles under contention.
- A `sbinuptime()` call: tens of cycles.
- A `log(9)` call: hundreds to thousands of cycles plus potential lock contention.
- A `printf()` call: thousands of cycles plus a possible sleep.

Design the instrumentation accordingly. A reasonable shape: per-CPU counter(9) for counts, `DPCPU_DEFINE` histogram buckets for latency distribution, no per-call `printf()`, no shared atomics. Measure the before-and-after throughput carefully and write up what you found.

**Hint:** Compile with `-O2` and inspect the generated assembly for the hot function with `objdump -d perfdemo.ko | sed -n '/<perfdemo_read>/,/^$/p'` to confirm the instrumentation did not defeat compiler optimisations, such as pushing variables to the stack that the non-instrumented version kept in registers.

### 33.14.5 Challenge 5: Expose a Live Histogram

Extend the `perfdemo` sysctl tree with a latency histogram. Define a set of latency buckets (for example, 0 to 1us, 1 to 10us, 10 to 100us, 100us to 1ms, 1ms to 10ms, 10ms and above). For each read, measure the latency,

find its bucket, and increment a per-CPU counter for that bucket. Expose the whole bucket array as a single sysctl that returns an opaque structure.

A reasonable handler:

```
static int
perfdemo_sysctl_hist(SYSCTL_HANDLER_ARGS)
{
 uint64_t snapshot[PD_HIST_BUCKETS];
 for (int i = 0; i < PD_HIST_BUCKETS; i++)
 snapshot[i] = counter_u64_fetch(perfdemo_read_hist[i]);
 return (SYSCTL_OUT(req, snapshot, sizeof(snapshot)));
}
```

From userland, write a Python or shell script that polls the sysctl every second with `sysctl -b` and prints a live textual bar chart of the distribution. Compare the live distribution during a steady workload, during a bursty workload, and during an idle period.

The exercise demonstrates the full loop: in-kernel per-CPU counters, aggregation at read time, a binary sysctl, userland visualisation, and a continuous observation pipeline. This is the pattern real drivers use to ship latency telemetry.

**Hint:** `sysctl -b` reads the raw bytes; your userland code must unpack them using `struct.unpack` in Python or a small C program. Do not expose the buckets as a formatted string from the kernel; let the raw numbers flow out and let userland do the presentation.

### 33.14.6 Challenge 6: Tune for Two Workloads

Pick two realistic workloads for the driver. For example, *small, latency-sensitive reads at 100 per second* (latency-bound) and *large, throughput-sensitive reads at 1000 per second* (throughput-bound). Identify the driver settings that optimise each workload. Candidates include:

- Buffer size (small for latency, large for throughput).
- Coalescing threshold, if any (none for latency, yes for throughput).
- Batch size for batched operations (small for latency, large for throughput).
- Polling versus interrupts (interrupts for latency, polling for throughput).
- Debug logging level (off for both; leave on only for troubleshooting).

Write a `loader.conf`-style table showing the best settings for each workload. Explain the trade-offs: why the latency setting would hurt throughput, why the throughput setting would hurt latency, and what the cost would be of picking a middle setting.

This challenge makes concrete the point from Section 1 that one driver can be fast in different ways for different workloads. Real driver tuning often means exposing these knobs with sensible defaults and documenting the workload each corresponds to.

**Hint:** Do not guess the best settings; measure them. Run each workload against each setting and fill in a table. The winning setting is often not the one you would have guessed, especially for throughput-versus-latency trade-offs where caches and scheduler interactions surprise experienced engineers.

### 33.14.7 Challenge 7: Write the Manual Page

If `perfdemo` does not have a manual page, write one. Follow the convention of `/usr/src/share/man/man4/`: copy a simple existing page like `null(4)` or `mem(4)`, change the name, document the device, the `sysctl` knobs, the tunables, and any relevant errors.

A reasonable skeleton:

```
.Dd March 1, 2026
.Dt PERFDEMO 4
.Os
.Sh NAME
.Nm perfdemo
.Nd performance demonstration pseudo-device
.Sh SYNOPSIS
To load the driver as a module at boot time, place the following line in
.Xr loader.conf 5 :
.Pp
.Dl perfdemo_load="YES"
.Sh DESCRIPTION
The
.Nm
driver is a pseudo-device used to illustrate performance measurement and
tuning techniques in FreeBSD device drivers.
...
.Sh SYSCTL VARIABLES
.Bl -tag -width "hw.perfdemo.bytes"
.It Va hw.perfdemo.reads
Total number of reads served since load.
...
.El
.Sh SEE ALSO
.Xr sysctl 8 ,
.Xr dtrace 1
.Sh HISTORY
The
.Nm
driver first appeared as a companion to
.Em FreeBSD Device Drivers: From First Steps to Kernel Mastery .
```

Verify the formatting with `mandoc -Tlint perfdemo.4`. Render it with `mandoc -Tascii perfdemo.4 | less`.

A manual page is a small but real commitment to the rest of the system. It turns a private tool into a shared one. It also forces you to name and describe every knob, which occasionally reveals knobs that do not need to exist.

**Hint:** The mdoc language is terse but strict. Read mdoc(7) once and use a recent simple man page as a template. Do not invent mdoc macros; use only the ones documented.

### 33.14.8 Challenge 8: Profile an Unfamiliar Driver

Pick a FreeBSD driver you have not worked with. On a test system where the hardware is present (or in a VM with a compatible emulated device), run `pmcstat` or a DTrace profile during a realistic workload. Good VM-friendly targets:

- `virtio_blk` under a load generated by `dd if=/dev/vtbd0 of=/dev/null bs=1M count=1024`.
- `virtio_net` under a load generated by `iperf3`.
- `xhci(4)` if you have USB devices attached.
- `urtw(4)` or similar wireless drivers if you have the hardware.

Identify the top three functions by CPU time. For each, read the source and form a hypothesis about what the function is doing that takes the time. Record the hypothesis in your logbook. For example: *virtio\_blk\_enqueue spends most of its time in bus\_dmamap\_load\_ccb because each request copies into a bounce buffer*.

You do not have to verify the hypothesis. The exercise is about the habit of looking at an unfamiliar driver with measurement tools and reasoning about the data they produce. The hypothesis, right or wrong, sharpens the next read of the source.

**Hint:** If the top function is in common kernel code (`copyin`, `bcopy`, `memset`, `spinlocks`), that is still useful information: it tells you where the driver spends its time, even if the root cause is outside the driver itself. Follow the callers back to the driver function and form the hypothesis there.

### 33.14.9 Wrapping Up the Challenges

The challenges extend the chapter's ideas in directions the main text could not. If you have time to do one or two, they compound everything the chapter taught. If you have time for none of them, that is fine: the main labs give you the essential experience.

A useful selection strategy for limited time: pick one challenge from the *reading* family (Challenge 1 or 8), one from the *instrumentation* family (Challenge 3, 4, or 5), and one from the *shipping* family (Challenge 6 or 7). That mix exercises the three skills the chapter builds: reading existing drivers with performance eyes, instrumenting new drivers cleanly, and making finished drivers ready for others.

## 33.15 Performance Patterns Reference

The chapter covered a large surface. This section consolidates the practical patterns into a compact reference you can scan when working on a real driver. Each pattern names the situation it applies to, the technique, and the FreeBSD primitive to reach for.

### 33.15.1 Pattern: You Need a Hot-Path Counter

**Situation:** You want to count an event that happens thousands or millions of times per second, and your measurement must not dominate the cost of the event.

**Technique:** Use `counter(9)`. Declare a `counter_u64_t`, allocate it in attach with `counter_u64_alloc(M_WAITOK)`, increment on the hot path with `counter_u64_add(c, 1)`, sum on read with `counter_u64_fetch(c)`, free in detach with `counter_u64_free(c)`.

**Primitive path:** `/usr/src/sys/sys/counter.h`, `/usr/src/sys/kern/subr_counter.c`.

**Why this pattern:** `counter(9)` uses per-CPU slots updated without atomics. Read combines them. The increment is a handful of cycles with no cache-line contention.

**Anti-pattern:** An `atomic_add_64()` on a shared `uint64_t`. Each increment round-trips the cache line; at high rates, the counter becomes the bottleneck.

### 33.15.2 Pattern: You Need a Hot-Path Allocation

**Situation:** You want to allocate a fixed-size buffer many times per second, and `malloc(9)` is showing up in profiles.

**Technique:** Create a UMA zone at attach with `uma_zcreate("mydrv_buf", sizeof(struct my_buf), NULL, NULL, NULL, NULL, UMA_ALIGN_CACHE, 0)`. Allocate on the hot path with `uma_zalloc(zone, M_NOWAIT)`. Free with `uma_zfree(zone, buf)`. Destroy in detach with `uma_zdestroy(zone)`.

**Primitive path:** `/usr/src/sys/vm/uma.h`, `/usr/src/sys/vm/uma_core.c`.

**Why this pattern:** UMA gives each CPU a cache of free buffers. Allocate and free on the same CPU is typically a pop and push from a per-CPU stack, with no cross-CPU coordination. `UMA_ALIGN_CACHE` prevents false sharing between adjacent buffers.

**Anti-pattern:** A per-request `malloc(M_DEVBUF, M_WAITOK)` in the hot path. The general-purpose allocator has more overhead and less locality than a dedicated zone.

### 33.15.3 Pattern: You Need a Pre-Allocated Pool

**Situation:** You have a fixed known count of working buffers (for example, a ring descriptor count tied to hardware), and you want zero allocator cost on the hot path.

**Technique:** At attach, `malloc()` the array once. Keep the free list as a simple SLIST or stack. Pop and push from the free list under a single lock (or an atomic pointer if lock-free design is warranted). Free the whole array at detach.

**Why this pattern:** A pre-allocated pool has the lowest possible per-request cost and the most predictable latency. The rigidity (fixed count) is the trade.

**Anti-pattern:** Using `malloc()` or UMA when the count is known and fixed. The extra indirection costs cycles with no benefit.

### 33.15.4 Pattern: You Need to Avoid False Sharing

**Situation:** Two frequently-updated variables live in the same cache line and are written by different CPUs. Each write invalidates the other CPU's copy, causing a ping-pong.

**Technique:** Put each hot variable on its own cache line with `__aligned(CACHE_LINE_SIZE)`. For per-CPU data, use `DPCPU_DEFINE` which automatically pads.

**Primitive path:** `/usr/src/sys/amd64/include/param.h` and its siblings under `/usr/src/sys/<arch>/include/param` for `CACHE_LINE_SIZE` (pulled in via `<sys/param.h>`), `/usr/src/sys/sys/pcpu.h` for `DPCPU` macros.

**Why this pattern:** Cache coherence protocols bounce whole cache lines; if two writers share a line, every write by one invalidates the other's cache. Separating the variables costs a small padding and eliminates the ping-pong.

**Anti-pattern:** Adding alignment without measuring. Many variables do not need alignment; adding it blindly wastes memory without improving performance.

### 33.15.5 Pattern: You Need to Measure Latency Without Polluting It

**Situation:** You want to measure the latency of a hot path without the measurement dominating the cost.

**Technique:** For a sampled measurement, use `DTrace's profile` provider at 997 Hz. For every-call measurement, capture `sbinuptime()` at entry and exit, subtract, and bucket into a per-CPU `counter(9)` histogram. Read the histogram via `sysctl`.

**Why this pattern:** `DTrace profile` is near-free when disabled. `sbinuptime()` is fast (tens of cycles). A per-CPU histogram bucket is a `counter(9)` increment. The aggregate is read from userland without touching the hot path.

**Anti-pattern:** `nanotime()` per call (it is much slower than `sbinuptime()`), or a `printf()` per call (ruinous), or a shared atomic per call (cache-line bottleneck).

### 33.15.6 Pattern: You Need to Expose Driver State

**Situation:** You want operators to be able to query and configure the driver at runtime.

**Technique:** Build a `sysctl(9)` tree under `hw.<drivername>`. Use `SYSTCL_U64` for simple counters, `SYSTCL_INT` for simple integers, and procedural `sysctls` (`SYSTCL_PROC`) for computed values or bulk data. Flag tunables with `CTLFLAG_TUN` so `loader.conf` can set them.

**Primitive path:** `/usr/src/sys/sys/sysctl.h`, `/usr/src/sys/kern/kern_sysctl.c`.

**Why this pattern:** `sysctl` is the canonical FreeBSD interface for driver observability and configuration. Operators know how to use `sysctl(8)`; log aggregators can scrape `sysctl` trees; monitoring scripts can poll.

**Anti-pattern:** Custom `ioctl`s for every knob, or environment-variable reading at runtime, or `printf()` that expects the user to watch the log.

### 33.15.7 Pattern: You Need to Split an Interrupt Handler

**Situation:** The interrupt handler does more than the small amount of work safe in interrupt context; it sometimes blocks or contends.

**Technique:** Register a filter handler (returns `FILTER_SCHEDULE_THREAD`) that does the minimum (acknowledge hardware, read status). Register an ithread handler that runs in thread context and does moderate work. For work that can wait or that takes real time, schedule on a taskqueue.

**Primitive path:** `/usr/src/sys/sys/bus.h`, `/usr/src/sys/kern/kern_intr.c`, `/usr/src/sys/kern/-subr_taskqueue.c`.

**Why this pattern:** The filter runs in hard interrupt context with the strictest constraints; the ithread runs at a specific priority; the taskqueue runs at a configurable priority on a worker thread. Each layer is suited to its work.

**Anti-pattern:** Doing all the work in the filter, or blocking on locks in the ithread, or running heavy work at interrupt priority where it delays other drivers.

### 33.15.8 Pattern: You Need to Rate-Limit a Log Message

**Situation:** An error condition can produce many log entries per second; you want one per N seconds or per M events to avoid flooding.

**Technique:** Use `ratecheck(9)`. Hold a `struct timeval` and an interval; call `ratecheck(&last, &interval)`. The call returns 1 when enough time has passed and updates the state; return 0 means *skip this message*.

**Primitive path:** `/usr/src/sys/sys/time.h` (declaration) and `/usr/src/sys/kern/kern_time.c` (implementation).

**Why this pattern:** `log(9)` is cheap per call but not free at thousands per second. Unbounded logging causes log-disk fill and can contend the log device. A rate-limiter gives visibility without flood.

**Anti-pattern:** Unconditional `printf()` in error paths, or a manual counter that drifts and leaks.

### 33.15.9 Pattern: You Need a Stable Operator Interface

**Situation:** You want operators to rely on specific `sysctl` names or counter structures that will not change across versions.

**Technique:** Document the stable interface in the manual page. Version the driver with `MODULE_VERSION()` so operators can detect which interface they have. When you must change an interface, keep the old one (or a minimal shim) for at least one version and announce the deprecation in the release notes.

**Primitive path:** Manual page conventions in `/usr/src/share/man/man4/`.

**Why this pattern:** Operators write scripts against `sysctl` names. Breaking those names breaks their scripts. A stable published interface is a commitment you can keep.

**Anti-pattern:** Renaming a `sysctl` casually, or deleting a counter that scripts depend on, or reshaping a procedural `sysctl`'s output without versioning.



### 33.15.10 Pattern: You Need to Ship After Tuning

**Situation:** You have finished a tuning round and want to leave the driver in a clean, shippable state.

**Technique:** Remove temporary measurement code that was not meant to stay. Keep production-grade counters and SDT probes. Update `MODULE_VERSION()`. Run a full benchmark and record the numbers. Update the manual page. Write a short performance report that explains what was tuned and why. Commit with a message that references the report.

**Why this pattern:** The code change is only part of the work. The report, the numbers, and the documentation are what make the change durable. Without them, the next maintainer has no way to know what was done or why.

**Anti-pattern:** Committing tuning changes without benchmarks or documentation. The next maintainer cannot tell what changed, cannot reproduce the measurement, and may undo the work by accident.

### 33.15.11 A Short Tool Reference Table

Below is a one-page summary of the tools introduced in this chapter, their typical use, and their source or documentation path.

| Tool                    | Use                                            | Where to learn more                                                                         |
|-------------------------|------------------------------------------------|---------------------------------------------------------------------------------------------|
| <code>sysctl(8)</code>  | Read and write kernel state variables          | <code>sysctl(8)</code> , <code>/usr/src/sbin/sysctl/</code>                                 |
| <code>top(1)</code>     | Process and system resource overview           | <code>top(1)</code>                                                                         |
| <code>systat(1)</code>  | Interactive system statistics display          | <code>systat(1)</code>                                                                      |
| <code>vmstat(8)</code>  | Virtual memory, interrupt, and zone statistics | <code>vmstat(8)</code>                                                                      |
| <code>ktrace(1)</code>  | Trace system calls for a process               | <code>ktrace(1)</code> , <code>kdump(1)</code>                                              |
| <code>dtrace(1)</code>  | Dynamic tracing of userland and kernel         | <code>dtrace(1)</code> , <code>dtrace(7)</code> , <code>fbt(7)</code> , <code>sdt(7)</code> |
| <code>pmcstat(8)</code> | Hardware performance counter reader            | <code>pmcstat(8)</code> , <code>hwpmc(4)</code>                                             |

| Tool                       | Use                        | Where to learn more                         |
|----------------------------|----------------------------|---------------------------------------------|
| <code>flamegraph.pl</code> | Stack-sample visualisation | external, from Brendan Gregg                |
| <code>gprof(1)</code>      | Profile user programs      | <code>gprof(1)</code> ; not used for kernel |
| <code>netstat(1)</code>    | Network statistics         | <code>netstat(1)</code>                     |

A driver maintainer does not need all of these every day. But knowing which one answers which question saves hours of blind investigation.

### 33.15.12 A Short Primitive Reference Table

The core in-kernel primitives used in this chapter, in one place.

| Primitive                            | Purpose                             | Header                                                      |
|--------------------------------------|-------------------------------------|-------------------------------------------------------------|
| <code>counter(9)</code>              | Per-CPU hot counters                | <code>sys/counter.h</code>                                  |
| <code>DPCPU_DEFINE_STATIC</code>     | Define per-CPU variable             | <code>sys/pcpu.h</code>                                     |
| <code>DPCPU_GET / DPCPU_PTR</code>   | Access current CPU's slot           | <code>sys/pcpu.h</code>                                     |
| <code>DPCPU_SUM</code>               | Sum across all CPUs                 | <code>sys/pcpu.h</code>                                     |
| <code>uma_zcreate</code>             | Create UMA zone                     | <code>vm/uma.h</code>                                       |
| <code>uma_zalloc / uma_zfree</code>  | Allocate from / free to zone        | <code>vm/uma.h</code>                                       |
| <code>UMA_ALIGN_CACHE</code>         | Cache-align zone items              | <code>vm/uma.h</code>                                       |
| <code>bus_alloc_resource_any</code>  | Allocate bus resource (irq, memory) | <code>sys/bus.h</code>                                      |
| <code>bus_setup_intr</code>          | Register interrupt handler          | <code>sys/bus.h</code>                                      |
| <code>bus_dma_tag_create</code>      | Create DMA tag                      | <code>sys/bus_dma.h</code>                                  |
| <code>taskqueue_enqueue</code>       | Queue deferred work                 | <code>sys/taskqueue.h</code>                                |
| <code>callout(9)</code>              | One-shot or periodic timer          | <code>sys/callout.h</code>                                  |
| <code>SYSCTL_U64 / SYSCTL_INT</code> | Declare sysctl                      | <code>sys/sysctl.h</code>                                   |
| <code>SYSCTL_PROC</code>             | Declare procedural sysctl           | <code>sys/sysctl.h</code>                                   |
| <code>log(9)</code>                  | Log a message                       | <code>sys/syslog.h</code>                                   |
| <code>ratecheck(9)</code>            | Rate-limit a message                | <code>sys/time.h</code>                                     |
| <code>SDT_PROBE_DEFINE</code>        | Declare static DTrace probe         | <code>sys/sdt.h</code>                                      |
| <code>CACHE_LINE_SIZE</code>         | Per-architecture cache-line size    | <code>sys/param.h</code> (via MD<br><arch>/include/param.h) |

| Primitive                  | Purpose               | Header                  |
|----------------------------|-----------------------|-------------------------|
| <code>sbinuptime()</code>  | Fast monotonic time   | <code>sys/time.h</code> |
| <code>getnanotime()</code> | Fast coarse wall time | <code>sys/time.h</code> |
| <code>nanotime()</code>    | Precise wall time     | <code>sys/time.h</code> |

The habit of knowing where to find these in a fresh source tree is worth as much as the knowledge of when to use them. Open the header, read the macro definitions, and see the usage sites with `grep -r SYSCTL_PROC /usr/src/sys/dev/ | head`.

### 33.15.13 Wrapping Up the Patterns Reference

The patterns above are not commandments; they are well-supported defaults. A specific driver may have a reason to deviate from any of them. What matters is that the deviation is conscious, measured, and documented. *We do not use UMA here because...* is fine; *we do not use UMA here, I guess* is not.

When you work on an unfamiliar driver, you can use this table as a checklist: does the driver count its hot events cheaply, allocate from a suitable zone, avoid false sharing on shared variables, split its interrupt work appropriately, expose state through `sysctl`, rate-limit its logging, and ship with a versioned stable interface? Each negative answer is a candidate for improvement if performance is under investigation.

## 33.16 Troubleshooting and Common Mistakes

Performance work has its own class of problems, distinct from functional bugs. A driver that measures wrong is as broken as a driver that crashes, but the symptoms are subtler. This section catalogs the most common failures, their symptoms, and their fixes.

### 33.16.1 DTrace Reports Zero Samples

**Symptom:** A DTrace script compiles, runs, and on Ctrl-C reports an empty aggregation or zero counts.

**Most common causes:**

- The driver module is not loaded. `fbt:mymodule: probes` do not exist until the module is in the kernel.
- The function name is wrong. Check with `dtrace -l | grep myfunction`; DTrace lowercase the probe names in the listing, but the function name inside the kernel is the C identifier.
- The function was inlined by the compiler. An `fbt` probe only exists for a function the compiler emits as a callable symbol. Static functions may be inlined and disappear; check with `objdump -t module.ko | grep myfunction`.
- The probe is on a path that was not exercised. Run the workload that should hit the path, not just any workload.
- The predicate filters everything out. Remove the predicate and confirm the probe fires; then tighten the predicate.

**Debugging sequence:** Check `kldstat` (is the module loaded?), `dtrace -l -n probe_name` (does the probe exist?), `dtrace -n 'probe_name { printf("fired"); }'` (does it fire at all?), then add your real logic.

### 33.16.2 DTrace Drops Events

**Symptom:** DTrace prints *dropped N events* or the aggregation counts are suspiciously low for a known-high-rate event.

**Most common causes:**

- The buffer is too small. Increase it: `dtrace -b 64m ...` for a 64 MB buffer.
- The aggregation is too chatty. Every `printf()` inside a probe is a potential drop; use aggregations for high-rate events, not per-event prints.
- The switch rate is too high. DTrace switches buffers between kernel and userland; at very high rates this can drop. Tune with `dtrace -x bufpolicy=fill` for fewer drops at the cost of missing the tail.

### 33.16.3 pmcstat Reports *No Device* or *No PMCs*

**Symptom:** `pmcstat` prints errors like `pmc_init: no device, ENXIO`, or similar.

**Most common causes:**

- `hwpmc` is not loaded. Run `kldload hwpmc`.
- The underlying CPU does not expose PMCs. Many VMs and some cloud instances have PMCs disabled. Fall back to DTrace's `profile` provider.
- The event name is unsupported on this CPU. Run `pmcstat -L` to see the available events.

### 33.16.4 pmcstat Sampling Misses Hot Functions

**Symptom:** A function you *know* is hot does not appear in `pmcstat -T` output.

**Most common causes:**

- The function is inline and has no callable symbol. The sampler attributes samples to the outer function.
- The sample rate is too low. Try `-S cycles -e 100000` for one sample per 100k cycles.
- The function is too fast for a single sample to catch it reliably. Cumulate samples over a long workload.
- The function is in a module whose symbols were stripped. Rebuild with debugging symbols.

### 33.16.5 Counter Values Look Impossible

**Symptom:** A counter has a negative value, wraps around unexpectedly, or reports numbers inconsistent with the workload.

**Most common causes:**

- A counter was updated from multiple CPUs without atomics. Use `atomic_add_64()` or `counter(9)`.
- A counter was read mid-update. For `counter(9)`, this is not a problem because the fetch sums per-CPU values; for a plain atomic, a read is always consistent but an arithmetic combination of two reads might not be. Snapshot the counters in a consistent order.

- The counter overflowed. A 32-bit counter at a million events per second overflows in about an hour. Use `uint64_t`.
- The counter was not initialised. `counter_u64_alloc()` returns a zeroed counter, but a plain variable is only zeroed at load. A static global is fine; a local is not.

### 33.16.6 sysctl Handler Crashes

**Symptom:** Reading or writing a sysctl causes a kernel panic.

**Most common causes:**

- The sysctl was registered against memory that has been freed (for example, a `softc` field after detach).
- The handler dereferences a NULL pointer. Check `req->newptr` before assuming a write.
- The handler runs without appropriate locks. Use `CTLFLAG_MPSAFE` and do the locking in the handler, or omit `CTLFLAG_MPSAFE` and accept the giant lock.
- The handler calls a function that cannot be called from sysctl context (for example, one that sleeps while holding a spin lock).

### 33.16.7 The Driver Slows Down Under Measurement

**Symptom:** The driver's throughput drops when DTrace or `pmcstat` is running.

**Most common causes:**

- The probes are in a hot path. Move them to less-frequent paths or reduce the probe count.
- The DTrace script is too chatty. Use aggregations, not per-event prints.
- The PMC sample rate is too high. Reduce it.
- The probe predicate is expensive. Simple predicates (`arg0 == 0`) are cheap; complex ones with string compares are not.

### 33.16.8 Taskqueue Thread Starvation

**Symptom:** Taskqueue work accumulates without being processed; the driver's user-visible response degrades.

**Most common causes:**

- The taskqueue's thread was pinned to a CPU that is now busy with something else. Use `taskqueue_start_threads_c` for predictable placement or leave the thread unpinned.
- The taskqueue is shared (`taskqueue_fast`) and other subsystems are keeping it busy. Use a dedicated taskqueue.
- A previous task is blocked waiting on something that never completes. Kill the task; add a timeout to the wait.
- The thread was killed by a `MOD_UNLOAD` but the tasks were not drained. Use `taskqueue_drain(9)` before releasing resources.

### 33.16.9 Interrupt Storm Detected

**Symptom:** `dmesg` shows *interrupt storm detected* and the interrupt is throttled.

**Most common causes:**

- The driver does not acknowledge the interrupt in the handler, so the hardware keeps asserting it.
- The driver acknowledges the wrong register.
- The hardware genuinely fires at a rate the driver cannot serve. Enable coalescing if supported.
- A shared-interrupt siblings is misbehaving, and the filter handler claims the interrupt falsely by returning `FILTER_HANDLED` when it should return `FILTER_STRAY`.

### 33.16.10 Cache-Line Alignment Has No Effect

**Symptom:** You added `__aligned(CACHE_LINE_SIZE)` to a counter and saw no change in throughput.

**Most common causes:**

- The counter is not actually contended. Single-threaded or low-concurrency workloads do not benefit from alignment.
- The counter was already aligned by the compiler's default layout. Confirm with `offsetof()`.
- The neighbouring field is no longer hot. The hot field changed, the old alignment is still there, and no contention exists.
- The CPU does not bounce cache lines hard enough for the difference to be measurable. Some architectures have cheaper cache coherence than others.

Alignment is not free (it pads the struct and wastes cache); if it has no effect, remove it. Measurement tells you.

### 33.16.11 UMA Zone Leaks

**Symptom:** `vmstat -z` shows a UMA zone growing without bound, and the system eventually runs out of kernel memory.

**Most common causes:**

- The driver allocates with `uma_zalloc()` but does not free with `uma_zfree()` in the corresponding path. Match the allocation and free paths.
- A pointer was overwritten before being freed. Inspect with KASAN or MEMGUARD (Chapter 34 material).
- A failed operation's cleanup path is wrong. Every early return from a function that allocates must free what was allocated.

### 33.16.12 The Benchmark Is Noisy

**Symptom:** Repeated runs of the same benchmark produce significantly different numbers.

**Most common causes:**

- System activity interferes. Run benchmarks on an otherwise idle machine.
- CPU frequency scaling is changing the clock rate between runs. Pin frequency with `sysctl dev.cpu.freq.0.freq_levels` (check the options) or set performance mode.
- Cache warm-up differs between runs. Discard the first run's numbers or add a warm-up pass.
- The kernel's lazy data structures (UMA caches, buffer caches) differ between runs. Drop caches between runs or accept the variance.
- Hyper-threading pairs are contending. Disable SMT for clean numbers.

A benchmark with high variance is not necessarily wrong, but its conclusions need to survive the variance. If the tuning change improves the median by 5% but the run-to-run variance is 20%, you cannot conclude anything from a single run; run many and look at the distribution.

### 33.16.13 You Measured the Wrong Thing

**Symptom:** You tuned the driver to meet a goal, but the user-visible performance did not change.

**Most common causes:**

- The metric you optimised is not the one that matters. Tuning median latency while the P99 is the user-visible problem is a classic miss.
- The workload you benchmarked is not the workload the user runs. Optimising for sequential reads when the user does random reads is wasted effort.
- The driver was not the bottleneck. Sometimes the bottleneck is in userland, in the scheduler, or in a different subsystem. `pmcstat -s cycles` on the whole system tells you where the time really goes.

If tuning did not help, that is not failure; it is information. The metric or workload was wrong, and now you know. Restate the goal and measure again.

### 33.16.14 Numbers Get Worse After a Tuning Change

**Symptom:** You applied what looked like a beneficial change (added a per-CPU cache, shortened a critical section, moved work to a taskqueue), and the measured numbers are worse.

**Most common causes:**

- The change introduced a regression that is not obvious from the code. For example, moving work to a taskqueue added a context switch on every request, and at your rate the switch cost dominates.
- The change shifted the bottleneck. You removed lock contention, and now the allocator is the new bottleneck.
- The change broke a previously-invisible correctness property. A softer race now produces extra work (retries, error paths) and the driver appears slower because it is doing more.
- The measurement is noisy and the run that looked worse is within the variance band. Run it ten times and see.

**Debugging sequence:** Revert the change and confirm the baseline reproduces. Apply the change and measure again. If both reproduce, the change is the cause; if the baseline varies, the measurement is unstable. Check the diff for unrelated changes that crept in. Profile both variants and compare: the profile will show where time actually goes.

### 33.16.15 Throughput Climbs but P99 Latency Gets Worse

**Symptom:** A change improved the average throughput but made the tail latency worse. Users complain, even though your benchmark's primary metric improved.

**Most common causes:**

- The change traded tail latency for throughput. For example, larger batching improves throughput but increases the time an unlucky request waits for its batch to complete.
- Queue depth grew. Longer queues give more opportunities for backlog, which extends the P99 under load.
- A slower path was taken more often. For example, the hot path got faster but the error path still pays the old cost, and the benchmark now exercises the error path more.

**Debugging sequence:** Plot throughput versus offered load; plot P99 latency versus offered load. The knee of the latency curve is where the driver enters overload. If the knee moved to the left (lower load produces high latency), the change made tail latency worse even if peak throughput improved. Tune back toward the latency target.

### 33.16.16 Results Differ Between Hardware Configurations

**Symptom:** A change that improves performance on one machine makes no difference or is worse on another.

**Most common causes:**

- CPU microarchitecture differs. Cache-coherence protocols, branch predictors, and memory subsystems vary between generations; optimisations tuned for one generation may be irrelevant or harmful for another.
- Core count differs. Per-CPU caches scale with CPU count; on a small-CPU system, the overhead may exceed the benefit.
- NUMA topology differs. An optimisation that ignores NUMA may be fine on a single-socket system and harmful on a multi-socket one.
- The memory subsystem differs. DRAM speed, memory channels, and cache sizes vary widely.

**Debugging sequence:** Record the full hardware configuration with each measurement (CPU model, core count, memory configuration). Measure on at least two distinct machines. Accept that some optimisations are machine-specific; document them if so.

### 33.16.17 KASSERT Fires Only Under Load

**Symptom:** A `KASSERT()` that has been silent during functional testing fires when the driver is under heavy load.

**Most common causes:**

- A race that is rare under low concurrency becomes frequent under high concurrency. The assertion catches a state that only occurs when two CPUs reach the critical section simultaneously.
- An ordering violation that depends on exact timing. Slower paths hid it; faster paths expose it.
- A resource exhaustion that only appears at high rates. For example, `M_NOWAIT` returning `NULL` because the allocator briefly runs out of memory.



- A driver-visible effect of kernel lazy evaluation (UMA cache rebalancing, callout backlog processing).

**Debugging sequence:** Capture the panic's stack trace and the values of local variables. Add `mtx_assert(&sc->mtx, MA_OWNED)` at key points to verify lock ownership. Add temporary SDT probes around the suspect region and trace the order of events with DTrace. Sometimes the fix is a tighter lock; sometimes it is a memory barrier; sometimes it is a redesign.

### 33.16.18 Measurement Disagrees With User Perception

**Symptom:** Your benchmark says the driver is fast; users say the driver feels slow.

**Most common causes:**

- The benchmark measures an aggregate; users perceive individual worst-case events. A driver with 10us median latency and a 1s P99.99 feels awful even though the benchmark score is excellent.
- The benchmark uses synthetic load; the real workload has dependencies (file access patterns, syscall chains) that the benchmark does not reproduce.
- Users perceive end-to-end latency, not driver latency. If the driver is 5% of the end-to-end time, even a large improvement produces no user-visible change.
- The user complaint is correct and the benchmark is measuring the wrong thing.

**Debugging sequence:** Ask users exactly what they do. Reproduce it. Measure the actual experience with a stopwatch or a user-space profiler if necessary. Add per-operation histograms and look at the tail percentiles. A P99.99 is not a statistical artifact; it is someone's bad day.

### 33.16.19 DTrace Script Hangs or Takes a Long Time to Exit

**Symptom:** After Ctrl-C, DTrace takes seconds or minutes to finish printing aggregations; sometimes it appears to hang.

**Most common causes:**

- The aggregation is very large (millions of keys). Printing takes time proportional to the aggregation size.
- The aggregation contains large strings. String allocation in DTrace is not free.
- The kernel buffer is full of queued events waiting to be delivered to userland. DTrace drains the buffer before exiting.
- A probe is still firing rapidly and DTrace is trying to keep up.

**Debugging sequence:** Cap the aggregation with `trunc(@, N)` or `printa()` with a limited list. Use `dtrace -x aggsz=M` to bound the aggregation size. Use `dtrace -x switchrate=1hz` to reduce switch overhead during the run. If you need to kill a runaway DTrace, `kill -9` is safe; DTrace releases probes cleanly.

### 33.16.20 Sysctl Reports Different Numbers on Successive Reads

**Symptom:** Reading the same sysctl twice in quick succession returns different values, even when no workload is running.

**Most common causes:**

- The counter is still being updated by ongoing operations. This is expected and usually fine.
- A per-CPU counter is being snapshotted non-atomically; different CPUs are read at different instants.
- The handler uses a non-stable data structure (a linked list under traversal) without holding the appropriate lock.
- A bug in the handler reads beyond the allocated buffer, returning stale stack data.

**Debugging sequence:** Stop the workload completely. Read the sysctl ten times. If the numbers still vary, the handler is buggy (most likely missing a lock or a barrier). If the numbers are stable when idle, the variance is from background activity; document the counter as *approximate* if appropriate.

### 33.16.21 A Driver Runs Fast Alone, Slow in Production

**Symptom:** Benchmarks in isolation show the driver meeting its goals; production performance is much worse.

**Most common causes:**

- Shared resources in production are contested. The filesystem buffer cache, the scheduler, the network stack, or other drivers compete for the same CPUs and memory.
- Production workload profile differs from the benchmark. The benchmark uses a uniform pattern; production uses a mix that exercises code paths the benchmark never touches.
- Production memory pressure invalidates caches. In benchmarks, your UMA zone stays warm; in production, memory pressure flushes it.
- Security features (IBRS, retpolines, kernel-page-table isolation) have different costs in production than in the benchmark's environment.

**Debugging sequence:** Instrument the driver so it is observable in production. Start DTrace or a lightweight profiler in production briefly and compare the distributions with the benchmark's. If the gap is reproducible, narrow down which subsystem differs; the answer is often memory pressure or scheduler differences rather than the driver itself.

### 33.16.22 Wrapping Up the Troubleshooting

Every item in this section is a real failure mode that experienced performance engineers have made. The value is in reading the list once, recognising the patterns when they appear, and spending the measurement time to diagnose the specific failure rather than guessing. *Why is my driver slow* has thousands of possible answers; the troubleshooting patterns narrow the search space in a few specific steps.

A habit worth cultivating: when you hit a new failure mode not on this list, write it down. Capture the symptom, the cause, and the debugging sequence. Over a few years, your own list becomes more useful than any book's, because it reflects the drivers and hardware you actually work with.

## 33.17 Wrapping Up

Chapter 33 took us from *does the driver work?* to *does it work well?*. The two questions sound similar and differ in the shape of the answer. The first is a yes/no of correctness; the second is a numerical distribution of

throughput, latency, responsiveness, and CPU cost, measured under a specific workload in a specific environment.

We began in Section 1 with the four axes of driver performance and the discipline of stating measurable goals. A goal with a metric, a target, a workload, and an environment is what separates a performance project from a performance hope. The reminder that optimisation must preserve correctness, debuggability, and maintainability is the constraint that keeps performance work honest.

Section 2 introduced the kernel's measurement primitives: the `get *` and non-`get` time functions, the three grades of counter (plain atomic, `counter(9)`, `DPCPU`), the `sysctl` machinery for exposing state, and the FreeBSD tools (`sysctl`, `top`, `systat`, `vmstat`, `ktrace`, `dtrace`, `pmcstat`) that read it. The central insight: measurement has cost, and the cost of measurement must not contaminate the measurement itself.

Section 3 introduced DTrace. Its pervasive probes, near-zero disabled cost, kernel-side aggregation, and expressive scripting language make it the tool of choice for most driver observation. The `fbt` provider covers every function; the `sdt` provider gives you stable named probes in your own driver; the `profile` provider samples at a fixed rate; the `lockstat` provider specialises in lock contention.

Section 4 introduced hardware performance counters. `hwpmc(4)` exposes the CPU's built-in counters; `pmcstat(8)` drives them from userland. Counting mode gives totals; sampling mode gives distributions. The callgraph collector plus flame-graph renderer turns the samples into a visualisation that makes hotspots obvious. The data is raw and interpretation takes practice, but the combination answers questions no other tool can.

Section 5 turned to memory. Cache-line alignment avoids false sharing; `bus_dma(9)` handles DMA alignment; pre-allocation keeps the hot path out of the allocator; UMA zones with `UMA_ALIGN_CACHE` specialise for fixed-size frequent allocations; per-CPU counters with `counter(9)` and `DPCPU` eliminate cache-line contention on shared metrics. Each technique has a cost, and each deserves evidence before it is applied.

Section 6 turned to interrupts and taskqueues. Measuring interrupt rate and handler latency is easy with `vmstat -i` and DTrace. The filter-plus-ithread split keeps the interrupt context small; taskqueues move work to a calmer environment; dedicated threads offer isolation for special cases; coalescing and debouncing control the interrupt rate itself. The judgement of when to apply each technique comes from measurement, not from reading.

Section 7 turned inward, to the runtime metrics that belong in the shipped driver. A thoughtful `sysctl` tree exposes totals, rates, states, and configuration. Procedural `sysctls` fetch `counter(9)` sums. Rate-limited logging via `log(9)` and `ratecheck(9)` keeps informational messages from flooding the log. Debug modes give operators a safe way to raise verbosity without rebuilding. Each element is a small commitment; the cumulative result is a driver that is auditable, diagnosable, and maintainable.

Section 8 closed the loop. Tuning produces scaffolding; shipping requires cleanup. Remove temporary measurement code. Keep the counters, SDT probes, and configuration knobs that have lasting value. Benchmark the final driver, document the tuning knobs in the manual page, update `MODULE_VERSION()`, and write a plain-text performance report. These steps are the difference between a one-off improvement and a lasting one.

The labs walked through six concrete stages: baseline counters, DTrace scripts, `pmcstat` sampling, cache-aligned and per-CPU tuning, the interrupt-vs-taskqueue split, and the v2.3-optimized finalisation. Each lab produced a measurable output; the numbers are what the chapter is ultimately about.

The challenge exercises extended the work in directions too specific for the main text: latency budgets for real drivers, allocator comparisons, useful DTrace scripts of your own, an observability-budget study, a live histogram, workload-specific tuning, manual-page writing, and profiling an unfamiliar driver.

The troubleshooting section catalogued the common failure modes and their diagnoses. Each one is a recognisable pattern; spending the measurement time to name which pattern you are in is the half of the work that most beginners skip.

### 33.17.1 What You Should Be Able to Do Now

If you worked through the labs and read the explanations carefully, you can now:

- Set measurable performance goals for a driver before you start tuning.
- Choose the right kernel time function for a measurement based on the precision you need.
- Count events cheaply on the hot path with `counter(9)` or `atomic_add_64()`.
- Expose driver metrics to userland through a sysctl tree.
- Write DTrace one-liners and scripts that answer specific questions about a driver.
- Measure lock contention with the `lockstat` provider.
- Sample kernel CPU time with `pmcstat` or with DTrace's `profile` provider.
- Read a flame graph.
- Apply cache-line alignment, pre-allocation, UMA zones, and per-CPU counters selectively, with evidence.
- Split a driver's interrupt work between filter, `ithread`, and `taskqueue` based on measured latency.
- Ship a tuned driver: remove scaffolding, document tunables, update the version, write a report.

### 33.17.2 What This Chapter Leaves Implicit

There are two topics the chapter touched on but did not deep-dive. Each has its own chapter or its own literature, and trying to cover it fully here would have bogged down the main thread.

First, **the deep internals of hwpmc (4)**. The chapter showed you how to use it; the source tree under `/usr/src/sys/dev/hwpmc/` shows you how it works, across Intel, AMD, ARM, RISC-V, and PowerPC backends. If you want to understand how the kernel samples the PC register on an overflow interrupt, or how the userland `pmcstat` talks to the kernel, reading that tree pays off. Most readers never need to.

Second, **advanced DTrace**. The scripts we wrote are straightforward. DTrace's speculations, its USDT interfaces, its complex aggregations, and its advanced predicates are worth a book of their own. Brendan Gregg's *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X, and FreeBSD* is the canonical reference. For driver work, the patterns we covered carry you a long way.

### 33.17.3 Key Takeaways

Distilled into a page, the chapter's core lessons are:

1. **Measure before you change.** An optimisation without a before-and-after number is a guess.
2. **State the goal in numbers.** Metric, target, workload, environment. Without these, you cannot know when you are done.
3. **Prefer the simplest tool that answers the question.** A DTrace one-liner usually beats a full `pmcstat` session.
4. **The tools that measure also perturb.** Know the cost of each probe before you use it on a hot path.

5. **Four axes, not one.** Throughput, latency, responsiveness, CPU cost. Optimising one can harm another.
6. **counter(9) for hot counters.** Per-CPU update, summed on read. Scales from one CPU to many.
7. **UMA\_ALIGN\_CACHE for hot allocations.** Cache-aligned per-CPU caches, cheap alloc and free.
8. **Pre-allocate, do not allocate in the hot path.** Attach-time cost is fine; per-call cost is not.
9. **Keep the interrupt handler small.** Filter-plus-ithread-plus-taskqueue is the standard split; use it.
10. **Ship with a sysctl tree.** A driver with no observability is a driver nobody can diagnose.
11. **Rate-limit your logs.** A driver that logs every error can DoS itself.
12. **Write the report.** The tuning work is not done until the numbers are written down where the next maintainer can find them.

### 33.17.4 Before You Move On

Before you treat this chapter as completed and move to Chapter 34, take a moment to verify the following. These are the checks that separate a learner who has seen the material from a learner who owns it.

- You can write, from memory, the skeleton of a procedural sysctl that returns a counter(9) sum.
- You can write a DTrace one-liner that histograms a function's latency.
- You can explain, in one paragraph, the difference between `getnanotime()` and `nanotime()`.
- You can name three measurements you would capture before changing a driver's hot path, and at least one tool for each.
- Given a driver that appears to be CPU-bound, you have a plan: `vmstat -i` to check for interrupt storms, `dtrace -n 'profile-997 { @[stack()] = count(); }'` to find the hot function, `pmcstat -S cycles -T` to confirm, and `lockstat` if the profile points at locks.
- Given a driver whose throughput does not scale with CPU count, you have a plan: check for a single atomic counter being contended, check for a shared taskqueue, check for a global lock.
- You know which sections of your manual page would name the tunables, and roughly how to write them.

If any of these still feel shaky, loop back to the relevant section and re-read it. The chapter's material compounds; Chapter 34 will assume you have internalised most of what is here.

### 33.17.5 A Note on Sustained Practice

Performance work, more than almost any other area of kernel engineering, rewards steady practice. The first few `pmcstat` sessions are confusing; the tenth is natural. The first DTrace script you write is laborious; the hundredth is a one-liner typed without thinking. The first performance report reads like a mystery; the tenth reads like a decision document.

If you have driver work in your day job or on a personal project, spend ten minutes of your week on a performance measurement. Not a tuning pass, just a measurement. Record the numbers. Compare them to last week. Over a year this compounds into the habits this chapter is trying to teach. The tools are all there; the habit is what takes time.

### 33.17.6 Connecting Back to the Rest of the Book

Chapter 33 rests on material from earlier parts. A short tour of what you now see in a new light:

From Part 1 and Part 2, the module lifecycle and `DRIVER_MODULE()` registration. The driver you tuned in this chapter is the same shape of driver you built in those parts; tuning does not change the skeleton.

From Part 3, the Newbus framework. `simplebus`, the bus-resource APIs, `bus_alloc_resource_any()`: these are the building blocks your driver uses to find its hardware. Knowing them lets you reason about where time goes during attach and detach.

From Part 4, the driver-to-userspace interfaces. Your `read()` and `write()` paths are the ones you tuned in this chapter. The `sysctl` tree you built in Section 7 is a natural extension of the `ioctl` and `sysctl` material from Chapter 15.

From Part 5 and Part 6, the testing and debugging chapters. Those tools apply to tuned drivers in full. A driver that performs well but crashes under load is not a tuned driver; it is a broken one.

From Part 7, Chapter 32 on Device Tree and embedded development is the platform half. Performance on an embedded board has the same shape as performance on a server, but the constraints are tighter: less RAM, less CPU, less power. The techniques scale down in priority (cache-line alignment matters less when you have one core), but the measurement discipline matters as much or more.

The picture after Chapter 33 is that you have the full measurement-and-tuning toolkit for FreeBSD drivers. You can build drivers. You can test them. You can port them. You can measure them. You can tune them. You can ship them. The remaining chapters of Part 7 refine and finalise those skills, and the next one is about the debugging techniques that the performance work sometimes reveals are still needed.

### 33.17.7 Looking Ahead

Chapter 34 turns from performance to **advanced debugging**. When tuning reveals a deeper bug, when a driver panics under specific load, when memory corruption shows up in a crash dump, the tools in Chapter 34 are the ones you reach for. `KASSERT(9)` for in-code assertions, `panic(9)` for unrecoverable states, `ddb(4)` for interactive live kernel debugging, `kgdb` for remote and post-mortem debugging, `kttr(9)` for lightweight ring-buffer tracing, and `memguard(9)` for memory-corruption detection. The boundary between performance work and debugging is often fuzzy: a performance problem turns out to be a correctness bug in disguise, or a correctness bug hides behind a performance symptom. Chapter 34 gives you the tools for the other half of the diagnosis.

Beyond Chapter 34, the final chapters of Part 7 complete the mastery arc: asynchronous I/O and event handling, coding style and readability, contributing to FreeBSD, and a capstone project that brings several earlier threads together. Each builds on what you have done here.

The measurement work is done. The next chapter is about what to do when the numbers point at something the tuning tools alone cannot reach.

### 33.17.8 Glossary of Terms Introduced in This Chapter

A quick reference for the vocabulary the chapter introduced or used heavily. When a term has a dedicated manual page or a specific FreeBSD primitive, the reference is noted.

**Aggregation (DTrace):** A server-side running summary (count, sum, min, max, average, quantize) computed in kernel space and printed at session end. Use aggregations, not per-event prints, for any probe firing more than a few times per second.

**Benchmark harness:** A repeatable script that runs a workload against the driver with controlled parameters, collects measurements, and reports results in a consistent format. The harness reduces measurement-to-measurement variance by removing human variation.

**Bounce buffer:** A kernel buffer that sits between a device and user memory when direct DMA is not possible because of alignment or address-range constraints. Managed by `bus_dma(9)`. Bounce buffers cost an extra copy; eliminating them is a common tuning target.

**Cache coherence:** The hardware mechanism that keeps caches on multiple CPUs consistent. When one CPU writes a line, other CPUs that hold the line must invalidate or update their copies. The cost of coherence traffic is why false sharing is a problem.

**Cache-line alignment:** Placing a variable on a boundary that matches the hardware cache-line size (typically 64 bytes on amd64). Used to prevent false sharing between a hot variable and its neighbours. `__aligned(CACHE_LINE_SIZE)`.

**Callgraph (profiling):** A tree of caller-callee relationships weighted by sample count. Reading a callgraph tells you which call paths consume the most time, not just which leaf functions.

**Coalescing:** Combining many small events into fewer large ones. Hardware interrupt coalescing reduces interrupt rate at the cost of per-event latency.

**Counter(9):** A FreeBSD primitive for per-CPU summed counters. Cheap on update, cheap on aggregate read, scales with CPU count.

**DPCPU:** FreeBSD per-CPU variable storage with alignment and access macros. `DPCPU_DEFINE_STATIC`, `DPCPU_GET`, `DPCPU_PTR`, `DPCPU_SUM`.

**DTrace:** FreeBSD's dynamic tracing framework. Safe, pervasive, near-zero cost when disabled. Scripts compose probes, predicates, and actions.

**Event mode (PMC):** Hardware performance counter mode where the counter counts a specific event (cycles, instructions, cache misses). Compare with sampling mode.

**False sharing:** Cache-coherence ping-pong between CPUs caused by unrelated variables living in the same cache line. The effect is invisible in source code and obvious in profile results.

**fbt provider:** DTrace's *function boundary tracing* provider. Gives you probes on entry and return of every kernel function that is not inlined.

**Filter handler:** An interrupt handler that runs in the low-level interrupt context. Does the minimum and either claims the interrupt and signals the ithread (`FILTER_SCHEDULE_THREAD`), or declines (`FILTER_STRAY`).

**Flame graph:** A visualisation of profile samples as nested rectangles. Width represents sample count; height represents stack depth. Makes hotspot identification visual.

**hwpmc(4):** The FreeBSD kernel module that exposes hardware performance counters. Loaded with `kldload hwpmc`.

**Ithread:** Interrupt thread. A kernel thread that runs an interrupt handler in thread context, not interrupt context. Scheduled at a driver-specified priority.

**Latency:** The time from a request's arrival to its completion. Typically reported as percentiles (P50, P95, P99, P99.9) rather than an average.

**Lock contention:** The state where multiple threads compete for the same lock, causing some to wait. Detectable with the DTrace lockstat provider.

**MSI / MSI-X:** Message Signalled Interrupt (PCIe). Replaces legacy INTx interrupts with in-band messages. MSI-X allows many independent vectors per device, which enables per-queue or per-CPU interrupt handling.

**NAPI-style batching:** A Linux-borrowed but FreeBSD-applicable technique where an interrupt disables further interrupts from the device and a poll loop drains the queue until empty, then re-enables interrupts. Amortises interrupt overhead under load.

**Observability:** The property of a system that makes its internal state visible to external tools. In FreeBSD, sysctl trees, counters, SDT probes, and log messages contribute to observability.

**P99 / P99.9:** The 99th and 99.9th percentiles of a distribution. The value below which 99% (or 99.9%) of samples fall. Tail percentiles often reveal worse user experience than averages.

**Per-CPU data:** Data structures that have a distinct copy on each CPU, eliminating cache-line contention. FreeBSD's DPCPU macros manage layout and access.

**pmcstat(8):** The userland tool that drives hwpmc(4). Supports event mode and sampling mode, system-wide and per-process.

**profile provider (DTrace):** A DTrace provider that fires at a fixed frequency (for example, 997 Hz) on all CPUs. Used for portable statistical profiling.

**Responsiveness:** The time from an external stimulus (interrupt, ioctl) to the driver's first action. Distinct from latency (request-to-completion) and from throughput.

**Sampling mode (PMC):** Hardware performance counter mode where the counter triggers an interrupt every N events and the kernel records the program counter. Used for statistical profiling.

**sched provider (DTrace):** A DTrace provider for scheduler events: on-cpu, off-cpu, enqueue, dequeue. Used to diagnose scheduling-related latency.

**SDT probe:** Statically-defined trace probe. Compiled into the driver with SDT\_PROBE\_DEFINE and SDT\_PROBE\*( ) macros. Always present, always stable, near-zero cost when disabled.

**softc:** A driver's *software context* structure. Holds per-device state. Accessed with device\_get\_softc(dev).

**sysctl tree:** A hierarchy of kernel variables exposed to userland through sysctl(8). Drivers register nodes under hw.<drivername> by convention.

**Tail latency:** The high percentiles of a latency distribution (P99, P99.9, P99.99). Often what users perceive as *slow*, even when averages are good.

**Taskqueue:** A FreeBSD kernel thread pool for deferred work. Drivers enqueue tasks with taskqueue\_enqueue(9); the thread pool dequeues and runs them.

**TMA (Top-Down Microarchitecture Analysis):** A method for classifying CPU-bound performance problems. Branches into *frontend-bound*, *bad-speculation*, *backend-bound*, and *retiring*.

**Throughput:** The rate at which the driver completes work, measured as operations per second or bytes per second.

**UMA zone:** A specialised allocator for fixed-size objects. Created with uma\_zcreate(). Cheap per-CPU cached allocation; suitable for hot paths with fixed-size buffers.



**workload:** The specific pattern of requests used during measurement. A driver's performance depends on the workload; a measurement without a named workload is incomplete.

### 33.17.9 A Parting Word

Performance engineering has a reputation for being arcane. The reputation is half earned and half unearned. It is earned because the tools are many, the output is numeric, and the interpretation takes experience. It is unearned because the discipline is small and learnable: measure before you change, state the goal in numbers, prefer the simplest tool, know the cost of your probes, preserve correctness. That is the whole discipline in a sentence.

The chapter's hardest lesson is the one about humility. A beginner's instinct is to look at a line of code and think *I can make this faster*. An experienced engineer's instinct is to look at a measurement and think *this is the hot function; now let me understand why*. The gap between the two mindsets is the gap between years of practice. Every hour spent with DTrace, pmcstat, and a notebook full of real measurements narrows it.

You have now met the measurement half of the FreeBSD performance toolkit. The tools are in the tree; the manual pages are waiting; the habit of using them is the last piece. Make it part of every serious driver you write, and the rest follows.

One last encouragement before we move on. If the depth of the chapter felt intimidating, return to it one section at a time. Performance work is not a single skill to master in one sitting; it is a cluster of related skills, each small in isolation. Run one DTrace one-liner. Expose one sysctl counter. Run one pmcstat session. Each small practice earns a small confidence, and the confidences compound. Within a handful of drivers, the toolkit feels routine rather than exotic, and the mindset of *measure, then decide* becomes second nature rather than a checklist.

On to Chapter 34 and the debugging techniques that complement everything we have just covered.

Take a deep breath; you have earned the next chapter.

## 34 Advanced Debugging Techniques

### 34.1 Introduction

In the previous chapter we learned how to measure what a driver does and how fast it does it. We watched performance counters grow, we ran DTrace aggregations to spot hot paths, and we used `pmcstat` to see which instructions were actually consuming cycles. Measurement gave us a language for asking whether the driver behaves the way we expect.

Debugging asks a different question. Instead of “how fast is this?” it asks “why is this wrong?” A performance problem usually produces slow but running code. A correctness problem can produce a crash, a deadlock, silent data corruption, a driver that refuses to unload, a pointer that dereferences into garbage, or a lock that is somehow held by nobody. These are the bugs that make a seasoned kernel engineer take a slow breath and reach for better tools.

FreeBSD gives us those tools. They range from very small and very fast assertions that live inside the kernel and catch the bug the instant it happens, to full post-mortem analysis of a crash dump on a machine that is no longer running. There are lightweight tracing rings that cost almost nothing at runtime, heavyweight tracers that can unwind the entire call graph, and memory allocators that can be swapped in during development to turn subtle use-after-free bugs into immediate, diagnosable crashes. A well-equipped driver author learns to reach for the right tool for the right bug, rather than staring at `printf` output hoping for enlightenment.

The goal of this chapter is to teach you that toolkit. We will begin by understanding when advanced debugging is the right response and when a simpler approach will serve better. We will then work through the in-kernel assertion macros, the panic path, and how to read and analyze a crash dump offline with `kgdb`. We will build a debug-friendly kernel so those tools are actually available when we need them, learn how to trace driver behavior with DTrace and `ktrace`, and finally study how to hunt memory leaks and invalid accesses with `memguard(9)`, `redzone`, and guard pages. We will close with the discipline of debugging on production systems, where every action has consequences, and with a short study of how to refactor a driver after a hard failure so that it is more resilient to the next one.

Throughout the chapter we will use a small companion driver called `bugdemo`. It is a pseudo-device with deliberate, controlled bugs that we can trigger through simple `ioctl(2)` calls and then hunt with each of the techniques the chapter teaches. Nothing we do touches real hardware, so the lab environment stays safe even when we deliberately crash the kernel.

By the end of the chapter you will be able to add defensive assertions to a driver, build a debug kernel, capture a crash dump, open it in `kgdb`, trace live behavior with DTrace and `ktrace`, trap memory misuse with `memguard(9)` and friends, and apply all of this discipline safely on systems where other people are relying on the machine.

## 34.2 Reader Guidance: How to Use This Chapter

This chapter sits in Part 7 of the book, alongside performance tuning, asynchronous I/O, and other mastery topics. It assumes you have already written at least a simple character device driver, understand the load and unload lifecycle, and have worked with `sysctl`, `counter(9)`, and `DTrace` at the level introduced in Chapter 33. If any of that feels uncertain, a quick revisit of Chapters 8 through 14 and Chapter 33 will pay for itself several times over in this chapter.

### 34.2.1 Reading This Chapter Alongside Chapter 23

This chapter deliberately picks up where Chapter 23 left off. Chapter 23, “Debugging and Tracing,” introduced the fundamentals: how to think about bugs, how to reach for `printf`, how to use `dmesg` and the kernel log, how to read a simple panic, how to turn on `DTrace` probes, and how to make a driver easier to observe from the start. It stays close to the everyday debugging habits a new driver author needs.

Chapter 23 also ends with an explicit hand-off. It flags that deep `kgdb` scripting on top of a crash dump and the live-kernel breakpoint workflows are reserved for a later, more advanced chapter. That later chapter is this one. You are reading the second half of a pair. If Chapter 23 is the first aid kit, Chapter 34 is the full clinical toolbox.

In practice, this means two things. First, we will not re-explain the fundamentals Chapter 23 already covered; we will assume you are comfortable with `printf`, basic panic reading, and introductory `DTrace`. If any of those feel shaky, reread the relevant section of Chapter 23 first, because the advanced material builds directly on those habits. Second, when a technique here has a simpler counterpart in Chapter 23 (for example, a basic `bt` in `kgdb` is simpler than walking `struct thread` fields), we will point to the Chapter 23 version and then show why the advanced version earns its extra complexity.

Think of the two chapters as a single arc. Chapter 23 teaches you how to notice that something is wrong and take a first look. Chapter 34 teaches you how to reconstruct, in detail, what the kernel was doing at the moment the bug occurred, even on a machine that is no longer running.

The material is cumulative. Each section adds another layer onto the `bugdemo` driver, so the labs read most naturally in order. You can skim ahead for reference, but if this is your first encounter with kernel debugging tools, walking through the labs in sequence will build the mental model we want.

You do not need any special hardware. A modest FreeBSD 14.3 virtual machine is enough for every lab in the chapter. For Lab 3 and Lab 4 you will want to have configured a crash dump device, which the chapter walks through, and for Lab 5 you will need `DTrace` available in your kernel. Both are standard on an ordinary FreeBSD installation.

Some of the techniques in this chapter deliberately crash the kernel. This is safe on a development machine and expected as part of the learning process. It is not safe on a production machine where other people depend on uninterrupted service. The final section of the chapter is dedicated to that distinction, because the discipline of knowing when not to use a tool is as important as knowing how to use it.

### 34.3 How to Get the Most Out of This Chapter

The chapter is organized around a pattern you will see repeated throughout. First we explain what a technique is, then we explain why it exists and what kind of bug it is meant to catch, then we ground it in real FreeBSD source code so you can see where the idea lives in the kernel, and finally we apply it to the bugdemo driver through a small lab. Reading and experimenting together is the most effective approach. The labs are deliberately small enough to run in a few minutes each.

A few habits will make the work smoother. Keep a terminal open to `/usr/src/` so you can look at the real code whenever the chapter references it. The book teaches through observation of real FreeBSD practice, not through invented pseudo-code, and you will build stronger intuition by confirming with your own eyes that `KASSERT` really is defined where the chapter says it is, or that `memguard(9)` really has the API we describe.

Keep a second terminal open to your test VM, where you will load the bugdemo driver, trigger bugs, and watch the output. If you can attach a serial console to the VM, do so. A serial console is the most reliable way to capture the tail end of a panic message before the machine reboots, and we will use it in several labs.

Finally, keep your expectations calibrated. Kernel bugs are often not what they appear to be at first. A use-after-free might first present itself as a random data corruption in an unrelated subsystem. A deadlock might first look like a slow system call. One of the most valuable skills this chapter teaches is patience: gathering evidence before forming a theory, and confirming the theory before committing to a fix. The tools help, but the discipline is what separates a quick bug hunt from a long one.

With those expectations in place, let us begin by discussing when advanced debugging is actually the right response to a problem.

### 34.4 1. When and Why You Need Advanced Debugging

Most bugs in a driver can be solved without reaching for a crash dump or a tracing framework. A careful reading of the code, an extra `printf`, a second look at the return value of a function, a glance at `dmesg`: these together resolve the majority of defects a driver author encounters. If you can see the problem, reproduce it cheaply, and hold the relevant code in your head, the simplest tool is the right tool.

Advanced debugging exists for the bugs that do not yield to that approach. It is the toolkit we reach for when the problem is rare, when it appears far from its cause, when it only shows up under specific timing, when the driver hangs rather than crashes, or when the symptom is corruption rather than failure. Those bugs share a common property: they require evidence you cannot easily collect by reading code, and they require control over the kernel's execution that a normal user process does not have.

#### 34.4.1 Bugs That Need More Than a `printf`

The first class of bug that demands advanced tooling is the bug that destroys the evidence of its own cause. A use-after-free is the canonical example. The driver frees an object, then some later code, possibly in a different function or a different thread, reads or writes that memory. By the time the crash happens, the free has long since occurred, the memory has been reused for something unrelated, and the backtrace at the point of the crash points to the victim, not the culprit. A `printf` added at the crash site will faithfully print the nonsense it sees. It will not tell you who freed the memory or when.

A second class is the bug that only appears under concurrency. Two threads race for a lock. One of them takes the lock in the wrong order, deadlocking against another thread that took the same locks in the reverse order. The system goes quiet, and the bug leaves no message on the console. Adding `printf` calls to the locking path often perturbs the timing just enough to make the bug disappear, a frustrating property that Heisenbug enthusiasts are familiar with. Static lock order checking, which FreeBSD provides through `WITNESS`, exists precisely because this class of bug is hard to find any other way.

A third class is the bug that cannot be observed at all in user space. The driver corrupts a kernel data structure on one code path, and the consequence shows up many minutes later in an unrelated subsystem. The process that triggers the corruption is long gone by the time anything goes wrong. The only way to correlate cause and effect is to capture the full kernel state at the moment of the panic and walk it offline with `kgdb`, or to trace the kernel continuously with `DTrace` so that the suspicious event leaves a trail.

A fourth class is the bug that only appears on hardware you cannot attach a debugger to, or in production configurations you cannot directly instrument. The driver runs on a customer's machine, it crashes once a week, and nobody wants your development workstation physically plugged into it. The tool for this situation is the crash dump: a snapshot of kernel memory written to disk at the moment of panic, carried to a safe environment, and analyzed there. `dumpon(8)` configures where the dump goes, `savecore(8)` retrieves it after the reboot, and `kgdb` reads it offline.

Each of these bug classes has its own tool in the FreeBSD debugging toolkit. The rest of the chapter introduces them in turn. The point of this opening section is simply to set expectations: we are not about to learn a single technique that replaces `printf`. We are learning a family of techniques, each suited to a particular kind of difficulty.

### 34.4.2 The Cost of Advanced Tooling

Advanced debugging is not free. Each of the techniques we will study carries some combination of build-time cost, run-time cost, and disciplinary cost.

Build-time cost is the easiest to describe. `INVARIANTS` and `WITNESS` make the kernel slower because they add checks that a production kernel skips. `DEBUG_MEMGUARD` makes certain allocations dramatically slower because it replaces them with full-page mappings that are unmapped on free. A debug kernel built with `makeoptions DEBUG=-g` is several times larger than a release kernel because every function carries full debug information. None of these costs matter on a development machine, where correctness is worth orders of magnitude more than speed. All of them matter in production.

Run-time cost applies to the tools you enable in a running kernel. `DTrace` probes that are disabled cost essentially nothing, but an enabled probe still runs on every hit of the instrumented function. `kt(9)` entries are very cheap but not free. A verbose tracing session can generate enough log output to fill a disk. A `kdb` session pauses the entire kernel, which on a machine people are using is a disaster. Each tool has a run-time budget, and part of the discipline of this chapter is knowing what that budget is.

Disciplinary cost is the hardest to quantify but the easiest to underestimate. Advanced debugging requires patience, careful note taking, and a willingness to sit with incomplete information. It requires resisting the urge to patch a visible symptom before you understand the underlying defect. A crash that happens in module `X` almost never means the bug is in module `X`. The reader who learns to collect evidence before forming a theory will have an easier time with this chapter than the reader who wants to commit a fix as quickly as possible.

### 34.4.3 A Decision Framework

With those costs in mind, here is a simple decision framework for choosing your tool. If the bug is easy to reproduce and the cause is likely visible in the nearby code, start with reading and with strategic `printf` or `log(9)` statements. If the bug only appears under load or concurrency, enable `INVARIANTS` and `WITNESS` and rebuild. If the bug produces a panic, capture the dump and open it in `kgdb`. If the bug involves memory corruption, enable `DEBUG_MEMGUARD` on the suspect allocation type. If the bug involves a silent misbehavior rather than a crash, add SDT probes and watch them with `DTrace`. If you need to understand timing between events in an interrupt handler, use `ktr(9)`. And if the bug is on a production machine, see Section 7 before doing anything at all.

We will spend the rest of the chapter teaching each of these techniques in depth. The `bugdemo` driver we are about to meet gives us a safe place to apply every one of them, with known bugs to hunt and known answers to find.

### 34.4.4 Meeting the `bugdemo` Driver

The `bugdemo` driver is a small pseudo-device that we will use as our lab subject throughout the chapter. It has no hardware to drive. It exposes a device node at `/dev/bugdemo` and accepts a handful of `ioctl(2)` commands that deliberately trigger different classes of bug: a null pointer dereference, an unlocked access that `WITNESS` can catch, a use-after-free, a memory leak, an infinite loop inside a spinlock, and so on. Each `ioctl` is gated by a `sysctl` switch so that the driver can be loaded safely on a development system without accidentally triggering anything.

We will introduce the driver properly in Lab 1, when we have the assertion macros in hand. For now, keep in mind that every technique we study can be demonstrated on `bugdemo` with a known starting point and a known answer. That discipline, of reproducing bugs in a controlled setting, is itself one of the most important skills this chapter aims to teach.

Now we are ready to begin the toolkit proper, starting with the assertion macros that catch bugs at the instant they happen.

## 34.5 2. Using `KASSERT`, `panic`, and Related Macros

Defensive programming in user space often revolves around runtime checks and careful error handling. Defensive programming in the kernel adds one more tool: the assertion macro. An assertion states a condition that must be true at a given point. If the condition is false, something is very wrong, and the safest response is to stop the kernel immediately, before the wrong state has a chance to spread. Assertions are the cheapest and most effective debugging tool FreeBSD gives us, and they belong in every serious driver.

We will start with the two most important macros, `KASSERT(9)` and `panic(9)`, look at a handful of useful companions, and then discuss when each is appropriate.

**A note on line numbers.** When the chapter quotes code from `kassert.h`, `kern_shutdown.c`, or `cdefs.h`, the landmark is always the macro or function name. `KASSERT`, `kassert_panic`, `panic`, and `__dead2` will remain findable by those names in every FreeBSD 14.x tree even as the lines around them move. The sample backtrace you will see later, which quotes `file:line` pairs such as `kern_shutdown.c:400`,

reflects a 14.3 tree at the time of writing and will not match line-for-line on a freshly updated system. Grep for the symbol rather than scroll to the number.

### 34.5.1 KASSERT: A Check That Vanishes in Production

KASSERT is the in-kernel equivalent of the user-space `assert()` macro, but smarter. It takes a condition and a message. If the condition is false, the kernel panics with the message. If the kernel was compiled without the `INVARIANTS` option, the entire check disappears at compile time and costs nothing at runtime.

The macro lives in `/usr/src/sys/sys/kassert.h`. In a FreeBSD 14.3 source tree it looks like this:

```
#if (defined(_KERNEL) && defined(INVARIANTS)) || defined(_STANDALONE)
#define KASSERT(exp,msg) do { \
 if (__predict_false(!(exp))) \
 kassert_panic msg; \
} while (0)
#else /* !(KERNEL && INVARIANTS) && !STANDALONE */
#define KASSERT(exp,msg) do { \
} while (0)
#endif /* KERNEL && INVARIANTS */
```

Four details of this definition are worth pausing on.

First, the macro is defined differently depending on whether `INVARIANTS` is set. If it is not, `KASSERT` expands to an empty `do { } while (0)` block, which the compiler optimizes away entirely. A release kernel built without `INVARIANTS` pays no runtime cost for `KASSERT` calls, no matter how many the driver contains. This is the property that lets us write generous assertions in development without worrying about production performance. The `_STANDALONE` branch lets the same macro work in the bootloader, where `INVARIANTS` may be absent but the check is still desired.

Second, the `__predict_false` hint tells the compiler that the condition is almost always true. This improves code generation for the common path, because the compiler will arrange the branch so that the hot path does not take a jump. Defining `__predict_false` is one of the small performance disciplines that keeps a debug kernel usable.

Third, the body of a failing assertion calls `kassert_panic`, not `panic` directly. This is an implementation detail for making assertion messages easier to parse, but it matters when you see a panic message in practice: `KASSERT` failures produce a distinctive prefix that we will recognize later.

Fourth, notice that the `msg` argument is passed in double parentheses. That is because the macro passes it directly to `kassert_panic`, which has a `printf`-style signature. In practice you write:

```
KASSERT(ptr != NULL, ("ptr must not be NULL in %s", __func__));
```

The outer parentheses belong to the macro. The inner parentheses are the argument list for `kassert_panic`. A beginner's mistake is to write `KASSERT(ptr != NULL, "ptr is NULL")` with only one set of parentheses, which will not compile. The double parentheses are the discipline that reminds us a failing assertion will be formatted like a `printf`.

### 34.5.2 INVARIANTS and INVARIANT\_SUPPORT

INVARIANTS is the kernel build option that controls whether KASSERT is active. A debug kernel enables it. The GENERIC-DEBUG configuration shipped with FreeBSD 14.3 enables it by including `std.debug`, which you can see in `/usr/src/sys/conf/std.debug`. A release GENERIC kernel does not enable it.

There is also a related option called INVARIANT\_SUPPORT. INVARIANT\_SUPPORT compiles in the functions that assertions might call, without making them mandatory. This allows loadable kernel modules that were built with INVARIANTS to load into a kernel that was not built with INVARIANTS, as long as INVARIANT\_SUPPORT is present. For a driver author, the practical upshot is this: if you build your module with INVARIANTS, make sure the kernel you are loading it into has at least INVARIANT\_SUPPORT. The GENERIC-DEBUG kernel has both, which is one of the reasons we recommend using it throughout development.

### 34.5.3 MPASS: KASSERT With a Default Message

Writing a message for every assertion can be tedious, especially for simple invariants. FreeBSD provides MPASS as a shorthand for `KASSERT(expr, ("Assertion expr failed at file:line"))`:

```
#define MPASS(ex) MPASS4(ex, #ex, __FILE__, __LINE__)
#define MPASS2(ex, what) MPASS4(ex, what, __FILE__, __LINE__)
#define MPASS3(ex, file, line) MPASS4(ex, #ex, file, line)
#define MPASS4(ex, what, file, line) \
 KASSERT((ex), ("Assertion %s failed at %s:%d", what, file, line))
```

The four forms let you customize the message, the file, or both. The simplest form, `MPASS(ptr != NULL)`, stringifies the expression itself and embeds the location automatically. When the message can afford to be terse, MPASS produces less visual clutter in the source. When the message needs context a future reader will appreciate, prefer KASSERT with a written message.

A sensible rule of thumb is that MPASS is for internal invariants that should never happen and where the identity of the expression is self-explanatory. KASSERT is for conditions where the failure mode deserves a descriptive message.

### 34.5.4 CTASSERT: Compile-Time Assertions

Sometimes the condition you want to check can be decided at compile time. `sizeof(struct foo) == 64`, for example, or `MY_CONST >= 8`. For those, FreeBSD provides CTASSERT, also in `/usr/src/sys/sys/kassert.h`:

```
#define CTASSERT(x) _Static_assert(x, "compile-time assertion failed")
```

CTASSERT uses C11's `_Static_assert`. It produces a compile-time error if the condition is false and has zero runtime cost because there is no runtime involved. This is the ideal tool for structure layout checks that must hold for the driver to be correct.

A typical use in the kernel is to guard a structure against accidental size changes:



```
struct bugdemo_command {
 uint32_t op;
 uint32_t flags;
 uint64_t arg;
};

CTASSERT(sizeof(struct bugdemo_command) == 16);
```

If someone later adds a field without adjusting the size comment or reordering thoughtfully, the build breaks immediately. This is far better than finding out at runtime that the structure has grown and the ioctl no longer matches user-space expectations.

### 34.5.5 panic: The Unconditional Stop

Where KASSERT is a conditional check, panic is the unconditional version. You call it when you have decided that continuing execution would be worse than stopping:

```
void panic(const char *, ...) __dead2 __printflike(1, 2);
```

The declaration lives in `/usr/src/sys/sys/kassert.h` and the implementation in `/usr/src/sys/kern/kern_shutdown.c`. The `__dead2` attribute tells the compiler that panic does not return, which lets it produce better code downstream. The `__printflike(1, 2)` attribute tells it that the first argument is a printf-style format string, so the compiler can type-check the format against its arguments.

When would you use panic directly rather than KASSERT? Three common situations. First, when the condition is so catastrophic that there is no safe continuation even in a release kernel. Failing to allocate a soft context during attach, for example, might be a panic rather than a graceful cleanup if the driver has already been partially registered. Second, when you want the message to appear even in non-debug builds, because the event indicates a hardware or configuration failure that the user must be told about. Third, as a placeholder during early development, to make sure unreachable paths actually are unreachable, before you replace the panic with a `KASSERT(0, ...)` in mature code.

Some drivers in `/usr/src/sys/dev/` use panic sparingly. Reading a few examples will give you a feel for the tone: a panic message says something like “the controller returned an impossible status” or “we reached a case the state machine claims cannot happen.” It is not the normal response to an I/O error. It is the response to an invariant that has been broken so thoroughly that the driver cannot be trusted to continue.

### 34.5.6 \_\_predict\_false and \_\_predict\_true

We saw `__predict_false` in the KASSERT definition. These two macros, defined in `/usr/src/sys/sys/cdefs.h`, are compile-time hints to the branch predictor:

```
#if __GNUC_PREREQ__(3, 0)
#define __predict_true(exp) __builtin_expect((exp), 1)
#endif
```

```

#define __predict_false(exp) __builtin_expect((exp), 0)
#else
#define __predict_true(exp) (exp)
#define __predict_false(exp) (exp)
#endif

```

They do not change the semantics of the expression. They only tell the compiler which outcome is more likely, which influences how the compiler lays out the code. In a hot path, wrapping a likely-true condition in `__predict_true` can improve cache behavior; wrapping a likely-false one in `__predict_false` keeps the error-handling code off the fast path.

The first rule of using these macros is to be correct. If you predict wrong, you slow the code down rather than speeding it up. The second rule is to only use them in genuinely hot paths where the difference matters. For most driver code, the compiler's default heuristics are fine, and cluttering the code with predictions is more trouble than it is worth.

### 34.5.7 Where Assertions Belong in a Driver

With these macros in hand, where do you actually put assertions? A few patterns have proven useful in FreeBSD drivers.

The first is at function entry, for non-trivial preconditions. A driver function that expects to be called with a particular lock held is a perfect candidate:

```

static void
bugdemo_process(struct bugdemo_softc *sc, struct bugdemo_command *cmd)
{
 BUGDEMO_LOCK_ASSERT(sc);
 KASSERT(cmd != NULL, ("cmd must not be NULL"));
 KASSERT(cmd->op < BUGDEMO_OP_MAX,
 ("cmd->op %u out of range", cmd->op));
 /* ... */
}

```

The `BUGDEMO_LOCK_ASSERT` is a macro convention many drivers adopt that wraps a `mtx_assert(9)` or `sx_assert(9)` call. This pattern, where each subsystem has its own `_ASSERT` macro that checks its own lock, scales well across a large driver.

The second pattern is at state transitions. If a driver state machine has four valid states and an attach path that should only ever run in state `INIT`, an assertion at the top of `attach` will catch any future refactor that breaks that invariant:

```

KASSERT(sc->state == BUGDEMO_STATE_INIT,
 ("attach called in state %d", sc->state));

```

The third pattern is after subtle arithmetic. If a calculation should produce a value in a known range, check:

```
idx = (offset / PAGE_SIZE) & (SC_NRING - 1);
KASSERT(idx < SC_NRING, ("idx %u out of range", idx));
```

This is especially valuable in ring buffer code, where an off-by-one between producer and consumer can cause silent data corruption.

The fourth pattern is for pointers that could be NULL but should not be. If a function receives a pointer argument that is only valid when nonzero, a single `KASSERT(ptr != NULL, ...)` at the top of the function catches years of future misuse.

### 34.5.8 When Not to Use Assertions

Assertions are not a replacement for error handling. The rule is: `KASSERT` checks things that the programmer guarantees, not things the environment guarantees. If a memory allocation with `M_NOWAIT` can fail under memory pressure, you do not assert it succeeded. You check the return value and handle the failure. If a user-space program passes a structure larger than you expect, you return `EINVAL`, not `KASSERT(0)`. The assertion is for internal consistency, not for external input.

Another anti-pattern is using assertions for conditions that only hold in certain configurations. `KASSERT(some_sysctl == default)` is wrong if `some_sysctl` is user-tunable, because the assertion will fail on any system that has tuned it. Check the configuration explicitly and handle it, or assert only within the branch where the assumption actually holds.

A more subtle anti-pattern is using assertions as documentation. “This is how it works, and it had better stay that way” is a tempting use of `KASSERT`, but if the assertion only holds today and could reasonably change tomorrow, you have created a future bug for someone who does not remember your promise. Better to leave a comment that states the assumption, and let the code evolve. Assertions should capture permanent invariants, not temporary implementation choices.

### 34.5.9 A Small Real-World Example

Let us see these ideas applied in real FreeBSD code. Open `/usr/src/sys/dev/null/null.c` and look at a typical check near the read handler. The driver is extremely simple, so there are few assertions, but many drivers in `/usr/src/sys/dev/` use `KASSERT` liberally. For a richer example, scan `/usr/src/sys/dev/uart/uart_bus_pci.c` or `/usr/src/sys/dev/mii/mii.c`, where assertions at function entry catch callers that do not hold the expected locks.

The consistency of this pattern across the tree is not accidental. It reflects a cultural expectation that drivers will express their invariants in code, not only in comments. When you adopt the same habit in your own drivers, you join that culture. Your drivers will be easier to port, easier to review, and much easier to debug when something eventually goes wrong.

### 34.5.10 A Quick Example: Adding Assertions to bugdemo

Let us add a small set of assertions to an imagined bugdemo driver. Assume we have a `softc` structure with a mutex, a state field, and a counter, and an `ioctl` handler that takes a `struct bugdemo_command`.

```
static int
bugdemo_ioctl(struct cdev *dev, u_long cmd, caddr_t data, int fflag,
 struct thread *td)
{
 struct bugdemo_softc *sc = dev->si_drv1;
 struct bugdemo_command *bcmd = (struct bugdemo_command *)data;

 KASSERT(sc != NULL, ("bugdemo: softc missing"));
 KASSERT(sc->state == BUGDEMO_STATE_READY,
 ("bugdemo: ioctl in state %d", sc->state));

 switch (cmd) {
 case BUGDEMO_TRIGGER:
 KASSERT(bcmd->op < BUGDEMO_OP_MAX,
 ("bugdemo: op %u out of range", bcmd->op));
 BUGDEMO_LOCK(sc);
 bugdemo_process(sc, bcmd);
 BUGDEMO_UNLOCK(sc);
 return (0);
 default:
 return (ENOTTY);
 }
}
```

Four assertions, each catching a different class of future bug. The first checks that the driver's private pointer is actually set, which is easy to forget when `make_dev(9)` is called with `NULL` by mistake. The second checks the driver state, which will fire if someone adds a code path that can reach `ioctl` before `attach` has finished. The third checks user-provided input is in range, though in a production context this particular check would also be done as a real input validation that returns an error, because `ioctl` is a public interface. The fourth, not shown here but implied by `bugdemo_process`, asserts the lock is held.

These few lines express a lot of invariants. In a debug kernel, they will catch real bugs the moment they happen. In a release kernel, they disappear entirely. That is the bargain `KASSERT` offers, and taking it is one of the best habits a driver author can cultivate.

With this foundation in place, we can move on to what happens when an assertion actually fires, which brings us to the panic path and the crash dump.

## 34.6 3. Analyzing Panics and Crash Dumps

When a KASSERT fails or `panic` is called, the kernel takes a series of well-defined steps. Understanding those steps is the first part of making sense of a crash. The second part is knowing what traces the kernel leaves behind, and how to read them after the fact. This section walks through both.

### 34.6.1 What Happens During a Panic

A panic is the kernel's controlled shutdown in response to an unrecoverable error. The exact sequence depends on the build options, but a typical panic in a FreeBSD 14.3 kernel proceeds as follows.

First, `panic()` or `kassert_panic()` is called with a message. The message is formatted and written to the system log. If a serial console is attached, it appears there immediately. If only a graphical console is available, it appears on screen, though there is often too little time to read a long trace before the machine reboots, which is one reason we recommend a serial or virtual console during this chapter.

Second, the kernel captures a backtrace of the panicking thread. You will see this on the console as a list of function names with offsets. The backtrace is the most valuable piece of information a panic produces, because it tells you the chain of calls that led to the failure. Reading it top-down shows you the function that called `panic`, the function that called that one, and so on, back to the entry point.

Third, if the kernel has been built with KDB enabled and a backend such as DDB, the kernel enters the debugger. DDB is the in-kernel debugger. It accepts commands directly on the console: `bt` to show a backtrace, `show registers` to dump register state, `show proc` to show process information, and so on. We will use DDB briefly in Section 4. If KDB is not enabled, or if the kernel is configured to skip the debugger on panic, the kernel moves on.

Fourth, if a dump device is configured, the kernel writes a dump to it. The dump is the entire contents of kernel memory, or at least the portions marked as dumpable, serialized onto the dump device. This is the crash dump that `savecore(8)` will retrieve after the reboot.

Fifth, the kernel reboots the machine, unless it has been asked to stop at the debugger. After the reboot, when the system comes up, `savecore(8)` runs and writes the dump to `/var/crash/vmcore.N` along with a textual summary. You now have everything you need to analyze the crash offline.

The whole sequence takes anywhere from a fraction of a second to a few minutes, depending on how large the kernel is, how fast the dump device is, and how configured the system is. On a development VM, dumping a kernel of a few hundred megabytes to a virtual disk is typically a matter of seconds.

### 34.6.2 Reading a Panic Message

A panic message in FreeBSD 14.3 looks something like this:

```
panic: bugdemo: softc missing
cpuid = 0
time = 1745188102
KDB: stack backtrace:
db_trace_self_wrapper() at db_trace_self_wrapper+0x2b
```

```
vpanic() at vpanic+0x182
panic() at panic+0x43
bugdemo_ioctl() at bugdemo_ioctl+0x24
devfs_ioctl() at devfs_ioctl+0xc2
VOP_IOCTL_APV() at VOP_IOCTL_APV+0x3f
vn_ioctl() at vn_ioctl+0xdc
devfs_ioctl_f() at devfs_ioctl_f+0x1a
kern_ioctl() at kern_ioctl+0x284
sys_ioctl() at sys_ioctl+0x12f
amd64_syscall() at amd64_syscall+0x111
fast_syscall_common() at fast_syscall_common+0xf8
--- syscall (54, FreeBSD ELF64, sys_ioctl), rip = ..., rsp = ...
```

Read this top to bottom. The first line is the panic message itself. The `cpuid` and `time` lines are metadata; they are rarely useful for debugging but occasionally help when reconciling multiple logs. The `KDB: stack backtrace:` line marks the start of the trace.

The first few frames are the panic infrastructure itself: `db_trace_self_wrapper`, `vpanic`, `panic`. These are always there on a panic and can be skipped. The first interesting frame is `bugdemo_ioctl`, which is where our driver called `panic`. The frames below that are the path that got us to `bugdemo_ioctl`: `devfs_ioctl`, `vn_ioctl`, `kern_ioctl`, `sys_ioctl`, `amd64_syscall`. This tells us that the panic happened during an `ioctl` system call, which is already a useful clue. The final line shows the `syscall` number (54, which is `ioctl`) and the instruction pointer at entry.

The offsets (`+0x24`, `+0xc2`) are byte offsets into each function. On their own they are not human-readable, but they let `kgdb` resolve the exact line of source code when the debug kernel is available.

Writing this kind of message down, or capturing the serial console log, is the first thing you should do when a panic happens. If the machine reboots too quickly to read, configure a serial console or a text-mode virtual console where the history is retained.

### 34.6.3 Configuring the Dump Device

For `savecore(8)` to have anything to retrieve, the kernel must know where to write the dump. FreeBSD calls this the dump device, and `dumpon(8)` is the utility that configures it.

There are two common ways to set this up. The simplest is a swap partition. During install, `bsdinstall` typically creates a swap partition large enough for kernel memory, and FreeBSD 14.3 automatically configures it as the dump device if you enabled the relevant options. You can check with:

```
dumpon -l
/dev/da0p3
```

If that command lists your swap device, you are good. If it says no dump device is configured, you can set one manually:

```
dumpon /dev/da0p3
```

To make this persistent across reboots, put it in `/etc/rc.conf`:

```
dumpdev="/dev/da0p3"
dumpon_flags=""
```

You can see the defaults for these variables in `/usr/src/libexec/rc/rc.conf`, which is the authoritative source for all default `rc.conf` values in the base system. Grep for `dumpdev=` and `dumpon_flags=` to find the relevant block.

An alternative, introduced in modern FreeBSD, is to use a file-backed dump. This avoids the need to dedicate a disk partition to dumps. See `dumpon(8)` for the exact syntax; the short version is that you can point `dumpon` at a file on a filesystem, and the kernel will dump into it at panic time. File-backed dumps are convenient for development VMs where you do not want to re-partition the disk.

A second `rc.conf` variable controls where `savecore(8)` puts retrieved dumps:

```
dumpdir="/var/crash"
savecore_enable="YES"
savecore_flags="-m 10"
```

The `-m 10` argument keeps only the most recent ten dumps, which is a reasonable default. If you are chasing a rare bug, raise the number; if disk space is tight, lower it. `savecore(8)` runs from `/etc/rc.d/savecore` during boot, before most services are up, so your dump is preserved before anything else touches `/var`.

### 34.6.4 Enabling Dumps in the Kernel

For the kernel to be willing to write a dump, it must be built with the right options. In FreeBSD 14.3, the `GENERIC` kernel is already configured for the framework pieces. If you look at `/usr/src/sys/amd64/conf/GENERIC` around the top of the file, you will see something close to:

```
options KDB
options KDB_TRACE
options EKCD
options DDB_CTF
```

`KDB` is the kernel debugger framework. `KDB_TRACE` enables automatic stack traces on panic. `EKCD` enables encrypted kernel crash dumps, which is useful when dumps contain sensitive data. `DDB_CTF` tells the build to include CTF type information for the debugger. Together these options give a fully capable dumping kernel.

Notice what is *not* in `GENERIC`: `options DDB` and `options GDB` themselves. The `KDB` framework is there, but the in-kernel debugger backend (`DDB`) and the remote `GDB` stub (`GDB`) are added by `std.debug`, which `GENERIC-DEBUG` includes. A plain `GENERIC` kernel will still write a dump on panic, but if you drop to the console during a live system, there is no `DDB` prompt to greet you.

If you are building your own kernel, either add the backends explicitly or, simpler, start from `GENERIC-DEBUG`, which enables them plus the debugging options we will need for the rest of the chapter. `GENERIC-DEBUG` lives at `/usr/src/sys/amd64/conf/GENERIC-DEBUG` and is just two lines:

```
include GENERIC
include "std.debug"
```

The `std.debug` file in `/usr/src/sys/conf/std.debug` adds `DDB`, `GDB`, `INVARIANTS`, `INVARIANT_SUPPORT`, `WITNESS`, `WITNESS_SKIPSPIN`, `MALLOC_DEBUG_MAXZONES=8`, `ALT_BREAK_TO_DEBUGGER`, `DEADLKRES`, `BUF_TRACKING`, `FULL_BUF_TRACKING`, `QUEUE_MACRO_DEBUG_TRASH`, and a few subsystem-specific debug flags. Note that `DDB` and `GDB` themselves come from `std.debug`, not from `GENERIC`; the release kernel enables `KDB` and `KDB_TRACE` but leaves the backends out unless you opt in. This is the recommended debug kernel for driver development and the kernel we will assume in the rest of the chapter unless we say otherwise.

### 34.6.5 Retrieving the Dump with `savecore`

After a panic and reboot, `savecore(8)` runs early in the boot sequence. By the time you have a shell prompt, the dump is already in `/var/crash/`:

```
ls -l /var/crash/
total 524288
-rw----- 1 root wheel 1 Apr 20 14:23 bounds
-rw-r--r-- 1 root wheel 5 Apr 20 14:23 minfree
-rw----- 1 root wheel 11534336 Apr 20 14:23 info.0
-rw----- 1 root wheel 11534336 Apr 20 14:23 info.last
-rw----- 1 root wheel 524288000 Apr 20 14:23 vmcore.0
-rw----- 1 root wheel 524288000 Apr 20 14:23 vmcore.last
```

The `vmcore.N` file is the dump itself. The `info.N` file is a textual summary of the panic, including the panic message, the backtrace, and the kernel version. Always read `info.N` first. If the message and backtrace are enough to identify the bug, you may not need to go further.

A few common problems to watch for. If `ls` shows only `bounds` and `minfree`, no dump has been captured yet. This usually means the dump device is not configured or the kernel did not manage to write to it before rebooting. Check `dumpon -l` and `re-panic`. If `savecore` logs messages about a checksum mismatch, the dump was truncated, which typically indicates the dump device was too small. If the machine never panicked cleanly but simply rebooted, the kernel likely did not have `KDB` enabled, so there was no dump machinery to invoke.

The `info.N` file is short enough to read in full. It includes the kernel version, the panic string, and the backtrace the kernel captured at panic time. On FreeBSD 14.3 it looks something like this:

```
Dump header from device: /dev/da0p3
Architecture: amd64
Architecture Version: 2
Dump Length: 524288000
```



```
Blocksize: 512
Compression: none
Dumptime: 2026-04-20 14:22:34 -0300
Hostname: devbox
Magic: FreeBSD Kernel Dump
Version String: FreeBSD 14.3-RELEASE #0: ...
Panic String: panic: bugdemo: softc missing
Dump Parity: 3142...
Bounds: 0
Dump Status: good
```

If the `Dump Status` is good, the dump is usable. If it says bad, the dump was truncated or the checksum failed.

### 34.6.6 Opening a Dump with kgdb

Once you have a dump, the next step is to open it with `kgdb`. `kgdb` is FreeBSD's version of `gdb` specialized for kernel images. It needs three things: the kernel image that produced the dump, the debug kernel image that contains symbols, and the dump file itself. On most systems all three are in predictable places:

- The running kernel: `/boot/kernel/kernel`
- The debug kernel with full symbols: `/usr/lib/debug/boot/kernel/kernel.debug`
- The dump: `/var/crash/vmcore.N`

The simplest invocation is:

```
kgdb /boot/kernel/kernel /var/crash/vmcore.0
```

or equivalently:

```
kgdb /usr/lib/debug/boot/kernel/kernel.debug /var/crash/vmcore.0
```

`kgdb` is a normal GDB session with kernel-specific tweaks. If your kernel was built with `makeoptions DEBUG=-g` (which `GENERIC-DEBUG` does), the debug symbols are included, and `kgdb` will be able to resolve every frame to source code.

When `kgdb` starts, it automatically runs a few commands:

```
(kgdb) bt
#0 __curthread () at /usr/src/sys/amd64/include/pcpu_aux.h:57
#1 doadump (textdump=...) at /usr/src/sys/kern/kern_shutdown.c:400
#2 0xffffffff80b6cf77 in kern_reboot (howto=260)
 at /usr/src/sys/kern/kern_shutdown.c:487
#3 0xffffffff80b6d472 in vpanic (fmt=..., ap=...)
```

```

 at /usr/src/sys/kern/kern_shutdown.c:920
#4 0xffffffff80b6d2c3 in panic (fmt=...)
 at /usr/src/sys/kern/kern_shutdown.c:844
#5 0xffffffff83e01234 in bugdemo_ioctl (dev=..., cmd=..., data=..., fflag=..., td=...)
 at /usr/src/sys/modules/bugdemo/bugdemo.c:142
...

```

The top frame is the panic infrastructure. The interesting frame is frame 5, `bugdemo_ioctl` at `bugdemo.c:142`. To jump to that frame:

```

(kgdb) frame 5
#5 0xffffffff83e01234 in bugdemo_ioctl (dev=..., cmd=...)
 at /usr/src/sys/modules/bugdemo/bugdemo.c:142
142 KASSERT(sc != NULL, ("bugdemo: softc missing"));

```

`kgdb` prints the source line. From here you can inspect local variables with `info locals`, look at `sc` directly with `print sc`, or list the surrounding source with `list`:

```

(kgdb) print sc
$1 = (struct bugdemo_softc *) 0x0

```

That tells us `sc` really is `NULL`, confirming the panic message. Now we can work out why, which usually means walking up the stack to find where `sc` should have been set:

```

(kgdb) frame 6

```

and so on. The sequence of `frame N`, `print VAR`, `list` is the bread-and-butter of `kgdb` analysis. It is the same conversation a `gdb` user has with any crashing program, adapted to the kernel.

### 34.6.7 Useful `kgdb` Commands

Beyond `bt` and `frame`, a handful of commands cover most debugging sessions.

- `info threads` lists all threads in the dumped system. In a modern kernel this can be hundreds of entries. Each has a number and a state.
- `thread N` switches to a specific thread, as if that thread had been the one that panicked. This is essential when a deadlock has happened and the panicking thread is not the one holding the problematic lock.
- `bt full` prints a backtrace with local variables at each frame. This is often the fastest way to see the state of a function involved in the panic.
- `info locals` shows local variables in the current frame.
- `print *SOMETHING` dereferences a pointer and prints the contents of the structure it points to.
- `list` shows the source around the current line; `list FUNC` shows the source of a function by name.

There are many more, documented in `gdb(1)`, but these are the ones a driver author reaches for most often.

### 34.6.8 Walking struct thread in a Dump

A panic backtrace answers “where did the crash happen?” but it rarely answers “who was doing what?” The kernel keeps a dense record of every live thread in `struct thread`, and once a dump is open in `kgdb` we can read that record directly. For a driver author the value is concrete: the fields of `struct thread` tell you what job this thread was performing when the kernel crashed, which lock it was waiting on, which process it belonged to, and whether it was still inside your code when the panic occurred.

`struct thread` is defined in `/usr/src/sys/sys/proc.h`. It is a large structure, so rather than read every field, we focus on a small set that matters most for driver debugging. In `kgdb` the quickest way to see those fields is to take the current thread and dereference it:

```
(kgdb) print *(struct thread *)curthread
```

On the panicking CPU `curthread` is already correct, but you can also reach a specific thread from the `info threads` listing. `kgdb` numbers every thread sequentially. Once you know the number, `thread N` switches context, and from there `print $td` (or `print *(struct thread *)0xADDR` if you have the raw address) prints the structure.

The fields to know are these. `td_name` is a short, readable name for the thread, often set by `kthread_add(9)` or by the userspace program that spawned it. When a driver creates its own kernel thread, this is the name that shows up. `td_tid` is the numeric thread identifier the kernel assigns; `ps -H` in userspace shows the same number. `td_proc` is a pointer to the owning process, which gives us access to the larger `struct proc` for more context. `td_flags` carries the `TDF_*` bit field that records scheduler and debugger state; the definitions live next to the structure in `/usr/src/sys/sys/proc.h`, and many panics can be partially explained by reading those bits. `td_lock` is the spin mutex that currently protects this thread’s scheduler state. It is almost always a CPU-local lock in a running kernel; in a dump, a `td_lock` that points to an unexpected address is a strong hint that something corrupted the scheduler view of this thread.

Two more fields are decisive when the panic involves sleeping or waiting. `td_wchan` is the “wait channel,” the kernel address the thread is sleeping on. `td_wmesg` is a short human-readable string describing why (for example, “biord” for a thread waiting on a buf read, or “select” for a thread inside `select(2)`). If the panic happened while threads were sleeping, these two fields tell you what each thread was waiting for. `td_state` is the `TDS_*` state value (defined just below `struct thread`); it tells you whether the thread was running, runnable, or inhibited at the moment of the crash.

For lock bugs specifically, `td_locks` counts the non-spin locks currently held by the thread, and `td_lockname` records the name of the lock the thread is currently blocked on, if any. If a thread panics with a non-zero `td_locks`, that thread was holding one or more sleep locks at the time of the crash: useful context when the panic is a mutex not owned or `Lock (sleep mutex) ... is not sleepable` message.

A short `kgdb` session that pulls these fields out might look like this:

```
(kgdb) thread 42
[Switching to thread 42 ...]
(kgdb) set $td = curthread
(kgdb) print $td->td_name
$2 = "bugdemo_worker"
```

```
(kgdb) print $td->td_tid
$3 = 100472
(kgdb) print $td->td_state
$4 = TDS_RUNNING
(kgdb) print $td->td_wmesg
$5 = 0x0
(kgdb) print $td->td_locks
$6 = 1
(kgdb) print $td->td_proc->p_pid
$7 = 0
(kgdb) print $td->td_proc->p_comm
$8 = "kernel"
```

Reading this: thread 42 was a kernel thread called `bugdemo_worker`, running when the panic hit, not sleeping on anything (`td_wmesg` is `NULL`), and it was still holding exactly one sleep lock. The owning process is the kernel proc with pid 0 and command name `kernel`, which is the expected owner of kernel-only threads. The interesting fact is `td_locks == 1`, because it tells us the thread held a lock at panic time; a follow-up with `show alllocks` in DDB, or with `show lockedvnods` if file locks are relevant, would pinpoint which.

### 34.6.9 Walking struct proc in a Dump

Each thread belongs to a `struct proc`, defined next to `struct thread` in `/usr/src/sys/sys/proc.h`. `struct proc` carries the process-wide context: identity, credentials, address space, open files, parent relationship. For driver bugs, a handful of these fields are particularly useful.

`p_pid` is the process identifier, the same number userspace sees in `ps`. `p_comm` is the process command name, truncated to `MAXCOMLEN` bytes. Together they tell you which userspace process triggered the kernel path that panicked. `p_state` is the `PRS_*` process state, letting you distinguish a newly-forked process, a running one, and a zombie. `p_nthreads` tells you how many threads this process has; for a multithreaded userland program that called into your driver, the count can be surprising. `p_flag` holds the `P_*` flag bits, which encode properties like tracing, accounting, and single-threading; `/usr/src/sys/sys/proc.h` documents each bit near the flag block.

Three pointers give you the larger context. `p_ucred` references the process credentials, useful when the panic might be tied to a privilege check your driver performed. `p_vmspace` points to the address space, which matters when the panic involves a user pointer that turned out to belong to an unexpected process. `p_pptr` points to the parent process; walking this chain with `p_pptr->p_pptr` eventually leads to `initproc`, the ancestor of every userspace process.

A short walk from a thread to its process looks like this in `kgdb`:

```
(kgdb) set $p = curthread->td_proc
(kgdb) print $p->p_pid
$9 = 3418
(kgdb) print $p->p_comm
```

```
$10 = "devctl"
(kgdb) print $p->p_state
$11 = PRS_NORMAL
(kgdb) print $p->p_numthreads
$12 = 4
(kgdb) print $p->p_flag
$13 = 536871424
```

Now we know the panic happened while a userspace `devctl` process with pid 3418 was running, that the process had four threads, and that its flag bits decoded through the `P_*` constants in `/usr/src/sys/sys/proc.h` will tell us whether it was being traced, accounted for, or in the middle of an exec. The flag integer on its own looks opaque, but in kgdb you can let the enum-like `P_*` macros do the decoding by casting or by using `info macro P_TRACED`.

For drivers that expose a character device, `p_fd` is also worth knowing. It points to the file descriptor table of the process that called into your driver, and in an advanced session you can walk it to find which descriptor the call came in on. That is usually more than a first-pass crash analysis needs, but the mechanism is worth remembering for the rare bug that depends on how userspace had the device open.

Between `struct thread` and `struct proc`, you can reconstruct an astonishing amount of context from a dump that at first glance only shows a panic message and a backtrace. The cost is reading `/usr/src/sys/sys/proc.h` once carefully; after that, the same vocabulary is available to you in every debugging session for the rest of your career.

### 34.6.10 Using kgdb Against a Live Kernel

So far we have treated kgdb as a post-mortem tool: open a dump, explore it offline, think at your own pace. kgdb also has a second mode, where it attaches to a running kernel through `/dev/mem` rather than to a saved dump. This mode is capable, but it is also the most easily misused tool in the whole debugging toolbox, so we will discuss it with explicit warnings.

The invocation looks almost identical to the post-mortem form, except that the “core” is `/dev/mem`:

```
kgdb /boot/kernel/kernel /dev/mem
```

What actually happens is that kgdb uses the `libkvm` library to read kernel memory through `/dev/mem`. The interface is documented in `/usr/src/lib/libkvm/kvm_open.3`, which makes the distinction clear: the “core” argument can be either a file produced by `savecore(8)` or `/dev/mem`, and in the latter case the currently running kernel is the target.

This is genuinely useful. You can inspect global variables, walk lock graphs, look at in-flight I/O, and confirm whether a `sysctl` you just set has taken effect. You can do this without rebooting, without interrupting service, and without having to reproduce a crash. On a development system that is hosting a long-running test, it is often the fastest way to answer “what is the driver actually doing right now?”

The risks are real. First, the kernel is running while you are reading. Data structures change under you. A linked list you start walking may have an entry removed halfway through; a counter you print may have been

incremented between the moment you asked for it and the moment `kgdb` printed it; a pointer you follow may be reassigned before the dereference completes. Unlike a dump, you are reading a moving target, and you will sometimes see states that are transiently inconsistent.

Second, `kgdb` on a live kernel is strictly read-only in practical use. You can read memory, print structures, and walk data, but you must not write to kernel memory through this path. The `libkvm` interface does not provide locking or barriers, and an uncoordinated write would race with the kernel itself. Treat every operation through `/dev/mem` as inspection, never modification. If you want to change running kernel state, use `sysctl(8)` or `sysctl(3)`, or load a module, or use DDB from the console. Those mechanisms are designed to coordinate with the rest of the kernel; raw writes through `/dev/mem` are not.

Third, the perturbation is not zero. Reading through `/dev/mem` can cause TLB traffic, and on large structures the cost is measurable. If you are also profiling, attribute the noise accordingly.

Finally, access to `/dev/mem` requires root privilege, for obvious reasons: anything that can read `/dev/mem` can read any secret the kernel has ever held. On production systems, restricting this is a security concern, and the policy around who may run `kgdb` against a live kernel should reflect that.

Given those cautions, the guidance is straightforward. Prefer a crash dump for any session where you want to take your time, where you want to share the state with a colleague, or where consistency matters. Prefer a live `kgdb` session for quick, read-only glances at a running system where the question is small and the cost of reboot would be high. When in doubt, take a dump with `sysctl debug.kdb.panic=1` (if the system is expendable) or `dumpon` and a deliberate triggering event, and do your analysis on the frozen snapshot. The snapshot will still be there tomorrow; the running kernel will not.

### 34.6.11 A Note on Symbols and Modules

When the panicking driver is a loadable module, `kgdb` also needs the debug information for the module. If the module is in `/boot/modules/bugdemo.ko` and was built with `DEBUG_FLAGS=-g`, the debug symbols are embedded. `kgdb` will load them automatically when it resolves frames in that module.

If the module lives somewhere nonstandard, you may need to tell `kgdb` where to find its debug info:

```
(kgdb) add-symbol-file /path/to/bugdemo.ko.debug ADDRESS
```

where `ADDRESS` is the module's load address, which you can find in `kldstat(8)` output. In practice this is rarely needed on a modern FreeBSD system, because `kgdb` looks in the right places by default.

What you do need to avoid is mixing kernels. If the running kernel and the debug kernel came from different builds, symbols will not match and `kgdb` will show confusing or wrong information. Rebuild both from the same source tree, or keep matched pairs. On a development system this is usually not a problem because you build and install both together.

### 34.6.12 Closing Thoughts on Dumps

The crash dump is valuable because it preserves the kernel's state at the moment of the panic. Unlike a running system, where every read perturbs the state, a dump is a frozen snapshot. You can examine it as long as you like, come back to it tomorrow, share it with a colleague, or diff the state against the source. Even once you

have moved on to other bugs, a dump from an interesting failure is worth keeping, because it is often the only record of that exact sequence of events.

With panic mechanics and dump analysis behind us, we can move on to the kernel configuration choices that make debugging actually comfortable. That is the topic of Section 4.

## 34.7 4. Building a Debug-Friendly Kernel Environment

Everything we have learned so far depends on having the right kernel options enabled. A stock `GENERIC` kernel is a production configuration. It is optimized for speed, ships no debug information, and does not include the checks that catch many driver bugs. For the work of this chapter, we want the opposite: a kernel that is slow but thorough, that carries full debug symbols, and that actively looks for bugs rather than trusting the driver to behave. FreeBSD calls this `GENERIC-DEBUG`, and setting it up is the topic of this section.

We will walk through building and installing a debug kernel, then look at each of the interesting options in detail, including the debugger backends (`DDB`, `GDB`), the invariant checks (`INVARIANTS`, `WITNESS`), the memory debuggers (`DEBUG_MEMGUARD`, `DEBUG_REDZONE`), and the console controls that let you enter the debugger from the keyboard.

### 34.7.1 Building `GENERIC-DEBUG`

On a FreeBSD 14.3 system with `/usr/src/` populated, building a debug kernel is a three-command operation. From `/usr/src/`:

```
make buildkernel KERNCONF=GENERIC-DEBUG
make installkernel KERNCONF=GENERIC-DEBUG
reboot
```

The `buildkernel` step takes longer than a release build because debug information is generated and many more checks are compiled in. On a modest four-core VM it typically takes twenty to thirty minutes. `installkernel` places the result in `/boot/kernel/` and keeps the previous kernel in `/boot/kernel.old/`, which is a safety net if the new kernel fails to boot.

After reboot you can confirm the running kernel with `uname -v`:

```
uname -v
FreeBSD 14.3-RELEASE-p2 #0: ...
```

The `#0` indicates a locally built kernel. You can also check that the debug options are active by reading `sysctl` debug entries, which we will return to shortly.

### 34.7.2 What `GENERIC-DEBUG` Enables

As we saw in Section 3, `GENERIC-DEBUG` is a thin configuration that simply includes `GENERIC` and `std.debug`. The interesting content is in `std.debug`, which is worth reading in full because it documents the kernel's opinion

on what good debug options look like. In a FreeBSD 14.3 tree the file is at `/usr/src/sys/conf/std.debug`, and the core options are:

```
options BUF_TRACKING
options DDB
options FULL_BUF_TRACKING
options GDB
options DEADLKRES
options INVARIANTS
options INVARIANT_SUPPORT
options QUEUE_MACRO_DEBUG_TRASH
options WITNESS
options WITNESS_SKIPSPIN
options MALLOC_DEBUG_MAXZONES=8
options VERBOSE_SYSINIT=0
options ALT_BREAK_TO_DEBUGGER
```

Plus a handful of subsystem-specific debug flags for networking, USB, HID, and CAM that we do not need to dwell on. Let us look at each of the driver-relevant options in turn.

Note one thing that `std.debug` does *not* contain: `makeoptions DEBUG=-g`. That line lives in `GENERIC` itself, near the top of `/usr/src/sys/amd64/conf/GENERIC`. A release `GENERIC` kernel is already built with `-g`, because the release engineering process wants the debug information available even when `INVARIANTS` and `WITNESS` are off. `GENERIC-DEBUG` inherits this through its `include "GENERIC"`.

### 34.7.3 `makeoptions DEBUG=-g`

This passes `-g` to the compiler for every kernel file, producing a kernel with full DWARF debug information. `kgdb` uses this debug information to map from addresses back to source lines. Without `-g`, `kgdb` can still show function names, but it cannot show the source line where a crash happened, and `print someVariable` becomes `print *(char *)0xffffffff...` with no symbolic names.

The cost is that the kernel binary is larger. On amd64 a debug `GENERIC-DEBUG` kernel is several times the size of a non-debug `GENERIC` kernel. For a development VM this does not matter. For a production system it is often the reason to keep debug information in a separate file (`/usr/lib/debug/boot/kernel/kernel.debug`) while the running kernel is stripped.

### 34.7.4 `INVARIANTS` and `INVARIANT_SUPPORT`

We met these in Section 2. `INVARIANTS` activates `KASSERT` and a number of other runtime checks scattered through the kernel. Functions throughout `/usr/src/sys/` have `#ifdef INVARIANTS` blocks that check things like “this list is well-formed,” “this pointer points into a valid zone,” or “this reference count is nonzero.” With `INVARIANTS` enabled, these checks fire at runtime. Without it, they are compiled out.

The checks cost CPU cycles. As a rough order-of-magnitude figure on typical FreeBSD 14.3-amd64 hardware, a busy `INVARIANTS` kernel runs roughly five to twenty percent slower than a release kernel, and sometimes more



on allocation-heavy workloads. This is why `INVARIENTS` is not enabled in `GENERIC`. For driver development, this overhead is worth accepting in exchange for the bugs it catches. See Appendix F for a reproducible workload that measures this ratio on your own hardware.

`INVARIANT_SUPPORT` compiles in the helper routines that assertions call, without activating the assertions in base kernel code. As noted earlier, it allows modules built with `INVARIENTS` to load against kernels without `INVARIENTS`. You almost always want both.

### 34.7.5 WITNESS: Lock Order Verifier

`WITNESS` is one of the most effective debugging tools in the FreeBSD arsenal. It tracks every lock operation and every lock dependency in the kernel, and it fires a warning if it ever sees a lock order that could deadlock. Because deadlocks are a class of bug that is extremely hard to catch any other way, `WITNESS` is indispensable for any driver that takes more than one lock.

The way `WITNESS` works is worth understanding. Every time a thread acquires a lock, `WITNESS` notes which other locks the thread already holds. From these observations it builds a lock order graph: “lock A has been seen held before lock B,” and so on. If the graph ever contains a cycle, that is a potential deadlock, and `WITNESS` prints a warning on the console with backtraces for the offending acquisitions.

The output looks something like this:

```
lock order reversal:
 1st 0xffffffff80003abc000 bugdemo_sc_mutex (bugdemo_sc_mutex) @
 ↪ /usr/src/sys/modules/bugdemo/bugdemo.c:203
 2nd 0xffffffff80003def000 sysctl_lock (sysctl_lock) @
 ↪ /usr/src/sys/kern/kern_sysctl.c:1842
stack backtrace:
#0 kdb_backtrace+0x71
#1 witness_checkorder+0xc95
#2 __mtx_lock_flags+0x8f
...
```

Read this as: your driver’s `bugdemo_sc_mutex` was acquired first, and then later another thread was observed taking `sysctl_lock` first and `bugdemo_sc_mutex` second. This is a potential deadlock, because enough concurrent activity could make the two threads wait for each other. The fix is always the same: establish a consistent lock order across all paths that take both locks, and stick to it.

`WITNESS` is not cheap. It adds bookkeeping to every lock acquisition and release. In our lab environment, on a busy kernel running a lock-heavy workload, the overhead can approach twenty percent; the exact figure varies with the amount of locking the workload does. But the bugs it finds are the kind that destroy production uptime when they slip through, so the investment is worth it in development. See Appendix F for a reproducible workload that isolates this overhead against a baseline kernel.

`WITNESS_SKIPSPIN` turns off `WITNESS` on spin mutexes. Spin locks are typically short-lived and performance-critical, so checking them adds overhead where it matters most. The default is to check them anyway, but `std.debug` disables that check to keep the kernel usable. You can re-enable it if you are specifically hunting spin-lock bugs.

### 34.7.6 A Worked Race-Condition Walkthrough: Lock-Order Bug in bugdemo

Reading about WITNESS in the abstract is one thing; catching a real lock-order reversal in a driver you wrote is another. This subsection walks through a complete cycle: we introduce a deliberate ordering bug into bugdemo, run it on a GENERIC-DEBUG kernel, read the WITNESS report, and fix the bug. The walkthrough is short, but the pattern repeats in every deadlock you will ever debug.

Assume our bugdemo driver has grown two locks as it gained features. `sc_mtx` protects the per-unit state, and `cfg_mtx` protects a configuration blob shared across units. Most of the driver already takes them in the order “state first, config second,” which is a reasonable choice and which the author has followed in `bugdemo_ioctl` and in the read/write entry points. But a recent `sysctl` handler, written in a hurry, acquired the config lock first to validate a value and then reached for the state lock to apply it. In source, the two relevant excerpts look like this:

```
/* bugdemo_ioctl: established ordering, state then config */
mtx_lock(&sc->sc_mtx);
/* inspect per-unit state */
mtx_lock(&cfg_mtx);
/* adjust shared config */
mtx_unlock(&cfg_mtx);
mtx_unlock(&sc->sc_mtx);
```

```
/* bugdemo_sysctl_set: new path, config then state */
mtx_lock(&cfg_mtx);
/* validate new value */
mtx_lock(&sc->sc_mtx);
/* propagate into per-unit state */
mtx_unlock(&sc->sc_mtx);
mtx_unlock(&cfg_mtx);
```

Both paths individually are fine. The problem is that taken together they form a cycle. If thread A enters `bugdemo_ioctl` and acquires `sc_mtx`, and thread B concurrently enters `bugdemo_sysctl_set` and acquires `cfg_mtx`, each thread now waits on the lock the other holds. That is a classic AB-BA deadlock. It may not trigger on every run; it depends on timing. WITNESS is the tool that refuses to wait for a rare production failure to discover it.

On a GENERIC-DEBUG kernel the reversal is caught the first time both orderings are observed, even if no actual deadlock has occurred yet. The console message has a specific shape. Using the format emitted by `witness_output` in `/usr/src/sys/kern/subr_witness.c`, which prints pointer, lock name, witness name, lock class, and source location for each lock involved, the real report looks like this:

```
lock order reversal:
 1st 0xfffff80012345000 bugdemo sc_mtx (bugdemo sc_mtx, sleep mutex) @
 ↪ /usr/src/sys/modules/bugdemo/bugdemo.c:412
 2nd 0xfffff80012346000 bugdemo cfg_mtx (bugdemo cfg_mtx, sleep mutex) @
 ↪ /usr/src/sys/modules/bugdemo/bugdemo.c:417
```

```
lock order bugdemo cfg_mtx -> bugdemo sc_mtx established at:
#0 witness_checkorder+0xc95
#1 __mtx_lock_flags+0x8f
#2 bugdemo_sysctl_set+0x7a
#3 sysctl_root_handler_locked+0x9c
...
stack backtrace:
#0 kdb_backtrace+0x71
#1 witness_checkorder+0xc95
#2 __mtx_lock_flags+0x8f
#3 bugdemo_ioctl+0xd4
...
```

Each 1st and 2nd line carries four pieces of information. The pointer (0xfffff80012345000) is the address of the lock object in kernel memory. The first string is the instance name, set when the lock was initialized. The two strings in parentheses are the WITNESS name for the lock class and the lock class itself, in this case `sleep mutex`. The path and line are where the lock was last acquired along this reversed ordering. The block after `lock order ... established at:` shows the earlier backtrace that first taught WITNESS the now-violated ordering, and the final `stack backtrace` shows the current call path that violates it.

Reading all of that, the diagnosis is immediate. The driver has established `sc_mtx -> cfg_mtx` in its normal paths, and `bugdemo_sysctl_set` has just taken `cfg_mtx -> sc_mtx`. Both paths are ours. The fix is to pick one ordering (here, the established one) and rewrite the offending path to match:

```
/* bugdemo_sysctl_set: corrected to follow house ordering */
mtx_lock(&sc->sc_mtx);
mtx_lock(&cfg_mtx);
/* validate new value and propagate in one atomic window */
mtx_unlock(&cfg_mtx);
mtx_unlock(&sc->sc_mtx);
```

If the locked region needs to be narrower, a common pattern is to read state under `sc_mtx`, drop it, validate without holding any lock, and then reacquire in the house order to apply. Either way, the order is fixed at the driver level, not at the call site. A useful habit is to document the order in a comment near the lock declarations so future contributors do not have to rediscover it.

After the fix, rebuilding `bugdemo` on the same debug kernel and rerunning the triggering test produces no further WITNESS output. If a reversal reappears, WITNESS also supports querying the graph interactively with `show all_locks` in DDB, which can show the current state even without a full reversal report; for deeper introspection, the source of `/usr/src/sys/kern/subr_witness.c` is the authoritative explanation of both the bookkeeping and the report format.

### 34.7.7 The Same Bug Seen Through `lockstat(1)`

WITNESS tells you that an ordering is wrong. It does not tell you how often each lock is actually contended, how long each acquisition waits, or which callers are pushing on a given lock hardest. Those are questions about

contention, not correctness, and `lockstat(1)` is the tool for them.

`lockstat(1)` is a DTrace-backed profiler for kernel locks. It works by instrumenting the entry and exit points of lock primitives and reporting summaries, including spin time on adaptive mutexes, sleep time on sx locks, and hold time when asked. The classical invocation is `lockstat sleep N`, which gathers data for N seconds and then prints a summary.

If we run the buggy `bugdemo` under a workload that stresses both paths (a small userspace program that opens several unit nodes and simultaneously twiddles the `sysctl` in a tight loop), and profile with `lockstat` for five seconds, the output on a FreeBSD system looks roughly like this:

```
lockstat sleep 5
```

```
Adaptive mutex spin: 7314 events in 5.018 seconds (1458 events/sec)
```

| Count | indv | cuml | rcnt | nsec | Lock            | Caller                          |
|-------|------|------|------|------|-----------------|---------------------------------|
| 3612  | 49%  | 49%  | 0.00 | 4172 | bugdemo sc_mtx  | bugdemo_ioctl+0xd4              |
| 2894  | 40%  | 89%  | 0.00 | 3908 | bugdemo cfg_mtx | bugdemo_sysctl_set+0x7a         |
| 412   | 6%   | 95%  | 0.00 | 1205 | bugdemo sc_mtx  | bugdemo_read+0x2f               |
| 220   | 3%   | 98%  | 0.00 | 902  | bugdemo cfg_mtx | bugdemo_ioctl+0xe6              |
| 176   | 2%   | 100% | 0.00 | 511  | Giant           | sysctl_root_handler_locked+0x4d |

```
Adaptive mutex block: 22 events in 5.018 seconds (4 events/sec)
```

| Count | indv | cuml | rcnt | nsec   | Lock            | Caller                  |
|-------|------|------|------|--------|-----------------|-------------------------|
| 14    | 63%  | 63%  | 0.00 | 184012 | bugdemo sc_mtx  | bugdemo_sysctl_set+0x8b |
| 8     | 36%  | 100% | 0.00 | 41877  | bugdemo cfg_mtx | bugdemo_ioctl+0xe6      |

Each table follows the same column convention: `Count` is how many events of this type were seen, `indv` is the percentage of events in this class, `cuml` is the running total, `rcnt` is the average reference count (always 1 for mutexes), `nsec` is the average duration in nanoseconds, and the last two columns identify the lock instance and the caller. The header line `Adaptive mutex spin` indicates contention that was resolved by a short spin; `Adaptive mutex block` indicates contention that actually forced a thread to sleep on the mutex. Those headers, and the column layout, are standard `lockstat` output; the format is documented in `/usr/src/cddl/contrib/opensolaris/cmd/lockstat/lockstat.1` along with worked examples at the end of that man page.

Two things are worth noting. First, both `bugdemo sc_mtx` and `bugdemo cfg_mtx` appear in both directions of the table: the `sysctl` path blocked on `sc_mtx` (line 1 of the block table), and the `ioctl` path blocked on `cfg_mtx` (line 2). That is the contention signature of the same ordering bug WITNESS reported, seen from the other side. WITNESS told us the ordering was unsafe; `lockstat` tells us that under this workload the unsafe ordering is also costing real time.

Second, after we apply the fix from the previous subsection, `lockstat` becomes a validation tool: rerun with the same workload and the `Adaptive mutex` block table should shrink dramatically, because the mutual wait between the two paths is gone. If it does not shrink, we have fixed the ordering but created a pure contention problem, and the next step is to narrow the critical section rather than to change the order.

Useful `lockstat` options beyond the default include `-H` to watch hold events (how long locks are held, not just contended), `-D N` to show only the top `N` rows per table, `-s 8` to include eight-frame stack traces with each row, and `-f FUNC` to filter on a single function. For driver work, `lockstat -H -s 8 sleep 10` while a targeted test runs is a remarkably productive default.

### 34.7.8 Reading WITNESS and lockstat Together

`WITNESS` and `lockstat` are complementary. `WITNESS` is a correctness tool: it detects bugs that will eventually produce a deadlock, regardless of whether the current workload happens to hit them. `lockstat` is a performance tool: it quantifies how much current traffic is touching each lock and how long the traffic waits. The same driver path often shows up in both, and the two views together are frequently decisive.

A useful discipline when a driver grows past its first lock is to make both tools part of the routine. Run `GENERIC-DEBUG` during development so `WITNESS` sees every new code path the moment it executes. Periodically run `lockstat` on a realistic workload to see whether any of your locks are becoming bottlenecks even when their ordering is sound. A lock that passes `WITNESS` and shows low `lockstat` contention is a lock you can mostly stop thinking about. A lock that passes `WITNESS` but dominates `lockstat` output is a performance problem waiting for a refactor, not a correctness bug. A lock that fails `WITNESS` is a bug regardless of what `lockstat` says.

With that framework in mind, we can continue looking at the other debug-kernel options that surface different classes of bug.

### 34.7.9 MALLOC\_DEBUG\_MAXZONES

FreeBSD's kernel memory allocator (`malloc(9)`) groups similar allocations into zones for speed. `MALLOC_DEBUG_MAXZONES=8` increases the number of zones used by `malloc`, which spreads allocations across more distinct memory regions. The practical effect is that use-after-free and invalid free bugs are more likely to land in a different zone than the original allocation, making them more detectable.

This is a low-cost option. It is always on in debug kernels.

### 34.7.10 ALT\_BREAK\_TO\_DEBUGGER and BREAK\_TO\_DEBUGGER

These two options control how the user enters the kernel debugger from the console. `BREAK_TO_DEBUGGER` enables the traditional `Ctrl-Alt-Esc` or serial `BREAK` sequence. `ALT_BREAK_TO_DEBUGGER` enables an alternative sequence, typed as `CR ~ Ctrl-B`, which is useful over network consoles (`ssh`, `virtio_console`, and so on) where sending a real `BREAK` is awkward.

`GENERIC` ships with `BREAK_TO_DEBUGGER` enabled. `GENERIC-DEBUG` adds `ALT_BREAK_TO_DEBUGGER`. If you are on a serial console, either sequence will drop you into `DDB`. From `DDB` you can inspect kernel state, set breakpoints, and optionally continue execution or panic.

This is an important convenience during development. A driver that hangs the system without panicking can be investigated by dropping into the debugger on command.

### 34.7.11 DEADLKRES: The Deadlock Detector

DEADLKRES enables the deadlock resolver, which is a periodic thread that watches for threads stuck in uninterruptible wait for too long. If it finds any, it prints a warning and optionally panics. This complements WITNESS by catching deadlocks that WITNESS did not predict, which happens when the lock graph is not traversable statically (for example, when locks are acquired by address through a generic locking API).

DEADLKRES has some false positives in practice, particularly for long-lived operations like filesystem I/O under heavy load. Reading the warning and deciding whether it is a real deadlock is part of the debugging skill this chapter is teaching.

### 34.7.12 BUF\_TRACKING

BUF\_TRACKING records a short history of operations on each buffer in the buffer cache. When a corruption is found, the buffer's history can be printed, showing which code paths touched it and in what order. This is useful for storage driver bugs but less commonly needed in other drivers.

### 34.7.13 QUEUE\_MACRO\_DEBUG\_TRASH

The `queue(3)` macros (`LIST_`, `TAILQ_`, `STAILQ_`, and so on) are used pervasively in the kernel for linked lists. When an element is removed from a list, the usual behavior is to leave the element's pointers alone. `QUEUE_MACRO_DEBUG_TRASH` overwrites them with recognizable garbage values. Any later attempt to dereference those pointers will crash in a recognizable way, rather than silently corrupting the list.

This is a cheap option and catches a very common class of bug: forgetting to remove an element from a list before freeing it, then finding the list corrupted later.

### 34.7.14 Memory Debuggers: `DEBUG_MEMGUARD` and `DEBUG_REDZONE`

Two more options that deserve attention are `DEBUG_MEMGUARD` and `DEBUG_REDZONE`. They are not part of `std.debug` but are commonly added for memory debugging sessions.

`DEBUG_MEMGUARD` is a specialized allocator that can be dropped in for particular `malloc(9)` types. It backs each allocation with a separate page or set of pages, marks the pages around the allocation as inaccessible, and unmaps the allocation on free. Any access beyond the bounds of the allocation, or any access after free, causes a page fault that is trivial to diagnose. The trade-off is that every allocation costs a full page of virtual memory plus kernel bookkeeping, so you normally turn `DEBUG_MEMGUARD` on for one specific `malloc` type at a time.

The relevant header is `/usr/src/sys/vm/memguard.h`, and the configuration appears in `/usr/src/sys/conf/NOTES` on the options `DEBUG_MEMGUARD` line. We will use `memguard(9)` in detail in Section 6.

`DEBUG_REDZONE` is a lighter-weight memory debugger that places guard bytes before and after each allocation. When the allocation is freed, the guard bytes are checked, and any corruption is reported. It does not catch

use-after-free but is very good at catching buffer overruns and underruns. See the options `DEBUG_REDZONE` line in `/usr/src/sys/conf/NOTES` for the configuration.

Both `DEBUG_MEMGUARD` and `DEBUG_REDZONE` cost memory. For a debug kernel on a development VM, both are often enabled. For a large production server, neither is.

### 34.7.15 KDB, DDB, and GDB Together

We have referenced these three options throughout this chapter. Let us pin down the distinction, because it confuses many beginners.

KDB is the kernel debugger framework. It is the plumbing. It defines entry points that the rest of the kernel calls when a panic or a break-to-debugger event happens. It also defines an interface for backends.

DDB and GDB are two such backends. DDB is the in-kernel interactive debugger. When you hit `KDB_ENTER` and DDB is the selected backend, you are dropped into an interactive prompt on the console. DDB has a small set of commands: `bt`, `show`, `print`, `break`, `step`, `continue`, and a few others. It is primitive but self-contained: no other machine is needed.

GDB is the remote backend. When you hit `KDB_ENTER` and GDB is the selected backend, the kernel waits for a remote GDB client to attach over a serial line or a network connection. The client runs on a different machine and sends commands over a protocol called the GDB remote serial protocol. This is much more flexible because you have full `gdb` on the client side, but it requires a second machine (or another VM) and a connection between the two.

In practice, you enable both backends and switch between them at run time. `sysctl debug.kdb.current_backend` names the active backend. `sysctl debug.kdb.supported_backends` lists all compiled-in backends. You can set `debug.kdb.current_backend` to `ddb` or `gdb` depending on what kind of session you want. This is a useful convenience, because the overhead of having both compiled in is negligible compared to the benefit of flexibility.

The KDB support in `GENERIC` is enough for most panics. We will use GDB in Section 7 when we talk about remote debugging.

### 34.7.16 KDB\_UNATTENDED

One more option worth mentioning is `KDB_UNATTENDED`. This causes the kernel to skip entry into the debugger on panic and go straight to the dump and reboot. In production systems without anyone watching the console, this is a sensible default; there is no point waiting indefinitely for a debugger interaction that will never come. In development, you usually want the opposite: to stay in DDB after a panic so you can investigate before the state is lost to reboot. Set this option via the `debug.debugger_on_panic` `sysctl` at runtime, or options `KDB_UNATTENDED` in a kernel config.

### 34.7.17 CTF and Debug Info Paths

One last piece of the debug environment is CTF, the Compact C Type Format. CTF is a compressed representation of type information that DTrace uses to understand kernel structures. `GENERIC` includes options

DDB\_CTF, which tells the build to generate CTF information for the kernel. On a debug kernel, CTF information lets DTrace print structure fields by name instead of hex offsets, which makes its output dramatically more useful.

You can confirm CTF is present with `ctfdump`:

```
ctfdump -t /boot/kernel/kernel | head
```

If this produces output, the kernel has CTF. If not, either the build did not include DDB\_CTF or the CTF generation tool (`ctfconvert`) was not installed. In FreeBSD 14.3 both are standard.

For modules, you need to build with `WITH_CTF=1` in your environment (or passed to `make`) to get CTF information for the module. This is what lets DTrace understand the structures your driver defines.

### 34.7.18 Confirming Your Debug Kernel

When you first boot into a debug kernel, spend a minute verifying that the options you care about are actually active. Useful sysctls:

```
sysctl debug.kdb.current_backend
debug.kdb.current_backend: ddb
sysctl debug.kdb.supported_backends
debug.kdb.supported_backends: ddb gdb
sysctl debug.debugger_on_panic
debug.debugger_on_panic: 1
sysctl debug.ddb.
debug.ddb.capture.inprogress: 0
debug.ddb.capture.bufsize: 0
...
```

If these print sensible values, your debug kernel is wired up. If `debug.kdb.supported_backends` lists only `ddb` but you expected `gdb`, something in your configuration is off. Go back and check that options `GDB` is in your kernel config or in `std.debug`.

### 34.7.19 Running on Top of the Debug Kernel

With the debug kernel running, the rest of the chapter's techniques become available. `KASSERT` actually fires. `WITNESS` actually complains about lock orders. `DDB` is there when you press the break-to-debugger sequence. Crash dumps include full debug information that `kgdb` can use to show source lines. You have moved from a kernel that trusts the driver to a kernel that actively helps you prove your driver is correct.

A small but meaningful consequence of running a debug kernel for driver development is that you will see bugs in your driver much earlier, before they reach the field, and you will have an easier time fixing them when they do appear. The discipline of always developing on a debug kernel, even when you are only writing simple code, is one of the habits that separates casual hobbyist drivers from drivers that are trustworthy enough for serious use.



With the environment set up, we can move on to the next class of tool: tracing. Unlike assertions, which catch failures, tracing records what happens so you can understand the shape of a bug even when it is not crashing. That is the topic of Section 5.

## 34.8 5. Tracing Driver Behavior: DTrace, ktrace, and ktr(9)

Assertions catch what is wrong. Tracing shows what is happening. When a driver misbehaves without crashing, or when you need to understand the precise order of events across multiple threads, tracing is usually the right tool. FreeBSD offers three complementary tracing facilities for kernel code: DTrace, `ktrace(1)`, and `ktr(9)`. Each has a different sweet spot, and a driver author should know when to reach for which.

Chapter 33 introduced DTrace as a performance measurement tool. Here we come back to it as a correctness debugging tool, because the same framework that can aggregate hot functions can also follow a bug through the kernel. We will also meet `ktr(9)`, the lightweight in-kernel tracing ring, and `ktrace(1)`, which traces system calls from user space.

### 34.8.1 DTrace for Correctness Debugging

DTrace is FreeBSD's production-grade dynamic tracing framework. It works by letting you attach small scripts to probe points throughout the kernel. A probe is a named point in the code that can be instrumented. When a probe fires, the script runs. If the script has useful things to record, it records them; if not, the probe is effectively free.

Chapter 33 used DTrace with the `profile` provider for CPU sampling. In this chapter we will use different providers for different purposes: `fbt` (function boundary tracing) to follow entry and exit from functions, `sdt` (statically defined tracing) to fire at explicitly placed probe points in our driver, and `syscall` to watch user-kernel transitions.

Let us take these in turn.

### 34.8.2 The fbt Provider

The `fbt` provider gives you a probe at every function entry and exit in the kernel. To list all `fbt` probes in our driver:

```
dtrace -l -P fbt -m bugdemo
```

Each function produces two probes, an entry and a return. You can attach actions to either. A common first step in debugging a new bug is simply to see what functions are being called:

```
dtrace -n 'fbt::bugdemo_*:entry { printf("%s\n", probefunc); }'
```

This prints every entry into any function in the `bugdemo` module, showing the order in which they are called. If you suspect a particular function is or is not being reached, this one-liner will tell you immediately.

For a deeper view, you can also record arguments. `fbt` probe arguments are the function's parameters, accessible as `arg0`, `arg1`, etc.:

```
dtrace -n 'fbt::bugdemo_ioctl:entry { printf("cmd=0x%lx\n", arg1); }'
```

Here `arg1` is the second parameter to `bugdemo_ioctl`, which is the `ioctl` command number. You can watch the stream of `ioctl` calls in real time.

Exit probes let you see return values:

```
dtrace -n 'fbt::bugdemo_ioctl:return { printf("rv=%d\n", arg1); }'
```

On a return probe, `arg1` is the return value. A stream of `rv=0` entries confirms success. A sudden `rv=22` (which is `EINVAL`) tells you the driver rejected a call. By combining entry and return probes you can match each call with its result.

### 34.8.3 SDT Probes: Statically Defined Tracing

`fbt` is flexible but gives you function boundaries, not semantic events. If you want a probe that fires at a specific point inside a function, representing a specific event, you use SDT. SDT probes are placed explicitly in the code. They cost essentially nothing when disabled and produce exactly the information you want when enabled.

In FreeBSD 14.3, SDT probes are defined using macros from `/usr/src/sys/sdt.h`. The key macros are:

```
SDT_PROVIDER_DEFINE(bugdemo);

SDT_PROBE_DEFINE2(bugdemo, , , cmd__start,
 "struct bugdemo_softc *", "int");

SDT_PROBE_DEFINE3(bugdemo, , , cmd__done,
 "struct bugdemo_softc *", "int", "int");
```

The naming convention is `provider:module:function:name`. The leading `bugdemo` is the provider. The two empty strings are the module and function, which we leave empty for a driver-level probe. The trailing name identifies the probe. The underscore-underscore convention in probe names is a DTrace idiom that becomes a dash in the user-facing name.

The numeric suffix on `SDT_PROBE_DEFINE` indicates how many arguments the probe takes. The string arguments are the C type names of those arguments, which DTrace uses for display.

To fire a probe in the driver:

```
static void
bugdemo_process(struct bugdemo_softc *sc, struct bugdemo_command *cmd)
{
```

```

 SDT_PROBE2(bugdemo, , , cmd__start, sc, cmd->op);

 /* ... actual work ... */

 SDT_PROBE3(bugdemo, , , cmd__done, sc, cmd->op, error);
}

```

SDT\_PROBE2 and SDT\_PROBE3 fire the corresponding probe with the given arguments.

Now in DTrace you can watch these probes:

```
dtrace -n 'sdt:bugdemo::cmd-start { printf("op=%d\n", arg1); }'
```

Notice the dash in cmd-start: DTrace converts the double underscore in the name to a dash in the probe spec. arg0 is the softc, arg1 is the op.

SDT probes are particularly useful for state transitions. If your driver has three states and you want to follow the sequence, define probes at each transition and aggregate on them:

```
dtrace -n 'sdt:bugdemo::state-change { @[arg1, arg2] = count(); }'
```

This counts how often each (from\_state, to\_state) pair occurs, giving a distribution of the state machine's behavior during your workload.

### 34.8.4 Following a Bug With DTrace

Consider a scenario. The bugdemo driver sometimes returns EIO to user space, but you cannot tell from user space which code path produced that error. With DTrace, you can walk backwards from the return to the origin:

```

dtrace -n '
fbt::bugdemo_ioctl:return
/arg1 == 5/
{
 stack();
}
'
```

arg1 == 5 checks for the return value 5, which is EIO. When the return matches, stack() prints the kernel stack trace at the point of the return. This tells you exactly which code path returned the error.

A more sophisticated version records the start time and duration:

```
dtrace -n '
fbt::bugdemo_ioctl:entry
{
 self->start = timestamp;
}

fbt::bugdemo_ioctl:return
/self->start != 0/
{
 @latency["bugdemo_ioctl", probefunc] = quantize(timestamp - self->start);
 self->start = 0;
}
'
```

This produces a latency distribution for the `ioctl`, which is useful when a bug manifests as unusual latency. The `self->` notation is DTrace's thread-local storage, scoped to the current thread.

These scripts are not full programs; they are small observations that you iterate on. The cycle of “add a probe, run the workload, read the output, refine the probe” is one of DTrace's strengths. A full debugging session might go through a dozen variations of a script before the shape of the bug becomes clear.

### 34.8.5 Understanding `ktrace(1)`

`ktrace(1)` is a different beast. It traces system calls made by a user-space process, along with their arguments and return values. It is not about the kernel's internal behavior; it is about the interface between user space and the kernel. When a user-space tool is using a driver and something odd happens, `ktrace(1)` is often the first tool to reach for, because it shows exactly what the process asked of the kernel.

To trace a program:

```
ktrace -t cnsi ./test_bugdemo
kdump
```

`ktrace` writes a binary trace file (`ktrace.out` by default), and `kdump` renders it as human-readable text. The `-t` flags select what to trace: `c` for system calls, `n` for `namei` (pathname lookups), `s` for signals, `i` for `ioctls`. For driver debugging, `i` is the most directly useful.

Sample output:

```
5890 test_bugdemo CALL ioctl(0x3, BUGDEMO_TRIGGER, 0x7fffffffef0c0)
5890 test_bugdemo RET ioctl 0
5890 test_bugdemo CALL read(0x3, 0x7fffffffef0d0, 0x100)
5890 test_bugdemo RET read 32/0x20
```

The process made two syscalls. An `ioctl` on file descriptor 3 with command `BUGDEMO_TRIGGER` succeeded. A read on the same fd returned 32 bytes. If the test fails, the trace tells you exactly what the kernel was asked for and what it returned.

Notice that `ktrace(1)` does not show internal kernel behavior. For that you need `DTrace` or `ktr(9)`. But `ktrace(1)` is the canonical way to see user-space interactions, and combined with `DTrace` it gives a complete picture.

`ktrace(1)` can also be attached to a running process:

```
ktrace -p PID
```

and detached:

```
ktrace -C
```

For a driver that is being used by a long-running daemon, this is more practical than restarting the daemon under `ktrace`.

### 34.8.6 `ktr(9)`: Lightweight In-Kernel Tracing

`ktr(9)` is FreeBSD's in-kernel trace ring. It is a ring buffer of trace entries that code can write to cheaply. Each entry includes a timestamp, the CPU number, the thread pointer, a format string, and up to six arguments. The ring is sized by the `KTR_ENTRIES` kernel config option, and its contents can be dumped from `DDB` or from user space.

`ktr(9)` is the right tool when you need very fine-grained information about timing or ordering, especially in an interrupt context where `printf` is too slow. Because each entry is small and writes are lock-free, `ktr(9)` can be used in hot paths without distorting the behavior you are trying to observe.

The macros are in `/usr/src/sys/sys/ktr.h`. The common ones are `CTR0` through `CTR6`, varying by how many arguments follow the format string. Each macro takes a class mask as its first argument, then the format string, then the values:

```
#include <sys/ktr.h>

static void
bugdemo_process(struct bugdemo_softc *sc, struct bugdemo_command *cmd)
{
 int error;

 CTR2(KTR_DEV, "bugdemo_process: sc=%p op=%d", sc, cmd->op);
 /* ... */
 CTR1(KTR_DEV, "bugdemo_process: done rv=%d", error);
}
```

CTR2 writes a two-argument entry to the trace ring. `KTR_DEV` is the class mask: the kernel decides at runtime whether entries for a given class are recorded, based on `debug.ktr.mask`. At compile time, `KTR_COMPILE` (the set of classes actually compiled in) controls which calls are emitted at all. Classes that are not in `KTR_COMPILE` disappear entirely, so you can leave calls in the source permanently without paying for them when the class is disabled.

The classes are defined in `/usr/src/sys/sys/ktr_class.h`. Common ones include `KTR_GEN` (general-purpose), `KTR_DEV` (device drivers), `KTR_NET` (networking), and many more. For a driver, you would typically pick `KTR_DEV` or, in larger subsystems, define a new bit alongside the existing ones.

To enable and view the trace ring:

```
sysctl debug.ktr.mask=0x4 # enable KTR_DEV (bit 0x04)
sysctl debug.ktr.entries
```

and dump it with:

```
ktrdump
```

`ktrdump(8)` reads the kernel's trace buffer through `/dev/kmem` and formats it. The output is a time-ordered list of entries with timestamps, CPUs, threads, and messages.

The beauty of `ktr(9)` is its low overhead. A trace entry is essentially a handful of memory writes. You can leave them in the code, compile them in to a debug kernel, and enable them at runtime when you need them. They are especially valuable for interrupt handler debugging, where `printf` would add milliseconds of delay and actually change the behavior being measured.

### 34.8.7 When to Use Which

With three tracing tools, the question is which to reach for first.

Use `DTrace` when the bug is about what the kernel is doing, when you need to aggregate across many events, when you need filtering, or when the probes can be placed dynamically. `DTrace` is the most capable of the three, but it requires a running kernel and a reasonable rate of probe firings.

Use `ktrace(1)` when the bug is about what user space is asking of the kernel, when the symptom is a wrong return value or a sequence of syscalls that does not match expectations. `ktrace(1)` is simple, fast, and immediately shows the kernel-user boundary.

Use `ktr(9)` when you need the lowest possible overhead, when the code you are tracing is in an interrupt handler, or when you want persistent trace points that can be turned on in production with minimal risk. `ktr(9)` is the most primitive of the three but also the most durable.

In practice, a debugging session often uses two or three. You might start with `ktrace(1)` to see the syscall sequence, then add `DTrace` probes to narrow down which driver function is misbehaving, then add `ktr(9)` entries to nail down timing in the interrupt path. Each tool answers a different question, and a full picture often requires all three.

### 34.8.8 Tracing and Production

A quick word on production. DTrace is production-safe in most configurations; its design specifically includes safeguards against infinite loops and against crashing the kernel from a bad probe. You can run DTrace on a busy production server without bringing it down. `kttr(9)` is also production-safe, with the caveat that enabling verbose classes costs CPU. `kttrace(1)` writes to a file and can grow unbounded if left unchecked; use with size limits.

Contrast these with crash dumps, DDB, and `memguard(9)`, which are development-only tools. The distinction matters because Section 7 will return to the question of what is safe to do on a production machine. For now, remember that tracing is among the lightest-touch techniques we have, and that is why it is often the right first step when diagnosing a live problem.

With tracing in hand, we can turn to the bugs that tracing and assertions tend to miss: memory bugs that corrupt state without producing clear symptoms until much later. That is the domain of Section 6.

## 34.9 6. Hunting Memory Leaks and Invalid Memory Accesses

Memory bugs are the most treacherous bugs a driver author faces. They are rarely visible at the moment they happen. They corrupt state quietly, accumulate across many runs, and manifest much later in ways that seem completely unrelated to the original defect. A use-after-free might surface as a corrupted structure in a different subsystem. A buffer overrun might overwrite the next allocation and show up as a bogus field value several minutes later. A small leak might drain memory over days until the kernel finally refuses an allocation and the system locks up.

FreeBSD provides a family of tools for these bugs: `memguard(9)` for use-after-free and modify-after-free detection, `redzone` for buffer overruns, guard pages in the VM layer, and `sysctls` that expose the state of the kernel memory allocator. Used together, they can transform a class of bug that was nearly impossible to find into a class that crashes immediately at the moment of misuse.

### 34.9.1 Understanding the Kernel Memory Allocator

To use these tools effectively we need a rough mental model of how the kernel allocates memory. FreeBSD has two primary allocators, both in `/usr/src/sys/kern/`:

`kern_malloc.c` implements `malloc(9)`, the general-purpose allocator. It is a thin wrapper over UMA, the Universal Memory Allocator, with accounting by `malloc` type. Every allocation is charged to a `struct malloc_type` (commonly declared with `MALLOC_DEFINE(9)`), which lets the kernel track how much memory each subsystem has used.

`subr_vmem.c` and `uma_core.c` implement the lower layers. UMA is a slab allocator: it maintains per-CPU caches and central slabs, so most allocations are very fast and contention-free. When a driver calls `malloc(9)` or `uma_zalloc(9)`, what actually happens depends on size, zone configuration, and cache state.

For debugging, the practical consequence is that a corrupted allocation might look different depending on where it landed. The same bug can produce different symptoms on different kernels or under different loads, simply because the underlying memory layout differs.

### 34.9.2 sysctl vm and kern.malloc: Observing Allocation State

Before reaching for memory debuggers, a useful first step is to look at the live state of the allocator. Two sysctls are particularly useful:

```
sysctl vm.uma
sysctl kern.malloc
```

The first dumps per-zone statistics for UMA: how many items are allocated, how many are free, how many failures have occurred, and how many pages each zone is using. The output is long, but it is text-searchable. If you suspect a leak in a particular driver type, find its zone in the output and watch it grow.

The second dumps per-type statistics for malloc(9). Each entry shows the type name, the number of requests, the amount allocated, and the high-water mark. Running the driver under a workload and comparing before and after is a simple leak-detection technique that requires no special tooling:

```
sysctl kern.malloc | grep bugdemo
bugdemo:
 inuse = 0
 memuse = 0K
```

Run a workload, re-query, and compare. If inuse rises and does not fall after the workload finishes, something is leaking.

The related vmstat(8) command has a -m flag that presents the same malloc(9) state in a more compact form:

```
vmstat -m | head
 Type InUse MemUse HighUse Requests Size(s)
 acl 0 0K - 0 16,32,64,128,256,1024
 amd6 4 64K - 4 16384
 bpf_i 0 0K - 2
 ...
```

For ongoing monitoring during a workload:

```
vmstat -m | grep -E 'bugdemo|Type'
```

gives you a periodic snapshot of a single type's footprint.

### 34.9.3 memguard(9): Hunting Use-After-Free

memguard(9) is a special allocator that can replace malloc(9) for a specific type. The idea is simple: instead of returning a piece of memory from a slab, it returns memory backed by dedicated pages. When the memory



is freed, the pages are not returned to the pool; they are unmapped, so any subsequent access faults. And the pages around the allocation are left inaccessible, so any read or write past the end of the allocation also faults. This turns use-after-free, buffer overrun, and buffer underrun bugs from silent corruptors into immediate panics with a backtrace that points directly at the misuse.

The cost is that each allocation now costs at least one full page of virtual memory (plus management overhead), and every free burns an unmapped page. For that reason, `memguard(9)` is typically turned on for a single malloc type at a time, not for everything.

The configuration involves two steps. First, the kernel must be built with options `DEBUG_MEMGUARD`, which `std.debug` does not enable by default. You add it to your kernel config:

```
include "std.debug"
options DEBUG_MEMGUARD
```

and rebuild.

Second, at runtime you tell `memguard` which malloc type to guard:

```
sysctl vm.memguard.desc=bugdemo
```

From that moment on, every allocation of type `bugdemo` goes through `memguard`. Note the type string matches the name passed to `MALLOC_DEFINE(9)` in the driver source. Typos here silently do nothing.

You can also use `vm.memguard.desc=*` to guard everything, but as noted, this is expensive. For a targeted bug hunt, guard only the type you suspect.

#### 34.9.4 A memguard Session in Action

Imagine `bugdemo` has a use-after-free bug: the driver frees a buffer when its `ioctl` completes, but then an interrupt handler reads from the same buffer a moment later. Without `memguard`, the read usually succeeds because the slab allocator has not yet reused the memory, or it returns some unrelated data that happens to have replaced the buffer. The driver gets plausible-but-wrong output, which corrupts some later state, which manifests much later as a subtle bug.

With `memguard` enabled for the driver's malloc type, the same sequence of events fires a page fault the instant the interrupt handler dereferences the freed pointer. The fault produces a panic with a backtrace through the interrupt handler. The panic message identifies the faulting address as inside a `memguard` region, and `kgdb` on the dump shows you exactly which function dereferenced the freed memory.

Compare that to the days of detective work the bug would have demanded without `memguard`, and you understand why this tool is so valuable.

#### 34.9.5 redzone: Buffer Overrun Detection

`memguard` is heavyweight. For the narrower case of buffer overruns and underruns, FreeBSD offers `DEBUG_REDZONE`, a lighter debugger that adds a few guard bytes before and after each allocation. When the allocation is freed, the guard bytes are checked. If they have been modified, `redzone` reports the corruption, including the stack at the time of allocation.

DEBUG\_REDZONE is added to the kernel config:

```
options DEBUG_REDZONE
```

Unlike memguard, it is always active once compiled in, and it applies to all allocations. Its overhead is memory, not time: each allocation grows by a few bytes.

redzone does not catch use-after-free, because the memory it guards is still within the original allocation. It does catch writes that step past the intended buffer, which is a common class of bug in drivers that compute offsets from user-provided sizes.

### 34.9.6 Guard Pages in the VM Layer

A third mechanism, available independently of memguard and redzone, is the use of guard pages around critical kernel allocations. The VM system supports allocating a memory region with inaccessible pages placed before and after it. Kernel thread stacks use this mechanism: the page below each stack is unmapped, so a runaway recursion faults instead of overwriting the adjacent allocation.

Drivers that allocate stack-like objects can use `kmem_malloc(9)` with the right flags, or can set up guard pages manually through `vm_map_find(9)`. In practice, driver code rarely does this directly; the mechanism is more commonly used by subsystems that manage their own memory regions. But it is useful to know that the capability is there, because you may see it in kernel messages and want to understand what it means.

### 34.9.7 Leak Detection in Practice

Leaks are the quietest class of memory bug. They produce no crashes, no faults, no assertions. The only symptom is that memory usage grows over time. FreeBSD gives you a few tools for finding them.

The first, as we saw, is `kern.malloc`. Snapshot before, run the workload, snapshot after, and look for types whose `inuse` grew and did not shrink. This is crude but effective for driver leaks.

The second is to add counters to your driver. If every allocation bumps a counter (9) and every free decrements it, a lingering positive value at unload time tells you the driver leaked something. A companion `sysctl` exposes the counter for inspection:

```
static counter_u64_t bugdemo_inflight;

/* in attach: */
bugdemo_inflight = counter_u64_alloc(M_WAITOK);

/* in allocation path: */
counter_u64_add(bugdemo_inflight, 1);

/* in free path: */
counter_u64_add(bugdemo_inflight, -1);
```

```
/* in unload: */
KASSERT(counter_u64_fetch(bugdemo_inflight) == 0,
 ("bugdemo: %ld buffers leaked at unload",
 (long)counter_u64_fetch(bugdemo_inflight)));
```

This idiom, counting inflight allocations explicitly, is useful in any subsystem that owns a pool of objects. The assertion at unload time fires if anything leaked, giving you an immediate report at the moment you notice the leak, not hours later.

The third tool is DTrace. If you know which malloc type is leaking but not why, a DTrace script can track every allocation and every free, accumulating the difference per stack trace:

```
dtrace -n '
fbt::malloc:entry
/arg1 == (uint64_t)&M_BUGDEMO/
{
 @allocs[stack()] = count();
}

fbt::free:entry
/arg1 == (uint64_t)&M_BUGDEMO/
{
 @frees[stack()] = count();
}
'
```

After a workload, comparing the two aggregations often reveals a code path that allocates but never frees. The stack traces point you directly to the offending call sites.

### 34.9.8 When Memory Bugs Hide

Sometimes memory bugs do not match any of these patterns. The symptom is a panic in an unrelated subsystem, with a backtrace that seems impossible. The driver looks fine on review; its allocations and frees appear balanced. Yet the kernel keeps crashing with messages about corrupt lists or invalid pointers.

The usual cause in these cases is that the driver writes past the end of a buffer into the next allocation. The next allocation belongs to someone else; your overrun silently corrupts that other subsystem's data. The crash happens when the other subsystem next touches its corrupted data, which might be soon or might be much later.

For this class of bug, the diagnostic is to enable `DEBUG_REDZONE` and watch for warnings. When redzone reports that guard bytes have been modified, the stack trace it prints for the allocation is the allocation in question, and the code that overran it is the code that was writing into that allocation. redzone's report tells you both ends of the bug.

Another trick is to enable `MALLOC_DEBUG_MAXZONES=N` with a large `N`. This spreads allocations across more zones, so a driver's allocations are less likely to share a zone with unrelated subsystems. If the symptom disappears or changes with more zones, it is a strong hint that the bug involves cross-zone corruption.

### 34.9.9 Working With DDB on Memory Bugs

When the kernel panics on a memory bug, entering DDB can help narrow down the cause. Useful DDB commands include:

- `show malloc` dumps `malloc(9)` state.
- `show uma` dumps UMA zone state.
- `show vmockk` runs a consistency check on the VM object tree.
- `show allpcpu` shows per-CPU state.

These commands produce output that helps correlate a crash with the state of the allocators at the moment of the crash. They do not replace `kgdb` analysis, but they can be faster to consult when you are already in DDB.

#### 34.9.10 Reality Check on Memory Debuggers

`memguard`, `redzone`, and their relatives are effective. They are also disruptive. They change allocator behavior, they slow the kernel down, and some of them consume memory aggressively. Leaving them on in production is not a good idea.

The right use is targeted. When a bug presents, enable the appropriate debugger, reproduce the bug, capture the evidence, and then turn the debugger off. Most of your driver development happens on a kernel with `INVARIANTS` and `WITNESS` but without `DEBUG_MEMGUARD`. `DEBUG_MEMGUARD` comes out when you are actively chasing a memory bug and goes away when you are done.

One last consideration. Some memory debuggers, notably `memguard`, change the observable behavior of the allocator in ways that can mask bugs. If a driver depends on two allocations being adjacent in memory (which it never should, but sometimes does as an accidental invariant), `memguard` will break that dependency and make the bug go away. This does not mean the bug is fixed; it means the bug is now latent. Always re-test without `memguard` after a fix, to make sure the fix is real and not an artifact of the debugger's presence.

#### 34.9.11 Wrapping Up the Memory Section

Memory bugs are the quiet killers of driver code. The patience to find them is built on a small, focused set of tools. `memguard(9)` catches use-after-free and buffer overrun directly. `redzone` catches overruns with less overhead. The `kern.malloc` and `UMA` sysctls expose the allocator state that normal code cannot see. And the discipline of counting inflight allocations in your own driver catches leaks at unload time. Put these together, and a class of bug that used to take days to find can be made to announce itself in minutes.

With the major technical tools now covered, we turn to the discipline of using them safely, particularly on systems where other people are watching. That is the content of Section 7.

## 34.10 7. Safe Debugging Practices

Every tool we have learned in this chapter has a cost, and every cost has a context in which it is acceptable. A debug kernel on a development VM is a small price for catching bugs early. The same debug kernel on a production server is a disaster in slow motion. Knowing which tools to reach for in which context is part of what separates a competent driver author from a dangerous one.

This section collects the practices that keep you out of trouble: the conventions for using each tool safely, the signs that you are about to make a mistake, and the mindset that helps you work with discipline when the stakes are high.

### 34.10.1 The Development vs Production Divide

The most important distinction in safe debugging is between a development system, where you can crash the kernel freely, and a production system, where you cannot.

On a development system, everything in this chapter is fair game. Deliberately trigger panics. Enable `DEBUG_MEMGUARD`. Load and unload the driver repeatedly. Attach `kgdb` to the live kernel. Run DTrace scripts that collect megabytes of data. The worst that can happen is that you reboot the VM, which is measured in seconds.

On a production system, the opposite posture applies. You do not enable debug options unless you have a specific, targeted reason. You do not load experimental drivers. You do not run DTrace scripts that might destabilize the probe framework. You do not break into DDB on a live system. Every intervention is preceded by a clear answer to “what do I do if this goes wrong?”

The discipline of keeping these two environments separate is the single most effective way to avoid accidentally breaking production. Have a development VM, keep the production kernel on a different partition, and never confuse the two.

### 34.10.2 What Is Safe in Production

A surprising amount of the debugging toolkit is actually safe in production, if used carefully. Here is a partial list.

DTrace scripts are generally safe in production. The DTrace framework is specifically designed with safety guarantees: probe actions cannot loop indefinitely, cannot allocate arbitrary memory, and cannot dereference arbitrary pointers without falling into a well-defined recovery path. You can run DTrace aggregations on a busy server without bringing it down. The caveats are that very high-frequency probes can consume significant CPU (a probe on every packet in a network driver is unlikely to be free), and that DTrace output can fill filesystem space if not rate-limited.

`ktrace(1)` on a specific process is safe, though it writes to a file that grows unbounded. Set a size limit or watch the file.

`ktr(9)` is safe if the relevant classes are already compiled in. Enabling a class through `sysctl debug.ktr.mask=` is safe. Compiling in a new class requires a kernel rebuild, which is a development activity.

Reading `sysctl` is always safe. `kern.malloc`, `vm.uma`, `hw.ncpu`, `debug.kdb.*`, and all the others expose state without changing anything. A production system with a sick driver can be interrogated extensively through `sysctl` alone.

### 34.10.3 What Is Unsafe in Production

A shorter list, but an important one.

Panics are unsafe. Deliberately crashing a production server is only acceptable as a last resort when the server is already irrecoverably damaged and a dump is the best path to understanding the cause. `sysctl debug.kdb.panic=1` triggers an immediate panic and dump. Do not do this lightly.

Breaking into DDB on a production console is unsafe. The entire kernel stops while you are in DDB. User processes freeze. Network connections time out. Real-time work halts. Unless the alternative is worse (often the case during a catastrophic crash), stay out of DDB on production.

`DEBUG_MEMGUARD` on all allocation types is unsafe. Memory use balloons. Performance drops sharply. Memory-intensive workloads may fail outright. If you absolutely must use `memguard` in production, scope it to one `malloc` type at a time and monitor memory use.

Loadable kernel modules are a risk. Loading or unloading a module touches kernel state. A buggy module can crash the kernel at load time, unload time, or any time in between. On production, load only modules that have been tested in the development environment against the same kernel.

Overly aggressive DTrace scripts can destabilize the system. Aggregations that record stack traces produce memory pressure. Probes with side effects can interact with the workload in unexpected ways. Run DTrace scripts with explicit time limits and review the aggregations carefully before leaving them running.

### 34.10.4 Capturing Evidence on Production Systems

When something goes wrong in production and the bug is rare or difficult to reproduce in development, the challenge is to capture enough evidence to diagnose the problem, without destabilizing the running service. Several strategies help.

First, start with passive observation. `sysctl`, `vmstat -m`, `netstat -m`, `dmesg`, and the various `-s` system statistics commands can be run while the system is live and cost almost nothing. If the bug produces symptoms visible in these reports, capture the reports periodically.

Second, use DTrace with strict bounds. A script that runs for sixty seconds and exits produces a snapshot without leaving a standing risk. Aggregations are particularly well suited to this style: they collect statistics over a window, print them, and stop.

Third, if a crash dump is needed and the system has not yet crashed, the safest approach is to wait for the crash. Modern dump mechanisms are designed to capture the kernel state at the moment of panic; a dump triggered manually is useful only when you know the system is already irrecoverable.

Fourth, when a crash does happen, work on the dump, not the live system. A reboot into a fresh kernel restores service, while the dump remains available for offline analysis at leisure. The discipline of “reboot fast, analyze later” is often the right trade-off on production hardware.

### 34.10.5 Using log(9) Over printf for Diagnostics

Throughout the chapter we have used `printf` as a shorthand for kernel-side logging, which is how it is commonly presented in textbooks. On a production system you should prefer `log(9)`, which writes through the `syslogd(8)` facility rather than directly to the console. The reasons are practical: console output is unbuffered and slow, `log(9)` is rate-limited and buffered, and `log(9)` ends up in `/var/log/messages` where it is available to log-analysis tools.

The API is in `/usr/src/sys/sys/syslog.h` and `/usr/src/sys/kern/subr_prf.c`. Usage:

```
#include <sys/syslog.h>

log(LOG_WARNING, "bugdemo: unexpected state %d\n", sc->state);
```

The priority level (`LOG_DEBUG`, `LOG_INFO`, `LOG_NOTICE`, `LOG_WARNING`, `LOG_ERR`, `LOG_CRIT`, `LOG_ALERT`, `LOG_EMERG`) lets `syslogd` route messages differently.

A common extension is rate-limited logging, so a sick driver does not flood `/var/log/messages` with millions of entries per second. FreeBSD provides the `ratecheck(9)` primitive that you can wrap around your own `log` calls:

```
#include <sys/time.h>

static struct timeval lastlog;
static struct timeval interval = { 5, 0 }; /* 5 seconds */

if (ratecheck(&lastlog, &interval))
 log(LOG_WARNING, "bugdemo: error (rate-limited)\n");
```

`ratecheck(9)` returns nonzero once per interval, suppressing repeated logs in between. The technique is essential for any driver that might observe the same error repeatedly.

### 34.10.6 Do Not Mix Debug and Release Kernels in One Fleet

A subtle trap is to run a mix of debug and release kernels in a production fleet. The intuition is that debug kernels give you better diagnostics if a bug appears. The reality is that a debug kernel performs noticeably worse than a release kernel, has different memory usage, and can exhibit different timing. If a bug is sensitive to those factors (and many concurrency bugs are), running mixed kernels guarantees that your reproduction environment does not match your production environment.

The right approach is uniform: either the whole fleet runs release kernels (and you debug on development hardware), or the whole fleet runs debug kernels (and you accept the overhead). Mixed deployments are a third option only for very controlled experiments.

### 34.10.7 Working With a Recovery Plan

Before running any risky debugging action, know your recovery plan. If the system hangs, how will you recover? Is there an IPMI interface that can issue a hardware reset? Is there a second administrator who can cycle the power if needed? How much data loss is acceptable?

A good recovery plan is two steps. First, get the system back up quickly. Second, capture the evidence (dump, logs) for offline analysis. These two steps often involve different people or different timescales, and thinking through both in advance prevents panic in the moment.

### 34.10.8 Keep a Debug Journal

When debugging a hard bug, a written record is invaluable. Each entry should include:

- What hypothesis you were testing.
- What action you took.
- What result you observed.
- What the result rules in or out.

This sounds pedantic but is genuinely useful. A long debugging session involves dozens of micro-hypotheses, and losing track of which ones you have tested wastes enormous time. The written record also helps when you come back to the bug after a weekend, or when you hand it off to a colleague.

For a driver bug that goes across multiple systems, a shared record (bug tracker, wiki, or an internal ticket) is even more valuable. Each person who touches the bug can see what the others have already tried, and no one reruns the same experiment twice.

### 34.10.9 Practicing on Your Own Drivers

One habit that pays off over time is to keep a deliberately buggy version of your driver around for practice. Every time you find an interesting bug in real work, add a variant of it to your practice driver. Then, periodically, run through the practice driver with fresh eyes and make sure you can still find the bugs using the tools of this chapter. This builds muscle memory that is invaluable when a real bug appears under time pressure.

The bugdemo driver we have been using throughout this chapter is one starting point for such a practice driver. Fork it, add your own bugs, and use it to stay sharp.

### 34.10.10 Knowing When to Stop

A final piece of safe debugging wisdom is knowing when to stop. Not every bug needs to be hunted to the last instruction. If a bug is rare, has a workaround, and the cost of finding its root cause is measured in days, there is sometimes a case for documenting the workaround and moving on. This is a judgment call, not a rule, but the ability to make the call is part of professional maturity.

The opposite mistake (declaring victory too early, accepting a surface fix that does not address the underlying defect) is also common. The symptom is a bug that keeps coming back in new forms. When a “fix” does not produce a stable result, something deeper is wrong, and more investigation is called for.



Between these extremes is the healthy zone where you invest time proportional to the importance of the bug. Kernel development rewards patience, but it also rewards pragmatism. The tools of this chapter exist to make the investment efficient, not to make every bug a multi-day research project.

With safe practices established, we can turn to the last major topic of the chapter: what to do after a debugging session that found something serious, and how to make the driver more resilient against the next time something similar goes wrong.

## 34.11 8. Refactoring After a Debugging Session: Recovery and Resilience

A hard-won debugging victory is not the end of the work. Finding the bug is finding evidence. The real question is: what does the evidence tell us about the driver, and how should the driver change in response?

A common failure mode is to patch the immediate symptom and move on. The patch makes the test pass, the crash stop, the corruption vanish. But the underlying weakness that allowed the bug in the first place is still there, lurking. The next subtle change in surrounding code, or the next new environment, finds the same weakness and produces the next bug.

This section is about resisting that failure mode. We will walk through a small set of techniques for using a debugging result to strengthen the driver, not just to fix the particular bug.

### 34.11.1 Reading the Bug as a Message

Every bug carries a message about the design. A use-after-free says “the driver’s ownership model for this buffer is unclear.” A deadlock says “the driver’s lock order is not explicitly documented or enforced.” A memory leak says “the driver’s lifecycle for this object is incomplete.” A panic in `attach` says “the driver’s error recovery during initialization is weak.” A race condition says “the driver’s assumptions about thread context are not strict enough.”

When you find a bug, spend a few minutes asking what the bug is telling you about the design. The specific defect is usually a symptom of a broader pattern, and understanding the pattern makes future bugs easier to prevent.

### 34.11.2 Strengthening Invariants

One concrete response to a bug is to add assertions that would have caught it sooner. If a use-after-free was the bug, add a `KASSERT` somewhere in the path that confirms the buffer is still valid when used. If a lock order was violated, add a `mtx_assert(9)` at the point where the violation occurred. If a structure field was corrupted, add a `CTASSERT` on its alignment or a runtime check on its value.

The goal is not to duplicate every check with an assertion, but to convert each bug into one or two new invariants that make the same class of bug impossible in the future. Over time, the driver accumulates a set of defensive checks that reflect its actual behavior, documented in code rather than in your head.

### 34.11.3 Documenting the Ownership Model

Another common response is to clarify documentation. Many bugs arise because ownership of a resource (who allocated it, who is responsible for freeing it, when is it safe to access) is implicit. Writing a few lines of comment that explicitly state the ownership rules makes the rules visible to the next reader, and often forces you to confront cases where the rules were not actually consistent.

For example, a comment like:

```
/*
 * bugdemo_buffer is owned by the softc from attach until detach.
 * It may be accessed from any thread that holds sc->sc_lock.
 * It must not be accessed in interrupt context because the lock
 * is a regular mutex, not a spin mutex.
 */
struct bugdemo_buffer *sc_buffer;
```

This comment is not decorative. It is a statement of the invariants that the driver will enforce. If a future bug violates the invariants, the comment is a reference point for understanding what went wrong.

### 34.11.4 Narrowing the API Surface

A third response is to narrow the API. If the bug was caused because a function was called in a context where it should not have been, can the function be made private, so that it is only called from contexts where it is safe? If a state was reached through a path that should not have existed, can the state be made unreachable?

The principle is that every external entry point into a driver is a surface for bugs. Reducing the surface, by making functions internal, by hiding state behind accessors, by combining related operations into atomic transactions, makes the driver harder to misuse.

This is not about ideological minimalism. It is about recognizing that surface area is proportional to bug risk, and that many bugs can be prevented by the simple expedient of not exposing the thing that was misused.

### 34.11.5 Hardening the Unload Path

Unload is a frequently underhardened path in drivers. The attach path is usually well tested; the unload path is usually not. This is a major source of bugs: a driver that works perfectly in long-running use might crash on `kldunload`.

A good unload path satisfies several invariants. Every object allocated in attach is freed. Every thread spawned by the driver has exited. Every timer is cancelled. Every callout is drained. Every taskqueue has finished its pending work. Every device node is destroyed before the memory that supports it is freed.

After a bug in an unload path, audit the entire unload function against this checklist. Each item is an invariant that the driver should maintain, and violations are common.

### 34.11.6 The Shape of a Resilient Driver

Putting these habits together, what does a resilient driver look like? A few traits stand out.

Its locking is explicit. Every shared data structure is protected by a named lock, and every function that accesses the structure has either an assertion that the lock is held or a documented reason why the lock is not needed. Lock orders are documented in comments at each site of multi-lock acquisition. WITNESS produces no warnings during normal operation.

Its error handling is complete. Every allocation has a corresponding free. Every attach has a complete detach. Every path through the code cleans up after itself on failure. Partial states do not linger. The driver does not get stuck in half-initialized or half-torn-down states.

Its invariants are expressed in code. Preconditions are checked with KASSERT at function entry. Structural invariants are checked with CTASSERT at compile time. State transitions are verified with explicit checks.

Its observability is built in. Counters expose allocation and error rates. SDT probes fire at key events. Sysctls expose enough state that an operator can inspect the driver without a debugger. The driver tells you what it is doing.

Its error messages are useful. `log(9)` messages include the subsystem name, the specific error, and enough context to locate the problem. They are rate-limited. They do not log spurious warnings that train the operator to ignore them.

These traits are not free. They take time to implement and discipline to maintain. But once a driver has them, the cost of future bugs drops dramatically, because bugs are caught earlier, diagnosed more easily, and fixed more definitively.

### 34.11.7 Revisiting the bugdemo Driver

By the end of the chapter's labs, we will have applied many of these ideas to the bugdemo driver. What starts out as a handful of deliberately broken code paths grows, through iteration, into a driver with assertions at every key point, counters on every operation, SDT probes on every interesting event, and an unload path that passes scrutiny. The trajectory is deliberately designed to mirror the trajectory of a real driver as it matures.

### 34.11.8 Closing the Refactoring Loop

A last thought on refactoring. Every time you modify a driver in response to a bug, you are taking a small risk that the modification introduces a new bug. This risk is unavoidable but manageable. A few practices help.

First, isolate the change. Make the smallest modification that addresses the root cause, and commit it separately from cosmetic changes. If something regresses, the blame is easy to assign.

Second, add a test. If the bug was triggered by a specific sequence of ioctls, add a small test program that runs that sequence and verifies the correct result. Keep the test in your repository. A test suite that grows with each bug becomes an asset over years.

Third, run the existing tests. If the driver has any automated tests at all, run them after a fix. Surprisingly many regressions are caught this way, even with a small test suite.

Fourth, note the lesson. In your debug journal or commit message, write a brief note about what the bug revealed about the driver's design. The note is a gift to your future self, who will encounter similar patterns.

With these habits in place, debugging becomes a cycle of discovery rather than a string of firefights. Each bug teaches something, each lesson strengthens the driver, and each strengthened driver becomes easier to work with. The tools of this chapter are the means by which that cycle turns.

With the conceptual material behind us, we can move into the hands-on lab section, where we will apply every technique in the chapter to the bugdemo driver and see each one produce a concrete result.

## 34.12 Hands-on Labs

Each lab in this section is self-contained but builds on the previous ones. They use the bugdemo driver, whose companion source is under `examples/part-07/ch34-advanced-debugging/`.

Before starting, make sure you have a development FreeBSD 14.3 VM where you can crash the kernel safely, a copy of the FreeBSD source tree at `/usr/src/`, and the ability to attach a serial or virtual console that preserves output across a reboot.

### 34.12.1 Lab 1: Adding Assertions to bugdemo

In this lab we build the first version of bugdemo and add assertions that catch internal inconsistencies. The goal is to see KASSERT, MPASS, and CTASSERT working in practice.

#### Step 1: Build and load the baseline driver.

The baseline driver lives at `examples/part-07/ch34-advanced-debugging/lab01-kassert/`. It is a minimal pseudo-device with a single ioctl that triggers a bug when instructed to. From the lab directory:

```
$ make
$ sudo kldload ./bugdemo.ko
$ ls -l /dev/bugdemo
```

If the device node appears, the driver loaded correctly.

#### Step 2: Run the test tool to confirm the driver works.

The lab also contains a small user-space program, `bugdemo_test`, that opens the device and issues ioctls:

```
$./bugdemo_test hello
$./bugdemo_test noop
```

Both should return success. Without any bugs triggered, the driver behaves correctly.

#### Step 3: Inspect the assertions in the source.

Open `bugdemo.c` and find the function `bugdemo_process`. You will see something like this:

```
static void
bugdemo_process(struct bugdemo_softc *sc, struct bugdemo_command *cmd)
{
 KASSERT(sc != NULL, ("bugdemo: softc missing"));
 KASSERT(cmd != NULL, ("bugdemo: cmd missing"));
 KASSERT(cmd->op < BUGDEMO_OP_MAX,
 ("bugdemo: op %u out of range", cmd->op));
 MPASS(sc->state == BUGDEMO_STATE_READY);
 /* ... */
}
```

Each assertion documents an invariant. If any of them fires, the kernel panics with a message identifying the broken invariant.

#### Step 4: Trigger an assertion.

The driver has an ioctl called `BUGDEMO_FORCE_BAD_OP` that intentionally sets `cmd->op` to an out-of-range value before calling `bugdemo_process`:

```
$./bugdemo_test force-bad-op
```

With a debug kernel, this produces an immediate panic:

```
panic: bugdemo: op 255 out of range
```

and the system reboots. On a release kernel (no `INVARIANTS`), the `KASSERT` is compiled out and the driver continues on with an out-of-range value. The difference is exactly the value of having a debug kernel during development.

#### Step 5: Confirm the assertion fires on the right line.

After the reboot, if a dump was captured, open it with `kgdb`:

```
kgdb /boot/kernel/kernel /var/crash/vmcore.last
(kgdb) bt
```

The backtrace will show `bugdemo_process`, and frame `N` to that entry will show the assertion line. This is the end-to-end chain: assertion fires, kernel panics, dump captures state, `kgdb` identifies the code.

#### Step 6: Add your own assertion.

Modify the driver to add an assertion that a counter is nonzero in a specific code path. Rebuild, reload, and trigger a case that makes the counter zero. Observe that your assertion fires as expected.

**What this lab teaches.** The `KASSERT` macro is a live check, not a theoretical one. It fires, it panics, it identifies the code. The discipline of adding assertions is backed by a discipline of testing that they fire when they should.

### 34.12.2 Lab 2: Capturing and Analyzing a Panic With kgdb

In this lab we focus on the post-mortem workflow. Starting from a clean debug kernel, we trigger a panic, capture the dump, and walk through it with kgdb.

#### Step 1: Confirm the dump device is configured.

On the VM, run:

```
dumpon -l
```

If the output shows a device path (typically the swap partition), you are ready. If not, configure one:

```
dumpon /dev/ada0p3 # replace with your swap partition
echo 'dumpdev="/dev/ada0p3"' >> /etc/rc.conf
```

#### Step 2: Confirm the debug kernel is running.

```
uname -v
sysctl debug.debugger_on_panic
```

`debug.debugger_on_panic` should be 0 or 1 depending on whether you want to pause at the debugger before the dump. For automated lab work, 0 is easier; for interactive exploration, 1 is educational.

```
sysctl debug.debugger_on_panic=0
```

#### Step 3: Load bugdemo and trigger a panic.

```
kldload ./bugdemo.ko
./bugdemo_test null-softc
panic: bugdemo: softc missing
Dumping ...
Rebooting ...
```

The panic message, the dump notice, and the reboot all appear on the console. Dump writing takes a few seconds on a VM with virtual disk.

#### Step 4: After the reboot, inspect the saved dump.

```
ls /var/crash/
bounds info.0 info.last minfree vmcore.0 vmcore.last
cat /var/crash/info.0
```

The `info.0` file summarizes the panic: the kernel version, the message, and the initial backtrace captured before dumping.

#### Step 5: Open the dump in kgdb.

```
kgdb /boot/kernel/kernel /var/crash/vmcore.0
```

kgdb automatically runs a backtrace. Identify the frame that is inside `bugdemo_ioctl` or `bugdemo_process`. Switch to it:

```
(kgdb) frame 5
(kgdb) list
(kgdb) info locals
(kgdb) print sc
```

Observe that `sc` is `NULL`, confirming the panic message.

#### Step 6: Explore adjacent state.

From kgdb, examine the process that triggered the panic:

```
(kgdb) info threads
(kgdb) thread N # where N is the panicking thread
(kgdb) proc # driver-specific helper for process state
```

`proc` is a kernel-specific command that prints the current process. Between these commands and `bt`, you can build a full picture of the panic's context.

#### Step 7: Exit kgdb.

```
(kgdb) quit
```

The dump remains on disk; you can reopen it anytime.

**What this lab teaches.** The full cycle of panic, dump, and offline analysis is routine, not mysterious. A development VM should be able to complete this cycle in under a minute. The discipline is to practice it before the first real bug, so that when you need it you are not learning the tool in a hurry.

### 34.12.3 Lab 3: Building GENERIC-DEBUG and Confirming Options Active

This lab is about kernel configuration rather than code. The aim is to walk through the full process of building, installing, and validating a debug kernel.

#### Step 1: Start from a fresh `/usr/src/`.

If you have a source tree, update it. If not, install one with:

```
git clone --depth 1 -b releng/14.3 https://git.freebsd.org/src.git /usr/src
```

#### Step 2: Review the existing **GENERIC-DEBUG** config.

```
$ ls /usr/src/sys/amd64/conf/GENERIC*
$ cat /usr/src/sys/amd64/conf/GENERIC-DEBUG
```

Observe that it is only two lines: `include GENERIC` and `include "std.debug"`. Review `std.debug` next:

```
$ cat /usr/src/sys/conf/std.debug
```

Confirm the options we discussed: `INVARIANTS`, `INVARIANT_SUPPORT`, `WITNESS`, and friends.

### Step 3: Build the kernel.

```
cd /usr/src
make buildkernel KERNCONF=GENERIC-DEBUG
```

On a modest VM this takes twenty to forty minutes. The build produces detailed output; if it stops with an error, investigate and retry.

### Step 4: Install the kernel.

```
make installkernel KERNCONF=GENERIC-DEBUG
ls -l /boot/kernel/kernel /boot/kernel.old/kernel
```

The previous kernel is preserved in `/boot/kernel.old/` as a recovery option.

### Step 5: Reboot into the new kernel.

```
shutdown -r now
```

After reboot, confirm:

```
$ uname -v
$ sysctl debug.kdb.current_backend
$ sysctl debug.kdb.supported_backends
```

The backend should list both `ddb` and `gdb`.

### Step 6: Confirm `INVARIANTS` is active.

Build and load the `lab01 bugdemo.ko`, then trigger the out-of-range op as in Lab 1. On a debug kernel, the panic fires. On a release kernel, it does not. This round-trip confirms that `INVARIANTS` is genuinely compiled in.

### Step 7: Confirm `WITNESS` is active.

The `lab03` variant of `bugdemo` has a deliberate lock order inversion, triggered by a specific `ioctl`. Load it, run the triggering test, and watch for a `WITNESS` warning on the console:



```
lock_order_reversal:
```

```
...
```

No panic is produced, just a warning. This is the expected behavior: WITNESS detects potential deadlocks and reports them, without forcing a system failure.

#### Step 8: Recover if the new kernel does not boot.

If your new kernel fails to boot for any reason, the FreeBSD boot loader offers a recovery option. From the loader menu, select “Boot Kernel” and then “kernel.old”. Your previous kernel boots, and you can investigate the debug kernel’s failure at leisure.

**What this lab teaches.** Building a debug kernel is not a mysterious operation. It is a rebuild with different options and a reboot. The hazards are predictable: long build times, large binaries, and the need to keep the previous kernel available as a fallback.

### 34.12.4 Lab 4: Tracing bugdemo With DTrace and ktrace

This lab exercises the three tracing tools we studied: DTrace fbt probes, DTrace SDT probes, and ktrace(1).

#### Step 1: Load a bugdemo variant with SDT probes.

The lab04-tracing variant of bugdemo defines SDT probes at key points:

```
SDT_PROVIDER_DEFINE(bugdemo);
SDT_PROBE_DEFINE2(bugdemo, , , cmd__start, "struct bugdemo_softc **", "int");
SDT_PROBE_DEFINE3(bugdemo, , , cmd__done, "struct bugdemo_softc **", "int", "int");
```

Load it:

```
kldload ./bugdemo.ko
```

#### Step 2: List the probes.

```
dtrace -l -P sdt -n 'bugdemo:::*
```

You should see the cmd-start and cmd-done probes listed.

#### Step 3: Watch the probes fire.

In one terminal:

```
dtrace -n 'sdt:bugdemo::cmd-start { printf("op=%d\n", arg1); }'
```

In another terminal:

```
$./bugdemo_test noop
$./bugdemo_test hello
```

The first terminal shows each probe firing with its op value.

#### Step 4: Measure latency per op.

```
dtrace -n '
sdt:bugdemo::cmd-start
{
 self->start = timestamp;
}

sdt:bugdemo::cmd-done
/self->start != 0/
{
 @by_op[arg1] = quantize(timestamp - self->start);
 self->start = 0;
}
'
```

Run a workload of many ioctls, then Ctrl-C the DTrace. An aggregation prints, showing a latency histogram per op.

#### Step 5: Use fbt to trace entries.

```
dtrace -n 'fbt::bugdemo_*:entry { printf("%s\n", probefunc); }'
```

Trigger some ioctls from user space. The DTrace terminal shows each entry, giving you a live view of the driver's flow.

#### Step 6: Use ktrace to trace the user-space side.

```
$ ktrace -t ci ./bugdemo_test hello
$ kdump
```

Observe the ioctl calls visible in the kdump output.

#### Step 7: Combine ktrace and DTrace.

Run DTrace in one terminal, watching SDT probes, while running ktrace in another on the user-space test. The two outputs, read together, give a complete picture of the interaction from user space to kernel and back.

**What this lab teaches.** Tracing is not a single tool; it is a family. DTrace is the richest, ktrace(1) is the simplest way to see the user-kernel boundary, and combining them gives the most complete view.

### 34.12.5 Lab 5: Catching a Use-After-Free With memguard

This lab walks through a real memory debugging scenario. The `lab05-memguard` variant of `bugdemo` contains a deliberate use-after-free bug: under certain `ioctl` sequences, the driver frees a buffer and then reads it from a callout.

#### Step 1: Build a kernel with `DEBUG_MEMGUARD`.

Add options `DEBUG_MEMGUARD` to your `GENERIC-DEBUG` config, or create a new config:

```
include GENERIC
include "std.debug"
options DEBUG_MEMGUARD
```

Rebuild and install as in Lab 3.

#### Step 2: Load the `lab05 bugdemo` and enable `memguard`.

```
kldload ./bugdemo.ko
sysctl vm.memguard.desc=bugdemo
```

The second command tells `memguard(9)` to guard all allocations made with `malloc` type `bugdemo`. The exact type name comes from the driver's `MALLOC_DEFINE` call.

#### Step 3: Trigger the use-after-free.

```
$./bugdemo_test use-after-free
```

The user-space call returns quickly. A moment later (when the callout fires), the kernel panics with a page fault inside the callout routine:

```
Fatal trap 12: page fault while in kernel mode
fault virtual address = 0xfffff80002abcdef
...
KDB: stack backtrace:
db_trace_self_wrapper()
...
bugdemo_callout()
...
```

`memguard(9)` has converted a silent use-after-free into an immediate page fault. The backtrace points directly at `bugdemo_callout`.

#### Step 4: Analyze the dump with `kgdb`.

```
kgdb /boot/kernel/kernel /var/crash/vmcore.last
(kgdb) bt
(kgdb) frame N # into bugdemo_callout
(kgdb) list
(kgdb) print buffer
```

The source line shows the read from `buffer`, and `buffer` is a freed memguard-protected address. `kgdb` prints it as an address that is no longer mapped.

#### Step 5: Fix the bug and verify.

The fix is to cancel the callout before freeing the buffer. Modify the driver source accordingly, rebuild, reload, and run the same test. The panic no longer fires. Keep memguard enabled during the verification, then disable it and re-test:

```
sysctl vm.memguard.desc=
```

Both runs should succeed. If the release-mode run (without memguard) still fails, the bug is not fully fixed.

#### Step 6: Count inflight allocations.

The lab also shows an alternative technique: counting inflight allocations. Add a counter(9) to the driver, bump it on allocate, decrement on free. At unload, assert that the counter is zero:

```
KASSERT(counter_u64_fetch(bugdemo_inflight) == 0,
 ("bugdemo: leaked %ld buffers",
 (long)counter_u64_fetch(bugdemo_inflight)));
```

Unload without first freeing all buffers, observe the assertion fire.

**What this lab teaches.** `memguard(9)` is a specific tool for a specific class of bug. When it applies, it turns hard bugs into easy ones. Knowing when to reach for it is the practical skill.

### 34.12.6 Lab 6: Remote Debugging With the GDB Stub

This lab demonstrates remote debugging over a virtual serial port. It assumes you are using `bhyve` or `QEMU` with a serial console exposed to the host.

#### Step 1: Configure KDB and GDB in the kernel.

Both should already be present in `GENERIC-DEBUG`. Confirm with:

```
sysctl debug.kdb.supported_backends
```

#### Step 2: Configure the serial console in the VM.

In `bhyve`, add `-l com1,stdio` to the launch command, or equivalent. In `QEMU`, use `-serial stdio` or `-serial pty`. The goal is to have a virtual serial port accessible from the host.

#### Step 3: In the VM, switch to the GDB backend.

```
sysctl debug.kdb.current_backend=gdb
```

**Step 4: In the VM, drop into the debugger.**

Send the break-to-debugger sequence on the serial console, or trigger a panic:

```
sysctl debug.kdb.enter=1
```

The kernel halts. The serial console shows:

```
KDB: enter: sysctl debug.kdb.enter
[thread pid 500 tid 100012]
Stopped at kdb_enter+0x37: movq $0, kdb_why
gdb>
```

**Step 5: On the host, attach kgdb.**

```
$ kgdb /boot/kernel/kernel
(kgdb) target remote /dev/ttyXX # the host-side serial device
```

The host's kgdb connects to the kernel over the serial line. You can now run full kgdb commands on the live kernel: `bt`, `info threads`, `print`, `set variable`, and so on.

**Step 6: Set a breakpoint.**

```
(kgdb) break bugdemo_ioctl
(kgdb) continue
```

The VM resumes. In the VM, run `./bugdemo_test hello`. The breakpoint fires, and kgdb on the host shows the state.

**Step 7: Detach cleanly.**

```
(kgdb) detach
(kgdb) quit
```

In the VM, the kernel resumes running.

**What this lab teaches.** Remote debugging is a specialized but valuable tool. It is most useful when you need live inspection of a running kernel, particularly for intermittent bugs that are hard to capture as dumps.

## 34.13 Challenge Exercises

The following challenges build on the labs. They are open-ended by design: there are multiple valid approaches, and the point is to practice choosing the right tool for each bug.

### 34.13.1 Challenge 1: Find the Silent Bug

The lab-challenges/silent-bug variant of bugdemo contains a bug that produces no crash and no error. Instead, a counter sometimes reports the wrong value after a specific ioctl sequence. Your task:

1. Write a test program that reproduces the bug.
2. Use DTrace to narrow down which function produces the wrong counter value.
3. Fix the bug and verify that the DTrace signature disappears.

Hint: the bug is a missing memory barrier, not a missing lock. The symptom is cache coherence, not contention.

### 34.13.2 Challenge 2: Hunt the Leak

The lab-challenges/leaky-driver variant leaks an object each time a specific ioctl path is exercised. Your task:

1. Confirm the leak using `vmstat -m` before and after a workload.
2. Use DTrace to record every allocation and free of the leaked object type, aggregated by stack.
3. Identify the code path that allocates without freeing.
4. Add a counter (9)-based inflight check to the driver and verify that it fires when the buggy path is taken.

### 34.13.3 Challenge 3: Diagnose the Deadlock

The lab-challenges/deadlock variant sometimes hangs when two ioctls are run concurrently. Your task:

1. Reproduce the hang.
2. Attach to the hung kernel with `kgdb` (or drop into `DDB`).
3. Use `info threads` and `bt` on each stuck thread to identify the lock ordering.
4. Determine the fix (reorder locks, or eliminate one of them).

### 34.13.4 Challenge 4: Read a Real Panic

Load a kernel module you did not write (for example, one of the USB class drivers or a filesystem module). Deliberately trigger a bad interaction by sending malformed input from user space. When it panics (or fails to), write up:

1. The exact sequence that caused the symptom.
2. The backtrace or error observed.
3. Whether the module had assertions that would have caught the problem earlier.
4. One suggestion for strengthening the module's invariants.

### 34.13.5 Challenge 5: Build Your Own bugdemo Variant

Create a new variant of bugdemo that contains a bug you have encountered in real-world code. Write a test program that triggers the bug deterministically. Then, using any subset of the techniques in this chapter, diagnose

the bug from scratch. Write up what you learned. The point is to practice the conversion of “I recognize this pattern” into reproducible teaching material.

## 34.14 Troubleshooting Common Issues

Even the best tools run into problems in practice. This section collects the issues you are most likely to hit and how to resolve them.

### 34.14.1 The Dump Is Not Being Captured

After a panic, `/var/crash/` shows only bounds and `minfree`, no `vmcore.N`. Possible causes:

- **No dump device configured.** Run `dumpon -l` after a normal boot. If it reports “no dump device configured,” set one with `dumpon /dev/DEVICE` and persist it in `/etc/rc.conf` with `dumpdev=`.
- **Dump device too small.** A dump needs space equal to kernel memory plus some overhead. A 1GB swap partition will not hold a dump of an 8GB-memory machine. Enlarge the dump device or use a compressed dump (`dumpon -z`).
- **savecore disabled.** Check `/etc/rc.conf` for `savecore_enable="NO"`. Switch to YES and reboot.
- **Crash too severe.** If the panic itself prevents the dump machinery from running, you may see no output at all. In that case a serial console is essential to capture at least the panic message.

### 34.14.2 kgdb Says “No Symbols”

When opening a dump, `kgdb` prints “no debugging symbols found” or similar. Possible causes:

- **Kernel built without -g.** Debug kernels include `-g` automatically via `makeoptions DEBUG=-g`. Release kernels do not. Either build a debug kernel or install the debug symbols package if available.
- **Mismatch between kernel and dump.** If the dump came from a different kernel than the one `kgdb` is loading, symbols will not match. Use the exact kernel binary that was running at the time of the panic.
- **Module symbols missing.** If the panic is inside a module that was built without `-g`, `kgdb` shows addresses without source lines for that module. Rebuild the module with `DEBUG_FLAGS=-g`.

### 34.14.3 DDB Freezes the System

Entering DDB intentionally halts the kernel. This is by design, but on a production-like system it can look like a hang. If you are in DDB and want to resume:

- `continue` exits DDB and returns to the kernel.
- `reset` reboots immediately.
- `call doadump` forces a dump and then reboot.

If you entered DDB accidentally, `continue` is almost always the right action.

### 34.14.4 A Module Refuses to Unload

`kldunload bugdemo` returns `Device busy`. Causes:

- **Open file descriptors.** Something still has `/dev/bugdemo` open. Use `fstat | grep bugdemo` to find processes and close them.
- **Reference counts.** Another module references this one. Unload that module first.
- **Pending work.** A callout or taskqueue is still scheduled. Wait for it to drain, or make the driver explicitly cancel and drain in its unload path.
- **Stuck thread.** A kernel thread spawned by the driver has not exited. Terminate it from within the driver on unload.

### 34.14.5 memguard Does Nothing

`vm.memguard.desc=bugdemo` is set, but `memguard` does not seem to be catching any bugs. Causes:

- **Wrong type name.** `vm.memguard.desc` must match a type passed to `MALLOC_DEFINE(9)` exactly. If you set `vm.memguard.desc=BugDemo` but the driver uses `MALLOC_DEFINE(..., "bugdemo", ...)`, the names do not match.
- **Kernel not built with `DEBUG_MEMGUARD`.** The `sysctl` node exists only if the option is compiled in. Check `sysctl vm.memguard.waste` or similar; if it returns “unknown oid,” the feature is not compiled in.
- **Allocation path not traversed.** If the code path where the bug lives does not actually use the guarded type, `memguard` cannot catch it. Confirm the allocation type with `vmstat -m`.

### 34.14.6 DTrace Says “Probe Does Not Exist”

```
dtrace: invalid probe specifier sdt:bugdemo::cmd-start: probe does not exist
```

Causes:

- **Module not loaded.** SDT probes are defined by the module that provides them. If the module is not loaded, the probes do not exist.
- **Probe name mismatch.** The name in the source has double underscores (`cmd__start`), but DTrace uses a single dash (`cmd-start`). This is the conversion rule; the underscore form appears in C, the dash form in DTrace.
- **Provider not defined.** If `SDT_PROVIDER_DEFINE(bugdemo)` is missing or in a different file than `SDT_PROBE_DEFINE`, the probes will not exist.

### 34.14.7 Kernel Builds Fail With Symbol Conflicts

When building a kernel with unusual option combinations, you may see link errors like “multiple definition of X.” Causes:

- **Conflicting options.** Some options are mutually exclusive. Review the option documentation in `/usr/src/sys/conf/NOTES`.



- **Stale objects.** Old build artifacts can interfere with new builds. Try `make cleandir` && `make cleandir` in the kernel build directory.
- **Tree inconsistency.** A partial update of `/usr/src/` can leave headers and sources out of sync. Run a full `svnlite update` or `git pull` and retry.

### 34.14.8 The System Boots Into the Old Kernel

After `installkernel`, you reboot and `uname -v` shows the old kernel. Causes:

- **Boot entry not updated.** The default is `kernel`, which points to the current kernel. If you installed with `KERNCONF=GENERIC-DEBUG` but did not run `make installkernel` cleanly, the old binary may still be in place. Check `/boot/kernel/kernel` timestamp.
- **Wrong kernel selected in loader.** The FreeBSD loader menu has a “Boot Kernel” option that can select between available kernels. Pick the right one, or set `kernel="kernel"` in `/boot/loader.conf`.
- **Boot partition unchanged.** On some systems the boot partition is separate and needs a manual copy. Check that you installed to the correct partition.

### 34.14.9 WITNESS Reports False Positives

Sometimes WITNESS warns about a lock order that you know is safe. Possible reasons and responses:

- **The order really is unsafe but harmless in practice.** WITNESS reports potential deadlocks, not actual ones. A cycle in the lock graph that is never exercised concurrently is still a bug waiting to happen. Refactor the locking.
- **Locks are acquired by address.** Generic code that locks by pointer can produce orders that depend on runtime data, not static structure. See `witness(4)` for how to suppress specific orders with `witness_skipspin` or manual overrides.
- **Multiple locks of the same type.** Acquiring two instances of the same lock class is always a potential issue. Use `mtx_init(9)` with a distinct type name per instance if you need them treated as separate classes.

## 34.15 Putting It All Together: A Debugging Session Walkthrough

Before we wrap up, let us walk through a complete debugging session that uses several of the techniques we have studied. The scenario is fictional but realistic: a driver that sometimes fails with a misleading error, and we track it down from first symptom to root cause.

### 34.15.1 The Symptom

A user reports that their program sometimes gets `EBUSY` from an `ioctl` on `/dev/bugdemo`. The program always calls the `ioctl` in the same way, and most of the time it works. Only under heavy load does `EBUSY` appear, and then inconsistently.

### 34.15.2 Step 1: Collect Evidence

The first step is to observe the phenomenon without perturbing it. We run the user's program under `kt race(1)` to confirm the symptom:

```
$ ktrace -t ci ./user_program
$ kdump | grep ioctl
```

The output confirms that a specific `ioctl` returns `EBUSY` now and then. No other user-space call misbehaves. This tells us the bug is in the kernel's handling of that `ioctl`, not in the user program's logic.

### 34.15.3 Step 2: Form a Hypothesis

The error code `EBUSY` usually indicates a resource conflict. Reading the driver source, we find that `EBUSY` is returned when an internal flag indicates a previous operation is still in progress. The flag is cleared by a callout that completes the operation.

Our hypothesis: under heavy load, the callout is delayed long enough that a new `ioctl` arrives before the previous one has completed. The driver was not designed to serialize such requests, so it rejects the newcomer.

### 34.15.4 Step 3: Test the Hypothesis With DTrace

We write a DTrace script that records the delay between consecutive `ioctls` and the state of the busy flag at each entry:

```
dtrace -n '
fbt::bugdemo_ioctl:entry
{
 self->ts = timestamp;
}

fbt::bugdemo_ioctl:return
/self->ts != 0/
{
 @[pid, self->result] = lquantize(timestamp - self->ts, 0, 1000000, 10000);
 self->ts = 0;
}
'
```

Running the user program under load, we observe that `EBUSY` returns happen almost exclusively when the previous `ioctl` completed more than 50 microseconds ago and the callout has not fired yet. This corroborates the hypothesis.

### 34.15.5 Step 4: Confirm With SDT Probes

We add SDT probes around the busy flag manipulation and watch them:

```
dtrace -n '
sdt:bugdemo::set-busy
{
 printf("%lld set busy\n", timestamp);
}

sdt:bugdemo::clear-busy
{
 printf("%lld clear busy\n", timestamp);
}

sdt:bugdemo::reject-busy
{
 printf("%lld reject busy\n", timestamp);
}
'
```

The trace shows a clear pattern: set, reject, reject, reject, clear, set, clear. The clear is coming late because the callout is competing for a shared taskqueue with other work.

### 34.15.6 Step 5: Identify the Fix

With the evidence gathered, the fix is clear. Either the driver needs to serialize incoming ioctls instead of rejecting them (a queue or a wait), or it needs to complete the previous operation synchronously instead of via callout.

We pick the queuing approach because it preserves the callout's benefits. The driver accumulates pending requests and dispatches them as the callout fires. Under light load, nothing changes. Under heavy load, requests wait rather than failing.

### 34.15.7 Step 6: Implement and Verify

We modify the driver. We run the user program under the original workload. EBUSY no longer appears. The DTrace latency distribution now shows a tail that reflects the queueing delay, which is acceptable for this driver's use case.

We also enable `DEBUG_MEMGUARD` on the driver's malloc type and run the workload for a while, to make sure the queuing code does not introduce memory bugs. No faults fire.

Finally, we run the full test suite. Everything passes. The fix is committed with a descriptive message that explains the root cause, not just the symptom.

### 34.15.8 Lessons From This Session

Two things are worth noting.

First, the tools we used were relatively lightweight. No crash dump was needed. No DDB was entered. The bug was diagnosed through passive observation, DTrace, and careful reading. For many driver bugs, this is the shape of a session: not a dramatic panic but a systematic narrowing of hypotheses.

Second, the fix addressed the root cause, not the symptom. A surface fix might have been to increase the priority of the callout's taskqueue. That would have reduced the frequency of the bug without eliminating it. A more principled fix changes the driver's contract from "reject if busy" to "queue and serve." This is the refactoring mindset we discussed in Section 8: every bug is a message about the design.

## 34.16 Additional Techniques Worth Knowing

The chapter has covered the core of FreeBSD's debugging toolkit. A handful of additional techniques did not fit into the main narrative but deserve mention, because you will eventually encounter them.

### 34.16.1 witness\_checkorder With Manual Lists

WITNESS can be tuned. In `/usr/src/sys/kern/subr_witness.c` there is a table of known-good lock orders that the kernel recognizes. When building a driver that uses a subsystem locked by an existing lock, adding the driver's own lock to this table lets WITNESS verify the combined order across the driver and the subsystem.

This is rarely needed in small drivers but becomes useful for drivers that interact deeply with multiple subsystems.

### 34.16.2 sysctl debug.ktr

Beyond just enabling and disabling `ktr(9)` classes, there are additional controls:

- `debug.ktr.clear=1` clears the buffer.
- `debug.ktr.verbose=1` sends trace entries to the console in real time, in addition to the ring.
- `debug.ktr.stamp=1` adds timestamps to each entry.

The combination of these is especially useful when you want to watch a live trace without running `ktrdump(8)` repeatedly.

### 34.16.3 DDB Commands Beyond bt

DDB has a rich set of commands that are sparsely documented. A few are particularly useful for driver authors:

- `show all procs` lists every process.
- `show lockedvnods` shows currently locked vnods (useful for storage driver bugs).
- `show mount` shows mounted filesystems.
- `show registers` dumps CPU registers.
- `break FUNC` sets a breakpoint.

- `step` and `next` advance one instruction or one line.
- `watch` sets a watchpoint on an address.

The `help` command in DDB lists all available commands. Reading the list once is a useful way to discover features you did not know about.

#### 34.16.4 Kernel Option `KDB_TRACE`

`KDB_TRACE` causes the kernel to print a stack trace on every panic, even if the operator does not interact with the debugger. This is useful in automated testing where nobody is watching the console. It is already in `GENERIC`.

#### 34.16.5 EKCD: Encrypted Kernel Crash Dumps

If kernel dumps contain sensitive data (process memory, credentials, keys), the kernel can encrypt them at dump time. The `EKCD` option enables this feature. A public key is loaded at runtime with `dumpon -k`; the matching private key is used at `savecore` time to decrypt.

This matters on production systems where dumps might be transported over untrusted channels. It does not matter on a development VM.

#### 34.16.6 Lightweight Debug Output: `bootverbose`

Another low-overhead option is `bootverbose`. Setting `boot_verbose` in the loader or `bootverbose=1` in `sysctl` causes many kernel subsystems to print additional diagnostic information at boot. If your driver has not yet reached the point where `DTrace` applies, `bootverbose` can help you see what the driver is doing during attach.

The way to make your own driver honor `bootverbose` is to check `bootverbose` in your probe or attach code:

```
if (bootverbose)
 device_printf(dev, "detailed attach info: ...\n");
```

This is a well-established pattern in `/usr/src/sys/dev/` drivers.

### 34.17 A Closer Look at DDB

The in-kernel debugger, DDB, deserves more attention than we have given it so far. Many driver authors use DDB only reactively, when a panic lands them in it unexpectedly. With a little practice, DDB is also a useful tool to enter deliberately, for interactive inspection of a live kernel.

#### 34.17.1 Entering DDB

There are several ways to enter DDB. We have seen some of them already:

- By panic, if `debug.debugger_on_panic` is nonzero.

- By serial BREAK (or Ctrl-Alt-Esc on a keyboard console), when BREAK\_TO\_DEBUGGER is compiled in.
- By the alternative sequence CR ~ Ctrl-B, when ALT\_BREAK\_TO\_DEBUGGER is compiled in.
- Programmatically, with `sysctl debug.kdb.enter=1`.
- From code, by calling `kdb_enter(9)`.

In development, the programmatic entry is the most convenient. You can drop into DDB at a specific point in a script without waiting for a panic.

### 34.17.2 DDB Prompts and Commands

Once inside, DDB presents a prompt. The standard prompt is simply `db>`. Commands are typed, followed by Enter. DDB has a command history (press up-arrow on the serial console) and tab-completion for many command names.

A useful first command is `help`, which lists categories of commands. `help show` lists the many show subcommands. Most exploration happens through `show`.

### 34.17.3 Walking a Thread

The most common diagnostic task in DDB is to walk a specific thread. Start with `ps`, which lists all processes:

```
db> ps
 pid ppid pgrp uid state wmesg wchan cmd
 0 0 0 0 RL (swapper) [...] swapper
 1 0 1 0 SLs wait [...] init
 ...
 500 499 500 0 SL nanslp [...] user_program
```

Pick the thread of interest. In DDB, switching to a thread is done by the `show thread` command:

```
db> show thread 100012
 Thread 100012 at 0xfffffe00...
 ...
db> bt
```

This walks the stack of that specific thread. A kernel deadlock investigation typically involves walking each stuck thread to see where it is waiting.

### 34.17.4 Inspecting Structures

DDB can dereference pointers and print structure fields if the kernel was built with `DDB_CTF`. Example:

```
db> show proc 500
db> show malloc
db> show uma
```

Each of these prints a formatted view of the relevant kernel state. `show malloc` gives a table of malloc types and their current allocations. `show uma` does the same for UMA zones. `show proc` shows a specific process in detail.

### 34.17.5 Setting Breakpoints

DDB supports breakpoints. `break FUNC` sets a breakpoint at the entry to a function. `continue` resumes execution. When the breakpoint fires, the kernel returns to DDB and you can inspect state at that point.

This is the machinery that makes DDB a real debugger, not just a crash inspector. With breakpoints you can pause the kernel at a specific code location, examine arguments, and decide whether to continue.

The catch is that a kernel paused in DDB really is paused. While you are in DDB, no other thread runs. On a network server, every client times out. On a desktop, the GUI freezes. For local, development-VM debugging, this is fine. For any remote or shared use, it is not.

### 34.17.6 Scripting DDB

DDB supports a simple scripting facility. You can define named scripts that execute a sequence of DDB commands. `script kdb.enter.panic=bt; show registers; show proc` makes those three commands run automatically every time the debugger enters due to a panic. This is useful for unattended dumps: the scripted output appears on the console and in the dump, giving you information without needing an interactive session.

The scripts are stored in kernel memory and can be configured at boot via `/boot/loader.conf` or at runtime with `sysctl` calls. See `ddb(4)` for the exact syntax.

### 34.17.7 Exiting DDB

When you are done, `continue` exits DDB and the kernel resumes. `reset` reboots. `call doadump` forces a dump and reboot. `call panic` triggers a panic intentionally (useful when you want a dump from the current state but did not reach DDB via a panic).

For the developer practicing on a VM, `continue` is the one command to remember. It brings the kernel back to life and lets you keep working.

### 34.17.8 DDB vs kgdb: When to Use Each

DDB and kgdb overlap but are not interchangeable.

Use DDB when the kernel is running (or paused on a specific event) and you want to poke around. DDB runs inside the kernel and has direct access to kernel memory and threads. It is the right tool for quick state checks, for setting breakpoints, and for stopping on specific events.

Use kgdb on a crash dump, after the machine has rebooted. kgdb has no access to a running system's threads, but it has full gdb features for offline analysis: command history, source browsing, scripting with Python, and so on.

For a live kernel that you cannot reboot, the GDB stub backend of KDB bridges the gap: the kernel pauses, and kgdb on another machine attaches over a serial line for full gdb features on the live state. This is the most capable combination but requires two machines (or VMs).

## 34.18 Practical Worked Example: Following a Null Pointer

To pull the tools together, let us walk through one more worked example. The symptom: bugdemo occasionally panics with page fault: supervisor read instruction and a backtrace through bugdemo\_read. The panic address is low, suggesting a null pointer dereference.

### 34.18.1 Step 1: Capture the Dump

After the panic, we confirm a dump was saved:

```
ls -l /var/crash/
```

and open it:

```
kgdb /boot/kernel/kernel /var/crash/vmcore.last
```

### 34.18.2 Step 2: Read the Backtrace

```
(kgdb) bt
#0 __curthread ()
#1 doadump (textdump=0) at /usr/src/sys/kern/kern_shutdown.c
#2 db_fncall_generic at /usr/src/sys/ddb/db_command.c
...
#8 bugdemo_read (dev=..., uio=..., ioflag=0)
 at /usr/src/sys/modules/bugdemo/bugdemo.c:185
```

The interesting frame is 8, bugdemo\_read. The code at line 185 is:

```
sc = dev->si_drv1;
amt = MIN(uio->uio_resid, sc->buflen);
```

### 34.18.3 Step 3: Inspect the Variables



```
(kgdb) frame 8
(kgdb) print sc
$1 = (struct bugdemo_softc *) 0x0
(kgdb) print dev->si_drv1
$2 = (void *) 0x0
```

`si_drv1` is NULL on the dev. This is the private pointer that `make_dev(9)` sets; it should have been set during attach.

#### 34.18.4 Step 4: Walk Back

```
(kgdb) print *dev
```

We see the device structure. The name field says “bugdemo”, the flags look reasonable, but `si_drv1` is NULL. Something cleared it.

#### 34.18.5 Step 5: Form a Hypothesis

In the source, `si_drv1` is set once, in `attach`, and it is read in every `read`, `write`, and `ioctl` handler. It is never explicitly cleared. However, in the unload path, the device is destroyed with `destroy_dev(9)`, which returns before pending handlers finish. If a read is in progress when unload starts, the dev might be partially destroyed.

#### 34.18.6 Step 6: Add an Assertion

A `KASSERT` at the top of `bugdemo_read` catches the case:

```
KASSERT(sc != NULL, ("bugdemo_read: no softc"));
```

With this assertion in place, the next panic gives us the same information without requiring a dump walk. We also know immediately that the condition is real, not a random corruption.

#### 34.18.7 Step 7: Fix the Bug

The real fix is to make the unload path wait for pending handlers before destroying the device. FreeBSD provides `destroy_dev_drain(9)` for exactly this purpose. Using it:

```
destroy_dev_drain(sc->dev);
```

ensures that no read or write is in flight when the `softc` is freed.

### 34.18.8 Step 8: Verify

Load the fixed driver. Run concurrent reads and unloads. The panic does not reproduce. The KASSERT stays in the code as a safety net for future refactoring.

### 34.18.9 Takeaway

This workflow (capture, read, inspect, hypothesize, verify) is the shape of most productive debugging sessions. Each tool plays a small, specific role. The discipline is to gather evidence before acting, and to leave assertions behind as witnesses for the future.

## 34.19 Making Drivers Observable From Day One

A theme running through this chapter is that the best time to add debugging infrastructure is before you need it. A driver that was designed with observability in mind is easier to debug than a driver that was designed for speed alone.

A few concrete habits support this.

### 34.19.1 Name Every Allocator Type

MALLOC\_DEFINE(9) requires a short and a long name. The short name is what appears in `vmstat -m` output and in `memguard(9)` targeting. Picking a descriptive name, unique to the driver, makes later diagnosis easier. Never share a malloc type between unrelated subsystems; the tooling cannot distinguish them.

### 34.19.2 Count Important Events

Every major event in a driver (open, close, read, write, interrupt, error, state transition) is a candidate for a counter(9). Counters are cheap, they accumulate over time, and they are exposed through `sysctl`. A driver with good counters answers most “what is this thing doing” questions without any additional tooling.

### 34.19.3 Declare SDT Probes

Every state transition is a candidate for an SDT probe. Unlike assertions or counters, probes cost nothing when disabled. Leaving them in the source for the lifetime of the driver is a net win: when a bug appears, DTrace can see the event flow without requiring a rebuild.

### 34.19.4 Use Consistent Log Messages

`log(9)` messages should follow a consistent format. A prefix that identifies the driver, a specific error code or state, and enough context to locate the problem are the essentials. Avoid cleverness in log messages; a reader under time pressure wants to know what happened, not to admire your prose.

### 34.19.5 Provide Useful Sysctls

Every internal flag, every counter, every configuration value should be exposed through sysctl unless there is a specific reason not to. The reader who needs to debug your driver will thank you; the reader who never needs to debug your driver pays nothing for the exposure.

### 34.19.6 Write Assertions As You Go

The best time to add a KASSERT is when the invariant is fresh in your mind, which is while you are writing the code. Going back later to sprinkle assertions is less effective because you have forgotten some invariants and have rationalized others as “obvious.”

### 34.19.7 Expose the State of the State Machine

Every nontrivial driver has a state machine. Exposing the current state through a sysctl, an SDT probe at each transition, and a counter per state makes the state machine visible to both humans and tools. This is particularly important for asynchronous drivers, which is the subject of the next chapter.

### 34.19.8 Test the Unload Path

An underhardened unload path is a classic source of crashes. In development, write a test that loads the driver, exercises it briefly, and unloads it, repeatedly and under various conditions. If the driver cannot sustain a hundred load/unload cycles, it has bugs.

These habits cost a little time in development and pay for themselves many times over in debugging. A disciplined driver author applies them all, even to drivers that look too simple to need them.

## 34.20 Real-World Reading List

Every tool in this chapter is documented more completely in its own man page or source file. That is good news: you do not have to carry the whole toolbox in your head. When a bug points you at a specific subsystem, opening the right man page or source file will almost always take you further than any chapter can. The following list gathers the references that matter most for this material, in the order you are likely to need them.

The `witness(4)` man page is the first thing to read when `GENERIC-DEBUG` starts printing lock order reversals and you want to understand exactly what the output means, which sysctl controls change behavior, and which counters you can inspect. It documents the `debug.witness.*` sysctls, the `show all_locks` DDB command, and the general approach `WITNESS` takes to bookkeeping. For the actual implementation, `/usr/src/sys/kern/subr_witness.c` is the authoritative source. Reading the structures it maintains and the shape of its output functions (the ones that produce the “1st ... 2nd ...” lines you saw earlier in this chapter) removes most of the mystery from a `WITNESS` report. The file is long, but the top comment and the output-producing functions together cover most of what a driver author needs to know.

For lock profiling, `lockstat(1)` is documented in `/usr/src/cddl/contrib/opensolaris/cmd/lockstat/lockstat.1`. The man page ends with several worked examples whose output format matches what you

will see on your own systems, which makes it a useful reference to keep open the first time you try `-H -s 8` on a real workload. Because `lockstat(1)` is DTrace-backed, the `dt race(1)` man page is its natural companion; you can express the same queries in raw D if you need flexibility the `lockstat` command-line flags do not offer.

For kernel debugger work, `ddb(4)` documents the in-kernel debugger in full, including every builtin command, every script hook, and every way to enter the debugger. When in doubt, read this page before using a DDB command you have not tried before. For offline post-mortem analysis, `kgdb(1)` on your installed FreeBSD system documents the kernel-specific extensions over stock `gdb`. The underlying access layer is in `libkvm`, described in `/usr/src/lib/libkvm/kvm_open.3`, which explains both the dump and the live-kernel modes you met in Section 3.

Two smaller pointers are worth keeping in your reading queue. The first is `/usr/src/share/examples/witness/lockgraphs.sh`, a tiny shell script shipped in the base system that demonstrates how to turn WITNESS's accumulated lock-order graph into a visual diagram. On a real driver, running it once gives you a picture of where your locks sit relative to the rest of the kernel's lock hierarchy, and it can be surprising. The second is the FreeBSD kernel source tree itself: reading `/usr/src/sys/kern/kern_shutdown.c` (the panic and dump path) and `/usr/src/sys/kern/kern_mutex.c` (the mutex implementation that WITNESS instruments) grounds the whole debugging workflow in the code that actually implements it.

Beyond the source tree, the FreeBSD Developers' Handbook and the FreeBSD Architecture Handbook both contain longer-form articles on kernel debugging. Both are shipped with the documentation set on any FreeBSD system and kept up to date alongside the source. They are worth browsing once, even if you do not read them end to end, because they give names to patterns you will later recognize in your own debugging sessions.

A final note on choosing references. Man pages age more slowly than blog posts, and source comments age more slowly than man pages. When two references disagree, trust the source, then the man page, then the handbook, then everything else. That hierarchy has served FreeBSD developers well for decades, and the habit will serve you through the rest of this book and the rest of your career.

## 34.21 Wrapping Up

Advanced debugging is a patient craft. Each tool in this chapter exists because someone, somewhere, faced a bug that could not be found any other way. `KASSERT` exists because invariants that live only in the programmer's head are not invariants. `kgdb` and crash dumps exist because some bugs destroy the machine that produced them. DDB exists because a frozen kernel cannot explain itself through any other channel. WITNESS exists because deadlocks are catastrophic in production and impossible to debug after the fact. `memguard(9)` exists because silent memory corruption was the hardest class of bug until someone built a tool that made it loud.

None of these tools replace understanding. A debugger cannot tell you what your driver should do. A crash dump cannot tell you the right locking discipline. DTrace cannot infer your design. The tools are instruments; you are the player. The music is the shape of the driver you are building.

The habits that make this craft successful are small and undramatic. Develop on a debug kernel. Add assertions for every invariant you can articulate. Capture dumps routinely so you can open them without ceremony. Keep a journal when you are chasing something hard. Read the FreeBSD source when a mechanism mystifies you. Reach for the lightest tool that will answer your question, and graduate to heavier tools only when the lighter ones come up short.

Debugging is also a social craft. A bug that takes you a day to find, written up clearly, can save another author a week. Good commit messages, detailed test cases, and honest accounts of what worked and what did not are contributions to the common practice. The FreeBSD project's historical patience with bug reports, its habit of capturing root causes in commit logs, and its consistent use of `KASSERT` and `WITNESS` across decades of drivers all stem from this collective habit of treating bug hunting as a shared responsibility.

You now have the toolkit to participate. Load a debug kernel, pick a driver in `/usr/src/sys/dev/` that interests you, and read it with a debugger's eye. Where are the invariants? Where are the assertions? Where is the locking discipline? Where might a bug hide, and what tool would catch it? The exercise sharpens the instincts the rest of this book has been building.

In the next chapter, we will leave correctness behind and look at how drivers handle asynchronous I/O and event-driven work: the patterns by which a driver serves many users at once without blocking, and the kernel facilities that make such designs possible. The debugging skills you have gained here will serve you well in that territory, because asynchronous code is where subtle concurrency bugs tend to live. A driver with solid assertions, a clean lock ordering verified by `WITNESS`, and a set of SDT probes to trace its event flow is also a driver that is much easier to reason about when its work is spread across callbacks, timers, and kernel threads.

## 34.22 Bridge to Chapter 35: Asynchronous I/O and Event Handling

Chapter 35 picks up where this one leaves off. Synchronous code is straightforward to reason about: a call arrives, the driver does its work, the call returns. Asynchronous code is not: callbacks fire at unpredictable times, events arrive out of order, and the driver must manage state that persists across many thread contexts.

The complexity of asynchronous drivers is exactly the kind of complexity that benefits from the tools of this chapter. A synchronous driver with a bug might crash at a predictable place. An asynchronous driver with a bug might crash hours later, in a callback that has no obvious connection to the original misbehavior. `KASSERT` on the state at each callback entry catches such bugs early. `DTrace` probes on each event transition make the sequence visible. `WITNESS` detects the deadlocks that arise naturally when multiple asynchronous paths need to coordinate.

In the next chapter we will meet the building blocks of asynchronous work in FreeBSD: `callout(9)` for deferred timers, `taskqueue(9)` for background work, `kqueue(9)` for event notification, and the patterns for using them correctly. We will build a driver that serves many concurrent users without blocking, and we will exercise the debugging techniques of this chapter to keep that complexity under control.

By the time you finish Chapter 35, you will have the full synchronous and asynchronous toolkit: drivers that handle traffic efficiently, scale to many users, maintain correctness under concurrency, and can be debugged when something nevertheless goes wrong. That combination is what it takes to write drivers that survive in production.

See you in Chapter 35.

## 35 Asynchronous I/O and Event Handling

### 35.1 Introduction

Up to this point, almost every driver we have written has operated on a simple schedule. A user process calls `read(2)` and waits. Our driver produces data, the kernel copies it out, and the call returns. A user process calls `write(2)` and waits. Our driver consumes the data, stores it, and the call returns. The user thread sleeps while the driver works, and it wakes up when the work is done. This is the synchronous model, and it is the right starting point for teaching drivers because it matches the shape of an ordinary function call: you ask for something, you wait, you get an answer.

Synchronous I/O works well for many devices, but it fails for others. A keyboard does not decide to produce a keypress because a program called `read()`. A serial port does not time its incoming bytes to match the reader's schedule. A sensor might produce data at irregular intervals, or only when something interesting happens in the physical world. If we insist that every user of such a device must block in `read()` until the next event arrives, we force the userland program into a terrible choice. It can either dedicate a thread to blocking on each device, which makes the program hard to write and slow to respond to anything else, or it can loop in userland calling `read()` with a short timeout over and over, which wastes CPU cycles and still misses events that happen between polls.

FreeBSD solves this problem by giving drivers a set of asynchronous notification mechanisms, each built on the same underlying idea: a process does not need to block in `read()` to learn that data is ready. It can instead register interest in a device, go do other useful work, and let the kernel wake it up when the device has something to say. The mechanisms differ in their details, their performance profiles, and their intended use cases, but they share a common shape. A waiter declares what it is waiting for, the driver records that interest, the driver later discovers that the condition is satisfied, and the driver delivers a notification that causes the waiter to be woken, scheduled, or signalled.

Four of these mechanisms matter to driver authors. The classic `poll(2)` and `select(2)` system calls let a userland program ask the kernel which of a set of file descriptors are ready. The newer `kqueue(2)` framework offers a more efficient, more expressive event interface and is the preferred choice for modern high-performance applications. The `SIGIO` signal mechanism, invoked through `FIOASYNC` and `fsetown()`, delivers signals to a registered process whenever the device state changes. And drivers that need to track their own internal events typically build a small event queue inside the softc so that readers see a consistent sequence of readable records rather than raw hardware state.

In this chapter we will learn how each of these mechanisms works, how to implement it correctly in a character driver, how to combine them so that a single driver can serve callers from `poll(2)`, `kqueue(2)`, and `SIGIO` simultaneously, and how to audit the resulting code for the subtle races and missed wakeups that are the bane of asynchronous programming. We will ground every piece in real FreeBSD 14.3 source, looking at how `if_tuntap.c`, `sys_pipe.c`, and `evdev/cdev.c` solve the same problems in production.

By the end of the chapter you will be able to take a blocking driver and give it full asynchronous support without breaking its synchronous semantics. You will know how to implement `d_poll()`, `d_kqfilter()`, and `FIOASYNC` handlers correctly. You will understand why `selrecord()` and `selwakeup()` must be called in a specific order with specific locking. You will know what a `knlist` is, how `knote` attaches to it, and why `KNOTE_LOCKED()` is the call you want in almost every driver. You will see how `fsetown()` and `pgsigio()` combine to deliver signals to exactly the right process. And you will know how to build an internal event queue that ties the whole mechanism together so that each asynchronous notification leads the reader to a single, consistent, well-defined record in the driver.

Throughout the chapter we will develop a companion driver called `evdemo`. It is a pseudo-device that simulates an event source: timestamps, state transitions, and occasional “interesting” events that a userland program wants to observe in real time. Each section of the chapter builds another layer onto `evdemo`, so by the end you will have a small but complete asynchronous driver that you can load, inspect, and extend. Like `bugdemo` in the previous chapter, `evdemo` touches no real hardware, so every experiment is safe to run on a development FreeBSD virtual machine.

## 35.2 Reader Guidance: How to Use This Chapter

This chapter sits in Part 7 of the book, in the Mastery Topics part, directly after Advanced Debugging. It assumes you have already written at least a simple character driver, know how to load and unload a module safely, and have worked with the synchronous `read()`, `write()`, and `ioctl()` handlers. If any of those feel uncertain, a quick revisit of Chapters 8 through 12 will pay for itself several times in this one.

You do not need to have finished every earlier mastery chapter to follow this one. A reader who has mastered the basic character driver pattern and has seen `callout(9)` or `taskqueue(9)` in passing will be able to keep up. Where a previous chapter’s material is essential, we will give you a short reminder in the relevant section.

The material is cumulative within the chapter. Each section adds a new asynchronous mechanism to the `evdemo` driver, and the final refactor ties them together. You can skim forward to learn about one specific mechanism, but the labs read most naturally in sequence, because later labs assume the code from the earlier ones.

You do not need any special hardware. A modest FreeBSD 14.3 virtual machine is enough for every lab in the chapter. A serial console is useful but not required. You will want a second terminal open so you can watch `dmesg`, run the user-space test programs, and monitor wait channels in `top(1)` while the driver is loaded.

A reasonable reading schedule looks like this. Read the first three sections in one sitting to build the mental model for `poll` and `select`. Take a break. Read Sections 4 and 5 on a separate day, because `kqueue` and `signals` each introduce a new set of ideas. Work through the labs at your own pace. The chapter is long on purpose: asynchronous I/O is where a lot of driver complexity lives, and rushing through the material is the surest way to write a driver that works most of the time but misses wakeups in rare cases.

Some of the code in this chapter deliberately does the wrong thing so that we can see the symptoms of common mistakes. Those examples are clearly labelled. The finished labs do the right thing, and the final refactored driver is safe to load.

## 35.3 How to Get the Most Out of This Chapter

The chapter follows a pattern you will see repeated in every section. First we explain what a mechanism is and what problem it solves. Then we show how userland expects it to behave, so that you understand the contract your driver must honour. Then we look at the real FreeBSD kernel source to see how existing drivers implement the mechanism. And finally we apply it to the `evdemo` driver in a lab.

Several habits will help you absorb the material.

Keep a terminal open to `/usr/src/` so that you can look up any FreeBSD source file the chapter references. Asynchronous I/O is one of the areas where reading real drivers pays off the most, because the patterns are short enough to see in one pass and the variations between drivers teach you what is essential and what is merely stylistic. When the chapter mentions `if_tuntap.c` or `sys_pipe.c`, open the file and look. A minute spent with the real source builds more intuition than any secondhand description.

Keep a second terminal open to your FreeBSD virtual machine so that you can load and unload `evdemo` as the chapter progresses. Type the code yourself the first time you see it. The companion files under `examples/part-07/ch35-async-io/` contain the finished sources, but typing the code builds muscle memory that reading does not. When a section introduces a new callback, add it to the driver, rebuild, reload, and test before moving on.

Pay close attention to locking. Asynchronous I/O is the area where a careless lock acquisition can turn a clean driver into a deadlock or a silent data corruption. When the chapter shows a mutex being acquired before a call to `selrecord()` or `KNOTE_LOCKED()`, notice the order and ask yourself why it must be that way. When a lab instruction says to take the softc mutex before modifying an event queue, take it. Discipline about locking is the single habit that most reliably separates working asynchronous drivers from ones that mostly work.

Finally, remember that asynchronous code tends to reveal its bugs only under pressure. A driver that passes a single-threaded test may still have missed wakeups or races that manifest when two or three threads contend for the same device. Several labs in this chapter include multi-reader stress tests for exactly that reason. Do not skip them. Running your code under contention is the best way to prove that it works for real.

With those habits in mind, let us begin with the difference between synchronous and asynchronous I/O, and the question of when each is the right choice.

## 35.4 1. Synchronous vs. Asynchronous I/O in Device Drivers

Synchronous I/O is the model we have used in almost every driver so far. A user process calls `read(2)`. The kernel dispatches into our `d_read` callback. We either hand back the data that is already available, or we put the calling thread to sleep on a condition variable until data arrives. When the data is ready, we wake the thread, it copies the data out, and `read(2)` returns. The user program blocks for the duration of the call, then resumes.

This pattern is easy to reason about. It matches the way ordinary functions work: you call, you wait, you get a result. It is also a very good fit for devices where the caller's demand drives the device's work. A disk reader asks for data, and the disk controller is told to fetch it. A sensor with a `read_current_value` operation naturally fits a synchronous call. For these devices, the user process always knows when to ask, and the cost of the wait is the cost of the actual I/O.

But for many real devices, the driver's work is not driven by the caller's demand. It is driven by the world.



### 35.4.1 The World Does Not Wait for read()

Consider a keyboard. The device has no opinion about who is calling `read(2)` when a key is pressed. The user hits the key, an interrupt fires, the driver pulls a scan code out of the hardware, and the data is now available. If a userland program is blocked in `read()`, it wakes up and gets the key. If no program is reading, the key sits in a buffer. If several programs share interest in the keyboard, only one of them receives the key under classic blocking semantics, which is almost never what the programmer wants.

Consider a serial port. Bytes arrive at the speed of the wire, independent of any program's readiness to receive them. If the driver blocks every incoming byte behind a reader, it effectively forces the reader to keep one thread always asleep in `read()`, just in case something happens. That thread cannot do anything else. A single well-designed process might want to react to several serial ports, a network socket, a timer, and a keyboard, all at once. The synchronous model cannot express that.

Consider a USB sensor that reports a value only when the measured quantity crosses a threshold. A temperature sensor might raise an event only when the temperature changes by more than half a degree. A motion sensor might fire only when motion is detected. The device's own schedule, not the userland's schedule, decides when data is ready. A reader who blocks in `read()` might wait milliseconds, or seconds, or minutes, or forever.

Each of these situations shares a property: the event is external to the program's request. The driver knows when data is ready. The userland does not. If the userland has to block in `read()` every time to learn what the driver knows, the program is held hostage to the driver's rhythm.

### 35.4.2 Why Busy Waiting Is a Bad Answer

One naive solution is for the userland program to poll the driver. Instead of calling `read()` once and blocking, it calls `read()` in non-blocking mode over and over. `open(/dev/...) with O_NONBLOCK returns immediately if no data is available. The program can spin in a loop, calling read(), doing other work, calling read() again, and so on.`

This pattern is called busy waiting, and it is almost always wrong. It burns CPU even when nothing is happening, because the program keeps asking the driver whether it has work. It misses events that happen between polls. It adds latency to every event: a key pressed a hundred microseconds after the last poll waits until the next poll to be seen. And it scales poorly: a program that watches ten such devices has to poll ten of them in every iteration, making all the problems worse.

Busy waiting is appropriate in exactly one situation: when the polling frequency is known, the device latency is measured in microseconds, and the program has no other work. Even in that case, the right answer is usually to use the CPU's high-precision timing facilities and `usleep()` between polls rather than spin. For any other case, busy waiting is the wrong tool.

The synchronous blocking model and the busy-waiting model are the two endpoints of a spectrum. Both waste resources. What we want is a third option: the userland asks the kernel to tell it when the device is ready, then does other work until the kernel raises its hand. That third option is what asynchronous I/O provides.

### 35.4.3 Asynchronous I/O Is Not Just Non-blocking Read

A common beginner mistake is to think that asynchronous I/O means calling `read()` with `O_NONBLOCK`. It does not. Non-blocking `read()` returns immediately if data is not available; that is a useful property, but it is not

asynchronous I/O on its own. Non-blocking `read()` without a notification mechanism is just busy waiting dressed up a little.

Asynchronous I/O, in the sense this chapter uses the term, is a notification protocol between the driver and the userland. The userland does not have to be reading to learn that the driver has data. The driver does not have to guess who is interested. When the driver's state changes in a relevant way, it notifies its waiters through a well-defined mechanism: `poll/select`, `kqueue`, `SIGIO`, or some combination of those. The waiter wakes up, reads the data, and goes back to waiting.

This distinction matters because it separates three independent concerns in a driver:

The first concern is wait registration. A userland program declares interest in a device by calling `poll()`, `kevent()`, or by enabling `FIOASYNC`. The driver remembers that registration so that it can find the waiter later.

The second concern is wakeup delivery. When the driver's state changes, it calls `selwakeup()`, `KNOTE_LOCKED()`, or `pgsigio()` to deliver the notification. This is a separate operation from producing the data. A driver can produce data without delivering a notification (for example, during an initial fill that happens before anyone has registered). A driver can deliver a notification without producing data (for example, when a device hangs up). And a driver can deliver several notifications for one unit of data, if several mechanisms are registered.

The third concern is event ownership. A `SIGIO` signal is delivered to a specific process or process group. A `knote` belongs to a specific `kqueue`. A `select()` waiter belongs to a specific thread. If the driver cannot match the wakeup to the right owner, notifications get lost or delivered to the wrong party. Each mechanism has its own rules for matching notifications to owners, and we must get those rules right for each one separately.

Keeping these three concerns distinct is one of the main themes of this chapter. A lot of the subtle bugs in asynchronous drivers come from mixing them up. If you find yourself wondering why a specific wakeup call exists or why a specific lock is held, nine times out of ten the answer lies in keeping registration, delivery, and ownership separate.

#### 35.4.4 Real-World Patterns: Event Sources That Want Asynchronous I/O

It helps to name the patterns where asynchronous I/O is the right choice, because once you recognise them you will see them everywhere.

Character input devices are the classic case. A keyboard, a mouse, a touchscreen, a joystick: each produces events when the user interacts with it, at a rate nobody can predict in advance. The user might press a key now, or in five minutes. The driver knows when the event arrives. Userland needs a way to learn.

Serial and network interfaces are another case. Bytes arrive from the wire at the wire's pace. A terminal emulator does not want to block waiting for the next byte, because it also has to redraw its screen, respond to keyboard input, and update its cursor. A network program does not want to block waiting for the next packet, because it usually has to watch several sockets at once.

Sensors that report on condition are a third case. A button that reports "pressed" or "released." A temperature sensor that fires when the measured value crosses a threshold. A motion detector. A door contact. All of these are event-driven in the strict sense: nothing happens until something interesting happens in the world.

Control lines and modem signals are a fourth case. The `CARRIER`, `DSR`, and `RTS` lines on a serial port change state independently of the data flow. A program that cares about them wants to be told when they change, not

to poll them continuously.

Any device that combines several kinds of events into one stream is a fifth case. Consider an evdev input device that aggregates keystrokes, mouse motion, and touchscreen events into a unified event stream. The driver builds an internal queue of events, one record per interesting thing, and readers draw events from the queue. We will build a small version of exactly this pattern later in the chapter, because it illustrates how an event queue, asynchronous notification, and synchronous `read()` semantics combine into one well-structured driver.

### 35.4.5 When Not to Use Asynchronous I/O

For the sake of balance, let us name a few cases where asynchronous I/O is not the right answer.

A driver whose only operation is a bulk transfer at the caller's request has no reason to expose `poll()` or `kqueue()`. If every interaction is a round trip that the user initiates, the synchronous blocking model is both simpler and correct. Adding asynchronous notification to such a driver only increases complexity.

A driver whose data rate is so high that any notification overhead matters might need a different approach entirely. `netmap(4)` and similar kernel-bypass frameworks exist for precisely this case, and they are well beyond the scope of this chapter. An ordinary `kqueue()`-based design is fine up to millions of events per second, but at some point the cost of any notification mechanism becomes a bottleneck.

A driver whose consumer is another kernel subsystem rather than a userland program usually does not need userland-facing asynchronous notification at all. It needs in-kernel synchronization: mutexes, condition variables, `callout(9)`, `taskqueue(9)`. Those are the patterns we studied in earlier chapters, and they remain the right answer when both sides of the event live inside the kernel.

For everything in between, asynchronous I/O is the right tool, and learning it properly is one of the most durable skills a driver author can acquire. The next three sections build up the mental model and the code: first `poll()` and `select()`, then `selrecord()` and `selwakeup()`, then `kqueue()`. The later sections add signals, event queues, and the combined design.

### 35.4.6 A Mental Model for the Rest of the Chapter

Before we move on, let us fix a mental model that will guide the rest of the chapter. Every asynchronous driver has three kinds of code paths.

The first is the producer path. This is where the driver learns that something has happened. For hardware, it is the interrupt handler. For a pseudo-device like `evdemo`, it is whatever code simulates the event. The producer's job is to update the driver's internal state so that a reader who looked now would see the new event.

The second is the waiter path. This is where a userland caller registers interest. The caller's thread enters the kernel through a system call (`poll`, `select`, `kevent`, or `ioctl(FIOASYNC)`), the kernel dispatches to our `d_poll` or `d_kqfilter` callback, and we record the caller's interest in a way that the producer can find later.

The third is the delivery path. This is where the producer notifies the waiters. The producer has just updated state. It calls `selwakeup()`, `KNOTE_LOCKED()`, `pgsigio()`, or some combination of those, and those calls wake up the waiting threads, which then typically call `read()` to pick up the actual data.

This three-path model is the frame through which we will approach every mechanism. When we study `poll()`, we will ask: what is the producer doing, what does the waiter register, and what does the delivery look like? When we study `kqueue()`, we will ask the same three questions. When we study `SIGIO`, same three questions. The mechanisms differ in their details, but they all fit the same shape, and knowing the shape makes each one easier to learn.

With the mental model established, let us look at `poll(2)` and `select(2)`, the oldest and most portable of the three mechanisms.

## 35.5 2. Introducing `poll()` and `select()`

The `poll(2)` and `select(2)` system calls are the original UNIX answer to the question, “how do I wait on multiple file descriptors at once?” They have been in UNIX for decades, they work across every platform that matters, and they are still the most portable way for a userland program to watch several devices, sockets, or pipes in one loop.

They share the same underlying abstraction. A program passes a set of file descriptors and a mask of events it cares about: readable, writable, or exceptional. The kernel examines each descriptor, asks its driver or subsystem whether the event is ready, and if none are, puts the calling thread to sleep until one becomes ready or a timeout expires. When it wakes, the kernel returns which descriptors are now active, and the program can service them.

From the driver’s point of view, `poll` and `select` both funnel into the same `d_poll` callback on a `cdev`. Whether the userland program used `poll(2)` or `select(2)` is invisible to the driver. We answer one question: given this set of events the caller is interested in, which of them are ready right now? If none are ready, we also register the caller so that we can wake it when something becomes ready.

That double role (answer now, register for later) is the heart of the `d_poll` contract. The driver must answer the current state immediately and must not forget the waiter if the answer was “nothing.” Getting either half wrong produces the two classic poll bugs. If the driver reports “not ready” when data actually is ready, the caller goes to sleep and never wakes, because no further event will occur to trigger a wakeup. If the driver fails to register the waiter when nothing is ready, the caller also never wakes, because the driver never knows who to wake when data finally does arrive. Both bugs produce the same symptom (a hung process) and both are a consequence of failing to implement exactly the right pattern.

### 35.5.1 What Userland Expects From `poll()` and `select()`

Before we implement `d_poll`, it helps to know exactly what the userland caller is doing. The user code typically looks something like this:

```
#include <poll.h>
#include <fcntl.h>
#include <unistd.h>

struct pollfd pfd[1];
int fd = open("/dev/evdemo", O_RDONLY);
```

```
pfd[0].fd = fd;
pfd[0].events = POLLIN;
pfd[0].revents = 0;

int r = poll(pfd, 1, 5000); /* wait up to 5 seconds */
if (r > 0 && (pfd[0].revents & POLLIN)) {
 /* data is ready; do a read() now */
 char buf[64];
 ssize_t n = read(fd, buf, sizeof(buf));
 /* ... */
}
```

The user passes an array of struct `pollfd`, each with an `events` mask indicating what it cares about. The kernel returns by writing the `revents` field with the events that are actually ready. The third argument is a timeout in milliseconds, with `-1` meaning “wait forever” and `0` meaning “do not block at all, just poll the state.”

`select(2)` does the same thing with a slightly different API: three `fd_set` bitmaps for readable, writable, and exceptional descriptors, and a timeout as a struct `timeval`. Inside the kernel, both calls normalize into the same operation on each involved descriptor, which ends in our `d_poll` callback.

The caller expects these semantics:

If any of the requested events is currently ready, the call must return promptly with the ready events set.

If none of the requested events is ready and the timeout has not expired, the call must block until one of the events becomes ready or the timeout fires.

If the descriptor is closed or becomes invalid during the call, the kernel returns `POLLNVAL`, `POLLHUP`, or `POLLERR` as appropriate.

The event mask bits that a driver typically deals with are:

`POLLIN` and `POLLRDNORM`, both meaning “data is available to read.” FreeBSD defines `POLLRDNORM` as distinct from `POLLIN`, but in most driver code we treat them together because programs commonly ask for one or the other and expect either to work.

`POLLOUT` and `POLLWRNORM`, both meaning “the device has buffer space to accept a write.” FreeBSD defines `POLLWRNORM` as identical to `POLLOUT`, so in practice they are the same bit.

`POLLPRI`, meaning “out-of-band or priority data is available.” Most character drivers do not have a priority notion and leave this alone.

`POLLERR`, meaning “an error has occurred on the device.” The driver typically sets this when something has gone wrong and the device cannot recover.

`POLLHUP`, meaning “the peer has hung up.” A pty master sees this when the slave closes. A pipe reader sees it when the writer closes. A device driver usually sets this during the detach path, or when a layered service has disconnected.

POLLNVAL, meaning “the request is not valid.” The driver usually leaves this bit to the kernel framework, which sets it when the descriptor is invalid or when the driver has no `d_poll`.

The combination of `POLLHUP` and `POLLIN` is worth noting: when a device closes and it had buffered data, readers should see `POLLHUP` along with `POLLIN`, because the buffered data can still be read even though no more is coming. Well-written userland programs handle this case explicitly.

### 35.5.2 The `d_poll` Callback

Now we can look at the `d_poll` callback itself. Its signature, defined in `/usr/src/sys/sys/conf.h`, is:

```
typedef int d_poll_t(struct cdev *dev, int events, struct thread *td);
```

The `dev` argument is our `cdev`, from which we retrieve the `softc` through `dev->si_drv1`. The `events` argument is the mask of events the caller is interested in. The `td` argument is the calling thread, which we need to pass to `selrecord()` so that the kernel can match future wakeups to the right waiter. The return value is the subset of events that are ready right now.

A skeletal implementation looks like this:

```
static int
evdemo_poll(struct cdev *dev, int events, struct thread *td)
{
 struct evdemo_softc *sc = dev->si_drv1;
 int revents = 0;

 mtx_lock(&sc->sc_mtx);

 if (events & (POLLIN | POLLRDNORM)) {
 if (evdemo_event_ready(sc))
 revents |= events & (POLLIN | POLLRDNORM);
 else
 selrecord(td, &sc->sc_rsel);
 }

 if (events & (POLLOUT | POLLWRNORM))
 revents |= events & (POLLOUT | POLLWRNORM);

 mtx_unlock(&sc->sc_mtx);
 return (revents);
}
```

This is the classic pattern. Let us walk through it one line at a time.

We take the `softc` mutex because we are about to look at the driver’s internal state, and no other thread should be modifying it while we decide whether an event is ready. Holding the lock while we call `selrecord()` is also what closes the race between the answer and the registration, as we will see in a moment.

We look at each event type the caller cares about. For the readable events, we ask the driver whether any data is ready. If it is, we add the matching bits to `revents`. If it is not, we call `selrecord()` to register this thread as a waiter on the `sc_rsel` `selinfo`. That `selinfo` lives in the `softc`, is shared across all potential waiters, and is what we will later pass to `selwakeup()` when data arrives.

For the writable events, we do not have an internal buffer that can fill up in this example, so we always report the device as writable. Many drivers are in this category: writes always fit. Drivers with bounded buffers should check the buffer state the same way they check the read state, and only report `POLLOUT` when there is space.

We release the lock and return the mask of ready events.

Three things about this pattern deserve emphasis.

First, we return immediately with `revents` in every case. The `d_poll` callback does not sleep. If nothing is ready, we register a waiter and return zero. The kernel's generic poll framework takes care of the actual blocking: after `d_poll` returns, the kernel atomically sleeps the thread if no file descriptor returned any events. The driver author does not see this sleep; it is handled entirely by the poll dispatch logic in the kernel.

Second, we must call `selrecord()` only for event types that are not currently ready. If an event is ready and we also call `selrecord()`, we do not break anything (the framework handles this), but it is wasteful: the thread is not going to sleep, so registering it is pointless. The pattern “check, and if not ready then register” keeps the work proportional.

Third, the lock we hold during the check and the `selrecord()` call is the same lock we will take during the producer path when we call `selwakeup()`. This is what prevents the missed wakeup race: if the producer fires after we check the state but before we register the waiter, the producer cannot deliver the wakeup until our `selrecord()` has completed, so the wakeup will find us. We will look at this in detail in Section 3.

### 35.5.3 Registering the `d_poll` Method on the `cdevsw`

To make our driver answer `poll()` calls, we fill in the `d_poll` field of the `struct cdevsw` we pass to `make_dev()` or `make_dev_s()`:

```
static struct cdevsw evdemo_cdevsw = {
 .d_version = D_VERSION,
 .d_name = "evdemo",
 .d_open = evdemo_open,
 .d_close = evdemo_close,
 .d_read = evdemo_read,
 .d_write = evdemo_write,
 .d_ioctl = evdemo_ioctl,
 .d_poll = evdemo_poll,
};
```

If we do not set `d_poll`, the kernel provides a default. In `/usr/src/sys/kern/kern_conf.c`, the default is `no_poll`, which calls `poll_no_poll()`. That default returns the standard readable and writable bits unless the caller asked for anything exotic, in which case it returns `POLLNVAL`. The behaviour makes sense for devices that are always ready, like `/dev/null` and `/dev/zero`, but it is almost never what you want for an event-driven device. For any driver that has real asynchronous semantics, you want to implement `d_poll` yourself.

### 35.5.4 What Real Drivers Look Like

Let us look at two real implementations, because the pattern will become clearer when you see it in production code.

Open `/usr/src/sys/net/if_tuntap.c` and find the function `tunpoll`. It is short enough to quote:

```
static int
tunpoll(struct cdev *dev, int events, struct thread *td)
{
 struct tuntap_softc *tp = dev->si_drv1;
 struct ifnet *ifp = TUN2IFP(tp);
 int revents = 0;

 if (events & (POLLIN | POLLRDNORM)) {
 IFQ_LOCK(&ifp->if_snd);
 if (!IFQ_IS_EMPTY(&ifp->if_snd)) {
 revents |= events & (POLLIN | POLLRDNORM);
 } else {
 selrecord(td, &tp->tun_rsel);
 }
 IFQ_UNLOCK(&ifp->if_snd);
 }
 revents |= events & (POLLOUT | POLLWRNORM);
 return (revents);
}
```

This is our skeleton almost verbatim, with the tun driver's outgoing packet queue as the data source and the `tun_rsel` selinfo as the wait point. The lock here is `IFQ_LOCK`, the queue lock, which the producer also takes before modifying the queue and calling `selwakeuppri()`. That matched locking is what makes the design correct.

Now open `/usr/src/sys/dev/evdev/cdev.c` and find `evdev_poll`. This is a slightly longer and more instructive example because it handles a revoked device explicitly:

```
static int
evdev_poll(struct cdev *dev, int events, struct thread *td)
{
 struct evdev_client *client;
 int ret;
 int revents = 0;

 ret = devfs_get_cdevpriv((void **)&client);
 if (ret != 0)
 return (POLLNVAL);
}
```



```

if (client->ec_revoked)
 return (POLLHUP);

if (events & (POLLIN | POLLRDNORM)) {
 EVDEV_CLIENT_LOCKQ(client);
 if (!EVDEV_CLIENT_EMPTYQ(client))
 revents = events & (POLLIN | POLLRDNORM);
 else {
 client->ec_selected = true;
 selrecord(td, &client->ec_selp);
 }
 EVDEV_CLIENT_UNLOCKQ(client);
}
return (revents);
}

```

Note two extra pieces of behaviour that we did not have in the skeleton.

When the client has been revoked (which happens when the device is being detached while the client still has the file descriptor open), the function returns `POLLHUP` so that the userland program knows to give up. This is the right handling of the detach case. Our skeleton does not do this yet, but the final refactored `evdemo` will.

The driver sets a flag, `ec_selected`, to remember that a waiter has been registered. This lets the producer avoid calling `selwakeup()` for clients that have never polled, which is a small optimization. Most drivers skip this optimization and just call `selwakeup()` every time, which is simpler and still correct.

### 35.5.5 What the User Sees

On the userland side, the caller does not care which implementation we picked. It calls `poll()` with a timeout and sees the result. The first call returns zero if nothing is ready and the timeout expires, or a positive number of ready descriptors otherwise. The second call sees the `revents` bitmask and dispatches to the right handling.

This is the clean separation asynchronous I/O achieves. The user program does not know or care about `selinfo` or `knlist`. It knows only that it asked the kernel “is this ready yet?” and got an answer. The driver’s job is to make that answer truthful and to ensure that the next relevant event will wake the waiter.

### 35.5.6 Wrapping Up Section 2

We now have the userland view of `poll` and `select`, the kernel signature of `d_poll`, and a first skeletal implementation that registers waiters and reports readable events. But the skeleton is still incomplete. We have used `selrecord()` without explaining what it really does with the `struct selinfo`, and we have not yet seen the matching `selwakeup()` call that produces the notification. That is the subject of the next section, and it is where the subtle correctness issues of poll-based asynchronous I/O live.

### 35.6 3. Using `selwakeup()` and `selrecord()`

`selrecord()` and `selwakeup()` are the two halves of the classic poll-wait protocol. They have been in BSD kernels since the original introduction of `select(2)` in 4.2BSD, and they are still the canonical way to implement wait/wakeup for `poll(2)` and `select(2)` in FreeBSD drivers. The pair is simple in outline but subtle in detail, and most of the interesting bugs in poll-based drivers come from getting the subtlety wrong.

This section takes you through the selinfo machinery step by step. First we look at what `struct selinfo` actually contains. Then we look at exactly what `selrecord()` does and what it does not do. Then we look at `selwakeup()` and its companions. Finally we examine the classic missed-wakeup race, the locking discipline that prevents it, and the diagnostic techniques you can use to confirm that your driver is doing the right thing.

#### 35.6.1 `struct selinfo`

Open `/usr/src/sys/sys/selinfo.h` and look at the definition:

```
struct selinfo {
 struct selfdlist si_tdlist; /* List of sleeping threads. */
 struct knlist si_note; /* kernel note list */
 struct mtx *si_mtx; /* Lock for tdlist. */
};

#define SEL_WAITING(si) (!TAILQ_EMPTY(&(si)->si_tdlist))
```

Three fields only. `si_tdlist` is a list of threads currently sleeping on this `selinfo` because they called `selrecord()` and their `poll()` or `select()` call decided to block. `si_note` is a `knlist`, which we will meet in Section 4 when we implement `kqueue` support; it allows the same `selinfo` to serve both `poll()` and `kqueue()` waiters. `si_mtx` is the lock that protects the list.

The `SEL_WAITING()` macro tells you whether any thread is currently parked on this `selinfo`. Drivers occasionally use it to decide whether to bother calling `selwakeup()`, though the wakeup routine itself is cheap enough that the test is usually unnecessary.

Two important habits for `struct selinfo`:

First, the driver must zero-initialize the `selinfo` before first use. The usual way is to embed it in a `softc` that is zeroed by `malloc(..., M_ZERO)`, but if you allocate a `selinfo` separately you must zero it with `bzero()` or equivalent. An uninitialized `selinfo` will crash the kernel the first time `selrecord()` is called on it.

Second, the driver must drain waiters from the `selinfo` before destroying it. The canonical sequence at detach time is `seldrain(&sc->sc_rsel)` followed by `knlist_destroy(&sc->sc_rsel.si_note)`. The `seldrain()` call wakes up any currently parked waiters so that they see the descriptor has become invalid rather than blocking forever. The `knlist_destroy()` call cleans up the knote list for `kqueue` waiters, which we will implement in the next section.

### 35.6.2 What selrecord() Does

`selrecord()` is called from `d_poll` when the driver decides the current event is not ready and the thread will need to wait. Its signature:

```
void selrecord(struct thread *selector, struct selinfo *sip);
```

The implementation lives in `/usr/src/sys/kern/sys_generic.c`. The essence of it is short enough to summarize:

1. The function checks that the thread is in a valid poll context.
2. It takes one of the per-thread preallocated `selfd` descriptors attached to the thread's `seltd` structure.
3. It links that descriptor into the thread's list of active waits and into the `selinfo`'s `si_tdlst`.
4. It remembers the `selinfo`'s mutex on the descriptor, so that the wakeup path knows which lock to take.

The key thing to understand is what `selrecord()` does not do. It does not sleep the thread. It does not block. It does not transition the thread to any blocked state. It merely records the fact that this thread has an interest in this `selinfo`, so that later, when the kernel's poll dispatch code decides to block the thread (if no descriptor returned any events), it knows which `selinfos` the thread is parked on.

After all of a thread's `d_poll` callbacks have returned, the poll dispatch code looks at the results. If any file descriptor returned events, the call returns immediately without blocking. If none did, the thread goes to sleep. The sleep is on a per-thread condition variable inside `struct seltd`, and the wakeup is delivered through that condition variable. The `selinfo`'s role is to link the thread's `seltd` to all the relevant drivers so that each driver can find the thread later.

This separation between “record” and “sleep” is what lets a single `poll()` call monitor many file descriptors. The thread is registered with every `selinfo` of every driver it cares about. When any one of them fires, the wakeup finds the thread through its `seltd` and walks back out to the poll dispatch, which then looks at all the registered file descriptors to see which ones are ready.

### 35.6.3 What selwakeup() Does

`selwakeup()` is called from the producer path when the driver's state changes in a way that might satisfy a waiter. Its signature:

```
void selwakeup(struct selinfo *sip);
```

There is also a variant called `selwakeuppri()` that takes a priority argument, useful when the driver wants to control the priority at which the woken thread resumes. In practice, `selwakeup()` is fine for almost every driver; `selwakeuppri()` is used in a few subsystems that want to emphasize latency at the cost of fairness.

The implementation walks the `selinfo`'s `si_tdlst` and signals each parked thread's condition variable. It also walks the `selinfo`'s `si_note` list and delivers kqueue-style notifications to any knotes attached there, so a single `selwakeup()` call serves both poll waiters and kqueue waiters.

Critically, `selwakeup()` must be called after the driver's internal state has been updated to reflect the new event. If you call `selwakeup()` before the data is visible, the woken thread runs through `d_poll` again, sees

that nothing is ready (because the producer has not yet made it visible), re-registers, and sleeps. When the producer finally updates state, nobody gets woken, because the re-register happened after the wakeup. The driver then has to wait for the next event to unstick the waiter, which may never come.

The correct order is always: update state, then wake up. Never the reverse.

### 35.6.4 The Missed Wakeup Race

The most famous bug in poll-based drivers is the missed wakeup. It looks like this:

```
/* Producer thread */
append_event(sc, ev); /* update state */
selwakeup(&sc->sc_rsel); /* wake waiters */

/* Consumer thread, in d_poll */
if (events & POLLIN) {
 if (event_ready(sc))
 revents |= POLLIN;
 else
 selrecord(td, &sc->sc_rsel);
}
return (revents);
```

If the producer runs between the consumer's `event_ready()` check and the consumer's `selrecord()` call, the wakeup is lost. The consumer saw no event, the producer posted an event and called `selwakeup()` on an empty waiter list, and the consumer then registered. Nobody is going to call `selwakeup()` again until the next event arrives. The consumer now sleeps until that next event, even though an event is already ready.

This is the classic TOCTOU race between the check and the register. The standard fix is to use a single mutex to serialize the check, the register, and the wakeup:

```
/* Producer thread */
mtx_lock(&sc->sc_mtx);
append_event(sc, ev);
mtx_unlock(&sc->sc_mtx);
selwakeup(&sc->sc_rsel);

/* Consumer thread, in d_poll */
mtx_lock(&sc->sc_mtx);
if (events & POLLIN) {
 if (event_ready(sc))
 revents |= POLLIN;
 else
 selrecord(td, &sc->sc_rsel);
}
```

```
mtx_unlock(&sc->sc_mtx);
return (revents);
```

Now the check and the register are atomic with respect to the producer. If the producer updates state before the consumer checks, the consumer sees the event and returns `POLLIN` without registering. If the producer is about to update state while the consumer is in the critical section, the producer has to wait for the consumer to finish. In both cases, the wakeup reaches the consumer.

The important subtlety is that `selwakeup()` is called outside the softc mutex. This is the standard pattern in the FreeBSD kernel: update the state under the lock, drop the lock, deliver the notification. `selwakeup()` itself is safe to call from many contexts, but it takes the `selinfo`'s internal mutex, and we do not want to nest that lock inside an arbitrary driver lock. In practice the rule is, hold the softc lock across the state update, drop it, then call `selwakeup()`.

You will see this pattern across FreeBSD drivers. In `if_tuntap.c` the producer path calls `selwakeuppri()` from outside any driver lock. In `evdev/cdev.c` the same. The producer updates state under its internal lock, releases the lock, and then issues the wakeup. The consumer, in `d_poll`, takes the same lock across the check and the `selrecord()`. That discipline eliminates the missed wakeup race.

### 35.6.5 Thinking About the Lock

Why does this work? Because the lock serializes two specific operations: the producer's state update and the consumer's check plus register. The `selwakeup()` call and the thread's subsequent sleep are outside the lock, but that is fine, because the condition variable semantics of the underlying mechanism handle that race separately.

Here is the argument in more detail. Suppose the consumer acquires the lock first. It checks the state, sees nothing, calls `selrecord()` to register, and releases the lock. Some time later the producer acquires the lock, updates the state, releases the lock, and calls `selwakeup()`. The consumer is already registered, so the wakeup finds it. Good.

Now suppose the producer acquires the lock first. It updates the state, releases the lock, and calls `selwakeup()`. The consumer was not yet registered, so the wakeup finds no waiters. That is fine because the consumer has not yet reached the point where it would have slept; the consumer is still about to acquire the lock. When the consumer does acquire the lock, it checks the state, sees the event (because the producer has already updated it), and returns `POLLIN` without calling `selrecord()`. The consumer is correctly notified.

The third case is the tricky one. The consumer has just checked the state (under the lock) and is about to call `selrecord()`, but in fact, because the lock is held the whole time, this case cannot arise. The producer cannot update the state until the consumer releases the lock, at which point the consumer has already registered.

So the lock discipline is: always hold the lock across the consumer's check and register, and always hold the lock across the producer's state update. The `selwakeup()` call itself happens outside the lock, because it has its own internal synchronization.

### 35.6.6 Common Mistakes

A few mistakes are worth calling out explicitly.

Calling `selwakeup()` inside the state-update lock is wrong in most cases because `selwakeup()` itself may need to take other locks (the `selinfo` mutex, the thread's `selinfo` queue lock). Doing this from within the `softc` mutex creates a lock-ordering opportunity that is easy to get wrong. The rule of thumb is, update under the lock, drop, then `selwakeup()`.

Forgetting to wake up all interested `selinfos` is the other common mistake. If the driver has separate read and write `selinfos` (say, one for `POLLIN` waiters and one for `POLLOUT` waiters), it must wake the right one when state changes. Waking the wrong one means the actual waiter sleeps forever.

Calling `selrecord()` without holding any lock produces a time window in which the event can arrive without delivering the wakeup. This is the race we just analyzed, and the fix is always the same: hold the lock.

Calling `selrecord()` every time, even when data is ready, is not a correctness bug but it is a pointless load on the per-thread `selfd` pool. If data is ready, the thread is not going to sleep, so registering it is wasted work. The pattern “check; if ready, return; if not, register” is the correct one.

Calling `selwakeup()` on a destroyed `selinfo` is a crash waiting to happen. The detach path must call `selfdrain()` before freeing the `selinfo` or the surrounding `softc`.

### 35.6.7 Diagnostic Techniques

When a driver's poll support is not working, a few tools help you isolate the problem.

The first tool is `top(1)`. Load the driver, open a descriptor in a userland program, and have the program call `poll()` with a long timeout. Look at the program in `top -H` and check the `WCHAN` column. If the poll is working correctly, the thread's wait channel will be `select` or something similar. If the thread is in some other state (running, runnable, short sleep), the poll call may have returned prematurely, or the program may be spinning.

The second tool is counters on the driver. Add a counter for each call to `selrecord()`, one for each call to `selwakeup()`, and one for each time `d_poll` returns a ready mask. After a test, print these counters through `sysctl`. If `selrecord()` fires but `selwakeup()` never does, the producer path is never triggering. If `selwakeup()` fires but the program stays asleep, you probably have a missed wakeup because the state-update and register happen outside the lock.

The third tool is `ktrace(1)` and `kdump(1)`. Run the test program under `ktrace`, and the dump will show every system call and its timing. A program that calls `poll()` and blocks will show up as a `RET poll` entry after the wakeup, and the timestamp will tell you when the wakeup actually arrived. If the producer event happened at time `T` and the wakeup arrived seconds later, you have a bug.

The fourth tool is `DTrace`, which can instrument `selwakeup` itself. A script that probes `fbt:kernel:selwakeup:entry` and prints the calling driver's `softc` pointer shows every wakeup in the system. If your driver's wakeup never fires, `DTrace` will tell you so in milliseconds.

### 35.6.8 Closing the Loop: evdemo With Poll Support

Putting the pieces together, here is the minimum additional code our `evdemo` driver needs to support `poll()` correctly:

```
/* In the softc */
struct evdemo_softc {
 /* ... existing fields ... */
 struct selinfo sc_rsel; /* read selectors */
};

/* At attach */
knlist_init_mtx(&sc->sc_rsel.si_note, &sc->sc_mtx);

/* In d_poll */
static int
evdemo_poll(struct cdev *dev, int events, struct thread *td)
{
 struct evdemo_softc *sc = dev->si_drv1;
 int revents = 0;

 mtx_lock(&sc->sc_mtx);
 if (events & (POLLIN | POLLRDNORM)) {
 if (sc->sc_nevents > 0)
 revents |= events & (POLLIN | POLLRDNORM);
 else
 selrecord(td, &sc->sc_rsel);
 }
 if (events & (POLLOUT | POLLWRNORM))
 revents |= events & (POLLOUT | POLLWRNORM);
 mtx_unlock(&sc->sc_mtx);

 return (revents);
}

/* In the producer path (for evdemo, this is the event injection
 * routine triggered from a callout or ioctl) */
static void
evdemo_post_event(struct evdemo_softc *sc, struct evdemo_event *ev)
{
 mtx_lock(&sc->sc_mtx);
 evdemo_enqueue(sc, ev);
 mtx_unlock(&sc->sc_mtx);
 selwakeup(&sc->sc_rsel);
}

/* At detach */
seldrain(&sc->sc_rsel);
knlist_destroy(&sc->sc_rsel.si_note);
```

Notice that we called `knlist_init_mtx()` on the `selinfo`'s embedded `si_note` knlist even though we are not yet implementing `kqueue`. This costs us almost nothing and makes the `selinfo` compatible with `kqueue` support we will add in Section 4. If you do not pre-initialize `si_note`, the first `selwakeup()` call that tries to walk the knlist will crash. Many drivers initialize the knlist during attach as a matter of habit.

Also notice that the `evdemo_post_event` helper holds the `softc` mutex while it updates the event count, drops the mutex, and then calls `selwakeup()`. That is the standard producer pattern, and it is the one we will reuse throughout the rest of the chapter.

### 35.6.9 Wrapping Up Section 3

At this point you have all the conceptual and practical pieces of poll-based asynchronous I/O. You know the contract, the kernel structures, the correct locking discipline, and the common failure modes. You can take an existing blocking driver, add `d_poll` support, and have it behave correctly under `poll(2)` and `select(2)`.

The problem is that `poll(2)` and `select(2)` have well-known scalability limitations. Every call re-declares the full set of descriptors the caller is interested in, which is  $O(N)$  per call. For programs watching thousands of descriptors, this overhead dominates. FreeBSD has offered a better mechanism since the late 1990s, namely `kqueue(2)`, and that is the subject of the next section.

## 35.7 4. Supporting `kqueue` and `EVFILT_READ/EVFILT_WRITE`

`kqueue(2)` is FreeBSD's scalable event-notification facility. Unlike `poll(2)` and `select(2)`, which require the userland program to re-declare its interests on every call, `kqueue(2)` lets the program register interests once and then ask only for events that have actually fired. For a program watching ten thousand file descriptors where only a few are active, this is the difference between a fast, interactive program and a slow, loaded one.

`kqueue` is also more expressive than `poll`. Beyond the basic readable and writable filters, it offers filters for signals, timers, file-system events, process lifecycle events, user-defined events, and several other categories. A driver that wants to participate only in the classic readable and writable notifications still fits cleanly into the framework; the broader features are available if needed.

From the driver's point of view, `kqueue` support adds one callback to the `cdevsw`, `d_kqfilter`, and one set of filter operations, a `struct filterops`, that provides the lifecycle and event-delivery functions for each filter type. The whole mechanism reuses the `struct selinfo` we met in Section 3, so drivers that already support `poll()` can add `kqueue` support by writing about a hundred lines of extra code and calling a handful of new APIs.

### 35.7.1 What `kqueue` Looks Like to Userland

Before we implement the driver side, let us see what the user program looks like. A caller opens a `kqueue`, registers interest in a file descriptor, and then reaps events:

```
#include <sys/event.h>

int kq = kqueue();
```



```

int fd = open("/dev/evdemo", O_RDONLY);

struct kevent change;
EV_SET(&change, fd, EVFILT_READ, EV_ADD | EV_CLEAR, 0, 0, NULL);
kevent(kq, &change, 1, NULL, 0, NULL);

for (;;) {
 struct kevent ev;
 int n = kevent(kq, NULL, 0, &ev, 1, NULL);
 if (n > 0 && ev.filter == EVFILT_READ) {
 char buf[256];
 ssize_t r = read(fd, buf, sizeof(buf));
 /* ... */
 }
}

```

The `EV_SET` macro constructs a `struct kevent` describing the interest: “watch file descriptor `fd` for `EVFILT_READ` events, using edge-triggered (`EV_CLEAR`) semantics, and keep it active (`EV_ADD`).” The first `kevent()` call registers that interest. The loop then calls `kevent()` in blocking mode, asking for the next event, and services it when it arrives.

The driver never sees the `kqueue` file descriptor or the `kevent` structure directly. It sees only the per-interest `struct knote` and its attached `struct filterops`. The registration flows through the framework to our `d_kqfilter` callback, which chooses the right filter operations and attaches the `knote` to our `softc`. The delivery flows through `KNOTE_LOCKED()` calls in the producer path, which walks our `knote` list and notifies each attached `kqueue` of the ready event.

### 35.7.2 The Data Structures

Two structures matter on the driver side: `struct filterops` and `struct knlist`.

`struct filterops`, defined in `/usr/src/sys/sys/event.h`, holds the per-filter lifecycle functions:

```

struct filterops {
 int f_isfd;
 int (*f_attach)(struct knote *kn);
 void (*f_detach)(struct knote *kn);
 int (*f_event)(struct knote *kn, long hint);
 void (*f_touch)(struct knote *kn, struct kevent *kev, u_long type);
 int (*f_userdump)(struct proc *p, struct knote *kn,
 struct kinfo_knote *kin);
};

```

The fields we care about for a driver are:

`f_isfd` is 1 if the filter is attached to a file descriptor. Almost all driver filters have this set to 1. A filter that watches something not tied to an fd (like `EVFILT_TIMER`) would set it to 0.

`f_attach` is called when a knote is being attached to a newly registered interest. Many drivers leave this as `NULL` because all the attach work happens in `d_kqfilter` itself.

`f_detach` is called when a knote is being removed. The driver uses this to unregister the knote from its internal knote list.

`f_event` is called to evaluate whether the filter's condition is currently satisfied. It returns non-zero if yes, zero if no. It is the kqueue equivalent of the state check in `d_poll`.

`f_touch` is used when the filter supports `EV_ADD/EV_DELETE` updates that should not be treated as full re-registration. Most drivers leave this as `NULL` and accept the default behaviour.

`f_userdump` is used for kernel introspection and can be left `NULL` in driver code.

`struct knlist`, defined in the same header, holds a list of knotes attached to a particular object. It carries pointers to the object's lock operations so that the kqueue framework can grab and release the right lock when delivering events:

```
struct knlist {
 struct klist kl_list;
 void (*kl_lock)(void *);
 void (*kl_unlock)(void *);
 void (*kl_assert_lock)(void *, int);
 void *kl_lockarg;
 int kl_autodestroy;
};
```

Drivers rarely touch this structure directly. The framework provides helper functions, starting with `knlist_init_mtx()` for the common case of a `knlist` protected by a single mutex.

### 35.7.3 Initializing a knlist

The simplest way to initialize a `knlist` is:

```
knlist_init_mtx(&sc->sc_rsel.si_note, &sc->sc_mtx);
```

The first argument is the `knlist` to initialize. The second is the driver's mutex. The framework stores the mutex and will take it when needed to protect the knote list. The knote list is usually embedded in a `struct selinfo`, as we saw in the previous section; reusing the same `selinfo` for both poll and kqueue waiters lets a single `selwakeup()` call cover both mechanisms.

For a driver that is already zeroing the `softc` through `M_ZERO`, the initialization is just this one call during attach.

### 35.7.4 The d\_kqfilter Callback

The `d_kqfilter` callback is the entry point for kqueue registration. Its signature, in `/usr/src/sys/sys/conf.h`, is:

```
typedef int d_kqfilter_t(struct cdev *dev, struct knote *kn);
```

The `dev` argument is our `cdev`. The `kn` argument is the knote being registered. The callback decides which filter operations apply, attaches the knote to our knote list, and returns zero on success.

A minimal implementation for a driver that supports `EVFILT_READ`:

```
static int
evdemo_kqfilter(struct cdev *dev, struct knote *kn)
{
 struct evdemo_softc *sc = dev->si_drv1;

 switch (kn->kn_filter) {
 case EVFILT_READ:
 kn->kn_fop = &evdemo_read_filterops;
 kn->kn_hook = sc;
 knlist_add(&sc->sc_rsel.si_note, kn, 0);
 return (0);
 default:
 return (EINVAL);
 }
}
```

Let us walk through this.

The switch on `kn->kn_filter` decides which filter type we are dealing with. A driver that supports only `EVFILT_READ` returns `EINVAL` for anything else. A driver that also supports `EVFILT_WRITE` has a second case that points to a different filter operations structure.

We set `kn->kn_fop` to the filter operations for this filter type. The kqueue framework calls these ops as the knote's lifecycle progresses.

We set `kn->kn_hook` to the `softc`. The knote has this generic pointer for per-driver use. Our filter functions will pull the `softc` back out of `kn->kn_hook` when they are called.

We call `knlist_add()` to link the knote into our knote list. The third argument, `islocked`, is zero here because we are not holding the `knlist` lock at this point. If we were, we would pass 1.

Return zero to indicate success.

### 35.7.5 The filterops Implementation

The filter operations are where the per-filter behaviour lives. For `EVFILT_READ` on `evdemo`, they look like this:

```

static int
evdemo_kqread(struct knote *kn, long hint)
{
 struct evdemo_softc *sc = kn->kn_hook;
 int ready;

 mtx_assert(&sc->sc_mtx, MA_OWNED);

 kn->kn_data = sc->sc_nevents;
 ready = (sc->sc_nevents > 0);

 if (sc->sc_detaching) {
 kn->kn_flags |= EV_EOF;
 ready = 1;
 }

 return (ready);
}

static void
evdemo_kqdetach(struct knote *kn)
{
 struct evdemo_softc *sc = kn->kn_hook;

 knlist_remove(&sc->sc_rsel.si_note, kn, 0);
}

static const struct filterops evdemo_read_filterops = {
 .f_isfd = 1,
 .f_attach = NULL,
 .f_detach = evdemo_kqdetach,
 .f_event = evdemo_kqread,
};

```

The `f_event` function, `evdemo_kqread`, is called every time the framework wants to know whether the filter is ready. It looks at the `softc`, reports the number of available events in `kn->kn_data` (a convention that `kqueue` users rely on to learn how much data is available), and returns non-zero if at least one event is waiting. It also flips the `EV_EOF` flag when the device is detaching, which lets the userland see that no more events are coming.

Note the assertion that the `softc` mutex is held. The framework takes our `knlist`'s lock, which we told it is the `softc` mutex via `knlist_init_mtx`. Because the `f_event` callback is invoked inside that lock, we can look at `sc_nevents` and `sc_detaching` safely.

The `f_detach` function removes the `knote` from our `knlist` when the userland no longer cares about this registration.

The constant `evdemo_read_filterops` is what `d_kqfilter` pointed at in the previous subsection. `f_isfd = 1` tells the framework that this filter is tied to a file descriptor, which is the correct value for any driver-level filter.

### 35.7.6 Delivering Events Through `KNOTE_LOCKED`

On the producer side, we need to notify registered knotes when the driver's state changes. The macro is `KNOTE_LOCKED()`, defined in `/usr/src/sys/sys/event.h`:

```
#define KNOTE_LOCKED(list, hint) knote(list, hint, KNF_LISTLOCKED)
```

It takes a `knlist` pointer and a hint. The hint is passed through to each knote's `f_event` callback, giving the producer a way to pass context (for example, a specific event type) to the filter. Most drivers pass zero.

The `KNOTE_LOCKED` variant is what you want when you are already holding the `knlist`'s lock. The `KNOTE_UNLOCKED` variant is used when you are not. Since the `knlist`'s lock is usually your `softc` mutex, and since the rest of the producer path is holding that lock anyway, `KNOTE_LOCKED` is the usual choice.

Adding it to our producer path:

```
static void
evdemo_post_event(struct evdemo_softc *sc, struct evdemo_event *ev)
{
 mtx_lock(&sc->sc_mtx);
 evdemo_enqueue(sc, ev);
 KNOTE_LOCKED(&sc->sc_rsel.si_note, 0);
 mtx_unlock(&sc->sc_mtx);
 selwakeup(&sc->sc_rsel);
}
```

We now notify both `kqueue` and `poll` waiters from the same producer. `KNOTE_LOCKED` inside the `softc` mutex walks the knote list and evaluates each knote's `f_event`, queuing notifications to any `kqueues` that have active waiters. `selwakeup` outside the lock wakes up `poll()` and `select()` waiters. The two mechanisms are independent and neither interferes with the other.

### 35.7.7 Detach: Cleaning Up `knlist`

At detach time, the driver must drain the `knlist` before destroying it. The clean sequence is:

```
knlist_clear(&sc->sc_rsel.si_note, 0);
seldrain(&sc->sc_rsel);
knlist_destroy(&sc->sc_rsel.si_note);
```

`knlist_clear()` removes every knote that is still attached. After this call, any userland program that still has a `kqueue` registration will see the knote go away on its next reap. `seldrain()` wakes up any parked `poll()` waiters so that they return. `knlist_destroy()` checks that the list is empty and frees the internal resources.

The order matters. If you destroy the knlist without clearing it first, the destroy will panic on the assertion that the list is empty. If you clear the knlist but leave poll waiters parked, they will sleep until something wakes them, which is wasteful. Follow the sequence above and the detach path is clean.

### 35.7.8 A Fuller Example: Pipes

Open `/usr/src/sys/kern/sys_pipe.c` and look at the pipe `kqfilter` implementation. It is one of the most extensive examples in the kernel, and it is worth reading in full because pipes support both read and write filters with proper EOF handling. The key pieces are the two filterops structures:

```
static const struct filterops pipe_rfiltops = {
 .f_isfd = 1,
 .f_detach = filt_pipedetach,
 .f_event = filt_piperead,
 .f_userdump = filt_pipedump,
};

static const struct filterops pipe_wfiltops = {
 .f_isfd = 1,
 .f_detach = filt_pipedetach,
 .f_event = filt_pipewrite,
 .f_userdump = filt_pipedump,
};
```

And the read filter's event function:

```
static int
filt_piperead(struct knote *kn, long hint)
{
 struct file *fp = kn->kn_fp;
 struct pipe *rpipe = kn->kn_hook;

 PIPE_LOCK_ASSERT(rpipe, MA_OWNED);
 kn->kn_data = rpipe->pipe_buffer.cnt;
 if (kn->kn_data == 0)
 kn->kn_data = rpipe->pipe_pages.cnt;

 if ((rpipe->pipe_state & PIPE_EOF) != 0 &&
 ((rpipe->pipe_type & PIPE_TYPE_NAMED) == 0 ||
 fp->f_pipegen != rpipe->pipe_wgen)) {
 kn->kn_flags |= EV_EOF;
 }
}
```

```

 return (1);
 }
 kn->kn_flags &= ~EV_EOF;
 return (kn->kn_data > 0);
}

```

Note the handling of EOF, the explicit clearing of `EV_EOF` when the pipe is no longer at EOF (which matters if the named pipe has a new writer), and the use of `kn->kn_data` to report the amount of available data. These are the details a finished driver gets right.

### 35.7.9 The Anatomy of struct knote

We have been passing a `struct knote` pointer around without looking closely at it, but the driver's life is easier once we know what it contains. `struct knote`, defined in `/usr/src/sys/sys/event.h`, is the kernel's per-registration record. Every call to `kevent(2)` that registers an interest creates exactly one knote, and that knote persists until the registration is removed. For a driver, the knote is the unit of currency: every `knlist` operation takes a knote, every filter callback receives a knote, and every delivery walks a list of them. Knowing what lives inside the structure turns the callback contracts we have been following into something we can reason about instead of memorise.

The fields the driver cares about are a small subset of the whole structure, but each one repays attention.

`kn_filter` identifies which filter type the userland asked for. Inside `d_kqfilter`, this is what we switch on: `EVFILT_READ`, `EVFILT_WRITE`, `EVFILT_EXCEPT`, and so on. The value comes from the `filter` field of the `struct kevent` that userland submitted. A driver that supports only one filter type checks this field and rejects any mismatch with `EINVAL`.

`kn_fop` is the pointer to the `struct filterops` table that will service this knote for the rest of its life. The driver sets this inside `d_kqfilter`. After that point, the framework calls through this pointer to reach our `attach`, `detach`, `event`, and `touch` callbacks. The `filterops` table is always `static const` in the drivers we examine, because the framework does not take a reference on it and the driver is expected to keep the pointer valid for the knote's lifetime.

`kn_hook` is a generic per-driver pointer. The driver typically sets it to the `softc`, to a per-client state record, or to whichever object the filter should react to. The framework never reads or writes it. When the filter callbacks run, they pull the driver state out of `kn_hook` rather than going through a global lookup, which both avoids the lookup cost and avoids a class of lock-ordering problems that global lookups can introduce.

`kn_hookid` is an integer companion to `kn_hook`, available for per-driver tagging. Most drivers leave it alone.

`kn_data` is how the filter's `f_event` callback communicates "how much is ready" back to the userland. For readable filters, drivers conventionally store the number of bytes or records available. For writable filters, they store the amount of space available. Userland reads this through the `data` field of the returned `struct kevent`, and tools like `libevent` rely on that convention. The `/dev/klog` driver stores a raw byte count here, while the `evdev` driver stores the queue depth scaled into bytes by multiplying the record count by `sizeof(struct input_event)`, because `evdev` clients read `struct input_event` records rather than raw bytes.

`kn_sfflags` and `kn_sdata` hold the per-registration flags and data that userland requested through the `fflags` and `data` fields of `struct kevent`. Filters that support fine-grained control, like `EVFILT_TIMER` with its period

or `EVFILT_VNODE` with its note mask, look at these to decide how to behave. Simple driver filters usually ignore them.

`kn_flags` holds the delivery-time flags that the framework passes through to the userland on the next reap. The one every driver uses is `EV_EOF`, which signals “no more data will ever arrive from this source.” Drivers set `EV_EOF` in `f_event` when the device is being detached, when a pseudo-terminal’s peer has closed, when a pipe has lost its writer, or whenever the readiness signal has become permanent.

`kn_status` is internal state owned by the framework: `KN_ACTIVE`, `KN_QUEUED`, `KN_DISABLED`, `KN_DETACHED`, and a handful of others. Drivers must not modify it. The driver’s job is simply to report readiness through `f_event`; the framework updates `kn_status` accordingly.

`kn_link`, `kn_selnext`, and `kn_tqe` are the linked-list linkage fields used by the various `kqueue` framework lists. The `knlist` helpers manipulate them on our behalf. Drivers should never touch them directly.

Put together, these fields tell a simple story. The driver creates a knote’s association with its filter operations inside `d_kqfilter`, sets `kn_hook` and optionally `kn_hookid` so the filter callbacks can recover their context, and then lets the framework manage linkage and status. The driver owns readiness reporting through `f_event` and nothing else. The handoff between driver and framework is clean, and most driver bugs in this area come from trying to reach across that boundary, either by modifying framework-owned status flags or by retaining stale pointers into the knote after `f_detach` has fired.

One point worth emphasising: the knote outlives any single `f_event` call, but it does not outlive `f_detach`. Once the framework invokes `f_detach`, the knote is being torn down; the driver must unhook it from any internal structure it is attached to and must not keep the pointer. The `kn_hook` pointer, which is driver-owned, must be treated the same way. If the driver was keeping a back-pointer from a `softc` field to the knote for any reason (uncommon, but sometimes useful for driver-initiated detach), it must clear that back-pointer during `f_detach` before the framework frees the knote.

### 35.7.10 Inside struct knlist: How the Driver’s Waiting Room Works

`struct knlist`, declared in `/usr/src/sys/sys/event.h`, is where a driver accumulates the knotes that are currently interested in one of its notification sources. Every driver object that can wake `kqueue` waiters owns at least one `knlist`. The pipe object owns two, one for readers and one for writers. The `tty` owns two as well, `t_inpoll` and `t_outpoll`, each with its own `knlist`. The `evdev` client object owns one per client. In our `evdemo` driver, we piggy-back on the `struct selinfo.si_note` we already have for `poll`, so the same `knlist` is the one waking both `poll` and `kqueue` consumers.

The structure itself is small:

```
struct knlist {
 struct klist kl_list;
 void (*kl_lock)(void *);
 void (*kl_unlock)(void *);
 void (*kl_assert_lock)(void *, int);
 void *kl_lockarg;
 int kl_autodestroy;
};
```



`kl_list` is the singly-linked list head of `struct knote` entries, with linkage through each `knote`'s `kn_selnext` field. The list head is manipulated by the framework, never by the driver directly.

`kl_lock`, `kl_unlock`, and `kl_assert_lock` are function pointers that the framework uses when it needs to take the object's lock. The `knlist` does not own a lock of its own; it borrows the driver's locking regime. This is why a `struct selinfo` can carry a `knlist` without creating a separate lock: the lock is whatever the driver has already declared.

`kl_lockarg` is the argument passed to those lock functions. When we initialise a `knlist` with `knlist_init_mtx(&kn1, &sc->sc_mtx)`, the framework stores `&sc->sc_mtx` in `kl_lockarg` and arranges for the lock callbacks to wrap `mtx_lock` and `mtx_unlock`. The driver never sees this wiring and never needs to.

`kl_autodestroy` is a flag used by a few specific subsystems, most notably AIO, where the `knlist` lives inside the `struct kaiocb` and must be torn down automatically when the request completes. Driver code almost never sets this. The `aio_filtops` path in `/usr/src/sys/kern/vfs_aio.c` is the canonical use, and it is worth remembering that the flag exists so that reading that file later does not surprise you.

The lock contract deserves emphasis because it is the single most common source of `kqueue` driver bugs. When the framework calls our `f_event`, it holds the `knlist` lock, which is our softc mutex. Our `f_event` may read softc state but must not take the softc mutex again (it is already ours), must not sleep, and must not block on any other lock that could be held across an `f_event` invocation. When we invoke `KNOTE_LOCKED`, we are asserting that we already hold the lock, so the framework skips locking on its way through the list. When we invoke `KNOTE_UNLOCKED`, the framework takes and releases the lock on our behalf. Mixing the two styles inside one producer path is a classic source of subtle double-lock panics under load.

The unification with `struct selinfo` is worth noticing. Back in Section 3 we treated `struct selinfo` as a poll-only concept, but it actually embeds a `struct knlist` in its `si_note` member. This is why a driver that already supports `poll()` has the infrastructure for `kqueue()` sitting in its softc: adding `kqueue` is largely a matter of initialising the `knlist` with `knlist_init_mtx` and wiring up the filter operations. The producer path already calls into `selwakeup()`, which itself walks `si_note` under the appropriate lock and notifies any attached `knotes`. Doing the notification explicitly with `KNOTE_LOCKED(&sc->sc_rsel.si_note, 0)` is clearer and lets us choose exactly when the `kqueue` fan-out happens relative to any other producer work. In the drivers we will read below, both styles appear; either one is correct as long as the locking is consistent.

### 35.7.11 The `knlist` Lifecycle in Detail

The lifecycle of a `knlist` follows the lifecycle of the driver object that owns it. A `knlist` comes into existence during attach (either the driver's attach entry point for a real hardware driver or the `SYSINIT` for a pseudo-device), lives through the open-read-close cycles of userland consumers, and is torn down at detach. The functions we need, all declared in `/usr/src/sys/sys/event.h` and implemented in `/usr/src/sys/kern/kern_event.c`, are `knlist_init`, `knlist_init_mtx`, `knlist_add`, `knlist_remove`, `knlist_clear`, and `knlist_destroy`.

`knlist_init_mtx` is the one almost every driver calls. It initialises the list head, configures the `knlist` to use `mtx_lock`/`mtx_unlock` with the driver's mutex as the argument, and marks the `knlist` as live. The caller passes a pointer to the `knlist` (usually `&sc->sc_rsel.si_note` or, for drivers with per-direction notification, `&sc->sc_wsel.si_note` as well) and a pointer to a mutex that already exists in the driver.

`knlist_init` is the general form, used when the driver's lock regime is not a simple mutex. It accepts three

function pointers (lock, unlock, assert), an argument pointer passed to those functions, and the underlying list head. Pipes use the `_mtx` form with their pipe-pair mutex; socket buffers use a customised `knlist_init` because they have their own locking discipline. Most drivers do not need the general form.

`knlist_add` is called from `d_kqfilter` to link a newly registered knote into the list. Its prototype is `void knlist_add(struct knlist *knl, struct knote *kn, int islocked)`. The `islocked` argument tells the function whether the caller already holds the `knlist` lock. If it is zero, the function takes the lock for us. If it is one, we are asserting that we already hold it. Drivers that do no extra locking inside `d_kqfilter` pass zero; drivers like `/dev/klog` that took the `msgbuf` lock on entry pass one. Either pattern is correct; the choice depends on what the driver wants to protect around the `knlist_add` call.

`knlist_remove` is the reverse operation, normally called from the `f_detach` callback. Its prototype is `void knlist_remove(struct knlist *knl, struct knote *kn, int islocked)`. The framework invokes `f_detach` with the `knlist` lock already held, so `islocked` is one in that context. If for any reason the driver needs to remove a specific knote from outside `f_detach` (which is unusual and rarely correct), it must arrange its own locking.

`knlist_clear` is the bulk-removal function used at driver detach time. It walks the list, removes every knote, and marks each of them with `EV_EOF | EV_ONESHOT` so that the userland sees a final event and the registration is discarded. The signature `void knlist_clear(struct knlist *knl, int islocked)` is actually a wrapper around `knlist_cleardel` in `/usr/src/sys/kern/kern_event.c` with a `NULL` `struct thread *` and the kill flag set, meaning “remove everything.” Drivers call this from `detach` right before tearing down the `knlist`.

`knlist_destroy` releases the internal machinery of the `knlist`. Before calling it, the `knlist` must be empty. If you destroy a `knlist` with live knotes, the kernel asserts and panics. This is why the detach sequence we saw earlier is rigid:

```
knlist_clear(&sc->sc_rsel.si_note, 0);
selldrain(&sc->sc_rsel);
knlist_destroy(&sc->sc_rsel.si_note);
```

`knlist_clear` empties the list. `selldrain` wakes any `poll()` waiters that are still parked on the same `selinfo`, so their waiting threads return from the kernel. `knlist_destroy` tears down the internals and validates that the list is empty. If any of these steps is skipped, the detach becomes unsafe: live knotes trying to call an unloaded driver’s `f_event` would crash the kernel; a poll waiter whose `selinfo` has been freed would wake to a dangling pointer.

Two further points are worth noticing in the implementation of `knlist_remove` in `/usr/src/sys/kern/kern_event.c`. It walks into the internal `knlist_remove_kq` helper which also acquires the `kq` lock so that the removal is coherent with any in-progress event dispatch. And it sets `KN_DETACHED` in `kn_status` to signal to the rest of the framework that this knote is gone. Drivers never observe `KN_DETACHED` directly, but understanding that it exists explains why concurrent detach and event delivery can race safely: the framework’s internal state machine keeps them consistent.

### 35.7.12 The kqfilter Callback Contract

`d_kqfilter` is called from the `kqueue` registration path in `/usr/src/sys/kern/kern_event.c`, specifically from `kqueue_register` via the file descriptor's `fo_kqfilter` method. By the time the callback runs, the framework has already validated the file descriptor, allocated the `struct knote`, and filled in the userland's request. Our job is narrow: pick the right filterops, attach to the right `knlist`, and return zero.

What `d_kqfilter` must do. It must inspect `kn->kn_filter` to decide which filter type the userland is asking for. It must set `kn->kn_fop` to a valid `struct filterops` for that type. It must attach the `knote` to a `knlist` that belongs to our driver, typically by calling `knlist_add`. And it must return zero on success or a sensible `errno` on failure. If the driver cannot service the requested filter, `EINVAL` is the right answer.

What `d_kqfilter` must not do. It must not sleep, because the `kqueue` registration path holds locks that are not safe to sleep under. It must not allocate memory with `M_WAITOK`, for the same reason. It must not call any function that can block on another process. If the driver needs more than a fast lookup and a `knlist` insertion, it is doing something wrong. The callback is essentially a fast-path wiring operation.

Lock state on entry is worth understanding. The framework does not hold the `knlist` lock when it calls `d_kqfilter`. We can therefore pass `islocked = 0` to `knlist_add` if we have not taken the `knlist`'s lock ourselves. If our driver needs to look at softc state as part of the filter-selection logic, for example to report `ENODEV` on a revoked `cdev` as the `evdev` driver does, we can take the `softc` mutex ourselves, check the state, do the `knlist_add` with `islocked = 1`, and release the mutex before returning. The `evdev` example below shows exactly that pattern.

Returning a non-zero value from `d_kqfilter` means “the userland will get this `errno` back from `kevent(2)`.” It does not mean “try again.” A driver that returns `EAGAIN` will confuse userland because `kevent` does not interpret that value the way `read` does. Stick to `EINVAL` for unsupported filters and `ENODEV` for revoked or torn-down devices, and avoid clever error returns.

One subtlety about when `d_kqfilter` is invoked: a single `kevent(2)` call that registers a new interest with `EV_ADD` enters the framework, finds that no `knote` exists yet for this (file, filter) pair, allocates one, and then calls `fo_kqfilter` on the file descriptor's `fileops`. That is where our `d_kqfilter` is reached, through the `cdev` `fileops` table. If the caller is instead updating an existing registration (for example, toggling between enabled and disabled with `EV_ENABLE/EV_DISABLE`), our callback is not involved; the framework handles that internally through `f_touch` or direct status manipulation.

### 35.7.13 Worked Example: The /dev/klog Driver

The simplest real driver-side `kqfilter` implementation in the tree is the kernel log device, `/dev/klog`, in `/usr/src/sys/kern/subr_log.c`. Its entire `kqueue` support fits in about forty lines and uses exactly the pattern we have been discussing. Let us read it.

The filterops table is a minimal one, with only detach and event callbacks:

```
static const struct filterops log_read_filterops = {
 .f_isfd = 1,
 .f_attach = NULL,
 .f_detach = logkqdetach,
```

```
.f_event = logkqread,
};
```

The attach hook is NULL because all the driver-side work happens in `logkqfilter` itself. There is no need for a separate `f_attach` callback; the `d_kqfilter` entry point does everything it needs to do. Drivers that need to perform per-knote setup beyond what `d_kqfilter` does can use `f_attach`, but that is uncommon.

`logkqfilter` is the `d_kqfilter` callback:

```
static int
logkqfilter(struct cdev *dev __unused, struct knote *kn)
{

 if (kn->kn_filter != EVFILT_READ)
 return (EINVAL);

 kn->kn_fop = &log_read_filterops;
 knlist_add(&logsoftc.sc_selp.si_note, kn, 1);

 return (0);
}
```

The `/dev/klog` driver supports only readable events; a request for any other filter type gets `EINVAL`. The callback sets `kn_fop` to the static filterops table and then attaches the knote to the `softc`'s `selinfo` knlist. The third argument to `knlist_add` is 1 here, which means “the caller already holds the knlist lock.” The driver takes the message-buffer lock before entering the callback for its own reasons, so passing 1 is correct.

The event function is just as short:

```
static int
logkqread(struct knote *kn, long hint __unused)
{

 mtx_assert(&msgbuf_lock, MA_OWNED);

 kn->kn_data = msgbuf_getcount(msgbufp);
 return (kn->kn_data != 0);
}
```

It asserts the message-buffer lock (which is what the knlist uses), reads the number of queued bytes, and returns non-zero if anything is available. The userland sees the byte count in `kn->kn_data` on the next reap.

The detach function is one line:

```
static void
logkqdetach(struct knote *kn)
{
 knlist_remove(&logsoftc.sc_selp.si_note, kn, 1);
}
```

It removes the knote from the knlist, again passing 1 because the framework has taken the lock before entering `f_detach`.

The last piece is the producer. When the log timeout fires and there is new data to notify waiters about, `/dev/klog` calls `KNOTE_LOCKED(&logsoftc.sc_selp.si_note, 0)` under the message buffer lock. That walks the knlist, calls each registered knote's `f_event`, and queues notifications for any kqueues that have waiters. The hint of zero is ignored by `logkqread`, which is the common case.

The whole kqueue integration is initialised once at subsystem start via `knlist_init_mtx(&logsoftc.sc_selp.si_note, &msgbuf_lock)`. `/dev/klog` is never unloaded in practice, so there is no teardown sequence to study here. That comes later, in the `evdev` example.

The takeaway is how small this code is. A complete, working, production-grade `kqfilter` integration for a real driver in FreeBSD 14.3 is under forty lines. The complexity of kqueue is in the framework, not in the driver's contribution.

### 35.7.14 Worked Example: TTY Read and Write Filters

The terminal subsystem in `/usr/src/sys/kern/tty.c` gives us the next step up: a driver that supports both readable and writable filters, and that uses `EV_EOF` to signal that the device is gone. The pattern is the one we use in any driver that wants to expose two independent sides of the same device.

The two filterops tables in `/usr/src/sys/kern/tty.c` are:

```
static const struct filterops tty_kqops_read = {
 .f_isfd = 1,
 .f_detach = tty_kqops_read_detach,
 .f_event = tty_kqops_read_event,
};

static const struct filterops tty_kqops_write = {
 .f_isfd = 1,
 .f_detach = tty_kqops_write_detach,
 .f_event = tty_kqops_write_event,
};
```

The `d_kqfilter` entry point, `ttydev_kqfilter`, switches on the requested filter and attaches to one of two knlists:

```
static int
ttydev_kqfilter(struct cdev *dev, struct knote *kn)
{
 struct tty *tp = dev->si_drv1;
 int error;

 error = ttydev_enter(tp);
 if (error != 0)
 return (error);

 switch (kn->kn_filter) {
 case EVFILT_READ:
 kn->kn_hook = tp;
 kn->kn_fop = &tty_kqops_read;
 knlist_add(&tp->t_inpoll.si_note, kn, 1);
 break;
 case EVFILT_WRITE:
 kn->kn_hook = tp;
 kn->kn_fop = &tty_kqops_write;
 knlist_add(&tp->t_outpoll.si_note, kn, 1);
 break;
 default:
 error = EINVAL;
 break;
 }

 tty_unlock(tp);
 return (error);
}
```

Three things are worth noticing here.

First, each direction has its own selinfo (`t_inpoll`, `t_outpoll`) and therefore its own knlist. A readable knote goes on one list and a writable knote goes on the other. This lets the producer notify only the side that changed: when incoming characters arrive, only readable waiters wake; when the output buffer drains, only writable waiters wake. Drivers that unify both sides onto one knlist would have to waste cycles waking everybody for every state change.

Second, the third argument to `knlist_add` is 1, because `ttydev_enter` has already taken the tty lock before the switch runs. The tty subsystem keeps that lock held from entry to exit across most entry points, so every knlist operation inside is already-locked.

Third, the read event callback demonstrates the `EV_EOF` discipline we described earlier:

```
static int
tty_kqops_read_event(struct knote *kn, long hint __unused)
{
 struct tty *tp = kn->kn_hook;

 tty_lock_assert(tp, MA_OWNED);

 if (tty_gone(tp) || (tp->t_flags & TF_ZOMBIE) != 0) {
 kn->kn_flags |= EV_EOF;
 return (1);
 } else {
 kn->kn_data = ttydisc_read_poll(tp);
 return (kn->kn_data > 0);
 }
}
```

If the tty is gone or is a zombie, `EV_EOF` is set and the filter reports ready so that userland wakes up, reads, gets nothing, and learns from the EOF flag that the device is finished. Otherwise the filter reports the number of readable bytes and whether that count is positive. The write-side callback `tty_kqops_write_event` mirrors this pattern, reporting `ttydisc_write_poll` for the output buffer's free space. The detach callbacks simply remove the knote from whichever list it was on, with `islocked = 1` again.

What the tty example teaches is that a driver with two directions needs two knlists, two filterops tables, two event functions, and a `d_kqfilter` that steers the registration to the right one. The producer side is symmetric: incoming characters trigger `KNOTE_LOCKED` on `t_inpoll.si_note`; outgoing buffer space triggers the same on `t_outpoll.si_note`. The separation is clean and predictable, and it matches the way userland programs think about terminal I/O.

### 35.7.15 Worked Example: evdev Detach Discipline

For the last worked example we turn to the input-event subsystem in `/usr/src/sys/dev/evdev/cdev.c`. Its `kqfilter` is structurally similar to `/dev/klog`, but the `evdev` driver demonstrates something the previous two examples glossed over: a complete detach sequence that tears down the knlist safely even when live userland processes may still have `kqueue` registrations outstanding.

The filterops and attach paths look familiar. The `evdev` filterops table is:

```
static const struct filterops evdev_cdev_filterops = {
 .f_isfd = 1,
 .f_detach = evdev_kqdetach,
 .f_event = evdev_kqread,
};
```

The `d_kqfilter` implementation adds an important extra check on revocation, which makes `evdev` a little richer than `/dev/klog`:

```

static int
evdev_kqfilter(struct cdev *dev, struct knote *kn)
{
 struct evdev_client *client;
 int ret;

 ret = devfs_get_cdevpriv((void **)&client);
 if (ret != 0)
 return (ret);

 switch (kn->kn_filter) {
 case EVFILT_READ:
 kn->kn_fop = &evdev_cdev_filterops;
 kn->kn_hook = client;
 EVDEV_CLIENT_LOCKQ(client);
 if (client->ec_revoked)
 ret = ENODEV;
 else
 knlist_add(&client->ec_selp.si_note, kn, 1);
 EVDEV_CLIENT_UNLOCKQ(client);
 break;
 default:
 ret = EINVAL;
 }

 return (ret);
}

```

If the client has been revoked, because the device is going away or because a controlling process has explicitly revoked access, the driver returns `ENODEV` rather than attaching a knote. Notice that the driver takes its own per-client lock around both the `ec_revoked` check and the `knlist_add`, so the two operations are atomic with respect to revocation. This is the contract we described earlier, applied cleanly: cheap lookups, a brief lock hold, no sleeping, no memory allocation in the hot path.

The event function reports readiness from the per-client event queue:

```

static int
evdev_kqread(struct knote *kn, long hint __unused)
{
 struct evdev_client *client = kn->kn_hook;

 EVDEV_CLIENT_LOCKQ_ASSERT(client);

 kn->kn_data = EVDEV_CLIENT_SIZEQ(client) *
 sizeof(struct input_event);
}

```



```

 if (client->ec_revoked) {
 kn->kn_flags |= EV_EOF;
 return (1);
 }
 return (kn->kn_data != 0);
}

```

Notice the `kn->kn_data` convention: not just “number of items” but “number of items in bytes,” because userland reads raw `struct input_event` values and expects byte counts in the way that `read()` returns them. This kind of detail matters for userland libraries that use `kn->kn_data` to size buffers.

The producer path in `evdev_notify_event` combines every asynchronous notification mechanism the subsystem supports:

```

if (client->ec_blocked) {
 client->ec_blocked = false;
 wakeup(client);
}
if (client->ec_selected) {
 client->ec_selected = false;
 selwakeup(&client->ec_selp);
}
KNOTE_LOCKED(&client->ec_selp.si_note, 0);

if (client->ec_sigio != NULL)
 pgsigio(&client->ec_sigio, SIGIO, 0);

```

This is a complete asynchronous producer: blocking `read()` waiters are signaled via `wakeup()`, `poll()` and `select()` waiters are signaled via `selwakeup()`, kqueue waiters are signaled via `KNOTE_LOCKED`, and registered SIGIO consumers are signaled via `pgsigio`. Any given consumer sees exactly one of these, but the producer does not need to know which one; it calls all of them unconditionally and lets each mechanism filter itself. Our `evdemo` driver will adopt the same layered producer as we finish the chapter.

The detach sequence is the piece that is uniquely instructive. When an `evdev` client goes away, the driver runs:

```

knlist_clear(&client->ec_selp.si_note, 0);
seldrain(&client->ec_selp);
knlist_destroy(&client->ec_selp.si_note);

```

This is exactly the three-step discipline we described. The result is that any userland process still holding a kqueue registration for this client reaps a final `EV_EOF` event and then sees the registration disappear; any `poll()` waiter still parked on the `selinfo` wakes and returns; any in-flight kqueue delivery that was about to call back into our filterops completes safely before the `knlist` memory is released.

Getting the order wrong turns this from a clean teardown into a panic. `knlist_destroy` before `knlist_clear` asserts on a non-empty list. `knlist_clear` without `seldrain` leaves `poll` waiters hanging. `seldrain` without

a preceding `knlist_clear` will work but will leave `kqueue` registrations pointing at a driver that is about to disappear, and the first event delivery attempt will crash. Follow the sequence.

The `evdev` example brings together everything we have covered in this section: a revocation-aware attach, a byte-count-correct event report, a combined producer path, and a teardown that respects the lifetime rules. A driver that imitates this pattern will behave well in production.

### 35.7.16 The hint Parameter: What It Is and Why It Exists

Every `f_event` callback receives a long `hint` argument that we have been quietly setting to zero. It is worth understanding what that parameter is for, because it is not zero everywhere in the kernel.

The hint is a cookie passed through from the producer to the filter. When a producer calls `KNOTE_LOCKED(list, hint)`, the framework passes that same `hint` value to every filter's `f_event`. It is entirely up to the producer and the filter to agree on what the value means. The framework does not interpret it.

For simple drivers that have one meaning of “ready,” zero is the natural choice and the filter ignores the argument. For drivers with more than one producer path, the hint can distinguish them. The `vnode` filter uses non-zero hints to encode `NOTE_DELETE`, `NOTE_RENAME`, and related `vnode`-level events, and the `f_event` function tests the hint bits to decide which `kn->kn_fflags` bits to set in the delivered event. That is beyond what an ordinary character driver needs, but it explains the signature's generality.

The producer side is where the hint value originates. A driver can call `KNOTE_LOCKED(&sc->sc_rsel.si_note, MY_HINT_NEW_DATA)` and the filter can switch on the value to take different paths. In practice, ordinary drivers pass zero and keep the filter simple.

### 35.7.17 Delivering Events: `KNOTE_LOCKED` vs `KNOTE_UNLOCKED`, in Depth

The two delivery macros in `/usr/src/sys/sys/event.h` are:

```
#define KNOTE_LOCKED(list, hint) knote(list, hint, KNF_LISTLOCKED)
#define KNOTE_UNLOCKED(list, hint) knote(list, hint, 0)
```

Both call the same underlying `knote()` function in `/usr/src/sys/kern/kern_event.c`, which walks the `knlist` and invokes `f_event` on each `knote`. The difference is the third argument: `KNF_LISTLOCKED` says “the caller already holds the `knlist` lock,” while zero says “take it for me.”

Choosing between them is a matter of matching the producer's locking path. If the producer is called with the driver's mutex already taken (because it is invoked from a locked ISR handler, or from inside a producer function that needed the lock for its own work), `KNOTE_LOCKED` is correct. If the producer is called unlocked (because it is running in thread context and the lock would be taken specifically for the notification), `KNOTE_UNLOCKED` is correct. The mistake to avoid is calling `KNOTE_LOCKED` without actually holding the lock, which races horribly under load, or calling `KNOTE_UNLOCKED` while holding the lock, which recurses and panics.

An ISR-context example helps: if a device interrupt handler calls a bottom-half function that acquires the softc mutex, does some work, and needs to notify `kqueue` waiters, the cleanest pattern is to do the work and the `KNOTE_LOCKED` call inside the held mutex and drop the lock afterwards. The mutex is the `knlist` lock, so `KNOTE_LOCKED` is what to use. If the notification instead comes from a thread that has not yet taken the lock, the

thread takes the lock, does the work, calls `KNOTE_LOCKED`, and then drops the lock; or it uses `KNOTE_UNLOCKED` and lets the framework briefly take the lock on its way through the list.

A second subtlety is the behaviour of `knote` when the list is empty. Walking an empty list is cheap but not free; it still takes the lock. Drivers that deliver very high-rate notifications can test `KNLIST_EMPTY(list)` first and skip the `KNOTE_LOCKED` call if there are no waiters. The macro `KNLIST_EMPTY`, defined in `/usr/src/sys/sys/event.h` as `SLIST_EMPTY(&(list)->kl_list)`, is safe to read without the lock for the purpose of a hint, because the worst case of a stale read is a missed wakeup on a `knote` that was added a microsecond ago, and that `knote` will notice the next delivery. In practice this optimisation is rarely worth the complexity, but it is worth knowing about.

### 35.7.18 Common Pitfalls in Driver `kqfilter` Implementations

Over the course of reading `kqueue`-aware drivers in the tree, a handful of recurring bug patterns show up. Knowing the pitfalls in advance helps avoid them.

Forgetting to destroy the `knlist`. A driver that calls `knlist_init_mtx` in `attach` but does not call `knlist_destroy` in `detach` leaks the `knlist`'s internal state and, worse, may leave live `knotes` dangling. The fix is to include the clear-drain-destroy sequence in every `detach` path.

Calling `knlist_destroy` before `knlist_clear`. `knlist_destroy` asserts that the list is empty. If there are any `knotes` still attached, the assertion fails and the kernel panics. Always clear first.

Using `KNOTE_LOCKED` without the lock. This is subtle because it works most of the time. Under load, two producers can race in the `knote` walk, and the framework's assumption that the list is stable during traversal breaks. The symptom is usually a `knote` pointer corruption or a use-after-free in `f_event`.

Sleeping in `f_event`. The framework is holding the `knlist` lock, which is our `softc` mutex, when it calls us. Sleeping under a mutex is a kernel bug. If `f_event` needs state that is not already accessible under the `softc` mutex, the design is wrong; move the state into the `softc` or pre-compute it before the notification.

Returning stale `kn_data`. The `kn->kn_data` field should reflect the state at the moment the filter was evaluated. A driver that computes `kn_data` once in `d_kqfilter` and forgets to update it in `f_event` will deliver stale byte counts to the userland. Always recompute it in `f_event`.

Keeping `kn_hook` pointing at freed memory. If `kn_hook` is set to a `softc`, and the `softc` is freed before the `knote` is detached, the next `f_event` call will dereference freed memory. This is what `knlist_clear` and `sel drain` are supposed to prevent, but only if they are called in the correct order and before the `softc` is freed. The detach order in the driver's `detach` entry point matters.

Setting `EV_EOF` only once. `EV_EOF` is sticky in the sense that once it is set, the userland will see it, but `f_event` is called multiple times over a `knote`'s life. If the condition that caused `EV_EOF` can become false again (for example, a named pipe that gains a new writer), the filter must clear `EV_EOF` explicitly. The pipe filter in `/usr/src/sys/kern/sys_pipe.c` demonstrates this: `filt_piperead` both sets and clears `EV_EOF` depending on the pipe's state.

Confusing `f_isfd` with `f_attach`. `f_isfd = 1` means the filter is tied to a file descriptor; almost all driver filters want this. `f_attach = NULL` means "the registration path does not need a per-`knote` attach callback beyond what `d_kqfilter` already did." They are independent. A driver can set `f_isfd = 1` and `f_attach = NULL` at the same time, which is the common case.

Returning errors from `f_event`. `f_event` returns an int, but it is a boolean: zero means “not ready,” non-zero means “ready.” It is not an errno. Returning `EINVAL` from `f_event` means “ready,” which is almost certainly not what the driver intended.

### 35.7.19 A Mental Model for the kqueue Framework

It is worth pausing to assemble a mental model of the kqueue framework that fits what we have learned. Different readers will find different models helpful; one that works well for driver authors is this.

Imagine each driver object (a `cdev`, a per-client state record, a pipe, a `tty`) as a small office. The office has in-boxes and out-boxes, which are the `knlists`. When a visitor (a userland program) wants to be told when there is new mail in the in-box, they register a sticky note with the office: their kqueue file descriptor, plus the filter type they care about. The office clerks (our `d_kqfilter` callback) take the sticky note, check which in-box it belongs on (`EVFILT_READ` in-box or `EVFILT_WRITE` out-box), and pin it there. The sticky note records who to notify (the kqueue) and how (the `struct filterops` callbacks).

When mail actually arrives (the producer path inserts a record and wants to notify), the office clerks walk the in-box sticky notes and, for each one, check whether the condition is currently satisfied (the `f_event` callback). If it is, the clerk picks up the phone and rings the visitor’s kqueue, delivering a notification. The visitor reads the notification on their next `kevent(2)` reap.

When the visitor changes their mind and no longer wants mail notifications (removes the registration), the office clerks pull the sticky note down (the `f_detach` callback). When the office closes permanently (the driver detaches), the clerks pull down every sticky note at once (`knlist_clear`), wake any visitors who are physically sitting in the waiting room (`selldrain`), and then dismantle the sticky-note corkboard (`knlist_destroy`).

The lock on the corkboard is the driver’s softc mutex. The clerks hold it while walking the notes, pinning a note, or pulling a note down. This is why `f_event` must not sleep: the clerks cannot let go of the lock while they are working through the list, because other clerks might arrive with updates. It is also why `KNOTE_LOCKED` is the right call when the producer already holds the lock: the clerk saying “I am already holding it” lets the framework skip an unnecessary re-acquisition.

The model is simplified, and it leaves out complications like `EV_CLEAR` edge semantics and `f_touch` registration updates, but it captures the essential architecture. The driver owns the corkboard; the framework owns the sticky notes. The driver reports readiness; the framework handles delivery. The driver tears down the corkboard at detach; the framework’s sticky-note structures are freed as part of that teardown.

Keep this picture in mind as you read the code of other kqueue-using subsystems, and the unfamiliar names will map back onto familiar roles. `kqueue_register` is the visitor walking in to submit a sticky note. `knote` is the clerk walking the corkboard. `f_event` is each note’s individual readiness-check. `selwakeup` is the general fire alarm that also reaches the corkboard. The names are different; the shapes are the same.

### 35.7.20 Reading `kern_event.c`: A Guide for the Curious

For readers who want to go further than the callbacks, the kqueue framework itself is worth a tour. `/usr/src/sys/kern/kern_event.c` is about three thousand lines, which looks intimidating, but the structure of the file is predictable once we know what to look for.

Near the top of the file, the static filterops tables for the built-in filters are declared. `file_filtops` handles the generic read and write filters for file descriptors that do not provide their own `kqfilter`; `timer_filtops` handles `EVFILT_TIMER`; `user_filtops` handles `EVFILT_USER`; and several more follow. These are the filterops the framework installs at boot, and reading them gives a good sense of how a filterops table is meant to look in production code.

After the static declarations come the system call entry points: `kqueue`, `kevent`, and the legacy variants. These do argument validation and dispatch to the core machinery. A reader tracing a userland call through the kernel starts here.

The core machinery is a set of functions with names beginning `kqueue_`. `kqueue_register` handles `EV_ADD`, `EV_DELETE`, `EV_ENABLE`, `EV_DISABLE`, and `EV_RECEIPT`; it is where the framework calls our `d_kqfilter` through `fo_kqfilter`. `kqueue_scan` reaps ready events back to userland. `kqueue_acquire` and `kqueue_release` reference-count the `kqueue` for safe concurrent access. `kqueue_close` tears the `kqueue` down when the last file descriptor referencing it is closed. Tracing from the top of `kqueue_register` through `kqueue_expand`, `knote_attach`, and the `fo_kqfilter` call reveals the full registration path.

The `knote` function itself, about two-thirds of the way through the file, is the one we reach through `KNOTE_LOCKED` and `KNOTE_UNLOCKED`. It walks the `knlist`, invokes `f_event` on each `knote`, and queues notifications for any that report ready. Reading it shows exactly why the lock assertions on our `f_event` are necessary and how the framework interleaves list traversal with `kqueue` notification. The walk uses `SLIST_FOREACH_SAFE` with a temporary pointer, so an `f_detach` during the walk does not corrupt the iteration. That subtle detail is what makes concurrent detach and delivery safe.

Further down comes the `knlist` machinery: `knlist_init`, `knlist_init_mtx`, `knlist_add`, `knlist_remove`, `knlist_cleardel`, `knlist_destroy`, and the various helpers. These are the functions we have been calling. Reading them confirms the lock semantics we have been relying on and shows how the `islocked` arguments are consumed inside the helpers.

Near the end of the file come the filter implementations for the built-in filters, with names like `filt_timerattach`, `filt_user`, and `filt_fileattach`. These are worth reading because they are the closest thing to a reference implementation for how a filter should be structured. The pipe filter in `/usr/src/sys/kern/sys_pipe.c` is another good reference; socket `kqueue` support in `/usr/src/sys/kern/uipc_socket.c` is a third.

A reader who works through `kqueue_register`, `knote`, and `knlist_remove` in that order will understand most of the framework by the end of an afternoon. The remaining machinery (auto-destroy, timer implementation, proc-and-signal filters, vnode-note masks) is specialised enough that a driver author can skip it unless a specific need arises. The rest of this chapter does not require any of it.

### 35.7.21 Driver Patterns We Have Not Yet Used

Two patterns appear in the tree that we have not used in `evdemo` because they are not needed, but are worth recognising so that readers who see them elsewhere know what they are.

The first is the use of `f_attach` for per-`knote` setup beyond what `d_kqfilter` does. The `EVFILT_TIMER` filter uses `f_attach` to start a one-shot or repeating timer when the `knote` is first registered, and `f_detach` to stop it. The `EVFILT_USER` filter in `/usr/src/sys/kern/kern_event.c` uses `filt_userattach` as a no-op because the `knote` is not attached to anything in the kernel; the user-triggered `NOTE_TRIGGER` mechanism handles delivery entirely through `f_touch`. A driver that needs its own per-`knote` state could allocate it in `f_attach` and free

it in `f_detach`, using `kn_hook` or `kn_hookid` to remember the pointer. Almost no driver actually needs this, because the per-registration state usually fits naturally in the `softc`.

The second is `f_touch`, which intercepts `EV_ADD`, `EV_DELETE`, and `EV_ENABLE/EV_DISABLE` operations. The `filt_usertouch` function in `/usr/src/sys/kern/kern_event.c` is a good reference for how `f_touch` is structured: it inspects the `type` argument (one of `EVENT_REGISTER`, `EVENT_PROCESS`, or `EVENT_CLEAR`) to decide what the userland is asking for and updates the knote's fields accordingly. Most driver filters leave `f_touch` as `NULL` and accept the framework's default behaviour, which is to store `sfflags`, `sdata`, and the event flags in the knote directly during `EV_ADD`. The default is correct for filters that do not need extra behaviour on registration updates.

A third pattern the tree uses but our driver does not is the “kill” variant of knlist teardown. `knlist_cleared` in `/usr/src/sys/kern/kern_event.c` accepts a `killkn` flag that, when set, forces every knote off the list whether or not it is still in use. `knlist_clear` is the common wrapper with this flag set. A driver that wants to preserve knotes across an event (for example, to re-attach them to a new object) could call `knlist_cleared` with `killkn` false and the knotes would be unhooked but left alive. This is almost never what a driver wants. The common case is `knlist_clear`, which kills and frees.

### 35.7.22 A Note on `EV_CLEAR`, `EV_ONESHOT`, and Edge-Triggered Behaviour

The `kqueue` framework supports several delivery modes through flags on the `struct kevent`:

`EV_CLEAR` makes the filter edge-triggered: once a knote fires, it will not fire again until the underlying condition transitions from false to true again. This is the common choice for readable and writable filters on high-throughput descriptors, because it avoids spamming the userland with repeated notifications for the same data.

`EV_ONESHOT` makes the filter fire exactly once and then auto-delete itself. It is useful for one-time events.

`EV_DISPATCH` makes the filter fire at most once per `kevent()` reap, auto-disabling itself after each fire. The userland re-enables it by re-registering with `EV_ENABLE`. This is the mode preferred by multi-threaded userland programs that want to ensure only one thread reacts to each event.

The driver's filter functions do not need to know about these flags; the framework handles them. The driver just reports whether the underlying condition is satisfied, and the framework decides what to do with the resulting knote.

### 35.7.23 Wrapping Up Section 4

We now have `kqueue` support in our driver. The total code we added is not enormous: a `d_kqfilter` callback, a `struct filterops`, two short filter functions, and a `KNOTE_LOCKED()` call in the producer. The complexity is more about understanding the framework than about writing a lot of code.

But we have only covered the two most common filters, `EVFILT_READ` and `EVFILT_WRITE`. The chapter's scope deliberately excludes deeper `kqueue` topics such as user-defined filters (`EVFILT_USER`), custom `f_touch` implementations, and interactions with the AIO subsystem. Those are specialized enough that they rarely appear in ordinary drivers, and they would crowd out material that most readers need. If you do need them, the material in this section prepares you to read the corresponding parts of `/usr/src/sys/kern/kern_event.c` and understand what you find.

Looking back at what this section has covered, the reader should now be comfortable with several layers that tend to blur together in casual discussions of `kqueue`. The outermost layer is the userland API: `kqueue(2)`, `kevent(2)`, and the `struct kevent` values that programs submit and reap. The middle layer is the framework: `kqueue_register`, `knote`, `kqueue_scan`, and the machinery that matches registrations to deliveries. The inner layer is the driver contract: `d_kqfilter`, `struct filterops`, `struct knote`, `struct knlist`, and the small set of helper functions like `knlist_init_mtx`, `knlist_add`, `knlist_remove`, `knlist_clear`, and `knlist_destroy`. The three layers communicate through well-defined boundaries, and understanding which is which is the difference between guessing at `kqueue` and understanding it.

We have also walked through three real driver implementations: `/dev/klog`, the `tty` subsystem, and the `evdev` input stack. Each illustrates a different aspect of the `kqfilter` contract. The `klog` driver shows the minimum that a `kqueue`-aware driver needs. The `tty` subsystem shows how to handle two directions with two separate `knlists`. The `evdev` driver shows the revocation-aware attach, the byte-count-correct event report, the combined producer path that fans out to multiple asynchronous mechanisms, and the strict clear-drain-destroy detach sequence. A driver that combines the appropriate pieces of these three patterns will behave well in production, and a reader who has followed the discussion should be able to recognise the patterns when they appear in other subsystems in the tree.

In the next section we turn to the third asynchronous mechanism, `SIGIO`. Unlike `poll()` and `kqueue()`, which are pull-style notifications (the userland asks, the kernel answers), `SIGIO` is push-style: the kernel sends a signal to the registered process whenever the device state changes. It is older, simpler, and has some subtle problems in multi-threaded programs, but it is still useful in specific situations and is part of the standard driver toolkit.

## 35.8 5. Asynchronous Signals With `SIGIO` and `FIOASYNC`

The third classic asynchronous mechanism is signal-driven I/O, also called `SIGIO` notification after the signal it typically uses. The user enables it through the `FIOASYNC` `ioctl` on an open file descriptor, sets an owner with `FIOSETOWN`, and installs a handler for `SIGIO`. The driver, whenever it has a relevant state change, sends `SIGIO` to the registered owner. That signal can interrupt almost any system call in the owner, which then typically services the device and returns to its normal work.

Signal-driven I/O is older than `kqueue`, less scalable than `poll`, and has some subtle issues in multi-threaded programs. It is still the right mechanism in a small but real set of cases: single-threaded programs that want the simplest possible asynchronous notification, shell scripts using `trap`, and legacy code that has used `SIGIO` for decades and will not change. FreeBSD continues to support it in full, and most ordinary character drivers are expected to honour the mechanism.

### 35.8.1 How Signal-Driven I/O Works From Userland

A user program using `SIGIO` does three things. It installs a signal handler for `SIGIO`. It tells the kernel which process should own the signal for this descriptor. It enables asynchronous notification.

The code looks roughly like this:

```
#include <signal.h>
#include <sys/filio.h>
```

```
#include <fcntl.h>
#include <unistd.h>

static volatile sig_atomic_t got_sigio;

static void
on_sigio(int sig)
{
 got_sigio = 1;
}

int
main(void)
{
 int fd = open("/dev/evdemo", O_RDONLY | O_NONBLOCK);

 struct sigaction sa;
 sa.sa_handler = on_sigio;
 sigemptyset(&sa.sa_mask);
 sa.sa_flags = 0;
 sigaction(SIGIO, &sa, NULL);

 int pid = getpid();
 ioctl(fd, FIOSETOWN, &pid);

 int one = 1;
 ioctl(fd, FIOASYNC, &one);

 for (;;) {
 pause();
 if (got_sigio) {
 got_sigio = 0;
 char buf[256];
 ssize_t n;
 while ((n = read(fd, buf, sizeof(buf))) > 0) {
 /* process data */
 }
 }
 }
}
```

The sequence of `ioctl`s is important. The program first installs the signal handler so that `SIGIO` will not be ignored when it arrives. It then calls `FIOSETOWN` with its own PID (positive values mean process, negative values mean process group) so that the driver knows where to deliver the signal. Finally it calls `FIOASYNC` with



a non-zero value to enable notification.

Once asynchronous notification is enabled, every state change in the driver that would have satisfied a `POLLIN` causes a `SIGIO` signal to the owner. The program's handler runs asynchronously, sets a flag, and returns; the main loop then services the device. Drain the device to empty with non-blocking reads, because between the time the signal was sent and the time the handler ran, multiple events may have accumulated.

### 35.8.2 The `FIOASYNC`, `FIOSETOWN`, and `FIOGETOWN` `ioctl`s

Open `/usr/src/sys/sys/filio.h` to see the `ioctl` definitions:

```
#define FIOASYNC _IOW('f', 125, int) /* set/clear async i/o */
#define FIOSETOWN _IOW('f', 124, int) /* set owner */
#define FIOGETOWN _IOR('f', 123, int) /* get owner */
```

These are standard `ioctl`s that most of the file descriptor-handling layer already understands. For an ordinary file descriptor (a socket, a pipe, a `pty`), the kernel handles them without involving the driver. For a `cdev`, however, the driver is responsible for implementing them, because the driver owns the state that the `ioctl`s manipulate.

The conventional approach in a FreeBSD character driver is:

`FIOASYNC` takes an `int *` argument. Non-zero enables asynchronous notification. Zero disables it. The driver stores the flag in the `softc` and uses it to decide whether to generate signals.

`FIOSETOWN` takes an `int *` argument. A positive value is a PID, a negative value is a process group ID, and zero clears the owner. The driver uses `fsetown()` to record the owner.

`FIOGETOWN` takes an `int *` argument to be filled in. The driver uses `fgetown()` to retrieve the current owner.

### 35.8.3 `fsetown`, `fgetown`, and `funsetown`

The owner-tracking mechanism uses a `struct sigio` in the kernel. We do not have to allocate or manage that structure directly; the `fsetown()` and `funsetown()` helpers do it for us. The public API, in `/usr/src/c/sys/sys/sigio.h` and `/usr/src/sys/kern/kern_descrip.c`, consists of four functions:

```
int fsetown(pid_t pgid, struct sigio **sigiop);
void funsetown(struct sigio **sigiop);
pid_t fgetown(struct sigio **sigiop);
void pgsigio(struct sigio **sigiop, int sig, int checkctty);
```

The driver stores a single `struct sigio *` in the `softc`. All four helpers take a pointer to this pointer, because they may replace the whole structure as part of their work. The helpers take care of reference counting, locking, and safe removal during process exit through `eventhandler(9)`.

`fsetown()` installs a new owner. It expects to be called with the interrupted caller's credentials available (which is always the case inside an `ioctl` handler). If the target PID is zero, it clears the owner. If the target is a positive number, it looks up the process. If it is a negative number, it looks up the process group. It returns zero on success or an `errno` on failure.

`funsetown()` clears the owner and frees the associated structure. Drivers call it during close and during detach to make sure no stale references are left behind.

`fgetown()` returns the current owner as a PID (positive) or a process group ID (negative), or zero if no owner is set.

`pgsigio()` delivers a signal to the owner. The third argument, `checkctty`, should be zero for a driver that is not a controlling terminal. This is what the driver calls from the producer path whenever asynchronous notification is enabled.

### 35.8.4 Implementing SIGIO in evdemo

Putting the pieces together, here is what we add to our driver to support SIGIO:

In the `softc`:

```
struct evdemo_softc {
 /* ... existing fields ... */
 struct sigio *sc_sigio;
 bool sc_async;
};
```

In `d_ioctl`:

```
static int
evdemo_ioctl(struct cdev *dev, u_long cmd, caddr_t data, int fflag,
 struct thread *td)
{
 struct evdemo_softc *sc = dev->si_drv1;
 int error = 0;

 switch (cmd) {
 case FIOASYNC:
 mtx_lock(&sc->sc_mtx);
 sc->sc_async = (*(int *)data != 0);
 mtx_unlock(&sc->sc_mtx);
 break;

 case FIOSETOWN:
 error = fsetown(*(int *)data, &sc->sc_sigio);
 break;

 case FIOGETOWN:
 *(int *)data = fgetown(&sc->sc_sigio);
 break;
 }
```

```

 default:
 error = ENOTTY;
 break;
 }
 return (error);
}

```

In the producer path:

```

static void
evdemo_post_event(struct evdemo_softc *sc, struct evdemo_event *ev)
{
 bool async;

 mtx_lock(&sc->sc_mtx);
 evdemo_enqueue(sc, ev);
 async = sc->sc_async;
 KNOTE_LOCKED(&sc->sc_rsel.si_note, 0);
 mtx_unlock(&sc->sc_mtx);

 selwakeup(&sc->sc_rsel);
 if (async)
 pgsigio(&sc->sc_sigio, SIGIO, 0);
}

```

In `d_close` or during detach:

```

static int
evdemo_close(struct cdev *dev, int flags, int fmt, struct thread *td)
{
 struct evdemo_softc *sc = dev->si_drv1;

 funsetown(&sc->sc_sigio);
 /* ... other close handling ... */
 return (0);
}

```

Let us walk through the pieces.

The `softc` gains two new fields: `sc_sigio`, the pointer we pass to `fsetown()` and friends, and `sc_async`, a flag telling the producer whether signals are enabled. The flag is redundant with “`sc_sigio` is non-NULL” in a sense, but keeping it explicit makes the producer code clearer and faster.

The `d_ioctl` handler implements the three `ioctl`s. We take the `softc` mutex for `FIOASYNC` because we update `sc_async`. We do not take the mutex for `FIOSETOWN` and `FIOGETOWN` because the `fsetown()` and `fgetown()` helpers have their own internal locking and should not be called with a driver lock held.

In the producer, we copy `sc_async` into a local variable under the lock so that the value we use outside the lock is consistent. If we had simply read `sc->sc_async` after the lock was dropped, another thread could have changed it in between, which is a race. Taking a snapshot under the lock avoids the race.

We call `pgsigio()` outside the softc lock because `pgsigio()` takes its own locks and could create ordering problems if nested. The pattern is the same as `selwakeup()`: update under the lock, drop, then deliver the notification.

In `d_close`, we call `funsetown()` to clear the owner. This also handles the case where the process that set the owner has since exited, so the driver does not leak `struct sigio` allocations. If the process has already exited, `funsetown()` is essentially a no-op; if it has not, the call cleans up the registration.

### 35.8.5 Caveats: Signal Semantics in Multi-Threaded Programs

Signal-driven I/O has well-known weaknesses in multi-threaded programs. The primary issue is that signals in POSIX are sent to a process, not to a specific thread. When the kernel delivers `SIGIO`, any of the process's threads whose mask allows the signal may be the one to receive it. For a program that wants a specific thread to service the notification, this is inconvenient.

There are workarounds. `pthread_sigmask()` can be used to block `SIGIO` in all threads except the one that should service it. If you want to convert signals into readable events on a file descriptor, FreeBSD provides `EVFILT_SIGNAL` through `kqueue(2)`, which lets a `kqueue` report that a given signal has been delivered to the process. FreeBSD does not provide the Linux-specific `signalfd(2)` system call. The simplest workaround, and often the right one, is to use `kqueue` directly for the underlying driver events: threads can each own a separate `kqueue` and each can wait for exactly the events they care about, without fighting signal delivery rules at all.

A second weakness is that signals interrupt system calls. Under the default SA flags, an interrupted system call returns with `EINTR`, and the program must check for this and retry. This is unusual enough that it often produces bugs in programs that were written without `SIGIO` in mind. The workaround is to set `SA_RESTART` in `sa_flags`, which makes the kernel restart interrupted system calls automatically.

A third weakness is that signal delivery is asynchronous with respect to the program's execution. A signal that arrives while the program is in the middle of a data structure update can lead to inconsistent state if the signal handler touches the same structure. The fix is to keep signal handlers very simple (set a flag, return) and do the actual work in the main loop.

For modern programs, `kqueue` avoids all three of these issues. For legacy programs and simple single-threaded applications, `SIGIO` is fine, and implementing it in a driver is a small amount of code.

### 35.8.6 What Real Drivers Look Like

The `if_tuntap.c` driver provides a representative example of `SIGIO` handling. In the softc:

```
struct tuntap_softc {
 /* ... */
 struct sigio *tun_sigio;
```

```
/* ... */
};
```

In the `ioctl` handler, the driver calls `fsetown()` and `fgetown()` for `FIOSETOWN` and `FIOGETOWN` respectively, and stores the `FIOASYNC` flag. In the producer path (when a packet is ready to be read from the interface), the driver calls `pgsigio()`:

```
if (tp->tun_flags & TUN_ASYNC && tp->tun_sigio)
 pgsigio(&tp->tun_sigio, SIGIO, 0);
```

In the close path, it calls `funsetown()`.

The `evdev/cdev.c` driver has a similar structure. These are the patterns you will reuse in your own drivers.

### 35.8.7 Testing SIGIO From Shell

A nice property of `SIGIO` is that you can demonstrate it from the shell without writing any code. The Bourne-style shells (`sh`, `bash`) have a built-in `trap` command that runs an action when a signal arrives. Combined with the `FIOASYNC` `ioctl`, we can set up a test in a few lines:

```
trap 'echo signal received' SIGIO
exec 3< /dev/evdemo
(mechanism to enable FIOASYNC on fd 3 goes here)
Trigger events in another terminal and watch for "signal received"
```

The catch is that there is no direct shell-level way to issue an `ioctl`. You need either a small C helper, or a tool like the `ioctl(1)` command that some BSDs ship, or `truss` in a traced child. For the lab in this chapter we provide a small `evdemo_sigio` program that calls the right `ioctls` and then simply pauses, leaving the shell's `trap` handler to show the signal deliveries.

### 35.8.8 A Note on POSIX AIO

FreeBSD also supports the POSIX `aio_read(2)` and `aio_write(2)` APIs. These are beyond the normal scope of a character driver, and ordinary `cdev` drivers almost never need to implement anything special to participate in AIO. The remaining subsections of this section explain why that is, how AIO actually dispatches requests inside the kernel, and when (if ever) a driver should think about AIO at all. The intent is to head off a common source of confusion: when readers see “asynchronous I/O on files” in FreeBSD documentation, they are reading about POSIX AIO, and it is easy to assume that a driver needs its own AIO machinery to be a first-class citizen. It does not.

### 35.8.9 How AIO Dispatches: `fo_aio_queue` and `aio_queue_file`

When a userland program calls `aio_read(2)` or `aio_write(2)`, the request enters the kernel, is validated, and becomes a `struct kaiocb` (kernel AIO control block). The code path from there is worth tracing because it explains why a character driver almost never needs to do anything about POSIX AIO at all.

In `/usr/src/sys/kern/vfs_aio.c`, the dispatch is done at the file operations layer. The relevant decision, inside `aio_aqueue`, looks like this:

```
if (fp->f_ops->fo_aio_queue == NULL)
 error = aio_queue_file(fp, job);
else
 error = fo_aio_queue(fp, job);
```

The decision is made at the file-ops layer, not the cdev layer. If the file's struct `fileops` has its own `fo_aio_queue` function pointer, AIO delegates to it. Vnode file operations set `fo_aio_queue = vn_aio_queue_vnops`, which routes regular-file requests through a path that knows how to talk to the underlying file system. A cdev file's `fileops`, by contrast, leaves `fo_aio_queue` as `NULL`, so AIO falls into the generic `aio_queue_file` path.

`aio_queue_file` in `/usr/src/sys/kern/vfs_aio.c` tries two things. First, it attempts `aio_qbio` (the bio-based path, described in the next subsection) if the underlying object looks like a block device. Second, if the bio path is not applicable, it schedules `aio_process_rw` on one of the AIO worker threads. `aio_process_rw` is a daemon-based path that simply calls `fo_read` or `fo_write` synchronously from the AIO worker thread. In other words, for a generic cdev, “asynchronous I/O” is implemented by asking a kernel thread to perform a synchronous `read()` or `write()` on the application's behalf.

This is why ordinary character drivers do not need to implement AIO hooks of their own. The AIO subsystem does not call into the driver through a new entry point; it calls `fo_read` and `fo_write`, which in turn call the driver's existing `d_read` and `d_write`. If our driver already supports blocking and non-blocking reads correctly, it already supports AIO, just through a worker thread. No additional code is required on the driver side.

### 35.8.10 The Block Device Path: `aio_qbio` and Bio Callbacks

For block devices (disk, cd, and friends), the worker-thread path is inefficient because the block I/O layer already has its own asynchronous completion mechanism. FreeBSD takes advantage of this through `aio_qbio` in `/usr/src/sys/kern/vfs_aio.c`, which submits the request as a struct `bio` directly to the underlying device's strategy routine and arranges for `aio_biowakeup` to be called on completion. The bio carries a back-pointer to the struct `kaiocb` so that completion can find its way back to the AIO framework.

`aio_biowakeup` in `/usr/src/sys/kern/vfs_aio.c` retrieves the struct `kaiocb` that the bio is carrying, computes the residual byte count, and calls `aio_complete` with the result. `aio_complete` sets the status and error fields on the `kaiocb`, marks it as finished, and then calls `aio_bio_done_notify`, which fans out to any kqueue registration on the `kaiocb`, any blocking waiter in `aio_suspend`, and any signal registration that the userland requested through the `aiocb.aio_sigevent` field.

`aio_biocleanup` is the companion helper that frees the bio's buffer mappings and returns the bio itself to its pool. Every bio used on the AIO path passes through it, either on the wakeup path or in the cleanup loop when submission fails partway through a multi-bio request.

This path is entirely internal to the block I/O layer. A character driver that is not a block device will never see it. A block device driver sees exactly the same bios it would see from any other source: the driver cannot tell that this particular bio came from `aio_read` rather than from `read` on a buffer cache page. That is the point. AIO

fits into the block layer by reusing the existing strategy contract, not by adding a parallel path. A block driver that gets its strategy routine right gets AIO for free.

### 35.8.11 The Worker Thread Path: `aio_process_rw`

When `aio_qbio` is not applicable, as is the case for almost every character driver, `aio_queue_file` falls through to `aio_schedule(job, aio_process_rw)`. That puts the job on the AIO work queue. One of the pre-spawned AIO daemon threads (the pool size is tunable through the `vfs.aio.max_aio_procs` sysctl) picks it up, runs `aio_process_rw`, and does the actual I/O.

`aio_process_rw` in `/usr/src/sys/kern/vfs_aio.c` is the heart of the worker path. It prepares a `struct uio` from the `kaiocb`'s fields, calls `fo_read` or `fo_write` on the file, and passes the return value on to `aio_complete`. From the driver's point of view, the I/O comes in through a perfectly ordinary read or write call, with one subtle difference: the calling thread is an AIO daemon, not the process that submitted the request. The user credentials are correct because the AIO framework preserved them, but the process context is the AIO daemon's. Drivers that rely on `curthread` or `curproc` for their own bookkeeping might see surprising values; drivers that do not, which is almost all of them, behave identically whether the caller is the user's own thread or an AIO daemon.

The worker-thread path is not “async” in the hardware sense. It is “async” in the API sense: the userland did not block. The substitution happens at the thread boundary, not at the I/O boundary, so a slow device still ties up an AIO worker while it serves the request. For most `cdev` drivers, this is exactly the right trade-off. The userland gets the programming model it wants; the kernel uses a worker thread to get the work done; the driver does nothing special. If the driver already honours `O_NONBLOCK` correctly, the worker thread can even submit non-blocking requests to it and return `EAGAIN` back to the userland through the normal path.

### 35.8.12 Completion: `aio_complete`, `aio_cancel`, and `aio_return`

Once `aio_complete` has been called, the `kaiocb` enters its finished state. The userland program will eventually call `aio_return(2)` to retrieve the byte count, or `aio_error(2)` to check the error code, or wait on a `kqueue` or signal to be told that the job has finished. Those calls look up the `kaiocb` by its userland pointer and return the fields that `aio_complete` set.

From the driver's point of view, there is nothing to do on the return path. The driver does not own the `kaiocb`, does not free it, and does not signal completion directly. Completion is announced by `aio_complete`; `aio_return` is a userland concern handled entirely by the kernel's AIO layer. The driver's job ended when it satisfied the `fo_read` or `fo_write` call, or when the strategy routine called `biodone` on the bio.

For cancellation, `aio_cancel` in `/usr/src/sys/kern/vfs_aio.c` ultimately calls `aio_complete(job, -1, ECANCELED)`. That is it. The job is marked complete with an error, and the usual wakeup paths fire. The driver does not need to know about cancellation at all unless it implements its own long-running request-holding queue, which is exceptional.

One distinction is worth making explicit. `aio_cancel` is the kernel-side cancellation function used internally by AIO; it is not the userland syscall. The userland-facing `aio_cancel(2)` takes a file descriptor and a pointer to an `aiocb` and asks the kernel to cancel one or all pending requests. Internally that winds up calling the kernel `aio_cancel` on each matching `kaiocb`. The naming is a little unfortunate; reading the source makes it obvious which one is which.

### 35.8.13 EVFILT\_AIO: How AIO Uses kqueue

It is worth knowing, though not acting on, that EVFILT\_AIO exists. Declared in `/usr/src/sys/sys/event.h` and implemented in `/usr/src/sys/kern/vfs_aio.c` as the `aio_filtops` table, it lets userland programs wait for AIO completions on a kqueue. The filterops are registered once at AIO module load by `kqueue_add_filterops(EVFILT_AIO, &aio_filtops)`. The per-kaiocb callbacks are:

```
static const struct filterops aio_filtops = {
 .f_isfd = 0,
 .f_attach = filt_aioattach,
 .f_detach = filt_aiodetach,
 .f_event = filt_aio,
};
```

`f_isfd` is zero here because an AIO registration is keyed on the kaiocb, not on a file descriptor. `filt_aioattach` links the knote into the kaiocb's own knlist. `filt_aio` reports the completion status by checking whether the kaiocb has been marked finished. The `kl_autodestroy` field of the kaiocb's knlist is set, so the knlist can be torn down automatically when the kaiocb is freed. This is one of the few places in the tree where `kl_autodestroy` is actually exercised, which makes `vfs_aio.c` useful reading if you ever need to understand how that flag is used.

None of this is driver business. The AIO module registers EVFILT\_AIO once at boot, and from then on the userland can wait for completions through kqueue without any further driver involvement. A driver that wants userland to be able to wait for driver-originated events through kqueue does so through EVFILT\_READ or EVFILT\_WRITE, not through EVFILT\_AIO.

### 35.8.14 Why kqueue Is the Right Answer for Most Drivers

Bringing this together, the guidance for driver authors is clear.

If the driver is a block device, the kernel already wires AIO into the bio path through `aio_qbio`. No additional work is needed. A block driver that serves its strategy routine correctly also serves AIO correctly.

If the driver is a character device that emits events and wants userland to wait for them without blocking a thread, the right mechanism is kqueue. Userland registers EVFILT\_READ or EVFILT\_WRITE on the driver's file descriptor and the driver notifies waiters through `KNOTE_LOCKED`. This is what we have been building throughout this chapter, and it is what the drivers we have read all do.

If the driver is a character device that userland programmers would like to call with `aio_read(2)` for portability reasons, no work is required on the driver side. AIO will serve the request through a worker thread that calls the driver's existing `d_read`. The userland gets the portability it wants; the driver gets to stay simple.

The only time a driver might consider implementing `d_aio_read` or `d_aio_write` is when it has a high-performance, genuinely asynchronous hardware path that can complete work without blocking a worker thread, and when the cost of the worker-thread fallback would be prohibitive. This is exceptionally rare in ordinary drivers, and the drivers that do have such a path (storage drivers, mostly) usually expose it through the block layer rather than as a cdev.



In short: for cdev drivers, “implement AIO” almost always means “implement kqueue.” The remaining AIO machinery belongs to the kernel, not to us. And that is the note we wanted to end this part of the chapter on, because it closes the loop: of the four asynchronous mechanisms (poll, kqueue, SIGIO, AIO), the one that needs the most driver code is kqueue, and the one that needs the least is AIO. The chapter has therefore spent its time on the mechanism that matters.

### 35.8.15 Reading `vfs_aio.c`: A Guide

For readers who want to trace the AIO path through the kernel, `/usr/src/sys/kern/vfs_aio.c` is organised as follows.

Near the top of the file, the `struct kaiocb` and `struct kaioinfo` are discussed (through comments in the surrounding code, since the structures themselves are declared in `/usr/src/sys/sys/aio.h`). The `filt_aioattach/filt_aiodetach/filt_aio` set of static functions and the `aio_filtops` table appear next. These are the kqueue integration for `EVFILT_AIO`.

The `SYSINIT` and module registration come after that, with `aio_onceonly` doing the one-time setup that includes `kqueue_add_filteropts(EVFILT_AIO, &aio_filtops)`. This is where the system-wide `EVFILT_AIO` filter is installed. No driver participates; the AIO module does it alone.

The middle portion of the file is the heart of AIO: `aio_aqueue` (the syscall-layer entry point), `aio_queue_file` (the generic dispatcher), `aio_qbio` (the bio-based path), `aio_process_rw` (the worker-thread path), `aio_complete` (the completion announcement), and `aio_bio_done_notify` (the wake-up fan-out). Tracing from `aio_aqueue` through each of these in turn maps the life of an AIO request from submission to completion.

The completion-signalling functions include `aio_bio_done_notify`, which walks the `kaiocb`’s `knlist` and fires `KNOTE_UNLOCKED` on any registered `EVFILT_AIO` knote, wakes any thread blocked in `aio_suspend`, and delivers any registered signal through `pgsigio`. This is the AIO analogue of the combined producer path we saw in the `evdev` driver.

Cancellation lives in `aio_cancel` and the syscall-layer `kern_aio_cancel`. `aio_cancel` on a `kaiocb` simply calls `aio_complete(job, -1, ECANCELED)`, which pushes the job through the same completion path as a successful one. The userland sees `ECANCELED` instead of a byte count.

The file ends with the syscall implementations for `aio_read`, `aio_write`, `aio_suspend`, `aio_cancel`, `aio_return`, `aio_error`, and friends, plus the `lio_listio` batch submission. These all eventually call into the core dispatchers in the middle of the file.

A reader who traces a userland `aio_read` through `aio_aqueue` to `aio_queue_file`, through either `aio_qbio` or `aio_process_rw`, and then through `aio_complete` back to `aio_bio_done_notify`, has seen the whole AIO path end to end. The file is long but the structure is regular, and the parts that concern drivers are a small fraction of the whole.

### 35.8.16 A Driver-Side Checklist

Now that we have discussed what AIO does and does not ask of drivers, here is a short checklist that driver authors can use as a quick reference.

For a cdev driver that only needs basic readable-event notification, there is nothing to do for AIO. Implement `d_read`, implement `d_poll` or `d_kqfilter` for non-blocking notification, and the userland can use `aio_read(2)` through the AIO worker thread with no additional driver code.

For a cdev driver that wants to be friendly to userland programs using AIO for portability reasons, the same answer applies: nothing extra is needed. The AIO worker thread handles it.

For a block device driver, the bio layer handles AIO through `aio_qbio` and `aio_biowakeup`. A block driver that services its strategy routine correctly also services AIO correctly. Again, nothing extra is needed.

For a driver that has a genuinely asynchronous hardware path and wants to expose it through AIO without going through a worker thread, the `d_aio_read` and `d_aio_write` hooks on `cdevsw` exist but are rare enough that implementing them is outside the scope of this chapter. Such a driver should study the file-ops `fo_aio_queue` mechanism in `/usr/src/sys/kern/vfs_aio.c` and the handful of subsystems that use it.

For every other driver, the answer is simpler still: implement `kqueue`, let userland wait for events the efficient way, and treat AIO as a userland convenience that the kernel handles without the driver's involvement.

### 35.8.17 Wrapping Up Section 5

We now have three independent asynchronous notification mechanisms in our driver: `poll()`, `kqueue()`, and `SIGIO`. Each is relatively small on its own, and each can be implemented without interfering with the others. The pattern, in every case, is the same: register interest in the waiter path, deliver notification in the producer path, and be careful about locking and cleanup.

But these three mechanisms assume the driver has a well-defined notion of “an event is ready.” So far our discussion has been abstract about what an event actually is. In the next section we look at how a driver organizes its events internally, so that a single `read()` call can produce a clean, well-typed record rather than raw hardware state. The internal event queue is the piece that ties the whole asynchronous design together.

## 35.9 6. Internal Event Queues and Message Passing

Up to this point we have treated “an event is ready” as a fuzzy condition. In real drivers, the condition is usually concrete: there is a record in an internal queue. The producer inserts records, the consumer reads them, and the asynchronous notification mechanisms tell the consumer when the queue has gained or lost records. Getting the queue right is what makes the rest of the driver simple.

An event queue has several attributes that distinguish it from a raw byte buffer. Each entry is a structured record, not a stream of bytes: a typed event with a payload. Entries are delivered whole, not partially: a reader either gets a complete record or gets no record. The queue has a bounded size, so producers must have a policy for what happens when the queue fills up: drop the oldest, drop the newest, report an error, or wait for room. And the queue is consumed in order: events are delivered in the order they were inserted, unless the design explicitly allows otherwise.

Designing the queue carefully pays off throughout the driver. A reader who sees a stream of well-typed records can write simple, robust userland code. A producer who knows the queue's policy on overflow can make sensible decisions when events arrive faster than they can be consumed. The asynchronous notification mechanisms (`poll`, `kqueue`, `SIGIO`) all become cleaner, because they can each express their condition in terms of queue emptiness rather than in terms of arbitrary per-device state.

### 35.9.1 Designing the Event Record

The first decision is what a single event looks like. A minimal record for our evdemo driver:

```
struct evdemo_event {
 struct timespec ev_time; /* timestamp */
 uint32_t ev_type; /* event type */
 uint32_t ev_code; /* event code */
 int64_t ev_value; /* event value */
};
```

This mirrors the layout of real event interfaces like evdev, which is no accident: a timestamp plus a (type, code, value) triple is enough to describe most event streams, from keyboard keypresses to sensor readings to button events on a game controller. The timestamp lets userland reconstruct when the event happened regardless of when it was consumed, which matters for latency- sensitive applications.

A driver that needs more structure can add fields, but the discipline of keeping the record fixed-size is worth defending. A fixed-size record makes the queue's memory management easy, makes the read path a simple copy, and avoids ABI issues that arise when records have variable length.

### 35.9.2 The Ring Buffer

The queue itself can be a simple ring buffer of fixed capacity:

```
#define EVDEMO_QUEUE_SIZE 64

struct evdemo_softc {
 /* ... */
 struct evdemo_event sc_queue[EVDEMO_QUEUE_SIZE];
 u_int sc_qhead; /* next read position */
 u_int sc_qtail; /* next write position */
 u_int sc_nevents; /* count of queued events */
 u_int sc_dropped; /* overflow count */
 /* ... */
};

static inline bool
evdemo_queue_empty(const struct evdemo_softc *sc)
{
 return (sc->sc_nevents == 0);
}

static inline bool
evdemo_queue_full(const struct evdemo_softc *sc)
{

```

```

 return (sc->sc_nevents == EVDEMO_QUEUE_SIZE);
}

static void
evdemo_enqueue(struct evdemo_softc *sc, const struct evdemo_event *ev)
{
 mtx_assert(&sc->sc_mtx, MA_OWNED);

 if (evdemo_queue_full(sc)) {
 /* Overflow policy: drop oldest. */
 sc->sc_qhead = (sc->sc_qhead + 1) % EVDEMO_QUEUE_SIZE;
 sc->sc_nevents--;
 sc->sc_dropped++;
 }

 sc->sc_queue[sc->sc_qtail] = *ev;
 sc->sc_qtail = (sc->sc_qtail + 1) % EVDEMO_QUEUE_SIZE;
 sc->sc_nevents++;
}

static int
evdemo_dequeue(struct evdemo_softc *sc, struct evdemo_event *ev)
{
 mtx_assert(&sc->sc_mtx, MA_OWNED);

 if (evdemo_queue_empty(sc))
 return (-1);

 *ev = sc->sc_queue[sc->sc_qhead];
 sc->sc_qhead = (sc->sc_qhead + 1) % EVDEMO_QUEUE_SIZE;
 sc->sc_nevents--;
 return (0);
}

```

Several things about this code are worth calling out.

We use a simple modular arithmetic ring rather than a linked list. This keeps the memory footprint fixed, avoids allocations at event time, and makes the queue lock-free from a cache-line point of view (two reads and a write per operation). Most drivers with this pattern use a ring.

We track `sc_nevents` separately from the head and tail pointers. Using head and tail alone, without a count, leads to the classic ambiguity between “empty” and “full”: when head equals tail, the queue could be either state. The count field resolves the ambiguity and makes the fast paths cheap.

We have an overflow policy baked into `evdemo_enqueue`. When the queue is full, we drop the oldest event. This is the right policy for an event stream where recent events are more valuable than stale ones; a security

log or a metrics stream might prefer the opposite. We also increment `sc_dropped` so that userland can tell how many events were lost.

Both `evdemo_enqueue` and `evdemo_dequeue` assert that the softc mutex is held. This is a structural safety net: if the caller forgets to take the lock, the assertion fires on a debug kernel and points at exactly the wrong call site. Without the assertion, the bug might manifest only under rare timing as silent queue corruption.

### 35.9.3 The Read Path

With the queue in place, the synchronous `read()` handler becomes short:

```
static int
evdemo_read(struct cdev *dev, struct uio *uio, int flag)
{
 struct evdemo_softc *sc = dev->si_drv1;
 struct evdemo_event ev;
 int error = 0;

 while (uio->uio_resid >= sizeof(ev)) {
 mtx_lock(&sc->sc_mtx);
 while (evdemo_queue_empty(sc) && !sc->sc_detaching) {
 if (flag & O_NONBLOCK) {
 mtx_unlock(&sc->sc_mtx);
 return (error ? error : EAGAIN);
 }
 error = cv_wait_sig(&sc->sc_cv, &sc->sc_mtx);
 if (error != 0) {
 mtx_unlock(&sc->sc_mtx);
 return (error);
 }
 }
 if (sc->sc_detaching) {
 mtx_unlock(&sc->sc_mtx);
 return (0);
 }
 evdemo_dequeue(sc, &ev);
 mtx_unlock(&sc->sc_mtx);

 error = uiomove(&ev, sizeof(ev), uio);
 if (error != 0)
 return (error);
 }
 return (0);
}
```

The pattern is standard: loop while the caller has buffer space left, wait for a record if the queue is empty, dequeue one under the lock, release the lock, copy out through `uiomove(9)`. We handle `O_NONBLOCK` by returning `EAGAIN` when the queue is empty, and we handle detach by returning zero (end-of-file) so that readers can cleanly terminate.

The `cv_wait_sig()` call is a condition variable wait that also returns on signal delivery, so that a reader blocked in `read()` can be interrupted by `SIGINT` or other signals. This is the interruptible-wait pattern you may remember from earlier chapters on synchronization. The condition variable is signalled from the producer path, which we look at next.

### 35.9.4 Integrating the Producer Path

The producer now has three things to do: enqueue the event, signal any blocked readers through the condition variable, and deliver asynchronous notifications through the three mechanisms we have studied:

```
static void
evdemo_post_event(struct evdemo_softc *sc, struct evdemo_event *ev)
{
 bool async;

 mtx_lock(&sc->sc_mtx);
 evdemo_enqueue(sc, ev);
 async = sc->sc_async;
 cv_broadcast(&sc->sc_cv);
 KNOTE_LOCKED(&sc->sc_rsel.si_note, 0);
 mtx_unlock(&sc->sc_mtx);

 selwakeup(&sc->sc_rsel);
 if (async)
 pgsigio(&sc->sc_sigio, SIGIO, 0);
}
```

This is the producer's canonical shape. All the state updates and all the in-lock notifications happen inside the softc mutex; the out-of-lock notifications happen outside. The order matters: the in-lock `cv_broadcast` and `KNOTE_LOCKED` happen before we drop the lock, and the out-of-lock `selwakeup` and `pgsigio` happen after.

One detail is the use of `cv_broadcast()` rather than `cv_signal()`. If multiple readers are blocked in `read()`, we usually want to wake all of them so that each can try to claim a record. With `cv_signal()` we wake only one, and the others stay asleep until another event arrives. In a single-reader design `cv_signal()` would be fine; in the general case `cv_broadcast()` is safer.

### 35.9.5 The Poll and Kqueue Integration

The beauty of the internal event queue is that `d_poll` and `d_kqfilter` become one-liners in terms of queue state:

```

static int
evdemo_poll(struct cdev *dev, int events, struct thread *td)
{
 struct evdemo_softc *sc = dev->si_drv1;
 int revents = 0;

 mtx_lock(&sc->sc_mtx);
 if (events & (POLLIN | POLLRDNORM)) {
 if (!evdemo_queue_empty(sc))
 revents |= events & (POLLIN | POLLRDNORM);
 else
 selrecord(td, &sc->sc_rsel);
 }
 if (events & (POLLOUT | POLLWRNORM))
 revents |= events & (POLLOUT | POLLWRNORM);
 mtx_unlock(&sc->sc_mtx);

 return (revents);
}

static int
evdemo_kqread(struct knote *kn, long hint)
{
 struct evdemo_softc *sc = kn->kn_hook;

 mtx_assert(&sc->sc_mtx, MA_OWNED);

 kn->kn_data = sc->sc_nevents;
 if (sc->sc_detaching) {
 kn->kn_flags |= EV_EOF;
 return (1);
 }
 return (!evdemo_queue_empty(sc));
}

```

The readable filter reports `kn->kn_data` as the number of queued events, and returns true whenever the queue is non-empty. The userland program sees `kn_data` and can tell how many events are available without having to call `read()` yet. This is a small but useful feature of the `kqueue` API, and it costs us nothing to support.

### 35.9.6 Exposing Queue Metrics Through `sysctl`

A diagnostic-friendly driver exposes its queue state through `sysctl(9)`. For `evdemo` we add a few counters:

```
SYSCALL_NODE(_dev, OID_AUTO, evdemo, CTLFLAG_RW, 0, "evdemo driver");

SYSCALL_UINT(_dev_evdemo, OID_AUTO, qsize, CTLFLAG_RD,
 &evdemo_qsize, 0, "queue capacity");
SYSCALL_UINT(_dev_evdemo, OID_AUTO, qlen, CTLFLAG_RD,
 &evdemo_qlen, 0, "current queue length");
SYSCALL_UINT(_dev_evdemo, OID_AUTO, dropped, CTLFLAG_RD,
 &evdemo_dropped, 0, "events dropped due to overflow");
SYSCALL_UINT(_dev_evdemo, OID_AUTO, posted, CTLFLAG_RD,
 &evdemo_posted, 0, "events posted since attach");
SYSCALL_UINT(_dev_evdemo, OID_AUTO, consumed, CTLFLAG_RD,
 &evdemo_consumed, 0, "events consumed by read(2)");
```

These can be turned into counter(9) counters for cache-friendliness on multi-core systems, but a simple `uint32_t` is fine for instructional purposes. With these counters, a `sysctl dev.evdemo` invocation shows the queue's runtime state at a glance, which is invaluable when debugging a driver that seems to be missing events or dropping them.

### 35.9.7 Overflow Policies: A Design Discussion

Our code drops the oldest event when the queue fills. Let us think about when that is the right choice and when it is not.

Dropping the oldest is right when recent events are more valuable than old ones. A user-interface event queue is a good example: a program that woke up to find a hundred keystrokes queued usually cares about the most recent ones, not the ones from five minutes ago. A telemetry stream where each record is timestamped is similar: the old records are stale.

Dropping the newest is right when the queue represents a ledger that must not have gaps. A security log should never lose an event to overflow; it should rather refuse to log the newest event (and increment a “dropped” counter) than silently rewrite history.

Blocking the producer is right when the producer can actually wait. A driver whose producer is an interrupt handler cannot block; a driver whose producer is a user-space write call can. If the producer can wait, then a full queue becomes back-pressure that slows the producer down to match the consumer, which is often exactly what you want.

Returning an error is right for a request-response protocol where the caller needs to know immediately whether the command succeeded. This is more common in `ioctl` paths than in event queues, but it is a valid policy.

The common mistake is to pick a policy without thinking about which one fits the device. A driver that drops old events when a security log is the underlying data will lose evidence. A driver that drops new events when a UI needs responsiveness will feel laggy. Picking the right policy is a design decision, and it is worth documenting it in the driver's comments so that future maintainers understand why you chose what you chose.



### 35.9.8 Avoiding Partial Reads

One small but important detail: the read path must either deliver a complete event or no event at all. It must not copy out half an event and return a short read count, because the userland caller would then have to reconstruct the event across multiple calls, which is fragile and error-prone.

The simplest way to enforce this is the guard at the top of the loop:

```
while (uio->uio_resid >= sizeof(ev)) {
 /* ... */
}
```

If the user's buffer has fewer bytes left than one event, we simply stop. The caller gets exactly as many complete events as fit. If the caller passed a zero-length buffer, we return immediately with zero bytes, which is the convention for an empty read.

### 35.9.9 Handling Event Coalescing

Some drivers have legitimate reasons to coalesce events. If a keyboard produces “key pressed” followed immediately by “key released” for the same key, the driver might be tempted to collapse these into a single “key tapped” event to save queue space. Our advice is to resist this temptation in most cases. Coalescing changes the event semantics and can confuse userland programs that were written to expect raw events.

Where coalescing is justified (for example, coalescing mouse movements in a way that preserves the final position), implement it carefully and document it. The coalescing logic should live in the enqueue path, not in the consumer path, so that all consumers see consistent behaviour.

### 35.9.10 Wrapping Up Section 6

The internal event queue is what ties the asynchronous mechanisms together. Every notification, every readable check, every kqueue filter, every SIGIO delivery: all of them reduce to “is the queue non-empty, or not?” Once the queue is in place, the rest of the driver becomes a matter of wiring, not of design.

In the next section we look at the design patterns for combining poll, kqueue, and SIGIO in a single driver, and at the locking audit that ensures the combination is correct. Adding each mechanism individually was the easy part. Making them all work together, with one producer and many simultaneous waiters of different kinds, is where real driver engineering happens.

## 35.10 7. Combining Asynchronous Techniques

So far we have looked at poll, kqueue, and SIGIO one at a time, each in its own section, each with its own lock discipline and wakeup pattern. In a real driver, all three mechanisms coexist. A single producer path has to wake condition-variable sleepers, poll waiters, kqueue knotes, and signal owners, in a specific order, under specific locks, without dropping any wakeup and without deadlocking.

This section is about getting that combination right. It is largely a review and a consolidation: we have seen each mechanism individually, and now we see them together. The review is worth doing because the interactions between mechanisms are exactly the place where driver bugs like to hide. Small differences in lock ordering or notification timing that would not cause a visible problem with one mechanism alone can lead to missed wakeups or deadlocks when several mechanisms are layered.

### 35.10.1 When to Use Each Mechanism

A driver that supports all three mechanisms lets its userland clients choose the right tool for the job. The three mechanisms have different strengths:

`poll` and `select` are the most portable. A userland program that needs to run unchanged on a wide range of UNIX systems will use `poll`. Drivers should support `poll` because it is the lowest common denominator, and implementing it is cheap.

`kqueue` is the most efficient and most flexible. Userland programs that watch thousands of descriptors should use `kqueue`. Drivers should support `kqueue` because it is the preferred mechanism for new FreeBSD code and because most applications that care about performance will choose it.

`SIGIO` is the simplest for a specific class of programs: shell scripts using `trap`, small single-threaded programs that want the simplest possible notification, and legacy code. Drivers should support `SIGIO` because the work is minimal and the supported use cases are real.

In practice, almost every character driver for an event-driven device should implement all three. The code is small, the maintenance is low, and the userland flexibility is high.

### 35.10.2 The Producer Path Template

The canonical producer path for a driver supporting all three mechanisms is:

```
static void
driver_post_event(struct driver_softc *sc, struct event *ev)
{
 bool async;

 mtx_lock(&sc->sc_mtx);
 enqueue_event(sc, ev);
 async = sc->sc_async;
 cv_broadcast(&sc->sc_cv);
 KNOTE_LOCKED(&sc->sc_rsel.si_note, 0);
 mtx_unlock(&sc->sc_mtx);

 selwakeup(&sc->sc_rsel);
 if (async)
 pgsigio(&sc->sc_sigio, SIGIO, 0);
}
```

Every piece of this template has a reason for its placement.

The `mtx_lock` acquires the softc mutex. This is the single lock that serializes all state transitions in the driver, and all readers and writers respect it.

`enqueue_event` is inside the lock. The queue is the shared state, and any update to it must be atomic relative to other updates and to state reads.

`async = sc->sc_async` is inside the lock. This captures a consistent snapshot of the `async` flag so that we can use it outside the lock without racing.

`cv_broadcast` is inside the lock. Condition variables require that the associated mutex be held when signalling. The signal is delivered immediately, but the actual wakeup of a blocked thread happens when the mutex is released.

`KNOTE_LOCKED` is inside the lock. It walks the knote list and delivers kqueue notifications, and it expects the knlist's lock (which is our softc mutex) to be held.

`mtx_unlock` releases the softc mutex. After this point we are outside the critical section.

`selwakeup` is outside the lock. This is the canonical ordering for `selwakeup`: it must not be called inside arbitrary driver locks because it takes its own internal locks.

`pgsigio` is outside the lock for the same reason.

This order is the least-error-prone arrangement. Many variations are possible, but departures from this pattern need to be justified by a specific reason.

### 35.10.3 Lock Ordering

With four distinct notification calls and one state update, lock ordering matters. Let us work through what locks are in play.

The softc mutex is acquired first and held across the state update and the in-lock notifications.

`cv_broadcast` does not acquire any additional locks beyond the one we already hold.

`KNOTE_LOCKED` evaluates each knote's `f_event` callback. The callbacks execute with the knlist's lock (our softc mutex) held. Those callbacks must not try to acquire any additional locks, because doing so would create a nested acquisition that other paths (say, the consumer in `d_poll`) might take in the opposite order. In practice, `f_event` callbacks only read state, which is exactly what we designed for.

`selwakeup` acquires the `selinfo`'s internal mutex and walks the list of parked threads, waking them. This is done outside the softc mutex. Internally, `selwakeup` also walks the `selinfo`'s knote list, but that was already handled by our earlier `KNOTE_LOCKED` call; doing it twice is harmless but wasteful, so we do the `KNOTE_LOCKED` while we have the lock and let `selwakeup` just handle the thread list.

`pgsigio` acquires the signal-related locks and delivers the signal to the owning process or process group. This is outside the softc mutex.

The lock ordering rule is: softc mutex first, never nested inside the `selinfo` or signal locks. As long as we follow this order, we cannot deadlock.

### 35.10.4 The Consumer Paths

Each of the three consumer paths uses the softc mutex in a consistent way:

```
/* Condition-variable consumer: d_read */
mtx_lock(&sc->sc_mtx);
while (queue_empty(sc))
 cv_wait_sig(&sc->sc_cv, &sc->sc_mtx);
dequeue(sc, ev);
mtx_unlock(&sc->sc_mtx);

/* Poll consumer: d_poll */
mtx_lock(&sc->sc_mtx);
if (queue_empty(sc))
 selrecord(td, &sc->sc_rsel);
else
 revents |= POLLIN;
mtx_unlock(&sc->sc_mtx);

/* Kqueue consumer: f_event */
/* Called with softc mutex already held by the kqueue framework */
return (!queue_empty(sc));

/* SIGIO consumer: handled entirely in userland; the driver
* only sends the signal, never consumes it */
```

All three consumers check the queue under the softc mutex. This is what closes the race between the producer's state update and the consumer's check: if the producer has the lock, the consumer waits and sees the post-update state; if the consumer has the lock, the producer waits and posts after the consumer has registered.

### 35.10.5 Common Pitfalls

A few specific bugs come up often enough to name them explicitly.

**Forgetting one of the notification calls in the producer.** The canonical ordering looks like a boilerplate sequence, and it is easy to leave out one of the four calls. Tests that exercise only one mechanism will pass, but the other mechanisms will be broken. Code review and automated tests help here.

**Holding the lock during selwakeup or pgsigio.** The chapter's advice is to drop the lock before these calls. Some drivers accidentally hold the lock (for example, because the producer is deep in an unlocked-lock-unlocked pattern that is hard to refactor). The result is a latent deadlock that manifests only when a specific lock is held by a different path.

**Calling cv\_signal instead of cv\_broadcast.** A single-reader driver can use cv\_signal. A driver that allows multiple readers must use cv\_broadcast, because only one of the signalled waiters will succeed in dequeuing an event and the others must see the updated state to re-sleep. If you pick cv\_signal and then allow multiple readers later, you have introduced a latent missed wakeup that only appears under contention.

**Forgetting `knlist_init_mtx` at attach.** A driver that never initializes its `knlist` will crash on the first `KNOTE_LOCKED` call, because the `knlist`'s lock function pointers are null. The symptom is a null-pointer dereference inside `knote()`, and it can be confusing if you forgot the `init` call in a refactor.

**Forgetting `funsetown` at close.** A process that enabled `FIOASYNC` and then exited without closing the `fd` leaves a stale `struct sigio` behind. The kernel handles process exit through an `eventhandler(9)` that calls `funsetown` for us, so this is usually safe, but leaking the structure during close is still a bug.

**Forgetting `seldrain` and `knlist_destroy` at detach.** Waiters parked on the `selinfo` must be woken when the device goes away. Forgetting this leaves waiters asleep forever and can panic the kernel when the `selinfo` is freed.

### 35.10.6 Testing the Combined Design

The best way to test a driver that supports all three mechanisms is to run three userland programs in parallel:

A `poll`-based reader that watches for events and prints them.

A `kqueue`-based reader that does the same with `EVFILT_READ`.

A `SIGIO`-based reader that enables `FIOASYNC` and prints on each signal.

Trigger events in the driver at a known rate and verify that all three readers see them all. If any reader lags or misses events, there is a bug in that mechanism's wiring. Counters on the driver side help here: if the driver reports 1000 events posted but a reader reports 900 events seen, one in ten notifications is being dropped.

Running all three readers at once against the same device stresses the producer in a way single-mechanism tests do not. Any lock ordering bug that only manifests when all three are active will show up under this workload.

### 35.10.7 Application Compatibility

A well-behaved driver can expect to work with legacy and modern userland code, with single-threaded and multi-threaded programs, with code that picks one mechanism and code that picks another. The way to achieve this is to support all three mechanisms and to honour each one's documented contract.

Legacy `select`-based code should work through our `poll` implementation, because `select` is translated into `poll` in the kernel.

Modern `kqueue`-based code should work through our `d_kqfilter`, because `kqueue` is the native mechanism for event-driven userland on FreeBSD.

Single-threaded programs using `SIGIO` should work through our `FIOASYNC/FIOSETOWN` handling.

Programs that mix mechanisms (for instance, watching some descriptors with `kqueue` and using `SIGIO` for urgent events) should also work, because the driver's producer path notifies all mechanisms on every event.

This is what "application compatibility" means for a driver. Honour the contracts, notify all waiters, handle cleanup correctly, and userland code of any vintage will work.

### 35.10.8 Wrapping Up Section 7

We have a complete picture now. Three asynchronous mechanisms, one producer, one queue, one set of locks, one detach sequence. The combined design is not much more code than any single mechanism alone; the art is in getting the locks and the ordering right, and in testing the combination so that latent bugs are found before they ship.

The next section takes this combined design and applies it as a refactor of our evolving evdemo driver. We will audit the final code, look at what changed, and release the driver as version v2.5-async. The refactor is where the abstract advice turns into concrete, working source.

## 35.11 8. Final Refactor for Asynchronous Support

The previous sections built up evdemo one mechanism at a time, so the code we have now is a working but slightly haphazard accumulation. In this section we refactor the driver as a single coherent whole, with a consistent locking discipline, a complete detach path, and a set of exposed counters that let us observe its behaviour. The result is the companion driver at `examples/part-07/ch35-async-io/lab06-v25-async/`, which serves as the reference implementation for the exercises in this chapter.

Calling this the “final” refactor is slightly aspirational: a real driver is never truly finished. But refactoring after a feature is built is a useful habit, because it is when the structure of the code becomes visible as a whole rather than as a series of additions. Bugs that hid during incremental development often become obvious once the code is laid out as a single flow.

### 35.11.1 Thread-Safety Review

Our review starts with the locking. Every state element in the softc is now protected by `sc_mtx`, with the following exceptions:

`sc_sigio` is protected internally by the `SIGIO_LOCK` global, not by our softc mutex. This is correct, because the `fsetown`, `fgetown`, `funsetown`, and `pgsigio` APIs take the global lock themselves. We must not take `sc_mtx` before calling those APIs, or we would invert lock order with the rest of the kernel’s signal code.

`sc_rsel` is protected internally by its own `selinfo` mutex. We do not touch the internal list directly; we only call `selrecord` and `selwakeup`. Those functions take the internal lock themselves.

Everything else (the queue, the counters, the async flag, the detaching flag, the condition variable wait queue) is protected by `sc_mtx`.

The audit is: every code path that reads or writes one of these fields takes `sc_mtx` before the access and drops it after. Let us walk through each path.

Attach: `sc_mtx` is initialized before any access. Everything else is zeroed. No concurrent access is possible at attach time because no handle to the driver exists yet.

Detach: `sc_mtx` is taken to set `sc_detaching = true`, `cv_broadcast` and `KNOTE_LOCKED` are issued, the lock is dropped, `selwakeup` is called, and `destroy_dev_drain` is invoked. After `destroy_dev_drain` returns, no more calls to our callbacks can start. We can then `seldrain`, `knlist_destroy`, `funsetown`, `mtx_destroy`, `cv_destroy`, and free the softc.

Open: `sc_mtx` is not strictly needed because the open is serialized by the kernel, but taking it for internal state updates is cheap and clarifies the code.

Close: `funsetown` is called outside `sc_mtx`.

Read: `sc_mtx` is held around the queue check, the `cv_wait_sig` call, and the dequeue. The `uiomove` is done outside the lock, because `uiomove` might page-fault and we do not want to hold driver locks across faults.

Write: not applicable in `evdemo`, but in a driver that accepts writes the pattern is symmetric.

Ioctl: `FIOASYNC` takes `sc_mtx`; `FIOSETOWN` and `FIOGETOWN` do not, because they use `fsetown/fgetown` which have their own locking.

Poll: `sc_mtx` is held across the check and `selrecord` call.

Kqfilter: `sc_mtx` is taken by the kqueue framework before calling our `f_event` callback. Our `d_kqfilter` takes it for the `knlist_add` call.

Producer (`evdemo_post_event` from the callout): `sc_mtx` is held across the enqueue, the `cv_broadcast`, and the `KNOTE_LOCKED` call; dropped before `selwakeup` and `pgsigio`.

Every read and write of every softc field is accounted for under `sc_mtx` or under the appropriate external lock. This is the audit you want to perform on every asynchronous driver, because it is the audit that finds latent concurrency bugs before they ship.

### 35.11.2 The Complete Attach Sequence

Putting the attach path together, in the order the calls must happen:

```
static int
evdemo_modevent(module_t mod, int event, void *arg)
{
 struct evdemo_softc *sc;
 int error = 0;

 switch (event) {
 case MOD_LOAD:
 sc = malloc(sizeof(*sc), M_EVDemo, M_WAITOK | M_ZERO);
 mtx_init(&sc->sc_mtx, "evdemo", NULL, MTX_DEF);
 cv_init(&sc->sc_cv, "evdemo");
 knlist_init_mtx(&sc->sc_rsel.si_note, &sc->sc_mtx);
 callout_init_mtx(&sc->sc_callout, &sc->sc_mtx, 0);

 sc->sc_dev = make_dev(&evdemo_cdevsw, 0, UID_ROOT, GID_WHEEL,
 0600, "evdemo");
 sc->sc_dev->si_drv1 = sc;
 evdemo_sc_global = sc;
 break;
 /* ... */
}
```

```
 }
 return (error);
}
```

The order is deliberate: first initialize all the synchronization primitives, then register the callbacks (which can start arriving at any time after the `make_dev` call), then publish the softc via `si_drv1` and the global pointer.

One subtlety is `M_WAITOK`. We want a blocking allocation at attach time because we are in a module-load context, which is always allowed to sleep. `M_ZERO` is essential because an uninitialized `selinfo`, `knlist`, or condition variable will crash the kernel. With these flags, the allocation either succeeds with a zeroed structure or the module load fails cleanly.

### 35.11.3 The Complete Detach Sequence

The detach path is more delicate, because we have to coordinate with in-flight callers and active waiters:

```
case MOD_UNLOAD:
 sc = evdemo_sc_global;
 if (sc == NULL)
 break;

 mtx_lock(&sc->sc_mtx);
 sc->sc_detaching = true;
 cv_broadcast(&sc->sc_cv);
 KNOTE_LOCKED(&sc->sc_rsel.si_note, 0);
 mtx_unlock(&sc->sc_mtx);
 selwakeup(&sc->sc_rsel);

 callout_drain(&sc->sc_callout);
 destroy_dev_drain(sc->sc_dev);

 seldrain(&sc->sc_rsel);
 knlist_destroy(&sc->sc_rsel.si_note);
 funsetown(&sc->sc_sigio);

 cv_destroy(&sc->sc_cv);
 mtx_destroy(&sc->sc_mtx);

 free(sc, M_EVDemo);
 evdemo_sc_global = NULL;
 break;
```

The sequence here is worth studying because it contains several order-sensitive steps.



Setting `sc_detaching` under the lock and broadcasting is what lets blocked readers wake up and see the flag. Without this, a reader stuck in `cv_wait_sig` would sleep forever because we are about to destroy the condition variable.

The `KNOTE_LOCKED` call (with the `EV_EOF` path in `f_event`) lets any kqueue waiters see the end-of-file.

The `selwakeup` outside the lock wakes poll waiters. They return to userland and see their file descriptors becoming invalid.

The `callout_drain` stops the simulated event source. Any callout that is about to fire completes first; no new ones start.

The `destroy_dev_drain` waits for any in-flight callbacks to return. After this, `d_open`, `d_close`, `d_read`, `d_write`, `d_ioctl`, `d_poll`, and `d_kqfilter` are all guaranteed to have returned.

The `sel drain` cleans up any lingering selinfo state.

The `knlist_destroy` verifies the knote list is empty (it should be, because every knote's `f_detach` was called when the file descriptor closed) and frees the internal lock state.

The `funsetown` clears the signal owner.

Finally we destroy the condition variable and the mutex, free the `softc`, and clear the global pointer.

This careful ordering is the difference between a driver that unloads cleanly and a driver that panics on the second load. The test regimen for any serious driver includes a “load and unload a hundred times in a loop” exercise, because the race windows in a detach path are often too narrow to hit on a single try.

### 35.11.4 Exposing Event Metrics

The finished driver exposes its event metrics through `sysctl`:

```

SYSCTL_NODE(_dev, OID_AUTO, evdemo, CTLFLAG_RW, 0, "evdemo driver");

static SYSCTL_NODE(_dev_evdemo, OID_AUTO, stats,
 CTLFLAG_RW, 0, "Runtime statistics");

SYSCTL_UINT(_dev_evdemo_stats, OID_AUTO, posted, CTLFLAG_RD,
 &evdemo_posted, 0, "Events posted since attach");
SYSCTL_UINT(_dev_evdemo_stats, OID_AUTO, consumed, CTLFLAG_RD,
 &evdemo_consumed, 0, "Events consumed by read(2)");
SYSCTL_UINT(_dev_evdemo_stats, OID_AUTO, dropped, CTLFLAG_RD,
 &evdemo_dropped, 0, "Events dropped due to overflow");
SYSCTL_UINT(_dev_evdemo_stats, OID_AUTO, qlen, CTLFLAG_RD,
 &evdemo_qlen, 0, "Current queue length");
SYSCTL_UINT(_dev_evdemo_stats, OID_AUTO, selwakeups, CTLFLAG_RD,
 &evdemo_selwakeups, 0, "selwakeup calls");
SYSCTL_UINT(_dev_evdemo_stats, OID_AUTO, knotes_delivered, CTLFLAG_RD,
 &evdemo_knotes_delivered, 0, "knote deliveries");
SYSCTL_UINT(_dev_evdemo_stats, OID_AUTO, sigio_sent, CTLFLAG_RD,
 &evdemo_sigio_sent, 0, "SIGIO signals sent");

```

Each counter is incremented under the softc lock from the producer. The counters are not required for correct operation, but they are required for the driver to be observable. A driver that reports zero events consumed while the queue is full tells us the reader is not draining. A driver that reports more selwakeups than knotes delivered tells us something about the mix of waiters. A driver that reports many sigio\_sent but no visible effect in userland tells us to check the owner's signal handler.

Observability costs almost nothing to add and pays for itself many times over in production debugging. Adding it to the final refactor is part of what makes the driver ready for real use.

### 35.11.5 Versioning the Driver

We tag this version as v2.5-async in the code and in the companion example directory. The convention is a simple `MODULE_VERSION` declaration:

```
MODULE_VERSION(evdemo, 25);
```

The number is the integer form of the version: 25 for 2.5. FreeBSD's module-loading infrastructure uses this number to enforce dependency constraints between modules. A module that depends on evdemo at a specific version can declare that with `MODULE_DEPEND(9)`. For our standalone driver the version is mostly informative, but bumping it with every feature release is a good habit.

### 35.11.6 Wrapping Up Section 8

The final evdemo driver supports blocking and non-blocking `read()`, `poll()` with `selrecord/selwakeup`, `kqueue()` with `EVFILT_READ`, and `SIGIO` through `FIOASYNC/FIOSETOWN`. It has a bounded internal event queue with a drop-oldest overflow policy. It exposes counters through `sysctl` for observability. Its attach and detach sequences are audited for thread safety. It is a small driver, about four hundred lines of C, but it demonstrates every pattern this chapter has taught.

More importantly, it is a template. The patterns you have seen here generalize to any driver that needs asynchronous I/O. A USB input device replaces the simulated callout with a real URB callback. A GPIO driver replaces the callout with a real interrupt handler. A network pseudo-device replaces the event queue with an mbuf chain. The asynchronous notification framework (`poll`, `kqueue`, `SIGIO`) stays the same across all of these. Once you know the pattern, adding asynchronous support to a new driver is a matter of wiring, not design.

We have now covered the chapter's core material. The next part of the chapter is hands-on: a sequence of labs that walks you through building evdemo yourself, adding each mechanism in turn, and verifying the behaviour with real userland programs. If you have been reading without running code, now is the time to open a terminal on your FreeBSD virtual machine and start typing.

## 35.12 Hands-On Labs

The labs in this section build evdemo incrementally. Each lab corresponds to a folder under `examples/part-07/ch35-async-io/` in this book's companion source. You can either type each lab from scratch (which is

slower but builds stronger intuition), or start from the provided sources and focus on the code the lab is teaching. Either approach is fine; pick whichever matches your learning style.

A few general notes before we begin.

Every lab uses the same `Makefile` pattern. A `KMOD` line names the module, a `SRCS` line lists the sources, and `bsd.kmod.mk` does the rest. Run `make` in the lab directory to produce `evdemo.ko`, and `sudo kldload ./evdemo.ko` to load it. `make test` builds the user-space test programs in the same directory.

Every lab exposes a device node at `/dev/evdemo`. If you forget to unload a previous version of the driver before building a new one, the load will fail with “device already exists.” Run `sudo kldunload evdemo` to clean up, then reload.

Every lab includes a small test program that exercises the mechanism the lab is teaching. Running the test program alongside the driver verifies that the mechanism works end to end. If a test program hangs or reports an error, something in the driver is broken, and the lab’s troubleshooting notes will usually help you find it.

### 35.12.1 Lab 1: Synchronous Baseline

The first lab establishes a synchronous baseline that the later labs build on. Our goal here is a minimal `evdemo` driver that supports blocking `read()` on an internal event queue. No asynchronous mechanisms yet. This lab teaches the queue data structures and the condition-variable pattern that everything else will layer on top of.

#### Files:

- `evdemo.c` - driver source
- `evdemo.h` - shared header with event record definition
- `evdemo_test.c` - user-space reader
- `Makefile` - module build plus test target

#### Steps:

1. Read the contents of the lab directory. Familiarize yourself with the structure of `evdemo_softc`, especially the queue fields and the condition variable.
2. Build the driver: `make`.
3. Build the test program: `make test`.
4. Load the driver: `sudo kldload ./evdemo.ko`.
5. In one terminal, run the test program: `sudo ./evdemo_test`. The program opens `/dev/evdemo` and calls `read()`, which will block because no events have been posted.
6. In a second terminal, trigger events: `sudo sysctl dev.evdemo.trigger=1`. The `sysctl` is wired up in the driver to call `evdemo_post_event` with a synthetic event. The test program should unblock, print the event, and call `read()` again.
7. Trigger a few more events. Watch the test program print each one as it arrives.
8. Unload the driver: `sudo kldunload evdemo`.

**What to observe:** The `read()` call in the test program blocks while the queue is empty and returns exactly one event at a time. The test program does not spin on the CPU while waiting; you can confirm this by watching

`top -H` in a third terminal and noting that the test process is in the S (sleeping) state on a wait channel named something like `evdemo` or the generic `cv`.

**Common mistakes to check for:** If the test program returns immediately with zero bytes, the queue may be reporting itself as empty but the `read()` path is not waiting on the condition variable. Check that the while loop in `evdemo_read` is actually calling `cv_wait_sig`. If the test program hangs and never unblocks even after triggering an event, check that the producer is actually calling `cv_broadcast` inside the mutex.

**Takeaway:** Blocking `read()` with a condition variable is the synchronous baseline. It works, but it is not enough for programs that need to watch multiple descriptors or react to events without having a thread blocked in `read()` all the time. The next labs add asynchronous support.

### 35.12.2 Lab 2: Adding `poll()` Support

The second lab adds `d_poll` to the driver so that userland programs can wait on multiple descriptors or integrate `evdemo` into an event loop. This lab teaches the `selrecord/selwakeup` pattern.

#### Files:

- `evdemo.c` - driver source (extended from Lab 1)
- `evdemo.h` - shared header
- `evdemo_test_poll.c` - poll-based test program
- `Makefile` - module build plus test target

#### Changes in the driver from Lab 1:

Add a `struct selinfo sc_rsel` to the `softc`.

Initialize it with `knlist_init_mtx(&sc->sc_rsel.si_note, &sc->sc_mtx)` during attach. Even though we are not yet using `kqueue`, pre-initializing the `si_note` knlist is cheap and makes the `selinfo` compatible with `kqueue` support later.

Add a `d_poll` callback:

```
static int
evdemo_poll(struct cdev *dev, int events, struct thread *td)
{
 struct evdemo_softc *sc = dev->si_drv1;
 int revents = 0;

 mtx_lock(&sc->sc_mtx);
 if (events & (POLLIN | POLLRDNORM)) {
 if (!evdemo_queue_empty(sc))
 revents |= events & (POLLIN | POLLRDNORM);
 else
 selrecord(td, &sc->sc_rsel);
 }
 if (events & (POLLOUT | POLLWRNORM))
 revents |= events & (POLLOUT | POLLWRNORM);
}
```

```
mtx_unlock(&sc->sc_mtx);

return (revents);
}
```

Wire it into the cdevsw:

```
.d_poll = evdemo_poll,
```

Call `selwakeup(&sc->sc_rsel)` from `evdemo_post_event` after the mutex is dropped.

Call `sel drain(&sc->sc_rsel)` and `knlist_destroy(&sc->sc_rsel.si_note)` during detach.

### Steps:

1. Copy the Lab 1 source to start from.
2. Apply the changes above.
3. Build: `make`.
4. Build the test program: `make test`.
5. Load: `sudo kldload ./evdemo.ko`.
6. Run the poll-based test: `sudo ./evdemo_test_poll`. It should call `poll()` with a 5-second timeout and print the result. With no events posted, `poll()` returns zero after the timeout.
7. Trigger an event while the test is running: `sudo sysctl dev.evdemo.trigger=1`. The `poll()` call should return immediately with `POLLIN` set, and the program should read the event.
8. Try `poll()` with several descriptors: the test program's extended mode opens `/dev/evdemo` twice and polls both descriptors. Trigger events and watch which one fires.

**What to observe:** `poll()` blocks until an event arrives, not until a timeout elapses, when an event is indeed triggered. The program does not spin on the CPU; it is genuinely asleep in the kernel. You can verify this with `top -H` and looking at `WCHAN`, which should show `select` or a similar wait channel.

**Common mistakes to check for:** If the poll returns immediately with `POLLIN` even when the queue is empty, check that your queue-emptiness check is correct. If the poll returns with the timeout even after you trigger events, the producer is not calling `selwakeup`, or it is calling `selwakeup` before updating the queue. If the kernel panics when you trigger an event, the `selinfo` was not properly initialized; check that `M_ZERO` was used in the softc allocation and that `knlist_init_mtx` was called.

**Takeaway:** `poll()` support is a hundred lines of extra code and gives every poll-based userland program the ability to integrate evdemo. The key is the lock discipline: the softc mutex serializes the check and the register in `d_poll` against the queue update in the producer. Without the lock, the race we analysed in Section 3 would cause occasional missed wakeups.

### 35.12.3 Lab 3: Adding kqueue Support

The third lab adds `d_kqfilter` so that programs using `kqueue(2)` can integrate evdemo. This lab teaches the filter operations structure and the `KNOTE_LOCKED` delivery pattern.

### Files:

- evdemo.c - driver source (extended from Lab 2)
- evdemo.h - shared header
- evdemo\_test\_kqueue.c - kqueue-based test program
- Makefile

### Changes in the driver from Lab 2:

Add the filter operations:

```
static int evdemo_kqread(struct knote *, long);
static void evdemo_kqdetach(struct knote *);

static const struct filterops evdemo_read_filterops = {
 .f_isfd = 1,
 .f_attach = NULL,
 .f_detach = evdemo_kqdetach,
 .f_event = evdemo_kqread,
};

static int
evdemo_kqread(struct knote *kn, long hint)
{
 struct evdemo_softc *sc = kn->kn_hook;

 mtx_assert(&sc->sc_mtx, MA_OWNED);
 kn->kn_data = sc->sc_nevents;
 if (sc->sc_detaching) {
 kn->kn_flags |= EV_EOF;
 return (1);
 }
 return (sc->sc_nevents > 0);
}

static void
evdemo_kqdetach(struct knote *kn)
{
 struct evdemo_softc *sc = kn->kn_hook;

 knlist_remove(&sc->sc_rsel.si_note, kn, 0);
}
```

Add the d\_kqfilter callback:

```
static int
evdemo_kqfilter(struct cdev *dev, struct knote *kn)
```

```

{
 struct evdemo_softc *sc = dev->si_drv1;

 switch (kn->kn_filter) {
 case EVFILT_READ:
 kn->kn_fop = &evdemo_read_filterops;
 kn->kn_hook = sc;
 knlist_add(&sc->sc_rsel.si_note, kn, 0);
 return (0);
 default:
 return (EINVAL);
 }
}

```

Wire it into the cdevsw:

```
.d_kqfilter = evdemo_kqfilter,
```

Add a `KNOTE_LOCKED(&sc->sc_rsel.si_note, 0)` call inside the producer's critical section. Between the `cv_broadcast` and the `mtx_unlock`.

Add `knlist_clear(&sc->sc_rsel.si_note, 0)` at the top of `detach`, before `sel drain`, to remove any still-attached knotes that did not get their `f_detach` called (for instance, because a kqueue was closed with the device's knote still attached).

#### Steps:

1. Copy the Lab 2 source.
2. Apply the changes above.
3. Build and load.
4. Run the kqueue-based test: `sudo ./evdemo_test_kqueue`. The program opens `/dev/evdemo`, creates a kqueue, registers `EVFILT_READ` for the device, and calls `kevent()` in blocking mode.
5. Trigger events and watch the kqueue reader print them.

**What to observe:** The kqueue reader reports events through the `kevent()` API rather than through `poll()`. It gets the `kn_data` value in `ev.data`, which tells it how many events are queued.

**Common mistakes to check for:** If the kqueue reader returns immediately with an error, `d_kqfilter` may be returning `EINVAL` because of a wrong case. Check the switch statement. If the kqueue reader hangs even after events are triggered, `KNOTE_LOCKED` is probably not being called, or is being called outside the lock. If the kernel panics on module unload with complaints about a non-empty knote list, `knlist_clear` is missing.

**Takeaway:** kqueue support is another hundred lines of code. The structure is similar to `poll`: a check in the event callback, a delivery in the producer, and a detach step. The framework handles the heavy lifting.

### 35.12.4 Lab 4: Adding SIGIO Support

The fourth lab adds asynchronous signal delivery. This lab teaches `FIOASYNC`, `fsetown`, and `pgsigio`.

**Files:**

- evdemo.c - driver source (extended from Lab 3)
- evdemo.h
- evdemo\_test\_sigio.c - SIGIO-based test program
- Makefile

**Changes in the driver from Lab 3:**

Add the async support to the softc:

```
bool sc_async;
struct sigio *sc_sigio;
```

Add the three ioctls to the ioctl handler:

```
case FIOASYNC:
 mtx_lock(&sc->sc_mtx);
 sc->sc_async = (*(int *)data != 0);
 mtx_unlock(&sc->sc_mtx);
 break;

case FIOSETOWN:
 error = fsetown(*(int *)data, &sc->sc_sigio);
 break;

case FIOGETOWN:
 *(int *)data = fgetown(&sc->sc_sigio);
 break;
```

Add pgsigio delivery to the producer, outside the lock:

```
if (async)
 pgsigio(&sc->sc_sigio, SIGIO, 0);
```

Add funsetown(&sc->sc\_sigio) to the close path and the detach path.

**Steps:**

1. Copy Lab 3.
2. Apply the changes above.
3. Build and load.
4. Run the SIGIO-based test: `sudo ./evdemo_test_sigio`. The program installs a SIGIO handler, calls FIOSETOWN with its PID, calls FIOASYNC to enable, and then pauses in a loop, draining the driver with non-blocking reads whenever the handler sets the flag.
5. Trigger events and watch the program print each one.



**What to observe:** Each event arrives via a signal, not through a blocking `read()` or a `poll()`. The signal handler itself does not read from the device; it sets a flag, and the main loop reads. This is the standard pattern for SIGIO handlers.

**Common mistakes to check for:** If the test program does not see any signals, `FIOASYNC` may not be enabling `sc_async`, or the producer is not checking `sc_async`. Also check that `fsetown` was called before the producer fires.

If the test program aborts with an error about SIGIO, the signal handler may not be installed, or the signal may be masked. Use `sigprocmask` or `sigaction` with `SA_RESTART` if you want system calls to be automatically restarted across signal delivery.

**Takeaway:** SIGIO is simpler than `poll` or `kqueue` from the driver's point of view: one `ioctl` handler, one call to `fsetown`, one call to `pgsigio`. The userland side is more complex because signals have inherently tricky semantics.

### 35.12.5 Lab 5: The Event Queue

The fifth lab focuses on the internal event queue itself. We reorganize the driver so that the queue is the single source of truth for all the asynchronous mechanisms, and we add `sysctl`-based introspection so we can watch queue behaviour at runtime.

#### Files:

- `evdemo.c` - driver source with polished queue implementation
- `evdemo.h` - shared header with the event record
- `evdemo_watch.c` - a diagnostic tool that prints queue metrics
- `Makefile`

#### What changes:

The queue functions become standalone and well-documented. Every operation takes the `softc` mutex, asserts it with `mtx_assert`, and uses a consistent naming convention.

A `sysctl` subtree under `dev.evdemo.stats` exposes queue length, total events posted, total events consumed, and total events dropped due to overflow.

A trigger `sysctl` allows the userland to post a synthetic event of a given type, which simplifies testing without having to write and load a custom test program.

A burst `sysctl` posts a batch of events all at once, which exercises the queue's overflow behaviour.

#### Steps:

1. Copy Lab 4.
2. Apply the queue polish: extract the enqueue/dequeue operations into clearly named helpers, add the counters, add the `sysctl` entries.
3. Build and load.
4. Run `sysctl dev.evdemo.stats` in a loop to watch queue state: `while ;; do sysctl dev.evdemo.stats; sleep 1; done`.
5. Trigger bursts: `sudo sysctl dev.evdemo.burst=100`. Watch the queue fill up, then drop overflow events when the queue is full.

6. Run any of the reader test programs (`poll`, `kqueue`, or `SIGIO`) while triggering bursts. Watch the reader drain the queue.

**What to observe:** The queue length reported in `sysctl` tracks the number of events that have been posted but not yet consumed. The dropped counter grows when events are posted while the queue is full. The posted and consumed counters diverge when the reader is slower than the producer, and converge when the reader catches up.

**Common mistakes to check for:** If the dropped counter grows without the overflow policy firing, the queue's fullness check is wrong. If the posted counter grows but the consumed counter does not, the producer is enqueueing but the reader is not dequeuing (which might be correct if no reader is running, but usually means a bug in the read path).

**Takeaway:** The event queue is what the three asynchronous mechanisms pivot around. With `sysctl` observability, we can watch the queue's behaviour directly and verify that it is doing what we expect under various loads.

### 35.12.6 Lab 6: The Combined v2.5-async Driver

The final lab is the consolidated `evdemo` driver, with all three asynchronous mechanisms, the audited locking discipline, the exposed metrics, and the clean detach path. This is the reference implementation that future drivers can be modeled on.

#### Files:

- `evdemo.c` - full reference driver
- `evdemo.h` - shared header
- `evdemo_test_poll.c` - `poll`-based test
- `evdemo_test_kqueue.c` - `kqueue`-based test
- `evdemo_test_sigio.c` - `SIGIO`-based test
- `evdemo_test_combined.c` - test that runs all three at once
- `Makefile`

#### What this lab demonstrates:

The combined test program forks three children. One uses `poll`, one uses `kqueue`, one uses `SIGIO`. Each child opens its own file descriptor to `/dev/evdemo` and watches for events. The parent triggers events at a known rate and reports after a fixed duration.

#### Steps:

1. Build and load.
2. Run the combined test: `sudo ./evdemo_test_combined`. It forks the three children, triggers 1000 events at a few hundred per second, and prints a summary at the end.
3. Observe that all three readers see all events.

**What to observe:** The posted counter in `sysctl` equals the sum of events seen across all three readers. None of the mechanisms drops events. The readers finish within a few milliseconds of each other, demonstrating that the driver is responsive to all three simultaneously.

**Common mistakes to check for:** If one reader is consistently behind, check that its mechanism's notification is being issued on every event. If the three readers produce different event counts, one mechanism is dropping notifications, which suggests a missed wakeup in the producer.

**Takeaway:** A driver that implements all three asynchronous mechanisms correctly serves any userland caller. This is the target to aim for when you build a production driver for an event-driven device. Once you know the pattern, the work is mechanical.

### 35.12.7 Lab 7: Unload Stress Test

The final lab is a stress test of the detach path, because detach is where the subtle bugs in asynchronous drivers tend to hide.

**Files:**

- `evdemo.c` from Lab 6
- `evdemo_stress.sh` - shell script that loads, exercises, and unloads the driver in a loop

**Steps:**

1. Load the driver.
2. In one terminal, run the combined test continuously in a loop.
3. In another terminal, run the stress script: `sudo ./evdemo_stress.sh 100`. This loads, exercises, unloads, and reloads the driver one hundred times in a row, exercising the attach and detach sequences under concurrent readers.
4. Observe that no panics occur, that all readers terminate cleanly across each unload-reload cycle, and that `sysctl` counters reset to zero on each attach.

**What to observe:** A driver with correct detach logic can sustain a hundred or a thousand load/unload cycles without panicking, leaking memory, or hanging. A driver with an incorrect detach will typically panic within ten or twenty cycles.

**Common mistakes to check for:** The most common detach bug is forgetting to drain in-flight callers before freeing the `softc`. `destroy_dev_drain` is the canonical tool for this; without it, an in-flight `read()` or `ioctl()` can touch a freed `softc`.

The second most common bug is a mismatch between the attach and detach initialization order. `knlist_init_mtx` must happen before the device is published, because a `kqfilter` call can arrive immediately after. Symmetrically, `knlist_destroy` must happen after the device is drained.

**Takeaway:** Stress-testing the unload path is the single most effective test for an asynchronous driver. If your driver survives 100 load/unload cycles under concurrent load, it is probably solid.

## 35.13 Challenge Exercises

These exercises are optional. They build on the labs to sharpen your skills in specific areas. Take your time; there is no rush.

### 35.13.1 Challenge 1: Dual Mechanism Shootout

Modify `evdemo_test_combined` to measure the per-event latency of each mechanism: the time between the producer's `evdemo_post_event` call and the userland reader's `read()` return. Use the `CLOCK_MONOTONIC` clock and record the timing on the event record itself.

Report a small table showing the mean, median, and 99th percentile latency for each of `POLL`, `KQUEUE`, and `SIGIO`. Try it with no contention (one reader per mechanism) and with contention (three readers per mechanism). Which mechanism has the lowest latency under no contention? Under contention?

The expected answer is that `kqueue` is lowest, `POLL` second, and `SIGIO` variable (because signal delivery latency depends on the reader's current execution state). But the details depend on your hardware, and the exercise is to measure rather than to predict.

### 35.13.2 Challenge 2: Multi-Reader Stress

Open twenty file descriptors to `/dev/evdemo` and poll all of them at once from a single thread using `kqueue`. Trigger 10000 events and verify that each event is delivered to all twenty descriptors exactly once.

This tests that the driver's knot list handles multiple knots correctly and that `KNOTE_LOCKED` walks the list completely on every event.

### 35.13.3 Challenge 3: Observe the Missed Wakeup Race

The third challenge asks you to deliberately break the driver so that you can observe a missed wakeup. Modify `evdemo_post_event` so that it updates the queue and calls the notifications outside the softc mutex instead of inside:

```
/* BROKEN: race with d_poll */
mtx_lock(&sc->sc_mtx);
evdemo_enqueue(sc, ev);
mtx_unlock(&sc->sc_mtx);
selwakeup(&sc->sc_rsel);
/* ... */
```

This unlocks the producer's enqueue from the consumer's check-and-register. With a high enough event rate and a busy consumer, you should occasionally see `POLL()` calls that return after a long delay despite events having been posted.

Try to reproduce the race. Time the `POLL()` calls. Report how often the race fires as a function of event rate. Then restore the correct locking and verify the race disappears.

The point of this exercise is not to write broken code. It is to see, with your own eyes, that the lock discipline we described in Section 3 is not a theoretical nicety but a real correctness property. Experiencing the race once is worth reading a hundred descriptions of it.

### 35.13.4 Challenge 4: Event Coalescing

Add an event coalescing feature to `evdemo`. When the producer posts an event of a type that matches the most recent event in the queue, merge them into a single event with an incremented counter instead of appending a new entry. This is similar to how some drivers coalesce interrupt events.

Test it with a burst of a hundred events of the same type. The queue length should stay at one. Now test it with a hundred events of alternating types: the queue should fill with alternating entries.

The challenge is as much about designing the userland contract as about implementing the feature. What does the reader see when coalescing happens? How does it know an event was coalesced? What does the `kqueue kn_data` field report when the queue has one entry but it represents many events?

There is no single right answer. Document your design choices in the source and be prepared to defend them.

### 35.13.5 Challenge 5: POLLHUP and POLLERR

Add graceful handling of `POLLHUP` and `POLLERR` to the driver. When the device is detached while a userland program still has it open, that program should see `POLLHUP` in its next `poll()` call (along with `POLLIN` if there are still queued events). When the driver has an internal error that prevents future operations, it should set an error flag and report `POLLERR` on subsequent `poll()` calls.

Test it by arranging for the driver to be detached while a reader is polling. The reader should wake up with `POLLHUP` and exit cleanly.

This teaches the full `poll()` contract and the subtleties of the `revents` bitmask. It also overlaps with the detach logic, which is the right place to set the HUP condition.

### 35.13.6 Challenge 6: evdev-Style Compatibility

Add a compatibility layer to `evdemo` that implements the `evdev` `ioctl` set, so that your driver becomes visible to existing `evdev`-aware userland programs. The key `ioctls` are `EVIIOCGVERSION`, `EVIIOCGID`, `EVIIOCGNAME`, and a few others documented in `/usr/src/sys/dev/evdev/input.h`.

This is a larger exercise and genuinely useful for understanding how real input devices expose themselves to userland. It requires reading the `evdev` source carefully and choosing a sensible subset to implement.

### 35.13.7 Challenge 7: Trace a kqueue Registration End to End

Using `dtrace(1)` or `ktrace(1)`, trace a single `kevent(2)` call that registers an `EVFILT_READ` on an `evdemo` file descriptor. Your trace should cover:

- The entry into the `kevent` syscall.
- The call into `kqueue_register` in the `kqueue` framework.
- The invocation of `fo_kqfilter` on the `cdev` fileops.
- The entry into `evdemo_kqfilter` (our driver's `d_kqfilter`).
- The `knlist_add` call.
- The return back through the framework to userland.

Capture the stack trace at each point. Then trigger a producer event in the driver and trace the delivery path:

- The `KNOTE_LOCKED` call in the producer.
- The entry into knote in the framework.
- The call to `evdemo_kqread` (our `f_event`).
- The queuing of the notification onto the kqueue.

Finally, the userland reaps the event with another `kevent` call. Trace that path as well:

- The second entry into the `kevent` syscall.
- The call into `kqueue_scan`.
- The walk of the queued knotes.
- The delivery to userland.

Submit your traces, annotated with a few sentences on what each part is doing. This exercise forces a direct confrontation with the kqueue framework source and is the surest way to go from “understands the callbacks” to “understands the framework.” A reader who completes this challenge will have the confidence to read any kqueue-using subsystem in the tree.

Tip: `dtrace -n 'fbt::kqueue_register:entry { stack(); }'` is a reasonable starting point. Build outwards from there, adding probes on `knote`, `knlist_add`, `knlist_remove`, and your driver’s entry points as you identify them in the source.

### 35.13.8 Challenge 8: Observe the knlist Lock Discipline

Write a small test program that opens an `evdemo` device twice from two different processes, registers an `EV_FILT_READ` on each, and then triggers a producer event. Use `dtrace` to measure how many times the `knlist` lock is acquired and released during the single delivery. Predict the number in advance based on what the chapter has taught about `KNOTE_LOCKED` and the `knlist` walk; then verify against the trace.

Next, modify `evdemo` so that the producer uses `KNOTE_UNLOCKED` instead of `KNOTE_LOCKED` (while adjusting the surrounding locking so the call is safe). Repeat the measurement. The number of acquisitions should change, and the change should match what the framework does differently in the two code paths.

Tip: `dtrace -n 'mutex_enter:entry /arg0 == (uintptr_t)&sc->sc_mtx/ { @ = count(); }'` will count mutex acquisitions on a specific mutex if you know its address. You can find the address through `kldstat -v` plus a bit of symbolic inspection.

## 35.14 Troubleshooting Common Mistakes

Asynchronous I/O bugs tend to fall into recognizable categories. This section collects the most common failure modes, their symptoms, and their usual causes, so that when you encounter one you can diagnose it quickly.

### 35.14.1 Symptom: `poll()` never returns

A `poll()` call blocks forever even though events are being triggered.

**Cause 1:** The producer is not calling `selwakeup`. Add a counter to `evdemo_post_event` and verify that it is actually incrementing when events are triggered.

**Cause 2:** The producer is calling `selwakeup` before the queue state is updated. Verify that `selwakeup` is called after `mtx_unlock`, not before.

**Cause 3:** The consumer's `d_poll` is not calling `selrecord` correctly. Check that the call is made under the softc mutex and that the `selinfo` passed is the same one the producer wakes.

**Cause 4:** The consumer is checking the wrong state. Verify that the queue-emptiness check in `d_poll` is looking at the same field that the producer updates.

### 35.14.2 Symptom: kqueue event fires but read() returns no data

A kqueue reader receives an `EVFILT_READ` event but a subsequent `read()` returns `EAGAIN` or zero bytes.

**Cause 1:** The queue was drained by another reader between the kqueue event delivery and the read. This is a benign symptom of multi-reader contention, not a bug. The reader should loop on `EAGAIN` and wait for the next event.

**Cause 2:** The `f_event` callback is returning true when the queue is actually empty. Check the `evdemo_kqread` logic.

**Cause 3:** The event was coalesced or re-filed after the kqueue delivery. Check for any queue manipulation that could remove the event after `KNOTE_LOCKED` was called.

### 35.14.3 Symptom: SIGIO is delivered but the handler is not called

The driver is calling `pgsigio`, but the userland program never sees the signal.

**Cause 1:** The program has not installed a handler for `SIGIO`. By default, `SIGIO` is ignored, not delivered.

**Cause 2:** The program has blocked `SIGIO` with `pthread_sigmask` or `sigprocmask`. Check the signal mask.

**Cause 3:** The program called `FIOSETOWN` with a wrong PID, so the signal is going to another process. Verify the argument is the current process's PID.

**Cause 4:** The driver is calling `pgsigio` only when `sc_async` is true, but the userland never enabled `FIOASYNC`. Check that the `ioctl` handler is updating `sc_async` correctly.

### 35.14.4 Symptom: Kernel panic on module unload

The kernel panics during `kldunload evdemo`.

**Cause 1:** `knlist_destroy` is being called on a `knlist` that still has knotes attached. Add `knlist_clear` before `knlist_destroy` to force-remove any remaining knotes.

**Cause 2:** `seldrain` is being called before in-flight callers have returned. Call `destroy_dev_drain` first, then `seldrain`.

**Cause 3:** The condition variable is being destroyed while a thread is still waiting on it. Set `sc_detaching = true` and `cv_broadcast` before `cv_destroy`.

**Cause 4:** The `softc` is being freed while another thread still holds a pointer to it. Ensure that the global `softc` pointer is cleared after `destroy_dev_drain` returns, not before.

### 35.14.5 Symptom: Memory leak on repeated load/unload

After many load/unload cycles, `vmstat -m` shows growing allocations for the driver's `MALLOC_DEFINE` type.

**Cause 1:** The `softc` is not being freed on detach. Check that `free(sc, M_EVDEMO)` is called.

**Cause 2:** `funsetown` is not being called. Each `fsetown` call allocates a `struct sigio` that must be freed.

**Cause 3:** Some internal allocation (for example, a per-reader structure) is not being freed at close time. Audit every allocation path and confirm every `malloc` has a matching `free`.

### 35.14.6 Symptom: Slow-to-wake `poll()` under load

A poll-based reader usually wakes quickly but occasionally takes significant time to see an event.

**Cause:** The scheduler's wakeup delivery latency on a busy system is in the millisecond range. This is not a driver bug; it is a general property of the kernel's scheduler.

If this latency is unacceptable for your use case, consider `kqueue` with `EV_CLEAR`, which has slightly lower wakeup overhead, or use a dedicated kernel thread for the consumer rather than a userland process.

### 35.14.7 Symptom: Events are dropped under load

The driver's dropped `sysctl` counter grows during a burst of events.

**Cause:** The queue is smaller than the burst size, and the overflow policy (drop-oldest) is kicking in.

This is working as designed for the default policy. If your application cannot tolerate drops, increase the queue size or change the overflow policy to block the producer.

### 35.14.8 Symptom: Only one reader wakes up even when multiple are waiting

Several readers are blocked in `read()` or `poll()`, but when an event is posted only one of them wakes.

**Cause:** The producer is calling `cv_signal` instead of `cv_broadcast`. `cv_signal` wakes exactly one sleeper; `cv_broadcast` wakes all of them.

For a driver with multiple concurrent readers, `cv_broadcast` is the correct choice, because each reader may race for the event and all of them need to see the wakeup to decide whether to re-sleep.

### 35.14.9 Symptom: The device hangs during detach

`kldunload` does not return, and the kernel shows the thread blocked somewhere in our detach code.

**Cause 1:** A call is blocked in `d_read` and we did not wake it before waiting for `destroy_dev_drain`. Set `sc_detaching`, `broadcast`, and `wake selinfo` before calling `destroy_dev_drain`.



**Cause 2:** A callout is in flight and we did not drain it. Call `callout_drain` before `destroy_dev_drain`, or the callout may re-enter the driver after we think we are finished.

**Cause 3:** A thread is parked in `cv_wait_sig` on a condition that will never be broadcast again. Ensure every wait loop checks `sc_detaching` as a separate exit condition.

### 35.14.10 Symptom: Reader wakes but finds nothing to do

A reader is woken by `poll`, `kqueue`, or a blocked read, but on returning to check the queue it finds the queue empty and has to go back to sleep. This happens occasionally even in a correct driver.

**Cause:** Spurious wakeups are a normal part of kernel life. The scheduler may deliver a wakeup that was meant for another waiter, a different event source sharing the same `selinfo` may have fired, or a race between the producer and another consumer may have drained the queue before this reader got a chance to look. None of these situations indicates a bug.

The correct response in the driver and in the reader is the same: always re-check the condition after waking, and treat a wakeup as a hint that something may have happened, not a guarantee that the specific event you expected is available. Every wait loop in the driver should look like the pattern we established in Section 3, with `cv_wait_sig` inside a `while` that checks the real condition. Every userland reader should expect to see `EAGAIN` or a zero-length read after a wakeup and loop back to poll again.

If wakeups without work are happening frequently enough to waste significant CPU, consider whether the producer is calling `selwakeup` more often than necessary, for example on every intermediate state change rather than only when a reader-visible event is ready. Coalescing wakeups at the producer is the fix; disabling the re-check loop in the consumer is not.

### 35.14.11 Symptom: Panic on module unload with “knlist not empty”

The module unload path panics with an assertion failure in `knlist_destroy` that reads something like “knlist not empty” or prints a non-zero count on the `knlist`’s list head.

**Cause 1:** `knlist_destroy` was called without a preceding `knlist_clear`. `knlist_destroy` asserts that the list is empty; live knotes on the list trigger the panic. Inspect the detach path and confirm that `knlist_clear(&sc->sc_rsel.si_note, 0)` runs before `knlist_destroy`.

**Cause 2:** A userland process still has a `kqueue` registration open and the driver tried to tear down without forcing the knotes off. The `knlist_clear` call is designed to handle exactly this case: it marks every remaining knote with `EV_EOF` | `EV_ONESHOT` so that the userland sees a final event and the registration dissolves. If the driver is skipping `knlist_clear` to “let the userland detach naturally,” the assertion fires. The fix is to call `knlist_clear` unconditionally in detach.

**Cause 3:** The detach path is being called while an event delivery is in progress. The `kqueue` framework uses its own internal locking to keep delivery and detach coherent, but a driver that tears down its `softc` while `f_event` is still running on another thread will corrupt the lifecycle. Ensure that all producer paths have stopped (for example, by setting a `sc_detaching` flag and draining any work queues) before entering the clear-drain-destroy sequence.

### 35.14.12 Symptom: Panic in `f_event` with a stale `kn_hook`

The kernel panics inside the driver's `f_event` function with a backtrace that shows a dereference of freed or garbage memory through `kn->kn_hook`.

**Cause 1:** The `softc` was freed before the `knlist` was torn down. The driver's detach path must clear and destroy the `knlist` before freeing the `softc`, in that order. Reversing the order leaves live knotes pointing at freed memory.

**Cause 2:** A per-client state object (for example, an `evdev_client`) was freed while a knote still references it. The cleanup logic for per-client state must run the `knlist_clear/seldrain/knlist_destroy` sequence on the client's `selinfo` before freeing the client struct, not after.

**Cause 3:** A different code path accidentally called `free()` on the `softc` or client state. Memory debuggers (KASAN on platforms that support it, or manually instrumented poison patterns on those that do not) will confirm that the memory is freed when `f_event` reads it. This is a general memory-corruption debugging exercise; the knote is the victim, not the cause.

### 35.14.13 Symptom: `KNOTE_LOCKED` panics with a lock-not-held assertion

A producer path that calls `KNOTE_LOCKED` panics with an assertion like “mutex not owned” inside the `knlist` lock check.

**Cause:** The producer is calling `KNOTE_LOCKED` without actually holding the `knlist`'s lock. `KNOTE_LOCKED` is the variant that tells the framework “skip locking, the caller has it”; if the caller does not, the framework's assertions catch it. The fix is either to take the lock (usually the `softc` mutex) around the `KNOTE_LOCKED` call, or to use `KNOTE_UNLOCKED` instead and let the framework take the lock itself.

Read the producer path carefully. A common mistake is to drop the `softc` lock partway through a producer function for some other reason (for example, to call a function that cannot be called under the lock), and then to forget to re-acquire it before the `KNOTE_LOCKED` call. The fix is to re-take the lock or to call `KNOTE_UNLOCKED` instead.

### 35.14.14 Symptom: `kqueue` events arrive but `kn_data` is always zero

A `kqueue` waiter wakes up and reads a struct `kevent` whose `data` field is zero, even though the driver has events pending.

**Cause 1:** The `f_event` function sets `kn->kn_data` only under certain conditions and leaves it untouched otherwise. The framework preserves whatever value was last written, so a stale zero from a previous invocation persists into the next delivery. The fix is to compute and assign `kn->kn_data` unconditionally at the top of `f_event`.

**Cause 2:** The `f_event` function is returning non-zero based on a condition other than the queue depth, and the `kn_data` field was not updated to reflect the actual count. Check that `kn_data` is assigned the real depth, not a boolean, and that the comparison that drives the return value is consistent with it.

### 35.14.15 Symptom: poll() works but kqueue never fires

A poll-based waiter sees events correctly, but a kqueue waiter on the same file descriptor never wakes up.

**Cause 1:** The driver's `d_kqfilter` entry point is not in the `cdevsw`. Check the `cdevsw` initializer and confirm that `.d_kqfilter = evdemo_kqfilter` is present. Without it, the kqueue framework has no way to register a knote on the descriptor.

**Cause 2:** The producer is calling `selwakeup` but not `KNOTE_LOCKED`. `selwakeup` does walk the knlist attached to the selinfo, but only under specific conditions; drivers that want to reliably wake kqueue waiters should call `KNOTE_LOCKED` (or `KNOTE_UNLOCKED`) explicitly in the producer path.

**Cause 3:** The `f_event` function always returns zero. Check whether the ready condition is being evaluated correctly. Add a `printf` to confirm that `f_event` is being called; if it is but returns zero, the bug is in the readiness check, not in the framework.

### 35.14.16 General Advice

When debugging an asynchronous driver, add counters liberally. Every `selrecord`, every `selwakeup`, every `KNOTE_LOCKED`, every `pgsigio` should have a counter. When behaviour looks wrong, printing the counters is the fastest way to tell you which mechanism is misbehaving.

Use `ktrace` on the userland side to see exactly when system calls return. If the driver thinks it delivered a wakeup at time *T* and the userland thinks it returned at time *T*+5 seconds, the wakeup was queued but not delivered, which often means a lock was held too long somewhere.

Use DTrace probes in the driver and on `selwakeup` itself. The `fbt:kernel:selwakeup:entry` probe shows every `selwakeup` system-wide. The `fbt:kernel:pgsigio:entry` probe does the same for signal deliveries. A missing call shows up as a gap in the probe output.

Do not suspect the framework. The kernel's async-I/O infrastructure is battle-tested and almost never has bugs at this level. Suspect your own driver first, particularly the lock ordering and the attach/detach sequence.

## 35.15 Wrapping Up

Asynchronous I/O is one of the places where a driver's correctness is most severely tested. A synchronous driver can hide many small locking mistakes behind a single-threaded flow that happens to run serially. An asynchronous driver exposes every corner of its locking discipline, every race between the producer and the consumer, and every subtle ordering constraint in the detach path. Getting an asynchronous driver right is harder than writing the synchronous version, but the rewards are significant: the driver serves many users at once, integrates cleanly with userland event loops, plays well with modern frameworks, and avoids the performance pathologies of blocking and busy waiting.

The mechanisms we have studied in this chapter are the classical ones. `poll()` and `select()` are portable across every UNIX system, and implementing them in a driver is a matter of one callback and a `selinfo`. `kqueue()` is the preferred mechanism for modern FreeBSD applications, and it adds one more callback and a set of filter operations. `SIGIO` is the oldest mechanism and has some sharp edges in multi-threaded code, but it remains useful for shell scripts and legacy programs.

Each mechanism has the same underlying shape: a waiter registers interest, a producer detects a condition, and the kernel delivers a notification to the waiter. The details differ, but the shape does not. Understanding the shape makes each specific mechanism easier to learn. The internal event queue we built in Section 6 is what ties the shape together: every mechanism expresses its condition in terms of queue state, and every producer updates the queue before notifying.

The locking discipline is the single habit that most consistently distinguishes a working asynchronous driver from a broken one. Take the softc mutex before checking state. Take it before updating state. Take it before registering a waiter. Take it before calling the in-lock notifications (`cv_broadcast`, `KNOTE_LOCKED`). Drop it before calling the out-of-lock notifications (`selwakeup`, `pgsigio`). This pattern is not an aesthetic choice; it is the pattern that prevents missed wakeups and deadlocks. When you see the pattern violated in a driver, ask why, because nine times out of ten the deviation is a bug.

The detach sequence is the second habit that deserves discipline. Set the detaching flag under the lock. Broadcast to wake every waiter. Deliver an `EV_EOF` to kqueue waiters. Call `selwakeup` to free poll waiters. Call `callout_drain` to stop the producer. Call `destroy_dev_drain` to wait for in-flight callers. Only after all of these can you safely `seldrain`, `knlist_destroy`, `funsetown`, `cv_destroy`, `mtx_destroy`, and free the softc. Skipping any step is a recipe for a panic at unload time, and those panics are especially painful to diagnose because they happen after the code you were testing.

The observability habit is the third. Every counter you add at development time saves hours of diagnosis when the driver is in production. Every `sysctl` entry you expose gives operators and debuggers a window into the driver's state without rebuilding the kernel. Every DTrace probe you declare lets a distant engineer with a production incident see into your code without shipping new software. Observability is not a luxury; it is a feature, and writing a driver without it is writing a driver you cannot debug.

You now have every piece of the async-I/O toolkit that a FreeBSD driver author needs for ordinary work. You can take a blocking character driver, audit its state transitions, identify the producer and the consumer paths, add `poll`, `kqueue`, and `SIGIO` support, and verify the whole thing under stress. The patterns generalize beyond character drivers: the same mechanisms apply to pseudo-devices, network devices with control channels, filesystems with file events, and any other subsystem that exposes an event stream to userland.

Two final notes before we move on.

First, asynchronous I/O is not a one-time lesson. You will find, as you read more of the FreeBSD source, that variations of these patterns appear everywhere: in network drivers using `grouptaskqueue`, in filesystems using `kqueue` for file events, in the audit subsystem using a ring buffer shared with userland. Each variation is an instance of the same underlying ideas. Being able to recognize the pattern when you see it is more valuable than memorizing any particular API.

Second, when you write your own driver, resist the temptation to invent your own asynchronous mechanism. The kernel's provided mechanisms cover virtually every use case, and userland programs know how to use them. A custom mechanism is work for you, work for your users, and work for whoever maintains the driver next. Reuse the standard patterns. They exist for a reason.

## 35.16 Bridge to Chapter 36: Creating Drivers Without Documentation

The next chapter changes the kind of challenge we face. Up to now, every chapter has assumed that the device we are writing for is documented. We knew its registers, its command set, its error codes, its timing requirements.

The book has shown how to turn that documentation into working kernel code, and how to test, debug, and optimize the result.

But not every device is documented. A driver author sometimes encounters hardware for which no datasheet is available, either because the vendor refuses to publish one, because the hardware is so old that the documentation has been lost, or because the device is a derivative of something documented but with undocumented changes. In those cases, the craft of driver writing shifts towards reverse engineering: observing the device's behaviour, deducing its interface, and producing a working driver from indirect evidence rather than from specifications.

Chapter 36 is about that craft. We will look at how experienced authors approach an undocumented device. We will study the tools for observing device behaviour, from bus analysers and protocol sniffers to the kernel's own built-in tracing facilities. We will learn how to build a register map by experiment, how to recognize common command patterns across vendors, and how to write a driver that is correct despite incomplete information about the hardware.

The asynchronous mechanisms from this chapter will reappear there, because event-driven hardware is exactly the kind of hardware that most rewards careful reverse engineering. A device whose documentation is missing still talks to the world in events, and making those events visible through `poll`, `kqueue`, and `SIGIO` is often the first step in figuring out what the device is actually doing.

The debugging skills from Chapter 34 will also matter, because an undocumented device produces many more surprising behaviours than a documented one, and `KASSERT`, `WITNESS`, and `DTrace` are the tools for catching those surprises early. The foundation we have built in Parts 2 through 7 is exactly what the reverse-engineering chapter needs.

If you have been reading this book from the beginning, take a moment to appreciate how far you have come. You started with an empty source tree and no knowledge of the kernel. You now know how to write a driver that supports synchronous and asynchronous I/O, handles concurrency correctly, observes its own behaviour through counters and `DTrace` probes, and can be debugged on a live system. You have written enough drivers by this point that the kernel is no longer an alien environment. It is a place you know how to work in.

The next chapter takes that knowledge and asks, what if the documentation for the device is missing? What does the same craft look like when you are working from evidence rather than from specifications? The answer, as it turns out, is that the craft changes less than you might think. The tools are the same, the disciplines are the same, and the habits you have built carry you most of the way.

Let us look at how that works.

## 36 Creating Drivers Without Documentation (Reverse Engineering)

### 36.1 Introduction

Up to this point in the book we have always written drivers for devices whose behaviour was at least partly documented. Sometimes the documentation was generous, with a full programmer's reference manual that named every register, defined every bit, and described every command. Sometimes it was sparse, with only a header file from the vendor and a short list of opcodes. Even in the sparse cases, however, we always had a starting point: a hint, a partial datasheet, an example from a related family of devices, or a manual page that told us what the device was supposed to do.

This chapter changes that assumption. Here we will learn how to write a driver for a device whose documentation is missing, lost, or deliberately withheld. The hardware exists. It works under some other operating system, or it once did, or someone has captured a few seconds of its behaviour with a logic analyser. But there is no register reference, no command list, no description of the data formats. Every fact we need we will have to discover ourselves.

If that sounds intimidating, take a slow breath. Reverse engineering a hardware device is a serious craft, but it is not magic. It is the same engineering discipline you have already practised throughout the book, applied in a slightly different direction. Instead of reading a datasheet and writing code that implements it, we will observe the device, form hypotheses about what it does, test those hypotheses with small experiments, and write the driver one verified fact at a time. The end result is the same kind of driver we have written in earlier chapters, with the same `cdevsw`, the same `device_method_t` table, the same `bus_space(9)` calls, and the same module build. The only difference is how we arrived at the contents.

Reverse engineering deserves a chapter for several reasons. The first is that the situation is more common than newcomers realise. Older hardware whose vendors have gone out of business; consumer devices that were never properly documented; embedded peripherals on a custom board where the manufacturer simply ships a binary blob and a datasheet expires when the contract does; specialised industrial, scientific, or medical equipment where the documentation lives on a CD that nobody can find. All of these are real situations that real FreeBSD developers run into, and several important FreeBSD drivers exist today only because someone was patient enough to do the work.

The second reason is that reverse engineering is a particularly disciplined kind of work, and the discipline is worth learning even if you never write a fully reverse-engineered driver. The habit of separating observation from hypothesis, the habit of writing down every assumption before testing it, and the habit of refusing to guess where a guess could damage hardware: these habits make ordinary driver work better, not just reverse-engineering work.

The third reason is that this work happens at a frontier where the distinction between writing software and doing experimental science becomes very thin. A reverse-engineering session looks more like a laboratory notebook

than like a programming session. You will run an experiment, observe a result, write down what you saw, propose an explanation, design the next experiment, and slowly build up a picture of an unknown system. That kind of work has its own rhythm, its own pacing, and its own small set of professional habits, and this chapter is where we will learn them.

We will begin by asking why reverse engineering is necessary at all, and where the legal and ethical lines lie. We will then build a small, safe, controlled lab where the work can happen without risk to production systems or expensive hardware. We will study the standard FreeBSD-friendly tools for observing devices, from `usbconfig(8)` and `pciconf(8)` to `usbdump(8)`, `dtrace(1)`, `ktr(9)`, and the `bus_space(9)` API itself. We will see how to capture a device's initialization sequence, how to compare runs to isolate single bits of meaning, and how to assemble a register map by experiment.

We will look at how to recognise the recurring shapes that hardware likes to use: ring buffers, command queues, descriptor rings, status-and-control register pairs, fixed-format command packets. We will write a minimal driver from scratch, one verified function at a time, starting with reset and slowly adding more behaviour. We will validate every assumption we can in a simulator before risking it on real hardware. We will study how the FreeBSD community already approaches this work, where to find prior art, and how to publish your findings in a form that helps others.

And finally, because this is the part that newcomers most often underestimate, we will spend a serious amount of time on safety. Some guesses can damage hardware. Some patterns of poking at unknown registers can erase non-volatile memory, brick a board, or leave a device in a state that nothing short of a vendor-only repair tool can recover from. The chapter will show you how to think about that risk, how to design wrappers that limit it, and how to recognise the operations that should never be performed without strong evidence that they are safe.

The companion code for this chapter, in `examples/part-07/ch36-reverse-engineering/`, includes a small collection of tools that you can build and run on a normal FreeBSD 14.3 development VM: a script that identifies and dumps device information for a target USB or PCI device, a kernel module that performs safe register probing on a memory region you point it at, a mock device that you can use to validate driver code before you have real hardware in front of you, and a Markdown template for the kind of pseudo-datasheet that should be the final product of a reverse-engineering session. Nothing in the labs touches hardware in a way that could damage it, and every example is safe to run inside a virtual machine.

By the end of this chapter, you will have a clear, repeatable method for approaching an undocumented device. You will know how to build the lab, how to observe, how to hypothesize, how to test, and how to document. You will know the legal lines that frame the work in the United States and the European Union, and you will know the professional habits that protect both you and the hardware. You will not finish this chapter as an expert reverse engineer, because that expertise is built up over years of practice, but you will know enough to start, and you will know how to keep yourself and the hardware safe while you learn.

## 36.2 Reader Guidance: How to Use This Chapter

This chapter sits in Part 7 of the book, in the Mastery Topics part, directly after Asynchronous I/O. It assumes you have read the asynchronous-I/O chapter, the advanced-debugging chapter, and the performance chapter, because the tools and habits from those chapters are the same tools and habits you will use here. If those chapters feel uncertain, a quick revisit will pay for itself several times over in this one.

You do not need any special hardware to follow the chapter. The worked examples use either a small kernel module that pokes at a memory region the operator gives it, or a mock device that simulates unknown hardware in software. Both run on a normal FreeBSD 14.3 virtual machine. If you have a real piece of unknown hardware that you would like to investigate, the chapter will give you the techniques and the safety habits to start, but the labs themselves will not require it.

The chapter is intentionally long because reverse engineering is a field where partial knowledge is dangerous. A reader who learns the glamorous parts of the craft and skips the safety parts is likely to brick something expensive. The labs and the safety sections deserve the same care as the technique sections.

A reasonable reading schedule looks like this. Read the first three sections in one sitting. They are the conceptual foundation: why reverse engineering matters, what the legal landscape looks like, and how to set up the lab. Take a break. Read Sections 4 through 6 in a second sitting. They cover the central techniques of the craft: building a register map, identifying buffers, and writing a minimal driver. Take another break. Read Sections 7 through 11 in a third sitting. They cover the disciplined work of expanding, validating, collaborating, staying safe, and documenting. Save the labs for a weekend or for several short evenings, because the labs will sink in much better if you have time to sit with the captured data and look at it carefully.

Some of the techniques in this chapter are slow on purpose. Capturing a device's initialisation sequence, for example, may require running the same boot ten times and diffing the captures to isolate the bits that change. The diffing is part of the lesson. If you find yourself wanting to skip to the part where the driver works, remember that the driver will only work if you have done the slow, careful observation work first. Reverse engineering rewards patience the way few other parts of systems programming do.

Several of the small kernel modules in the chapter are deliberately written as exploration scaffolds rather than as production drivers. They are commented as such. Do not load them on a production system. A development virtual machine where a kernel panic costs nothing more than a reboot is the right environment for this kind of work.

If you have hardware you would like to investigate after reading the chapter, take it slowly. Start with the safest observation tools. Resist the urge to write to anything until you have a clear, written-down hypothesis about what the write should do and what the worst case looks like. If a guess could trigger a flash erase, do not make the guess. The chapter will spell out which operations deserve special caution and why.

### 36.3 How to Get the Most Out of This Chapter

The chapter follows a pattern that mirrors the workflow of a real reverse-engineering session. Each section either teaches a technique that fits one phase of that workflow or shows you how the technique ties back to the underlying discipline. If you learn the workflow as a whole, the individual techniques will fall into place naturally.

Several habits will help you absorb the material.

Keep a notebook open beside the keyboard. A real notebook, not a text file, if you can manage it. Reverse engineering is a notetaking discipline, and the best practitioners keep written records of what they observe, what they hypothesize, what they tested, and what they learned. The act of writing slows your thinking down enough to catch mistakes, and a paper notebook resists the temptation to rearrange your record after the fact. If a paper notebook is not practical, use a Markdown file in a Git repository so that you can see how your understanding has changed over time.



Keep a terminal open to your FreeBSD development machine and another to `/usr/src/`. The chapter references several real FreeBSD source files, including drivers and utilities under `/usr/src/usr.sbin/` and `/usr/src/sys/dev/`. Reading those files is part of the lesson. The FreeBSD source is itself a body of reverse-engineering work, because many drivers in the tree exist because someone observed an undocumented device carefully enough to write code for it.

Type the example modules and scripts yourself the first time you see them. The companion files in `examples/part-07/ch36-reverse-engineering/` are there as a safety net and as a reference for when you want to compare your code against a known-good version, but typing the code the first time is the part that builds intuition. The whole chapter is about building intuition for unknown systems, and there is no shortcut for that.

Pay close attention to the language we use to describe what we know and what we suspect. Reverse-engineering writing draws a sharp line between an observation, a hypothesis, and a verified fact. An observation is what you saw. A hypothesis is what you think the observation means. A verified fact is a hypothesis that has survived deliberate attempts to falsify it. Different kinds of statement deserve different levels of confidence, and the chapter will model the language carefully so that you can adopt the same precision in your own notes.

Take seriously the safety advice. The most painful kind of mistake in this kind of work is the one that destroys the very piece of hardware you needed to study. Several of the techniques described here can, if applied carelessly, write to flash memory, change a device ID permanently, or leave a board in a state that the vendor cannot recover. The chapter will tell you which patterns deserve the most caution. Treat that advice the way you would treat a warning sign on a chemistry lab.

Finally, allow yourself to be slow. Reverse engineering is not a sprint. A complete register map for a serious peripheral is the result of weeks or months of patient observation, and the published results of community projects often hide an enormous amount of careful work behind a tidy summary. If a particular device resists your understanding for a long time, that is not a failure. It is what the work usually feels like.

With those habits in mind, let us begin with the question of why this work is necessary at all.

## 36.4 1. Why Reverse Engineering May Be Necessary

When a new driver author hears about reverse engineering for the first time, the immediate question is usually some version of “why should this ever be necessary?” If a hardware vendor wants their device to be useful, why would they hide its programming interface? And if the device is well known, why is there ever any documentation problem?

The honest answer is that the world of hardware and operating systems is messier than the world of well-documented standards. There are many real situations where a working device exists, where some operating system already supports it, but where no public, machine-readable, redistributable documentation exists for the programmer who wants to write a driver from scratch. The first section of this chapter walks through the most common of these situations so that you can recognise them when you encounter them and so that you know which kind of investigation each one calls for.

### 36.4.1 1.1 Legacy Hardware With No Vendor Support

The most common reverse-engineering situation is the old device whose manufacturer no longer exists. A small instrument board from the 1990s, a network card from a company that was bought out and shut down, an embedded controller from a research project that ran for a few years and then ended. The hardware works. The hardware was documented at the time it was sold. But the documentation was a paper manual that lived in a binder, or a PDF on a CD that was bundled with the device, and twenty years later neither the binder nor the CD exists.

In this situation, reverse engineering is the only way to bring the device back into useful service. Sometimes a community has already done some of the work and published partial notes. Sometimes a Linux or NetBSD driver exists and can be read for clues; the two cases are not equivalent, and the distinction matters both legally and technically. An OpenBSD or NetBSD driver is BSD-licensed and can be read, quoted, and, with attribution preserved, ported directly. A Linux driver is almost always GPL-licensed, which means it can be read for understanding but cannot be copied into a BSD-licensed driver at all. Even setting the licence aside, a direct port from Linux rarely works, because the Linux kernel's locking primitives, its memory allocators, and its device model differ from FreeBSD's in ways that reach deep into every line. We return to both the legal framing and the technical pitfalls of cross-platform reading in Section 12 and in Section 13's case studies. Sometimes the device is so obscure that no prior work exists at all, and the entire job falls on whoever wants to make it work.

FreeBSD has a long history of supporting devices in this category. A careful reading of the source under `/usr/src/sys/dev/` will turn up many drivers whose comments mention “no datasheet”, “based on observation”, or similar language. The driver authors did the work and the community benefits. This is not a marginal activity; it is part of how FreeBSD has always supported a long tail of devices that their original vendors abandoned long ago.

The challenges in the legacy-hardware case tend to be technical rather than legal. The hardware is old enough that the original vendor either no longer exists or no longer cares. The patents have expired. The trade secrets, if there were any, are no longer being defended. The risk is mostly that the documentation is genuinely gone, and no amount of polite asking will recover it.

### 36.4.2 1.2 Devices With Binary-Only Drivers on Other Operating Systems

A second common situation is the device whose vendor publishes a binary-only driver for one or more operating systems and refuses to publish documentation that would allow other operating systems to support the device. This is the case for many proprietary peripherals: graphics cards from certain vendors, specialised audio or video capture devices, scientific instruments with a Windows-only or Linux-only driver, fingerprint readers in laptops, certain wireless chipsets, and so on.

In this situation the device is in active production. Its documentation exists, but the vendor either treats it as a trade secret, or restricts it to companies that sign a non-disclosure agreement, or simply has not seen any business case for publishing it. The vendor's official position may be that FreeBSD users should not expect support, even if the underlying hardware would be perfectly capable of working under a properly written FreeBSD driver.

Reverse engineering in this situation is delicate, because the legal and ethical landscape is more complicated than it is for legacy hardware. The vendor may hold copyright on the driver binary. The firmware that runs on the device may itself be copyrighted. The distribution and use of the driver may be governed by an end-user license agreement that constrains certain kinds of analysis. We will return to the legal questions at the end of

this section. For now, note simply that the situation exists and that it is one of the recurring reasons why reverse engineering matters.

The technical challenges in this case are usually richer than in the legacy case, because there is more material to work with. You have a running driver that you can observe. You have a working device whose behaviour you can capture. You may have a firmware image you can analyse statically. The investigation can be very productive, but it also requires more careful attention to the legal context of the work.

### **36.4.3 1.3 Custom Embedded Systems With Little or No Documentation**

A third situation, increasingly common in industrial and embedded work, is the custom board with a custom chip. A small company designs an instrument or a controller for a specific application. They commission a custom integrated circuit, or they program a standard microcontroller with proprietary firmware, or they assemble a board from off-the-shelf parts in a configuration that has never been used elsewhere. They support the device only on their own operating environment, often a customised Linux distribution or a small real-time operating system.

When that company asks a contractor to integrate the device into a larger system that runs FreeBSD, or when an end user buys the hardware and wants to use it from FreeBSD, the only available information may be a one-page mechanical drawing, a short README file, and the binary firmware. No register map, no command set, no description of how the device starts up.

This case is in some ways the hardest, because the device is specific to one company and one customer. There is no community, because nobody else owns one. There is no prior driver to read, because nobody else has written one. The investigator is genuinely alone with the hardware, the captured traffic, and whatever can be deduced from observing the existing firmware. We will see in the labs how to approach this kind of investigation systematically, and we will see in Section 9 how to keep your own findings well enough documented that someone else can build on them later.

### **36.4.4 1.4 The Common Thread**

Every one of these situations shares a single underlying property: the device exists and works, but the description that would let a new driver be written from a specification is missing. The mechanical craft of writing the driver is the same craft we have practised in the rest of the book. What is different is the shape of the work that comes before the writing. We have to discover what we would ordinarily look up. That is the part this chapter teaches.

It is worth noticing what reverse engineering is not. It is not guessing. It is not poking randomly at registers and hoping something interesting happens. It is not trying to bypass copy protection or break encryption or do anything else that would shade into a different ethical universe. It is the patient, structured, written-down process of inferring how a piece of hardware works by observing what it does and what it produces, and then writing software that interacts with it correctly.

A reverse-engineering project, done well, looks far more like a laboratory science than like a hacker stereotype. There is a hypothesis, an experiment, a measurement, and a conclusion, repeated hundreds of times until enough conclusions have accumulated to write a driver that works. The romance of the scene where someone types furiously and a screen of text reveals “the secret” exists only in films. The reality is closer to a long, slow, methodical construction of understanding, one register and one bit at a time.

### 36.4.5 1.5 Legal and Ethical Considerations

Before we touch any tools, we have to talk about the legal landscape that surrounds this work. The picture is not complicated, but it is real, and a beginner who ignores it can stumble into trouble that nothing in the technique sections of this chapter will help with.

Reverse engineering for the purpose of interoperability, which is the purpose this chapter cares about, is broadly accepted in both United States and European Union law. The point of interoperability work is to allow a device, format, or interface to be used with a piece of software that the original vendor did not provide. Writing a FreeBSD driver for a USB Wi-Fi adapter that ships with a Windows driver is a textbook interoperability case. The driver lets FreeBSD talk to a piece of hardware that the user already owns. It does not copy the vendor's driver. It does not redistribute the vendor's firmware in a form that violates a license. It does not bypass any security measure that protects copyrighted content. It produces an independent piece of software that performs the same function the vendor's driver performs, written from a clean understanding of the underlying interface.

In the United States, the relevant legal framework is the doctrine of fair use under copyright law, with a long line of court cases recognising reverse engineering for interoperability as a legitimate fair use. The Sega versus Accolade case from 1992, the Sony versus Connectix case from 2000, and several similar precedents have established that disassembling code to understand its interface is fair use, provided the purpose is legitimate interoperability and the resulting product does not contain the original copyrighted code. The relevant statute that occasionally complicates matters is the Digital Millennium Copyright Act, which forbids circumventing “technological measures” that protect copyrighted works. The DMCA includes specific exemptions for interoperability research, but those exemptions are narrower than the underlying fair use right. For ordinary driver work the DMCA is rarely a problem, but if you ever need to defeat encryption to read firmware, the legal picture becomes more complex and a real lawyer becomes a worthwhile consultation.

In the European Union, the relevant framework is the Software Directive, originally Directive 91/250 and updated in 2009 as Directive 2009/24. Article 6 of the Software Directive explicitly permits decompilation for the purpose of achieving interoperability with an independently created program, provided several conditions are met: the person doing the decompilation has a right to use the program, the information needed for interoperability has not been readily available, and the decompilation is limited to the parts of the program necessary for interoperability. This is one of the most explicit legal endorsements of interoperability reverse engineering in any major jurisdiction.

Outside those two systems, the picture varies. Many countries follow similar principles in practice. A few do not. If you are working in a jurisdiction where the law is unclear, or where enforcement is unpredictable, talk to a lawyer who actually knows your local software-copyright law. The cost of a single hour of legal advice is small compared to the cost of finding out the hard way.

There is a clear ethical line between interoperability work and work that would harm the vendor or the user. Interoperability work produces a new program that lets a piece of hardware do its intended job under a new operating system. It does not redistribute copyrighted code. It does not strip license restrictions from a purchased product. It does not bypass security measures that protect something other than the vendor's commercial interest in the interface itself. If you find yourself wanting to do any of those things, you are no longer doing interoperability work, and the rest of the chapter does not apply to your situation.

A second ethical line is between observation and tampering. Watching how a device behaves is observation. Capturing the traffic between the device and the vendor's driver is observation. Reading the firmware that the vendor distributed in a form intended to be distributable is observation. Writing your own driver based on what

you observed is the legitimate end product. Writing modified firmware that replaces the vendor's firmware on the device, or distributing such modified firmware, is a different category of work that brings in different legal and ethical considerations. We will not cover that activity in this chapter. The chapter is about writing a clean, original driver that talks to the original hardware in its original configuration.

A third ethical consideration is honesty about your work. Document where your information came from. If a particular register definition came from reading the vendor's open-source driver, say so. If a packet format came from a published specification, cite it. If a behaviour was deduced from your own captures, describe the captures. This honesty is partly a legal matter, because it lets you prove that you did clean-room work, and partly a community matter, because it lets others build on your work without having to redo the parts you have already done.

### 36.4.6 1.6 Wrapping Up Section 1

This section has set the stage for the rest of the chapter. We have seen the three most common situations where reverse engineering is necessary: legacy hardware, devices with vendor-only support on other operating systems, and custom embedded systems with no documentation. We have seen that the underlying property is the same in every case: a working device whose programming interface is not described. And we have seen the legal framework that surrounds this work in the United States and the European Union, with a clear distinction between legitimate interoperability work and other, more legally fraught activities that this chapter will not pursue.

The next section turns from why to how. We will set up the small lab where the reverse-engineering work will happen, and we will introduce the tools you will need. The lab is the foundation for everything that follows, and a few hours spent setting it up properly will save many hours of confusion later.

## 36.5 2. Preparing for the Reverse Engineering Process

A reverse-engineering session is, at its core, an experimental activity. You will run experiments on a piece of hardware, capture the results, analyse them, and design follow-up experiments. Like any experimental activity, it benefits from a properly equipped laboratory. The lab does not need to be expensive. Most of what we need is software that is already in the FreeBSD base system, and the rest can be assembled from a short list of free or low-cost tools. The investment is mostly in setting up the equipment correctly so that your captures are reliable and your experiments are repeatable.

This section walks through the complete kit. It starts with the mental model of what the lab is doing, then enumerates the software tools, then discusses optional hardware tools, then describes how to bring up a vendor driver under another operating system on the same machine so that you can observe its behaviour, and finally suggests a workflow for keeping the lab organised.

### 36.5.1 2.1 The Mental Model: What the Lab Is For

Before listing tools, it helps to picture what the finished lab will look like. The lab is the place where you will:

1. **Identify** the device, recording every public fact about it.
2. **Observe** the device in known operational states.

3. **Capture** the traffic between the device and an existing driver.
4. **Experiment** with reads and writes to discover register behaviour.
5. **Document** every observation as it is made.
6. **Validate** every hypothesis by experiment, ideally with a simulator before risking the real device.

The lab is therefore a small system designed to support the observe-hypothesize-test-document loop that drives the entire craft. Every tool you add to it should serve that loop in some identifiable way. Tools that look impressive but do not feed into the loop are clutter, and clutter slows you down.

A first-time reader sometimes assumes that reverse engineering requires very expensive professional equipment. It does not. A modest FreeBSD development machine, a target system that can run the vendor's driver, and a small budget for cables and adapters will get you most of the way through most projects. The expensive tools are nice to have, and we will mention them, but the core work is done with software you already have access to.

### 36.5.2 2.2 The Software Tool Kit From the FreeBSD Base System

FreeBSD's base system includes a remarkable collection of tools that together cover most of what a driver author needs for the software side of the lab. Let us go through them in the order in which a reverse-engineering session typically uses them.

**pciconf(8)** is the starting point for any PCI or PCI Express device. It is an interface to the `pci(4)` ioctl that the kernel exposes through `/dev/pci`. Run as root, it lists every PCI device the kernel has enumerated, including devices for which no driver attached. The most important invocations for reverse engineering are:

```
$ sudo pciconf -lv
```

This produces a one-line summary of every PCI device on the system, followed by the human-readable vendor and device strings if a known database recognises them. Devices without a driver appear as `noneN@pci0:...`. For each unknown device, this line tells you the PCI bus address, the vendor identifier, the device identifier, the subsystem identifiers, the class code, and the revision. Those identifiers are the first piece of public information you will record about the device.

```
$ sudo pciconf -lvc
```

The `-c` flag adds a list of the device's PCI capabilities, such as MSI, MSI-X, power management, vendor-specific capabilities, and PCI Express link-training data. Repeated `-c -c` increases the verbosity for some capability types. For PCI Express devices, this is also where you will see link-state information that tells you whether the device negotiated the link width and speed it should have negotiated. A surprising number of "this device does not work" problems turn out to be link-training problems that this one command would have revealed.

```
$ sudo pciconf -r device addr:addr2
```

The `-r` form reads PCI configuration register values directly, returning the raw bytes at an offset in configuration space. This is the safest way to inspect a device, because configuration space is designed to be read without

side effects. The complementary `-w` form writes configuration registers; use it with extreme care, because some configuration registers do change device behaviour permanently.

**devinfo(8)** prints the FreeBSD device tree as the kernel sees it. Where `pciconf` shows you the bus level, `devinfo` shows you the full hierarchy: which bus the device hangs off, which parent controller, what resources it has been assigned, and what name the kernel has given it. The verbose form `devinfo -rv` is particularly useful early on, because it shows the exact ranges of memory and I/O ports allocated to the device, and those ranges are the playground in which all the bus-space experiments will happen.

**devctl(8)** is the device-control utility that lets you detach a driver from a device, attach a different driver, list events, and disable specific devices at the kernel level. During reverse engineering, the most useful invocations are `devctl detach deviceN` to remove the in-tree driver from a device and `devctl attach deviceN` to put it back. Detaching a driver is sometimes necessary so that your experimental driver can claim the device, and being able to put the in-tree driver back without rebooting saves a lot of time.

**usbconfig(8)** is the USB equivalent of `pciconf`. It enumerates USB devices, dumps their descriptors, and changes their state. Its most important invocations for reverse engineering are:

```
$ sudo usbconfig
$ sudo usbconfig -d ugen0.3 dump_device_desc
$ sudo usbconfig -d ugen0.3 dump_curr_config_desc
$ sudo usbconfig -d ugen0.3 dump_all_config_desc
```

The first form lists every USB device the system sees, with its unit number and address. The `dump_device_desc` form prints the USB device descriptor: `bDeviceClass`, `bDeviceSubClass`, `bDeviceProtocol`, `bMaxPacketSize0`, `idVendor`, `idProduct`, `bcdDevice`, the manufacturer and product strings, and the number of configurations. The `dump_curr_config_desc` and `dump_all_config_desc` forms walk the configuration descriptors and print the interfaces, alternate settings, and endpoints contained within them. Together, those three commands give you a nearly complete static picture of what the USB device claims to be, and that picture is the starting point of every USB investigation.

**usbdump(8)** is the FreeBSD equivalent of Linux's `usbmon`. It captures USB packets by opening `/dev/bpf` and binding to a cloned `usbN` interface created by the `usbpf` packet-filter module, and it writes the captured packets to a file in a format compatible with `libpcap`. Captures can be saved with `-w file`, replayed with `-r file`, and filtered with the standard BPF expression language. For reverse engineering, the workflow is usually:

```
$ sudo usbdump -i usb0 -w session1.pcap
```

This captures everything on the named USB bus to a file. After the capture, the file can be read back with `usbdump -r session1.pcap`, opened in Wireshark, or processed with custom scripts. The captured file format records every USB transfer, including `SETUP` packets, `IN` and `OUT` data, status responses, and timing information. Multiple sessions captured during different operations can be diffed against each other to isolate the packets responsible for one specific behaviour, and that diffing is one of the most effective techniques in the kit.

**dtrace(1)** is the dynamic tracing facility that we have used in earlier chapters. For reverse engineering, `DTrace` is particularly useful for tracing the points where the kernel interacts with a driver: which `device_probe` is

being called for which device, which interrupt handlers fire when, which `bus_space` operations the in-tree driver performs. A few well-chosen DTrace probes can tell you in detail what the existing driver is doing, even when no other documentation exists.

**ktr(9)** is the in-kernel trace facility used for fine-grained event tracking. It is more intrusive than DTrace, because it requires kernel options at build time, but it gives you a high-resolution log of every traced event. For reverse engineering, `ktr` is most useful when added to your own experimental driver so that the timing of register accesses can be reconstructed exactly.

**vmstat -i** and **procstat**: smaller utilities that let you watch the rate of interrupts a device is generating and the processes that are interacting with the device. Both are part of the base system. During an experiment, it can be useful to watch the interrupt count change as you exercise the device, because a sudden change in interrupt rate is itself a significant observation.

**hexdump(1)**, **xxd(1)**, **od(1)**, **bsdiff(1)**, and **sdiff(1)**: ordinary userland utilities for examining binary data and comparing files. You will use them constantly. A capture file viewed in `xxd` is often where the patterns first become visible, because the eye can pick out repeating structures in a hex dump that no automated tool would notice without being told what to look for. `sdiff` of two `xxd` outputs is one of the oldest and most reliable ways to find what is different between two captures.

This list is not exhaustive, but it covers the tools you will reach for in the first several weeks of any project. Every one of them is in the FreeBSD base system, with manual pages under `man pciconf`, `man usbconfig`, and so on.

Beyond the base system, a small family of third-party disassemblers and decompilers becomes important when the only artefact you have is a vendor binary, typically a Windows driver binary, a firmware image extracted from a device, or an option ROM pulled from a PCI card. **Ghidra**, the open-source reverse-engineering suite released by the United States National Security Agency, is the tool most FreeBSD developers reach for first because it is free, cross-platform, and comfortably decompiles x86, ARM, and many embedded architectures into readable C-like pseudocode. **radare2** and its graphical companion **Cutter** are a lighter open-source alternative, well suited to quick inspection of small firmware blobs. **IDA Pro** is the long-established commercial product; its decompiler is still the reference implementation in the industry, but its cost places it outside the budget of most individual developers. You do not need any of these tools to do excellent work on devices whose behaviour can be reconstructed from captures alone. When you do need them, the goal is always limited and documented: identify the names of registers, identify the structure of command buffers, understand the order in which the vendor's code programmes the hardware. You do not copy the vendor's code. You write down what the binary implied about the hardware's interface and then you discard the disassembly. This is the clean-room discipline outlined in section 1.5 translated into practice: the binary is a source of hardware facts, not a source of code to be copied. Use the disassembler briefly, write down what you learned, and build your driver from the notes.

### 36.5.3 2.3 Optional Hardware Tools

Hardware tools become valuable when the device's interaction with the host is not visible to software-only capture. A USB device's traffic passes through the host controller and can be captured by `usbdump`. A PCI Express device's transactions go through the root complex and are not captured by anything in the base system. An SPI peripheral on an embedded board may communicate with its host processor on wires that no operating-system tool can see. For those cases, hardware tools enter the picture.



A **logic analyser** is the most generally useful hardware tool. It attaches to wires and records the voltage on each wire over time, producing a digital trace that can be decoded into the protocol that the wires were carrying. For SPI, I2C, UART, and similar low-speed busses, a basic eight or sixteen channel logic analyser is sufficient. The Saleae Logic family is widely used in professional work and is well supported by sigrok, the open-source suite for analysing logic-analyser captures. Sigrok's pulseview GUI lets you import a capture, decode it as SPI or I2C, and step through the bus traffic byte by byte.

A **USB protocol analyser** is a specialised piece of hardware that sits on the USB bus and captures every packet, including bus-state events that are not visible from the host. The Beagle and Total Phase analysers are the high-end tools in this category. They are expensive, but for serious USB reverse engineering they reveal behaviour that software-side capture simply cannot see. Most hobby-level work, however, does fine with usbdump and a careful methodology.

A **PCI Express protocol analyser** is even more specialised, and practically nothing in the open-source world covers this niche. For PCI Express work, the usual fall-back is to capture the device's behaviour from the kernel side using DTrace, ktr(9), and your own instrumented driver, and to use pciconf -lvc to inspect the configuration-space state. Real PCIe analysers from companies like Teledyne and Keysight exist, but their cost places them outside the reach of most individual developers.

For embedded work, an **oscilloscope** is sometimes useful for diagnosing electrical problems that confuse the higher-level tools. A driver that times out for unknown reasons may be timing out because the device's clock signal is degraded; an oscilloscope will show this when no software tool will. A modest USB-connected oscilloscope is a reasonable investment for anyone doing serious embedded work.

You can do excellent reverse-engineering work with none of these hardware tools. Most consumer USB and PCI devices are accessible purely through software-side capture and the kernel's own introspection. The hardware tools become important when the investigation crosses into truly low-level territory: signal integrity, bus timing, embedded protocols, devices designed for industries that never expected to be opened up.

### 36.5.4 2.4 The Observation Rig

Once you have the tools, the next decision is the shape of the observation rig. The rig is the combination of machines and operating systems on which you will observe the existing driver running. There are several common configurations, each with its own advantages.

The simplest configuration is **one host, two operating systems**. The same physical machine boots either FreeBSD, where you will be writing the new driver, or another operating system whose driver already supports the device, where you will be observing it. Boot into the other OS to observe; boot into FreeBSD to develop. The configuration is straightforward and works well when the device is permanently attached to the host. The disadvantage is that you cannot observe and experiment in the same session, so iteration is slower.

A more flexible configuration is **two hosts**: one running the operating system whose driver supports the device, the other running FreeBSD as your development environment. The device can be attached to one host, observed, and then moved to the other. Captures, notes, and code travel between the two over the network. This configuration works well when both machines fit on a desk and when the device can be moved without harm.

For USB devices, a **single FreeBSD host running another OS in a virtual machine** is often the most efficient configuration. The device is attached to the FreeBSD host, where usbdump can capture its traffic. The virtual machine is configured to receive the device through USB passthrough, so that the vendor's driver inside the

VM sees the device. As the vendor's driver operates the device, `usbdump` on the host records every packet. This setup gives you both observation and rapid iteration in a single session, because you do not have to reboot anything to switch between observing and experimenting.

For PCI devices, the equivalent is **bhyve passthrough using the ppt(4) driver**. The FreeBSD host detaches the device from the in-tree driver, attaches it to `ppt(4)`, and exposes it to a bhyve guest where the vendor's driver can run. The vendor's driver in the guest operates the device, while the host can use DTrace and other tools to observe what passes between them. Bhyve passthrough is a valuable technique for PCI work and has the great advantage of keeping all your observation tools in a single FreeBSD host.

For very specialised hardware, **dedicated capture hardware** is the only option. A logic analyser permanently attached to a board's SPI bus, or a USB protocol analyser inserted between a USB device and its host, gives you observation that no software-side tool can provide. The trade-off is that the rig is more complex to set up and the captured data is in a format that requires its own tools to analyse.

### 36.5.5 2.5 Bringing Up the Vendor Driver Under Another OS

If you are going to observe the vendor's driver in operation, you need a working installation of an operating system that the vendor supports. The choice depends on what the vendor ships. For Linux drivers, a recent stable Linux distribution is usually the right choice. For Windows drivers, the standard procedure is a Windows installation in a virtual machine; this works well as long as the device can be passed through to the guest. For specialised embedded operating systems, the situation is more variable.

Whichever OS you choose, set it up with as little extra software as possible. The lab should be quiet. Background activity from other drivers, automatic updates, telemetry, or unrelated processes adds noise to your captures. A lean install lets the device's traffic stand out clearly.

For a USB device whose vendor supplies a Linux driver, the recommended configuration is:

1. Install a recent stable Linux distribution in a virtual machine on your FreeBSD host.
2. Configure bhyve to pass through the USB device to the guest.
3. Inside the guest, install the vendor's driver and verify that the device works.
4. On the host, attach `usbdump` to the USB bus through which the device communicates.
5. Repeat the device's operations in the guest while capturing on the host.

For a USB device whose vendor supplies a Windows driver, the configuration is similar, with Windows in the guest instead of Linux. Windows USB passthrough through bhyve has been improving steadily and is workable for most consumer devices.

For a PCI device, the analogous configuration uses bhyve PCI passthrough through the `ppt(4)` driver. Detach the device from its host driver with `devctl detach`, attach it to `ppt(4)` with the appropriate kernel configuration, and expose it to the bhyve guest with `bhyve -s slot,passthru,bus/slot/function`. The vendor's driver in the guest then operates the device. Software-side observation from the host is harder for PCI than for USB, because configuration-space and memory-space accesses are not visible to the host once the device has been passed through. The fall-back is to instrument your own experimental driver heavily and to learn from the differences between what your driver does and what the vendor's driver appears to be doing.

### 36.5.6 2.6 The Lab Notebook

Equally important to the tools is the discipline of recording what you do. A lab notebook, paper or digital, is not optional in reverse engineering. Without one, you will lose track of what you have tested, what you have observed, and what you have concluded. With one, you build up an artifact that, by itself, is part of the output of the project.

The notebook should record:

- The date and time of every observation session.
- The exact configuration of the lab when the observation was taken: kernel version, tool versions, the state of the device before the observation began.
- The exact commands you ran.
- The captured data, or a clear pointer to where the captured data is stored.
- Your immediate interpretation of what you observed, with the word “observation” or “hypothesis” clearly attached.
- Any decision you made about what to test next, and why.

A good notebook entry reads like a scientific protocol. It should be reproducible by someone else who has access to the same lab, and it should tell a future reader what you knew at the time and how you knew it. When the project is finished and you write the pseudo-datasheet that summarises everything you learned, the notebook is where the citations come from. When something later turns out to be wrong, the notebook is where you go to figure out when the wrong belief entered the picture and what other conclusions might be infected by it.

The discipline of writing down hypotheses before testing them is particularly important. Without that discipline, it is very easy to convince yourself, after the fact, that you predicted a result you did not actually predict. With it, you can tell precisely which experiments confirmed your understanding and which ones surprised you. The surprises are the most valuable observations, because they are the places where your model is wrong, but they only show up clearly when the prediction was written down before the result was known.

### 36.5.7 2.7 A Sample Lab Layout

By way of concrete example, here is a configuration that has worked well for many reverse-engineering projects on FreeBSD.

The host is a small FreeBSD 14.3 desktop with at least 16 GB of memory. It runs the development version of the driver, the observation tools, and the bhyve hypervisor. Its `/usr/src/` is populated with the FreeBSD source so that source-level browsing is fast.

A virtual machine inside bhyve runs a recent Linux distribution. The VM has the vendor’s driver installed and is configured for USB or PCI passthrough as appropriate.

A separate Git repository, on the host, holds the project notebook, the captured pcap files, the experimental driver code, and the pseudo-datasheet as it grows. Each commit is dated and described, so that the history of the project’s understanding is preserved.

A second terminal on the host always has `tail -F /var/log/messages` running, so that any kernel message produced by the experimental driver is immediately visible.

This layout is not the only one that works, but it is a reasonable starting point. The key features are: a clean FreeBSD development environment, a way to run the vendor's driver, a way to observe the device, and a Git-tracked notebook that grows with the project.

## 36.5.8 2.8 Wrapping Up Section 2

We now have a tool kit and a lab. The base system gives us `pciconf`, `usbconfig`, `usbdump`, `devinfo`, `devctl`, `dtrace`, and `ktr`. Optional hardware tools give us deeper visibility when software-side capture is not enough. `Bhyve` and `ppt(4)` give us a way to run the vendor's driver inside a virtual machine while observing from the FreeBSD host. And a written lab notebook gives us the discipline that turns ad-hoc experimentation into repeatable engineering.

The next section puts the lab to work. We will see how to capture a device's behaviour systematically, how to recognise the patterns that most hardware uses to communicate with its driver, and how to turn raw captures into the first signs of an emerging mental model. We will be doing experimental work in earnest, and the discipline that this section established becomes essential as we start producing the data that everything else will be built on.

## 36.6 3. Observing Device Behavior in a Controlled Environment

With the lab in place, we now turn to observation. This is the phase where you collect the raw data that everything else will be built on. The goal is not to understand the device yet. The goal is to capture, with as much fidelity as possible, what the device does in a small set of well-defined situations. Understanding will emerge later, from analysis. The first job is to get clean, labelled captures.

This section walks through the standard observation techniques in the order in which a project usually applies them. We start with static descriptors, which give you a snapshot of the device's identity. We move to initialisation captures, which show you what the device does when it is first powered up or attached. We then look at functional captures, which show what the device does when it performs each of its useful operations. Throughout, we emphasise the discipline of structured capture: each capture is named, dated, labelled with the operation it represents, and stored alongside a short note describing what the user did during the capture and what behaviour was expected.

### 36.6.1 3.1 Static Descriptors and Identity Information

The first thing to capture about any device is its identity. For a PCI or PCI Express device, this means recording the output of:

```
$ sudo pciconf -lv
```

For the specific device, the relevant lines look like this in practice. Suppose the device is the third unattached PCI device the kernel sees. After running `pciconf -lv`, you might see:

```
none2@pci0:0:18:0: class=0x028000 card=0x12341234 chip=0xabcd5678 \
 rev=0x01 hdr=0x00
 vendor = 'ExampleCorp'
 device = 'XYZ Wireless Adapter'
 class = network
 subclass = network
```

This single line records six facts that any future analysis will need: the bus location (0:18:0), the class code (0x028000, which the FreeBSD class-code table identifies as a wireless network controller), the subsystem identifier (0x12341234), the chip identifier (0xabcd5678, with vendor ID 0xabcd and device ID 0x5678), the revision (0x01), and the header type (0x00, a standard endpoint). Every one of those facts may matter later. The vendor and device IDs are how the kernel will find your driver. The subsystem ID is sometimes the only way to distinguish two devices that share a chip but use different layouts. The class code tells you what category the device belongs to. The revision distinguishes silicon revisions that may behave differently. Record all of them.

Add the capability list:

```
$ sudo pciconf -lvc none2@pci0:0:18:0
```

This will append a list of PCI capabilities, each a single line with a name, an ID, and a position in configuration space. The typical list for a modern PCI Express device includes power management, MSI or MSI-X, PCI Express, vendor-specific capabilities, and one or more extended capabilities. The vendor-specific capabilities are particularly interesting, because they are the place where vendors hide non-standard functionality and they are often the entry point for chip-specific configuration.

For a USB device, the equivalent capture is:

```
$ sudo usbconfig
$ sudo usbconfig -d ugen0.5 dump_device_desc
$ sudo usbconfig -d ugen0.5 dump_curr_config_desc
```

These three commands together produce the device descriptor, the current configuration descriptor, and a list of all configurations. A typical output for a simple USB device looks like:

```
ugen0.5: <ExampleCorp Foo Device> at usb0
 bLength = 0x0012
 bDescriptorType = 0x0001
 bcdUSB = 0x0210
 bDeviceClass = 0x00
 bDeviceSubClass = 0x00
 bDeviceProtocol = 0x00
 bMaxPacketSize0 = 0x0040
 idVendor = 0x1234
```

```
idProduct = 0x5678
bcdDevice = 0x0102
iManufacturer = 0x0001 <ExampleCorp>
iProduct = 0x0002 <Foo Device>
iSerialNumber = 0x0003 <ABC123>
bNumConfigurations = 0x0001
```

Each field is an answer to a question. The `bcdUSB` value tells you the USB protocol version the device claims to implement. `bDeviceClass`, `bDeviceSubClass`, and `bDeviceProtocol` are the USB class system, which sometimes identifies the device as a member of a standard class (HID, mass storage, audio, video, and so on) and sometimes leaves all three at zero, meaning that the class is determined per-interface inside the configuration descriptor. `idVendor` and `idProduct` are the unique numerical identifiers; combined, they are how a USB driver is bound to the device. `iManufacturer`, `iProduct`, and `iSerialNumber` are indices into the device's string table; `usbconfig` resolves them for you and prints the strings.

Dump the configuration descriptor too:

```
$ sudo usbconfig -d ugen0.5 dump_curr_config_desc
```

This shows the interface descriptors and endpoint descriptors. For each interface, you will see its number, its alternate setting, its class, subclass, and protocol, and the list of endpoints. For each endpoint, you will see its address (which encodes both the endpoint number and the direction), its attributes (which encode the transfer type: control, isochronous, bulk, or interrupt), its maximum packet size, and its polling interval if it is an interrupt endpoint.

This static information is the entire programming-visible identity of the USB device. It tells you exactly what kinds of pipes the device exposes, in which directions, of what type, and at what size. From this alone you can already make some educated guesses. A device that exposes a single bulk-IN and a single bulk-OUT endpoint is probably a transport for some application-specific protocol. A device that exposes two interrupt-IN endpoints is probably an event source of some kind. A device with isochronous endpoints is almost certainly handling time-sensitive data such as audio or video.

Save these dumps in your notebook. They are the device's static identity, and they will not change between captures. They are the header of every report you will write about the device.

### 36.6.2 3.2 The First Capture: Initialization

Once you have the static identity, the next capture is the initialisation sequence. This is the sequence of operations that the vendor's driver performs between the moment the device is attached and the moment the device is ready for use.

The initialisation sequence is one of the most informative things you can capture, because it usually exercises every register the device has. The driver writes initial values, sets configuration options, allocates buffers, sets up interrupts, enables data flow, and reports success. Almost every register the device exposes will be touched at least once during this sequence, and many of the registers will reveal their general purpose just by being touched in a way that fits the standard initialisation pattern.

For a USB device, the initialisation capture is:

```
$ sudo usbdump -i usb0 -w init.pcap
```

Start usbdump, then attach the device. After the device is fully initialised, stop the capture. The resulting pcap file contains every USB transfer that passed across the bus between the device and its driver, beginning with the USB enumeration sequence (which should match the static descriptor dumps), continuing with the class-specific or vendor-specific control transfers that the driver uses to configure the device, and ending when the driver has put the device into a ready state.

Open the file in Wireshark to view it interactively, or process it with `usbdump -r init.pcap` to print it textually. Wireshark has particularly good USB dissectors that will decode many standard class-specific transfers automatically. For dissection of vendor-specific transfers, you will be reading the raw bytes yourself.

For a PCI device, software-side capture of initialisation is harder, because configuration-space and memory-space writes are not visible from outside the device once you are not using passthrough. The usual technique is to instrument your own experimental driver, or to add tracing to a copy of the in-tree driver if one exists. We will return to this subject in Section 4 when we talk about register maps. For now, the equivalent of a “first capture” for a PCI device is the output of:

```
$ sudo devinfo -rv
```

restricted to your device. This shows you the resources that the kernel allocated for the device: the memory ranges, the I/O port ranges, the interrupt assignment. Those resources tell you the extent of the playground you will be exploring. They do not tell you what is happening inside the playground, but they are the boundary conditions for everything that follows.

### 36.6.3 3.3 Functional Captures

Once you have the initialisation, the next captures are functional. For each thing the device can do, you take a separate capture. Each capture should isolate one operation as cleanly as possible.

For a network device, you might take separate captures for “send one ping”, “receive one packet of unsolicited traffic”, “set the MAC address”, “change the channel”. For a sensor, you might take separate captures for “read once”, “set the sample rate”, “enable continuous mode”, “calibrate”. For a printer, you might capture “print one page of plain text”, “print one image”, “query status”.

The discipline of isolating operations is not optional. If your capture file contains a hundred different operations, sorting out which packets correspond to which operation will be impossible. If your capture file contains exactly one operation, the packets in it are exactly the packets for that operation, and your job is much easier.

Every capture should include:

1. The exact operation being captured, in plain English.
2. The exact user actions that triggered the operation.
3. The exact moment the user started the action, recorded as a timestamp in the file name or in a sidecar note.

4. The expected behaviour of the device.
5. The actual behaviour observed.

The sidecar note is essential. Six months from now, you will not remember which capture corresponded to which operation, and the file name alone will not always be enough.

A sample naming scheme that has worked well for many projects is:

```
init-001-attach-cold.pcap
init-002-attach-hot.pcap
op-001-set-mac-address-aa-bb-cc-dd-ee-ff.pcap
op-002-send-icmp-echo-request.pcap
op-003-receive-broadcast-arp.pcap
err-001-attach-with-no-power.pcap
```

The `init-`, `op-`, and `err-` prefixes group captures by purpose. The numbered suffix is unique. The English description in the file name is enough that you can find a specific capture without opening it. The sidecar note lives next to the file as a `.txt` or `.md` file with the same base name.

### 36.6.4 3.4 The Diff: Isolating Bits of Meaning

The single most important analysis technique in reverse engineering is the diff. Two captures of similar operations are likely to be mostly identical, with a few differences that correspond to the differences in what was done. Those differences are the parts that matter, because they are the parts whose meaning is most likely to be visible.

Suppose you have a capture of “set channel to 1” and a capture of “set channel to 6”. Visually, the two captures will look almost identical. They will start with the same setup, perform the same preliminary operations, and end with the same wrap-up. Somewhere in the middle, however, there will be a small number of bytes that differ between the two captures. Those bytes are almost certainly related to the channel number. By comparing them carefully, you can deduce: which transfer carries the channel value, where in that transfer the value lives, what numerical encoding is used (direct numerical value, an index into a table, a bitfield), and whether the value is sent as little-endian or big-endian.

The diff technique works best when the two captures differ in exactly one variable. If you compare “set channel to 1” against “set channel to 6”, and the channel-to-channel difference is the only difference between the captures, the diff is clean. If you compare “set channel to 1, mode A” against “set channel to 6, mode B”, you have two variables changing and the analysis is harder. Make captures that vary one thing at a time.

For text-format captures, `sdiff` is the simplest tool. For binary captures, `bsdiff` produces compact patches that can be inspected to see exactly which bytes changed. For pcap files, the combination of `tshark -r file.pcap -T fields -e ...` to extract specific fields, followed by `diff`, gives you a programmable way to compare specific aspects of the captures.

A more sophisticated technique is to capture multiple instances of the same operation, and compare them. Differences between captures of nominally identical operations reveal which bytes are actually constant (the parts of the protocol) and which bytes change between runs even though the operation does not (sequence numbers,



timestamps, random values). The constant bytes are the ones whose meaning you want to deduce; the variable bytes are the ones whose meaning you can ignore for the moment.

Save every capture. Storage is cheap and the diffing technique becomes more useful the more captures you have. A project with fifty captures of an operation can answer many more questions than a project with one capture, even though both projects “captured the operation”.

### 36.6.5 3.5 Wireshark and the USB Dissector

For USB work specifically, Wireshark is an indispensable tool. Wireshark dissects the USB packet stream into a structured tree view that is much easier to read than raw bytes, and it has display filters that let you isolate one device, one endpoint, or one direction of traffic.

The most useful filters are:

- `usb.device_address == 5` to limit the view to one specific device on the bus.
- `usb.endpoint_address == 0x81` to limit the view to one specific endpoint and direction (here, IN endpoint 1).
- `usb.transfer_type == 2` to limit the view to bulk transfers (1 = isochronous, 2 = bulk, 3 = interrupt).
- `usb.bRequest == 0xa0` to limit the view to a specific control request, useful when reverse engineering vendor-specific control transfers.

Combinations of these filters let you isolate exactly the part of the capture you care about. The “Statistics” menu in Wireshark also provides useful aggregate views, such as a list of every endpoint seen and the number of packets on each. For USB, the “Endpoints” view in particular is often the first thing you check after opening a new capture.

If you have a capture that Wireshark dissects into something class-specific (for example, a USB Mass Storage capture decoded into SCSI commands), the dissector has effectively done the hardest work for you. If you have a capture that Wireshark dissects only as raw bulk transfers, you will need to read the bytes yourself.

### 36.6.6 3.6 Observation Patterns to Recognise

Even before you understand a specific device, certain patterns recur in almost every device’s protocol, and learning to recognise them speeds up every project. Watch for these as you look at captures.

**Repeated writes followed by a single read.** This is the classic “write a command, then read the result” pattern. The repeated writes are usually setting up a request: command code, parameters, length. The read is fetching the response. Many devices use this shape for any operation that returns data.

**Status flags that change before and after events.** A bit somewhere in a status register that becomes set when the device finishes work, or becomes clear when work begins, is one of the most common ways for a device to communicate progress. Look for bits that change reliably in time with operations the user triggered.

**A sequence of writes to escalating addresses, all in multiples of four.** This is often a register block being written in sequence. The operation is reset followed by configuration: clear all the registers, then set them to their new values, then trigger the operation by writing to a “go” register at the end.

**Identical reads from the same address until a value changes.** This is polling. The driver is waiting for the device to do something, and is checking a status register repeatedly. The address being polled is almost certainly a status register. The value that ends the polling tells you which bit in that register is the “ready” indicator.

**Bulk-OUT followed by bulk-IN of the same length.** A common pattern for command-response on USB bulk endpoints is to send a fixed-size command on the OUT endpoint and then read a fixed-size response from the IN endpoint. The two endpoints work together as a request-response channel.

**Periodic interrupt-IN packets with timestamps that increase linearly.** This is a heartbeat or periodic-status pattern. The device is reporting its state at a fixed rate, regardless of what the host does. The packets often contain a small fixed structure with status bits and counters.

**Long sequences of writes followed by no observable response.** This is often firmware download. The device has a writable code region, and the driver is loading new instructions into it. Such captures are typically very large compared to other operations, and they often start with a fixed-size header that identifies the firmware image.

These patterns are not exhaustive, but they appear so often that recognising them on first sight saves enormous time. The first hour with a new capture file is almost always spent identifying which of these patterns the capture exhibits.

### 36.6.7 3.7 Wrapping Up Section 3

We have built a body of captures. Each capture is named, dated, labelled with the operation it represents, and accompanied by a short note describing the user actions and the expected device behaviour. We have used `pciconf` and `usbconfig` to record the static identity of the device. We have used `usbdump` to capture its initialisation and its functional operations. We have learned that the `diff` between two captures of similar operations is the best tool we have for extracting bit-level meaning. And we have learned the recurring patterns that show up in almost every device’s protocol.

The next section starts the active phase of the work. Instead of just observing the existing driver, we will start poking at the device ourselves, in a controlled way, to discover the meaning of its registers. The captures we made in this section are the data that the next section’s experiments will be measured against. With captures in hand, we know what “normal behaviour” looks like, and we can compare what happens when we issue our own writes against what happens when the vendor’s driver issues the equivalent writes. That comparison is the heart of register-map construction.

## 36.7 4. Creating a Hardware Register Map by Experimentation

The register map is the document that, once finished, would have made all the previous work unnecessary. It lists every address the device exposes, says what is at that address, defines the meaning of each bit, and describes any side effects that reads or writes have. With a complete register map in hand, writing the driver is a straightforward translation exercise. Without one, the driver cannot be written at all. The register map is the artifact that the rest of this chapter is, in many ways, designed to produce.

In the absence of documentation, the register map has to be built by experiment. You will write to addresses, see what happens, read addresses back, see what they return, change one bit at a time, look for changes in behaviour, and slowly accumulate a body of verified facts about each address. The work is patient and incremental, and

many of the techniques are simple enough to fit in a paragraph, but the discipline of doing them safely and recording the results carefully is what separates a successful project from a session that destroys the device.

This section covers the techniques. Section 10 will return to the safety side and spell out what you must not do. Read the two together; the techniques in this section are only safe in the hands of someone who has internalised the warnings in Section 10.

### 36.7.1 4.1 Mapping the Address Space

Before you can probe addresses, you have to know which addresses exist. For a PCI device, the device's BARs (Base Address Registers) declare the memory regions and I/O port ranges that the device responds to. The kernel has already enumerated these and made them available through `bus_alloc_resource_any(9)` calls in your driver's `attach` routine. The simplest way to see them in operation is to read them back from the kernel:

```
$ sudo devinfo -rv
```

Restricted to your device, this command lists the resources the kernel has assigned. A typical PCI device might have output like:

```
none2@pci0:0:18:0:
 pnpinfo vendor=0xabcd device=0x5678 ...
 Memory range:
 0xf7c00000-0xf7c0ffff (BAR 0, 64K, prefetch=no)
 0xf7800000-0xf7bfffff (BAR 2, 4M, prefetch=yes)
 Interrupt:
 irq 19
```

Two memory regions and one interrupt. The 64K region is most likely a register block, because register blocks are usually small. The 4M region is large enough to be a frame buffer, a descriptor ring, or a memory-mapped data area, but unlikely to be register space. These are educated guesses based on size, not yet verified facts. Note them as hypotheses.

The same kind of size-based intuition works in many cases. A register region is rarely larger than a megabyte. A data buffer or queue is rarely smaller than a few kilobytes. A region that is exactly 16 KB or 64 KB on power-of-two boundaries is suspicious in the right way: it looks like register space. A region that is several megabytes and prefetchable is much more likely to be a data region.

For a USB device, the equivalent of address-space mapping is the endpoint enumeration. Each endpoint is a “channel” that you can read from or write to. The endpoint addresses, sizes, and types were captured by `usbconfig dump_curr_config_desc` in the previous section. From the endpoint list you already know how many channels the device exposes, of which types, and in which directions. There is no equivalent of memory-mapped register space on USB; everything is done through the endpoints, including any “register” reads and writes (which appear as control transfers, vendor-specific requests, or data inside bulk transfers).

### 36.7.2 4.2 The Read-First Principle

The single most important rule for safe register exploration is to read before you write. A read from an unknown register is, almost always, harmless. The hardware returns whatever it considers to be the value at that address, with at most a side effect of clearing some specific status flags in some specific kinds of registers. A write to an unknown register, on the other hand, can do anything: trigger a reset, start an operation, change a configuration bit, write to flash memory, or in the worst case put the device into a state from which it cannot easily recover.

The principle is simple: assume nothing about a write until you have evidence that the write is safe. The evidence can come from the captures you made in Section 3 (a write that the vendor driver performs is presumably one of the writes the device expects), from analogous registers on similar devices, from a published header file or a related driver, or from your own analysis of the register's behaviour under reads.

Read the entire register region exhaustively before doing any writing. Save the values. Read it again ten minutes later. Compare. Registers whose values change between reads are interesting: they are either status registers reflecting some live state, or counters incrementing on their own, or registers connected to external inputs. Registers whose values are stable are either configuration registers (whose values are stable until something writes them) or data registers that happen not to have changed during the observation window.

The simplest tool for this kind of exploration is a small kernel module that takes a memory range and dumps it, and the companion code in `examples/part-07/ch36-reverse-engineering/lab03-register-map/` contains exactly such a module. The module attaches as a child of the bus you point it at, allocates the memory range as a resource, and exposes a sysctl that, when read, dumps every word in the range. Multiple reads at different times produce a picture of which words are stable and which are changing.

### 36.7.3 4.3 The Probing Module's Skeleton

For concreteness, here is the shape of a safe probing module. The full version is in the companion files; what follows is the essential structure that you should recognise from earlier chapters.

```
struct regprobe_softc {
 device_t sc_dev;
 struct mtx sc_mtx;
 struct resource *sc_mem;
 int sc_rid;
 bus_size_t sc_size;
};

static int
regprobe_probe(device_t dev)
{
 /* Match nothing automatically; only attach when explicitly
 * told to. The user adds the device by hand with devctl(8) so
 * that the wrong device cannot be probed by accident. */
 return (BUS_PROBE_NOWILDCARD);
}
```

```

static int
regprobe_attach(device_t dev)
{
 struct regprobe_softc *sc = device_get_softc(dev);

 sc->sc_dev = dev;
 sc->sc_rid = 0;
 mtx_init(&sc->sc_mtx, "regprobe", NULL, MTX_DEF);

 sc->sc_mem = bus_alloc_resource_any(dev, SYS_RES_MEMORY,
 &sc->sc_rid, RF_ACTIVE);
 if (sc->sc_mem == NULL) {
 device_printf(dev, "could not allocate memory resource\n");
 return (ENXIO);
 }
 sc->sc_size = rman_get_size(sc->sc_mem);
 device_printf(dev, "mapped %ju bytes\n",
 (uintmax_t)sc->sc_size);
 return (0);
}

```

The BUS\_PROBE\_NOWILDCARD return from probe is the protective gesture: this driver will not attach to any device unless the operator points it at one explicitly. This prevents the dangerous case where a probe module accidentally attaches to a device it was not meant to investigate.

The reading function, called from a sysctl handler, looks like this:

```

static int
regprobe_dump_sysctl(SYSCTL_HANDLER_ARGS)
{
 struct regprobe_softc *sc = arg1;
 char buf[16 * 1024];
 char *p = buf;
 bus_size_t off;
 uint32_t val;
 int error;

 if (sc->sc_size > sizeof(buf) - 64)
 return (E2BIG);

 mtx_lock(&sc->sc_mtx);
 for (off = 0; off < sc->sc_size; off += 4) {
 val = bus_read_4(sc->sc_mem, off);
 p += snprintf(p, sizeof(buf) - (p - buf),

```

```
 "%04jx: 0x%08x\n", (uintmax_t)off, val);
 }
 mtx_unlock(&sc->sc_mtx);

 error = sysctl_handle_string(oidp, buf, p - buf, req);
 return (error);
}
```

`bus_read_4` is a wrapper around `bus_space_read_4` that uses the resource as both tag and handle, and it is the simplest way to read four bytes from a memory-mapped region. Notice that we read in units of four bytes, on four-byte aligned offsets. Most modern devices expect register accesses to be aligned and of natural size. Unaligned accesses, or accesses of the wrong size, can produce garbage values or, on some hardware, cause bus errors that manifest as kernel panics. When the device might use 16-bit or 8-bit registers, use `bus_read_2` or `bus_read_1` respectively; when in doubt, start with 4-byte reads, because that is the most commonly correct choice for modern peripherals.

This module does no writes at all. It is purely an observation tool. Loading it on an unknown device gives you a snapshot of the device's memory region without changing anything. Multiple loads, or multiple invocations of the `dump sysctl`, give you a sequence of snapshots whose differences reveal which addresses are dynamic.

### 36.7.4 4.4 Inferring Register Purpose From Behavior

After you have read the address space repeatedly, you start to notice patterns. Each pattern suggests a purpose for the register.

**Stable values** are usually configuration registers, identifier registers, or default values that have not yet been changed. A register at offset 0 that always reads the same constant value is often a chip-identifier register, sometimes called a “version register” or “magic number register”. Many devices use such a register specifically so that drivers can confirm the chip is what they expect.

**Slowly changing values** are usually counters: bytes received, packets sent, errors detected, time elapsed. The slowly increasing register is a hallmark of a counter, and the difference between two readings tells you the count rate. If a counter increments by exactly one every second, you have probably found a heartbeat or a timestamp register. If it increments by a few thousand per second when traffic flows and stays flat when traffic does not, you have found a packet or byte counter for the live data path.

**Quickly changing values that wrap around** are usually production-side ring-buffer write pointers. A device that produces data into a ring buffer typically exposes a register holding the current write position. The write pointer increments rapidly while data is flowing, and wraps when it reaches the buffer's size. The matching read pointer, written by the driver, tells the device where the consumer is.

**Bits that toggle in time with operations** are status bits. A bit that becomes 1 when an operation begins and 0 when it completes is a “busy” bit. A bit that becomes 1 when an event occurs and stays 1 until cleared is a “pending interrupt” bit. A bit that becomes 0 when the device is healthy and 1 when an error has occurred is an error indicator.

**Registers that read back differently from what was written** are either masked (some bits are not implemented and read as zero regardless of writes), one-shot (writes have side effects but the register itself does not store the

written value), or self-clearing (writes set bits that the device clears automatically when the operation completes). All three variants are common, and the readback pattern is what tells them apart.

These observations build up a picture of the register map. Each verified observation goes into the notebook, with the address, the observed pattern, and the proposed purpose. Over time, the notebook entries solidify into the pseudo-datasheet that Section 11 will discuss.

### 36.7.5 4.5 Comparing Captured Writes Against Hypotheses

The captures from Section 3 are the bridge between observation and hypothesis. Once you have proposed that a particular register is, for instance, the channel-selection register, you can confirm or refute the hypothesis by comparing what you have seen the vendor driver write to that register against what you know about the operations the user performed.

Suppose your hypothesis is that the register at offset `0x40` is the channel-selection register, and that the value at that register encodes the channel number directly as a small integer. You have a capture in which the user selected channel 6, and a capture in which the user selected channel 11. If your hypothesis is correct, the capture-6 sequence should contain a write of `0x06` to the equivalent of `0x40`, and the capture-11 sequence should contain a write of `0x0b` to the same place. If you find those writes, the hypothesis is supported. If the captures show writes of different values, or no writes at all to that offset, the hypothesis is wrong.

This is the falsification step that makes the work scientific. A hypothesis that survives one falsification attempt is worth more than ten hypotheses that have only been informally proposed.

For a USB device, the same kind of comparison happens between the captured control transfers, bulk packets, or interrupt packets and the hypothesised meaning. If you believe a vendor-specific control transfer with `bRequest = 0xa0` and `wValue` in the low byte is the “set channel” command, you can verify it by examining the captures of channel-1, channel-6, and channel-11 selections and confirming that exactly that transfer appears in each, with the expected `wValue`.

### 36.7.6 4.6 The Discipline of One Hypothesis at a Time

One discipline that is easy to forget under pressure is the discipline of testing one hypothesis at a time. When several hypotheses are in the air, the temptation is to design an experiment that will test all of them at once. The experiment runs, the device produces some output, and the output is consistent with one of the hypotheses but not with another. Now you have learned something about hypothesis A. But because you also changed the conditions for hypothesis B, you have also lost the ability to interpret what happened from B’s perspective. The same experiment will need to be run again, with B varied independently.

The right discipline is: vary one thing, observe the result, draw the conclusion that follows for that one variable, then proceed to the next experiment. It is slower per experiment but faster overall, because every experiment leaves behind a clear, isolated fact rather than a tangle of partial information.

Newton’s laboratory notebooks are full of single-variable experiments. So are the notebooks of every successful reverse-engineering project. Multivariable experiments belong to a later phase, when you understand each variable well enough that you can predict their interactions; in the discovery phase, they mostly produce confusion.

### 36.7.7 4.7 Common Register-Layout Patterns

Across many devices, a few register-layout patterns recur often enough to be worth knowing in advance. Recognising them can save weeks of work.

**Status / Control / Data:** a small group of registers in which one register holds the current state of an operation (Status), one register accepts commands or configuration changes (Control), and one or more registers hold the data being moved into or out of the device (Data). Many simple peripherals use exactly this shape, and many more use it as the shape of one of several functional units within the device.

**Interrupt Enable / Interrupt Status / Interrupt Acknowledge:** three registers that together implement the interrupt mechanism. The Enable register controls which conditions cause an interrupt. The Status register reports which conditions are currently raising an interrupt. The Acknowledge register clears interrupt conditions once they have been handled. The pattern is so common that, when exploring an unknown device, you should look for it explicitly: three adjacent registers near the top of the register map, with identifying names like `INT_ENABLE`, `INT_STATUS`, `INT_CLEAR` if you can find any documentation.

**Producer Pointer / Consumer Pointer:** two registers that together implement a ring buffer. The Producer Pointer is updated by whoever is producing data (sometimes the device, sometimes the driver) and indicates the next free slot. The Consumer Pointer is updated by whoever is consuming the data and indicates the next slot to be processed. The difference between them tells you how many slots are currently full. This pattern is the basis of nearly every high-speed I/O interface in modern hardware, including most network controllers, most USB controllers, and most disk controllers.

**Window / Index / Data:** an indirect-access pattern where the device exposes a small number of registers but uses them to access a much larger internal address space. A Window register selects which “page” is currently visible, an Index register selects which register within the page, and a Data register reads or writes the selected register. The pattern is common in older devices and in devices with very large internal register spaces.

**Capability Pointer:** a register or small group of registers that points to the start of a chain of capability descriptors. PCI Express defines a standard form of this pattern; many other devices use ad-hoc variants. The capability list lets the driver discover what optional features the chip supports without having to consult external documentation.

When you are exploring an unknown device, scanning the register space for any of these recurring patterns is one of the first things to do. A device that uses a Status / Control / Data layout is much easier to drive than one that does not, and recognising the layout early saves substantial time.

### 36.7.8 4.8 What Not to Do Yet

Before moving on, a quick reminder of restraint. The temptation, when you have built a mostly-working register map, is to try writing things and see what happens. Resist this temptation in register exploration. Write only when you have a specific hypothesis to test, when the hypothesis predicts a specific outcome, when you have a way to verify that the outcome occurred, and when you have considered what the worst plausible outcome of an unintended interpretation would be. The discipline that Section 10 will spell out is built on top of this restraint, and the register-mapping phase is exactly where it must be applied first.



### 36.7.9 4.9 Wrapping Up Section 4

We have learned how to map the address space of a device, how to read it safely, how to interpret the patterns of stable, slowly changing, and quickly changing values, and how to compare captured writes against hypotheses to confirm or refute proposed register meanings. We have learned the recurring layouts that hardware likes to use. And we have set the stage for Section 10's treatment of safety, which is the precondition for all of this work being done responsibly.

The next section moves up one level of abstraction. Once we have identified individual registers, we look at how they group into larger structures: data buffers, command queues, descriptor rings. These higher-level structures are how devices actually move data, and identifying them is the next step in turning register knowledge into the basis of a working driver.

## 36.8 5. Identifying Data Buffers and Command Interfaces

Registers are the way you talk to a device's control surface. Data buffers are how the device moves real information. The register map you built in the previous section is necessary but not sufficient. To complete the picture you also need to understand how the device handles data: where its buffers live, how their boundaries are defined, what shape the data takes inside them, and how the producer and consumer coordinate.

This section walks through the common buffer and command structures that hardware uses, with FreeBSD-specific notes on how to identify them through observation and how to set up the matching `bus_dma(9)` mappings on the driver side.

### 36.8.1 5.1 The Big Three: Linear, Ring, and Descriptor

Almost every data buffer in modern hardware falls into one of three shapes.

A **linear buffer** is a simple contiguous block of memory that the driver hands to the device for one operation. The driver fills it with data, the device consumes it, the driver hands it back when needed. Linear buffers are the simplest shape, and they show up in command-response patterns where each operation has its own dedicated buffer. They are easy to identify because they appear in the captures as a single block of bytes whose size matches the expected operation size.

A **ring buffer** is a circular buffer with a producer pointer and a consumer pointer. Producer writes go into the slot indicated by the producer pointer, then the producer pointer advances. Consumer reads come from the slot indicated by the consumer pointer, then the consumer pointer advances. Both pointers wrap when they reach the end of the buffer. The buffer is full when the producer catches up to the consumer; it is empty when the two pointers are equal in the other sense. Ring buffers are everywhere in high-speed networking and storage, because they let producer and consumer run at different rates without coordinated handshakes.

A **descriptor ring** is a special kind of ring buffer in which each slot does not hold the data itself but a small fixed-size descriptor that points to the data. The descriptor typically contains a memory address (where the data lives), a length, a status field (filled in by the device after processing), and some control flags. Descriptor rings let the device do scatter-gather DMA, accept variable-size buffers, and report per-buffer status back to the driver. They are slightly more complex than ring buffers but vastly more flexible, and they are the dominant pattern in high-performance network and storage controllers.

Recognising which shape a particular device uses is the first job. The signs are usually quite distinctive. If captures show fixed-size blocks moved one at a time with no metadata, it is probably a linear-buffer arrangement. If captures show data flowing continuously with no obvious “next operation” boundary, and there is a register that looks like a producer pointer, it is probably a ring buffer. If the device documentation, prior driver sources, or related-device datasheets hint at “descriptors”, it is almost certainly a descriptor ring.

### 36.8.2 5.2 Telling Buffers From Each Other

When you have identified that a buffer exists, the next question is: how big is it, and how is it organised internally? The captures from Section 3 contain the data; the question is what shape that data has.

Several observations can help.

**Size of the underlying memory region.** If a PCI BAR is exactly 4 KB and a sysctl reveals that the device has a “ring size” register set to 64, the natural conclusion is that the ring has 64 entries of 64 bytes each. Many devices use power-of-two sizes for both the ring and the entry size, and the product of the two should equal the BAR size.

**Periodic structure in the data.** A capture viewed in xxd or Wireshark sometimes shows obvious periodic structure: every 32 bytes there is a small fixed pattern, every 64 bytes there is a counter, every 16 bytes there is a status byte that varies in a predictable way. Periodic structure of size N strongly suggests that the buffer is organised as a sequence of records of size N.

**Alignment of pointer values.** If you see the device or the driver using “addresses” whose low bits are always zero, the addresses are aligned to a power-of-two boundary. If the alignment is 16 bytes, the entries are probably 16 bytes or larger. If the alignment is a megabyte, the entries are very large or the device imposes a coarse alignment requirement for some other reason.

**Headers and trailers.** Many record formats include a header that identifies the record type, a payload, and a trailer containing a checksum or a length. The header is often a magic value that can be recognised on sight.

**Cross-platform clues.** If the device has a Linux driver in the open-source tree, that driver’s code may already define the record structure. Reading another driver to learn record formats is one of the most efficient research techniques available, even when you intend to write the FreeBSD driver from scratch in a clean-room style. Two cautions are worth stating inline so that the rhythm of the chapter does not let them slip past. The first is legal: Linux drivers are almost always GPL-licensed, so reading them is fine, but pasting or near-transcribing their expression into a BSD-licensed FreeBSD driver is not. The second is technical: record-layout information ports cleanly between kernels because it describes the device, not the host, but surrounding code does not. Linux’s buffer-allocation paths, its locking primitives, and its descriptor-ring helpers are shaped by its own kernel, and a FreeBSD driver that tries to mirror them line-for-line will fight the host kernel at every step. Read to understand the format, then write FreeBSD code from your own notes. Section 12 covers the legal framing in detail; Section 13 shows how real drivers have done this well.

### 36.8.3 5.3 Command Queues and Their Sequencing

For devices with command interfaces, the typical shape is a queue of command structures, each of which the device processes in order, with status reported back through a separate register or through a status field in the command structure itself. Command queues are usually a special form of descriptor ring.

The sequencing matters. Some devices process commands strictly in order; others process them out of order and identify completion by a tag in the command structure. Some devices process commands one at a time; others process many in parallel. Some devices require the driver to wait for a command to complete before posting the next one; others accept many concurrent commands. Identifying which mode the device is in is part of the investigation, and the captures from Section 3 are usually rich enough to make the answer visible.

A useful technique is to deliberately exercise the device with operations that will overlap in time, and to observe how the driver responds. If the driver always waits for a command to complete before posting the next one, the device probably uses sequential processing. If the driver posts multiple commands in rapid succession and then waits for several completions, the device probably uses concurrent processing. If completions arrive in a different order from postings, the device is doing out-of-order processing.

### 36.8.4 5.4 DMA and Bus Mastering

Most modern devices that move significant amounts of data use DMA. The device, acting as a bus master, reads and writes the host's memory directly, rather than requiring the CPU to copy every byte. From the driver's side, DMA introduces several constraints that you have to handle correctly.

The buffers must be physically contiguous, or at least scattered in a way that the device can express. They must be aligned to a boundary that the device requires. They must be visible to the device, which on FreeBSD means they have to be set up through the `bus_dma(9)` framework. And the driver and the device have to synchronise their views of the buffer using `bus_dmamap_sync(9)` calls, because modern processors maintain caches that may not be coherent with device DMA.

The full mechanics of `bus_dma(9)` are covered in Chapter 21 and need not be repeated here. For reverse engineering, the implications are:

1. The device almost certainly imposes alignment and size constraints on its buffers. Captures that show buffers always starting at addresses with the same low bits are revealing the alignment.
2. The device may impose a maximum buffer size. Captures that show large operations split across multiple smaller transfers are revealing the maximum.
3. The device may use specific DMA address bits to encode metadata. A 32-bit pointer in a 64-bit address field, for example, sometimes leaves the upper 32 bits available for other uses, and the device may interpret those bits as flags or tags.

Identifying these constraints is part of the reverse-engineering work for any DMA-capable device. The captures and the existing driver source, if available, are the primary sources of evidence.

### 36.8.5 5.5 Reversing Packet Formats

When a device's data path uses structured packets, reversing the packet format is one of the most rewarding parts of the work. The recurring techniques are:

**Look for fixed bytes that never change.** These are usually magic numbers, version bytes, or category codes. Mark them and explore what they mean.

**Look for fields whose values correlate with operations.** A byte that takes a small set of distinct values, with each value appearing in exactly one kind of operation, is almost certainly an opcode or a command type.

**Look for length fields.** A two-byte or four-byte field whose value matches the size of the rest of the packet, in some predictable encoding, is a length field. Length fields are often near the start of the packet and are often the field that lets you parse variable-length packets correctly.

**Look for sequence numbers or session identifiers.** Fields that increment monotonically across a session, or that change once per operation, are usually sequence numbers. Their value is rarely interesting, but their presence and position help you ignore them while looking for the real content.

**Look for checksums.** A field whose value depends on the rest of the packet, in a way that you cannot predict from any single byte, is probably a checksum. Standard checksums (CRC-16, CRC-32, sums) can sometimes be identified by computing them over candidate ranges and seeing which range matches. If a checksum exists, you will need to compute it correctly when the driver constructs its own packets, so identifying the algorithm is necessary.

Each of these observations builds the packet-format part of the pseudo-datasheet. Combined with the register map from Section 4, they form the documentation that Section 11 will assemble into a proper datasheet substitute.

### 36.8.6 5.6 Identifying Which Operations Are Read-Like and Which Are

#### Write-Like

Many devices have an asymmetric data path: some operations cause the device to read from host memory, others cause it to write to host memory. Identifying which is which is essential for getting the DMA direction right.

The clearest evidence is in the captures. A USB capture, for example, distinguishes IN endpoints (device to host) from OUT endpoints (host to device) explicitly. PCI captures, when you can get them, distinguish memory reads from memory writes by the transaction type. Bulk DMA from a network controller into the host is recorded as the device writing into host memory. Bulk DMA from the host into the network controller is recorded as the device reading from host memory.

If captures are not available, a useful technique is to set the buffer to a recognisable pattern (all `0xCC` bytes, for example) before triggering the operation and to inspect the buffer afterwards. If the buffer's pattern is overwritten with new data, the operation was a device-to-host transfer (the device wrote into the buffer). If the buffer's pattern is unchanged, the operation was a host-to-device transfer (the device only read from the buffer). This kind of distinguishing experiment is one of the clearest examples of an experiment whose result will tell you a fact you cannot easily learn any other way.

### 36.8.7 5.7 Wrapping Up Section 5

We have learned to recognise the three big shapes of buffer organisation: linear buffers, ring buffers, and descriptor rings. We have seen how to deduce buffer sizes and entry sizes from periodic structures, alignments, and the underlying memory regions. We have learned to identify command queue shapes and sequencing models. We have noted the constraints that DMA imposes and the kinds of evidence that reveal them. We have gone through the recurring techniques for reversing packet formats. And we have seen how to identify the direction of a transfer when the captures themselves do not say so directly.

The next section turns this growing understanding into code. We will write the first version of an actual driver, one that implements only the smallest useful piece of functionality: usually reset, sometimes reset plus a single

status read, sometimes the absolute minimum needed to put the device into a state where it can be driven further. The principle is that a small driver that does something verifiably correct is a better starting point than a large driver that does many things uncertainly.

## 36.9 6. Re-Implementing Minimal Device Functionality

After observation comes reconstruction. With captures in hand, register hypotheses confirmed, and at least the rough shape of buffer structure understood, you can start writing a driver. The key word is “start”. The first version of a reverse-engineered driver does not have to do everything the device can do. It does not even have to do most of what the device can do. It has to do the smallest piece that is useful, and it has to do that piece correctly.

This section walks through the principles of starting small and the practical steps of writing the first minimal driver. The work here builds on every chapter from earlier in the book, because the mechanics of “write a driver” are exactly the same mechanics we have been practising. What changes is the discipline around which features to implement first and how to gain confidence that each feature is correct.

### 36.9.1 6.1 Start With Reset, Always

The first feature you implement should almost always be reset. A reset operation is the foundation of every other operation, for several reasons.

Reset is usually the simplest operation a device has. It often takes a single write to a single register, or a sequence of three or four writes that put the device into a known state. The hypothesis is small and easy to test.

Reset is the operation whose result is most easily verified. Before the reset, the device’s registers may hold arbitrary values. After a successful reset, they hold predictable defaults: zero configuration, no pending interrupts, no active operations, identifier registers reading their constant magic values. If you do a reset and then read the registers, you should see the “freshly powered up” state.

Reset is the operation whose failure modes are the most benign. If your reset attempt does not actually reset the device, the device will be in an unknown state, and you can recover by unloading your driver and letting the kernel re-probe the device. A failed reset rarely damages anything; it just leaves you no worse off than before.

Reset is also the precondition for almost every other operation. A driver that cannot put the device into a known state cannot rely on any subsequent operation. By implementing reset first, you build the foundation that every later test of every later feature will rely on.

### 36.9.2 6.2 The Skeleton of a Minimal Driver

For a PCI device, the skeleton driver looks very similar to every other PCI driver you have seen in the book. Here is the essential shape:

```
struct mydev_softc {
 device_t sc_dev;
 struct mtx sc_mtx;
```

```
 struct resource *sc_mem;
 int sc_rid;
 struct resource *sc_irq;
 int sc_irid;
 void *sc_ih;
 bus_size_t sc_size;
 /* Driver-specific state goes here as it accumulates. */
};

static int
mydev_probe(device_t dev)
{
 if (pci_get_vendor(dev) == 0xabcd &&
 pci_get_device(dev) == 0x5678) {
 device_set_desc(dev, "ExampleCorp XYZ Reverse-Engineered");
 return (BUS_PROBE_DEFAULT);
 }
 return (ENXIO);
}

static int
mydev_attach(device_t dev)
{
 struct mydev_softc *sc = device_get_softc(dev);
 int error;

 sc->sc_dev = dev;
 mtx_init(&sc->sc_mtx, "mydev", NULL, MTX_DEF);

 sc->sc_rid = PCIR_BAR(0);
 sc->sc_mem = bus_alloc_resource_any(dev, SYS_RES_MEMORY,
 &sc->sc_rid, RF_ACTIVE);
 if (sc->sc_mem == NULL)
 goto fail;
 sc->sc_size = rman_get_size(sc->sc_mem);

 sc->sc_irid = 0;
 sc->sc_irq = bus_alloc_resource_any(dev, SYS_RES_IRQ,
 &sc->sc_irid, RF_SHAREABLE | RF_ACTIVE);
 if (sc->sc_irq == NULL)
 goto fail;

 error = mydev_reset(sc);
 if (error != 0)
```

```
 goto fail;

 error = mydev_verify_id(sc);
 if (error != 0)
 goto fail;

 device_printf(dev, "attached, mapped %ju bytes\n",
 (uintmax_t)sc->sc_size);
 return (0);

fail:
 mydev_detach(dev);
 return (ENXIO);
}
```

The structure should look familiar. The driver allocates a memory resource for its register region and an IRQ resource for its interrupt. It then performs a reset and an identifier check. If either fails, the driver bails out cleanly. The reset and the identifier check are the first two functions worth writing.

### 36.9.3 6.3 Implementing Reset

The reset function is, in the simplest case, three lines of code:

```
static int
mydev_reset(struct mydev_softc *sc)
{
 bus_write_4(sc->sc_mem, MYDEV_REG_CONTROL, MYDEV_CTRL_RESET);
 pause("mydev_reset", hz / 100);
 bus_write_4(sc->sc_mem, MYDEV_REG_CONTROL, 0);
 return (0);
}
```

We write the reset bit to the control register, wait a short time for the device to act on it, and then clear the control register to leave the device in a quiescent state. The constants `MYDEV_REG_CONTROL` and `MYDEV_CTRL_RESET` come from the register map you built in Section 4.

In practice, reset is rarely this simple. Many devices have a specific reset protocol that involves several register writes in a specific order, or that involves polling a status register until the reset is complete, or that involves waiting for a specific duration to pass before proceeding. The captures from Section 3 show you what the vendor driver does on attach. Replicating that sequence is the safest reset implementation.

A more realistic reset might look like:

```

static int
mydev_reset(struct mydev_softc *sc)
{
 int i;
 uint32_t status;

 /* Disable any pending interrupts. */
 bus_write_4(sc->sc_mem, MYDEV_REG_INT_ENABLE, 0);

 /* Trigger reset. */
 bus_write_4(sc->sc_mem, MYDEV_REG_CONTROL, MYDEV_CTRL_RESET);

 /* Wait for reset complete, with timeout. */
 for (i = 0; i < 100; i++) {
 status = bus_read_4(sc->sc_mem, MYDEV_REG_STATUS);
 if ((status & MYDEV_STATUS_RESETTING) == 0)
 break;
 pause("mydev_reset", hz / 100);
 }
 if (i == 100) {
 device_printf(sc->sc_dev, "reset timed out\n");
 return (ETIMEDOUT);
 }

 /* Clear any pending interrupt status. */
 bus_write_4(sc->sc_mem, MYDEV_REG_INT_STATUS, 0xffffffff);

 return (0);
}

```

This version handles the case where the reset takes some time to complete and where the device exposes a status bit reporting “reset in progress”. Adding a timeout, with an explicit error path when the timeout expires, is a habit worth adopting from the start. A driver that cannot tell the difference between “the device is slow today” and “the device has failed” will eventually hang the kernel, and the cost of adding the timeout is trivial.

### 36.9.4 6.4 Verifying the Identifier

Every reverse-engineered driver should verify that the device attached to is the device the driver expects. The verification is typically one or two register reads, comparing the values against known constants:

```

static int
mydev_verify_id(struct mydev_softc *sc)
{
 uint32_t id, version;

```



```
id = bus_read_4(sc->sc_mem, MYDEV_REG_ID);
if (id != MYDEV_ID_MAGIC) {
 device_printf(sc->sc_dev,
 "unexpected ID 0x%08x (expected 0x%08x)\n",
 id, MYDEV_ID_MAGIC);
 return (ENODEV);
}

version = bus_read_4(sc->sc_mem, MYDEV_REG_VERSION);
if ((version >> 16) != MYDEV_VERSION_MAJOR) {
 device_printf(sc->sc_dev,
 "unsupported version 0x%08x\n", version);
 return (ENODEV);
}

device_printf(sc->sc_dev, "ID 0x%08x version 0x%08x\n",
 id, version);
return (0);
}
```

The check serves two purposes. First, it confirms that the driver's understanding of the register layout is correct: if the identifier register holds the expected value at the expected offset, the layout is at least consistent with what we believe. Second, it gives you a fast-fail path if the driver is somehow attached to the wrong device, perhaps because two devices share a PCI ID prefix or because the device has multiple silicon revisions.

### 36.9.5 6.5 Adding Logging

Logging is your eyes during reverse engineering. Every register write, every register read whose value matters, every milestone in the driver's startup sequence: log it. The kernel's `device_printf` goes to `dmesg`, where you can see it in real time as the driver loads.

Logging is not a debug aid you remove later. In a reverse-engineering project, the logging is part of the driver, and it stays in the driver until the driver is well understood. The cost is small: a handful of `device_printf` calls take negligible CPU time and produce a few lines of output. The benefit is large: when something goes wrong, you have a complete record of what the driver tried to do.

A common pattern is to wrap register access in a small inline helper that logs the access:

```
static inline uint32_t
mydev_read(struct mydev_softc *sc, bus_size_t off)
{
 uint32_t val = bus_read_4(sc->sc_mem, off);
 if (mydev_log_reads)
 device_printf(sc->sc_dev,
```

```

 "read off=0x%04jx val=0x%08x\n",
 (uintmax_t)off, val);
 return (val);
}

static inline void
mydev_write(struct mydev_softc *sc, bus_size_t off, uint32_t val)
{
 if (mydev_log_writes)
 device_printf(sc->sc_dev,
 "write off=0x%04jx val=0x%08x\n",
 (uintmax_t)off, val);
 bus_write_4(sc->sc_mem, off, val);
}

```

The `mydev_log_reads` and `mydev_log_writes` flags can be tunable sysctls, so you can enable or disable logging without rebuilding the driver. The companion code in `examples/part-07/ch36-reverse-engineering/lab03-register-map/` uses exactly this pattern.

### 36.9.6 6.6 Comparing Logged Output Against Captures

With logging in place, you can compare what your driver does against what the vendor's driver did. Run the vendor's driver in the observation rig and capture the result. Run your driver in your development environment and capture its log output. Compare.

If the two sequences match, your driver is doing what the vendor's driver does, at least for the operations you have implemented. If they differ, the differences are telling you something. Maybe your driver is missing a write that the vendor's driver makes. Maybe your driver is writing a value that differs from what the vendor's driver wrote. Maybe your driver is reading a register that the vendor's driver does not read. Each difference is a hypothesis to investigate.

This comparison technique is one of the most effective you have. It turns the captures from passive evidence into an active specification: the driver should produce the same sequence of operations as the captures show. Where the driver deviates, the deviation is a bug or a missing piece, and the captures tell you what to add.

### 36.9.7 6.7 Writing the Module Boilerplate

The minimal driver also needs the standard module boilerplate. By this stage in the book the boilerplate should be familiar:

```

static device_method_t mydev_methods[] = {
 DEVMETHOD(device_probe, mydev_probe),
 DEVMETHOD(device_attach, mydev_attach),
 DEVMETHOD(device_detach, mydev_detach),
}

```

```
 DEVMETHOD_END
};

static driver_t mydev_driver = {
 "mydev",
 mydev_methods,
 sizeof(struct mydev_softc),
};

DRIVER_MODULE(mydev, pci, mydev_driver, NULL, NULL);
MODULE_VERSION(mydev, 1);
```

The `pci` parent in `DRIVER_MODULE` makes the kernel offer this driver every PCI device during probe, and the driver's probe method decides whether to claim each one. The `MODULE_VERSION` macro is a courtesy to anyone using `kldstat -v` to inspect loaded modules; it is also required if other modules want to declare a dependency on yours.

The Makefile is the standard FreeBSD kernel-module Makefile:

```
KMOD= mydev
SRCS= mydev.c device_if.h bus_if.h pci_if.h

SYSDIR?= /usr/src/sys

.include <bsd.kmod.mk>
```

The `device_if.h`, `bus_if.h`, and `pci_if.h` listed in `SRCS` are generated header files. The build system creates them on demand, and listing them tells `make` to do so before compiling `mydev.c`. If the driver becomes more complex and is split into multiple source files, list each source file separately.

## 36.9.8 6.8 The First Successful Load

When you load the minimal driver for the first time, the success criterion is modest. The driver should:

1. Be successfully loaded by `kldload`.
2. Recognise the device through probe.
3. Allocate its memory and IRQ resources through `bus_alloc_resource_any`.
4. Reset the device.
5. Verify the identifier.
6. Report success in `dmesg` and stay loaded without errors.
7. Be successfully unloaded by `kldunload` without panicking.

A driver that meets these criteria has done useful work. It has proven that the register layout is at least partially correct, that the reset sequence works, and that the identifier register is where you thought it was. Compared to where you started, this is substantial progress.

If any of the steps fail, the failure is informative. A `bus_alloc_resource_any` failure usually means the BAR layout is not what you expected. An ID-verification failure usually means the register offset is wrong, or the device's identifier value is not what you assumed. A panic on unload usually means a resource was not properly released; this is a bug in the detach path that needs to be fixed before any further work proceeds.

### 36.9.9 6.9 Unloading Cleanly

The detach path is at least as important as the attach path. A driver that can be loaded but not unloaded cleanly is a driver that requires a reboot every time you want to test a change, and the reboot will quickly become the bottleneck of your work. Spending time on a clean detach path early pays large dividends in development speed.

```
static int
mydev_detach(device_t dev)
{
 struct mydev_softc *sc = device_get_softc(dev);

 if (sc->sc_ih != NULL)
 bus_tearardown_intr(dev, sc->sc_irq, sc->sc_ih);
 if (sc->sc_irq != NULL)
 bus_release_resource(dev, SYS_RES_IRQ,
 sc->sc_irid, sc->sc_irq);
 if (sc->sc_mem != NULL)
 bus_release_resource(dev, SYS_RES_MEMORY,
 sc->sc_rid, sc->sc_mem);
 mtx_destroy(&sc->sc_mtx);
 return (0);
}
```

Each resource that was allocated must be released. Each registered interrupt handler must be torn down. Each mutex must be destroyed. The order is the reverse of the allocation order, which is a useful general rule.

### 36.9.10 6.10 Wrapping Up Section 6

The minimal driver is a foundation. It does very little, but it does that little correctly. Reset works, identifier verification works, the resource allocations are right, the detach path is clean. From this base, every further feature can be added incrementally, with confidence that the underlying scaffolding is sound.

The next section takes the minimal driver and shows how to grow it one feature at a time, always confirming that each new feature is correct before adding the next. The technique is the discipline of incremental development applied to a domain where there is no specification to refer to: instead of validating against a spec, we validate against captures and against measured behaviour. The result is the same: a driver that does what we believe it does, backed by evidence rather than by hope.

## 36.10 7. Iteratively Expanding the Driver and Confirming Behavior

The minimal driver in Section 6 is the seed of the real driver. To grow it into something useful, you add features one at a time, and for each feature you confirm that what the feature does in the running driver matches what you have observed the device doing under the vendor's driver. The process is straightforward but demanding: every step has to be verified, every verification has to be recorded, and every recording becomes part of the pseudo-datasheet that Section 11 will assemble.

This section walks through the methodology of incremental expansion. It does not introduce new low-level techniques, because the low-level techniques are the same as in Sections 4 and 6. What it adds is the discipline of growing a driver responsibly under uncertainty.

### 36.10.1 7.1 The One-Feature-at-a-Time Rule

The rule is in the name. Add one feature, verify it works, commit the result to your repository, then add the next feature. Do not add three features at once and then try to debug the combination. Do not add a feature and proceed to the next without verification. Do not skip the commit, because the commit is what lets you roll back when a later change breaks something that worked before.

The discipline is the same as test-driven development, with one significant adaptation: the test is often a comparison against the captures from Section 3, not a unit test in the traditional sense. The “test” is “the driver, when invoked to perform operation X, produces the same sequence of register accesses as the vendor's driver does”. The comparison is mechanical, but it is a real test: either the sequences match or they do not, and a failure tells you that something has changed.

A typical sequence of features for a network controller might be:

1. Read MAC address.
2. Set MAC address.
3. Set up receive descriptor ring.
4. Receive one packet.
5. Set up transmit descriptor ring.
6. Send one packet.
7. Enable interrupts.
8. Implement interrupt handler.
9. Bring the link up.
10. Bring the link down.
11. Reset and recover from error.

Each item is a discrete feature. Each item builds on the previous items. Each item has a clear success criterion (the operation either works or it does not). And each item has a corresponding capture from Section 3 against which the driver's behaviour can be compared.

### 36.10.2 7.2 The Verification Loop

For each new feature, the verification loop is:

1. Form the hypothesis: this feature works by doing X.

2. Implement the feature in the driver, with logging that records each register access.
3. Trigger the feature using a small userspace program, a `sysctl`, or whatever interface you have wired up.
4. Capture the kernel log output.
5. Compare against the relevant capture from Section 3.
6. If they match, declare the feature implemented.
7. If they do not match, identify the difference and decide whether your implementation is wrong or your hypothesis was wrong.

The loop is slow. A single feature might require several iterations through the loop, with hypothesis revisions in between. But the slowness is the point. Each iteration is a deliberate test, with a deliberate prediction and a deliberate observation. Over many iterations, the body of verified facts grows steadily and the chance of an incorrect belief surviving into the final driver shrinks rapidly.

### 36.10.3 7.3 The Compare-Against-Vendor Pattern

The comparison against the vendor's driver is so central to this work that it deserves its own discussion. The pattern looks like this.

You have a capture from Section 3 of the vendor's driver performing operation X. The capture contains a list of register accesses (for memory-mapped devices) or USB transfers (for USB devices), each labelled with timestamp, address or endpoint, direction, and value.

You have your driver, with logging enabled, performing the same operation X. The kernel log contains a list of register accesses or USB transfers, each labelled with timestamp, address or endpoint, direction, and value.

You compare the two lists. The accesses should be in the same order, to the same addresses or endpoints, in the same direction, with values that either match exactly or differ in ways that you can explain (because, for example, the values include sequence numbers or timestamps that naturally differ between runs).

Where the two lists agree, your driver is doing what the vendor's driver does. Where they disagree, you have a discrepancy to investigate. Some discrepancies are benign: your driver might omit a redundant register read that the vendor's driver does for some internal reason, and the omission has no effect on the device's behaviour. Other discrepancies are real bugs: a value you wrote incorrectly, a sequence step you missed, a register you did not realise the operation needed.

The compare-against-vendor pattern is essentially a specification-by-example. The specification of operation X is "the sequence of accesses the vendor's driver makes when performing operation X". Your implementation conforms to the specification when its sequence of accesses matches. The technique is not perfect: it cannot detect bugs that affect the device's internal state without affecting the access sequence, and it cannot detect bugs in the values you write if those values change between runs in the vendor's capture too. But it catches the great majority of implementation errors, and it catches them at exactly the level of detail where they happen.

### 36.10.4 7.4 Documenting Each Discovered Register

As you add features, you discover registers. Every register that you write to or read from in any feature is part of the register map, and every one of them needs to be added to the pseudo-datasheet you are building.

A useful technique is to maintain the pseudo-datasheet as a separate Markdown file alongside the driver source, and to update it every time you discover a new fact. Section 11 will discuss the pseudo-datasheet's structure

in detail; for now, the rule is simply that no fact you have learned should remain only in the driver code. The driver code is the implementation; the pseudo-datasheet is the documentation. They serve different purposes, and someone reading either should be able to understand the device.

A common mistake is to defer documentation until the project is “finished”. Reverse-engineering projects are rarely finished in that sense. They reach increasing levels of completeness over time, and the documentation should grow alongside the implementation. A driver with twenty features and a one-paragraph pseudo-datasheet is a maintenance disaster waiting to happen, because nobody can extend it without rediscovering everything you already knew.

### **36.10.5 7.5 When the Vendor’s Driver Disagrees With Itself**

A subtle complication arises when the vendor’s driver performs the same operation differently on different occasions. Sometimes this happens because the driver has multiple code paths for the same operation, depending on the device’s state or the user’s configuration. Sometimes it happens because the driver has optional behaviour that activates under specific conditions.

When you encounter such variation, the right response is to capture more, not to give up. Take captures of the operation under several different starting conditions and compare them. If you can identify what condition selects each variant, you have learned something about the driver’s logic. If the variants are functionally equivalent (they all achieve the same end state), you can choose the simplest one as the canonical implementation. If they are not functionally equivalent, the device may have multiple modes of operation that you need to support.

Sometimes the variation is just non-determinism: the vendor’s driver chose a sequence number from a counter, or used a timestamp, and the resulting capture differs slightly from one run to another. Non-determinism is benign, and learning to recognise it (so you can ignore it) is part of the skill of comparing captures.

### **36.10.6 7.6 Branching the Driver as Hypotheses Diverge**

When you have two competing hypotheses and an experiment that will distinguish them, branch the driver. Make a Git branch for the experiment, implement the change, run the experiment, observe the result, and then either merge the branch or discard it. Branches are cheap; you can have several hypothesis-testing branches in flight at once, and you can come back to any of them whenever a new experiment seems worthwhile.

The branching discipline is particularly valuable when an experiment requires significant code changes. Without branches, you might be reluctant to spend an hour writing experimental code that you will throw away. With branches, the experimental code lives in its own branch, and your main branch stays clean. If the experiment succeeds, you merge. If it fails, you discard. Either way, you have a clean history.

### **36.10.7 7.7 Wrapping Up Section 7**

We have learned how to grow a minimal driver into a more capable one through a disciplined cycle of one-feature-at-a-time implementation, comparison against captures, and documentation of every discovered fact. The mechanics are simple. The discipline is demanding. The reward is a driver whose every feature is backed by evidence, whose register map is recorded as it grows, and whose behaviour can be defended against any future challenge.

The next section turns to a slightly different topic: how to build confidence in your driver before you risk running it against real hardware. Simulation, mock devices, and validation loops are the techniques that let you find bugs in safety, where a kernel panic or a hang costs only a VM reboot, rather than in production, where the cost can be much higher.

## 36.11 8. Validating Assumptions With Simulation Tools

Reverse-engineering work moves between two kinds of risk. The first is the risk of being wrong: of writing code that does something other than what you believe it does. The second is the risk of being right too late: of finding out about a bug only when it has already damaged something, when the bug has driven the device into an unrecoverable state, or when the bug has crashed a production machine. The first risk is unavoidable; bugs are intrinsic to the work. The second risk is largely avoidable, and the way to avoid it is simulation.

A simulator is anything that lets you run your driver code without the real device being attached. Simulators range from simple mock implementations of a few register reads, through full-featured software emulations of an entire device, to virtualised hardware environments that pretend to be the real device for the operating-system stack above them. FreeBSD provides several simulator-like tools that fit naturally into the reverse-engineering workflow.

This section covers three kinds of simulation: mock devices that you write inside the kernel, the USB template framework for emulating USB devices, and bhyve passthrough for running an unmodified real device under a controlled hypervisor. We will see how each kind helps you validate assumptions, where each is most useful, and how to combine them with the captures and the register-map work from earlier sections.

### 36.11.1 8.1 The Kernel-Space Mock Device

The simplest form of simulation is a small kernel module that exposes the same interfaces as the real device but implements them in software. The mock module attaches as a child of an existing bus, allocates a software-backed “register region”, and implements the register reads and writes in C code that runs in the kernel.

For a memory-mapped device, the mock works like this. Where the real driver would allocate `SYS_RES_MEMORY` from the bus and use `bus_read_4` and `bus_write_4` to talk to the device, the mock hands the driver a pointer to a software-allocated array of words. Reads return the current value of the word at the requested offset. Writes update the word and trigger any software-implemented side effects. Side effects are how the mock implements the device’s behaviour: when the driver writes the “go” bit to the control register, the mock might internally start a callout that, after a simulated processing time, updates the status register to indicate completion and asserts a simulated interrupt.

The mock is small, often a few hundred lines for a device whose real driver might be several thousand lines. The point is not to implement everything; the point is to implement enough that the driver’s behaviour can be tested in isolation. A driver that correctly handles “command issued, command complete, results returned” against a mock has at least proved that its top-level sequencing is correct, even if the mock cannot test what happens when the real device produces values the driver did not anticipate.

The companion files in `examples/part-07/ch36-reverse-engineering/lab04-mock-device/` contain a small mock device that demonstrates the pattern. The mock implements a tiny “command and status” interface: there is a `CMD` register the driver writes to, a `STATUS` register the mock updates after a simulated delay, and a



DATA register the driver reads after the status indicates completion. The mock is intentionally simple, but it shows the structure: define the register addresses, allocate backing storage for them, implement read and write callbacks, and arrange for callbacks to do whatever side-effect logic the simulated device requires.

### 36.11.2 8.2 Trapping Driver Accesses

A more sophisticated mock can be built by interposing on the bus space layer itself. FreeBSD's `bus_space(9)` API uses opaque handles, and a custom bus-space implementation can intercept every read and write that the driver performs. The interception can log the access, modify the value, return a pre-arranged response, or simulate any other behaviour.

This technique is useful when you want to test a driver that has already been written for a real device, against a simulator, with minimal changes to the driver itself. The driver continues to use `bus_read_4` and `bus_write_4` as it always does. The intercept layer takes the calls and either dispatches them to the real device or fakes them as needed.

The technique is also useful when you want to record exactly what the driver does during a test, in a form that can be compared against captures from Section 3. An intercepting bus-space layer can produce a log file that has the same structure as a capture file, ready for direct comparison.

Implementing this kind of intercept is more involved than the simple mock, because you have to provide a complete bus-space implementation. The pattern is well established in the FreeBSD source. Readers interested in the deep version of this technique can study the architecture-specific bus-space backends in `/usr/src/sys/x86/include/bus.h` and the related files; the function-pointer table approach used there is the same approach you would use for an interception layer.

### 36.11.3 8.3 USB Template Drivers

For USB devices, FreeBSD includes an unusually capable simulation mechanism: the USB template framework. The template framework lets the kernel pretend to be a USB device, with a programmable set of descriptors, by exposing the USB device endpoint via the host controller in device mode. The relevant source files are under `/usr/src/sys/dev/usb/template/`, and the framework includes ready-made templates for several common device classes:

- `usb_template_audio.c` for USB audio devices.
- `usb_template_kbd.c` for USB keyboards.
- `usb_template_msc.c` for USB Mass Storage devices.
- `usb_template_mouse.c` for USB mice.
- `usb_template_modem.c` for USB modems.
- `usb_template_serialnet.c` for USB serial-network composite devices.

Several additional templates under the same directory cover other classes, including CDC Ethernet (`usb_template_cdce.c`), CDC EEM (`usb_template_cdceem.c`), MIDI (`usb_template_midi.c`), MTP (`usb_template_mtp.c`), and multi-function and phone composite devices (`usb_template_multi.c`, `usb_template_phone.c`). Browse the directory on your own system to see the full list; the set is expanded over time as contributors add new device classes.

Each template is a programmable description of a USB device: which class it claims to be, which endpoints it exposes, what its descriptors look like. Loaded into a USB host controller running in device mode (the controller pretends to be the device side of a USB cable), the template lets the host see a USB device that matches the descriptors.

For reverse-engineering work, the templates have two uses. The first is as a probe-test substrate: you can write a USB device descriptor that mimics the static identity of an unknown device and use the template framework to expose it to a development machine. The development machine's USB stack will probe the descriptors, attach drivers (yours, or the in-tree drivers if your descriptors collide), and you can observe what the host expects from a device with this identity. The exercise is particularly valuable when you have captured the descriptor of an unknown device but cannot get the device itself onto your development machine; the template lets you simulate what the host would see.

The second use is as a foundation for a software simulator that implements the full protocol of an unknown USB device. Building such a simulator is a substantial undertaking, but for very important reverse-engineering targets it can be the only way to test driver code without risking the real hardware. The templates provide the descriptor side of the equation; you have to add the data-handling logic yourself.

#### 36.11.4 8.4 bhyve and Passthrough

bhyve is FreeBSD's native hypervisor, and its PCI passthrough support is a valuable tool for reverse engineering. The relevant code is in `/usr/src/usr.sbin/bhyve/pci_passthru.c`, and the kernel side uses the `ppt(4)` driver (`/usr/src/sys/amd64/vmm/io/ppt.c`) to claim the device and expose it to the guest.

The passthrough workflow is:

1. On the host, detach the device from any in-tree driver:

```
$ sudo devctl detach pci0:0:18:0
```

2. Configure `ppt(4)` to claim the device, by adding an entry to `/boot/loader.conf`:

```
pptdevs="0/18/0"
```

This requires a reboot to take effect, after which the device will be claimed by `ppt(4)` instead of any other driver.

3. Start a bhyve guest with the device passed through:

```
$ sudo bhyve -c 2 -m 2G \
-s 0,hostbridge \
-s 1,virtio-blk,disk.img \
-s 5,passthru,0/18/0 \
-s 31,lpc \
-l com1,stdio \
myguest
```

The `-s 5, passthru, 0/18/0` argument tells bhyve to expose the host's PCI device at 0/18/0 to the guest, where it will appear in the virtual PCI bus.

Inside the guest, the device behaves like a normal PCI device. The vendor's driver, running in the guest, can access it just as it would on bare metal. From the host, you cannot directly see the device's register accesses, because the hypervisor handles them for the guest, but you can use bhyve's logging facilities to see configuration-space accesses, interrupt routing, and other operations that pass through the hypervisor.

For reverse engineering, the value of bhyve passthrough is that it lets you run the vendor's driver in a controlled environment where you can capture some kinds of activity that would otherwise be invisible. It also lets you snapshot and restore the host between experiments, so that you can recover quickly if an experiment goes wrong.

A particularly useful pattern is to develop your own driver on the host, with the device detached from ppt(4), and to switch to guest-passthrough only when you want to compare your driver's behaviour against the vendor's. The switch requires changing the ppt devs configuration and rebooting, but the inconvenience is small compared to the value of being able to run both implementations on the same hardware.

### 36.11.5 8.5 Loop With Known Return States

A simple but effective form of simulation is the validation loop with known return states. The idea is to instrument your driver so that, in test mode, certain register reads return values that you specify rather than values from the device. You then exercise the driver and observe what it does in response to each hand-chosen value.

For example, suppose you want to know what your driver does when the device returns an unexpected status value. In production, the device might never produce such a value, and there is no obvious way to test the path. With a validation loop, you replace the read of the status register with a function that returns a sequence of values you specify: 0x00, then 0x01, then 0xff, then back to 0x00. The driver behaves as if the device returned each of those values in turn, and you can verify that the driver's response is correct in each case.

The validation loop technique is particularly valuable for error paths. Most error conditions are rare in real operation, and testing them by waiting for them to occur naturally would take forever. By forcing them to occur on demand, you can test the error paths systematically. The technique is, in spirit, the same as the fault-injection techniques used in mature systems software, adapted to the reverse-engineering context where the driver itself is the artifact under test.

### 36.11.6 8.6 Limits of Simulation

Simulation is enormously valuable, but it has limits. A simulator implements the simulator's understanding of the device, not the device itself. When the simulator and the real device differ, the simulator's tests pass and the real device misbehaves, and you have learned that your understanding of the device was incomplete in some specific way.

The right attitude is to treat simulation as a way to find certain classes of bugs (driver logic errors, sequence errors, error-handling errors) and not as a guarantee that the driver will work on real hardware. Bugs in the driver's understanding of the device's contract will not be caught by a simulator that shares that misunderstanding. Bugs in the device's actual behaviour, that differ from what the simulator implements, will not be caught by simulator testing at all.

The combination is the answer. Simulator testing catches the bugs it can. Real-device testing, performed carefully and incrementally with the safety techniques from Section 10, catches the rest. The two together are far better than either alone.

### 36.11.7 8.7 Wrapping Up Section 8

We have seen the simulation tools that complement the reverse-engineering workflow: small in-kernel mock devices, the USB template framework, bhyve passthrough, and validation loops with known return states. Each tool addresses a different aspect of the problem, and together they let you find a great many bugs before any of them reach the real device.

The next section steps back from the technical work and looks at the social context: how reverse-engineering projects collaborate, where to find prior work, and how to publish your own findings. The technical part of reverse engineering is the visible part, but the community part is what makes the work durable. A driver that is well documented and well shared lasts. A driver that is written quickly and never published often disappears with its author.

## 36.12 9. Community Collaboration in Reverse Engineering

Reverse engineering is rarely a solitary activity, even when it appears to be. Almost every successful reverse-engineering project either builds on prior work or eventually contributes to it. The community of people who care about a particular device, or about a particular family of devices, is small but persistent, and learning to find and contribute to that community is one of the most efficient things a reverse engineer can do.

This section discusses how to find prior work, how to evaluate its trustworthiness, how to build on it without legal or ethical problems, and how to share your own findings in a form that helps others. The work is not glamorous, but it is what makes reverse engineering a productive long-term activity rather than a series of disconnected efforts.

### 36.12.1 9.1 Finding Prior Work

When you start investigating a device, the first thing to do is search for prior work. Even devices that seem to be obscure are sometimes the subject of significant prior research, and finding that research saves enormous time.

The most valuable sources, in approximate order of usefulness:

**FreeBSD source tree.** Search `/usr/src/sys/dev/` for the device's vendor ID, device ID, USB descriptor, or vendor name. A driver in the FreeBSD tree is the gold standard of prior work, because it is already validated, builds against the current kernel, and follows FreeBSD conventions. Even when no driver exists, related drivers for the same vendor often share patterns that apply.

**OpenBSD and NetBSD source trees.** The other BSDs have their own driver collections, and they often have drivers for devices that FreeBSD does not yet support. The code is generally straightforward to read and to port. Licensing is usually compatible with FreeBSD (most BSD-licensed code is).

**Linux source tree.** The Linux kernel has the broadest collection of device drivers in any open-source project. Most are GPL-licensed, which means you cannot copy code into a FreeBSD driver, but you can read them for understanding and write your own implementation from a clean slate. The clean-room approach of reading the Linux driver to understand the device, then writing original code from that understanding, is well established and legally accepted in interoperability cases; the full framework for doing this safely, including the paper trail that a clean-room defence actually depends on, appears later in this chapter in Section 12. Even when the legal

question is settled, a direct port from Linux to FreeBSD is rarely workable for purely technical reasons. Linux drivers lean on kernel APIs that have no direct FreeBSD equivalent: its spinlock and mutex primitives have different wake-up semantics, its memory allocators distinguish contexts that FreeBSD expresses differently, and its device model (with `struct device`, `sysfs`, and the driver-core probe flow) does not map onto Newbus. The right way to use a Linux driver is to mine it for device-level facts (register layouts, magic values, initialisation order, quirk tables) and then rebuild the control flow against FreeBSD's own primitives. Section 13 shows two FreeBSD drivers that did exactly this.

**Hardware-specific community sites.** For many devices, enthusiasts have built dedicated sites that collect everything known about a particular family. The Wireshark wiki has extensive documentation on USB protocols. The OpenWrt project has documentation on many embedded devices. Many specific hardware families have their own community wikis or forums.

**Vendor mailing lists, technical support archives, application notes.** Some vendors publish more in technical support contexts than in their public documentation. A search through their support archives or developer mailing lists can sometimes turn up information that is not in the official documentation.

**Patents.** Patent applications often contain detailed descriptions of how a device works, because the legal requirement to describe the invention forces some level of disclosure. Patent databases are searchable by company and by year, and a search for the right vendor's patents can sometimes uncover a wealth of detail.

**Academic papers.** When the device is in a specialised field (scientific instrumentation, industrial control, certain classes of network equipment), academic papers in that field may have already documented the device's interface for research purposes.

The first hour of any project should be spent searching these sources. Even an unsuccessful search is informative: knowing that no prior work exists changes the project's scope significantly.

### 36.12.2 9.2 Evaluating Trustworthiness

Not all prior work is equally trustworthy. A working open-source driver is highly trustworthy, because the author's claims are backed by code that demonstrably works. A community wiki page written by an enthusiast may or may not be trustworthy, depending on the source's track record. A vendor white paper is generally trustworthy on points the vendor wants to make and possibly silent or misleading on points the vendor would rather not discuss.

The skill of evaluating sources is the same skill that historians and journalists develop. You ask: who wrote this, when, on the basis of what evidence, with what motivations? You compare claims across multiple sources. You give the most weight to claims that are supported by code or by direct measurement. You give less weight to claims that depend on the author's reputation alone.

When you adopt a fact from prior work, record where it came from. The notebook entry should say "from the Linux ath5k driver, register XYZ at offset N is the channel selector". A year later, when the fact turns out to be wrong, you will know where the error came from and what other facts from the same source might also be wrong.

### 36.12.3 9.3 The Clean-Room Discipline

The clean-room discipline is the standard methodology for using prior work without infringing on its copyright. The discipline is straightforward but requires care.

In the strict clean-room form, two people are involved. One reads the prior work (a vendor driver, a leaked specification, a disassembled binary) and produces a description of what the device does. The description is in natural language and contains no copyrighted material. The second person reads only the description and writes the new driver. The second person never sees the original work. Because the new driver was written without reference to the original, it cannot be a derivative of it under copyright law.

In the relaxed form, often used in solo projects, the same person performs both roles, but is careful to keep them separated in time and to document the separation. Read the prior work. Take notes. Put the prior work away. Wait. Then, working only from your notes, write the driver. The notes are the bridge; the prior code is never in front of you while you are writing the new code.

The relaxed form is less legally bulletproof than the strict form, but it is what most individual reverse engineers actually do, and the case law on interoperability work generally supports it. The key is the discipline of separating reading from writing, so that you can prove (and prove to yourself) that the new code is independent.

In all cases, the new driver must not contain copied code, copied register-name macros, copied data structures, or copied comments from the original. It can contain ideas, structural patterns, and the device's interface itself, because those are not copyrightable. Anything that is copyrightable must be re-expressed in your own words and your own structure.

### 36.12.4 9.4 Sharing Your Findings

Reverse-engineering work, when complete, should be published. The publication serves several purposes: it lets others use the driver, it lets others extend the driver, it preserves the knowledge against the case where you stop working on it, and it contributes to the body of public knowledge about the device.

The minimum useful publication is a Git repository that contains:

- The driver source code, with a clear license.
- A README explaining what the driver does and how to build it.
- The pseudo-datasheet (Section 11) that documents the device.
- The test programs and any other tools needed to use the driver.
- The notebook, or a cleaned-up version of it, so that others can trace the reasoning behind any claim.

A more polished publication adds:

- A document describing the methodology, so that others can reproduce the work.
- A list of open questions and unresolved issues.
- A clear distinction between verified facts and remaining hypotheses.
- Attribution to all sources of prior work, with links.

The license matters. For a driver intended for FreeBSD inclusion, the BSD two-clause license is the standard choice. The license must make the work compatible with the FreeBSD source tree's license requirements. If you have used material from a GPL-licensed project, even cleanroom-style, document the provenance carefully so that any concerns about derivation can be resolved by examining your notes.

Once you have published, announce the work in places where interested people will see it: the FreeBSD-arm mailing list for embedded work, the FreeBSD-net mailing list for network drivers, relevant project-specific communities, and your personal channels. The announcement should be brief but informative: what the driver does, what state of completeness it is in, where the repository is, and who is welcome to contribute.

### 36.12.5 9.5 Maintaining a Pseudo-Datasheet in Markdown or Git

The pseudo-datasheet itself deserves special attention. It is the single most valuable artifact you produce, often more valuable than the driver code. The driver code is one implementation of your understanding of the device; the pseudo-datasheet is the understanding itself, and from it any number of drivers (FreeBSD, Linux, NetBSD, custom embedded systems) can be written.

The format that has proven most useful in community projects is a Git-tracked Markdown file (or a small set of files), structured around the device's logical components: identity, register map, buffer layouts, command set, error codes, initialisation sequence, programming examples. We will look at the structure in detail in Section 11.

The Git tracking is essential. The pseudo-datasheet is a living document; it will be updated as new facts are discovered, as old facts are corrected, as additional reverse-engineering work extends its coverage. The Git history is the record of how the understanding has evolved. When a fact is corrected, the Git log shows when it was corrected and by whom, so that other readers can know when their old printout has gone out of date.

The companion code in `examples/part-07/ch36-reverse-engineering/lab06-pseudo-datasheet/` contains a Markdown template for a pseudo-datasheet, suitable as a starting point for your own projects. The template is opinionated about structure, because consistent structure makes the document easier to read, easier to compare against other pseudo-datasheets, and easier to extract automatically into other formats.

### 36.12.6 9.6 Wrapping Up Section 9

The community-collaboration side of reverse engineering is the side that gives the work its longevity. A driver written in isolation and never published may serve its author well, but it disappears with that author. A driver written in dialogue with prior work and published clearly serves a much wider audience and lasts much longer.

The next section turns to the safety side of the work, which we have referenced repeatedly. Safety deserves its own dedicated section, because the consequences of getting it wrong can include permanent damage to expensive hardware, and the techniques for avoiding such damage are concrete enough to be taught explicitly.

## 36.13 10. Avoiding Bricking or Hardware Damage

This is the section that this chapter has been deferring all along. Reverse engineering, done carelessly, can damage hardware. Some kinds of damage are recoverable with effort. Other kinds turn a working device into an inert brick that nothing short of a specialised vendor recovery tool, if it exists at all, can fix. A reverse engineer who works without understanding which patterns are dangerous will eventually destroy something. The goal of this section is to tell you, concretely, which patterns to avoid and which patterns are safe.

The advice in this section applies to every other section in the chapter. The probing techniques in Section 4 are safe only when combined with these warnings. The minimal driver in Section 6 is safe only when it writes

are constrained as described here. The iterative expansion in Section 7 is safe only when each new feature is evaluated against these criteria before it is allowed to write to the device. Read this section in full; do not skip to the technique sections without it.

### 36.13.1 10.1 The General Principle

The general principle is simple to state and, with practice, straightforward to apply.

**Never write to an unknown register or send an unknown command without strong evidence that the operation is safe.**

The strong evidence can come from several sources, and each is worth its own short discussion.

**The vendor's driver performed the same operation.** If you have captured the vendor's driver writing a specific value to a specific register, and the resulting device behaviour was healthy, then performing the same write in your driver is, almost certainly, safe. The vendor's driver is presumably well-tested and would not casually do something that bricks the device.

**The operation is a documented standard operation.** Some operations are so universal that they can be assumed to be safe. PCI configuration-space reads are safe by design. PCI standard control register writes (clear pending interrupts, enable bus mastering, set memory access enable) are safe in the standard range. USB control transfers using standard requests on standard endpoints are safe. The standardised parts of a protocol can be trusted; the vendor-specific parts cannot be trusted without evidence.

**The operation is reversible.** A write that you can immediately undo is much safer than one you cannot. Setting a configuration bit, observing the effect, and then restoring the original value is far safer than performing an operation that has permanent consequences.

**The operation has been tested in simulation.** A write that you have first tested against a mock or a simulator, where the worst-case outcome is a kernel panic in a test VM, is safer than a write you have only modeled in your head.

In the absence of any of these forms of evidence, the operation is unknown, and the rule is: do not perform it.

### 36.13.2 10.2 Operations That Deserve Particular Caution

Some categories of operations are dangerous enough to deserve explicit attention. These are the operations where a careless write can permanently damage a device, and they are the operations that should never be performed without strong evidence.

**Flash writes.** Many devices contain flash memory used for firmware, configuration, calibration, or device identifiers. Writes to flash are irreversible (or at best reversible by restoring a backup). A driver that accidentally writes to flash can corrupt the firmware (rendering the device unbootable), overwrite calibration data (rendering the device inaccurate or unusable), or change the device's identity (rendering it unrecognisable to its own firmware). The patterns that trigger flash writes vary by device, but they often involve specific "unlock" sequences followed by writes to addresses that are explicitly the flash region. If your captures show such a sequence in a context where the user explicitly performed a "firmware update" operation, do not replicate the sequence in your driver unless you are explicitly implementing firmware update.



**Hard resets that affect non-volatile state.** Most resets are soft resets that return the device to its initial in-memory state. Some resets, however, also clear non-volatile state: calibration data, configuration, identifier programming. These “factory reset” operations are sometimes triggered by the same register that triggers a soft reset, with a different bit pattern. Use only the soft-reset pattern that the vendor’s driver uses on attach, and do not experiment with bit patterns that you have not seen the vendor’s driver use.

**EEPROM writes.** Similar to flash, EEPROM contains long-term configuration. EEPROM writes are usually orchestrated through a specific protocol of register writes, and the wrong protocol can corrupt EEPROM state. Avoid EEPROM writes during exploration; if you must perform them, do so only with a clear understanding of what value should end up in EEPROM and a way to verify the result.

**Modifying device identifiers.** Some devices store their vendor and device IDs in flash or EEPROM, and a driver bug can overwrite them. A device whose ID has been changed will not be recognised by any driver that matches on the original ID, and recovering it can require physical reprogramming with a hardware programmer. Do not write to any register that is plausibly related to device identification.

**Power management state changes.** Some devices have power states whose transitions are managed by complex sequences of register writes. The wrong sequence can leave the device in a state from which it cannot easily recover, sometimes requiring power cycling the host or even removing the device physically. Power management is one of the most fragile areas of any device’s interface.

**PHY or PLL configuration.** Devices that include their own clock generation (PCI Express devices in particular) have PLL configuration that, if set wrong, can leave the device unable to communicate at all. PHY configuration on networking devices has similar pitfalls. These are subsystems where the vendor’s driver must be followed exactly, because there is no good way to recover from an incorrect setting through any further configuration.

**Bus master enable on a misconfigured DMA address.** If you enable bus mastering on a device that has a DMA address pointing to a wild memory location, the device may read or write that location and corrupt the host’s memory. This is one of the few ways a driver bug can crash the host through hardware action. Always set up valid DMA mappings before enabling bus mastering, and never enable bus mastering if you are not yet ready to manage DMA.

These categories are not exhaustive, but they cover the most common ways to damage a device. The general lesson is: respect the operations whose consequences you do not fully understand, and do not experiment with them.

### 36.13.3 10.3 Soft Resets and Watchdog Timers

The constructive side of the safety discussion is what to do when something does go wrong. The first technique is the soft reset.

A soft reset is the device’s “I have got into a bad state, please restart” mechanism. The pattern is the same as the reset you implemented in Section 6: a write to the control register that puts the device back into its known initial state. Used conservatively, a soft reset can recover from many kinds of problems without requiring intervention.

The pattern in code is straightforward:

```
static void
mydev_recover(struct mydev_softc *sc)
{
 device_printf(sc->sc_dev, "recovering device\n");
 mydev_reset(sc);
 /* Reapply any necessary configuration. */
 mydev_init_after_reset(sc);
}
```

The recovery routine should be called from any code path that detects something has gone wrong: a timeout, an error status, an unexpected register value. The cost of an unnecessary reset is small; the cost of an unrecovered error can be much larger.

A watchdog timer is the next layer of defence. The driver periodically checks that the device is making progress, and if progress has stalled, it triggers a recovery. The pattern is:

```
static void
mydev_watchdog(void *arg)
{
 struct mydev_softc *sc = arg;
 uint32_t counter;

 mtx_lock(&sc->sc_mtx);
 counter = bus_read_4(sc->sc_mem, MYDEV_REG_COUNTER);
 if (counter == sc->sc_last_counter) {
 sc->sc_stall_ticks++;
 if (sc->sc_stall_ticks >= MYDEV_STALL_LIMIT) {
 device_printf(sc->sc_dev, "device stalled, resetting\n");
 mydev_recover(sc);
 sc->sc_stall_ticks = 0;
 }
 } else {
 sc->sc_stall_ticks = 0;
 }
 sc->sc_last_counter = counter;
 callout_reset(&sc->sc_watchdog, hz, mydev_watchdog, sc);
 mtx_unlock(&sc->sc_mtx);
}
```

The watchdog reads a counter that the device should be incrementing during normal operation. If the counter does not change for several iterations, the watchdog assumes the device has stalled and triggers recovery. The pattern is robust against transient slowdowns (a single missed increment is tolerated) but catches genuine stalls within a few seconds.

### 36.13.4 10.4 Backing Up Firmware Before Doing Anything Risky

For devices with firmware (which is most modern devices), the firmware itself should be backed up before any risky operation. If your captures show that the firmware is loaded by the host during initialisation, then the firmware image is presumably already available somewhere as a file, and you can keep a copy. If the firmware lives in flash on the device and is not loaded from the host, you have to read it back through whatever mechanism the device exposes for reading flash, and store the result.

The backup is your insurance against the case where some experiment overwrites the firmware. With the backup, you can restore the original. Without the backup, you may have a brick.

The discipline of backup-before-risky-operation is the same discipline as backup-before-risky-system-administration. The cost is small, the value when it matters is enormous, and the people who skip it are the people who, sooner or later, regret having skipped it.

### 36.13.5 10.5 Read-Only Probes Where Possible

Whenever an experiment can be performed read-only, perform it read-only. The information gained is usually worth far more than the time saved by writing instead of reading. Section 4's register-mapping work was almost entirely read-only for a reason: reading is safe, writing is not.

A useful pattern when you must investigate the effect of writes is to read the value, compute what you expect the new value to be, perform the write, read the value again, and verify that the result matches your expectation. The verification step lets you catch the case where the write did not have the effect you expected, before you base further work on a wrong assumption.

```
static int
mydev_set_field(struct mydev_softc *sc, bus_size_t off,
 uint32_t mask, uint32_t value)
{
 uint32_t old, new, readback;

 old = bus_read_4(sc->sc_mem, off);
 new = (old & ~mask) | (value & mask);
 bus_write_4(sc->sc_mem, off, new);
 readback = bus_read_4(sc->sc_mem, off);
 if ((readback & mask) != (value & mask)) {
 device_printf(sc->sc_dev,
 "set_field off=0x%04jx mask=0x%08x value=0x%08x "
 "readback=0x%08x mismatch\n",
 (uintmax_t)off, mask, value, readback);
 return (EIO);
 }
 return (0);
}
```

The helper performs the read-modify-write-verify cycle in one place, with logging of mismatches. Used consistently, it catches the cases where a write did not stick (because the register is read-only, because the write encoded the value differently than you expected, or because the device's state did not allow the change at this moment) before they cause downstream confusion.

### **36.13.6 10.6 Safe Probing Wrappers**

The companion code in `examples/part-07/ch36-reverse-engineering/lab05-safe-wrapper/` contains a small library of safe probing wrappers that combine several of the techniques discussed here: timeouts on every operation, automatic recovery on detected stalls, read-only modes, and per-operation logging. The wrappers add a few hundred microseconds to each operation, which is irrelevant during exploration, and they catch the great majority of safety problems before the problems become harmful.

The pattern is recommended for all exploratory drivers. The production driver, when it eventually exists, can drop the wrappers in favour of more efficient direct accesses, but during exploration the wrappers are worth their cost in safety.

### **36.13.7 10.7 Recognising When to Stop**

The hardest safety decision is sometimes the decision to stop. When an experiment is not behaving as expected, when the device is producing values you cannot explain, when your hypotheses are all wrong and you do not yet have new ones, the temptation is to keep poking until something becomes clear. Resist this temptation. Stop, take a break, look at the captures again, talk to someone else about what you have observed, and let the situation clarify before you resume.

The reason is that the moments of confusion are exactly the moments when accidental damage is most likely. A clear mind is careful about which writes are safe; a frustrated mind tries things that look promising and discovers, too late, that one of them was destructive.

A short break also lets your subconscious work. Many of the most useful reverse-engineering insights arrive while you are doing something else: walking, sleeping, working on an unrelated problem. The brain has a remarkable ability to integrate observations into coherent understanding when given a chance. Forcing yourself to stay at the keyboard when the work has stalled often produces nothing useful and sometimes produces disasters.

### **36.13.8 10.8 Wrapping Up Section 10**

Safety in reverse engineering is built on a small set of disciplines: never write without strong evidence, recognise the operations that deserve particular caution, build in soft-reset recovery and watchdog timers, back up firmware before risky operations, prefer read-only experiments, use safe probing wrappers, and know when to stop. None of these disciplines is complicated. All of them are necessary. A reverse engineer who follows them is unlikely to damage hardware. A reverse engineer who ignores them is, sooner or later, certain to.

The next section closes the technical part of the chapter by discussing how to turn the body of work the previous sections have built up into a maintainable driver and a usable pseudo-datasheet. The reverse-engineering process that started with observations and hypotheses ends with a driver and a document, and how those final artifacts are constructed determines how useful the project will be to its future readers.

## 36.14 11. Refactoring and Documenting the Reversed Device

The end of a reverse-engineering project is not the moment when the driver works. It is the moment when the driver and the pseudo-datasheet together can be handed to another engineer who can read them and understand both the device and the implementation. Until that moment, the project is incomplete, even if the driver appears to function. This section walks through the work of consolidating the project's findings into a maintainable form.

The work has two parts. The driver itself needs to be cleaned up, restructured to follow normal driver-writing conventions, and documented in the way any FreeBSD driver should be documented. The pseudo-datasheet, which has been growing throughout the project as a notebook of facts, needs to be reorganised into a standalone document that someone who has never seen the project can read and learn from.

### 36.14.1 11.1 The Driver Cleanup

A reverse-engineered driver, in its working-but-not-cleaned-up state, typically carries traces of its history: comments referring to “the suspected control register”, code blocks that test multiple hypotheses with conditional compilation, debugging sysctls that were added during a specific investigation, log messages that were useful at the time but no longer needed. Cleaning up means removing the historical noise while preserving the substantive content.

The cleanup checklist:

1. Remove every conditional compilation block that was used to test alternatives. The chosen alternative should remain; the alternatives should be deleted.
2. Replace speculative names with confirmed names. If a register was originally named `MY_DEV_REG_UNKNOWN_40` and is now known to be the channel-selection register, rename it to `MY_DEV_REG_CHANNEL`.
3. Replace investigation comments with explanatory comments. A comment that says “I think this might be the polling delay” is a hypothesis comment from the investigation phase; replace it with a comment that says “Wait for the device to complete the reset” once the hypothesis is confirmed.
4. Remove debugging sysctls that are no longer useful, and keep the ones that operators or maintainers will want.
5. Remove `device_printf` calls that were valuable during investigation but are now noise in production.
6. Verify that the detach path is clean. A driver that loads cleanly but fails to unload cleanly is a maintenance problem.
7. Verify that all error paths free their resources correctly. A driver that leaks resources on error is a problem that will eventually surface.

The companion code in `examples/part-07/ch36-reverse-engineering/` includes both the investigation-phase scaffolding and the cleaned-up form, so that you can see the difference. The cleaned-up form is what would be suitable for inclusion in the FreeBSD source tree (we will discuss inclusion in Chapter 37).

### 36.14.2 11.2 Driver Documentation

A FreeBSD driver should have at least a manual page in section 4, and ideally a section in the kernel-developer's documentation as well. The manual page is for users who want to know what the driver does and how to

configure it. The developer documentation is for kernel hackers who want to understand the driver internally.

The manual page follows the standard FreeBSD style. The relevant manual page for `style(4)` documents the conventions, and the manual pages for similar drivers are good examples to follow. A typical driver manual page covers:

- The name of the driver and a one-line description.
- The synopsis showing how to load the driver in `loader.conf`.
- The description, explaining what hardware the driver supports and what features it provides.
- The hardware support section, listing the specific devices the driver works with.
- The configuration section, describing any sysctls or loader variables the driver exposes.
- The diagnostics section, listing the messages the driver may produce and what they mean.
- The see-also section, with cross-references.
- The history section, explaining when the driver appeared.
- The author section.

For a reverse-engineered driver, the description should be honest about the driver's provenance: it was developed by reverse engineering, the supported feature set is what was achievable through that process, and certain corners of the device may behave differently than the driver expects. Operators appreciate honest documentation; vague claims of full support set up users for confusion when they encounter unimplemented behaviour.

### 36.14.3 11.3 The Pseudo-Datasheet Structure

The pseudo-datasheet is the standalone document that captures everything you have learned about the device, in a form that someone who has never seen your driver can read and understand. A well-structured pseudo-datasheet often becomes the most referenced document in any project that uses the device, because it answers questions that the driver source code cannot answer without close reading.

A practical pseudo-datasheet structure looks like this:

1. Identity
  - 1.1 Vendor and device IDs
  - 1.2 USB descriptors (if applicable)
  - 1.3 Class codes (if applicable)
  - 1.4 Subsystem identifiers (if applicable)
  - 1.5 Hardware revisions covered
2. Provenance
  - 2.1 Sources consulted
  - 2.2 Methodology used
  - 2.3 Verification status of each fact (high / medium / low)
3. Resources
  - 3.1 Memory regions and their sizes
  - 3.2 I/O ports (if applicable)
  - 3.3 Interrupt assignment

4. Register Map
  - 4.1 Register list with offsets and short descriptions
  - 4.2 Per-register details: size, access type, reset value, fields
5. Buffer Layouts
  - 5.1 Ring buffer layouts
  - 5.2 Descriptor formats
  - 5.3 Packet formats
6. Command Interface
  - 6.1 Command sequencing
  - 6.2 Command list with formats and responses
  - 6.3 Error reporting
7. Initialization
  - 7.1 Cold attach sequence
  - 7.2 Warm reset sequence
  - 7.3 Required register settings before operation
8. Operating Patterns
  - 8.1 Data flow
  - 8.2 Interrupt handling
  - 8.3 Status polling
9. Quirks and Errata
  - 9.1 Known bugs in the device
  - 9.2 Workarounds in the driver
  - 9.3 Edge cases that have not been fully characterised
10. Open Questions
  - 10.1 Behaviours observed but not understood
  - 10.2 Registers whose purpose is not yet known
  - 10.3 Operations not yet investigated
11. References
  - 11.1 Prior work consulted
  - 11.2 Related drivers in other operating systems
  - 11.3 Public documentation, if any

The structure is comprehensive, and not every project will fill every section. Sections that are not relevant can be omitted; the template is a checklist of “what would be worth documenting if the relevant information existed”, not a requirement to invent information that does not exist.

The most important sections are the Provenance, the Register Map, and the Open Questions. Provenance lets readers evaluate the trustworthiness of the document. The Register Map is the central reference. Open Questions

tells future contributors where the work needs more attention.

### 36.14.4 11.4 The Provenance Section in Detail

The Provenance section deserves special attention because it is how a pseudo-datasheet establishes its credibility. The section should answer:

- What sources of information were used? (Captures, prior code, experiments, public documentation.)
- What methodology was applied to each source? (Direct reading, diffing, statistical analysis.)
- Which facts come from which sources?
- For each fact, what is the verification status?

A useful convention is to label each substantive fact in the register map and elsewhere with a short verification tag:

- **HIGH**: confirmed by multiple independent observations and by experiment.
- **MEDIUM**: confirmed by a single source or a single experiment.
- **LOW**: hypothesis based on inference, not yet directly tested.
- **UNKNOWN**: stated for completeness, but with no evidence.

Readers can then weight each fact by its verification status, and contributors can prioritise where to invest more investigation. The convention costs little to maintain and gives the document a much more honest character than a flat statement of facts that treats all claims as equal.

### 36.14.5 11.5 Registers as Tables

The register map itself is best presented as a table. For each register, the table should give:

- Offset within the register block.
- Symbolic name (the macro name in the driver).
- Size (8, 16, 32, or 64 bits, usually).
- Access type (read-only, read-write, write-only, write-1-to-clear, and so on).
- Reset value.
- One-line description.

A separate, more detailed entry for each register lists the fields within the register. For example:

MYDEV\_REG\_CONTROL (offset 0x10, RW, 32 bits, reset 0x00000000)

| Bits  | Name       | Description                                  |
|-------|------------|----------------------------------------------|
| ----- |            |                                              |
| 0     | RESET      | Write 1 to trigger reset.                    |
| 1     | ENABLE     | Set to enable normal operation.              |
| 2-3   | MODE       | Operating mode (0=idle, 1=rx, 2=tx, 3=both). |
| 4     | INT_ENABLE | Enable interrupts globally.                  |
| 31:5  | reserved   | Read as zero, write as zero.                 |



This table format is the FreeBSD convention. It scales well, it is easy to maintain in Markdown, and it is what readers expect.

### 36.14.6 11.6 Pseudo-Datasheet as a Living Document

The pseudo-datasheet is rarely complete. As the driver evolves, new behaviour is discovered, old facts are refined, edge cases are characterised. The Git-tracked Markdown form lets the document evolve naturally, with each commit explaining what was learned and when.

The discipline that supports this is to update the pseudo-datasheet first, then update the driver to match. The pseudo-datasheet is the specification; the driver is the implementation. When you discover that a register's bit-3 has a purpose you did not previously know, write the new bit-3 description into the pseudo-datasheet first, with provenance, and then update the driver to use it. The order matters because it forces you to think about the change as a fact about the device, separately from a change to your code.

Over time, the pseudo-datasheet becomes the trustworthy artifact, and the driver becomes one of several possible implementations of the contract the document specifies. New implementations (in NetBSD, in Linux, in custom embedded code) can be written from the pseudo-datasheet alone, without re-deriving everything from scratch.

### 36.14.7 11.7 Versioning the Driver

The book's worked examples use version strings such as `v2.5-async` to mark major iterations of a driver. For reverse-engineered drivers, a useful convention is to use a `-rev` suffix to indicate the reverse-engineered nature of the work, with version numbers tracking the maturity of the implementation:

- `v0.1-rev`: minimal driver, reset and identifier only.
- `v0.2-rev`: read path implemented and verified.
- `v0.5-rev`: most operations implemented, some quirks understood.
- `v1.0-rev`: full functionality, all known quirks handled.
- `v2.1-rev`: a stable, mature, refactored driver suitable for general use.

The version string can appear in the manual page, in a `MODULE_VERSION` macro, and in the pseudo-datasheet. It tells operators what level of support to expect from any given build.

### 36.14.8 11.8 Wrapping Up Section 11

We have seen how to consolidate the work of a reverse-engineering project into a maintainable driver and a standalone pseudo-datasheet. The driver is cleaned up, documented in the manual page, and versioned to indicate its maturity. The pseudo-datasheet captures what was learned about the device, with provenance, in a structured form that future contributors can extend. Together, the two artifacts are what justifies the considerable investment that the reverse-engineering project required.

Before turning to hands-on practice, two shorter sections round out the theoretical material. Section 12 revisits the legal and ethical framework from Section 1 with a practical eye, giving you a compact set of rules for licence compatibility, contractual restrictions, clean-room discipline, and safe-harbour activities. Section 13 then walks through two worked case studies from the FreeBSD tree itself, showing how the techniques of the chapter appear in drivers that are in production today. After those two sections, the remaining material of the chapter gives you

labs to apply what you have learned, challenge exercises to stretch your understanding, troubleshooting notes to help when things go wrong, and a forward-looking transition to Chapter 37, where we will see how to take a driver like the one you have just built and submit it for inclusion in the FreeBSD source tree.

## 36.15 12. Legal and Ethical Guardrails in Practice

Section 1 opened this chapter by sketching the legal and ethical landscape in which reverse engineering for FreeBSD happens. That sketch was deliberately broad, because it had to introduce concepts like fair use, interoperability research, and clean-room method before the reader had seen any of the technical work the rest of the chapter covers. Now, at the end of that technical work, it is worth taking a second, more practical pass. The goal of this section is not to turn you into a lawyer. It is to give you a small set of habits that protect you, the project, and the reader of your code from the predictable ways a reverse-engineering effort can go wrong. Each habit is concrete, each one can be documented inside your pseudo-datasheet from Section 11, and each one is directly informed by how the FreeBSD tree has handled similar questions in the past.

### 36.15.1 12.1 Why a Second Legal Section

Section 1 answered the question “is this allowed?”. Section 12 answers the question “how do I do it in a way that stays allowed?”. The distinction matters, because many of the activities that are legal in principle become risky in practice if they are done without structure. Reading another driver to understand a chip is legal. Reading another driver and then writing your own from memory, without any intermediate document to show where your understanding came from, looks the same from the outside but is much harder to defend if a question is ever raised. Treating a proprietary datasheet as a reference is legal. Quoting long passages from that datasheet into your driver’s source comments is not. The difference in both cases is process, not intent.

The rest of this section works through four concrete areas: licence compatibility (what you are allowed to copy), contractual restrictions (what you are allowed to disclose), clean-room practice (how to keep interoperability work defensible), and safe-harbour activities (what is always permitted). None of the content below is a substitute for legal advice on a specific situation; it is instead the common discipline that experienced FreeBSD developers already follow, written down in one place so that a new author can adopt it without having to reconstruct it.

### 36.15.2 12.2 Licence Compatibility

The FreeBSD kernel uses a permissive BSD-style licence. When you pull knowledge from another codebase into your driver, the licence of that other codebase constrains what you are allowed to copy, even if it does not constrain what you are allowed to learn. The categories that come up in practice are BSD, GPL, CDDL, and proprietary, and each has a different rule.

**BSD-licensed sources.** Drivers that already live in the OpenBSD or NetBSD trees, or in older FreeBSD ports of the same device, use a permissive licence that is compatible with the FreeBSD kernel. You can copy code directly, with appropriate attribution in the copyright block. The `zyd` wireless driver at `/usr/src/sys/dev/usb/wlan/if_zyd.c` is a concrete example: the file header preserves the original `$OpenBSD$` and `$NetBSD$` tags at the top of the source, indicating that the code was ported from those trees, and Damien Bergamini’s BSD-style copyright is intact alongside the copyrights of later FreeBSD contributors. When the licence is compatible,

moving code between trees is a legitimate way to amortise work across the BSD ecosystem, and it is already part of FreeBSD's routine practice.

**GPL-licensed sources.** Linux drivers are almost always licensed under the GPL. You are allowed to read GPL code to understand how a device works, because reading is not copying. You are not allowed to paste GPL code into a BSD-licensed driver, even in small amounts, even temporarily, even with attribution. The FreeBSD project's position is clear: the kernel does not accept code of incompatible licence. A driver that was reviewed and found to contain copied GPL code would be rejected, and the maintainer would lose standing in future reviews. The rule is strict because the cost of relaxing it would be the licence integrity of the tree.

**CDDL-licensed sources.** Some device drivers in the OpenSolaris and Illumos trees are released under the CDDL. The CDDL is file-level copyleft, which means that you cannot freely intermix CDDL source and BSD source within the same file. The ZFS code in FreeBSD is a well-known accommodation of this rule: CDDL files are kept in their own directory under `/usr/src/sys/contrib/openzfs/`, their licence header is preserved, and BSD-licensed glue code around them sits in separate files. For a device driver that structure is rarely a reasonable workflow, so the safer rule is that CDDL code is look but do not copy in the same way as GPL code.

**Proprietary sources.** Vendor SDKs, reference drivers, and sample code are usually distributed under a proprietary licence that forbids redistribution. Even when the vendor encourages you to use the code as a reference, that permission is not transferable to the FreeBSD source tree, because the vendor cannot speak for every downstream user of FreeBSD. Reading a proprietary driver to understand a device is generally acceptable to a limited extent, but you cannot paste from it, and in many jurisdictions you cannot quote from it at length either.

The practical rule is straightforward: treat anything that is not plain BSD as read-only. If you want bytes in your driver, those bytes must either come from your own typing or from a source whose licence the FreeBSD project already accepts. When you make this rule explicit in your pseudo-datasheet, by writing next to each register description where the information was observed, you create a record that will still be useful years later when a new maintainer needs to know how the file came to look the way it does.

### 36.15.3 12.3 Contractual Restrictions

Contracts can bind you in situations where copyright law does not. The three kinds of contract that come up in reverse-engineering work are non-disclosure agreements, end-user licence agreements, and firmware licences. Each of them constrains a different part of the workflow.

**Non-disclosure agreements.** Some vendors will share detailed documentation in exchange for a signed NDA. The documentation is often excellent, but the NDA typically forbids publication of the information, sometimes indefinitely, and sometimes with liquidated damages attached. Signing an NDA about a device and then writing an open-source driver for that same device is a legal minefield: every register you document in a pseudo-datasheet, every field name, every value, could be challenged later as a disclosure. The safer choice is almost always to refuse the NDA and work from publicly observable information. If you have already signed one and then want to contribute an open driver, the clean option is to recuse yourself from the observation work and let a second author build the pseudo-datasheet from scratch, using only sources you never touched under the agreement.

**End-user licence agreements.** Many vendor-supplied drivers and SDKs include an EULA that explicitly forbids reverse engineering. The enforceability of such clauses varies across jurisdictions, but the safer path is to avoid clicking "I agree" in the first place. If you only ever touched the vendor driver as a binary shipped on an

installation image, without accepting an online agreement, your position is stronger than if you signed up for a developer programme and downloaded the SDK under an agreement that forbade disassembly. Record which vendor materials you consulted and under what terms. That record becomes part of the provenance field in your pseudo-datasheet.

**Firmware licences.** Many modern devices need a binary firmware blob that the driver loads at attach time. The vendor usually distributes that blob under a redistribution licence that is more permissive than the surrounding driver code but still not fully free. FreeBSD's ports tree has a well-established pattern of packaging firmware blobs in dedicated `-firmware-kmod` ports so that the kernel code stays BSD while the blob keeps its vendor licence. You do not need to reverse-engineer the firmware itself; you need to understand how the driver hands it to the device. The legal framework for firmware is separate from the legal framework for the driver, and both have to be satisfied independently.

### 36.15.4 12.4 Clean-Room Practice

Reading a GPL driver to learn what a device does, and then writing BSD code that does the same thing, is a recognised interoperability technique. United States courts have upheld it in cases such as *Sega v. Accolade* and *Sony v. Connectix*, and Article 6 of the European Union's Software Directive (2009/24/EC) carves out reverse engineering specifically for interoperability. But the legal protection depends on how you do the work, not just on what you do. A driver that looks line-for-line like the Linux original will not be protected by the clean-room defence, because the record will show that no clean room existed.

Good clean-room practice has two elements: structural separation and structural documentation.

Structural separation means that the person who reads the other-platform driver, and the person who writes the FreeBSD driver, are different activities. On a team of two, they are literally different people. On a solo project, they are different work sessions with different artifacts. The reader's output is a pseudo-datasheet: a document in plain English that describes registers, bit fields, command sequences, and quirks, with no reference to the specific identifiers the other driver used. The writer's input is that pseudo-datasheet, plus the FreeBSD header files and driver examples. The writer never opens the other-platform driver while the FreeBSD code is being written.

Structural documentation means that the pseudo-datasheet records where each fact came from. A register description might be annotated "observed on wire with `usbdump(8)` on 2026-02-14" or "inferred from vendor datasheet section 3.4" or "deduced from Linux driver, reread into prose". That annotation is the record that would, if ever needed, support the claim that your driver was written from understanding rather than from copying. Patent and copyright cases have sometimes turned on the quality of exactly this kind of record, and maintaining it is not paranoia; it is the same discipline that keeps the rest of your engineering honest.

In practice, the time cost of clean-room discipline is small once the pseudo-datasheet habit from Section 11 is already part of your workflow. The legal value is large, because it moves you from "I hope this is not a problem" to "I can show how it was done".

### 36.15.5 12.5 Safe Harbour

It helps to know what is always safe, so that you can push the uncertain cases through the structure of this section without second-guessing every keystroke. The following activities are defensible in every jurisdiction the FreeBSD project cares about, and you can perform them without hesitation.

**Reading code is always safe.** Whatever licence the other-platform driver carries, reading it to learn how a device works is not a copyright violation. It is sometimes called fair use, sometimes called interoperability research; the label depends on the jurisdiction, but the principle is stable in every legal system that the FreeBSD project is likely to encounter.

**Observing your own hardware is always safe.** Running a device you own on a machine you own and recording what happens on the bus is not actionable under copyright, contract, or trade-secret law. The recordings you produce from `usbdump(8)`, from `pciconf(8)`, from JTAG probes, and from logic analysers are your own work, and you may describe and publish them freely.

**Writing from understanding is always safe.** If you can describe the device in your own words, in a pseudo-datasheet, you can then write a FreeBSD driver from that document without restriction. The writing is yours, derived from facts about the device rather than from the expression of another author.

**Publishing interoperability information is generally safe.** Documenting a register layout, a command sequence, or a wire protocol is publishing facts about a device, not publishing someone else's code. Even when a vendor dislikes the publication, they rarely have a legal basis to prevent it, and courts in both the United States and the European Union have consistently supported interoperability research.

Activities outside this list are not automatically unsafe, but they deserve deliberate thought. When in doubt, widen your clean room, strengthen your documentation, and ask for a second opinion on the `freebsd-hackers` mailing list or in a project chat channel before committing. The maintainers have seen these questions many times, and the cost of asking is low.

### 36.15.6 12.6 Wrapping Up Section 12

We have taken a practical second pass over the legal and ethical framework introduced in Section 1. Licence compatibility dictates what you can copy; contractual restrictions dictate what you can disclose; clean-room practice keeps interoperability research defensible; and a short list of always-safe activities gives you room to work without continuous worry. The habits are simple to adopt once the pseudo-datasheet from Section 11 is already part of your workflow, because the pseudo-datasheet is itself the audit trail of the clean room.

With this foundation in place, we are ready to look at how the FreeBSD tree itself documents reverse-engineering work. The next section presents two worked case studies, both drawn directly from the current source, where the driver authors recorded their reasoning in comments that are still readable today.

## 36.16 13. Case Studies From the FreeBSD Tree

We have now covered the techniques, the documentation discipline, and the legal framework. It is time to look at two real drivers that were written under exactly these constraints and see how their authors handled the work. Both drivers are in the FreeBSD 14.3 tree right now. Both have header comments and inline notes that record, in the author's own words, what the datasheet failed to say and what the driver does about it. Reading those comments gives you a direct view of the reverse-engineering discipline as it appears in production code, with no retouching and no retrospective polish.

### 36.16.1 13.1 How to Read These Case Studies

For each driver we will look at four things. First, a short description of the device and its history, so that you know what kind of hardware is involved and roughly when the work was done. Second, the approach the author used to work around missing documentation, including what sources were consulted and what observations were made. Third, the specific code that encodes the finding, with exact file paths, function names, and the register or constant identifiers that the driver uses today. Fourth, the ethical and legal context in which the work was done, including how the licence of each source shaped what the author was allowed to write. A closing paragraph then translates the historical method into its modern form, so that you can see how you would approach the same problem today with the techniques from earlier in this chapter.

None of the history below is speculative. Every claim is anchored in a file you can open on your own FreeBSD system, and every fact about the code is directly observable in the current source. If anything here drifts out of date in a future release, the files themselves will still be the source of truth, and the method of reading them will still apply.

### 36.16.2 13.2 Case Study 1: The umcs Driver and Its Undocumented GPIO

**Device and history.** The MosChip MCS7820 and MCS7840 are USB-to-serial bridge chips that appear in inexpensive multi-port RS-232 adapters. The MCS7820 is a two-port part, the MCS7840 is a four-port part, and the USB interface is electrically the same in both cases. The FreeBSD driver, `umcs`, was written by Lev Serebryakov in 2010 and now lives at `/usr/src/sys/dev/usb/serial/umcs.c`. The companion header, `/usr/src/sys/dev/usb/serial/umcs.h`, spells out the register layout.

**Approach.** The header comment at the top of `umcs.c` is unusually candid about the documentation situation. The author writes that the driver supports the `mos7820` and `mos7840` parts, and notes directly that the public datasheet does not contain full programming information for the chip. A full datasheet, distributed under restriction by MosChip technical support, filled in some of the gaps, and a vendor-supplied reference driver filled in the rest. The author's task was to write an original BSD driver that behaved the same way the confirmed information said it should, with the vendor driver used as an observational check rather than as a source to copy.

The cleanest place to see this discipline at work is the port-count detection inside the `attach` routine. A program that drives a USB-to-serial chip must know whether the chip has two ports or four, because the user-visible device nodes and the internal data structures depend on that count. The datasheet prescribes one method, the vendor driver uses another, and experiments on real hardware show that the datasheet's method is unreliable.

**Code.** In `/usr/src/sys/dev/usb/serial/umcs.c`, the function `umcs7840_attach` performs the detection. The inline comment records the problem in plain text, and the code records the chosen workaround. The relevant fragment reads:

```
/*
 * Documentation (full datasheet) says, that number of ports
 * is set as MCS7840_DEV_MODE_SELECT24S bit in MODE R/Only
 * register. But vendor driver uses these undocumented
 * register & bit. Experiments show, that MODE register can
 * have `0' (4 ports) bit on 2-port device, so use vendor
```

```

* driver's way.
*/
umcs7840_get_reg_sync(sc, MCS7840_DEV_REG_GPIO, &data);
if (data & MCS7840_DEV_GPIO_4PORTS) {

```

The register involved is `MCS7840_DEV_REG_GPIO`, defined at offset `0x07` in `/usr/src/sys/dev/usb/serial/umcs.h`. The header is equally candid about its status. The register block as a whole is annotated with a note explaining that the registers are documented only in the full datasheet, which can be requested from MosChip technical support, and the GPIO register specifically is commented as holding the `GPIO_0` and `GPIO_1` bits that are undocumented in the public datasheet. A longer note further down the header explains that `GPIO_0` must be grounded on two-port boards and pulled up on four-port boards, and that the convention is enforced by board designers rather than by the chip itself. The single-bit flag `MCS7840_DEV_GPIO_4PORTS`, defined as `0x01`, is what the attach routine tests against the value returned by `umcs7840_get_reg_sync`.

What makes this a good study is that the code records the story. A reader who finds the driver fifteen years after it was written can reconstruct the reasoning: the datasheet said one thing, the vendor driver did another, the real hardware was tested, and the chosen method is explained in the comment. No one has to rediscover the same dead end.

**Ethical and legal context.** The author had two classes of source available: the restricted full datasheet, which MosChip distributes on request, and the vendor reference driver, which ships with the chip evaluation kit. Both are proprietary in the sense that they are not freely redistributable, and neither can be copied into an open-source driver. What you can do is learn from them, and then write your own code against the information they contain. The `umcs` comment does exactly that. It names the behaviour the vendor driver exhibits, it names the experiment that confirmed the behaviour on real hardware, and the code it produces is original BSD-licensed prose. The copyright block at the top of the source file identifies Lev Serebryakov as the sole author. This is a textbook clean-room outcome: the driver benefits from vendor information without importing vendor code.

**Modern replication.** If you were writing `umcs` today, the workflow would be the same, with better tooling. You would capture a `usbdump(8)` trace of the vendor driver's attach sequence to see the exact GPIO read, you would run the same read on your own hardware across known two-port and four-port boards, and you would record the discrepancy between the datasheet and the observed behaviour in your pseudo-datasheet with a clear provenance line for each source. The driver would then implement the observed method, with a comment identical in spirit to the one Lev Serebryakov wrote. The technique has not changed; the tools around it have improved.

### 36.16.3 13.3 Case Study 2: The axe Driver and the Missing IPG Initialisation

**Device and history.** The ASIX AX88172 is a USB 1.1 Ethernet adapter chip, and the AX88178 and AX88772 are its USB 2.0 descendants. Cheap Ethernet dongles based on these parts have been common since the early 2000s, and the FreeBSD driver, `axe`, lives at `/usr/src/sys/dev/usb/net/if_axe.c`. The original code was written by Bill Paul between 1997 and 2003, and the AX88178 and AX88772 support was backported from OpenBSD by J. R. Oldroyd in 2007. The register header lives alongside the driver at `/usr/src/sys/dev/usb/net/if_axereg.h`.

**Approach.** The top-of-file comment block in `/usr/src/sys/dev/usb/net/if_axe.c` reads, in part, that there is information missing from the chip manual which the driver must know in order for the chip to function

at all. The author lists two specific facts. A bit must be set in the RX control register or the chip will receive no packets at all. The three inter-packet-gap registers must all be initialised, or the chip will send no packets at all. Neither of these requirements appears in the public datasheet, and both were established by reading the vendor's Linux driver and observing the chip on real hardware.

The story of the IPG registers is especially clear. Inter-packet gap is a standard Ethernet concept: the transmitter must wait a minimum number of bit times between frames, and the exact number depends on the link speed. A silicon designer has several ways to expose IPG to software, and the AX88172 chose to expose three distinct registers that the driver must write during initialisation. The datasheet names the registers but says nothing about the need to program them, so a naïve driver following only the datasheet would leave them at their reset values and find that the chip, mysteriously, refuses to transmit.

**Code.** The initialisation sequence appears inside the function `axe_init` in `/usr/src/sys/dev/usb/net/if_axe.c`. A short helper called `axe_cmd` provides the standard way to issue a vendor-specific USB control request, and `axe_init` calls it once for each IPG write. The relevant branch reads:

```
if (AXE_IS_178_FAMILY(sc)) {
 axe_cmd(sc, AXE_178_CMD_WRITE_NODEID, 0, 0, if_getlladdr(ifp));
 axe_cmd(sc, AXE_178_CMD_WRITE_IPG012, sc->sc_ipgs[2],
 (sc->sc_ipgs[1] << 8) | (sc->sc_ipgs[0]), NULL);
} else {
 axe_cmd(sc, AXE_172_CMD_WRITE_NODEID, 0, 0, if_getlladdr(ifp));
 axe_cmd(sc, AXE_172_CMD_WRITE_IPG0, 0, sc->sc_ipgs[0], NULL);
 axe_cmd(sc, AXE_172_CMD_WRITE_IPG1, 0, sc->sc_ipgs[1], NULL);
 axe_cmd(sc, AXE_172_CMD_WRITE_IPG2, 0, sc->sc_ipgs[2], NULL);
}
```

The two branches expose a second piece of reverse-engineering information that is invisible in the datasheet. On the older AX88172, each of the three IPG values is programmed as an independent command: `AXE_172_CMD_WRITE_IPG0`, `AXE_172_CMD_WRITE_IPG1`, and `AXE_172_CMD_WRITE_IPG2`, defined in `/usr/src/sys/dev/usb/net/if_axereg.h`. On the newer AX88178 and AX88772, a single command writes all three at once by packing them into the same control request: `AXE_178_CMD_WRITE_IPG012`. The two opcodes have the same numeric value; the AX88178 reused the opcode slot that the AX88172 used for writing a single IPG register, and expanded its semantics to cover all three. A driver that treated the two chip families as interchangeable would corrupt the IPG programming on one of them. The only way to distinguish the families is the macro `AXE_IS_178_FAMILY(sc)`, and the need for that macro is itself a finding of the reverse-engineering work.

**Ethical and legal context.** The `axe` driver has a long history, and its provenance is recorded in the copyright block. Bill Paul wrote the original driver against the AX88172, and the comments show that his information came from a mix of the public datasheet and observational work. J. R. Oldroyd's AX88178 support was ported from OpenBSD's sibling driver, which is BSD-licensed and directly compatible with FreeBSD. That port was legally straightforward: the code came from a permissive source tree, and the copyright block was preserved across the move. The reverse-engineering notes about IPG and RX-control were carried along with the code, so the finding itself is documented forever in the driver's header.

What did not happen is as informative as what did. The `axe` driver does not import code from the Linux `asix_usb` driver, even though that driver was available when Bill Paul was writing the FreeBSD equivalent. Linux uses



the GPL, and pasting from it would have made `axe` unshippable under the kernel's licence rules. Reading it for understanding is exactly what Bill Paul did, and the FreeBSD driver is his own writing throughout.

**Modern replication.** If you were writing `axe` today, you would start by loading the Linux driver on a Linux test machine and running `usbdump(8)` on a FreeBSD machine that shared the same USB hub, so that you could capture the vendor driver's attach sequence without needing to read a single line of GPL code. The `usbdump(8)` output would show the three IPG writes directly, because they travel on the wire as visible USB control transfers. You would record the observation in your pseudo-datasheet, cite the packet-capture file for provenance, and implement the writes in your own driver. The comment above `axe_init` would read very much like the one Bill Paul originally wrote.

### 36.16.4 13.4 Shared Lessons

Reading the two drivers together surfaces a handful of lessons that are worth naming explicitly.

**The comment is the audit trail.** In both `umcs` and `axe`, the header and inline comments are the place where the reverse-engineering story is preserved. Without those comments, a future maintainer staring at `MCS7840_DEV_REG_GPIO` or at `AXE_178_CMD_WRITE_IPG012` would have no way to know where the information came from or why the driver does what it does. In both cases the author took the extra minute to write down the reasoning, and the result is a driver that remains maintainable fifteen or twenty years after it was written. The author's pseudo-datasheet, in each case, was partly absorbed into the comments and partly preserved in the structure of the header file. That absorption is what makes the code readable today.

**Observation wins over documentation.** Both drivers trust observation over the datasheet when the two disagree. `umcs` trusts the experimentally verified GPIO read over the datasheet's MODE-register method, and `axe` trusts the observation that the chip refuses to transmit without IPG programming over the datasheet's silence on the subject. This is not a rejection of documentation; both authors read the datasheets carefully. It is a recognition that a datasheet is a description of the intended behaviour, and the chip's actual behaviour is what the driver has to match.

**Licence discipline is not an afterthought.** Both drivers respect the licence of the sources they drew on. `umcs` uses a proprietary reference driver as an observational check without copying from it. `axe` imports AX88178 and AX88772 support from OpenBSD, which is licence-compatible, and stays clear of the Linux `asix_usb` driver, which is not. Neither driver includes code that the FreeBSD project would have had to strip out during review, and neither needed a lawyer's review before it was committed. The habits from Section 12 were already part of how the authors worked.

**The work is reproducible.** Everything both drivers did can be reproduced today with modern tools. `usbdump(8)`, `usbconfig(8)`, and the bus-space tracing techniques from earlier in the chapter would give you the same information that the original authors had to gather by hand, and the pseudo-datasheet habit from Section 11 would give you a cleaner audit trail than a set of inline comments. The drivers you write for obscure hardware today will be better documented than the drivers from 2007 and 2010, because the techniques have matured. The spirit of the work, however, is the same.

### 36.16.5 13.5 Wrapping Up Section 13

Two case studies have shown what a disciplined reverse-engineering effort looks like when it has been absorbed into the FreeBSD tree. In both cases the author identified a specific gap in the public documentation, confirmed

the correct behaviour through observation, recorded the finding in a comment that is still readable today, and wrote original code under a licence compatible with FreeBSD. The drivers work, the code is maintainable, and the legal record is clean.

With the legal framework from Section 12 and the worked examples from Section 13 in place, you now have both the principles and a set of precedents to draw on. The remaining sections of the chapter give you hands-on practice with the techniques that the case studies illustrate, challenge exercises to push your understanding further, troubleshooting notes for the common ways the work can go wrong, and a closing bridge to Chapter 37.

## 36.17 Hands-On Labs

The labs in this chapter give you safe, repeatable practice with the techniques the chapter has covered. None of them touches real unknown hardware in a way that could damage anything; the “unknown device” in the labs is either a software mock or a benign region of memory that you control. Treat each lab as a chance to internalise one specific technique before adding it to your kit.

The companion code lives in `examples/part-07/ch36-reverse-engineering/`. Each lab subfolder has its own `README.md` with step-by-step instructions, and the code is organised so that you can build and run each lab independently of the others. As with earlier chapters, type the code yourself the first time you work through a lab; the companion files are there as a reference and as a known-good version to compare against.

### 36.17.1 Lab 1: Identifying a Device and Dumping Descriptors

This lab is the simplest possible reverse-engineering exercise. You will use `pciconf(8)` and `usbconfig(8)` to enumerate every device on your FreeBSD system and to dump the static descriptors of one device of your choice. The output of the lab is a small text file that records, for one specific device, every public fact that the kernel can tell you about it.

Steps:

1. Run `sudo pciconf -lvc` and save the output. Note any device that appears as `noneN@...`, indicating that no in-tree driver has claimed it.
2. Run `sudo usbconfig` and save the output. Pick one USB device that you understand (a USB stick, for example) and that you own physically.
3. Run `sudo usbconfig -d ugen0.X dump_device_desc` and `sudo usbconfig -d ugen0.X dump_curr_config_desc` for the chosen device, where `ugen0.X` is the device’s `usbconfig` identifier.
4. Open the captured output and identify each field: `bDeviceClass`, `idVendor`, `idProduct`, `bMaxPacketSize0`, `bNumConfigurations`, the endpoint descriptors, and so on.
5. Write a one-page summary identifying the device’s identity, its class, its endpoints, and any notable features.

The lab is intentionally easy. The point is to internalise the shape of the static-identity capture before applying it to a device whose identity is genuinely unknown.

### 36.17.2 Lab 2: Capturing a USB Initialization Sequence

This lab moves from static identity to dynamic behaviour. You will use `usbdump(8)` to capture the initialisation sequence of a USB device, save the capture, and explore it in Wireshark.

Steps:

1. Identify a USB device that you can attach and detach freely (a USB stick is a good choice, because you can unplug and replug it without consequence).
2. Run `sudo usbdump -i usb0 -w stick-init.pcap` to start capturing on the USB bus the device is attached to. Use the correct bus number for your system; `usbconfig` will tell you which bus the device is on.
3. With `usbdump` running, plug in the device. Wait for it to be recognised by the kernel. Then unplug it.
4. Stop `usbdump` with Control-C.
5. Open the resulting pcap file in Wireshark. You should see a series of USB transfers corresponding to enumeration: the `GET_DESCRIPTOR` requests for the device, configuration, and string descriptors, the `SET_ADDRESS` request, the `SET_CONFIGURATION` request, and so on.
6. Identify each transfer and write down what it does. Compare the captured device descriptor against the output of `usbconfig dump_device_desc` from Lab 1; they should match field for field.

This lab is the foundation of all USB reverse engineering. Every USB device you ever investigate will go through an enumeration sequence at attach time, and being able to read the enumeration sequence is the entry point to understanding what the device is doing.

### 36.17.3 Lab 3: Building a Safe Register Probing Module

This lab introduces the kernel-side tool that you will use for register-map work in real projects. You will build a small kernel module called `regprobe` that allocates a memory resource on a device of your choosing and exposes a `sysctl` that dumps the contents of the resource. No writes are performed.

Steps:

1. Build the `regprobe` module from `examples/part-07/ch36-reverse-engineering/lab03-register-map/`.
2. Identify a PCI device on your system that you do not need for anything else. A spare network card or any unused PCI device is ideal. (Do not use the device that backs your console or your storage.)
3. Detach the in-tree driver from the device with `sudo devctl detach <device>`.
4. Use the operating procedures in the lab's README to attach `regprobe` to the device.
5. Read the dump `sysctl` multiple times, with a few seconds between each read. Compare the dumps and identify which words are stable and which words are changing.
6. Reattach the in-tree driver with `sudo devctl attach <device>`.

The lab demonstrates two things: that a read-only probe is safe, and that even a read-only probe can reveal interesting structure in a device's register space. The dynamic words are most likely counters or status registers; the stable words are most likely configuration registers or identifiers.

### 36.17.4 Lab 4: Writing and Driving a Mock Device

This lab introduces the simulation side of the workflow. You will build a small kernel module that simulates a tiny “command and status” device entirely in software, and a small user-space test program that drives the simulated device through its sysctls.

Steps:

1. Build the mockdev module from `examples/part-07/ch36-reverse-engineering/lab04-mock-device/`.
2. Load the module with `sudo kldload ./mockdev.ko`.
3. Use the test program (also in the lab folder) to send a few commands to the mock device.
4. Observe the kernel log to see the commands being processed and the simulated status updates.
5. Modify the mock device to introduce a deliberate error: have it report failure for one specific command code. Verify that the test program detects the failure correctly.
6. Modify the test program to use a watchdog: if the mock device does not complete a command within a timeout, the program should report a failure rather than hanging.

The lab teaches the structure of mock-based testing in a setting where the mock is small enough to understand fully. In real projects, mocks become more complex, but the structure is the same.

### 36.17.5 Lab 5: Building a Safe Probing Wrapper

This lab consolidates the safety techniques from Section 10. You will build a small library of safe probing wrappers (read-modify- write-verify, timeout-protected operations, automatic recovery on failure) and use them to perform an experiment on the mock device from Lab 4.

Steps:

1. Open the safeprobe lab folder.
2. Read the wrapper library carefully. Notice how each operation is protected by a timeout, how each register write is followed by a readback, and how failures are reported clearly.
3. Build the wrapper library and the example driver that uses it.
4. Load the mock device from Lab 4.
5. Run the example driver against the mock device. Observe how the wrappers report the operations they perform.
6. Modify the mock device to inject a failure (an unexpected readback value, for example). Verify that the wrappers detect the failure and report it clearly, rather than letting the driver continue with corrupt state.

This lab teaches the safety wrappers as a tool you can apply to your own drivers. The cost of using them is small; the information they produce when something goes wrong is large.

### 36.17.6 Lab 6: Writing a Pseudo-Datasheet

This lab is the documentation-side equivalent of the technique labs. You will take the mock device from Lab 4 and write a pseudo-datasheet for it, following the structure from Section 11.

Steps:

1. Open the pseudo-datasheet template in `examples/part-07/ch36-reverse-engineering/lab06-pseudo-datasheet/`.
2. Read the template carefully and understand the structure.
3. Examine the mock device's source code to learn its register layout, command set, and behaviour.
4. Fill in the template with the mock device's information, following the structure: identity, provenance, resources, register map, buffer layouts, command interface, initialisation, operating patterns, quirks, open questions, references.
5. Save the result as a Markdown file alongside the mock device's source.

The mock device is small enough that the pseudo-datasheet can be complete in an hour or two. The exercise teaches you the structure of the document and the level of detail that each section deserves. When you later write a pseudo-datasheet for a real device, you will already know how to organise the information, even though the information itself will be much more extensive.

## 36.18 Challenge Exercises

The challenge exercises stretch the techniques from the labs into larger, more open-ended investigations. None of them requires exotic hardware; they all use either real common devices or the mock devices from the labs. Take your time with them. The labs have given you the techniques; the challenges give you the practice of applying them in less guided settings.

### 36.18.1 Challenge 1: Triage an Unknown Capture

The companion files include a small set of pcap captures of an unknown device's USB traffic. Open the captures in Wireshark. By analysis alone, identify:

- The device's vendor and product identifiers.
- The device class (HID, mass storage, vendor-specific, and so on).
- The device's endpoint layout (which endpoints, of which types, in which directions).
- The general shape of the device's data flow (does it appear to use bulk transfers, control transfers, interrupt transfers).
- Any obvious patterns in the data (periodic transfers, command and response, continuous streaming).

Write a one-paragraph summary of what kind of device the captures came from and what protocol it appears to use. The captures are labelled with the answer in a file you should not look at until you have written your own answer; compare and reflect.

### 36.18.2 Challenge 2: Extend the Mock Device

Take the mock device from Lab 4 and extend it to support a multi-byte data transfer through a small ring buffer. The ring buffer should have a producer pointer that the mock advances when data is “produced” (you can simulate this with a callout that generates synthetic data at a fixed rate), a consumer pointer that the driver advances when data is consumed, and a “queue size” register that the driver can read to learn the ring's capacity.

Write a small driver that:

1. Identifies the mock device.
2. Reads the queue-size register to learn the ring capacity.
3. Periodically reads the producer pointer to learn how many new entries are available.
4. Reads each new entry and prints it to the kernel log.
5. Updates the consumer pointer after processing each entry.

The exercise practises the ring-buffer-recognition skill from Section 5 in a setting where you control both sides. The challenge is to write the driver so that it correctly handles both the empty-ring case (no new entries) and the full-ring case (more entries than the driver has consumed).

### 36.18.3 Challenge 3: Detect Device Identity From Behaviour Alone

The mock device from Lab 4 has, in its companion code, a “mystery mode” that disables the identifier register and changes some of its behaviour to disguise its identity. Without reading the mock device’s source code, write a small driver that:

1. Probes the mystery-mode mock device.
2. Performs a series of safe register reads.
3. Observes the device’s response to a small set of test operations.
4. Identifies which of three known device identities the mock matches, based purely on the observed behaviour.

The challenge teaches the higher-level skill of fingerprinting a device from behaviour rather than from explicit identifiers. Real devices sometimes hide their identity for compatibility reasons (presenting themselves as a more common chip), and identifying them by behaviour is the only way to support them correctly.

### 36.18.4 Challenge 4: Document a Device You Own

Pick a piece of USB hardware that you own and that does not have particularly sensitive functionality. A USB game controller, a webcam, a USB-to-serial adapter, a USB Wi-Fi adapter. Write a pseudo-datasheet for it, using only what you can learn through `usbconfig`, `usbdump`, and Wireshark.

The challenge requires you to apply, in sequence, every technique from the chapter: identification, capture, observation, hypothesis, documentation. The result is a real artifact: a pseudo-datasheet for a real device. Many such pseudo-datasheets have grown into community projects that produced production drivers; yours might be the next one.

## 36.19 Hands-On Exercise: Your Own Observation

The labs have given you controlled practice against software mocks and known-good devices, and the challenges have asked you to apply those techniques to slightly less structured targets. The case studies in Section 13 walked you through reverse engineering work that has long since been absorbed into the FreeBSD tree. What none of those has asked you to do is to sit down with a device of your own choosing, observe it cold, map what you see onto interface and endpoint structures, and sketch the beginning of a driver from scratch. That is what this exercise does, and it is the closest this chapter can bring you to the work the umcs and axe authors did at the start of their projects.

Treat the exercise as a capstone rather than another lab. It does not introduce a new technique; it asks you to combine the techniques you have already practised into a single self-directed session, using a device you physically own and tools from the FreeBSD base system. The goal is not a shippable driver. The goal is a faithful short record of what you observed, a skeleton file that compiles and attaches cleanly, and a list of the questions your observation did not answer. If you finish with those three artefacts, the exercise has worked as intended.

### 36.19.1 Before You Start: The Ethical Gate

Because this exercise touches a real device that you own, the legal and ethical framework from Sections 1 and 12 applies to it directly, and it applies before the first command runs. Work through the short checklist below, in order. Do not proceed until every item has a clear answer.

1. **Pick a device you own outright.** The exercise is not a licence to probe equipment you have borrowed, rented, or been granted limited access to. The device must be yours, and it must be a device you are willing to see briefly misbehave under observation. A working rule is that if you would not feel comfortable unplugging the device in the middle of normal operation, it is not a good candidate for this exercise.
2. **Pick a device whose firmware and protocol are not vendor-protected.** Good targets are simple class-compliant USB devices that implement an open specification. A USB HID keyboard or mouse, a USB-to-serial adapter built around a commodity chip, a class-compliant USB audio interface, or a small USB-attached LED enclosure with a straightforward vendor-specific protocol are all reasonable choices. Rule out anything whose firmware is known to be DRM-encumbered, anything whose protocol is covered by a non-disclosure agreement that you have signed, and anything whose vendor driver ships under an end-user licence that specifically forbids the kind of observation described here. When a device is ambiguous, set it aside and pick something else. The exercise is about technique, not about probing any specific piece of hardware.
3. **Observe only, do not import.** Everything the walkthrough asks you to do is passive observation of traffic on a bus you control, followed by writing your own code against your own notes. That stays firmly inside the clean-room framework from Section 12. The moment you move from observation to copying vendor firmware, pasting vendor driver source, or redistributing a binary blob you did not author, you leave that framework and enter a space where the answer depends on the licence of the specific material. Do not cross that line inside the exercise. If you later want to carry the work toward a real driver, revisit Sections 9 and 12 and work through the licensing questions carefully before you do.
4. **Defer to the chapter's existing guardrails.** The discipline described in Sections 1 and 12 is the authoritative framework for this project. If any step below appears to conflict with those sections, treat the sections as correct and narrow the step, not the other way around. The walkthrough here is a concrete application of that framework, not an exception to it.

With those four items settled, the observation can begin. The rest of the exercise assumes that all four are cleared; if any of them is uncertain, close this section and choose a different device.

### 36.19.2 Step 1: Initial Identification With `usbconfig(8)`

Attach the target device to your FreeBSD host and confirm that the kernel has enumerated it:

```
$ sudo usbconfig list
ugen0.1: <0x1022 XHCI root HUB> at usb0, cfg=0 md=HOST spd=SUPER (5.0Gbps) pwr=SAVE
↳ (0mA)
ugen0.2: <Example device Example vendor> at usb0, cfg=0 md=HOST spd=FULL (12Mbps)
↳ pwr=ON (100mA)
```

Note the `ugenB.D` coordinate of your device. That coordinate is how every later `usbconfig` command will address it.

Next, check which FreeBSD driver, if any, has claimed an interface on the device:

```
$ sudo usbconfig -d ugen0.2 show_ifdrv
```

If a kernel driver already owns the interface, the output names it. A device whose interface is claimed by an existing driver will still respond to read-only descriptor queries, but you should not run your own experimental code against it until you have detached the existing driver with `devctl detach`. For a first pass, pick a device whose interface is either unclaimed or whose claiming driver you are willing to detach.

Finally, dump the full descriptor tree and save it for later:

```
$ sudo usbconfig -d ugen0.2 dump_all_desc > device-descriptors.txt
```

The output lists the device descriptor, each configuration descriptor, each interface descriptor, and each endpoint descriptor. You will compare this file against your packet capture in Step 3, and you will use it as the raw material for the short pseudo-datasheet fragment at the end of the exercise.

### 36.19.3 Step 2: Packet-Level Observation With `usbdump(8)`

Descriptors tell you what the device advertises about itself. To see what the device actually does, capture the USB bus while the device is used:

```
$ sudo usbdump -i usb0 -s 2048 -w trace.pcap
```

The `-i usb0` argument selects the USB bus to listen on; use the bus number your device is attached to, as reported by `usbconfig list`. The `-s 2048` argument caps each captured payload at 2048 bytes, which is comfortably above the maximum packet size of any full-speed or high-speed endpoint you are likely to encounter in this exercise. The `-w trace.pcap` argument writes a binary pcap file that Wireshark can open and dissect with the built-in USB dissector.

With `usbdump` running, exercise the device. For an HID keyboard, press a few keys. For a USB-to-serial adapter, open the port from another tool and send a handful of bytes in each direction. For a USB LED enclosure, cycle the LED through its supported states. Each action produces USB transfers on the bus, and every transfer ends up in the capture.

Stop `usbdump` with Control-C. A human-readable text record is also useful while you are annotating transfers by hand, so re-run the capture once with output redirected to a text file:



```
$ sudo usbdump -i usb0 -s 2048 > trace.txt
```

Keep both versions. The text form is easier to read line by line and to annotate in a plain editor; the pcap form is easier to navigate in Wireshark when you want to follow a specific flow.

### 36.19.4 Step 3: Map Descriptors to Interface and Endpoint Structures

Open `device-descriptors.txt` alongside `trace.txt`. Work through the descriptor dump and, for each interface descriptor, write down:

- The device or interface class (`bInterfaceClass`), subclass (`bInterfaceSubClass`), and protocol (`bInterfaceProtocol`). Standard values are tabulated in the USB class specifications: HID is class `0x03`, mass storage is class `0x08`, communications device class is `0x02`, audio is class `0x01`, and vendor-specific is class `0xFF`, among others.
- Each endpoint's address (`bEndpointAddress`), direction (the high bit of the address: set means IN, clear means OUT), and transfer type (the low two bits of `bmAttributes`: 0 control, 1 isochronous, 2 bulk, 3 interrupt).
- Each endpoint's maximum packet size (`wMaxPacketSize`).

Simple devices typically expose one interface with a small number of endpoints. A USB HID keyboard commonly has a single interrupt IN endpoint that the host polls for key-state reports. A USB-to-serial adapter commonly has a bulk IN endpoint for data from the device, a bulk OUT endpoint for data to the device, and uses the control endpoint for line-state commands.

Cross-check what you wrote down against the capture. Each endpoint address you recorded in the descriptor dump should appear in `trace.txt` as the source or destination of at least one transfer. Each transfer type you recorded (bulk, interrupt, control) should appear in the capture in the direction you expect. If the two disagree, look again before you sketch any driver code; one of the two observations is wrong and you want to find out which.

### 36.19.5 Step 4: Sketch a Newbus-Style Driver Skeleton

With the device's identity, interfaces, and endpoints in hand, you can sketch the outline of a driver that would attach to it. The sketch is not a runnable driver. It is the scaffold into which you would later drop real transfer setup, real data handling, and real error handling. The point of producing it now is to confirm that your observations are consistent enough to support a driver shape at all.

Choose a short, lowercase Newbus identifier. The convention in the FreeBSD USB tree is a brief lowercase string that echoes either the chip family or the device function: `umcs` for the MosChip serial bridge, `axe` for the ASIX Ethernet chip, `uftdi` for FTDI adapters, `ukbd` for USB HID keyboards. Pick something that does not collide with any name already in `/usr/src/sys/dev/usb/`. Your local sketch can use a placeholder such as `myusb` until you commit to a real name.

The skeleton below is the minimum a USB driver needs, and it is also what you will find in the companion file `skeleton-template.c`:

```
#include <sys/param.h>
#include <sys/system.h>
#include <sys/kernel.h>
#include <sys/module.h>
#include <sys/bus.h>

#include <dev/usb/usb.h>
#include <dev/usb/usbdi.h>

/* TODO: replace with the VID/PID you observed in Step 1. */
static const STRUCT_USB_HOST_ID myusb_devs[] = {
 { USB_VP(0x0000 /* VID */, 0x0000 /* PID */) },
};

static device_probe_t myusb_probe;
static device_attach_t myusb_attach;
static device_detach_t myusb_detach;

static int
myusb_probe(device_t dev)
{
 struct usb_attach_arg *uaa = device_get_ivars(dev);

 if (uaa->usb_mode != USB_MODE_HOST)
 return (ENXIO);

 return (usbdi_lookup_id_by_uaa(myusb_devs,
 sizeof(myusb_devs), uaa));
}

static int
myusb_attach(device_t dev)
{
 /* TODO: allocate the endpoints you catalogued in Step 3. */
 /* TODO: set up the usb_config entries that match them. */
 device_printf(dev, "attached\n");
 return (0);
}

static int
myusb_detach(device_t dev)
{
 /* TODO: unwind whatever attach allocated, in reverse order. */
 device_printf(dev, "detached\n");
}
```

```

 return (0);
}

static device_method_t myusb_methods[] = {
 DEVMETHOD(device_probe, myusb_probe),
 DEVMETHOD(device_attach, myusb_attach),
 DEVMETHOD(device_detach, myusb_detach),
 DEVMETHOD_END
};

static driver_t myusb_driver = {
 .name = "myusb",
 .methods = myusb_methods,
 .size = 0,
};

DRIVER_MODULE(myusb, uhub, myusb_driver, NULL, NULL);
MODULE_DEPEND(myusb, usb, 1, 1, 1);
MODULE_VERSION(myusb, 1);

```

The TODO markers show where the observations from Steps 1 through 3 feed into the driver. The vendor and product identifiers come straight from the device descriptor you saved. The endpoint allocation and transfer setup inside `myusb_attach` mirror the endpoints you catalogued. In the full form of the driver, each endpoint becomes an entry in a struct `usb_config` array whose `.type` and `.direction` fields record what you observed: a bulk IN endpoint uses `.type = UE_BULK` and `.direction = UE_DIR_IN`, an interrupt IN endpoint uses `.type = UE_INTERRUPT` and `.direction = UE_DIR_IN`, and so on. Existing drivers in `/usr/src/sys/dev/usb/` are full of examples of the pattern; `/usr/src/sys/dev/usb/serial/umcs.c` and `/usr/src/sys/dev/usb/serial/uftdi.c` are both reasonable references.

Do not try to fill in the transfer bodies yet. The goal at this stage is a skeleton that compiles, loads, and attaches to your device, and that prints a short message when it attaches and another when it detaches. If the skeleton does those three things cleanly, your observations were consistent. If it panics, refuses to attach, or prints nonsense, the fault is almost always in the descriptor mapping rather than in the skeleton itself; go back to Step 3 and check the IN/OUT directions and the transfer types before touching the code.

### 36.19.6 Step 5: Write Down What You Learned

Close the exercise by producing two short written artefacts to keep alongside the descriptor dump, the trace file, and the driver skeleton:

1. A **pseudo-datasheet fragment**, following the structure from Section 11. Identify the device, list its class and subclass, list its endpoints with their transfer types, directions, and maximum packet sizes, and note any behaviour you observed in the capture that surprised you. Keep the fragment short. The goal is to have something you could hand to a collaborator if the two of you wanted to pick the work up later.

2. A **list of open questions** you could not answer from observation alone. Every honest reverse engineering session ends with open questions, and writing them down is how you avoid pretending to know things you do not. Typical entries include a vendor-specific control request whose purpose is unclear, a block of bytes inside a bulk transfer whose layout is not obvious, or a descriptor field whose meaning depends on a device state you have not yet seen.

These two artefacts, together with `device-descriptors.txt`, `trace.pcap`, `trace.txt`, and `skeleton-template.c`, make up the output of the exercise. They are, at a smaller scale, the same kind of output the `umcs` and `axe` authors would have produced at the start of their own projects.

### 36.19.7 A Closing Reminder on Scope

Everything this exercise has asked you to do is passive observation of traffic on your own bus, followed by writing your own code against your own notes. That sits squarely inside the clean-room tradition the chapter has described. If, after working through the steps, you want to carry the work further toward a shippable driver, stop and re-read Sections 9, 11, and 12 before the next session. The techniques scale, but the legal and ethical framework scales with them, and it is easier to apply early than to unwind a commit later. The companion folder for this exercise, at `examples/part-07/ch36-reverse-engineering/exercise-your-own/`, collects a template walkthrough script, a skeleton source file with the `TODO` markers shown above, and a short `README` that gathers the safety notes in one place.

## 36.20 Common Mistakes and Troubleshooting

Reverse engineering has a small set of mistakes that nearly everyone makes early on, and an equally small set of mistakes that, even with experience, are easy to slip into. Recognising them in advance shortens the painful learning curve significantly.

### 36.20.1 Mistake 1: Believing the First Hypothesis

The most common mistake is to form a hypothesis early, then to interpret every subsequent observation as supporting the hypothesis, even when a more careful reading would suggest a different explanation. Confirmation bias is human, and it is particularly dangerous in reverse engineering because the observations are noisy and the hypothesis space is large.

The defence is the discipline of writing down each hypothesis explicitly, designing a test that could falsify it, running the test, and accepting the result honestly. If the test does not falsify the hypothesis, the hypothesis survives, but it is not yet “proven”; it is only “not yet refuted”. The next test might still kill it.

### 36.20.2 Mistake 2: Skipping the Notebook

The second most common mistake is to skip the notebook because the keyboard is faster. The notebook feels like overhead. The results, in the moment, seem clear. The patterns are obvious. There is no need to write them down, because they will be remembered.

A week later, the patterns are not obvious. Another week later, they are forgotten. A month later, the project has stalled because nobody, including the author, can reconstruct what was known. The notebook is what would have prevented this. Skip it at your peril.

### **36.20.3 Mistake 3: Writing to Unknown Registers**

The third common mistake is to write to a register without sufficient evidence that the write is safe. The temptation is strong: you have a hypothesis, you want to test it, the test involves a write, and what is the worst that could happen? Section 10 has spelled out what can happen, but it is worth repeating here. Devices can be permanently damaged by careless writes, and the people who learn this from experience usually learn it expensively.

The defence is the discipline of evidence-before-write. Before performing a write, name the source of evidence that the write is safe. If you cannot name a source, do not perform the write.

### **36.20.4 Mistake 4: Treating the Vendor Driver as Complete**

A fourth common mistake is to assume that the vendor's driver implements the full functionality of the device, and that matching the vendor's driver is sufficient to support the device fully. This is often wrong. Vendor drivers sometimes implement only the subset of device functionality that the vendor's products use; the full device may have features that the vendor's driver never exercises. Conversely, the vendor's driver sometimes contains workarounds for hardware bugs that you would not need if you implemented the operations differently.

The defence is to read the vendor's driver as one source of information among many, with the understanding that it represents the vendor's choices, not the complete description of the device.

### **36.20.5 Mistake 5: Not Testing the Detach Path**

A fifth common mistake is to focus on the attach path, where visible progress is made, and to neglect the detach path. The result is a driver that loads and works, but cannot be unloaded cleanly. Each test cycle then requires a reboot, which slows the work down by an order of magnitude.

The defence is to write the detach path early, test it in every build, and treat unload-time panics as serious bugs that block further work. A driver that cannot be unloaded is, for practical purposes, a driver that requires a reboot to test changes, and that is not a productive way to work.

### **36.20.6 Mistake 6: Not Saving Captures**

A sixth mistake is to throw away captures because they "did not show anything new". They might not show anything new today. They might show something important six weeks from now, when you have learned enough to interpret them. Save every capture. Storage is cheap. The captures themselves are part of the project's history, and they are sometimes the only way to reconstruct what was known at a given point.

### **36.20.7 Mistake 7: Working Alone**

A seventh mistake, particularly common in beginners, is to work alone. Reverse engineering is much faster when there is at least one other person to talk to about the work, even if that person is not also a reverse engineer.

The act of explaining what you have observed, what you believe, and what you are about to test forces clarity on the explanation, and clarity is what makes the work go forward.

If you do not have a colleague who is interested, find a community that is. The mailing lists, IRC channels, and forums for the device's family are full of people who understand the issues and can sometimes provide the missing piece. The community-collaboration side of Section 9 is not just about consuming prior work; it is also about contributing to a discussion that helps everyone involved.

### 36.20.8 Troubleshooting: When the Driver Silently Does Nothing

Sometimes the driver loads, the device appears to attach, the log says everything is fine, but no expected behaviour occurs. The most common causes:

- The driver thinks it has set up an interrupt handler, but the interrupt is not actually being delivered. Check `vmstat -i` to see whether interrupts are arriving. If not, the interrupt setup is wrong.
- The driver is reading the wrong register and seeing what looks like a legitimate value. Compare against the `pciconf -r` or `regprobe` output to verify that the values you are reading match the values present in the device.
- The driver is using the wrong endianness for multi-byte values. This is particularly common when reading values that look like integers but are actually byte strings.
- The driver has a bug in its sequence: it performed step B before step A, or it skipped a required step entirely.

The defence in each case is comparison against a known reference: the captures from Section 3, the `regprobe` dump, the working vendor driver. When the driver does not behave as expected, the question is: where does its behaviour first diverge from the reference?

### 36.20.9 Troubleshooting: When the Driver Panics on Unload

A panic on unload usually indicates that the driver freed something prematurely or kept a reference to something the kernel has destroyed.

The most common causes:

- A callout was not drained before the softc was freed. Check that every `callout_init` is matched by `callout_drain` in the detach path.
- A taskqueue task was still pending when the taskqueue was destroyed. Check that every `taskqueue_enqueue` is matched by `taskqueue_drain` in the detach path.
- An interrupt handler was not torn down before the IRQ resource was released. The order matters: tear down the handler with `bus_tear_down_intr`, then release the resource with `bus_release_resource`.
- A character device node was not destroyed before the softc was freed. Use `destroy_dev_drain` if there is any chance the device might still be open.

These are the same patterns we discussed in Chapter 35 for asynchronous drivers; reverse-engineered drivers face the same issues, with the additional complication that you may not yet fully understand which resources are in use at unload time.

### 36.20.10 Troubleshooting: When Behaviour Differs Between Runs

Sometimes the device behaves differently from one run to the next, even though nothing has changed in the driver. The causes are usually one of:

- The device contains state that is not reset between runs. Investigate what the device's reset actually clears, and what it leaves alone.
- The driver has a race condition that produces different outcomes depending on timing.
- The device is sensitive to environmental conditions (temperature, voltage) that vary slightly between runs.
- A second driver, or a userland program, is also accessing the device and interfering with your investigation.

Distinguishing these causes requires careful, repeated runs with each variable controlled. Reverse engineering against a device whose behaviour is not deterministic is significantly harder than the deterministic case, and identifying the source of the non-determinism is itself part of the work.

## 36.21 Wrapping Up

Reverse engineering is a craft built on patience, discipline, and careful documentation. The chapter has walked through the entire process: why this work is sometimes necessary, where the legal lines fall, how to set up the lab, how to observe systematically, how to build a register map, how to identify buffer structures, how to write a minimal driver and grow it incrementally, how to validate assumptions in simulation, how to collaborate with the community, how to avoid damaging hardware, and how to consolidate the result into a maintainable driver and a usable pseudo-datasheet.

The techniques are concrete. The discipline is what holds them together. A reverse engineer who follows the discipline will, in time, be able to take an undocumented device and produce a working driver for it. A reverse engineer who skips the discipline will produce code that works some of the time, that fails for reasons that are hard to diagnose, and that cannot be maintained or extended without rediscovering everything that was forgotten the first time around.

Several themes have run through the chapter and deserve a final explicit summary.

The first theme is the separation of observation from hypothesis from verified fact. An observation is what you saw. A hypothesis is what you think it means. A verified fact is a hypothesis that has survived deliberate attempts to falsify it. Mixing the three together produces confusion; keeping them separate produces clarity. The notebook discipline that the chapter has emphasised is, at root, the discipline of keeping these three categories distinct.

The second theme is the value of starting small. A minimal driver that does one thing correctly is a better foundation than a large driver that does many things uncertainly. Each new feature added incrementally, with verification, is far cheaper to get right than a large multi-feature change. The pace of work in reverse engineering should always feel slower than the pace of work in normal driver development; the slowness is what catches the bugs that would otherwise survive into the final code.

The third theme is the centrality of safety. Reverse engineering is one of the few software activities where careless mistakes can permanently damage real hardware. The safety techniques of Section 10, the simulation techniques of Section 8, and the read-first principle of Section 4 are all manifestations of the same underlying

principle: respect the unknown, and earn the right to perform an operation by accumulating evidence that the operation is safe.

The fourth theme is the importance of documentation. The pseudo-datasheet, the notebook, the manual page, the comments in the code: these are not afterthoughts, they are part of the output of the work. A driver without documentation is a driver that nobody can maintain, including its author six months from now. A pseudo-datasheet that records what was learned is the artifact that gives the work durability beyond the author's involvement.

The fifth theme is community. Reverse engineering is rarely truly solitary work. Prior work exists for almost every device worth investigating; new work, when done well, contributes to the ongoing community knowledge. Searching, evaluating, and contributing are part of the craft. A reverse engineer who works in isolation reinvents wheels; a reverse engineer who engages with the community builds on others and is built upon in turn.

You have now seen the entire shape of the work. The techniques are within reach. The discipline takes practice. The first serious project will be slow and full of mistakes; the second will be faster; by the third or fourth, the rhythm will start to feel natural. The labs in this chapter are the beginning of that practice, and the challenge exercises are the next step. Real projects on real devices are where the skills consolidate, and the FreeBSD community has plenty of devices that would benefit from someone willing to do the work.

Take a moment to appreciate what has changed in your toolkit. Before this chapter, an undocumented device was a stop sign. Now it is a project. The methods you have learned are the same methods that the authors of many drivers in `/usr/src/sys/dev/` used to bring those drivers into existence. Some of them worked alone, others in small groups, but all of them followed recognisably the same workflow: observe, hypothesize, test, document. You now know how to do that work.

## 36.22 Bridge to Chapter 37: Submitting Your Driver to the FreeBSD Project

The driver you have just learned to build, whether it implements a fully reverse-engineered device or some more conventional piece of hardware, is most useful when other people can find it, build it, and rely on it. So far we have treated the driver as a private project, something you load on your own systems and document for your own future reference. The next chapter changes that: we will see how to take a finished driver and offer it for inclusion in the FreeBSD source tree itself.

The shift is significant. A driver in your private repository serves you and anyone who happens to find your repository. A driver in the FreeBSD source tree is built by every FreeBSD release, exposed to every FreeBSD user, maintained by the FreeBSD project, and tested by every commit that touches the surrounding code. The amplification of value is enormous, and so are the responsibilities that come with it. The submission process is FreeBSD's mechanism for ensuring that the driver meets the standards of the source tree before it is allowed in.

Chapter 37 walks through that process. We will look at the FreeBSD development model: the difference between a contributor and a committer, the role of the source-tree organisation, the review process, and the conventions that the FreeBSD source tree enforces. We will learn the style guidelines (`style(9)` for code and `style(4)` for manual pages, with a few related conventions for makefiles, headers, and naming), how to structure your driver's files for inclusion in `/usr/src/sys/dev/`, how to write the manual page that every driver should ship with, and how to write commit messages in the form FreeBSD expects.

We will also look at the social dynamics of contribution: how to engage with reviewers, how to respond to feedback, how to iterate on a patch series, and how to participate in the project in a way that builds long-term



reputation. The technical side of the submission process is straightforward; the social side is where most first-time contributors find the surprises.

Several themes from this chapter will carry forward. The clean detach path is essential, because reviewers will check it. The safety wrappers and conservative behaviour from Section 10 are the same disciplines that production-grade drivers exhibit, and they will be appreciated by reviewers. The pseudo-datasheet is not part of the submission, but the understanding it represents is what justifies the driver's claims and what lets reviewers trust the implementation. The community engagement from Section 9 is the foundation of the longer-term relationship the contribution is intended to build.

The reverse-engineered driver is a particularly interesting case for FreeBSD inclusion, because the project has long-standing experience with such drivers and well-developed processes for handling them. The provenance discipline from Section 11 is exactly what the project needs to evaluate whether a driver is clean of copyright concerns. The documented limitations and open questions from the pseudo-datasheet help the project set appropriate expectations for users. The safety wrappers and the conservative behaviour help reviewers feel confident that the driver will not damage user hardware.

If you have followed the labs and the challenges in this chapter, you have produced at least a few small pieces of code that could, with some polish, be candidates for inclusion. Chapter 37 will show you how to take such a piece of code and walk it through the submission process. The mock-device driver from Lab 4 might not be a candidate (it does not drive real hardware), but the patterns it demonstrates are exactly the patterns a real driver should follow, and the safety wrappers from Lab 5 are patterns that reviewers will recognise and approve.

The book is approaching its concluding chapters. You started in Part 1 with no kernel knowledge at all. You learned UNIX and C in Parts 1 and 2. You learned the structure and lifecycle of a FreeBSD driver in Parts 3 and 4. You learned the bus, network, storage, and pseudo-device patterns in Parts 4 and 5. You learned the Newbus framework in detail in Part 6. You have now worked through Part 7's mastery topics: portability, virtualisation, security, FDT, performance, advanced debugging, asynchronous I/O, and reverse engineering. The skill set is comprehensive. The remaining chapters tie it back to the project, the community, and the practice of contributing to a real-world operating system.

Take a moment, before moving on to Chapter 37, to look back at what you can now do. You can write a simple character driver from scratch. You can write a network driver that participates in the kernel's networking stack. You can write a driver for a device discovered through Newbus. You can write an asynchronous driver that supports `poll`, `kqueue`, and `SIGIO`. You can debug a driver with `dtrace`, `KASSERT`, and `kgdb`. You can work with a device for which no documentation exists. Each of these capabilities is a real one, built on the previous ones, and together they constitute a working knowledge of FreeBSD device-driver development.

The next chapter will help you take that knowledge and turn it into a contribution to the FreeBSD project. Let us see how that process works.

## 37 Submitting Your Driver to the FreeBSD Project

### 37.1 Introduction

If you have followed this book from the beginning, you have travelled a long way. You started without any kernel knowledge at all, learned UNIX and C, walked through the anatomy of a FreeBSD driver, built character and network drivers by hand, explored the Newbus framework, and worked through a full part on mastery topics: portability, virtualisation, security, FDT, performance, advanced debugging, asynchronous I/O, and reverse engineering. You have now reached the point where you can sit down in front of a lab machine, open a text editor, and write a driver for a device that FreeBSD does not yet support. That is a serious engineering skill, and it did not come for free.

This chapter is where the work turns outward. Up to now, the drivers you have built have lived on your own systems. You loaded them with `kldload`, tested them, debugged them, and unloaded them when you were done. They were useful to you, and perhaps to a few friends or colleagues who copied them from your repository. That is already worthwhile work. But a driver that lives only on your machine serves only the people who happen to find your machine. A driver that lives inside the FreeBSD source tree serves every FreeBSD user, on every release, on every architecture the driver targets, for as long as the code is maintained. The amplification of value is enormous, and the responsibilities that come with it are the subject of this chapter.

The FreeBSD Project has been accepting contributions from outside developers since the early 1990s. Thousands of people have submitted patches; hundreds have eventually become committers. The process by which new code enters the tree is not a bureaucratic obstacle course. It is a review workflow designed to preserve the qualities that make FreeBSD trustworthy: code consistency, long-term maintainability, portability across architectures, legal cleanness, careful documentation, and continuity of care. Every one of those qualities is something the reviewers protect on behalf of everyone who runs FreeBSD. When you submit a driver, you are asking the project to take on a long-term responsibility for it. The review process is how the project confirms that the driver is worth that responsibility, and also how the project helps you get the driver into a shape where the answer is yes.

That framing matters. New contributors often arrive at the review process expecting an adversarial experience, where reviewers look for reasons to reject the work. The reality is the opposite. Reviewers are, overwhelmingly, trying to help. They want your driver to be merged. They want it to be merged in a form that will still work five releases from now. They want it to not be a maintenance burden for whoever has to touch the surrounding code next year. The comments they leave on a first-round patch are not a score; they are a list of things that, when addressed, will let the patch be committed. A contributor who internalises that framing finds the review process cooperative rather than stressful.

There is, however, a distinction that needs to be clear from the start. A working driver is not the same thing as an upstream-ready driver. A driver that loads on your laptop, drives your hardware, and does not panic when you unload it has cleared only the first few checkpoints. To be upstream-ready it also needs to pass the project's style guidelines, carry a proper licence, come with a manual page that explains how a user interacts with it,

build on every architecture the project supports, integrate cleanly into the existing source-tree layout, and be accompanied by a commit message that another reviewer can read in five years without needing to reconstruct the context. None of these are busywork. Each of them exists because experience has shown what happens to code bases where they are skipped.

The chapter is organised around a natural workflow. We will begin by looking at how the FreeBSD Project is organised from the point of view of a contributor, and what the distinction between contributor and committer actually means in practice. We will then walk through the mechanical preparation of a driver for submission: what file layout to use, what code style to follow, how to name things, and how to write a commit message that a reviewer can read with gratitude. We will look at licensing and legal compatibility, because even excellent code cannot be accepted if its provenance is unclear. We will spend a serious amount of time on manual pages, because the manual page is the reader-facing half of the driver and it deserves the same care as the code. We will walk through testing expectations, from local builds through make universe, and we will see how to generate a patch in a form the reviewers find convenient. We will discuss the social side of the review process: how to work with a mentor, how to respond to feedback, how to iterate through review rounds without losing momentum. And we will end with the longest-lasting commitment of all, which is maintenance after the driver has been merged.

The companion code for this chapter, in `examples/part-07/ch37-upstream/`, includes several practical artifacts: a reference driver-tree layout that mirrors the shape a small driver would take in `/usr/src/sys/dev/`; a sample manual page you can adapt to your own driver; a submission checklist you can use as a final review before sending a patch; a draft cover letter for an email to a project mailing list; a helper script that generates a patch with the conventions the project expects; and a pre-submission validation script that runs the lint, style, and build checks in the correct order. None of these are a substitute for understanding the underlying material, but they will save you from the common mistakes that cost first-time contributors a round or two of reviews.

One more note before we begin. This chapter is not going to teach you the political or governance history of the FreeBSD Project. We will touch on the Core Team and the roles of the various committees only as much as is necessary for a contributor to navigate the project. If you are curious about FreeBSD governance after reading this chapter, the project's own documentation is the right next stop, and we will point you toward it. The scope of this chapter is the practical work of turning a driver you have written into a driver that can be merged upstream.

By the end of the chapter, you will have a clear picture of the submission workflow, a hands-on understanding of the style and documentation conventions, a rehearsal of the review cycle, and a realistic sense of what happens after your driver is in the tree. You will not be a FreeBSD committer by the end of this chapter; the project grants commit rights only after a sustained history of quality contributions, and that is by design. But you will know how to make the first contribution, how to make it well, and how to build up the reputation that could, in time, lead to commit rights if that is a direction you choose to pursue.

## 37.2 Reader Guidance: How to Use This Chapter

This chapter sits in Part 7 of the book, immediately after the chapter on reverse engineering and immediately before the closing chapter. Unlike many earlier chapters, the subject matter here is more about workflow and discipline than about kernel internals. You will not need to write any new driver code to follow the chapter, although you will benefit greatly if you apply what you learn to a driver you have already written.

The reading time is moderate. If you read straight through without stopping to try anything, the prose will take around two to three hours. If you work through the labs and challenges, budget a full weekend or several evenings. The labs are structured as short, focused exercises that you can do against any small driver you have on hand, including one of the earlier chapter drivers, one of the mock drivers from Chapter 36, or a fresh driver you write for this chapter.

You do not need any special hardware. A FreeBSD 14.3 development virtual machine, or a bare-metal FreeBSD system where you are comfortable running build and test commands, is sufficient. The labs will ask you to apply style checks to real code, build a real driver as a loadable module, validate a manual page with `mandoc(1)`, and rehearse the patch-generation workflow against a throwaway git branch. Nothing will touch the real FreeBSD Phabricator or GitHub, so there is no risk of accidentally submitting half-finished work to the project.

A reasonable reading schedule looks like this. Read Sections 1 and 2 in one sitting; they set up the conceptual framing of how FreeBSD development works and how your driver should be laid out. Take a break. Read Sections 3 and 4 in a second sitting; they cover licensing and manual pages, which together make up most of the paperwork side of a submission. Read Sections 5 and 6 in a third sitting; they cover testing and actual patch generation, which is where the chapter turns from preparation to action. Read Sections 7 and 8 in a fourth sitting; they cover the human and long-term sides of contribution. The labs are best done over a weekend, with enough time to redo them more than once if the first pass exposes something you would like to improve.

If you are already a confident FreeBSD user and a confident kernel developer, the material in this chapter will feel familiar in broad shape but may still surprise you in the specifics. The specifics matter. A reviewer who knows the tree well will notice within seconds whether the file layout matches the conventions, whether the copyright header is in the current recommended form, whether the manual page uses the modern `mdoc` idioms, and whether the commit message follows the expected subject-line style. Getting these small things right up front makes the difference between a review that lasts one round and a review that lasts five.

If you are a beginner, do not let the specifics intimidate you. Every committer in the project was, at one point, someone whose first patch bounced off five rounds of review before landing. The ability to write good code is something you have already built up over the book. The ability to submit it well is what this chapter adds. You will not get it perfect on the first try. That is normal. What matters is that you understand the shape of the process and that you approach each review with the intent to improve the submission rather than to defend it.

Several of the guidelines in this chapter, especially around licensing, manual pages, and the review workflow, reflect the state of the FreeBSD Project as of FreeBSD 14.3. The project evolves, and a few of the specific conventions may shift over time. Where we know a convention is changing, we will flag it. Where we are citing a specific file in the tree, we will name the file so you can open it yourself and verify the current state. The reader who trusts but also verifies is the reader the project benefits from most.

One last note about pace. This chapter deliberately teaches discipline as much as it teaches process. Several of the sections will seem almost repetitive in their insistence on small details: trailing whitespace, correct header comments, exact manual-page macro usage. That insistence is part of the lesson. FreeBSD is a large code base with a long institutional memory, and the small details are what keep it maintainable. If you find yourself tempted to skim a style section, slow down instead. The slowness is the craft.

## 37.3 How to Get the Most Out of This Chapter

The chapter is organised to be read linearly, but every section stands on its own well enough that you can come back to a specific section when you need to. Several habits will help you absorb the material.

First, read each section with the FreeBSD source tree open on your screen. Every time the chapter mentions a reference file such as `/usr/src/share/man/man9/style.9`, open it and skim it. The chapter gives you the shape and the motivation; the reference files give you the authoritative detail. A habit of cross-checking what you read in the chapter against what the tree actually says will serve you for the rest of your career as a FreeBSD contributor.

**A note on line numbers.** When the chapter points you at a named piece of infrastructure in the tree, such as `make_dev_s`, `DRIVER_MODULE`, or the `style(9)` rules themselves, the pointer is anchored on that name. The `mydev.c:23` style-check transcripts you will see later refer to lines in your own in-progress driver and will change as you edit it. Either way, the durable reference is the symbol, not the number: `grep` for the name rather than rely on a line.

Second, keep a small note file as you read. Every time a section mentions a convention, a required section, or a command, write it down. By the end of the chapter you will have a personalised submission checklist. The companion `examples/part-07/ch37-upstream/` directory includes a checklist template you can start from, but a checklist you have typed up yourself, in your own words, will be more useful than any template.

Third, have a small driver of your own in mind as you read. It could be the LED driver from earlier chapters, the mock device from Chapter 36, or a character driver you wrote for practice. The chapter will ask you to imagine preparing that specific driver for submission. Working against a concrete driver makes the instructions land far more firmly than trying to absorb them in the abstract.

Fourth, do not skip the labs. The labs in this chapter are short and practical. Most of them take less than an hour. They exist because there are parts of the submission process that only become clear once you try them against real code. A reader who works through the labs will emerge with genuine muscle memory; a reader who skips them will re-read the chapter six months from now and discover that most of it did not stick.

Fifth, treat the early mistakes as part of the training. The first time you run `tools/build/checkstyle9.pl` against your code, you will see warnings. The first time you run `mandoc -Tlint` against your manual page, you will see warnings. The first time you run `make universe` against your tree, you will see errors on at least one architecture. Every one of those warnings is teaching you something. The reviewers in the project see these same warnings every day; the craft of preparing a submission is, in large part, the craft of noticing and fixing them before anyone else has to.

Finally, be patient with the rhythm of the chapter. A few of the later sections spend time on what might seem like social or interpersonal material: how to handle feedback, how to respond to a reviewer who has misunderstood your patch, how to build the relationships that lead to sponsorship. That material is not optional. Software engineering at the level of an open-source kernel project is a collaborative craft, and the collaboration is the craft. Reading those sections carelessly will cost you more in practice than reading the style sections carelessly.

You now have the map. Let us turn to the first section and look at how the FreeBSD Project is organised from the point of view of a contributor.

## 37.4 Section 1: Understanding the FreeBSD Contribution Process

### 37.4.1 What the FreeBSD Project Actually Is

Before we can talk about contributing to the FreeBSD Project, we need a clear picture of what the project is. The FreeBSD Project is a community of volunteers and paid contributors who together develop, test, document, release, and support the FreeBSD operating system. The project has been continuously active since 1993. It is organised around a set of shared source trees, a code-review culture, a release engineering discipline, and a body of accumulated institutional knowledge about how kernels, userlands, ports, and documentation should be put together.

The project is often summarised in three words: source, ports, and documentation. These correspond to three main repositories or subprojects, each with its own maintainers, reviewers, and conventions. Source, usually written as `src`, is the base system: the kernel, the libraries, the userland utilities, everything that a FreeBSD installation ships with. Ports is the collection of third-party software that can be built on FreeBSD, such as programming languages, desktop environments, and application servers. Documentation is the Handbook, the articles, the books such as the Porter's Handbook and the Developer's Handbook, the FreeBSD website, and the translation infrastructure.

Device drivers live primarily in the `src` tree, because they are part of the base system kernel and base-system support for hardware. When this chapter talks about submitting a driver, it means submitting it to the `src` tree. Ports and Documentation have their own contribution pipelines, which follow similar principles but differ in detail. This chapter focuses exclusively on `src`.

The `src` tree is large. You can see its top-level structure by browsing `/usr/src/`. The manual page `/usr/src/share/man/man7/development.7` gives a short, readable introduction to the development process, and the file `/usr/src/CONTRIBUTING.md` is the project's own current guidance for contributors. If you read only two files before your first submission, read those two. We will quote from both repeatedly in this chapter.

### 37.4.2 The Project's Decision-Making Structure

FreeBSD uses a relatively flat decision-making structure compared to some other large projects. The core of the structure is the group of committers, who are developers with write access to the source repositories. Committers are elected by existing committers after a sustained history of quality contributions. A nine-person elected body called the Core Team handles certain kinds of project-wide decisions and disputes. Smaller teams such as the Release Engineering Team (`re@`), the Security Officer Team (`so@`), the Ports Management Team (`portmgr@`), and the Documentation Engineering Team (`doceng@`) look after specific areas.

For the purposes of submitting a driver, most of that structure does not matter much in day-to-day practice. The people who will review your driver are individual committers who happen to know the subsystem your driver fits into. If your driver is a network driver, the reviewers will probably be people active in the networking subsystem. If it is a USB driver, the reviewers will be people active in USB. The Core Team is not involved in individual driver submissions; neither is the Release Engineering Team, although they will be the ones who decide which release your driver first appears in once it is merged.

The practical mental model is this. The FreeBSD Project is a large community of engineers. Some of them can commit to the tree directly. A much larger number contribute through review processes. When you submit a

driver, you become part of that larger number, and the review process is how the committer community evaluates whether the driver is ready to enter the tree under their shared responsibility.

### 37.4.3 Contributor Versus Committer

The distinction between contributor and committer is central to how the project works, and it is often misunderstood by newcomers.

A contributor is anyone who submits changes to the project. You become a contributor the first time you open a Phabricator review or a GitHub pull request against the FreeBSD source tree. There is no formal process for becoming a contributor. You simply submit work. If the work is good, it gets reviewed, revised, and eventually committed to the tree by a committer on your behalf. The commit carries your name and email in the Author : field, so that you get credit for the code even though you did not push it yourself.

A committer is a contributor who has been granted direct write access to one of the repositories. Commit rights are granted after a sustained history of quality contributions, usually over several years, and only after a nomination by an existing committer and a vote by the relevant committer group. Commit rights come with responsibilities: you are expected to review other people's patches, participate in project discussions, and take ownership of the code you have committed over the long term.

The two roles are not a hierarchy of prestige. They are a division of labour. Contributors focus on writing and submitting good patches. Committers focus on reviewing, integrating, and maintaining the tree. A contributor with a single high-value patch is more valuable to the project than a committer who does not actively participate. The project depends on both.

For this chapter, you should think of yourself as a contributor. Your goal is to produce a submission that a committer can review, accept, and commit. If, years from now, you find yourself with a long history of contributions and a sustained relationship with the project, the question of commit rights may arise organically. But that is a question for later. The focus here is making your first contributions count.

### 37.4.4 How src Work Is Organised

The src repository is a single git tree. The main branch, confusingly called `main` in git but also referred to as `CURRENT` in release-engineering language, is where all active development happens. Changes are committed first to `main`. Then, if the change is a bug fix or a small feature that fits within a stable release, it may be cherry-picked back into one of the `stable/` branches, which correspond to the major FreeBSD releases such as 14 and 15. Releases themselves are tagged points on the `stable/` branches.

As a driver contributor, your default target is `main`. Your patch should apply to current `main`, should build against current `main`, and should be tested against current `main`. If the driver is something users of FreeBSD 14 would also want, a committer may choose to cherry-pick the commit back to the relevant `stable/` branch after it has been in `main` for a bit, but that is the committer's decision, not yours to make in the submission.

The git repository is visible at <https://cgit.freebsd.org/src/> and is also mirrored at <https://github.com/freebsd/src>. You can clone from either. The authoritative push URL, for those with commit access, is `ssh://git@gitrepo.FreeBSD.org/src.git`, but as a contributor you will not push directly. You will generate patches and send them through the review workflow.

### 37.4.5 Where Device Drivers Live in the Source Tree

Most device drivers live under `/usr/src/sys/dev/`. This directory contains hundreds of subdirectories, one per driver or device family. If you browse it, you will see a cross-section of the hardware FreeBSD supports: Ethernet chips, SCSI adapters, USB devices, sound cards, I/O controllers, and a long list of other categories.

A small selection of existing driver subdirectories worth knowing:

- `/usr/src/sys/dev/null/` for the `/dev/null` character device.
- `/usr/src/sys/dev/led/` for the generic LED framework.
- `/usr/src/sys/dev/uart/` for UART drivers.
- `/usr/src/sys/dev/virtio/` for the VirtIO family.
- `/usr/src/sys/dev/usb/` for the USB subsystem and the USB-side device drivers.
- `/usr/src/sys/dev/re/` for the RealTek PCI/PCIe Ethernet driver.
- `/usr/src/sys/dev/e1000/` for the Intel Gigabit Ethernet driver family.
- `/usr/src/sys/dev/random/` for the kernel random-number subsystem.

Some categories of driver live elsewhere. Network drivers whose role is more about the networking stack than about the device itself sometimes live under `/usr/src/sys/net/`. Filesystem-like devices and pseudo-devices sometimes live under `/usr/src/sys/fs/`. Architecture-specific drivers sometimes live under `/usr/src/sys/<arch>/`. For most beginner submissions, however, the question will be which subdirectory under `/usr/src/sys/dev/` is the right home, and the answer is almost always obvious. If your driver is for a new network chip, it probably belongs in its own subdirectory under `/usr/src/sys/dev/`, possibly inside an existing family subdirectory if it extends an existing family. If it is for a USB device, you may find that it lives under `/usr/src/sys/dev/usb/` instead. If you are unsure, a search through the existing tree for a similar driver will usually tell you where yours belongs.

### 37.4.6 The Second Half of the Driver: Kernel Build Integration

In addition to the driver source files themselves, a driver that is merged into FreeBSD has a second home under `/usr/src/sys/modules/`. This directory contains the kernel-module Makefiles that let the driver be built as a loadable kernel module. For each driver in `/usr/src/sys/dev/<driverdir>/`, there is typically a corresponding directory in `/usr/src/sys/modules/<moduledir>/` containing a small Makefile that tells the build system how to assemble the module. We will look at that Makefile in detail in Section 2.

A few drivers have additional integration points. Drivers that ship as part of the default kernel are listed in the architecture configuration files under `/usr/src/sys/<arch>/conf/GENERIC`. Drivers that come with device-tree bindings may have entries under `/usr/src/sys/dts/`. Drivers that expose tunable sysctls or loader variables need entries in the relevant documentation.

As a first-time contributor, you do not need to worry about all of these integration points at once. The minimum set for a typical driver submission is the files under `/usr/src/sys/dev/<driver>/`, the Makefile under `/usr/src/sys/modules/<driver>/`, and the manual page under `/usr/src/share/man/man4/<driver>.4`. Anything beyond that is incremental.



### 37.4.7 The Review Platforms

FreeBSD currently accepts source-code contributions through several channels. The `/usr/src/CONTRIBUTING.md` file lists them explicitly:

- A GitHub pull request against <https://github.com/freebsd/freebsd-src>.
- A code review in Phabricator at <https://reviews.freebsd.org/>.
- An attachment on a Bugzilla ticket at <https://bugs.freebsd.org/>.
- Direct access to the git repository, for committers only.

Each of these channels has its own conventions and its own preferred use cases.

Phabricator is the traditional code-review platform for the project. It handles full review workflows: multi-round feedback, revision history, inline comments, reviewer assignment, and commit-ready patches. Most significant patches, including most driver submissions, go through Phabricator. You will see it referred to as “review D12345” or similar, where D12345 is the Phabricator differential revision identifier.

GitHub pull requests are an increasingly accepted submission route, particularly for small, self-contained, uncontroversial patches. The `CONTRIBUTING.md` file explicitly notes that GitHub PRs work well when the change is limited to fewer than about ten files and fewer than about two hundred lines, and when it passes the GitHub CI jobs and has limited scope. A typical small driver fits within those bounds; a larger driver with many files and integration points may be better handled through Phabricator.

Bugzilla is the project’s bug tracker. If your driver fixes a specific reported bug, a patch attached to the corresponding Bugzilla entry is the right home for it. If the driver is new work rather than a bug fix, Bugzilla is usually not the right starting point, although a reviewer may ask you to open a Bugzilla ticket so that the work has a tracking number.

For a first driver submission, either Phabricator or a GitHub pull request is appropriate. Many contributors start with a GitHub PR because the workflow is familiar, and switch to Phabricator if the review grows beyond what GitHub handles well. We will walk through both routes in Section 6.

The review-platform landscape does shift over time, and the specific URLs, scope limits, and preferred routes described in this chapter can be superseded by changes to `/usr/src/CONTRIBUTING.md` or to the project’s contribution pages. The process details above were last verified against the in-tree `CONTRIBUTING.md` on 2026-04-20. Before you prepare your first submission, re-read the current `CONTRIBUTING.md` and the committer’s guide linked from the FreeBSD documentation site; if they disagree with this chapter, trust the project’s files, not the book.

### 37.4.8 Exercise: Browse the Source Tree and Identify Similar Drivers

Before moving on to Section 2, spend half an hour browsing `/usr/src/sys/dev/` and building an intuition for what a FreeBSD driver looks like from the outside.

Pick three or four drivers that are roughly similar in scope to the one you intend to submit, or to any driver you have built during this book. For each, look at:

- The directory contents. How many source files? How many headers? What are the file names?
- The corresponding Makefile under `/usr/src/sys/modules/`. What does it list in `KMOD=` and `SRCS=`?
- The manual page under `/usr/src/share/man/man4/`. Open it and note the section structure.

- The copyright header in the main `.c` file. Note its format.

You are not trying to memorise anything in this exercise. You are building a baseline intuition. By the time you have looked at three or four real drivers, the conventions of the tree will feel less abstract. When Section 2 talks about where files go and how they are named, the recommendations will land on top of a mental picture you have already built. That is the right way to absorb this material.

### 37.4.9 Wrapping Up Section 1

The FreeBSD Project is a long-lived community organised around three main subprojects: `src`, `ports`, and `documentation`. Device drivers live in the `src` tree, mostly under `/usr/src/sys/dev/`, with corresponding module Makefiles under `/usr/src/sys/modules/` and manual pages under `/usr/src/share/man/man4/`. Contributions enter the tree through a review process handled by committers who are active in the relevant subsystem. The distinction between contributor and committer is a division of labour, not a hierarchy of prestige. Your goal as a first-time contributor is to produce a submission that a committer can review, accept, and commit.

With that framing in place, we can now turn to the mechanical question of what shape your driver should be in before you submit it. Section 2 walks through the preparation step by step.

## 37.5 Section 2: Getting Your Driver Ready for Submission

### 37.5.1 The Gap Between a Working Driver and a Submission-Ready Driver

A driver that loads, runs, and unloads cleanly on your test machine is a working driver. A driver that a FreeBSD committer can review, merge, and maintain is a submission-ready driver. The gap between the two is almost always larger than first-time contributors expect, and closing the gap is the work of this section.

The gap has three parts. The first is layout: where the files go, what they are named, and how they integrate with the existing build system. The second is style: how the code is formatted, named, and commented, and how closely it matches the project's `style(9)` guidelines. The third is presentation: how the commit is packaged, what the commit message says, and how the patch is structured for review. None of these are hard once you know what to look for, but each of them involves a dozen or two small conventions that collectively determine whether a reviewer's first impression is smooth or bumpy.

Before we begin, take a moment to appreciate why these conventions exist. FreeBSD has thirty years of accumulated code. Thousands of drivers have entered the tree over that time. The conventions that feel arbitrary when you first encounter them are, in almost every case, the result of an earlier painful experience that the community decided never to repeat. A convention that prevents a bug, or that reduces a recurring source of reviewer friction, pays for itself many times over. When you follow the conventions, you are benefiting from thirty years of institutional memory. When you ignore them, you are volunteering to relearn those lessons yourself, and to put your reviewers through them again.

### 37.5.2 Where the Files Go

For a standalone driver in the tree, the typical layout looks like this. Let us assume your driver is called `mydev` and that it drives a PCI-attached sensor board.

- `/usr/src/sys/dev/mydev/mydev.c` is the main driver source file. For a small driver, this may be the only source file.
- `/usr/src/sys/dev/mydev/mydev.h` is the driver header. If you only have a single `.c` file and its internal declarations do not need to be exposed, you may not need this header.
- `/usr/src/sys/dev/mydev/mydevreg.h` is a common name for a header that defines the hardware registers and bit fields. This convention, using a `reg` suffix, is widespread in the tree, and separating register definitions from driver-internal declarations is good practice.
- `/usr/src/sys/modules/mydev/Makefile` is the Makefile for building the driver as a loadable kernel module.
- `/usr/src/share/man/man4/mydev.4` is the manual page.

You may encounter existing drivers that do not follow this exact layout. Older drivers from before the current conventions were established sometimes put everything in one place, or use different file names. The conventions continue to evolve. For a new driver, following the modern layout will save you review friction.

### 37.5.3 What Goes in `mydev.c`

The main source file typically contains, in order:

1. The copyright and licence header, in the format we will cover in Section 3.
2. The `#include` directives, typically starting with `<sys/cdefs.h>` and `<sys/param.h>`, followed by the other kernel headers your driver needs.
3. Forward declarations and static variables.
4. The driver methods: `probe`, `attach`, `detach`, and anything else your `device_method_t` table references.
5. Any helper functions.
6. The `device_method_t` table, the `driver_t` structure, the `DRIVER_MODULE` registration, and the `MODULE_VERSION` declaration. Modern FreeBSD drivers no longer declare a static `devclass_t` variable; the current `DRIVER_MODULE` signature takes five arguments (name, bus, driver, event handler, event handler argument) and the bus code manages the device class for you.

There is a visible rhythm to a well-organised driver file that experienced FreeBSD readers pick up on immediately. Methods come before the tables that reference them. Static helpers come near the methods that use them. The registration macros come last, so that the file reads as a linear story from dependencies through functions to registration.

### 37.5.4 A Minimal Driver File

For orientation, here is a minimal shape for `mydev.c`. It is not complete, but it shows the structural elements that a reviewer will expect to see. You have seen the mechanics of each of these macros in earlier chapters; here we are focusing on how they arrange themselves on the page.

```
/* -
 * SPDX-License-Identifier: BSD-2-Clause
 *
 * Copyright (c) 2026 Your Name <you@example.com>
```

```
*
* Redistribution and use in source and binary forms, with or
* without modification, are permitted provided that the
* following conditions are met:
* 1. Redistributions of source code must retain the above
* copyright notice, this list of conditions and the following
* disclaimer.
* 2. Redistributions in binary form must reproduce the above
* copyright notice, this list of conditions and the following
* disclaimer in the documentation and/or other materials
* provided with the distribution.
*
* THIS SOFTWARE IS PROVIDED BY THE AUTHORS ``AS IS'' AND ANY
* EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,
* THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
* PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
* AUTHORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
* SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
* NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
* LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
* HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
* CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR
* OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE,
* EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
*/

#include <sys/cdefs.h>
#include <sys/param.h>
#include <sys/system.h>
#include <sys/kernel.h>
#include <sys/module.h>
#include <sys/bus.h>

#include <machine/bus.h>
#include <machine/resource.h>
#include <sys/rman.h>

#include <dev/pci/pci.h>
#include <dev/pci/pcivar.h>

#include <dev/mydev/mydev.h>

static int mydev_probe(device_t dev);
static int mydev_attach(device_t dev);
```

```
static int mydev_detach(device_t dev);

static int
mydev_probe(device_t dev)
{
 /* match your PCI vendor/device ID here */
 return (ENXIO);
}

static int
mydev_attach(device_t dev)
{
 /* allocate resources, initialise the device */
 return (0);
}

static int
mydev_detach(device_t dev)
{
 /* release resources, quiesce the device */
 return (0);
}

static device_method_t mydev_methods[] = {
 DEVMETHOD(device_probe, mydev_probe),
 DEVMETHOD(device_attach, mydev_attach),
 DEVMETHOD(device_detach, mydev_detach),
 DEVMETHOD_END
};

static driver_t mydev_driver = {
 "mydev",
 mydev_methods,
 sizeof(struct mydev_softc),
};

DRIVER_MODULE(mydev, pci, mydev_driver, 0, 0);
MODULE_VERSION(mydev, 1);
MODULE_DEPEND(mydev, pci, 1, 1, 1);
```

Several things are worth noticing. The copyright header uses the `/*` - opening marker, which the automatic licence-harvesting script recognises. The `SPDX` line names the licence explicitly. The indentation is tabs, not spaces, as mandated by `style(9)`. The function declarations are tab-separated, also per `style(9)`. The `DRIVER_MODULE` and related macros appear at the bottom, in the order the build system expects. This is the

shape a reviewer will expect to see.

### 37.5.5 The Module Makefile

The Makefile for the module is usually tiny. Here is a realistic example, modelled on the one at `/usr/src/sys/modules/et/Makefile`:

```
.PATH: ${SRCTOP}/sys/dev/mydev

KMOD= mydev
SRCS= mydev.c
SRCS+= bus_if.h device_if.h pci_if.h

.include <bsd.kmod.mk>
```

Several conventions are encoded in this short file.

`SRCTOP` is a build-system variable that points to the top of the source tree. Using it means the Makefile works regardless of where in the tree the build is invoked from. Do not hardcode `/usr/src`.

`KMOD` names the module. This is what `kldload` uses. Match it to the driver name.

`SRCS` lists the source files. The `.c` files are your driver sources. The `.h` files that look like `bus_if.h` and `pci_if.h` are not regular headers; they are auto-generated by the build system from the method definitions in the corresponding `.m` files. You list them so that the build system knows to generate them before compiling your driver. Include `device_if.h` because every driver uses `device_method_t`; include `bus_if.h` if your driver uses `bus_*` methods; include `pci_if.h` if it is a PCI driver; and so on.

`bsd.kmod.mk` is the standard kernel-module build infrastructure. Including it at the end gives you all the build rules you need.

A few additional conventions apply:

- Do not add copyright headers to trivial Makefiles. The tree convention is that tiny Makefiles such as this one are treated as mechanical files and do not carry licences. Real Makefiles with substantial logic do carry copyright headers.
- Do not use GNU make features. FreeBSD's base build system uses the in-tree BSD make, not GNU make.
- Keep indentation with tabs, not spaces, for rule bodies.

### 37.5.6 The Header File

If your driver has a header file for internal declarations, place it in the same directory as the `.c` file. The convention is to name the internal header `<driver>.h` and any hardware register definitions `<driver>reg.h`. Keep the header's scope narrow. It should declare structures and constants that are used across multiple `.c` files within the driver or that are needed for interoperation with closely related subsystems. It should not leak driver-internal details into the broader kernel namespace.

The header begins with the same copyright header as the `.c` file, followed by the standard include guard:

```
#ifndef _DEV_MYDEV_MYDEV_H_
#define _DEV_MYDEV_MYDEV_H_

/* header contents */

#endif /* _DEV_MYDEV_MYDEV_H_ */
```

The include guard name follows the convention of the full path, in uppercase, with slashes and dots replaced by underscores, and with a leading and trailing underscore. The convention is consistent across the tree and a reviewer will spot deviations.

### 37.5.7 Following style(9): The Short Summary

The complete FreeBSD coding style is documented in `/usr/src/share/man/man9/style.9`. You should read that manual page before submitting a driver, and at least skim it periodically as your own style matures. Here we will pull out the points that most commonly trip up first-time contributors.

#### 37.5.7.1 Indentation and Line Width

Indentation uses real tabs, with a tab stop of 8 columns. Second and subsequent levels of indentation that are not aligned to a tab stop use 4 spaces of additional indent. Line width is 80 columns; a few exceptions are allowed when breaking a line would make it less readable or would break something that gets grepped for, such as a panic message.

#### 37.5.7.2 Copyright Header Form

The copyright header uses the `/*` - opening marker. This marker is magic. An automated tool harvests licences from the tree by looking for multi-line comments that start in column 1 with `/*` -. Using the marker flags the block as a licence; using a regular `/*` does not. Immediately after `/*` -, the next significant line should be `SPDX-License-Identifier`: followed by the SPDX licence code, such as `BSD-2-Clause`. Then come one or more Copyright lines. Then the licence text.

#### 37.5.7.3 Function Declarations and Definitions

Function return type and storage class go on the line above the function name. The function name starts in column 1. Arguments fit on the same line as the name unless that would exceed 80 columns, in which case subsequent arguments are aligned to the opening parenthesis.

Correct:

```
static int
mydev_attach(device_t dev)
{
```

```
struct mydev_softc *sc;

sc = device_get_softc(dev);
return (0);
}
```

Incorrect, as a reviewer would immediately flag:

```
static int mydev_attach(device_t dev) {
 struct mydev_softc *sc = device_get_softc(dev);
 return 0;
}
```

The differences look small: the position of the return type, the position of the opening brace, the use of spaces instead of tabs, the declaration with initialisation on a single line, and the missing parentheses around the return value. Every one of those differences violates `style(9)`. Collectively they make the function look out of place in the tree. A reviewer will ask you to fix them, and fixing them after the fact is more work than writing them correctly the first time.

#### 37.5.7.4 Variable Names and Identifier Conventions

Use lower-case identifiers with underscores rather than camelCase. `mydev_softc`, not `MydevSoftc` or `mydev-Softc`. Functions follow the same convention.

Constants and macros are in upper case with underscores: `MYDEV_REG_CONTROL`, `MYDEV_FLAG_INITIALIZED`.

Global variables are rare in drivers; prefer per-device state in the `_softc`. When a global is unavoidable, give it a name prefixed with the driver name to avoid collision with the rest of the kernel.

#### 37.5.7.5 Return-Value Parentheses

FreeBSD style requires that return expressions be parenthesised: `return (0);` not `return 0;`. This is a convention going back to the original BSD kernel and is enforced reasonably strictly.

#### 37.5.7.6 Comments

Multi-line comments use the form:

```
/*
 * This is the opening of a multi-line comment. Make it real
 * sentences. Fill the lines to the column 80 mark so the
 * comment reads like a paragraph.
 */
```



Single-line comments can use either the traditional `/* ... */` or `// ...` form. Be consistent within a file; do not mix styles.

Comments should explain why, not what. `/* iterate over the array */` is not useful when the reader can see the loop. `/* the hardware requires a read-back to flush the write before we proceed */` is useful because it explains a non-obvious constraint.

### 37.5.7.7 Error Messages

Use `device_printf(dev, "message\n")` for device-specific log output. Do not use `printf` directly from a driver if you have a `device_t` handy; `device_printf` prefixes the message with the driver and unit number, which is what every reader of kernel logs expects to see.

Error messages that get grepped for should stay on a single line even if they exceed 80 columns. The `style(9)` manual page is explicit about this.

### 37.5.7.8 Magic Numbers

Do not use magic numbers in the body of the code. Hardware register offsets, bit masks, and status codes should be named constants in the `<driver>reg.h` header. This makes the code readable and also makes it easy to patch the register definitions when, inevitably, you discover that something was slightly off.

## 37.5.8 Using tools/build/checkstyle9.pl

The project ships an automated style checker at `/usr/src/tools/build/checkstyle9.pl`. It is a Perl script that reads source files and warns about common style violations. It is not perfect, and some warnings will be false positives or reflect conventions that the script does not quite get right, but it catches a large fraction of the easy mistakes.

Run it against your driver before submission:

```
/usr/src/tools/build/checkstyle9.pl sys/dev/mydev/mydev.c
```

You will see output like:

```
mydev.c:23: missing blank line after variable declarations
mydev.c:57: spaces not tabs at start of line
mydev.c:91: return value not parenthesised
```

Fix each warning. Re-run. Repeat until the output is clean.

The `CONTRIBUTING.md` file is explicit about this: “Run `tools/build/checkstyle9.pl` on your Git branch and eliminate all errors.” Reviewers do not want to be the style checker for you. Submitting code that has not been run through `checkstyle9.pl` wastes their time.

### 37.5.9 Using `indent(1)` Carefully

FreeBSD also ships `indent(1)`, a C source reformatter. It can reformat a file to conform to parts of `style(9)` automatically. It is useful but it is not magical. `indent(1)` handles some style rules well, such as tab-based indentation and brace placement, but it handles other rules poorly or not at all, and in some cases it makes things worse by reformatting comments or function signatures in ways that violate `style(9)` even though the input was correct.

Treat `indent(1)` as a rough first pass rather than as a canonical formatter. Run it on a file to get close to conforming, then read the output carefully and fix anything it got wrong. Do not run it on existing tree files as part of an unrelated patch; mixing style changes with functional changes is a review anti-pattern.

### 37.5.10 Commit Messages

A good commit message does two things. First, it tells the reader at a glance what the commit does. Second, it tells the reader in more detail why the commit does it. The subject line is the first; the body is the second.

The subject line conventions in the FreeBSD tree look like this:

```
subsystem: Short description of the change
```

The `subsystem` prefix tells the reader which part of the tree is affected. For a driver submission, the prefix is typically the driver name:

```
mydev: Add driver for MyDevice FC100
```

The first word after the colon is capitalised, and the description is a fragment, not a full sentence. The subject line is capped around 50 characters, with 72 as a hard limit. Look at recent commits in the tree with `git log --oneline` to see the pattern:

```
rge: add disable_aspm tunable for PCIe power management
asmc: add automatic voltage/current/power/ambient sensor detection
tcp: use RFC 6191 for connection recycling in TIME-WAIT
pf: include all elements when hashing rules
```

The body of the commit message comes after a blank line. It explains the change in more detail: what the change does, why it is needed, what hardware or scenario it affects, and any considerations a future reader might need to know. Wrap the body at 72 columns.

A good commit message for a driver submission might look like this:

```
mydev: Add driver for FooCorp FC100 sensor board
```

```
This driver supports the FooCorp FC100 series of PCI-attached
environmental sensor boards, which expose a simple command and
```

status interface over a single BAR. The driver implements probe/attach/detach following the Newbus conventions, exposes a character device for userland communication, and supports sysctl-driven sampling configuration.

The FC100 is documented in the FooCorp Programmer's Reference Manual version 1.4, which the maintainer has on file. Tested on amd64 and arm64 against a hardware sample; no errata were observed during the test period.

Reviewed by:     someone@FreeBSD.org  
MFC after:    2 weeks

Several pieces of that message are standard. Reviewed by: names the committer who signed off on the review. MFC after: suggests a period before the commit can be merged from CURRENT back to STABLE (MFC stands for Merge From Current). You do not fill in these lines yourself as a contributor; the committer who commits your patch will add them.

What you do write is the body: the descriptive paragraphs that explain the change. Write them as if you are writing them for a future reader who will see the commit in `git log` five years from now and wonder what it was about. That reader is you, possibly, or it is someone maintaining your driver after you have moved on. Make the commit message kind to them.

### 37.5.11 Signed-off-by and the Developer Certificate of Origin

For GitHub pull requests in particular, the CONTRIBUTING.md file asks that commits include a Signed-off-by: line. This line certifies the Developer Certificate of Origin at <https://developercertificate.org/>, which in plain terms is a statement that you have the right to contribute the code under the project's licence.

Adding a Signed-off-by: is easy:

```
git commit -s
```

The -s flag adds a line of the form:

```
Signed-off-by: Your Name <you@example.com>
```

to the commit message. Use the same name and email you use for the commit author line.

### 37.5.12 What a Complete Submission-Ready Tree Looks Like

After all of this, your driver's tree within the FreeBSD source tree should look roughly like this:

```
/usr/src/sys/dev/mydev/
 mydev.c
 mydev.h (optional)
 mydevreg.h (optional but recommended)

/usr/src/sys/modules/mydev/
 Makefile

/usr/src/share/man/man4/
 mydev.4
```

And you should be able to build the module with:

```
cd /usr/src/sys/modules/mydev
make obj
make depend
make
```

And validate the manual page with:

```
mandoc -Tlint /usr/src/share/man/man4/mydev.4
```

And run the style checker with:

```
/usr/src/tools/build/checkstyle9.pl /usr/src/sys/dev/mydev/mydev.c
```

If all three of these complete without errors, your driver is mechanically ready for submission. There is still the licensing, the manual page content, the testing, and the patch generation to cover, which we will do in the next sections. But the basic layout is now in place, and a reviewer opening the patch will find that the file names, the file layouts, the style, and the build integration match what they expect to see in the tree.

### 37.5.13 Common Mistakes in Section-2 Preparation

Before we close this section, let us collect the most common preparation mistakes that first-time contributors make. Treat this as a quick self-check before moving on to Section 3.

- Files in the wrong location. The driver lives under `/usr/src/sys/dev/<driver>/`, not at the top of `/usr/src/sys/`. The module Makefile lives under `/usr/src/sys/modules/<driver>/`. The manual page lives under `/usr/src/share/man/man4/`.
- File names that do not match the driver. If the driver is `mydev`, the main source file is `mydev.c`, not `main.c` or `driver.c`.
- Missing or incorrect copyright header. The header uses `/* -` as the opening marker, the SPDX identifier comes first, and the licence text matches one of the project's accepted licences.

- Spaces instead of tabs. `style(9)` is explicit about tabs, and the style checker will flag space indentation immediately.
- Missing parentheses on return expressions. A recurring small mistake that the style checker will catch.
- Missing blank line between variable declarations and code. Another small convention that the style checker catches.
- Commit message that does not follow the subsystem: Short description form. The reviewer will ask you to rewrite it.
- Trailing whitespace. The `CONTRIBUTING.md` file explicitly flags trailing whitespace as something reviewers dislike.
- Makefile that hardcodes `/usr/src` instead of using `${SRCTOP}`.

Every one of these is an easy fix when you know to look for it. Every one of them adds one round of back-and-forth to a review when you do not. The goal of this section was to give you the knowledge to catch all of them before you submit.

### 37.5.14 Wrapping Up Section 2

Getting a driver ready for submission is less about cleverness than about attention to detail. The file layout, the style, the copyright header, the Makefile, the commit message: each of them has a conventional form, and a driver whose files match those conventions is a driver whose reviewer's first impression is "this looks right." That first impression is worth more than any other single factor in determining how many rounds of review the patch will need.

We have not yet talked about the licence itself in detail, nor about the manual page, nor about testing. Those are the subjects of the next three sections. But the mechanical preparation of the source tree, which is where first-time contributors most often stumble, is now covered.

Let us turn next to licensing and the legal considerations that frame every FreeBSD contribution.

## 37.6 Section 3: Licensing and Legal Considerations

### 37.6.1 Why Licensing Matters Up Front

The easiest way to get your driver rejected is to get the licence wrong. Licensing is not a procedural preference in FreeBSD; it is a foundation of how the project works. The FreeBSD operating system ships under a combination of permissive licences that users can rely on without surprise. A contribution that carries an incompatible licence, or an unclear licence, or no licence at all, cannot be accepted into the tree, no matter how excellent the code is in every other respect.

This is not legal formalism for its own sake. It is a practical necessity. FreeBSD is used in many environments, including commercial products that are shipped to millions of users. Those users rely on the project's licence to understand their obligations. A single file in the tree that carries a surprise licence could expose the entire project's downstream users to obligations they did not sign up for. The project cannot accept that risk.

For you as a contributor, the practical takeaway is this: get the licence right up front. It is easier, by a wide margin, than trying to fix it after the review process has flagged it. This section walks through what the project

accepts, what it does not accept, and how to structure the copyright header so that your submission sails through the licensing check.

### 37.6.2 What Licences FreeBSD Accepts

The FreeBSD Project prefers, as a default, the two-clause BSD licence, often written as BSD-2-Clause. This is the permissive licence that most of FreeBSD itself ships under, and it is the default recommendation for new code. BSD-2-Clause allows redistribution in source and binary form, with or without modification, as long as the copyright notice and the licence text are preserved. It imposes no requirement on downstream users to distribute their source, no requirement for compatibility notices, and no patent grant clause that might complicate commercial use.

The three-clause BSD licence, BSD-3-Clause, is also accepted. It adds one clause prohibiting the use of the author's name in endorsements. Some older FreeBSD code uses this form, and it is equivalent for most practical purposes.

A few other permissive licences appear in the tree for specific files contributed under them historically. The MIT-style licences and the ISC licence both appear in places. The Beerware licence, a whimsical permissive licence introduced by Poul-Henning Kamp, also appears in a few files such as `/usr/src/sys/dev/led/led.c`. These licences are compatible with FreeBSD's overall licensing scheme and are accepted when they accompany specific code contributed under them.

For a new driver that you are writing yourself, the right default is BSD-2-Clause. Unless you have a specific reason to use a different licence, use BSD-2-Clause. It is the licence your reviewers will expect, and any deviation from it will trigger a conversation you probably do not want to have on a first submission.

### 37.6.3 What Licences FreeBSD Does Not Accept

Several licences are not compatible with the FreeBSD source tree and code under them cannot be merged. The most common ones that first-time contributors sometimes try to use are:

- The GNU General Public Licence (GPL), in any version. GPL code is not compatible with FreeBSD's licensing model because it imposes source-distribution obligations on downstream users that the rest of the tree does not carry. FreeBSD does include some GPL-licensed components in userland, such as the GNU Compiler Collection, but these are specific historical cases and are not a template for new contributions. Driver code, in particular, is not accepted under GPL.
- The Lesser GPL (LGPL). Same reasoning as GPL.
- The Apache Licence, version 2 or otherwise, unless there is specific discussion and approval. The Apache Licence includes a patent grant clause that interacts in complicated ways with the permissive BSD licences. Some Apache-licensed code is accepted in specific contexts but it is not the default for new code.
- The MIT licence in its various forms, while technically permissive, is not a default choice for FreeBSD. If you have a specific reason to use MIT, discuss it with a reviewer before submission.
- Anything proprietary. The tree cannot accept code whose licence restricts redistribution or modification.
- Code with unclear licensing, including code copied from other projects whose licence is not known, code generated by tools whose licensing terms are unclear, and code contributed without a clear licence declaration.

If you are porting or adapting code from another open-source project, check the licence of the source project carefully before you start. Bringing in code from a GPL-licensed project into your driver, even a small function, contaminates the driver and makes it unable to enter the FreeBSD tree.

### 37.6.4 The Copyright Header in Detail

The copyright header at the top of every source file in the tree has a specific structure, documented in `style(9)`. Let us walk through a complete header and examine every part.

```
/*-
 * SPDX-License-Identifier: BSD-2-Clause
 *
 * Copyright (c) 2026 Your Name <you@example.com>
 *
 * Redistribution and use in source and binary forms, with or
 * without modification, are permitted provided that the
 * following conditions are met:
 * 1. Redistributions of source code must retain the above
 * copyright notice, this list of conditions and the following
 * disclaimer.
 * 2. Redistributions in binary form must reproduce the above
 * copyright notice, this list of conditions and the following
 * disclaimer in the documentation and/or other materials
 * provided with the distribution.
 *
 * THIS SOFTWARE IS PROVIDED BY THE AUTHORS ``AS IS'' AND ANY
 * EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,
 * THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
 * PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
 * AUTHORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
 * SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
 * NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
 * LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
 * CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR
 * OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE,
 * EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 */
```

The opening `/* -` is not a typo. The dash after the star is significant. An automated script in the tree harvests licence information from files by looking for multi-line comments that start in column 1 with the sequence `/* -`. Using `/*` flags the block as a licence; using plain `/*` does not. `style(9)` is explicit: if you want the tree's licence harvester to pick up your licence correctly, use `/* -` for the opening line.

Immediately after the opening is the `SPDX-License-Identifier` line. `SPDX` is a standardised vocabulary for

describing licences in machine-readable form. The line tells the harvester which licence the file is under, and it does so in a form that cannot be misinterpreted. Use `BSD-2-Clause` for a two-clause BSD licence, `BSD-3-Clause` for a three-clause BSD licence. For other licences, consult the SPDX identifier list at <https://spdx.org/licenses/>. Do not invent identifiers.

The copyright line names the year and the copyright holder. Use your full legal name, or the name of your employer if you are contributing work done under employment, followed by an email address that is stable enough to contact you years from now. If you are contributing as an individual, use your personal email rather than a throwaway address.

Multiple copyright lines may be present if the file has had multiple authors. When you add a copyright line, add it at the bottom of the existing list, not at the top. Do not delete anyone else's copyright line. The existing attributions are legally significant.

The licence text itself follows. The text reproduced above is the standard `BSD-2-Clause` text. Do not modify it. The wording is legally specific, and changing it, even in a way that seems clearer, may make the licence legally distinct from what the project accepts.

Finally, there is a blank line after the closing `*/`, before the code begins. This blank line is a convention in the tree and is called out in `style(9)`. Its purpose is visual.

### 37.6.5 Reading Existing Headers to Build Intuition

The best way to internalise the licence header conventions is to look at real headers in the tree. Open `/usr/src/sys/dev/null/null.c` and read its header. Open `/usr/src/sys/dev/led/led.c` and read its header (which is under the Beerware licence, an unusual but accepted case). Open a couple of network drivers under `/usr/src/sys/dev/re/` or `/usr/src/sys/dev/e1000/` and read theirs. Within fifteen minutes you will have absorbed the pattern.

A few things you will notice:

- Some older files in the tree do not have SPDX identifiers. These predate the SPDX convention. For new contributions, use SPDX.
- Some older files still have a `$FreeBSD$` tag near the top. This was a CVS-era marker that is no longer active since the project moved to git. New contributions do not include `$FreeBSD$` tags.
- Some files have multiple copyright lines spanning multiple contributors across years. This is normal and correct. When you add a copyright line to an existing file, append it.
- A few files have non-standard licences (Beerware, MIT-style, ISC). These are historical and are accepted on a case-by-case basis. Do not use them as templates for new contributions.

### 37.6.6 Derivative Works and External Code

If your driver is entirely your own work, the header is straightforward. If it includes code derived from another project, the picture is more complicated.

Any code you copy or adapt from another project carries that project's licence along with it. If the project's licence is BSD-compatible, you can use the code, but you must preserve the original copyright notice and make the adaptation visible. If the licence is not BSD-compatible, such as GPL, you cannot use the code at all.



The convention in the tree for derivative works is to preserve the original copyright line and add your own as a separate line:

```
/*-
 * SPDX-License-Identifier: BSD-2-Clause
 *
 * Copyright (c) 1998 Original Author <original@example.com>
 * Copyright (c) 2026 Your Name <you@example.com>
 *
 * [licence text]
 */
```

If the original licence is BSD-3-Clause and you are contributing additions under BSD-2-Clause, the file is effectively BSD-3-Clause overall, because the three-clause requirement flows through derivative works. Use the more restrictive of the two licences in the SPDX identifier, or keep separate licensing at a per-section level if the code is clearly separable. When in doubt, ask a reviewer.

If the code is taken from a specific external source whose origin is relevant, mention it in a comment near the relevant function:

```
/*
 * Adapted from the NetBSD driver at
 * src/sys/dev/foo/foo.c, revision 1.23.
 */
```

This helps reviewers and future maintainers understand the provenance. It also helps anyone tracking bugs back to their upstream fix.

### 37.6.7 Adapted Code from Vendor Sources

A common scenario for hardware drivers is that the vendor provides sample code or a reference driver under some licence. If the vendor's code is under a BSD-compatible licence, you may be able to use it directly, possibly with adaptation, preserving the vendor's copyright. Read the vendor's licence carefully. If the licence is not BSD-compatible, you cannot use the vendor's code in a driver destined for the tree. You may be able to use the vendor's documentation as a reference for independently implementing the driver, but you cannot lift code.

If the vendor provides documentation under a non-disclosure agreement (NDA), the situation is more delicate still. An NDA typically prohibits you from disclosing the documentation. It may not prohibit you from using the documentation to write code, but the resulting code must be your own work, not a copy of any code the vendor provided. Be scrupulous about keeping this line clear. If there is any doubt, do not proceed without legal advice.

### 37.6.8 Code You Did Not Write but Are Submitting

If you are submitting code that someone else wrote, such as a colleague's contribution, you need their explicit permission and their copyright line in the header. You cannot contribute code on someone else's behalf without

their knowledge. The Signed-off-by: line that the project asks for is, in part, a mechanism for tracking this; the Developer Certificate of Origin that the line certifies includes a statement that you have the right to contribute the code.

If you are a company employee contributing work done under employment, your employer typically holds the copyright, not you. The copyright line should name the employer. Many companies have internal processes for approving open-source contributions; follow those before submitting. Some companies prefer to have their employees sign a contributor licence agreement (CLA) with the FreeBSD Foundation for clarity; if yours does, coordinate with your company before submitting.

### 37.6.9 Adding Licence Headers to an Existing Driver

If you are retrofitting a driver you have already written but never prepared for submission, you will need to add the proper header to each file. The steps are:

1. Decide on the licence. For a new driver, use BSD-2-Clause.
2. Write the SPDX identifier line.
3. Write the copyright line with your name, email, and the year of first creation.
4. Paste in the standard BSD-2-Clause licence text.
5. Verify that the opening is `/* -` and the file starts in column 1.
6. Verify that there is a single blank line after the closing `*/`.
7. Repeat for each file: the `.c` files, the `.h` files, the manual page (where the licence appears as `.\"` - style comments rather than `/* -` style), and any other files that contain substantial content.

For the Makefile, as noted earlier, a licence header is conventionally omitted for trivial files. The module Makefile shown in Section 2 is trivial enough that no header is required.

### 37.6.10 Validating the Header

There is no single automated tool that validates every aspect of a FreeBSD copyright header. The `check-style9.pl` script catches some kinds of formatting errors near the header. The licence harvester in the tree works on the `/* -` marker and the SPDX line. The most reliable validation, however, is to compare your header directly against a known-good header from a recent commit in the tree, such as the header in `/usr/src/sys/dev/null/null.c` or any recently added driver.

Build a small habit: when you open a new source file, paste in a known-good header as the first action. This prevents the easy mistake of forgetting the header entirely and also ensures that the shape is right from the start.

### 37.6.11 Wrapping Up Section 3

Licensing is one of the places where getting it right up front saves enormous amounts of time. The FreeBSD Project accepts BSD-2-Clause, BSD-3-Clause, and a few other permissive licences for historical files. New contributions should default to BSD-2-Clause. The copyright header uses a specific form, opening with `/* -`, followed by an SPDX identifier, followed by one or more copyright lines, followed by the standard licence text. Code derived from other projects carries its original licence obligations forward, and derivative work

must preserve the original attributions. Code you did not write yourself requires the author's permission and attribution.

With the legal side handled, we can turn to the manual page. Every driver in the tree ships with a manual page, and writing a good one is one of the places where first-time contributors most often underestimate the effort involved. Section 4 walks through the conventions and provides a template you can adapt.

## 37.7 Section 4: Writing a Manual Page for Your Driver

### 37.7.1 Why the Manual Page Matters

The manual page is the user-facing half of your driver. When someone finds your driver in the tree and wants to know what it does, they will not read the source code. They will run `man 4 mydev`. What they see will be, for most of them, the only documentation they ever have of your driver. If the manual page is clear, complete, and well-organised, users will understand what the driver supports, how to use it, and what its limitations are. If the manual page is missing, sparse, or poorly organised, users will be confused, they will file bug reports that are really documentation problems, and they will, reasonably, form a negative impression of the driver.

From the project's point of view, the manual page is a first-class artifact of the contribution. A driver without a manual page cannot be merged. A driver with a poor manual page will be delayed at review until the manual page is brought up to standard. You should think of the manual page as part of the driver, not as an afterthought.

From a practical point of view, writing the manual page is often a useful discipline in its own right. The act of explaining to a user what the driver does, what hardware it supports, what tunables it exposes, and what its known limitations are, forces you to articulate those things clearly. It is not uncommon for a well-written manual page to surface questions that the driver's design had not yet resolved. Writing the manual page is therefore part of the work of finishing the driver, not a step after the driver is done.

### 37.7.2 Manual Page Sections: A Quick Orientation

The manual pages in FreeBSD are organised into numbered sections. The sections are:

- Section 1: General user commands.
- Section 2: System calls.
- Section 3: Library calls.
- Section 4: Kernel interfaces (devices, device drivers).
- Section 5: File formats.
- Section 6: Games.
- Section 7: Miscellaneous and conventions.
- Section 8: System administration and privileged commands.
- Section 9: Kernel internals (APIs and subsystems).

Your driver belongs in Section 4. The manual-page file goes under `/usr/src/share/man/man4/` and is conventionally named `<driver>.4`, for example `mydev.4`. The `.4` suffix is the manual-page convention; it marks the file as a section-4 page.

The file itself is written in the mdoc macro language, not in plain text. Mdoc is a structured macro set that produces formatted manual pages from a source file that is more or less human-readable. The project's style for

mdoc is documented in `/usr/src/share/man/man5/style.mdoc.5`; you should read that file before writing your first manual page, though much of what it says will make more sense after you have tried to write one.

### 37.7.3 The Structure of a Section-4 Manual Page

A section-4 manual page has a well-established structure. The following sections appear in roughly this order:

1. **NAME:** The driver name and a one-line description.
2. **SYNOPSIS:** How to include the driver in the kernel or load it as a module.
3. **DESCRIPTION:** What the driver does, in prose.
4. **HARDWARE:** The list of hardware the driver supports. This section is required for section-4 pages and is drawn verbatim into the Release Hardware Notes.
5. **LOADER TUNABLES, SYSCTL VARIABLES:** If the driver exposes tunables, document them here.
6. **FILES:** The device nodes and any configuration files.
7. **EXAMPLES:** Usage examples, when relevant.
8. **DIAGNOSTICS:** Explanations of the driver's log messages.
9. **SEE ALSO:** Cross-references to related manual pages and documents.
10. **HISTORY:** When the driver first appeared.
11. **AUTHORS:** The primary author(s) of the driver.
12. **BUGS:** Known issues and limitations.

Not every section is required for every driver. For a simple driver, **NAME**, **DESCRIPTION**, **HARDWARE**, **SEE ALSO**, and **HISTORY** are the minimum. For a more complex driver, add the others as relevant.

### 37.7.4 A Minimum Working Manual Page

Here is a complete, working section-4 manual page for a hypothetical `mydev` driver. Save it as `mydev.4`, run `mandoc -T` lint on it, and you will see it pass clean. This is the kind of manual page you can adapt to your own driver.

```
.\"-
.\" SPDX-License-Identifier: BSD-2-Clause
.\"
.\" Copyright (c) 2026 Your Name <you@example.com>
.\"
.\" Redistribution and use in source and binary forms, with or
.\" without modification, are permitted provided that the
.\" following conditions are met:
.\" 1. Redistributions of source code must retain the above
.\" copyright notice, this list of conditions and the following
.\" disclaimer.
.\" 2. Redistributions in binary form must reproduce the above
.\" copyright notice, this list of conditions and the following
.\" disclaimer in the documentation and/or other materials
.\" provided with the distribution.
```

```
.\"
.\" THIS SOFTWARE IS PROVIDED BY THE AUTHORS ``AS IS'' AND ANY
.\" EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,
.\" THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
.\" PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
.\" AUTHORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
.\" SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
.\" NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
.\" LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
.\" HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
.\" CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR
.\" OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE,
.\" EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
.\"
.Dd April 20, 2026
.Dt MYDEV 4
.Os
.Sh NAME
.Nm mydev
.Nd driver for FooCorp FC100 sensor boards
.Sh SYNOPSIS
To compile this driver into the kernel,
place the following line in your
kernel configuration file:
.Bd -ragged -offset indent
.Cd "device mydev"
.Ed
.Pp
Alternatively, to load the driver as a
module at boot time, place the following line in
.Xr loader.conf 5 :
.Bd -literal -offset indent
mydev_load="YES"
.Ed
.Sh DESCRIPTION
The
.Nm
driver provides support for FooCorp FC100 series PCI-attached
environmental sensor boards.
It exposes a character device at
.Pa /dev/mydev0
that userland programs can open, read, and write using standard
system calls.
.Pp
```

Each attached board is enumerated with an integer unit number beginning at 0.

The driver supports probe, attach, and detach through the standard Newbus framework.

.Sh HARDWARE

The

.Nm

driver supports the following hardware:

.Pp

.Bl -bullet -compact

.It

FooCorp FC100 rev 1.0

.It

FooCorp FC100 rev 1.1

.It

FooCorp FC200 (compatibility mode)

.El

.Sh FILES

.Bl -tag -width ".Pa /dev/mydev0"

.It Pa /dev/mydev0

First unit of the driver.

.El

.Sh SEE ALSO

.Xr pci 4

.Sh HISTORY

The

.Nm

driver first appeared in

.Fx 15.0 .

.Sh AUTHORS

.An -nosplit

The

.Nm

driver and this manual page were written by

.An Your Name Aq Mt you@example.com .

That manual page is a complete, valid section-4 page. It is short because the hypothetical driver is simple. A more complex driver would have larger DESCRIPTION, HARDWARE, and possibly LOADER TUNABLES, SYSCTL VARIABLES, DIAGNOSTICS, and BUGS sections. But the skeleton is the same.

Let us walk through the parts that are most frequently misunderstood.

### 37.7.5 The Header Block

The header block at the top is a set of comment lines that begin with `.\`. These are mdoc comments. They are not rendered in the manual page output. They exist to carry the copyright header and any notes to future editors.

The opening marker is `.\` - with a dash, equivalent to the `/*` - in C files. The licence harvester recognises it.

The `.Dd` macro sets the document date. It is formatted as month, day, year with full month name. The project's mdoc style is to update `.Dd` whenever the content of the manual page changes meaningfully. Do not bump the date for trivial changes such as whitespace fixes, but do bump it for anything semantic.

The `.Dt` macro sets the document title. The convention is the driver name in uppercase, followed by the section number: `MYDEV 4`.

The `.Os` macro emits the operating system identifier in the footer. Use it bare; mdoc will fill in the right thing from build-time macros.

### 37.7.6 The NAME Section

The `.Sh NAME` macro opens the NAME section. The content is a pair of macros:

```
.Nm mydev
.Nd driver for FooCorp FC100 sensor boards
```

`.Nm` sets the name of the thing being documented. Once set, `.Nm` with no arguments elsewhere in the page expands to the name, which is how we avoid repeating the driver name over and over.

`.Nd` is the short description, a sentence-fragment of the form “driver for ...” or “API for ...” or “device for ...”. Do not capitalise the first word or add a period at the end.

### 37.7.7 The SYNOPSIS Section

For a driver, the SYNOPSIS typically shows two things: how to compile the driver into the kernel as a built-in, and how to load it as a module. The built-in form uses `.Cd` for a kernel configuration line. The loadable form shows the `_load="YES"` entry for `loader.conf`.

If the driver exposes a header file that userland programs must include, or if it exposes a library-like API, the SYNOPSIS can also include `.In` for the include directive and `.Ft/.Fn` for function prototypes. See `/usr/src/share/man/man4/led.4` for an example of a driver manual page whose SYNOPSIS shows function prototypes.

### 37.7.8 The DESCRIPTION Section

The DESCRIPTION section is where you explain, in prose, what the driver does. Write it for a user who has installed FreeBSD, has the hardware in front of them, and wants to know what the driver offers.

Keep the paragraphs focused. Use `.Pp` to separate paragraphs. Use `.Nm` to refer to the driver, not the driver name typed out. Use `.Pa` for path names, `.Xr` for cross-references to other manual pages, `.Ar` for argument names, and `.Va` for variable names.

Describe the driver's behaviour, its lifecycle (probe, attach, detach), its device-node structure, and any concepts a user needs to understand before interacting with it. Do not document internal implementation details here; the source code is the right place for those.

### 37.7.9 The HARDWARE Section

The HARDWARE section is required for section-4 pages. This is the section that gets drawn verbatim into the Release Hardware Notes, which is the document users consult to see whether their hardware is supported.

Several specific rules apply to this section. These rules are documented in `/usr/src/share/man/-man5/style.mdoc.5`:

- The introductory sentence should be in the form: “The `.Nm` driver supports the following :” followed by the list.
- The list should be a `.Bl -bullet -compact` list with one hardware model per `.It` entry.
- Each model should be named with its official commercial name, not an internal code name or chip revision.
- The list should include all hardware that is known to work, including revisions.
- The list should not include hardware that is known not to work; those belong in BUGS.

For a brand-new driver, the list may be short. That is fine. For a driver that has been in the tree for a while and has accumulated support for many hardware variants, the list grows over time as each new variant is tested.

### 37.7.10 The FILES Section

The FILES section lists the device nodes and configuration files that the driver uses. Use a `.Bl -tag` list with `.Pa` entries for the file names. For example:

```
.Sh FILES
.Bl -tag -width ".Pa /dev/mydev0"
.It Pa /dev/mydev0
First unit of the driver.
.It Pa /dev/mydev1
Second unit of the driver.
.El
```

Keep the `.Bl -tag -width` value wide enough to accommodate the longest path in the list. If the widths do not match, the list will render badly.

### 37.7.11 The SEE ALSO Section

The SEE ALSO section cross-references related manual pages. It is written as a list of `.Xr` cross-references, comma-separated, with the list sorted first by section number and then alphabetically within a section:



```
.Sh SEE ALSO
.Xr pci 4 ,
.Xr sysctl 8 ,
.Xr style 9
```

A driver's SEE ALSO typically includes the bus it attaches to (such as `pci(4)`, `usb(4)`, or `iicbus(4)`), any userland tool that interacts with it, and any section-9 APIs that are central to the driver's implementation.

### 37.7.12 The HISTORY Section

The HISTORY section says when the driver first appeared. For a brand-new driver that will first appear in the next release, write the release version as a placeholder:

```
.Sh HISTORY
The
.Nm
driver first appeared in
.Fx 15.0 .
```

The committer who commits your patch will verify the version number against the release schedule and may adjust it. That is fine.

### 37.7.13 The AUTHORS Section

The AUTHORS section names the primary authors of the driver. Use `.An -nosplit` at the top to tell mdoc not to split the author list across lines at name boundaries. Then use `.An` for each author with `.Aq Mt` for the email address.

```
.Sh AUTHORS
.An -nosplit
The
.Nm
driver was written by
.An Your Name Aq Mt you@example.com .
```

For a driver with multiple authors, list them in order of contribution, with the primary author first.

### 37.7.14 Validating the Manual Page

Once the manual page is written, validate it with `mandoc(1)`:

```
mandoc -Tlint /usr/src/share/man/man4/mydev.4
```

`mandoc -Tlint` runs the page through the mandoc parser in strict mode and reports any structural or semantic problems. Fix every warning. A clean `mandoc -Tlint` run is a prerequisite for submission.

You can also render the page to see what it looks like:

```
mandoc /usr/src/share/man/man4/mydev.4 | less -R
```

Read the rendered output as a user would. If something reads awkwardly, fix the source. If a cross-reference renders in a way you did not expect, check the macro usage. Read the output at least twice.

The project additionally recommends the `igor(1)` tool, available from the ports tree as `textproc/igor`. `igor` catches prose-level issues that `mandoc` does not, such as double spaces, unmatched quotes, and common prose mistakes. Install it with `pkg install igor` and run it on your page:

```
igor /usr/src/share/man/man4/mydev.4
```

Fix any warnings it produces.

### 37.7.15 The One-Sentence-Per-Line Rule

An important convention in FreeBSD mdoc pages is the one-sentence-per-line rule. Each sentence in the manual page source starts on a new line, regardless of line width. This is not about display formatting; mdoc will reflow the text for display. It is about source readability and about how `diff` shows changes. When changes are line-oriented, a diff of a manual page change shows which sentences changed; when sentences span lines, the diff is harder to read.

The `CONTRIBUTING.md` file is explicit about this:

Please be sure to observe the one-sentence-per-line rule so manual pages properly render. Any semantic changes to the manual pages should bump the date.

In practice this means you write:

```
The driver supports the FC100 family.
It attaches through the standard PCI bus framework.
Each unit exposes a character device under /dev/mydev.
```

Not:

```
The driver supports the FC100 family. It attaches through the
standard PCI bus framework. Each unit exposes a character device
under /dev/mydev.
```

The first form is conventional, the second is not.

### 37.7.16 Common Manual-Page Mistakes

Several mistakes recur in first-time submissions:

- Missing `HARDWARE` section. Section-4 pages are required to have one. If your driver supports no hardware yet (because it is a pseudo-device), document that explicitly.
- Period at the end of the `NAME` description. The `.Nd` description should be a fragment without a trailing period.
- Capitalised section headings spelled wrongly. The headings are canonical. Use `DESCRIPTION`, not `DESCRIPTIIONS`. Use `SEE ALSO`, not `See Also`. Use `HISTORY`, not `History`.
- Multi-sentence paragraphs without `.Pp` separation. Use `.Pp` between paragraphs in prose.
- Forgetting to bump `.Dd` when making semantic changes. If you change the content of the manual page, update the date.
- Using `.Cm` or `.Nm` where `.Ql` (literal quotation) or plain text is appropriate.
- Missing or malformed `.Bl/.El` pairs. Every list must be properly opened and closed.

Running `mandoc -Tlint` catches most of these. Running `igor` catches a few more. Reading the rendered output catches the remainder.

### 37.7.17 Looking at Real Manual Pages

Before you finalise your own manual page, spend time reading real ones. Three useful models:

- `/usr/src/share/man/man4/null.4` is a minimal page. Good for the basic shape.
- `/usr/src/share/man/man4/led.4` is a slightly more complex page that shows `SYNOPSIS` with function prototypes.
- `/usr/src/share/man/man4/re.4` is a full-featured network driver page. Good for seeing `HARDWARE`, `LOADER TUNABLES`, `SYSCTL VARIABLES`, `DIAGNOSTICS`, and `BUGS` in action.

Read each of them. Open them in `less`, read the rendered version, then open the source in an editor. Compare the rendered output to the source. You will see how the macros produce the formatted text, and you will absorb the conventions by osmosis.

### 37.7.18 Wrapping Up Section 4

The manual page is not an afterthought. It is a first-class artifact that ships with your driver and is the primary user-facing documentation. A good manual page has a specific structure (`NAME`, `SYNOPSIS`, `DESCRIPTION`, `HARDWARE`, and so on), is written in `mdoc`, follows the one-sentence-per-line rule, and passes `mandoc -Tlint` clean. The page deserves the same care as the code. A driver with a poor manual page will be held at review until the page is brought up to standard; a driver with a good one will clear that part of the review easily.

With the licence and the manual page in hand, you have all the paperwork of a driver submission covered. The next section turns to the technical side of testing, because a driver that compiles cleanly and passes style checks still needs to build on every supported architecture and behave correctly in a variety of situations. Section 5 walks through those tests.

## 37.8 Section 5: Testing Your Driver Before Submission

### 37.8.1 The Tests That Matter

Testing a driver before submission is not a single action. It is a sequence of verifications, each of which checks a different property. A driver that passes all of them is a driver the reviewers can focus on in terms of design and intent, rather than in terms of preventable mechanical problems. A driver that skips some of them is a driver that will have avoidable problems surfaced during review, each of which adds a round to the review cycle.

The tests fall into several categories:

1. Code-style tests, which verify that the source conforms to `style(9)`.
2. Manual-page tests, which verify that the `mdoc` source is syntactically valid and renders cleanly.
3. Local build tests, which verify that the driver builds as a loadable kernel module against the current FreeBSD source tree.
4. Runtime tests, which verify that the driver loads, attaches to its device, handles a basic workload, and detaches cleanly.
5. Cross-architecture build tests, which verify that the driver compiles on every architecture the project supports.
6. Lint and static-analysis tests, which catch bugs that the compiler does not flag but that are visible to more aggressive tooling.

Each category has its own tools and its own workflow. This section walks through them in order.

### 37.8.2 Code-Style Tests

We have already seen `tools/build/checkstyle9.pl` in Section 2. Here we will elaborate on its use.

The script lives at `/usr/src/tools/build/checkstyle9.pl`. It is a Perl program, so you invoke it as a script with Perl:

```
perl /usr/src/tools/build/checkstyle9.pl /usr/src/sys/dev/mydev/mydev.c
```

Or, if the script is executable and Perl is in its shebang line, simply:

```
/usr/src/tools/build/checkstyle9.pl /usr/src/sys/dev/mydev/mydev.c
```

The output is a list of warnings with line numbers. Typical warnings include:

- “space(s) before tab”
- “missing blank line after variable declarations”
- “unused variable”
- “return statement without parentheses”
- “function name is not followed by a newline”

Each warning maps to a specific rule in `style(9)`. Fix each one. Re-run. Repeat until the output is clean.

If you find yourself disagreeing with a warning, check `style(9)` first. It is possible for the script to produce false positives, but those are rare. Most of the time, a disagreement with the style checker is a misunderstanding of `style(9)`. Read the manual page before arguing.

Run `checkstyle9.pl` on every `.c` and `.h` file in your driver. The Makefile does not need to pass it since it is not C code.

### 37.8.3 Manual-Page Tests

For the manual page, the canonical test is `mandoc -Tlint`:

```
mandoc -Tlint /usr/src/share/man/man4/mydev.4
```

Fix every warning. Re-run. Repeat until the output is clean.

In addition, run `igor` if you have it installed:

```
igor /usr/src/share/man/man4/mydev.4
```

And render the page to read it as a user would:

```
mandoc /usr/src/share/man/man4/mydev.4 | less -R
```

You can also install your page into the system for a realistic test:

```
cp /usr/src/share/man/man4/mydev.4 /usr/share/man/man4/
makewhatis /usr/share/man
man 4 mydev
```

This last check is useful because it verifies that `man` can find the page, that `apropos` can find it via `whatis`, and that the page renders correctly in the standard pager.

### 37.8.4 Local Build Tests

Before doing anything else with the driver, verify that it builds. From the module directory:

```
cd /usr/src/sys/modules/mydev
make clean
make obj
make depend
make
```

The output should be a single `mydev.ko` file in the module's object directory. No warnings, no errors. If you see warnings, fix them. `style(9)` calls out that warnings should not be ignored; submissions that introduce warnings are held for review.

If you are running on the same machine where you will load the module, install it:

```
sudo make install
```

This copies `mydev.ko` to `/boot/modules/` so that `kldload` can find it.

### 37.8.5 Runtime Tests

Once the module is built and installed, test it:

```
sudo kldload mydev
dmesg | tail
```

The `dmesg` output should show your driver probing, attaching to any available hardware, and completing attach without error. If there is no matching hardware, the driver should simply not attach, which is fine for the loading test.

Exercise the driver as a user would. Open its device nodes, read and write to them, run the operations your driver supports, and watch for any diagnostic messages. Run it under load. Run it with multiple simultaneous openers. Run it with edge-case inputs. These are the kinds of tests that catch bugs the compiler cannot see.

Then unload:

```
sudo kldunload mydev
dmesg | tail
```

The unload should complete silently, with no “device busy” errors and no panics. If the unload produces a warning about busy resources, the driver's detach path has a leak; fix it before submission.

Repeat the load/unload cycle several times. A driver that loads and unloads once is not the same as a driver that loads and unloads repeatedly. Bugs in the detach path often show up only on the second or third unload, when the state left over from the first unload interferes with the second load.

### 37.8.6 Cross-Architecture Build Tests

FreeBSD supports several architectures. The active ones as of FreeBSD 14.3 include:

- amd64 (64-bit x86).
- arm64 (64-bit ARM, also called aarch64).
- i386 (32-bit x86).
- powerpc64 and powerpc64le (POWER).
- riscv64 (64-bit RISC-V).

- armv7 (32-bit ARM).

A driver that builds on amd64 may or may not build on all of the others. Common cross-architecture problems include:

- Integer-size assumptions. A long is 64 bits on amd64 and arm64 but 32 bits on i386 and armv7. If your code assumes `sizeof(long) == 8`, it will break on 32-bit architectures. Use `int64_t`, `uint64_t`, or similar fixed-size types when the size matters.
- Pointer-size assumptions. Similarly, pointers are 64 bits on amd64 and 32 bits on i386. Casting between pointers and integers requires `intptr_t/uintptr_t`.
- Endianness. Some architectures are little-endian, some are big-endian, some are configurable. If your driver reads or writes network-byte-order data, use the explicit byte-swap macros (`htonl`, `htons`, `bswap_32`, and friends), not hand-rolled conversions.
- Alignment. Some architectures enforce strict alignment on multi-byte loads. Use `memcpy` or the `bus_space(9)` API rather than direct casts when accessing hardware registers.
- Bus abstractions. The `bus_space(9)` API abstracts hardware access correctly across architectures; using `inline volatile *` casts does not.

The best way to catch cross-architecture problems is to build the driver for each architecture. Fortunately, FreeBSD has a build target that does exactly that:

```
cd /usr/src
make universe
```

`make universe` builds the world and the kernel for every supported architecture. The full build can take an hour or more depending on the machine, so it is not something you run on every change, but it is the canonical pre-submission test. The Makefile in `/usr/src/` describes it:

```
universe - Really build everything (buildworld and all kernels on all architectures).
```

If you do not want to build everything, you can build just a single architecture:

```
cd /usr/src
make TARGET=arm64 buildkernel KERNCONF=GENERIC
```

This is faster and is often enough to catch the typical cross-architecture issues.

For just your module, you can sometimes cross-build with:

```
cd /usr/src
make buildenv TARGET_ARCH=aarch64
cd sys/modules/mydev
make
```

But `make universe` and `make buildkernel TARGET=...` are the canonical tests, and any serious submission should pass them.

### 37.8.7 tinderbox: The Failure-Tracking Variant of universe

A variant of `make universe` is `make tinderbox`:

```
cd /usr/src
make tinderbox
```

Tinderbox is the same as `universe`, but at the end it reports the list of architectures that failed and exits with an error if any did. For a submission workflow, this is often more useful than plain `universe`, because the failure list is a clear action item.

### 37.8.8 Running Kernel Lint Tools

FreeBSD's kernel build optionally runs additional checks. The `LINT` kernel configuration is a kernel built with every driver and option turned on, which surfaces cross-cutting problems that single-feature kernels miss. Building the `LINT` kernel is not usually required for a driver submission, but it is a useful sanity check if you are touching anything widely used.

`clang` itself, as FreeBSD's default compiler, performs sophisticated static analysis during normal compilation. Build with `WARNS=6` to see the most aggressive warning set:

```
cd /usr/src/sys/modules/mydev
make WARNS=6
```

And fix any warnings that appear. Clang also has a `scan-build` tool that runs static analysis as a separate pass:

```
scan-build make
```

Install it from the ports tree (`devel/llvm`) if it is not available.

### 37.8.9 Testing in a Virtual Machine

Much of this chapter assumes you are testing on a real machine or a virtual machine. Virtual machines are particularly useful for driver testing because a panic costs nothing more than a reboot. Two common approaches:

- `bhyve`, FreeBSD's native hypervisor. A FreeBSD guest under `bhyve` can be a good testing environment, particularly for network drivers using `virtio`.
- `QEMU`. `QEMU` can emulate architectures other than the host, which makes it useful for testing cross-architecture builds at runtime without needing physical hardware in each architecture.

For cross-architecture runtime testing, `QEMU` with a FreeBSD image in the target architecture is a good workflow. Build the module for the target architecture, copy it into the `QEMU` VM, and run `kld load` there. Crashes inside the VM do not affect your host.



### 37.8.10 Testing Against HEAD

The FreeBSD source tree's main branch is sometimes called HEAD, in the release-engineering sense. Your driver should build and run against HEAD, because that is where your patch will first land. If you have been developing against an older branch, update to HEAD before final testing:

```
cd /usr/src
git pull
```

Then rebuild and retest. Kernel APIs change; a driver that built against a six-month-old tree may need small adjustments to build against current HEAD.

### 37.8.11 A Shell Script for the Whole Pipeline

For a serious submission, consider putting the test sequence into a shell script. The companion examples include one, but the skeleton is straightforward:

```
#!/bin/sh
pre-submission-test.sh
set -e

SRC=/usr/src/sys/dev/mydev
MOD=/usr/src/sys/modules/mydev
MAN=/usr/src/share/man/man4/mydev.4

echo "--- style check ---"
perl /usr/src/tools/build/checkstyle9.pl "$SRC"/*.c "$SRC"/*.h

echo "--- mandoc lint ---"
mandoc -Tlint "$MAN"

echo "--- local build ---"
(cd "$MOD" && make clean && make obj && make depend && make)

echo "--- load/unload cycle ---"
sudo kldload "$MOD"/mydev.ko
sudo kldunload mydev

echo "--- cross-architecture build (arm64) ---"
(cd /usr/src && make TARGET=arm64 buildkernel KERNCONF=GENERIC)

echo "all tests passed"
```

Run this script before every submission. If it exits clean, your driver has cleared all the mechanical tests. Reviewers can then focus on the design.

### 37.8.12 What Testing Does Not Catch

Testing tells you that your driver compiles and that it works in the situations you tested. It does not tell you that it works in all situations. A driver that passes every test may still have bugs that surface only under rare workloads, on rare hardware, or in rare interleavings of kernel state.

This is normal. Software is never fully tested. The role of pre-submission testing is not to prove the driver correct, but to catch the mistakes that are easy to catch. Design-level bugs, rare race conditions, and subtle protocol violations will still find their way into the tree and will be caught, sometimes much later, by users who encounter them. That is what post-merge maintenance is for, and we will cover it in Section 8.

### 37.8.13 Wrapping Up Section 5

Testing is a multi-stage verification process. Style checks, manual-page lints, local builds, runtime load/unload cycles, cross-architecture builds, and static analysis each test a different property. A driver that passes all of them is a driver in shape for review. The tools are standard: `checkstyle9.pl`, `mandoc -Tlint`, `make`, `make universe`, `make tinderbox`, and `clang`'s built-in analysis. The discipline is to run them in order, fix every warning they produce, and not submit until they all pass clean.

With the driver tested, we can now turn to the mechanics of actually submitting it for review. Section 6 walks through patch generation and the submission workflow.

## 37.9 Section 6: Submitting a Patch for Review

### 37.9.1 What a Patch Is, in the FreeBSD Sense

A patch, in the FreeBSD sense, is a reviewable unit of change. It can be a single commit or a series of commits. It represents one logical change to the tree. For a new driver submission, the patch is typically one or two commits that introduce the new driver files, the new module Makefile, and the new manual page.

The mechanical form of a patch is a text representation of the changes, usually in unified-diff format. There are several ways to generate such a representation:

- `git diff` produces a diff between two commits or between a commit and the working tree.
- `git format-patch` produces a patch file per commit, with the full commit metadata included, in a form suitable for email or for attachment to a review.
- `arc diff`, from the Phabricator command-line tools, posts the current working state as a Phabricator revision.
- `gh pr create`, from the GitHub command-line tools, opens a GitHub pull request.

The right tool depends on where you are sending the patch. For Phabricator, `arc diff` is standard. For GitHub, `gh pr create` or the GitHub web UI is standard. For mailing lists, `git format-patch` with `git send-email` is standard.

All of them rely on the same underlying git commit. Before you worry about the submission tool, make sure the commit itself is clean.

### 37.9.2 Preparing the Commit

Start from a clean, up-to-date clone of the FreeBSD source tree:

```
git clone https://git.FreeBSD.org/src.git /usr/src
```

Or, if you already have a clone, update it:

```
cd /usr/src
git fetch origin
git checkout main
git pull
```

Create a topic branch for your work:

```
git checkout -b mydev-driver
```

Make your changes: add the driver files, the module Makefile, and the manual page. Run all the tests from Section 5. Fix any issues.

Commit your changes. The commit should be a single logical unit of change. If you are introducing a brand-new driver, one commit is usually appropriate: “mydev: Add driver for FooCorp FC100 sensor boards.” The commit message should follow the conventions from Section 2.

```
git add sys/dev/mydev/ sys/modules/mydev/ share/man/man4/mydev.4
git commit -s
```

The `-s` adds a Signed-off-by: line. The editor opens for the commit message; fill in the subject line and body following the conventions from Section 2.

Review the commit:

```
git show HEAD
```

Read every line. Check that no unrelated files are included. Check that no trailing whitespace is present. Check that the commit message reads well. If anything is off, amend:

```
git commit --amend
```

Repeat until the commit is exactly as you want it.

### 37.9.3 Generating a Patch for Review

Once the commit is clean, generate the patch. For a Phabricator review, `arc diff`:

```
cd /usr/src
arc diff main
```

arc will detect that you are on a topic branch, generate the diff, and open a Phabricator review in your browser. Fill in the summary, tag reviewers if you know any, and submit.

For a GitHub pull request, push your branch and use gh:

```
git push origin mydev-driver
gh pr create --base main --head mydev-driver
```

Or open the pull request through the GitHub web UI. The pull request form asks for a title (use the commit subject line) and a body (use the commit body). The title and body form the description of the pull request; they should match what the committed change will eventually carry.

For a mailing-list submission or an email to a maintainer:

```
git format-patch -1 HEAD
```

This produces a file like 0001-mydev-Add-driver-for-FooCorp-FC100-sensor-boards.patch containing the commit. You can attach it to an email or send it inline with `git send-email`. Mailing-list submissions are less common today than Phabricator or GitHub but are still accepted.

### 37.9.4 Which Route to Take

The `CONTRIBUTING.md` file gives specific guidance on which route to use when. The short version:

- GitHub pull requests are preferred when the change is small (fewer than about 10 files and 200 lines), self-contained, passes CI cleanly, and needs little developer time to land.
- Phabricator is preferred for larger changes, for work that needs extended review, and for subsystems where the maintainers prefer Phabricator.
- Bugzilla is appropriate when the patch fixes a specific reported bug.
- Direct email to a committer is appropriate when you know the maintainer of the subsystem and the change is small enough to handle informally.

A new driver is usually somewhere between the GitHub size limit and the Phabricator fit. If your driver has fewer than 10 files and fewer than 200 lines, a GitHub PR will work. If it is larger, try Phabricator first.

Whichever you choose, make sure the driver has been through all the pre-submission tests. The CI that runs on GitHub and the review process on Phabricator will both catch problems, but it saves everyone time if you have caught them first.

### 37.9.5 Writing a Good Review Description

The patch itself is only half of what you submit. The other half is the description: the text that accompanies the patch and explains what it does, why it is needed, and how it was tested.

On Phabricator, the description is the Summary field. On GitHub, it is the PR body. On a mailing list, it is the body of the email.

A good description has three parts:

1. A one-paragraph summary of what the patch does.
2. A discussion of the design and any interesting decisions.
3. A list of what was tested.

For a driver submission, a typical description might read:

This patch adds a driver for FooCorp FC100 environmental sensor boards. The boards are PCI-attached and expose a simple command-and-status interface over a single BAR. The driver implements probe/attach/detach following Newbus conventions, exposes a character device for userland interaction, and documents tunable sampling intervals via sysctl.

The FC100 is documented in FooCorp's Programmer's Reference Manual version 1.4. The driver supports revisions 1.0 and 1.1 of the board, and operates the FC200 in its FC100-compatibility mode.

Tested on amd64 and arm64 with a physical FC100 rev 1.1 board. Passes `make universe`, `mandoc -Tlint`, and `checkstyle9.pl`. Load/unload cycle verified 50 times without leaks.

Reviewer suggestions welcome on the sysctl structure.

This description does several things right. It explains the driver in a way a reviewer who has never seen it can understand. It establishes what has been tested. It explicitly invites feedback on a specific design question. It reads as a collaborative request for review, not as a “here is my code, merge it” demand.

### 37.9.6 The Draft Email for the Mailing List

Even if you plan to submit via Phabricator or GitHub, a draft email to one of the FreeBSD mailing lists can be a useful introduction. The general-purpose list for development questions is `freebsd-hackers@FreeBSD.org`; there are also subsystem lists such as `freebsd-net@` for network drivers and `freebsd-scsi@` for storage. Pick the list that most closely matches the subsystem your driver lives in, and if in doubt, start with `freebsd-hackers@`.

A draft email might look like this:

To: `freebsd-hackers@FreeBSD.org`  
Subject: New driver: FooCorp FC100 sensor boards

Hello,

I am working on a driver for FooCorp FC100 PCI-attached environmental sensor boards. The boards are documented, I have two hardware samples to test against, and the driver is in a state that passes `mandoc -Tlint`, `checkstyle9.pl`, and `make universe` clean.

Before I open a review, I wanted to ask the list if anyone has:

- \* Experience with similar sensor boards that might inform the `sysctl` structure.
- \* Strong preferences about whether the driver should expose a character device or a `sysctl` tree as the primary interface.
- \* Comments on the draft manual page (attached).

The code is available at <https://github.com/<me>/<branch>> for anyone who wants to take an early look.

Thanks,

Your Name <you@example.com>

This is the kind of email that tends to generate helpful responses. It shows that the work is serious, it asks specific questions, and it offers a way to see the code. Many successful FreeBSD submissions start with an email like this.

For this book, you do not need to actually send such an email. The companion examples include a draft you can use as a template. The exercise in Section 4 of this chapter includes writing your own draft.

### 37.9.7 What Happens After You Submit

Once you submit, the review process begins. The exact flow depends on the submission route, but the general pattern is similar across routes.

For a Phabricator review:

- The revision appears in Phabricator's queue. It is automatically subscribed to any mailing lists relevant to the subsystem.
- Reviewers may pick up the review from the queue, or you may tag specific reviewers you think are relevant.
- Reviewers leave comments on specific lines, general comments, and requested changes.
- You address the comments by updating the commit and running `arc diff --update` to refresh the revision.
- The review cycle repeats until the reviewers are satisfied.
- A committer eventually lands the patch, credited to your name as the author.

For a GitHub pull request:

- The PR appears in the GitHub queue for `freebsd/freebsd-src`.
- CI jobs run automatically; they must pass.
- Reviewers comment on the PR, leave line comments, or request changes.
- You address the comments by committing fixups to your branch. Eventually, you squash the fixups into the original commit.
- When a reviewer is ready, they will either merge the PR themselves (if they are a committer) or shepherd it to Phabricator for a deeper review.

- The merged commit retains your authorship.

For a mailing-list submission:

- Mailing list readers respond with feedback.
- You iterate based on the feedback and send updated versions as replies to the original thread.
- When a committer is ready, they will commit the patch, credited to you.

In all cases, the iteration is part of the process. A patch that enters the tree in exactly the form it was first submitted is rare. Expect at least one round of feedback, often several. Each round is the reviewers helping you polish the submission.

### 37.9.8 Response Time and Patience

Reviewers are volunteers, even those who are paid by employers to work on FreeBSD. Their time is finite. Response time for a review can range from hours (for a small, well-prepared patch that matches the reviewer's current interests) to weeks (for a large, complex patch that needs careful reading).

If your patch has not received a response within a reasonable time, it is acceptable to send a polite ping. The usual conventions:

- Wait at least a week before pinging.
- Keep the ping short: "Just a friendly ping on this review, in case it slipped off anyone's radar." Nothing more.
- Do not ping more than once per week. If a patch is not getting attention after multiple pings, the issue is probably not that the reviewers have forgotten; it may be that the patch needs more work, or that the reviewers who know the subsystem are busy with other things.
- Consider asking on the relevant mailing list for review attention. A public ask is sometimes more effective than private pings.

Do not respond to review silence with anger or pressure. The project is run by volunteers. A review that takes longer than you hoped is not a personal insult.

### 37.9.9 Iteration and Patch Updates

Every round of review will have comments you need to address. Some will be small (rename a variable, add a comment, fix a typo in the manual page). Some will be larger (rework a function, change an interface, add a test).

Address each comment. If you disagree with a comment, reply explaining your reasoning; do not just ignore it. Reviewers are open to being convinced, but only if you make a case.

When you update the patch, keep the commit history clean. If you pushed a "fixup" commit earlier, squash it into the original commit before the final submission. The tree commits should each be logically complete; they should not contain messy incremental steps.

The workflow for a GitHub PR update typically looks like:

```
make the fixes
git add -p
git commit --amend
git push --force-with-lease
```

For a Phabricator update:

```
make the fixes
git add -p
git commit --amend
arc diff --update
```

Always use `--force-with-lease` rather than `--force` when force-pushing. `--force-with-lease` refuses to push if the remote has moved in a way you did not know about, which prevents accidentally overwriting a reviewer's changes.

### 37.9.10 Common Mistakes in Submission

A few common submission-time mistakes:

- Submitting a draft. Polish the patch first. Submitting a patch you know is not ready wastes the reviewers' time.
- Submitting against an outdated tree. Rebase against current HEAD before submission.
- Including unrelated changes. Each submission should be one logical change. Style cleanups, unrelated bug fixes, and random improvements should be separate submissions.
- Not responding to feedback. A patch that stalls in review because the author never replied is a patch that will die.
- Pushing back defensively. Reviewers are trying to help. Responding defensively to feedback is a quick way to sour the relationship.
- Submitting the same patch to multiple routes simultaneously. Pick one route. If you submit to Phabricator, do not also open a GitHub PR with the same content.

### 37.9.11 Wrapping Up Section 6

Submitting a patch for review is a mechanical process once the patch is in shape. The patch is a commit (or series of commits) with a proper message, against a current tree. The submission route depends on the size and nature of the change: small self-contained changes go to GitHub PRs, larger or deeper changes go to Phabricator, specific bug fixes can attach to Bugzilla entries. The accompanying description explains the patch and invites review. What follows is an iterative review cycle that ends with a committer landing the patch.

The next section looks at the human side of that iteration: how to work with a mentor or committer, how to handle feedback, and how to turn a first submission into the beginning of a longer-term relationship with the project.



## 37.10 Section 7: Working With a Mentor or Committer

### 37.10.1 Why the Human Side Matters

The submission process is ultimately a collaboration with people, not with a platform. The patch you submit is reviewed by engineers who have their own contexts, their own workloads, and their own experiences with what makes a driver easy or hard to review. The success or failure of your submission depends as much on how you engage with these people as on the technical quality of the code.

That framing bothers some first-time contributors, who would prefer that the technical work stand alone. The preference is understandable, but it does not match reality. FreeBSD is a community project, not a code-submission service. Reviewers offer their time because they care about the project and because they enjoy helping other contributors succeed. When that care is reciprocated, the experience is good for everyone. When it is not, the experience can be frustrating even for patches that are technically good.

This section walks through the human side of the contribution process. Some of it will feel obvious. Much of it is rarely discussed explicitly, which is why first-time contributors sometimes stumble even when their code is solid.

### 37.10.2 The Role of a Mentor

A mentor, in the FreeBSD context, is a committer who has agreed to guide a specific new contributor through their first submissions. Not every contribution involves a mentor; many first submissions land through ordinary review without a formal mentorship. But when a mentor is involved, the relationship has a specific shape.

A mentor typically does these things:

- Reviews your patches in detail, often before they go to wider review.
- Helps you understand the project's conventions and the specific subsystem you are working in.
- Sponsors commits on your behalf, meaning they commit the patch to the tree crediting you as the author.
- Answers questions about project process, style, and social norms.
- Vouches for you in nomination discussions if, later, commit rights become appropriate.

A mentor is not doing your work. They are accelerating your integration into the project. A good mentor is patient, willing to explain, and willing to push back when you are going in a wrong direction. A good mentee is diligent, willing to listen, and willing to do the work of iterating.

Finding a mentor is often organic. It happens because you engaged productively with a specific committer over several rounds of review, and they offered to take on a more structured mentorship role. It is rarely because you asked cold. If you are interested in mentorship, the right move is to start contributing visibly and productively, and to let the relationship develop.

The FreeBSD Project also has more formal mentorship programmes at various times, including for specific demographic groups or specific subsystems. These programmes are the right place to look for a mentor if you want a structured starting point.

### 37.10.3 Sponsorship: The Commit Pathway

A sponsor is a committer who commits a patch on behalf of a contributor. Every contribution from a non-committer goes through a sponsor at commit time. The sponsor is not necessarily the same person as the primary reviewer, and not necessarily a mentor, though they can be both.

Finding a sponsor for a patch is usually straightforward. If the patch has been through review and at least one committer has approved it, that committer is usually willing to sponsor the commit. You do not need to ask formally; the commit will happen when the reviewer is ready.

If the patch has been reviewed but no one has moved to commit it, a polite question is appropriate: “Is anyone in a position to sponsor the commit of this patch?” Asking in the review thread or on the relevant mailing list will usually find someone.

Do not confuse sponsorship with support in the abstract. A sponsor is specifically the person who runs the `git push` that lands your patch. They take on a small amount of responsibility: their name appears in the commit metadata, and they are implicitly certifying that the patch was ready to land.

### 37.10.4 Receiving Feedback Gracefully

Feedback on your patch can be hard to read, especially the first time. Reviewers write in the mode of reviewing code, which means they name specific things that need to change. That mode reads as negative even when the underlying assessment of the patch is overwhelmingly positive. A review that says “this is a great start, but here are twenty things to fix” is normal for a first submission.

The right response to feedback is to address it. For each comment:

- Read it carefully. Make sure you understand what the reviewer is asking.
- If the comment is clear and actionable, make the change. Do not argue just because the suggestion was not your first choice.
- If the comment is unclear, ask for clarification. “Can you say more about what you mean by X?” is a perfectly fine response.
- If you disagree with the comment, reply explaining your reasoning. Be specific: “I thought about using X but went with Y because Z.” Reviewers are open to being persuaded.
- If the comment is outside the scope of the patch, say so and propose handling it separately. “Good catch, but this is really a separate change; I will send it as a follow-up.”

Never respond with hostility. Even if you believe the reviewer is wrong, respond calmly and with reasoning. A review thread that descends into anger is one that the reviewer will disengage from, and your patch will stall.

Several specific responses to avoid:

- “The code already works.” The code working is not the question. The question is whether it matches the conventions and design expectations of the tree.
- “This is just style; the code is fine.” Style is part of engineering quality. Reviewers are not wasting your time when they ask about style.
- “Other drivers in the tree do it this way.” They may well, and the tree has plenty of older drivers that do not match modern conventions. The goal for new contributions is to match modern conventions, not to reproduce historical drift.

- “I will do that later.” If you say you will do it later, the reviewer has no way to verify that you will. Do it now, or discuss why a later fix is appropriate.

The review process is cooperative. The reviewer is not your adversary. Every comment, even one you disagree with, is the reviewer investing time in your patch. Respond to that investment with your own.

### 37.10.5 Iteration and Patience

Patch review is iterative by design. A typical first driver submission goes through three to five rounds of review before it lands. Each round takes days to weeks, depending on reviewer availability and the size of the changes requested.

The total elapsed time from first submission to merge, for a new driver, is often several weeks. Sometimes it is months. That is normal. FreeBSD is a careful project; careful review takes time.

Several habits that help with iteration:

- Respond quickly. The faster you respond to feedback, the faster the review proceeds. Delays on your side are as damaging to the timeline as delays on the reviewer’s side.
- Batch small fixes. If the reviewer leaves ten comments, make all ten fixes in a single update rather than sending ten individual updates. Reviewers prefer to see the work integrated.
- Keep the commit clean. As you iterate, amend the original commit rather than stacking fixup commits. The commit that eventually lands should be a single clean commit, not a messy history.
- Test before re-submitting. Each iteration should pass the same pre-submission tests as the first submission. Do not break tests between rounds.
- Summarise each iteration. When you update the patch, a short reply on the review saying “updated to address all comments; specifically: did X, did Y, clarified Z” helps reviewers re-orient quickly.

Above all, be patient. The review process exists because code quality matters. Hurrying through it compromises the quality and produces a patch that lands quickly but creates problems later.

### 37.10.6 Handling Disagreements

Occasionally, reviewers will leave feedback you genuinely disagree with. The comment is not unclear; you have thought about it, and you believe the reviewer is wrong. What do you do?

First, consider that you might be wrong. Most of the time, when a reviewer raises a concern, there is something behind it that you may not be seeing. The reviewer has context about the tree, the subsystem, and the history that you may not have. Assume the concern is legitimate until you have evidence otherwise.

Second, if after thought you still disagree, reply with reasoning. Explain your perspective. Cite specific details from the code, the datasheet, or the tree. Ask the reviewer to engage with your reasoning.

Third, if the disagreement persists, escalate gently. Ask for a second opinion from another committer. Post to the relevant mailing list describing the question. Sometimes disagreements reveal that there are multiple defensible answers and that the project has not reached a clear position; that is useful information to surface.

Fourth, if the disagreement still persists and nothing is resolving it, you have a choice. You can make the change the reviewer requested, even though you disagree, and land the patch. Or you can withdraw the patch.

Both are legitimate. The project's culture is not one of forcing contributors to do things they disagree with, but neither is it one of rubber-stamping patches the committer community has concerns about. If the disagreement is fundamental, withdrawing is sometimes the right outcome.

Disagreements of this depth are rare. Most feedback is practical and either clearly right or clearly accommodatable. The serious disagreements, when they happen, are usually about design choices where multiple answers are defensible.

### 37.10.7 Building a Long-Term Relationship

A first submission is not the end of the work. If it goes well, it can be the beginning of a long-term relationship with the project. Many committers started as first-time contributors whose early patches went smoothly, whose later patches built on that trust, and whose involvement eventually grew to the point where commit rights made sense.

Building that kind of relationship is not about performing. It is about continuing to contribute consistently and productively. Several habits that help:

- Respond to bug reports about your driver. If a user reports a bug, triage it, confirm or deny, and follow up. A driver whose author is responsive is a driver the project values.
- Review other people's patches. Once you are familiar with a subsystem, you can review new patches in that subsystem. Reviewing is how you become known as an expert, and how you internalise the subsystem's conventions.
- Participate in discussions. The mailing lists and the IRC channels have ongoing technical discussions. Participating, thoughtfully, is part of being in the community.
- Keep your driver up to date. If kernel APIs change, update your driver. If new hardware variants appear, add support. The driver is not finished at merge; it is a living artifact you are caretaking.

None of this is required. The project is grateful for any contribution, including one-shot patches from contributors who never return. But if you are interested in deeper involvement, these are the ways in.

### 37.10.8 Identifying Existing Maintainers

Many subsystems in FreeBSD have identifiable maintainers or long-time contributors. Finding them is useful because they are often the best reviewers for related work.

Several ways to identify maintainers:

- `git log --format="%an %ae" <file>` shows who has committed changes to a specific file. The names that appear frequently are the active maintainers.
- `git blame <file>` shows who wrote each line. If a specific function is something you are extending, the person who wrote it is often the right person to ask.
- The MAINTAINERS file, where it exists, lists formal maintainers. FreeBSD does not have a single tree-wide MAINTAINERS file, but some subsystems have informal equivalents.
- The AUTHORS section of the manual page names the primary author.

For a driver that extends an existing family, the author of the existing driver is usually the first reviewer to approach. They have the context and the authority. For a completely new driver in a new area, find a reviewer by asking on the relevant mailing list.

### 37.10.9 Exercise: Identify a Similar Driver's Maintainer

Before moving on, pick a driver in the tree that is similar in scope to the one you are working on. Use `git log` to identify its maintainer. Note their name and email. Then read a few of their commits and look at the reviewers they typically engage with. This gives you a mental model of who the people in the subsystem are, and makes the human side of the submission feel more concrete.

You are not expected to contact them unless you have a specific question. The exercise is about building awareness.

### 37.10.10 Wrapping Up Section 7

The human side of the review process is as important as the technical side. A mentor or committer who is engaged with your patch is a resource; treating them with respect, responding to feedback constructively, and iterating patiently are the practical disciplines of that engagement. Disagreements happen and are usually productive; defensiveness is the main risk to avoid. A first submission, handled well, can be the beginning of a long-term relationship with the project.

The last piece of the submission workflow is what happens after the patch is in the tree. Section 8 covers the long arc of maintenance.

## 37.11 Section 8: Maintaining and Supporting Your Driver Post-Merge

### 37.11.1 The Merge Is Not the End

When your patch lands in the FreeBSD tree, a natural feeling is that the work is done. The driver is in. The review is over. The commit is in the history. You can move on.

The feeling is understandable, but the picture is incomplete. Merging the driver into the tree is the beginning of a different kind of work, not the end of the driver's life. As long as the driver is in the tree, it requires occasional maintenance. As long as it is used, it will occasionally surface bugs. As long as the kernel evolves, its APIs will drift and the driver will need to follow.

This section walks through the post-merge maintenance picture. The expectations are not heavy, but they are real. A driver whose author disappears after the merge is a driver the project has to maintain on its own, and eventually, if no one takes over, that can be a reason to mark the driver as deprecated.

### 37.11.2 Bugzilla Watching

The FreeBSD Bugzilla at <https://bugs.freebsd.org/> is the project's primary bug tracker. Bugs filed against your driver will appear there. You are not required to subscribe to Bugzilla as a contributor, but you should at least know how to check for open bugs against your driver.

A simple way to check:

J

[https://bugs.freebsd.org/bugzilla/buglist.cgi?component=kern&query\\_format=advanced&short\\_desc=](https://bugs.freebsd.org/bugzilla/buglist.cgi?component=kern&query_format=advanced&short_desc=)

Replace mydev with your driver's name. The query returns bugs whose summary mentions your driver.

If a bug is filed:

- Read the report carefully.
- If you can reproduce it, do so.
- If you can fix it, prepare a patch. The patch goes through the same review process as any other change.
- If you cannot reproduce it, ask the reporter for more information: FreeBSD version, hardware details, relevant log output.
- If it is a real bug but you do not have the time or capacity to fix it, say so in the bug report. A bug with an engaged author who cannot currently fix it is different from a bug with an absent author. At minimum, the engagement means that someone else who looks at the bug has context to work with.

Bugzilla also hosts enhancement requests (feature requests for new functionality). These are lower priority than bug reports, but they are useful signals about what users need. You do not need to implement every enhancement request, but acknowledging them and discussing priorities is part of maintenance.

### 37.11.3 Responding to Community Feedback

In addition to Bugzilla, community feedback can reach you through several other channels:

- Direct email from users.
- Discussions on the mailing lists.
- Questions on IRC channels.
- Comments on review threads for related work.

The expectation for a responsive maintainer is not that you respond to every one of these instantly. The expectation is that you are reachable at the email address on record for the driver (the one in the AUTHORS section of the manual page and in the commit history), and that when you respond to something, you do so productively.

A practical rhythm might look like this: once a week or once every two weeks, check your driver-related email and Bugzilla queries. Respond to anything that is waiting. Triage anything new. Keep the response times reasonable, on the scale of a week or two rather than months.

If your circumstances change and you cannot maintain the driver any more, say so publicly. The project can and will find new maintainers if the need is known. The worst outcome is disappearing silently, leaving bugs unacknowledged and users uncertain whether the driver is being maintained at all.

### 37.11.4 Kernel API Drift

The FreeBSD kernel evolves. APIs that were stable when your driver was written may change. When this happens, your driver needs to be updated, and you are the first person the project will look to for the update.

Several kinds of API drift that commonly affect drivers:

- Changes to the Newbus framework: new method signatures, new method categories, changes to `device_method_t` macros.
- Changes to the bus-specific attachment patterns: PCI, USB, iicbus, spibus, and others evolve over time.
- Changes to the `bus_space(9)` interface.
- Changes to the character-device interface (`cdevsw`, `make_dev`, etc.).
- Changes to the memory allocation API (`malloc(9)`, `contigmalloc`, `bus_dma`).
- Changes to the synchronisation primitives (`mtx`, `sx`, `rw`).
- Deprecation of old APIs in favour of new ones.

Usually, these changes are announced on the mailing lists before they are committed, and sometimes they come with a “tree-sweep” commit that updates all users of the old API to the new one. If your driver is in the tree, the tree-sweep will typically update it automatically. But not always; sometimes the sweep is conservative and leaves drivers that it cannot mechanically update for the maintainer to handle.

A good habit: check `freebsd-current@` at least occasionally for API change discussions that affect your driver. If you see one, check whether your driver still builds against current HEAD. If it does not, send a patch to update it.

### 37.11.5 The UPDATING File

The project maintains an `UPDATING` file at `/usr/src/UPDATING` that lists significant changes in the source tree, including API changes that drivers may need to respond to. Read it occasionally (especially before updating your tree) to see whether anything affects your driver.

A typical `UPDATING` entry might read:

20260315:

```
The bus_foo_bar() API has changed to require an explicit
flags argument. Drivers using bus_foo_bar() should pass
BUS_FOO_FLAG_DEFAULT to preserve historical behaviour.
Drivers using bus_foo_bar_old() should migrate to the new
API as bus_foo_bar_old() will be removed in FreeBSD 16.
```

If you see an entry like this mentioning a function your driver uses, update the driver accordingly.

### 37.11.6 Tree-Wide Refactors

Occasionally, the project does tree-wide refactors that touch every driver. Examples from FreeBSD’s history include:

- The conversion from `$FreeBSD$` CVS tags to git-only metadata.
- The introduction of SPDX-License-Identifier lines across the tree.
- Large-scale renames of APIs such as the `make_dev` family or the `contigmalloc` family.

When a tree-wide refactor happens, the refactor commit typically updates your driver along with everyone else’s. You may not need to do anything. But the refactor will show up in `git log` against your driver, and future developers looking at the history will see it. Understand what happened so that you can explain it if asked.

### 37.11.7 Participating in Future Releases

FreeBSD has a release cycle of roughly a year between major releases, with point releases on a more frequent schedule. Your driver participates in this cycle whether you do anything active or not.

A few things are worth understanding:

- Your driver is built for every release that comes out of a branch where it lives. If it lives in `main`, it will be in the next major release. If it is also cherry-picked to a `stable/` branch, it will be in the next point release of that branch.
- Before major releases, the release-engineering team may ask maintainers to confirm that their drivers are in good shape. If you get such a request for your driver, respond. It is a simple action that helps the project plan the release.
- After a release, your driver is in the wild on every installation that uses that release. Bug reports may be more frequent right after a release.

Participating in the release cycle is a low-effort form of maintenance. It mostly consists of being available for the release engineers to contact you if they need to.

### 37.11.8 Keeping Code Updated: A Rhythm

A reasonable rhythm for maintaining a driver in the tree:

- Monthly: check Bugzilla for open bugs against the driver. Respond to anything outstanding.
- Monthly: rebuild the driver against current `HEAD` and check for warnings or failures. If anything fails, investigate and fix.
- Quarterly: re-read the manual page. Update if the driver has changed since the last review.
- Before major releases: run through the full pre-submission test suite (style, mandoc, build, universe) on your driver as it currently stands. Fix anything that has drifted.
- Whenever you have a hardware sample and a spare afternoon: exercise the driver on the hardware and make sure it still works.

This rhythm is not mandatory. A driver can go months without maintenance if nothing is broken. But having a rhythm in mind keeps the driver healthy over time.

### 37.11.9 Exercise: Create a Monthly Maintenance Checklist

Before closing this section, open a text file and write down a monthly maintenance checklist for your driver. Include:

- The URL for the Bugzilla query that shows bugs against the driver.
- The commands to rebuild the driver against current `HEAD`.
- The commands to run the style and lint checks.
- The commands to check for API drift (e.g., `grep` for deprecated calls).
- A note about the email address where users might reach you.
- A reminder to update the manual page date if you make semantic changes.

Save this checklist with the driver's source, or in your personal notes. The act of writing it down commits you to the rhythm. Checklists that exist on paper get followed; checklists that live only in memory do not.



### 37.11.10 When You Cannot Maintain Anymore

Life changes. Jobs change. Priorities shift. At some point you may find that you cannot maintain your driver the way you once did. This is normal and the project has processes for handling it.

The right move is to say so publicly. Options:

- Post to `freebsd-hackers@` or the relevant subsystem list saying that you are stepping back from the driver and inviting someone else to take it over.
- File a Bugzilla entry tagged as a maintainership transition question.
- Email the committers who have reviewed your patches and tell them directly.

The project will then find a new maintainer, or mark the driver as orphaned, or decide on some other path. The important thing is that the status is known. Silent abandonment is worse than any of the alternatives.

If no one takes over the driver and it continues to be used, the project may eventually mark it deprecated. This is not a failure; it is a reasonable response to code that no one is actively caring for. Drivers can be deprecated, removed, and later re-added if someone steps up. The history of the tree is full of such cycles.

### 37.11.11 Wrapping Up Section 8

Post-merge maintenance is a lighter-weight activity than initial submission, but it is real. The expectations are: watch Bugzilla for bugs against your driver, respond to users who reach out, keep the driver building against current HEAD as the kernel evolves, participate in release cycles, and say so publicly if you cannot continue to maintain. A driver whose author is engaged over time is a driver the project values beyond the initial merge.

With Sections 1 through 8 complete, we have walked the full arc of a driver submission: from understanding the project to preparing the files, through licensing, manual pages, testing, submission, review iteration, and long-term maintenance. The remainder of the chapter offers hands-on labs and challenge exercises that let you practise the workflow against real code, followed by a consolidation of the mental model and a bridge to the closing chapter.

## 37.12 Hands-On Labs

The labs in this chapter are designed to be done against a real driver. The easiest approach is to take a driver you have already written during the book, such as the LED driver from earlier chapters or the mock device from Chapter 36, and walk it through the submission-preparation workflow.

If you do not have a driver at hand, the companion examples in `examples/part-07/ch37-upstream/` include a skeleton driver you can use.

All the labs can be done in a FreeBSD 14.3 development virtual machine. None of them will submit anything to the real FreeBSD project, so you can work freely without worrying about accidentally publishing half-finished work.

### 37.12.1 Lab 1: Prepare the File Layout

Goal: Take an existing driver and rearrange its files into the conventional FreeBSD layout.

Steps:

1. Identify the driver you will be working with. Call it `mydev`.
2. Create the directory structure:
  - `sys/dev/mydev/` for driver sources.
  - `sys/modules/mydev/` for the module Makefile.
  - `share/man/man4/` for the manual page.
3. Move or copy the `.c` and `.h` files into `sys/dev/mydev/`. Rename them if necessary so that the main source file is `mydev.c`, the internal header is `mydev.h`, and any hardware register definitions live in `mydevreg.h`.
4. Write a module Makefile at `sys/modules/mydev/Makefile` following the template from Section 2.
5. Build the module with `make`. Fix any build errors.
6. Verify that the module loads with `kldload` and unloads with `kldunload`.

Success criterion: the driver builds as a loadable module and the file layout matches the conventions of the tree.

Expected time: 30 to 60 minutes for a small driver.

Common problems:

- Include paths that assumed the old layout. Fix the includes to use `<dev/mydev/mydev.h>` rather than `"mydev.h"`.
- Forgotten entries in `SRCS`. If you have multiple `.c` files, list all of them.
- Missing `.PATH`. The Makefile needs `.PATH: ${SRCTOP}/sys/dev/mydev` so `make` can find the sources.

### 37.12.2 Lab 2: Audit the Code Style

Goal: Bring the driver source into compliance with `style(9)`.

Steps:

1. Run `/usr/src/tools/build/checkstyle9.pl` against every `.c` and `.h` file in the driver. Capture the output.
2. Read each warning carefully. Cross-reference against `style(9)` to understand what the rule is.
3. Fix each warning in the source. Rerun the style checker after each batch of fixes.
4. When the style checker is clean, read through the source by eye looking for anything the checker missed: inconsistent indentation inside multi-line arguments, comment styles, variable-declaration groupings.
5. Ensure every function that is not exported has the `static` keyword. Ensure every exported function has a declaration in a header.

Success criterion: the style checker produces no warnings against any file in the driver.

Expected time: one to three hours for a driver that has not previously been through a style audit.

Common surprises:

- Space-instead-of-tab warnings on lines you thought were fine. The checker is strict; trust it.
- Warnings about blank lines between variable declarations and code. `style(9)` requires a blank line.
- Warnings about return expressions without parentheses. Fix by adding parentheses.

### 37.12.3 Lab 3: Add the Copyright Header

Goal: Ensure every source file has a correct FreeBSD-style copyright header.

Steps:

1. Identify every file in the driver that needs a header: each `.c` file, each `.h` file, and the manual page.
2. For each file, check the existing header. If it is missing or malformed, replace it with a known-good template.
3. Use `/*` - as the opening of the header in `.c` and `.h` files. Use `\"` - as the opening in the manual page.
4. Include the SPDX-License-Identifier line with the appropriate licence, typically `BSD-2-Clause`.
5. Add your name and email to the Copyright line.
6. Include the standard licence text.
7. Verify that the file starts in column 1 with the `/*` - or `\"` - opening.

Success criterion: every file has a correctly-formatted header that matches the conventions of files already in the tree.

Expected time: 30 minutes.

Verification:

- Compare your header against the header in `/usr/src/sys/dev/null/null.c`. They should be structurally identical.
- If you are using an automated licence-harvester tool, it should recognise your headers.

### 37.12.4 Lab 4: Draft the Manual Page

Goal: Write a complete section-4 manual page for the driver.

Steps:

1. Create `share/man/man4/mydev.4`.
2. Start from the template in Section 4 of this chapter, or from the companion example.
3. Fill in each section for your driver:
  - NAME and NAME description.
  - SYNOPSIS showing how to compile in the kernel or load as a module.
  - DESCRIPTION in prose.
  - HARDWARE listing supported devices.
  - FILES listing device nodes.
  - SEE ALSO with relevant cross-references.
  - HISTORY noting the driver's first appearance.
  - AUTHORS with your name and email.
4. Follow the one-sentence-per-line rule throughout.
5. Run `mandoc -Tlint mydev.4` and fix every warning.
6. Render the page with `mandoc mydev.4 | less -R` and read it from a user's perspective. Fix anything awkward.
7. If you have `igor` installed, run it and address its warnings.

Success criterion: `mandoc -Tlint` is silent, and the rendered page reads clearly.

Expected time: one to two hours for a first manual page.

Reading assignment for the lab: before you start, read `/usr/src/share/man/man4/null.4`, `/usr/src/share/man/man4/led.4`, and `/usr/src/share/man/man4/re.4`. These three pages span the range of complexity that section-4 manual pages can have, and they will give you a strong intuition for what yours should look like.

### 37.12.5 Lab 5: Build and Load Automation

Goal: Write a shell script that automates the pre-submission build and load cycle.

Steps:

1. Create a script named `pre-submission-test.sh` in the companion examples directory.
2. The script should, in order:
  - Run the style checker on every source file.
  - Run `mandoc -Tlint` on the manual page.
  - Run `make clean && make obj && make depend && make` in the module directory.
  - Load the resulting module with `kldload`.
  - Unload the module with `kldunload`.
  - Report success or failure clearly.
3. Use `set -e` so the script exits on the first error.
4. Include helpful `echo` statements announcing each stage.
5. Test the script against your driver.

Success criterion: the script runs clean against a driver that is ready for submission, and produces clear error output for a driver that has problems.

Expected time: 30 minutes for a simple script; longer if you add polish.

### 37.12.6 Lab 6: Generate a Submission Patch

Goal: Practise the patch-generation workflow without actually submitting.

Steps:

1. In a throwaway git clone of the tree, create a topic branch for your driver:

```
git checkout -b mydev-driver
```

2. Add the driver files:

```
git add sys/dev/mydev/ sys/modules/mydev/ share/man/man4/mydev.4
```

3. Commit with a proper message following the conventions from Section 2:

```
git commit -s
```

4. Generate a patch:

```
git format-patch -1 HEAD
```

5. Read the generated .patch file. Verify that it looks clean: no unrelated changes, no trailing whitespace, a well-formed commit message.
6. Apply the patch to a fresh clone to verify that it applies cleanly:

```
git am < 0001-mydev-Add-driver.patch
```

Success criterion: you have a clean patch file that represents the driver submission, and it applies cleanly to a fresh tree.

Expected time: 30 minutes.

Common surprises:

- `git format-patch` produces a file per commit. If you have three commits on your branch, you will get three .patch files. For a driver submission that should be a single commit, amend or squash first.
- Trailing whitespace in the commit shows up as `^I` sequences in the patch. Remove it before committing.
- Line-ending issues. Make sure your editor is using LF, not CRLF.

### 37.12.7 Lab 7: Draft a Review Cover Letter

Goal: Practise writing the description that accompanies a submission.

Steps:

1. Open a text editor and write an email-style cover letter for your driver submission.
2. Include:
  - A subject line suitable for a mailing-list message.
  - A one-paragraph summary of what the driver does.
  - A description of the hardware supported.
  - A list of what was tested.
  - A statement of what feedback you are inviting.
3. Keep the tone professional and collaborative. You are asking for review, not demanding approval.
4. Save the letter as `cover-letter.txt` in your companion examples directory.
5. Share it with a friend or colleague for feedback before moving on.

Success criterion: the cover letter reads as a productive invitation to review.

Expected time: 15 to 30 minutes.

### 37.12.8 Lab 8: Dry-Run a Review Cycle

Goal: Rehearse the iteration side of the review cycle.

Steps:

1. Ask a colleague to read your patch and cover letter as if they were a reviewer.
2. Capture their feedback as a list of comments.
3. Treat each comment as a real review comment. Respond to each one: make the fix, explain your reasoning, or push back constructively.
4. Update the commit and regenerate the patch.
5. Repeat for at least two rounds of feedback.

Success criterion: you have experience iterating on a patch in response to feedback, and your commit at the end is still a single clean commit rather than a messy history.

Expected time: variable, depending on the colleague's availability.

Variant: if a colleague is not available, ask a reviewer to read the companion code and act as a mock reviewer. Or use an online code-review simulator if one is available in your environment.

## 37.13 Challenge Exercises

The challenge exercises are optional but highly recommended. Each of them takes an idea from the chapter and pushes it into territory that will exercise your judgement.

### 37.13.1 Challenge 1: Audit a Historical Driver

Pick an older driver in `/usr/src/sys/dev/` that has been in the tree for at least five years. Look at its current state and identify:

- Parts of the copyright header that do not match modern conventions.
- Style violations that `checkstyle9.pl` flags.
- Manual-page sections that do not match modern style.
- Deprecated APIs the driver still uses.

Write up your findings as a short report. Do not submit a patch to fix them (older drivers often have good reasons for their historical form), but understand why they look the way they do.

The goal is to build an eye for the difference between modern conventions and historical ones. After doing this exercise, you will recognise at a glance which parts of a driver were written recently and which are legacy.

Expected time: two hours.

### 37.13.2 Challenge 2: Cross-Architecture Debug

Take your driver and attempt to build it for a non-native architecture, such as `arm64` if you are on `amd64`:

```
cd /usr/src
make TARGET=arm64 buildkernel KERNCONF=GENERIC
```

Identify any warnings or errors that are specific to the target architecture. Fix them. Rebuild. Repeat.

If your driver builds cleanly on both amd64 and arm64, try i386. If you want an extra challenge, try powerpc64 or riscv64. Each architecture will surface different kinds of issues.

Write up a short note about what you found and how you fixed it. The cross-architecture discipline is one of the things that separates a casually-written driver from a production-grade one.

Expected time: three to six hours, depending on how many architectures you try.

### 37.13.3 Challenge 3: Manual Page Depth

Pick a driver in the tree whose manual page you find impressive. Copy the structure of that page and use it as a template to rewrite your own manual page at a similar level of depth.

Your rewritten manual page should:

- Have a SYNOPSIS that shows all the ways to load and configure the driver.
- Have a DESCRIPTION that gives a user a complete picture of what the driver does.
- Have a full HARDWARE section, including revision information where relevant.
- Have LOADER TUNABLES, SYSCTL VARIABLES, or DIAGNOSTICS sections if your driver has any of these features.
- Have a BUGS section that is honest about known issues.
- Pass mandoc -Tlint and igor clean.

The goal is to produce a manual page that reads like a first-class example of the genre, not a minimal compliance artefact.

Expected time: three to five hours.

### 37.13.4 Challenge 4: Run a Mock Review

Partner with another reader of this book, or a colleague familiar with FreeBSD. Exchange drivers with them. You review their driver. They review yours.

As a reviewer, do the following for the driver you are reviewing:

- Run all the pre-submission tests yourself and capture the results.
- Read the code carefully. Make specific comments on anything that seems unclear, incorrect, or non-idiomatic.
- Read the manual page. Make comments on anything that seems incomplete or unclear.
- Write a summary review note that includes your overall impression, your requested changes, and any questions you have.

As the contributor receiving review, do the following:

- Read the feedback carefully.
- Respond to each comment constructively.
- Update the patch.
- Send the updated patch back.

Do at least two rounds. At the end, write a short reflection on what you learned from both sides.

The goal is to experience both sides of the review process before you ever submit a patch to the real project. After this exercise, the first real review will feel much more familiar.

Expected time: variable, but at least a weekend.

### **37.13.5 Challenge 5: Trace the Life of a Real Commit**

Pick a recent driver-related commit in the FreeBSD tree, ideally one that was contributed by a non-committer and sponsored. Use `git log` to find it, or browse Phabricator archives.

Trace its history:

- When was the review first opened?
- What did the first version look like?
- What comments did reviewers leave?
- How did the author respond?
- How did the patch evolve?
- When was it finally committed?
- What does the final commit message say?

Write up a short narrative of what you found. This exercise builds intuition for what a real review looks like from the inside.

Expected time: two hours.

## **37.14 Troubleshooting and Common Mistakes**

Even with careful preparation, things can go wrong. This section collects the most common problems first-time contributors encounter and explains how to diagnose and fix them.

### **37.14.1 Patch Rejected Because of Style**

Symptom: the reviewers leave many small comments about indentation, variable names, comment formatting, or return statement parentheses.

Cause: the patch was submitted without running `tools/build/checkstyle9.pl` first, or the author ignored some warnings.

Fix: run `checkstyle9.pl` against every source file. Fix every warning. Rebuild and re-test. Resubmit.

Prevention: make `checkstyle9.pl` part of your pre-submission script. Run it before every submission.

### **37.14.2 Patch Rejected Because of Manual-Page Issues**

Symptom: the reviewer says the manual page has lint errors, or does not match the project's mdoc style.



Cause: the manual page was not validated with `mandoc -Tlint` before submission, or the one-sentence-per-line rule was not followed.

Fix: run `mandoc -Tlint` against the manual page. Fix every warning. Read the rendered output to verify that it reads well. Resubmit.

Prevention: treat the manual page with the same care as the code. Include it in your pre-submission script.

### 37.14.3 Patch Does Not Apply Cleanly

Symptom: the reviewer reports that the patch does not apply to current HEAD. Or the CI fails at the `git apply` stage.

Cause: the patch was generated against an older version of the tree, and HEAD has moved since.

Fix: pull the latest HEAD, rebase your branch on top, resolve any conflicts, retest, and regenerate the patch.

Prevention: rebase on current HEAD immediately before submission. Do not submit a patch that was generated a week ago.

### 37.14.4 Kernel Panic on Load

Symptom: `kldload` panics the kernel.

Cause: often, a NULL pointer dereference in the driver's probe or attach routine, or a missing initialisation step.

Fix: debug with the standard kernel-debugging tools (covered in Chapter 34). Common specific causes:

- `device_get_softc(dev)` returning NULL because the `driver_t.size` field is not set to `sizeof(struct mydev_softc)`.
- `bus_alloc_resource_any(dev, SYS_RES_MEMORY, ...)` returning NULL and the driver not checking for the NULL before using the result.
- A static variable incorrectly initialised, causing undefined behaviour.

Prevention: test on a development VM before submission. Repeatedly load and unload to catch initialisation bugs.

### 37.14.5 Kernel Panic on Unload

Symptom: `kldunload` panics, or the module refuses to unload.

Cause: the detach path is incomplete. Common specific causes:

- A callout that is still scheduled when the softc is freed. Use `callout_drain`, not `callout_stop`.
- A taskqueue task that is still pending. Use `taskqueue_drain` on every task.
- An interrupt handler that is still installed when the resource is freed. Tear down the handler with `bus_tearardown_intr` before calling `bus_release_resource`.
- A device node that is still open when `destroy_dev` is called. Use `destroy_dev_drain` if the node might be open.

Fix: audit the detach path. Make sure every resource is released, every callout is drained, every task is drained, every handler is torn down, and every device node is destroyed before the softc is freed.

Prevention: structure the detach code in reverse of the attach code. Every attach step has a corresponding detach step, and the order is strict.

### 37.14.6 Driver Builds But Does Not Probe

Symptom: the module loads, but when the hardware is present, the driver does not attach to it. `pciconf -l` shows the device with no driver.

Cause: usually, a mismatch in the probe routine between the driver's expected vendor/device ID and the actual one. Or the driver uses `ENXIO` incorrectly.

Fix: check the vendor and device IDs. Double-check with `pciconf -lv`. Verify that probe returns `BUS_PROBE_DEFAULT` or `BUS_PROBE_GENERIC` when the device matches, not an error code.

Prevention: test against real hardware before submission.

### 37.14.7 Manual Page Does Not Render

Symptom: `man 4 mydev` shows no output, or shows raw mdoc source.

Cause: usually, the file is in the wrong place, or it is not named correctly, or `makewhatis` has not been run.

Fix: verify the path (`/usr/share/man/man4/mydev.4`), verify the name (must end in `.4`), and run `makewhatis /usr/share/man` to rebuild the manual database.

Prevention: test the man page installation before submission.

### 37.14.8 Reviewer Is Unresponsive

Symptom: you submitted a patch, responded to the initial comments, and then the reviewer went silent.

Cause: reviewers are volunteers. Their time is finite. Sometimes a patch slips off the radar.

Fix: wait at least a week. Then send a polite ping on the review thread or the relevant mailing list. If still silent, consider asking for another reviewer to pick it up.

Prevention: submit patches that are small, well-prepared, and easy to review. Smaller patches get faster reviews.

### 37.14.9 Patch Is Approved But Not Committed

Symptom: a reviewer has explicitly said the patch looks good, but it has not been committed.

Cause: the reviewer may not be a committer, or they may be a committer but waiting for a second opinion, or they may be busy with other things.

Fix: politely ask whether anyone is in a position to commit the patch. "Is anyone able to sponsor the commit of this patch? I have responded to all feedback and the review is approved."

Prevention: none specifically; this is part of normal project flow.

### 37.14.10 Patch Committed But You Were Not Credited

Symptom: you look at the commit log and see your patch committed, but the author field is wrong.

Cause: a committer may have accidentally applied the patch without preserving authorship. This is rare but happens.

Fix: politely email the committer asking whether the authorship can be corrected. A `git commit --amend` with the correct author can fix it before push; after push, the commit is immutable, but the committer can add a note or amend the original commit message in rare cases.

Prevention: when generating a patch with `git format-patch`, ensure your `user.name` and `user.email` are set correctly.

### 37.14.11 Your Driver Was Accepted but Your Interface Choice Was Wrong

Symptom: your driver is in the tree, but you later realise that the userland interface you designed was a poor fit.

Cause: design choices made before full usage experience sometimes turn out wrong.

Fix: this is a real engineering problem that the project handles regularly. The options include: adding a new interface alongside the old one and deprecating the old one; documenting the old interface as legacy and introducing a successor; or, rarely, making a breaking change if the driver has few enough users that the breakage is acceptable.

Prevention: talk to the list about interface design before implementation, especially for interfaces that will be visible to userland for a long time.

## 37.15 Wrapping Up

Submitting a driver to the FreeBSD Project is a process with many steps, but it is not a mysterious one. The steps, taken in order, lead from a working driver on your machine to a maintained driver in the FreeBSD source tree. The process involves understanding how the project is organised, preparing the files according to the project's conventions, handling the licence correctly, writing a proper manual page, testing across the architectures the project supports, generating a clean patch, navigating the review process with patience, and committing to the long arc of maintenance after the driver merges.

A few themes have run through the chapter and deserve a final explicit summary.

The first theme is that a working driver is not the same as an upstream-ready driver. The code you wrote in the book was working code; making it upstream-ready is additional work, and most of that work is in small conventions rather than large changes. Attention to those conventions is the difference between a first submission that is welcomed and one that is held up repeatedly in review.

The second theme is that upstream review is collaborative, not adversarial. The reviewers on the other side of the patch are trying to help your driver land in a form that the tree can carry forward. Their comments are investments of their time, not attacks on your competence. Responding to those comments productively, patiently, and substantively is the craft of the review process. First-time contributors who internalise this framing have easier reviews than those who do not.

The third theme is that documentation, licence, and style are part of engineering quality, not bureaucracy. The manual page you write is the interface through which users will understand your driver for as long as it lives. The licence you attach determines whether the driver can be merged at all. The style you follow determines whether future maintainers will understand the code. None of these are administrative overhead; all of them are part of the work of being a software engineer in a large shared code base.

The fourth theme is that merge is a beginning, not an end. The driver in the tree requires ongoing care: bug triage, API drift fixes, release-time check-ins, and occasional enhancements. That care is lighter than the initial submission, but it is real, and it is part of what turns a one-shot submission into a sustained contribution to the project.

The fifth and most important theme is that this is all learnable. None of the skills in this chapter require talent beyond what you have already built up over the book. They require attention to detail, patience with iteration, and willingness to engage with a community. Those qualities are ones you can develop with practice. The committers in the FreeBSD Project all started where you are now, as contributors writing their first patches, and they built up their standing through the same steady accumulation of careful work that you can.

Take a moment to appreciate what has changed in your toolkit. Before this chapter, submitting a driver to FreeBSD was probably a vague aspiration. Now it is a concrete process with a finite number of steps, each of which you have seen in detail. The labs have given you practice. The challenges have given you depth. The mistakes section has given you a map of the common pitfalls. If you decide to submit a real driver in the coming weeks or months, you have everything you need to start.

Some of the specifics in this chapter will shift over time. The Phabricator / GitHub balance may tilt further toward GitHub, or back, or neither. The style-checking tools may evolve. The review conventions may get small refinements. Where we know a convention is in motion, we have flagged it. Where we cited a specific file, we named it so you can open it yourself and check the current state. The reader who trusts but verifies is the reader the project benefits from most.

You are now, in the practical sense, prepared to contribute. Whether you do so is up to you. Many readers of a book like this one never contribute; that is fine, and the skills you built here serve you in your own work regardless. Some readers will contribute once, land a patch, and move on; that is also fine, and the project thanks them. A smaller number will find that they enjoy the collaboration enough to keep contributing, and over time will become deeply involved. Any of these paths is legitimate. The choice is yours.

## 37.16 Bridge to Chapter 38: Final Thoughts and Next Steps

This chapter has been, in one sense, the culmination of the practical arc of the book. You started without kernel knowledge, worked through UNIX and C, learned the shape of a FreeBSD driver, built character and network drivers, integrated with the Newbus framework, and worked through a series of mastery topics covering the specialised scenarios that production drivers encounter. The chapter you have just finished walked you through the process by which a driver you have built can become part of the FreeBSD operating system itself, maintained by a community of engineers and shipped to users on every release.

Chapter 38 is the closing chapter of the book. It is not another technical chapter. Its role is different. It is a chance to take stock of your progress, reflect on what you have learned, consider where you stand now, and think about where you might go next.

Several themes from Chapter 37 will naturally carry into Chapter 38. The idea that merge is a beginning rather than an end, for example, applies not just to individual drivers but to the reader's relationship with FreeBSD as a whole. Writing one driver, or two, or ten, is a beginning; sustained engagement with the project is the longer arc. The collaborative mindset that this chapter argued for in the context of code review is the same mindset that makes someone a valued community member over time. The discipline of documentation, licensing, and style that this chapter argued for in the context of a single driver scales up to the discipline of being a careful engineer in any large code base.

Chapter 38 will also address topics that this book did not cover in full depth, such as filesystem integration, network- stack integration (Netgraph, for example), USB composite devices, PCI hotplug, SMP tuning, and NUMA-aware drivers. Each of these is a substantial topic in its own right, and the closing chapter will point you toward the resources you can use to study them on your own. The book has given you the foundation; the topics in Chapter 38 are the directions you can extend that foundation into.

There are also the other BSDs. Much of what you have learned transfers, with modifications, to OpenBSD and NetBSD. The drivers you write for FreeBSD may find useful analogues in those projects, and some of the mastery topics from Part 7 have direct equivalents in each of the other BSDs. If you are interested in the wider BSD world, Chapter 38 will suggest where to look.

And there is the question of community. The FreeBSD Project is not an abstraction; it is a community of engineers, documenters, release managers, and users who together produce and maintain the operating system. Chapter 38 will reflect on what it means to be part of that community, how to find your place in it, and how to contribute to it beyond just driver submissions. Translations, documentation, testing, bug triage, and mentoring are all forms of contribution, and the project values each of them.

One final reflection before we close this chapter. Submitting a driver is, at its heart, an act of trust. You are offering your code to a community of engineers who will carry it forward. They, in turn, are offering their attention, their review time, and their commit rights to a contributor who was a stranger until this patch arrived. The trust goes both ways. It is built up, over time, by many small acts of careful work and responsible engagement. The first submission is the beginning of that trust, not the end of it. By the time you are a committer, if that is a path you choose, the trust is something you have earned in hundreds of small interactions.

You have done most of the work of becoming someone the project could trust. The rest is practice, and time, and patience.

Chapter 38 will close out the book with reflection, suggestions for continued learning, and a final word on where you might go from here. Take a breath, close your laptop for a moment, and let the material of this chapter settle. When you are ready, turn the page.

## 38 Final Thoughts and Next Steps

### 38.1 Introduction

You have reached the final chapter of a long book. Before we begin, take a moment to notice that simple fact. You are reading the closing pages of a manuscript that started, many chapters ago, with someone who may have had no kernel experience at all. You opened the first chapter as a curious reader. You are closing the last chapter as a person who can write, debug, and reason about FreeBSD device drivers. That shift did not happen by accident. It happened because you kept showing up, chapter after chapter, and put real effort into material that most people never try to learn.

This chapter is not a technical chapter in the same way the others were. You will not find new APIs to memorise, new `DEVMETHOD` entries to study, or new bus attachments to trace. The preceding thirty-seven chapters have given you a dense working vocabulary of FreeBSD kernel practice. What you need now is not more of that. You need a chance to step back, take stock of the ground you have covered, understand what you can do with it, and choose where to aim the skill you have built. The goal of this chapter is to give you that space, in a structured and useful form.

Reading a technical book from cover to cover is harder than it looks. By the end, many readers feel a curious mix of satisfaction and uncertainty. Satisfaction, because the book is finished and the work was real. Uncertainty, because the end of a book is not a clear signal of arrival anywhere in particular. The last page closes, the laptop goes back on the shelf, and the reader wonders what happens next. If you are feeling that mix, you are in good company. It is one of the most honest feelings in technical learning, and it usually means you have learned more than you realise. This chapter exists in part to help you see that clearly.

What happens next, practically, is up to you. The book has given you a foundation. The world of FreeBSD is enormous, and there are many directions you could take from here. You could pick one of the book's drivers and turn it into something polished that you submit upstream, bringing it through the workflow you learned in Chapter 37. You could choose a new device you have always wanted to write a driver for and start from scratch with the patterns you now know. You could study one of the advanced areas that this book only touched briefly, like deep filesystem work or the inner mechanics of the network stack, and start working your way into it. You could shift toward community contribution, where your skills help other people rather than landing a specific driver. Any of these paths is legitimate, and each of them will make you a better engineer than the one who finished Chapter 37.

The distance between finishing the book and being able to work confidently on your own is, in some sense, the theme of this chapter. Those are not the same milestone. Finishing the book means you have seen and practised enough to recognise the shapes of the work. Being able to work on your own means that when you sit down in front of a real problem, with no one else to lean on, you can find your way through it. The gap between those two milestones is closed mostly by practice. By writing drivers, debugging them, breaking them, fixing them, reading other people's code, and repeating those cycles over weeks and months. No book can substitute for that,

but a book can help you choose the right practice to do next, and it can help you keep your bearings while you do it. That is most of what the chapter is about.

We will spend time celebrating what you have accomplished, because that is both honest and useful. An honest recognition of progress is the ground from which the next cycle of learning grows. Without it, it is easy to fall into the trap of measuring yourself against distant experts rather than against the person you were when you started the book. We will then step into a careful self-assessment of where you stand now, using the technical material of the book as the frame. You will see your own capabilities reflected back at you in the language of FreeBSD practice. That is a different kind of encouragement from the generic “you can do it” sort. It is the kind of encouragement that comes from being able to point to something specific you have learned and saying, yes, I can use that.

From there we will look outward. FreeBSD is a large system, and there are several important areas that this book did not cover in full depth. Some of those areas are simply too large for any single book to cover, and some are rich enough that each deserves its own dedicated study. We will name them, explain briefly what each is about and why it matters, and point you toward the real sources you can use to learn more if you choose to go in that direction. The intent here is not to teach you filesystems or network-stack internals in a closing chapter. The intent is to give you a map so that when you finish this book and start thinking about where to go next, you have a clearer sense of the landscape.

We will also spend meaningful time on the practical work of building yourself a development toolkit that you can reuse across projects. Every working driver developer accumulates a small collection of personal artifacts over time: a driver skeleton that captures the patterns they always use, a virtual lab that boots quickly and lets them try things without breaking anything, a set of scripts that automate the annoying parts of testing, and a habit of writing regression tests before they submit changes. You do not need to start from scratch with those. The chapter will walk you through building a reusable toolkit that you can carry forward into any future driver work, and the companion examples for this chapter include starter templates you can adapt.

A significant part of the chapter turns to the FreeBSD community, because much of the longer-term growth that follows a book like this comes from community engagement. The community is where you see code that was written by hundreds of different hands, where you hear how people reason about problems that do not fit neatly into a single chapter, and where you find the feedback that helps you mature as an engineer. The chapter will show you concretely how to engage: which mailing lists matter, how to use Bugzilla, how code review happens, and how documentation contributions work. It will also explain, more carefully than the earlier book managed, what it means to mentor or be mentored in a community like FreeBSD, and why contributing to documentation, review, and testing is as valuable as contributing code.

And finally, because the FreeBSD kernel is a living system that changes constantly, we will talk about how to stay current. The kernel you learned to write drivers for is FreeBSD 14.3, and by the time you read this book at a later date, there will already be a newer release with a different set of internal APIs, new subsystems, and small changes to the patterns you learned. That is not a problem. It is how a healthy kernel evolves. The chapter will show you how to follow the changes: through commit logs, through mailing lists, through release-notes reading, through the conference circuit, and through the habit of periodically refreshing your understanding of a subsystem whose driver you maintain.

The companion material for this chapter is a little different from the companion material for earlier chapters. You will find it under `examples/part-07/ch38-final-thoughts/`, and it contains practical artifacts rather than buildable drivers. There is a reusable driver project template that you can copy into any new project as a starting point. There is a personal learning roadmap template you can use to plan the next three to six months

of your study. There is a contribution checklist that you can apply the first time you send a patch upstream, and every time after that. There is a regression-test script skeleton that you can adapt to any driver. There is a “stay current” checklist you can use to track FreeBSD development on a monthly rhythm. And there is a self-assessment worksheet that you can fill out at the end of this chapter and keep, so that in six months, when you fill it out again, you can see how far you have come.

One more framing note before we begin. Some readers will approach this chapter as a kind of final exam: a last chance to test themselves against the material of the book. That is not the spirit of the chapter. There is no grade here and no final test. The reflections and exercises in this chapter are not designed to catch you out on anything you did not learn. They are designed to help you see what you did learn, to see what you might want to learn next, and to help you plan the practice that turns a reader into an independent practitioner. Approach the chapter with that attitude, and it will be useful. Approach it as an exam and you will miss most of its purpose.

By the end of this chapter you will have a clear, written view of what you accomplished during the book, a realistic picture of your current skill set, a named and sorted list of advanced topics you might want to pursue, a reusable personal development toolkit, a clear sense of how to engage with the FreeBSD community, and a concrete plan for how to stay connected to FreeBSD’s continued evolution. You will also, with any luck, feel the quiet confidence that comes from having seen a long piece of work through. That confidence is not the end of the work; it is the fuel for whatever comes next.

Let us now begin that final reflection together.

## 38.2 Reader Guidance: How to Use This Chapter

This chapter is different in texture from the ones that came before it. The earlier chapters were technical: they built up a body of concrete knowledge, layered on each other, and culminated in Chapter 37’s walkthrough of upstream submission. This chapter is reflective. Its job is to help you consolidate what you learned, plan what you will do with it, and end the book in a way that prepares you for the next stage of your work.

Because the chapter is reflective rather than technical, the reading rhythm is different. In the earlier chapters, a careful reader might pause to type code, run a module, or inspect a struct in the kernel source. In this chapter, the pauses will be different. You will pause to think about what you learned. You will pause to write in a notebook or in a Markdown file. You will pause to look up a mailing-list archive, or to browse a part of the source tree you have not visited, or to check a conference schedule. The pauses are the point. If you read straight through without stopping, you will lose most of the value of the chapter.

Plan to spend a longer block of uninterrupted time on this chapter than you spent on most of the earlier ones. Not because the reading itself is harder, but because the thinking is. A single sitting of two or three hours, with a notebook at hand and without interruptions, is the minimum for a useful first pass. Some readers will find it more natural to spread the chapter across several sessions, treating each section as its own reflection exercise. Both approaches work; what matters is that the reflection is real, not rushed.

The chapter contains labs, just as the earlier chapters did, but they are reflection labs rather than coding labs. They will ask you to look back at something you wrote during the book and examine it with new eyes. They will ask you to write a one-page summary of a topic. They will ask you to build a personal learning plan. They will ask you to subscribe to a mailing list and read a thread. These exercises are not filler. They are the practice that turns the book’s content into your own working knowledge.



There are also challenge exercises, again adapted to the chapter's reflective character. These are longer and more open-ended than the chapter's labs. A reader who does all of them will invest several weekends of work. A reader who does just one or two will still get substantial value. The challenges are not graded, and there is no single correct way to complete them. They are invitations to extend the book's material into your own life and your own projects.

A reasonable reading schedule looks like this. In the first sitting, read the introduction and the first two sections. These are the consolidation sections: what you accomplished and where you stand now. Give yourself time to let the recognition settle. In the second sitting, read Sections 3 and 4. These look outward toward advanced topics and then back toward the practical toolkit you can carry into any future project. In the third sitting, read Sections 5 and 6. These are the community and currency sections, the ones that matter most for your long-term engagement with FreeBSD. Save the labs, the challenges, and the planning exercises for a fourth block, or spread them across the week that follows. By the time you finish, you will have not only read the chapter but also produced a set of personal artifacts that will be useful for months or years ahead.

If you are reading this book as part of a study group, this chapter is especially valuable to discuss together. The technical chapters can be read in parallel with relatively little coordination. This chapter benefits from conversation. Two readers comparing self-assessments will usually see something each one missed. A group that sets shared learning goals for the three months after finishing the book is more likely to actually pursue those goals than individuals working alone.

If you are reading alone, keep a journal while you work through this chapter. The journal can be paper or digital, whichever suits you. Write down your reflections as you go, name the advanced topics you want to pursue, list the practical habits you want to build, and record the planning artifacts you produce. The journal will be something you can return to in the months after the book, as you begin the practice that turns reading into independent skill.

One practical suggestion about the companion files. The chapter's examples are not drivers that you compile and load. They are templates and worksheets. Copy them out of the book repository into a location you control, such as a personal git repository or a folder in your home directory. Edit them to reflect your own context. Date them. Save older versions when you revise them, because a history of your own self-assessments is one of the most motivating kinds of learning record. The templates are provided so you do not have to start from a blank page, not so you will use them verbatim.

Finally, give yourself permission to take this chapter seriously even if it feels different from the rest of the book. A chapter of reflection can feel, to a reader used to technical material, like a soft or optional part of the curriculum. It is not. The reflection is where the technical material is consolidated into working capability, and consolidation is the stage of learning that is most often skipped and most often missed. Skipping it leaves a reader with many isolated facts and no particular sense of what to do with them. Doing it well turns those facts into a platform for the next phase of work.

With that framing in place, let us turn to the section where this book is most explicit: a careful, honest celebration of what you have accomplished.

### 38.3 How to Get the Most Out of This Chapter

A few habits will help you get the most out of a chapter whose material is reflective rather than procedural.

The first habit is to bring something to compare against. Find a driver you wrote during the book, or a set of labs you worked through, or a notebook of questions you noted while reading. Bring something concrete that lets you measure your progress. The reflections in this chapter become real when they are anchored to specific artifacts you produced along the way, and they become abstract when they are not. Open the earlier material beside the chapter and refer back to it as you read.

The second habit is to write as you read. The labs in this chapter will ask you to write, but you will get more out of the prose sections if you also write informally as you go. Keep a notebook or a text file open beside the chapter. Jot down the skills you recognise in yourself. Jot down the topics you still feel uncertain about. Jot down the questions that surface as you read. Writing is where reflection becomes thinking, and thinking is where thinking becomes planning. A chapter like this one done entirely in your head leaves most of its value on the page.

The third habit is to slow down on the forward-looking sections. This chapter spends time on advanced topics, the FreeBSD community, and how to stay current with the kernel. Those sections name specific resources: mailing lists, conferences, parts of the source tree, tools you can use. The temptation is to read past the names and return to the more comfortable prose. A better habit is to actually open a browser tab, visit the resource, and note down whether it is one you want to pursue. A single hour of doing this during the reading is worth ten hours of good intentions afterward.

The fourth habit is to be specific. Vague reflection is of limited use. Writing “I want to learn more about the network stack” is less useful than writing “I want to spend two weeks reading `/usr/src/sys/net/if.c` and the `ifnet(9)` manual page, and then write a short driver for a virtual network interface using what I learn”. Specificity makes plans achievable and vagueness makes plans evaporate. When the chapter asks you to identify next steps, identify them at a level of detail you could actually start work on this weekend.

The fifth habit is to give yourself time. This chapter is the end of a long book, and a common pattern at the end of long books is a rush to finish. Resist that pattern. The reflections are worth more than the speed. If you read the chapter in one sitting and feel no different at the end, you probably moved too fast through the parts that asked you to think. A good rule of thumb is that if you did not stop to write at least twice during the chapter, you did not engage with it in the way it was designed to be engaged with.

The sixth habit is to avoid the comparison trap. When readers of a technical book reach the end, they sometimes compare themselves to imagined experts and find the comparison discouraging. This is not useful. The right comparison is between the person you were when you opened Chapter 1 and the person you are now. Measured that way, almost every reader has made real progress. Measured against a senior committer who has spent a decade inside the FreeBSD tree, almost every reader falls short, and that is normal and meaningless. Pick the useful comparison, not the discouraging one.

The final habit is to plan to return to this chapter. Unlike the technical chapters, which you will probably not reread in full, this chapter benefits from rereading at natural checkpoints in your post-book work. When you finish your first driver after the book, revisit the self-assessment. When you send your first patch upstream, revisit the contribution checklist. When you have been following `freebsd-hackers` for six months, revisit the stay-current section. The chapter is designed to be reusable in this way.

With those habits in mind, let us turn to the first section, where we take an honest look at what you accomplished during the book.

## 38.4 1. Celebrating What You Have Accomplished

We begin with a section that asks you to pause and take stock, honestly, of what you have done. This is not a pep talk. It is a working part of the chapter. Many readers underestimate how much they learned during a long technical book, especially a book that requires as much sustained attention as this one. That underestimation is costly. A reader who does not recognise their own progress has a harder time deciding what to do next, because they cannot see clearly what they already have to build on.

### 38.4.1 1.1 A Recap of What We Have Covered

Think back to the first chapter you read. The book opened with an author's story about curiosity, a quick overview of why FreeBSD matters, and an invitation to begin. The second chapter helped you set up a lab environment. The third chapter introduced UNIX as a working system. The fourth and fifth chapters taught you C, first as a general language and then as a dialect shaped for kernel use. The sixth chapter walked you through the anatomy of a FreeBSD driver as a conceptual structure, before you had written one yourself.

Part 2 of the book, spanning Chapters 7 through 10, is where you started writing real code. You wrote your first kernel module. You learned how device files work and how to create them. You implemented read and write operations. You explored efficient input and output patterns. Each of those chapters had its own labs, and by the end of Part 2 you had produced several small drivers that actually loaded, actually exposed device files, and actually moved bytes between userland and kernel.

Part 3, Chapters 11 through 15, took you into concurrency. You learned about mutexes, shared/exclusive locks, condition variables, semaphores, callouts, and taskqueues. You worked through synchronisation problems that newcomers to the kernel often find intimidating, and you practised the discipline of locking conventions that the rest of the book later depended on. This part is where many readers report the steepest learning curve, and if you made it through, you did real work.

Part 4, Chapters 16 through 22, was about hardware and platform-level integration. You accessed registers directly. You wrote simulated hardware for testing. You built a PCI driver. You handled interrupts at a basic level and then at an advanced level. You transferred data with DMA. You explored power management. This is the part where the book changed from teaching you C with a kernel flavour into teaching you how kernels talk to real hardware. The gap between those two skills is large, and closing it is one of the biggest jumps in the book.

Part 5, Chapters 23 through 25, turned to the practical side of driver work: debugging, integration, and the habits that separate a driver that works on a demo from a driver that holds up in real use. You practised with `dtrace`, `kgdb`, `KASSERT`, and the other tools the kernel provides for seeing inside itself. You learned how to handle real-world problems that do not have textbook answers.

Part 6, Chapters 26 through 28, covered three transport-specific driver categories: USB and serial, storage and VFS, and network drivers. Each of these is a specialised area in its own right, and each introduced a new way of thinking about devices that was quite different from the character drivers of Part 2. By the end of Part 6, you had seen the major shapes that FreeBSD drivers take, not just in the abstract but in working examples.

Part 7, the part that contains this chapter, has been the mastery arc. You studied portability across architectures. You looked at virtualisation and containerisation. You worked through security best practices, the device tree and embedded systems, performance tuning and profiling, advanced debugging, asynchronous I/O, and reverse

engineering. Finally, in Chapter 37, you walked through the complete process of submitting a driver to the FreeBSD Project, from the social dynamics of contribution to the practical mechanics of `git format-patch`. And now, in Chapter 38, you are closing the book.

Listed like this, what you have covered looks substantial, and it is. The book has not been padding. Every chapter added something specific to your working knowledge. Even chapters that felt familiar on first reading probably taught you patterns of thinking that you internalised without fully noticing. The patterns are the most valuable part, because they are what let you apply the knowledge in situations the book did not explicitly cover.

Take a moment, before moving on, to think about which chapters felt the most challenging to you. Not in the abstract sense of “which was the longest chapter” but in the personal sense of “which chapter changed the most for me as I read it”. For some readers it is the jump from ordinary C to kernel C in Chapter 5. For others it is the concurrency work in Part 3. For some it is the interrupt and DMA material in Part 4, for others the network or storage work in Part 6, and for some the reverse engineering of Chapter 36. There is no single correct answer. The chapter that changed the most for you is the one where you grew the most, and noticing which one it was is useful information about your own pattern of learning.

### 38.4.2 1.2 Skills You Have Developed

Your progress is measured not only in chapters read but in skills acquired. Let us take a careful look at what you can now do, in language that describes capability rather than content.

You can write and reason about C in the kernel dialect. That is not the same skill as writing C for a command-line program. Kernel C has its own set of constraints, its own preferred idioms, and its own error-handling conventions. You know about the `M_WAITOK` / `M_NOWAIT` choice. You understand why kernel allocations must sometimes not sleep. You have seen the `goto out` style of cleanup and understand why it fits the kernel’s needs. You know when to use `memcpy` and when to use `copyin`. You know why kernel-to-userland and userland-to-kernel data transfer must go through specific functions rather than ordinary pointer dereferences. Each of those is a small habit, but together they form a dialect that you now speak.

You can debug and trace kernel code. You know how `dtrace` works. You know the difference between the function boundary tracing provider and the statically defined tracing provider. You know how `kgdb` attaches to a crash dump. You know what a `KASSERT` does, how to read a panic message, and how to use `witness` to detect lock-order violations. You know that `printf` in the kernel is a legitimate tool but not a substitute for structured diagnostics. You have built up the habit, over the course of Part 5 and later parts, of using the right diagnostic tool for the situation rather than defaulting to whatever is most familiar.

You can integrate a driver with the device tree and embedded systems. You understand the FDT format and how FreeBSD parses it. You know how to declare a driver that binds to a device-tree compatible string. You know what an embedded target looks like in practice: a slow boot, a small memory budget, a non-console serial port as the primary interface, and a reliance on hand-compiled kernels for each target board.

You know the basics of reverse engineering and embedded development. You know how to identify a device with `pciconf` or `usbconfig`, how to capture its initialisation sequence with `usbdump`, and how to build a register map by experiment. You know the legal and ethical framing of interoperability work. You know the discipline of separating observation from hypothesis from verified fact, and you know how to write a pseudo-datasheet that records what you learned. You are not an expert reverse engineer, and the book has been honest about that, but you know enough to start and to stay safe while you learn.

You can build userland interfaces for your drivers. You know how `devfs` publishes a device node. You know how to implement `read`, `write`, `ioctl`, `poll`, `kqueue`, and `mmap` operations. You know the standard `cdevsw` pattern. You know how to expose tunable parameters through `sysctl(9)`. You know how to structure a driver so that userspace programs can cooperate with it cleanly, and you have seen how poor userland interfaces are one of the most common reasons a driver is hard to maintain.

You understand concurrency and synchronization as it is practised in the kernel, not just in the abstract. You know the difference between an ordinary mutex and a spin mutex. You know when to reach for an `sx` lock and when `rmlock` is the better choice. You know what condition variables are for and how to use them without introducing lost-wakeup bugs. You know how to structure a taskqueue for deferred work. You know the `epoch(9)` interface for RCU-style readers. Each of those is a tool you know by its proper name and purpose, and you can apply it to a real problem without needing to look up the basics.

You can interact with hardware through DMA, interrupts, and register access. You have written code that sets up a DMA tag, allocates a map, loads a map, and syncs a map before and after transfers. You have written interrupt handlers that are careful about what they can and cannot do. You have used `bus_space(9)` to read and write registers on real hardware and on simulators. You have handled both filter and `ithread` interrupts. These are not abstract ideas for you any longer; they are practices you have executed.

You are ready to think seriously about upstream submission. You know the shape of a FreeBSD submission. You know `style(9)` and `style.mdoc(5)`. You know the structure of `/usr/src/sys/dev/` directories. You know how to write a manual page that lints cleanly. You know the Phabricator workflow and the GitHub workflow. You know what a reviewer will look for and how to respond to review feedback productively. That is a substantial body of knowledge, and most self-taught kernel developers never reach it.

Each of these skills, taken alone, would be worth serious study. Taken together, they describe a working FreeBSD device-driver developer. You have all of them now. You may not be equally confident in each one. That is normal. Part of the next section will help you notice where you are strong and where you have room to grow.

### 38.4.3 1.3 Reflection: What Has Actually Changed

Before moving on to self-assessment, take a moment to consider what has actually changed about you as a technical reader since you started this book.

When you first opened Chapter 1, a line of kernel code may have looked opaque. The macros were unfamiliar. The structures felt arbitrary. The control flow seemed impossibly indirect. Now, when you open a driver in `/usr/src/sys/dev/`, you can read the shape of it. You can find the probe and attach routines. You can identify the method table. You can see where locks are acquired and released. The text has become navigable. That change is invisible until you pause to notice it, but it is among the most important things the book has done for you.

When you first encountered a kernel panic, it may have been a blank wall of fear. You did not know what to do with it, what to read, or how to recover from it. Now a panic is a piece of information. You know how to read the stack trace. You know what a page fault means in the kernel. You know the difference between a recoverable panic from a clean assertion and an unrecoverable panic from memory corruption. A panic has gone from being a wall to being a diagnostic tool, and that shift is a large one.

When you first heard the phrase “write a device driver,” the phrase may have felt impossibly advanced. Drivers

were what other people wrote, perhaps hardware engineers in large companies, and the technical apparatus required seemed remote. Now, writing a driver is a concrete activity. You know what you need: a FreeBSD 14.3 system, a text editor, a working knowledge of C, the set of kernel APIs the book covered, a piece of hardware or a simulator, and some time. That shift in perception is the difference between treating drivers as a mystery and treating them as a craft.

When you first read a line referring to a lock order, it may have seemed like an academic subtlety. Now you know what lock order actually means, and why reversing it can crash a kernel, and how `witness` detects the violation, and how to structure a driver so that the violation never happens. This is not a theoretical understanding any more. It is a working practice.

When you first encountered the FreeBSD source tree, it may have felt like a vast undifferentiated forest. You knew there was something in there somewhere, but you did not know how to navigate to it. Now you have a feel for the tree. You know that `/usr/src/sys/dev/` is where device drivers live, that `/usr/src/sys/kern/` is where the core kernel code lives, that `/usr/src/sys/net/` is the networking stack, and that `/usr/src/share/man/man9/` is where the kernel API manual pages live. You can navigate. You can find. You can follow a function call from caller to callee and back. The tree has become a workspace, not a maze.

These changes are the real measure of the book. They are not about specific APIs you can recite from memory. They are about the shape of the world you see when you sit down at a terminal and open a driver. Take a moment to appreciate the shift. It is real, and it is yours.

### 38.4.4 1.4 Why Celebrating Progress Matters

Technical cultures often underplay the importance of recognising progress. The implicit assumption is that the work is its own reward and that anything more than that is sentimental. This is bad psychology, and it leads to burnout and to the curious phenomenon of experienced engineers who do not recognise their own expertise.

Recognising progress is not sentimental. It is functional. A learner who can see their own progress chooses harder next steps with confidence. A learner who cannot see their own progress stays at the easy steps because they do not know they have outgrown them. The point of celebrating is not to feel good, or at least not only to feel good. It is to clear the fog from the next decision about what to learn and what to build.

The book has been a long investment. Measured in hours, it is probably a larger investment than you might notice in aggregate. An honest accounting of the time you spent reading, working through labs, and thinking between sessions would likely surprise you. That time should produce outputs. One of the outputs is skill. Another is the confidence to take on the next project. Both outputs are easier to use when you have acknowledged them.

There is a specific kind of progress worth noting. You have built up not only facts about FreeBSD but also a way of working. You have learned to slow down before complex problems. You have learned to write tests. You have learned to read source code before writing your own. You have learned to decompose a kernel-space task into manageable pieces. These are portable habits. They apply to work outside FreeBSD, outside kernel programming, and even outside systems software. Engineers who have them are better engineers in many contexts, and engineers who do not have them struggle even in familiar territory.

If you have been reading this section quickly and feel tempted to skip past it, pause. Take five minutes. Think back through the book. Notice one concrete thing you can do now that you could not do before. Write it down. That is the exercise that makes the section count.

### **38.4.5 1.5 Exercise: Write a Personal Reflection**

This subsection's exercise is simple in structure and hard to do well. Write a personal reflection on your experience working through this book. It can be a blog post, a private notebook entry, an email to a friend, or a personal journal page.

The reflection should not be a summary of what the book covered. The book has done that for itself. The reflection should be about what your experience was. What surprised you about kernel development? Which chapter changed your understanding the most? Where did you get stuck, and what got you unstuck? What do you think about FreeBSD now that you did not think at the start?

Writing reflections like this is more valuable than it looks. It is a form of learning-to-see. The act of writing forces a specific articulation, and a specific articulation is transferable in ways that a vague impression is not. Ten years from now, if you are still writing FreeBSD drivers, you will benefit from having written this reflection now. You will also benefit from being able to read it, because your sense of what was hard at this stage will shift as you grow, and the only way to keep an honest record of where you were is to write it down when you were there.

A good reflection is usually between five hundred and fifteen hundred words. It does not need to be polished. It does not need to be publishable. It needs to be honest. If you share it publicly, the BSD community has a long tradition of welcoming such reflections on mailing lists, on personal blogs, and at conferences, and you may find that sharing your reflection connects you with people who went through the same process at a different time. If you keep it private, that is equally legitimate, and the act of writing is still the point.

Date the reflection. Put it somewhere you can find it again. Keep it for the future-you who will want to look back at the moment when you finished this book.

### **38.4.6 1.6 Wrapping Up Section 1**

This section asked you to pause and recognise what you have accomplished. That recognition is not ornamental. It is the ground from which informed next steps are chosen. A reader who cannot point to specific skills gained, specific shifts in perception, and specific artefacts produced by the book will struggle to decide what to do next. A reader who can point to those things will find the next decision easier.

We will now move into a more analytical section, where we look carefully at what the accomplishment means in practical terms. What, concretely, are you now capable of doing? Where are you strong, and where do you have room to grow? Section 2 turns those questions into a careful self-assessment. The answers will inform the planning work that comes later in the chapter.

## **38.5 2. Understanding Where You Stand Now**

Celebrating progress is one thing. Knowing where you stand is another. This section asks you to look at your skill set with engineer's eyes: not to judge yourself harshly, and not to flatter yourself, but to form an accurate and specific picture of what you can do, what you could do with a little more practice, and what still lies ahead. That picture is what makes the later parts of the chapter useful.

### 38.5.1 2.1 You Are Now Capable Of

Let us state it plainly, in the voice of someone describing a capable colleague. If someone asked what you can do with FreeBSD device drivers, here is an honest version of the answer.

You can write a kernel module for FreeBSD 14.3 that loads cleanly, registers a device node, handles read and write operations, and unloads without leaking resources. You can do this starting from scratch, using a text editor and the FreeBSD source tree as your only references, without needing to copy an existing module verbatim. You have done that repeatedly throughout the book, and the pattern is now in your hands.

You can debug that module when it crashes. If it panics, you can interpret the stack trace. You can set up a crash-dump configuration and attach kgdb to the resulting dump. You can add `KASSERT` statements to catch invariant violations earlier. You can run `dttrace` to see what the module actually did. You can watch `vmstat -m` to catch memory leaks. None of these are abstract to you; they are tools you have used.

You can interface with userland cleanly. You know how to implement `ioctl`s with the `_IO`, `_IOR`, `_IOW`, and `_IOWR` macros, and you know why `ioctl` numbers matter for ABI stability. You know how to use `copyin` and `copyout` to move data across the kernel/ userland boundary safely. You know how to expose state through `sysctl(9)` so that administrators can see and change what the driver is doing without needing to write a specialised tool.

You can interface with hardware. You know how to allocate memory and IRQ resources from a Newbus bus. You know how to read and write registers through `bus_space(9)`. You know how to set up DMA transfers, including the tag, the map, and the synchronisation calls. You know how to write an interrupt handler that does only what an interrupt handler should do and defers the rest to a taskqueue or an `ithread`. Each of these is a practical capability, not just a topic you read about.

You can submit a patch to the FreeBSD Project. You know the pre-submission hygiene: style check, manual-page lint, build across architectures, load/unload cycle. You know how to generate a patch that the project's tools can read. You know how to engage with reviewers productively. You know the difference between Phabricator and GitHub workflows and why the project has been moving between them. You have not necessarily submitted a real patch yet, but the workflow is no longer a mystery to you, and the day you decide to submit will not be the day you first learn how.

These are not ambitious claims. They are an honest accounting of what finishing this book means.

### 38.5.2 2.2 Confidence With Core Technologies

Let us now look at specific technologies you have learned, and let us ask about your confidence with each one. The purpose is not to grade yourself but to locate where you are strong and where you might want to invest some time.

**bus\_space(9):** You have used it to read and write registers. You know the difference between the tag, the handle, and the offset. You know that `bus_space_read_4` and `bus_space_write_4` are byte-order-respectful abstractions that work across architectures. You probably do not yet have a deep sense of how `bus_space` is implemented on different architectures, because that is a deeper topic, but at the level the book covered, you are comfortable.

**sysctl(9):** You have used it to expose tunable parameters. You know the tree structure of `sysctl` names, how to add a leaf with `SYSCTL_PROC`, and how to use the `OID_AUTO` pattern. You can read the output of `sysctl -a`



and understand where your driver's entries live. This is one of the most approachable kernel interfaces, and you have practised with it enough to reach for it naturally.

**dtrace(1):** You have used it to observe kernel behaviour. You know the `fbt` provider, the `syscall` provider, and the basics of the D language. You have probably not written your own static tracepoints yet, because that is a topic the book touched but did not work through exhaustively. That is an excellent area for continued learning, and the `dtrace(1)` manual page along with the DTrace guide make it a pleasure to study.

**devfs(5):** You know that device nodes appear under `/dev/` because `make_dev_s` was called at attach time. You know how cloning device drivers work. You have not likely explored the internals of `devfs` in full depth, because that is kernel internal territory and not something most driver authors need. You know it as a user: you know what it provides, and you have used it.

**poll(2), kqueue(2):** You implemented these for your async I/O work in Chapter 35. You know how the `poll` and `kqueue` subsystems interact with the kernel's wakeup mechanisms. You know when to pick one over the other, and you know why `kqueue` is generally the preferred interface for new code on FreeBSD. You have the working knowledge to implement these for any new driver you write.

**Newbus, DEVMETHOD, DRIVER\_MODULE:** These are the registration mechanics of a FreeBSD driver. You have written them enough times that they no longer feel unfamiliar. You know which methods are standard, how to handle custom methods, and how the registration order affects attach sequencing.

**Locking:** You have used mutexes, sx locks, rmlocks, condition variables, and epoch protections. Your confidence with each of these probably varies. Mutexes are likely the most natural to you; epoch may still feel less so. That is typical, and it is fine. Each of the more advanced synchronisation primitives has its own set of patterns that become clearer with use.

**Upstream workflow:** You know the mechanics of generating a patch, handling review, responding to feedback, and maintaining a driver after merge. You probably have not yet done a real submission, because doing one takes more than reading a chapter. Your first submission will probably feel slow and uncertain, even with all this preparation, and that is normal. The second and third will be faster.

Take a moment to rate your confidence with each of these, not as a formal exercise but as a quick mental map. Rate yourself from one to five, where one is “I have heard of it” and five is “I could teach it”. The locations where you rate yourself at three or four are probably your most productive next learning targets: you know enough to practise, and a little more practice will turn the skill into something you trust.

### 38.5.3 2.3 Review Exercise: Map the APIs You Used

This subsection contains a review exercise that is genuinely valuable and only takes about an hour.

Pick a driver you wrote during the book. It can be the final async driver from Chapter 35, the network driver from Chapter 28, a Newbus driver from Part 6, or any other non-trivial piece of code you produced. Open the source file. Go through it top to bottom. Every time you see a function call, a macro, a structure, or an API that came from the kernel, write it down.

When you are finished, you will have a list of somewhere between twenty and eighty kernel identifiers. Each of them represents an interface you invoked. Next to each one, write a one-sentence summary of what it does and why you used it. For example:

- `make_dev_s(9)`: creates a device node under `/dev/`. I used it in `attach` to expose the driver to userland.

- `bus_alloc_resource_any(9)`: allocates a resource of a given type from the parent bus. I used it to get the memory window for the device registers.
- `bus_dma_tag_create(9)`: creates a DMA tag with constraint parameters. I used it to set up the DMA layer before loading any maps.

The exercise has two benefits. The first is that it forces you to articulate, in your own words, what each interface does. That articulation is the difference between recognising an interface and understanding it. The second benefit is that the resulting list is a concrete record of what you know. You can keep it as a reference, and you can compare it against the next driver you write to see how your vocabulary has grown.

If you find an API on the list that you cannot explain in a sentence, that is a signal to open the manual page and read it. The exercise doubles as a diagnostic. The places where your explanation is weak are the places where more practice will have the most impact.

### 38.5.4 2.4 The Difference Between Having Followed the Book and Being Independent

There is an important distinction worth naming clearly. Finishing this book is not the same thing as being able to work independently, and it is worth understanding the distinction because it shapes the rest of the chapter.

When you followed the book, you had a guided path. The chapters introduced concepts in an order that had been thought about. The labs built on each other. When you got stuck, the next paragraph of the book often addressed the sticking point. The author had anticipated many of your questions and answered them before you asked. That structure is valuable, but it has limits.

Working independently means you no longer have that structure. The problem in front of you is not organised in an order that was designed for your learning. The APIs you need to use may be ones that the book named but did not exercise in detail. The pitfalls you hit may be ones the book did not warn you about specifically. The time you spend stuck is not bounded by the length of a chapter; it is bounded by your own persistence.

Readers who finish a book like this sometimes feel disappointed when they sit down to their first independent project and discover that it is harder than the labs were. That is not a sign that the book failed them. It is the ordinary transition from guided to independent practice. The book has given you the tools to close the gap, but the closing is work that you do, not work that the book does for you.

Several things help with this transition. The first is to pick smaller independent projects than you think you need. An easy win is worth more than an ambitious failure, especially early on. A small pseudo-device driver that does one thing correctly is a better first independent project than an ambitious network driver that does not quite work. The second is to keep the book open beside you while you work. You are not cheating by referring back to it. You are using a reference, and that is what references are for. The third is to expect the first project to take longer than you think it should. That is not a bug. It is the shape of the transition.

Over time, your independence grows through practice rather than through reading. Each new driver you write, each bug you track down alone, each patch you submit to the project, adds to the pool of experience that lets you work without needing to look everything up. The book is a starting point for that process. The process itself is yours.

### 38.5.5 2.5 Identifying Your Strongest and Weakest Areas

A useful self-assessment exercise at this stage is to identify, specifically, which areas you feel most confident in and which areas you feel you still need to deepen.

Confidence comes from having applied the knowledge successfully several times. If you wrote a character driver in Part 2 and never again, your confidence in that area is single-instance. If you handled interrupts in Chapter 19 and again in Chapter 20, and in a lab in Chapter 35, your confidence in that area has been exercised multiple times. The former is knowledge; the latter is skill.

Go through the major topics of the book and, for each one, ask yourself honestly: if someone handed me a new problem in this area, could I start on it without looking things up? The topics where the answer is a clear yes are your strongest areas. The topics where the answer is “I would need to refresh” are your next-level areas, the ones a little more practice will bring up to strong. The topics where the answer is “I remember the idea but not the specifics” are the ones that would benefit from more serious study.

Be honest with yourself. There is no external standard you are measuring against, and there is no grade. The value of the exercise is the map it produces. A map that says “I am strong in concurrency and character drivers, moderate in network drivers and DMA, weak in VFS and storage” is more useful than a vague impression of overall competence.

Write the map down. Save it. In six months, when you have done some of the practice that the later sections of this chapter will recommend, redo the exercise. You will probably find that several topics have moved up a category. Seeing that movement, documented from your own hand, is one of the most motivating things you can do for yourself as a learner.

### 38.5.6 2.6 Calibrating Against Real FreeBSD Developers

There is a calibration question worth addressing, because readers often wonder how their current skill level compares to the people working in the FreeBSD community.

The answer, honestly, depends on who you compare yourself to. A senior committer who has been working in the kernel for twenty years will have a depth of understanding that a single book cannot give you. A FreeBSD developer who has been writing drivers professionally for five years will know idioms and pitfalls that are rarely written down. A core-team member will have a broad view of the project that only governance work produces. None of those people are the right comparison, and measuring yourself against them will only discourage you.

The useful comparison is to a working junior FreeBSD developer. A person who can write drivers for specific hardware, respond to reviewer feedback, maintain the drivers over time, and contribute productively to the community. Most people at that level have some overlap with you in experience. They know things you do not know; you also know things they do not know, especially if the book covered a topic they never formally studied. You are, at this point, close to that junior-developer peer group. A little more practice, a first upstream submission, and a few months of maintenance will likely close the gap.

Another useful comparison is to the version of yourself that started the book. This comparison is almost always favourable and is therefore useful for motivation rather than for strategic planning. Use it when you are tired and need to see your progress. Use the junior-developer comparison when you are planning what to work on next.

### **38.5.7 2.7 The Shape of Competence You Have Built**

A final observation about where you stand. The competence you have built from this book is shaped in a particular way. It is deeply oriented toward FreeBSD-specific practice. It is strong on the driver-writing side of kernel work and thinner on other areas of the kernel that the book did not cover in depth. It is grounded in real source-tree work rather than abstract systems-programming principles.

This shape is the shape the book was designed to produce. It means that you are well prepared for the kind of work that FreeBSD drivers involve, and less prepared for adjacent areas like filesystem internals or deep VM work. That is not a deficiency; it is the boundary of the book's scope. The later sections of the chapter will point you toward the resources for extending that shape in the directions you choose, and the list of advanced topics in Section 3 names the most common extensions.

The shape also means that your knowledge transfers reasonably well to the other BSDs. OpenBSD and NetBSD share many idioms with FreeBSD, and a driver developer who knows FreeBSD can read OpenBSD or NetBSD drivers with effort but without feeling lost. There are differences, and they matter, but the underlying craft is recognisably the same. Readers who want to extend their competence across the BSD family will find that the investment of this book carries them a substantial part of the way.

### **38.5.8 2.8 Wrapping Up Section 2**

You now have a clearer picture of where you stand: what you can do, what you know well, where your strongest and weakest areas are, and how your competence compares to the community you are joining. That picture is practical. It informs the next part of the chapter, where we look outward at the advanced topics that lie beyond the book's scope.

The purpose of the next section is not to teach those advanced topics. The book would need to double in length to do that properly, and several of them are their own multi-year study tracks. The purpose is to name them, explain briefly what each one covers, and point you toward the resources where you can pursue them seriously. A map, in other words, rather than a curriculum.

## **38.6 3. Exploring Advanced Topics for Continued Learning**

The FreeBSD kernel is a large system, and no single book can cover all of it in depth. This book has concentrated on device drivers, which is one of the most approachable entry points into kernel work and one of the most useful. There are several important areas of the kernel that intersect with driver work but deserve their own dedicated study. This section names those areas, describes briefly what each one covers, and points you toward the sources you can use to pursue them.

The spirit of this section is pointing, not teaching. If you find yourself wanting deeper treatment of any of these topics, that is a sign of interest rather than a gap in this chapter. The resources we name below are the next step. The chapter itself is not going to turn into a file systems course or a network-stack course. Each of those is a years-long study, and compressing them into Chapter 38 would do justice to neither.

### 38.6.1 3.1 Filesystems and Deeper VFS Work

One of the most interesting areas beyond driver work is the filesystem layer. The book touched filesystem-adjacent material in Chapter 27, where we looked at storage devices and the VFS integration that sits above them. That chapter showed you the shape of a GEOM provider and the relationship between a storage driver and the block layer. It did not teach you how to write a filesystem.

A filesystem in FreeBSD is a kernel module that implements the `vop_vector` set of operations on `vnodes`. The `vnode` is the kernel's abstraction of an open file, and a filesystem provides the backing for a family of `vnodes`. FreeBSD's native filesystems include UFS, ZFS, `ext2fs-compat`, and several others. Writing a new filesystem means implementing at least the core operations: `lookup`, `create`, `read`, `write`, `getattr`, `setattr`, `open`, `close`, `inactive`, and `reclaim`, along with lesser-used operations for directory management, symbolic links, and extended attributes.

Serious study of this area means reading the UFS source under `/usr/src/sys/ufs/` and the VFS(9), `vnode(9)`, and individual `VOP_*(9)` manual pages such as `VOP_LOOKUP(9)` and `VOP_READ(9)`. Marshall Kirk McKusick's book "The Design and Implementation of the FreeBSD Operating System," second edition, contains a chapter on the VFS architecture that is the single best starting point for serious study. The ZFS code under `/usr/src/sys/contrib/openzfs/` is a second, very different model: a copy-on-write filesystem with its own layered abstraction and a history of Solaris origins.

Filesystems intersect with driver work in two directions. First, storage drivers provide the block-layer backing that filesystems consume, and a driver developer who understands filesystems will write better storage drivers. Second, some drivers implement filesystem-like interfaces directly, exposing a tree of virtual files for user interaction. The `devfs` filesystem itself is an example, and so is `procfs`.

If this area interests you, a productive starting project is to read the entire source of one of FreeBSD's simpler filesystems, such as the `nullfs` or `tmpfs` implementation. Both are small enough to study in a week or two, and both illustrate the `vnode` and VFS patterns clearly without the complications of on-disk structures.

### 38.6.2 3.2 Network Stack Internals

A second major area is the FreeBSD network stack. The book covered network drivers in Chapter 28, which taught you how to write a driver that participates in the `ifnet` layer by providing packet transmit and receive functions. That is the driver side. The stack itself, which sits above `ifnet` and implements the protocols you use every day, is a much larger subject.

The FreeBSD network stack is among the most sophisticated code in the kernel. It implements IPv4, IPv6, TCP, UDP, and a dozen other protocols. It includes advanced features like VNET for network-stack virtualisation, `iflib` for multi-queue network interface frameworks, and hardware-offload interfaces for devices that implement parts of the stack directly in silicon. Reading this code is a substantial undertaking, and working in it productively is a multi-year project.

Key starting points for study are the files under `/usr/src/sys/net/`, `/usr/src/sys/netinet/`, and `/usr/src/sys/netinet6/`. The core `ifnet` layer lives in `/usr/src/sys/net/if.c`, and the interface-framework abstractions used by modern drivers live in `/usr/src/sys/net/iflib.c`, whose header credits Matthew Macy as the original author. BSDCan and EuroBSDcon recordings from recent years are a well-regarded starting point for the broader `iflib` and VNET story, and the `iflib(9)` manual page pulls the in-tree reference material together.

Adjacent to the core stack is netgraph, a framework for composable network processing that sits apart from the normal IP stack. Netgraph lets you build protocol pipelines out of small nodes that exchange messages. It is useful for specialised protocol work, PPP-style encapsulation, and network-device prototyping. The documentation lives in `ng_socket(4)`, the manual pages starting with `ng_`, and the source under `/usr/src/sys/net-graph/`.

If the network stack interests you, a useful starting project is to write a simple netgraph node that performs a straightforward transformation on packets, such as a statistics counter. That project lets you touch the netgraph interfaces, the mbuf system, and the VNET mechanics without needing to implement a protocol.

### 38.6.3 3.3 USB Composite Devices

A third area, narrower than the previous two, is USB composite devices. The book covered USB drivers in Chapter 26, which taught you how to write a driver for a single-function USB device. Composite devices are USB devices that present multiple interfaces on a single hardware connection, such as a printer that also exposes a scanner, or a headset that exposes audio output, audio input, and a HID control channel.

Writing a composite-device driver is significantly more complex than writing a single-function USB driver. The main additional complexity is in the interface-selection logic, in the coordination between the different functional components of the driver, and in the correct handling of USB configuration changes. The FreeBSD USB stack under `/usr/src/sys/dev/usb/` supports composite devices, and there are several composite drivers in the tree that you can study as examples.

Serious study of this area means reading the USB specification, the FreeBSD USB stack source, and the existing composite-device drivers. The `usb(4)` manual page is the starting point, and `/usr/src/sys/dev/usb/controller/` and `/usr/src/sys/dev/usb/serial/` contain many examples of real-world driver structure. The relevant USB specification is the USB 2.0 specification from the USB Implementers Forum, which is freely available and remarkably readable.

A useful starting project if this interests you is to pick up a cheap composite USB device (many multi-function printers work) and write a FreeBSD driver for its less-supported function. This kind of project gives you a clear target, real hardware that you can observe with `usbdump(8)`, and a concrete endpoint when the driver works.

### 38.6.4 3.4 PCI Hotplug and Runtime Power Management

A fourth area is PCI hotplug and runtime power management. The book covered PCI drivers in Chapter 18 and power management at the level of suspend and resume in Chapter 22. Those chapters prepared you for devices that appear at boot, remain throughout the session, and disappear at shutdown. They did not fully cover the case where devices come and go during runtime.

PCI hotplug has become important with the rise of PCI Express, which supports physical hotplug of cards through connectors like U.2, OCuLink, and the internal hotplug slots on server-class hardware. A driver that needs to support hotplug must handle detach at times other than shutdown, must reason about references held by other subsystems, and must be robust against partial detach.

Runtime power management is the companion topic. Modern PCI devices support low-power states that the driver can enter when the device is idle and exit when the device is needed again. The ACPI subsystem in

FreeBSD provides the underlying mechanism. A driver that uses runtime power management can save significant power, especially on laptops and battery-powered embedded devices, but it needs careful reference-counting and state-machine logic to work correctly.

The relevant FreeBSD sources are under `/usr/src/sys/dev/pci/` and `/usr/src/sys/dev/acpica/`. The `pci(4)` manual page and the `acpi(4)` manual page are the starting points. The ACPI specification itself is a substantial document but is readable, and it is worth at least a skim if runtime power management is an area you want to work in.

A reasonable starting project is to take a driver you wrote during the book and add runtime power management to it. Even though the driver may not have a real hardware device to exercise the low-power states on, the exercise of adding the correct state machine, the reference counting, and the wake handlers is valuable.

### 38.6.5 3.5 SMP and NUMA-Aware Drivers

A fifth area is the intersection of driver writing with symmetric multiprocessing (SMP) and non-uniform memory access (NUMA) tuning. The book has covered the basics of locking and concurrency throughout Part 3, and you have been writing SMP-safe drivers since then. What the book did not cover in depth is how to write drivers that are not just SMP-safe but also SMP-scalable, and how to handle NUMA topology explicitly.

An SMP-safe driver is one that will not crash or corrupt state on a multiprocessor machine. That is a baseline requirement. An SMP-scalable driver is one whose throughput grows reasonably as you add more CPUs. That is a much harder target. It requires careful attention to cache-line sharing, to the granularity of locks, to per-CPU data structures, and to the flow of work between processors. Most of the high-performance drivers in FreeBSD, particularly network drivers for 10 gigabit and faster interfaces, use advanced techniques to achieve scalability.

NUMA awareness is the next layer. On a NUMA machine, different regions of physical memory are closer to different CPUs. A driver that pins its DMA buffers and its interrupt handlers to the same NUMA node will be faster than one that does not. The FreeBSD NUMA subsystem under `/usr/src/sys/vm/` provides the mechanisms, and the `numa_setaffinity(2)` system call and the `cpuset(2)` interfaces are the starting points.

The advanced network drivers under `/usr/src/sys/dev/ixgbe/`, `/usr/src/sys/dev/ixl/`, and related directories are good examples of SMP-scalable and NUMA-aware driver design. Reading them is a graduate course in this area. The `iflib` framework under `/usr/src/sys/net/iflib.c` and the adjoining headers provides the primary scaffolding for modern network drivers of this class.

A useful starting project in this area is to take a driver you wrote during the book, add per-CPU counters, and benchmark it on a multi-CPU system. Even if the driver does not need SMP scalability in practice, the exercise of adding per-CPU state and measuring the difference is a valuable introduction to the thinking.

### 38.6.6 3.6 Additional Areas Worth Naming

Beyond the five topics above, several other areas of the FreeBSD kernel are worth knowing about, even if this chapter cannot cover them in detail.

**Jails and bhyve integration.** The jail subsystem and the bhyve hypervisor each have implications for driver work. A driver that can be used inside a jail, or that participates in bhyve's I/O virtualisation, has requirements

that ordinary drivers do not. The relevant sources are under `/usr/src/sys/kern/kern_jail.c` and `/usr/src/sys/amd64/vmm/`.

**Audit and MAC frameworks.** FreeBSD has a sophisticated audit framework, implemented through `auditd(8)`, and a mandatory access control (MAC) framework implemented through MAC policy modules. Drivers that need to participate in security audit trails, or that need to enforce MAC policies, have additional interfaces available. The `audit(8)` manual page and the `mac(9)` manual page are the entry points.

**Cryptographic framework.** The FreeBSD kernel has a built-in cryptographic framework under `/usr/src/sys/opencrypto/` that drivers can register with to expose hardware cryptographic acceleration. If your target hardware includes a crypto engine, this is the integration path. The `crypto(9)` manual page describes the interface.

**GPIO and I2C buses.** Embedded drivers often need to talk to GPIO lines and I2C peripherals. FreeBSD has first-class support for both, with manual pages `gpio(4)` and `iic(4)` and source under `/usr/src/sys/dev/gpio/` and `/usr/src/sys/dev/iicbus/`.

**Sound drivers.** The audio subsystem has its own conventions, defined in `/usr/src/sys/dev/sound/`. It is a bus-like framework with its own concepts of a PCM stream, a mixer, and a virtual sound card. Writing a sound driver is an interesting specialty and a reasonable next step for someone who enjoyed the character driver work of Part 2.

**DRM and graphics.** Graphics drivers in FreeBSD have traditionally been ports of the Linux DRM framework, living in the `drm-kmod` port. Writing or maintaining a graphics driver is a very different activity from writing a typical device driver, and it is a specialised field. The `drm-kmod` project has its own documentation and its own mailing list.

Each of these areas is a direction you could go. None of them is a required next step. The right next step is the one that interests you enough to sustain the work.

### 38.6.7 3.7 Exercise: Pick One, Read One Manpage, Write One Paragraph

Here is a small exercise that has disproportionate value. Pick one of the areas above, exactly one, that sounds interesting to you. Read the relevant FreeBSD manual page. This means opening a terminal, typing something like `man 9 vfs` or `man 4 usb`, and reading what you find. Expect it to take twenty to forty minutes.

After you finish reading, close the manual page. On a blank page in your notebook or on a fresh text file, write a one-paragraph summary of what the interface is for, what the main data structures are, and what a driver author would use it for. Do not refer back to the manual page while writing. The point is to test your own understanding, not to produce a polished description.

Then open the manual page again and compare. The places where your paragraph was vague or wrong are the places where you actually did not understand yet. Revise. Write a better paragraph. Save it.

This exercise is valuable for three reasons. First, it gives you hands-on practice with FreeBSD manual pages, which are an underused resource and one of the best technical documents in the system. Second, it trains the skill of summarising a technical interface in your own words, which is the foundation of understanding. Third, the paragraph itself is something you can keep as a reference. Over time, a collection of such paragraphs becomes a personal glossary of the kernel areas you have explored.



### 38.6.8 3.8 Wrapping Up Section 3

This section has given you a map of advanced topics rather than a curriculum. The map shows where the book's coverage ends and where further study begins. For each topic, we named the topic, described briefly what it covers, pointed to the relevant parts of the source tree and manual pages, and suggested a starting project.

The most important thing about this section is the suggestion that you not try to study all of these topics at once, and not try to study any of them at all unless you actually find yourself interested. Depth of understanding comes from sustained attention to one topic over time, not from a whirlwind survey of many topics. The list above is a menu, not a curriculum.

The next section turns from advanced topics to the practical work of building yourself a reusable development toolkit. That toolkit is the scaffolding that makes whatever specialised area you pursue easier to pursue, because it frees you from the friction of starting from scratch on every new project. The toolkit and the advanced topics together are the practical engine of your continued growth.

## 38.7 4. Building Your Own Development Toolkit

Every experienced driver developer has a set of tools and templates they reach for reflexively. A starter driver project that captures the patterns they always use. A virtual lab that boots in seconds and lets them try things without fear of breaking anything. A set of scripts that handle the tedious parts of testing. A habit of writing regression tests before submitting patches. These tools are not provided by the kernel, and they are not covered in any specific chapter of the book. They are built up over time by each developer, and they pay for themselves many times over.

This section walks through building that toolkit. The examples directory for this chapter contains starter templates, and you should treat the templates as a first draft that you then modify to suit your own working style. The point is not to use the templates verbatim; the point is to have something to start from, and then to evolve it over the next several projects into something that fits how you actually work.

### 38.7.1 4.1 Setting Up a Reusable Driver Template

The first tool in your toolkit is a driver project template. When you start a new driver, you should not begin with a blank file. You should begin by copying a template that already contains the copyright header you use, the standard includes, the Newbus method table shape, the probe-attach-detach skeleton, the softc convention, and the Makefile that builds it all into a loadable module.

The template should be opinionated. It should reflect your own style choices within the constraints of FreeBSD's `style(9)` guidelines. If you always use a particular locking pattern, the template should include it. If you always define a `*_debug sysctl`, the template should include it. If you always keep your register definitions in a separate header, the template should include an empty one named correctly.

The template does not need to be sophisticated. Five files is enough for most purposes: a main `.c` file with the driver skeleton, a `.h` file for internal declarations, a `reg.h` file for register definitions, a `Makefile` for the module build, and a manual page stub. The whole thing fits in a few hundred lines. You can iterate on it.

The companion directory for this chapter contains a working template you can start from. Copy it, place it under version control, date it, and start making it your own. Every time you finish a driver project and notice a pattern

you reached for repeatedly, consider whether the pattern should go into the template. Over the course of several projects, the template becomes a compressed record of your accumulated style.

There is a subtle trap worth avoiding. The template should be kept simple enough that you can type it from memory if you had to. If the template becomes so elaborate that you depend on it for patterns you could not reproduce manually, you have moved from having a useful tool to having a crutch. The tool helps you move faster; the crutch hides what you know and do not know. Keep the template on the useful side of that line.

A second subtle consideration is licensing. Your template should contain the copyright header you use for your own code. If you typically write drivers under the BSD-2-Clause license, as is standard in FreeBSD, the template should have that header. If you sometimes write drivers under a different license for professional reasons, keep separate templates for each license. Getting the license right at the start of a project is easier than changing it later, and a template that starts with the correct license is one fewer thing to remember on each new project.

### 38.7.2 4.2 Building a Reusable Virtual Lab

The second tool in your toolkit is a virtual lab. This is the environment where you load your drivers, test them, crash them, and iterate. It should be separate from any machine you care about, it should be easy to rebuild, and it should be fast enough that you do not dread turning it on.

FreeBSD provides two primary virtualisation paths for this purpose: bhyve and QEMU. Bhyve is the native FreeBSD hypervisor, and it is an excellent choice for a lab when your host is FreeBSD. QEMU is a portable emulator that runs on many host operating systems, and it is the right choice if your host is Linux, macOS, or Windows. Both are capable of running FreeBSD guests with good performance, and both have communities and documentation around them.

A reasonable lab setup has the following properties. It boots in under thirty seconds. It exposes a serial console rather than a graphical one, because serial consoles are much easier to automate and to log. It shares a source directory with the host so that you can edit code on the host and build it in the guest without transferring files manually. It has a snapshot of a known-good state so that you can recover quickly after a panic. It is configured to dump its memory on panic so that you can attach kgdb to the crash dump.

Setting up such a lab is a one-time investment of a few hours, and the payoff is an environment you will use for every future driver project. The companion directory for this chapter contains example bhyve and QEMU configurations, along with a short script that creates a disk image, installs FreeBSD into it, and configures the guest for driver development work.

For some drivers, you will want a slightly more elaborate lab. If you are writing a GPIO driver, you will want an attached simulator that can model GPIO transitions. If you are writing a USB driver, you will want the ability to pass a USB device through from the host to the guest, which both bhyve and QEMU support. If you are writing a network driver, you will want virtual network interfaces connecting the guest to the host and possibly to other guests. Each of these is a refinement of the base lab, and each of them takes an afternoon to set up properly.

One useful pattern is to keep your lab configuration in a version-controlled repository. Write the bhyve or QEMU command line in a script rather than typing it from memory each time. Store the disk image and the snapshots somewhere you can find them. Document the guest's network configuration in a README. The lab is code, and it should be treated that way.

### 38.7.3 4.3 Loopback Testing With User-Mode Helpers

A particular lab pattern worth highlighting is loopback testing with user-mode helpers. Many drivers can be exercised fully by a userland program that exercises them. A network driver can be tested by a program that opens a socket to the driver's interface and sends packets. A character driver can be tested by a program that opens the device node and performs a sequence of operations. A sysctl-controlled driver can be tested by a script that sets sysctl values and checks for expected behaviour.

The pattern is to pair every driver with a small testing helper. The helper is usually a userland program in C or a shell script that invokes `sysctl`, `devstat`, or custom userland tools. The helper should cover at least the happy path: the driver loads, responds to normal operations, and unloads cleanly. A more thorough helper also covers edge cases: error paths, concurrent access, resource exhaustion.

Writing the helper alongside the driver, rather than afterwards, has a subtle benefit. It forces you to think about how the driver will be used from userland as you design the driver. If the helper is hard to write, that is a signal that the driver's userland interface is hard to use, and the time to redesign the interface is during development rather than after submission. This is one of the small engineering habits that separates drivers that are easy to maintain from drivers that are hard to maintain.

The companion directory contains an example helper that uses the standard character-driver operations. It is a template rather than a complete test suite, but it illustrates the pattern and gives you something to evolve.

### 38.7.4 4.4 Creating Regression Tests for Your Driver

The third tool in your toolkit is regression testing. A regression test is an automated check that a specific driver behaviour works correctly. You run the regression tests before submitting a patch, you run them after pulling in upstream changes, and you run them any time you are uncertain whether something you changed broke something else.

FreeBSD has a first-class testing framework called `kyua`(1), defined under `/usr/src/tests/`. The framework provides a way to declare test programs, group them into test suites, run them, and report the results. Driver tests can be written as Kyua tests, and the infrastructure handles all the plumbing: isolating tests from each other, capturing their output, and producing reports.

For a driver that exposes a userland interface through a device node, a reasonable test suite covers the following kinds of cases. It tests that the driver loads without warnings. It tests that the device node appears in the expected location. It tests that the standard operations produce the expected results. It tests that edge cases (zero-length writes, reads at end-of-file, concurrent opens) produce the expected errors rather than panics. It tests that the driver unloads cleanly.

A more thorough test suite also covers stress cases. It might run a hundred concurrent writers against a driver that is supposed to serialise correctly. It might repeatedly load and unload the driver to check for leaks. It might use `kttr`(9) tracing or DTrace probes to verify that particular code paths are exercised.

The companion directory contains a regression-test script skeleton that illustrates the pattern. It is not a complete test suite, but it is enough to show the structure. Extending it for a specific driver is a matter of adding specific test cases, and the Kyua documentation under `/usr/share/examples/atf/` shows the idiomatic way to structure them.

One important habit to develop is to write the regression test for a bug before you fix the bug. This is the discipline that regression tests exist to support. A test that does not exist before the fix is a test that may never exist, because once the fix is in, the motivation to write the test is gone. A test written first captures the bug in a concrete, reproducible form, and then the fix turns the failing test into a passing test. The test stays in the suite forever, and if the bug is ever reintroduced, the test catches it. This is an old discipline in software engineering, and it works as well in FreeBSD kernel work as in any other setting.

### **38.7.5 4.5 Lightweight Continuous Integration**

The fourth tool in your toolkit is lightweight continuous integration. This does not need to be a complex system. A single script that runs on each commit or each push, that builds your driver and runs its regression tests, is enough for most purposes.

The script can be as simple as a shell script that invokes the build and the tests in sequence. If a step fails, the script exits with a non-zero status, and you know to investigate. Over time, the script can grow to include style checks, manual-page linting, and cross-architecture builds. Each addition is incremental, and each addition catches a class of mistake that you would otherwise discover only in review.

If you have access to a continuous-integration system, such as GitHub Actions, GitLab CI, or a self-hosted runner, you can wire the script up to run on every push to your repository. The feedback cycle becomes: push a change, wait a few minutes, see whether it broke anything. This feedback cycle catches mistakes much earlier than manual testing catches them, and it frees your own time for thinking rather than rote checking.

A warning about CI complexity. It is tempting to build elaborate CI pipelines with many stages, caches, artifacts, and notification systems. For a solo developer on a small driver project, elaborate CI is usually a time sink. Start with a script that does the minimum. Add to it only when you notice a specific mistake that the current pipeline missed. Let the pipeline grow in response to real needs, not imagined ones.

The companion directory contains an example CI script that builds a driver, runs the style check, lints the manual page, and performs a basic load/unload cycle. You can adapt it to your repository of choice.

### **38.7.6 4.6 Package Your Driver With Documentation and Test Scripts**

A related habit worth cultivating is to package every driver you write with its documentation and its test scripts from the very beginning. Each driver in your repository should have the following files, at minimum: the source code, a module Makefile, a manual page, a README that explains what the driver does and how to use it, and a set of test scripts. If the driver is for specific hardware, the README should name the hardware. If the driver has known limitations, the README should name them.

The habit of packaging takes a little extra discipline, but it pays off in two ways. First, anyone who encounters your driver, including future-you, can understand it and use it without having to reconstruct the context. Second, the act of writing the README forces you to articulate what the driver does, and that articulation often reveals inconsistencies or missing features that you would not have noticed otherwise.

A small tip about READMEs. Write them in the second person, as instructions for a reader. “To load this driver, run `kldload mydev`.” This style is the standard for FreeBSD documentation, and it is easier to read than other styles, especially for someone skimming the file to learn how to use the driver.

### **38.7.7 4.7 Exercise: Package a Driver End-to-End**

Pick one of the drivers you wrote during the book. It should be one that is substantial enough to be worth packaging but small enough to finish the exercise in a weekend. The async character driver from Chapter 35, or the network driver from Chapter 28, or a Newbus driver from Part 6 are all reasonable candidates.

Create a new directory for the driver, separate from the book's examples directory. Copy the driver source in. Build the Makefile from scratch, using your driver template if you have one. Write a short manual page. Write a README. Write at least three regression tests. Write a load/unload script. Commit everything to a git repository.

When you are done, the directory should be self-contained. Hand it to a colleague and they should be able to build, load, test, unload, and understand the driver without needing to ask you questions. If they need to ask questions, those are gaps in your packaging that you can address.

This exercise is valuable beyond the specific driver. It is a rehearsal of the packaging habit, and packaging habits scale. Once you have packaged one driver well, the second one is easier, and the tenth one is automatic. Each time you do it, the quality of your packaging improves, and so does your sense of what makes a driver easy or hard to use.

### **38.7.8 4.8 Wrapping Up Section 4**

This section has walked through the practical toolkit that experienced FreeBSD developers build up over time: a driver project template, a virtual lab, loopback testing, regression tests, lightweight CI, and packaging habits. None of these is strictly necessary for writing a driver, but each of them makes the work easier, more reliable, and more durable.

The companion directory contains starter artifacts for all of them. You can adopt them verbatim or use them as a reference for building your own. Either path is valid, and the important thing is that you take some version of this toolkit into whatever driver work you do next.

The next section turns to the community of FreeBSD developers. A toolkit makes you productive. A community connects your productivity to the rest of the world and, over time, shapes you into a better engineer than solo work alone would produce.

## **38.8 5. Giving Back to the FreeBSD Community**

The FreeBSD Project is a community. It is not an abstraction; it is a set of people who read each other's patches, answer each other's questions, show up at conferences, argue about design choices, and together produce an operating system that has been evolving for over thirty years. This section is about how to become part of that community, how to give back to it, and how the engagement shapes you as a professional engineer.

### **38.8.1 5.1 Why Community Engagement Matters**

Let us start with the question of why this is worth discussing at all. A reader who has finished this book has the technical skills to write and maintain drivers in relative isolation. Does it really matter whether they engage with the community?

It does, and the reasons fall into several categories.

Community engagement is how your skills grow past the point a book can take them. Books cover patterns that are well understood enough to be written down. Community mailing lists, code review threads, and conference talks cover patterns that are emerging, controversial, or specialised. If you stop reading after the book, you stop growing in a specific way that the community would continue to develop in you.

Community engagement is how your work reaches other people. A driver in your personal repository serves you and a few people who happen to find your repository. A driver submitted upstream, discussed on a mailing list, and maintained in the project serves every FreeBSD user who encounters the hardware. The amplification is enormous, and you unlock it through engagement.

Community engagement is how you find the work that matters. Some of the most interesting projects in a large open-source ecosystem are not visible from the outside. They are discussed on mailing lists, at conferences, in IRC channels, and in informal chats. If you are in those conversations, you will hear about them. If you are not, you will miss them.

Community engagement is how you become someone the project trusts. Commit rights, mentorship opportunities, and leadership positions are not handed out; they are earned through a long history of visible, substantive engagement. The technical skill is the prerequisite. The community engagement is the path from prerequisite to actual standing.

None of these are obligations. You can finish this book, write drivers for your own purposes, and never engage with the community in any way. That is a legitimate path. But if you are curious about what deeper involvement looks like, the rest of this section shows you how to start.

### 38.8.2 5.2 Participating in Mailing Lists

The primary forum for FreeBSD development discussions is mailing lists. The project has many of them, each focused on a different area or audience. The ones most relevant to a driver developer are:

- **freebsd-hackers@**: general kernel development discussion. Broader than driver-specific topics, and the best starting point if you want a general sense of what the project is working on.
- **freebsd-drivers@**: focused on device driver topics. Lower volume than **freebsd-hackers** and more directly relevant if your interests are in driver work.
- **freebsd-current@**: discussion about the development branch of FreeBSD. Useful if you want to track recent changes and the discussions around them.
- **freebsd-stable@**: discussion about the stable branches. Lower volume, more release-focused.
- **freebsd-arch@**: architectural discussions about major changes to the system. Low volume, high signal-to-noise.

Subscribing to one or two of these is a reasonable first step. Start with **freebsd-hackers** or **freebsd-drivers**, and read for a few weeks before posting. The goal of the initial reading is to get a sense of the tone, the typical topics, and the people who participate regularly. Once you have that sense, you will know how to participate in a way that fits the culture.

Posting to a mailing list is a skill. A good post has a clear subject line, a concise body, and a specific question or contribution. A poor post wanders, asks several questions at once, or lacks enough context for anyone to help. When you are ready to post, take the time to draft the message carefully. Most experienced mailing-list

users spend more time on their posts than newcomers expect, and the investment shows up as higher-quality discussion.

A subtle skill is replying. Mailing list replies should quote just enough of the previous message to provide context, should address the specific point being made, and should avoid top- posting. These are conventions, not rules, but following them signals familiarity with the culture and makes your posts easier to follow. The RFC-style quoting convention, where you reply below each quoted paragraph, is the preferred style in FreeBSD mailing lists.

There is an etiquette around asking for help that is worth internalising. Before asking a question on a mailing list, you should have already read the manual pages, checked the source tree, searched the mailing list archives, and tried a few obvious solutions. When you do ask, include the information a reader would need: what FreeBSD version you are using, what hardware, what you tried, what you expected, and what actually happened. A well-formed question gets a helpful answer. A poorly-formed question often gets no answer, not because the community is hostile but because the community cannot help without more information.

### **38.8.3 5.3 Using Bugzilla**

FreeBSD tracks bugs in Bugzilla, which lives at <https://bugs.freebsd.org/bugzilla/>. It is the project's primary tool for recording defects, tracking progress, and coordinating fixes. A driver developer will interact with Bugzilla in several ways.

The first is reporting bugs. If you encounter a bug in a driver or in the kernel, you can file a PR (problem report) in Bugzilla. A good PR includes the FreeBSD version, the hardware, a clear description of the problem, reproduction steps, and any relevant output such as `dmesg` excerpts or crash dumps. The clearer the report, the more likely someone can fix the bug.

The second is triaging existing bugs. Bugzilla has a backlog of reports, some of which are not well-categorised or well- understood. A new contributor can add value by reading through unassigned PRs in the driver-related categories, reproducing the bugs on their own systems, and adding clarifying information to the reports. This kind of work is not glamorous, but it is genuinely valuable, and the contributors who do it learn a great deal about the shape of problems that drivers encounter in the wild.

The third is fixing bugs. When you find a bug you can fix, Bugzilla is the tool that coordinates the fix. You attach a patch to the PR, mark it for review, and follow the review through to commit. The process is not very different from the upstream submission workflow you learned in Chapter 37, except that it is targeted at a specific existing issue rather than at a new feature.

One useful habit is to subscribe to the Bugzilla product that covers drivers, or to a specific component you care about. The subscription gives you email notifications when bugs are filed, updated, or resolved. Over time, this subscription becomes a lightweight way of staying aware of what is going wrong in the area you maintain.

### **38.8.4 5.4 Contributing to Documentation**

Not all contributions are code. FreeBSD's documentation is a first-class part of the project, and it needs continuous work to stay current. For a new contributor, documentation work is one of the most accessible entry points.

The FreeBSD Handbook is the primary end-user document. It covers installation, administration, development, and specific subsystems. Its source lives in a separate Git repository at <https://git.freebsd.org/doc.git>, which you can browse online at <https://cgит.freebsd.org/doc/>. The documentation is written in AsciiDoc and rendered into HTML by Hugo; earlier generations of the documentation used DocBook XML, and you will still find references to that history in some older material.

Contributing to the Handbook is as simple as identifying a section that is outdated, incomplete, or confusing, writing an improvement, and submitting it. The documentation team welcomes such contributions and is usually quick to review them. The workflow is similar to code submission: clone the repository, make a change, generate a patch or pull request, and submit.

Manual pages are a second documentation target. Every driver should have a manual page, and every new FreeBSD feature should be documented in a manual page. The format is `mdoc`, defined in `mdoc(7)`, and the style guide lives in `style.mdoc(5)`. Writing good manual pages is a specialised skill, and a contributor who develops it is valuable to the project beyond their code contributions.

A specific area where manual-page work is always welcome is fixing outdated examples. Over time, manual pages accumulate references to options, files, or behaviours that have changed. Walking through a manual page, testing each example, and updating the ones that no longer work is useful, repeatable work that does not require deep kernel knowledge. It also teaches you the interfaces the manual page describes, which is its own benefit.

### 38.8.5 5.5 Translating Documentation

A third form of documentation contribution is translation. The FreeBSD Project maintains translations of the Handbook and other documents into many languages, coordinated through <https://docs.freebsd.org/> and the translation tools under <https://translate-dev.freebsd.org/>. If you are fluent in a language other than English and are willing to spend time on translation, the project has real, unmet needs for this work.

Translation is not simple substitution. A good translation requires understanding the technical content, understanding the conventions of the target language, and understanding the idiomatic expression of technical ideas in that language. It is serious work, and good translators are appreciated in proportion to how rare they are.

Working on translations puts you in contact with the FreeBSD Documentation Engineering Team, a small group of committers who maintain the documentation infrastructure. That contact is valuable beyond the immediate work, because it connects you to people who can help you understand the rest of the project.

### 38.8.6 5.6 Mentoring Other Beginners

A fourth form of contribution, and one that does not require commit rights or any specific standing in the project, is mentoring. Somewhere in the world, there is a reader who is at Chapter 5 of this book, struggling with kernel C, and wondering whether they should give up. If you can help that reader, you are contributing to the project in a way that matters more than almost any code patch could.

Mentoring happens in many forms. You can answer questions on the forums at <https://forums.freebsd.org/>. You can respond to questions on IRC in `#freebsd` or `#freebsd-hackers` on Libera Chat. You can participate in the Discord or Telegram channels that some parts of the community run. You can write blog posts that answer questions you struggled with when you were at that stage. You can give talks at local BSD user groups.



You can review someone else's first patch and offer suggestions in the tone you would have wanted when you were first reviewed.

The specific channels matter less than the habit of being available to help. Every person who finishes this book becomes, by that fact alone, someone who can help a reader who has not yet finished it. That is a real contribution to the sustainability of the FreeBSD community. The community grows when its senior members teach its junior members, and the senior/junior distinction here is measured in chapters completed, not in years of experience.

A useful pattern for mentoring is to focus on a specific channel and be reliably present there. If you answer questions on `freebsd-questions@` once a week for a year, you become a known face to the people who ask questions. If you respond to every driver-related question on the forums that you can answer, you become someone that newcomers look for when they have questions. Consistency matters more than intensity.

Be patient with beginners. Some of the questions will be things you remember finding hard, and some will be things you no longer remember finding hard because they have become obvious to you. Answer the obvious-to-you questions with the same care as the hard-for-you questions. That is what good mentoring looks like.

### **38.8.7 5.7 Contributing Driver Fixes**

A fifth form of contribution is driver fixes specifically. The FreeBSD tree contains hundreds of drivers, and at any given time several of them have known bugs or limitations. A new contributor with the skills from this book can fix some of these.

Finding driver bugs to fix is not hard. You can look at the open PRs in Bugzilla filtered by the `kern` or driver-specific products. You can read the mailing lists for reports of bugs that have not been fixed. You can use drivers yourself and report or fix the bugs you find. Any of these paths leads to real work the project needs.

Fixing a driver bug has several benefits beyond the fix itself. It teaches you how that particular driver works, which is an education in itself. It gives you practice with the upstream review workflow on real code that real users depend on. It builds a small history of contributions that, over time, adds up to a visible profile in the project.

One specific category of work worth mentioning is driver cleanup. Many older drivers in the tree have accumulated style issues, deprecated API usage, or missing manual pages over the years. Cleaning them up is not glamorous work, but it is exactly the kind of work the project needs and often welcomes. A contributor who is willing to do careful cleanup work on several older drivers develops a reputation for careful work very quickly, because careful work is always scarce.

### **38.8.8 5.8 Translating Knowledge to Other BSDs**

A sixth and less-commonly discussed form of contribution is carrying knowledge between the BSDs. OpenBSD and NetBSD share much of their ancestry with FreeBSD, and patterns that work in one often work in the others with adjustments. A driver developer who knows FreeBSD well can, with some additional study, contribute to OpenBSD or NetBSD as well.

The idioms are different in important ways. OpenBSD has its own locking conventions, its own interrupt handling patterns, and a strong emphasis on security and simplicity. NetBSD shares many patterns with FreeBSD

but has its own approach to things like device-tree integration and autoconfiguration. Each of these is a living system with its own community, and each of them has driver needs that a FreeBSD-trained developer can address.

The value of cross-BSD contribution goes beyond the individual commits. It keeps the three projects aware of each other, spreads good ideas between them, and prevents each from accumulating bugs that the others have already fixed. In an era where the BSD community is smaller than it once was, this cross-pollination is particularly valuable.

If this interests you, the starting point is to pick a driver you know well in FreeBSD and to look at whether OpenBSD or NetBSD supports the same hardware. If they do not, you have a clear porting target. If they do, you can still contribute by reading their version of the driver, comparing, and noting any improvements or fixes that could be shared. Either way, the engagement deepens your understanding of all three systems.

### **38.8.9 5.9 Exercise: Write a Thank-You Message**

This subsection's exercise is small and unusual. Send a thank-you message to someone whose code or documentation helped you during this book.

It could be the maintainer of a specific driver whose source you read closely. It could be the author of a manual page that clarified something for you. It could be a committer whose talk at a conference you watched on YouTube. It could be the person who wrote the book's foreword if one exists, or the person who reviewed it. Whoever helped you, in whatever way, say thank you.

The message does not need to be long. A few sentences will do. Name the specific piece of work that helped you. Explain briefly how it helped. Say thank you. Send the message by email or post it to the appropriate mailing list.

This exercise is valuable for two reasons. First, open-source work is often thankless, and maintainers who receive thank-you messages are more likely to keep contributing. You are, through the act of thanking, investing in the ecosystem's sustainability. Second, the act of writing the message will make you think specifically about how someone else's work helped you. That thinking, in turn, will make you more aware of how your own work might help someone else, and that awareness is the foundation of becoming a contributor yourself.

### **38.8.10 5.10 The Relationship Between Contribution and Growth**

There is a deeper point about community engagement that is worth naming explicitly. Contributing to the FreeBSD Project, in any of the forms above, is not just a way of giving back. It is a way of growing.

When you answer a beginner's question on a mailing list, you learn what was unclear about the topic. When you review someone else's patch, you learn patterns you would not have encountered in your own code. When you fix a bug in an area you did not write, you learn that area's conventions and history. When you write a manual page, you learn the discipline of precise technical explanation. Each act of contribution is also an act of education, and over time the education accumulates into a depth of understanding that solo work cannot produce.

The most senior FreeBSD developers are often the most prolific contributors precisely because contribution is how they continued to grow. They did not arrive at their current depth and then start contributing. They started

contributing, which led to their current depth. If you want to continue growing in the way they did, the path is visible and well-trodden.

This is not a moral argument. It is a practical observation about how expertise develops in open-source ecosystems. The people who engage deeply grow deeply. The people who stay on the periphery stay at whatever level the materials they consume took them to. Both paths are valid; if you want the former, the chapter has now shown you how to start.

### 38.8.11 5.11 Wrapping Up Section 5

This section has walked through the many forms of community contribution available to a reader of this book: mailing-list participation, Bugzilla engagement, documentation work, translation, mentoring, driver fixes, and cross-BSD contribution. Each of these is a legitimate way to be part of the FreeBSD community, and all of them are valuable beyond the specific work they involve.

The key takeaway is that contribution is broader than committing driver code. A reader who thinks of contribution only as “write a driver and send it upstream” is missing most of the contribution surface. Some of the most important work in the project is done by people who rarely commit code but who contribute in other ways.

The next and final major section of the chapter turns from the community to the living system itself. The FreeBSD kernel is under active development, and staying connected to that development is its own skill. How do you follow changes? How do you notice when something you rely on is about to change? How do you stay current without drowning in the volume of daily activity?

## 38.9 6. Staying Up to Date With FreeBSD Kernel Development

The FreeBSD kernel is a moving target. The version you learned against was 14.3, and by the time you read this at a later date, there will already be a 14.4, or a 15.0, or both. Each release brings new subsystems, deprecates old ones, changes internal APIs, and shifts conventions in small ways. A driver developer who writes a driver once and walks away will find, returning a few years later, that the driver may not build against the current tree. A driver developer who stays engaged with the kernel’s development watches those changes happen in real time and adjusts along the way.

This section is about how to stay engaged. Like the community section before it, the advice here is optional. You can finish this book, write drivers against FreeBSD 14.3, and never update your knowledge; your drivers will work for a while and then gradually stop working, and that is a legitimate relationship to have with the kernel. But if you want your drivers to keep working, or if you want to grow beyond the snapshot of knowledge that this book gave you, the habits in this section are what make that possible.

### 38.9.1 6.1 Where to Follow FreeBSD Development

There are several primary sources for following FreeBSD development, and a well-calibrated developer watches a few of them regularly without trying to watch them all.

**The FreeBSD Git repository.** The source tree itself lives at <https://git.freebsd.org/src.git>. Every commit to the tree is visible there, along with its commit message, its author, and its review history. You can

clone the repository and run `git log` to see the recent activity. You can use `git log --since=1.week` to filter to recent changes. You can use `git log --grep=driver` to look at driver-related commits.

For most developers, the Git repository is the primary source of truth. The commit logs are where the project's day-to-day engineering happens, and reading them regularly is one of the most direct ways to stay aware of what is changing and why.

**Commit notification mailing lists.** The project publishes commit messages to mailing lists such as `svn-src-all@` (historically, when the project used Subversion) and their Git-era equivalents. Subscribing to these gives you a high-volume stream of every commit to the tree. This is too much information for most purposes, but it is useful for a narrow audience: developers who want to watch every change.

A lower-volume alternative is to watch only the head branch's commit messages for a specific subsystem. You can do this with Git's `--author` or `--grep` flags, or by setting up a custom filter on your mail client.

**freebsd-current@ and freebsd-stable@.** These mailing lists are where discussions about the development and stable branches take place. Subscribing to them gives you early awareness of proposed changes, migration questions, and breakage. They are moderate volume and often high signal.

**The Release Notes.** Every FreeBSD release has release notes that describe significant changes since the previous release. The notes are published on the project's website at <https://www.freebsd.org/releases/>. Reading the release notes for each new release is an efficient way to catch up on changes you may have missed in the day-to-day volume.

**UPDATING.** The `/usr/src/UPDATING` file in the source tree contains important notices about changes that may affect users or developers. If a subsystem is being deprecated, or if an API is changing incompatibly, UPDATING is where the notice lives. Developers should check UPDATING after any significant tree update, and specifically before updating from one major release to another.

### 38.9.2 6.2 Developer Summits and BSD Conferences

FreeBSD has a rich conference culture, and attending or watching recordings of these conferences is one of the most efficient ways to stay connected to the project.

**BSDCan.** An annual conference held in Ottawa, Canada, usually in June. It draws developers from FreeBSD, OpenBSD, NetBSD, and DragonFly BSD. The presentations cover a mix of development updates, architectural discussions, and deep technical talks. Many of the talks are recorded and published on the conference website.

**EuroBSDcon.** An annual European BSD conference, held at a rotating location each September or October. The focus is similar to BSDCan, with a more European-centric participant list.

**Asia BSDCon.** An Asia-Pacific BSD conference, held annually in Tokyo. Smaller than BSDCan or EuroBSDcon but with a distinct set of participants and topics.

**The FreeBSD Developer Summit.** Held twice a year, once co-located with BSDCan and once with EuroBSDcon. The summit is where the project's committers meet face-to-face to discuss architectural direction, plan releases, and coordinate on major changes. Summaries of the summit sessions are published on the wiki, and the summit is one of the places where the project's internal planning becomes visible to the broader community.

**Regional BSD user groups.** Local BSD user groups exist in many cities and often host talks, meetups, and workshops. These are much lower-key than the international conferences but are often the most accessible entry point for someone who is new to the community.

For readers who cannot attend in person, many conference talks are recorded and posted online. The BSD Channel on YouTube, the BSDNow podcast, and the AsiaBSDCon recordings are all valuable resources. A habit of watching one conference talk per month is a low-effort way to stay connected to the project's current thinking.

### 38.9.3 6.3 Tracking API and Driver Model Changes Across Releases

Beyond following commits and attending conferences, there is a specific skill worth naming: tracking how the APIs and driver model change from release to release. This is the skill that keeps drivers buildable over time, and it is one of the least discussed aspects of long-term kernel work.

The basic pattern is this. When a new FreeBSD release comes out, you ask: what changed in the subsystem my driver touches? You answer the question by diffing the relevant parts of the source tree between the previous release and the new one. You read the release notes for anything flagged. You check UPDATING. You rebuild the driver against the new release and see what warnings or errors appear.

The tools that help with this are the standard Unix tools. `git log` with a path argument shows the history of a specific file or directory. `git diff` between two tags shows the difference. `grep` with the right patterns finds usage of a specific API. Each of these is elementary, but putting them together into a checking habit is a skill that repays itself many times over.

A specific example. Suppose your driver uses `bus_alloc_resource_any` and you want to know whether its semantics have changed between FreeBSD 14.0 and 14.3. You can run:

```
$ git log --oneline releng/14.0..releng/14.3 -- sys/kern/subr_bus.c
```

in a clone of the FreeBSD source tree. The output is the list of commits that modified `subr_bus.c` between those two releases. You can read each commit message to see whether the changes affect your usage. If something looks relevant, you can investigate with `git show` to see the actual change.

This pattern is fundamental to long-term driver maintenance. It is not glamorous, but it is reliable, and it catches problems before they become silent breakage.

### 38.9.4 6.4 Tools for Comparing Kernel Trees

Beyond `git log` and `git diff`, a few other tools help with the work of following kernel changes.

**diff -ruN.** The classic recursive diff between two directories. Useful when you have checked out two versions of the tree and want to compare them systematically. The output is large but readable, and it catches changes that `git log` alone might miss.

**git grep.** The Git-aware grep. Faster than external grep on large repositories because it knows about Git's index. Useful for finding all usages of a specific function or macro.

**diffoscope.** A more elaborate diffing tool that handles many file formats intelligently. Useful when you want to compare compiled objects, images, or other non-text artifacts.

**The FreeBSD source-code search at <https://cgит.freebsd.org/>.** A web-based interface to the Git repository that lets you browse the tree, view commits, and search for identifiers. Often faster for casual browsing than a local clone.

**Bugzilla's search interface.** Useful for finding whether a specific issue has been reported, and often for finding the commit that fixed it.

Becoming fluent with these tools is a matter of a few hours of practice. Once they are in your fingers, you can answer questions about the tree's history that would otherwise take days of manual reading.

### 38.9.5 6.5 A Monthly Rhythm for Staying Current

Readers often ask how to make “staying current” a sustainable habit rather than a vague good intention. Here is a monthly rhythm that works for many developers.

**Weekly.** Read the `freebsd-hackers` or `freebsd-drivers` mailing list digest (or equivalent summary) once a week. Read one or two threads that catch your interest. Reply to one if you have something to contribute. This is an hour of work per week.

**Monthly.** Pull the latest FreeBSD source tree. Run your personal driver's test suite against it. Investigate any failures. Read the most recent conference talk you have not yet watched. This is an afternoon per month.

**Quarterly.** Skim the commit log of the subsystem your driver touches since the last time you looked. Check whether any of the changes affect your driver. Read the most recent committer summary or project update. Review your personal learning goals and adjust them if the landscape has shifted. This is a day per quarter.

**Annually.** Read the release notes for the latest FreeBSD release, top to bottom. Consider whether you should upgrade your development environment. Attend or watch at least one conference worth of talks. Review your driver portfolio and consider whether any of the older drivers need maintenance work.

This rhythm is not a rigid schedule. It is an illustration of how regular, low-intensity attention can keep you current without becoming a second job. Most developers who stay engaged long-term follow something like this pattern, sometimes denser and sometimes looser.

### 38.9.6 6.6 Exercise: Subscribe and Read

This subsection's exercise is a small commitment. Subscribe to `freebsd-hackers@` or `freebsd-drivers@`, whichever seems more appropriate for your interests. Commit to reading one thread per week for four weeks. At the end of the four weeks, decide whether the subscription is worth keeping.

The point of the exercise is not to learn a specific technical thing. It is to build the habit of being aware of what the project is discussing. If after four weeks you find the traffic unhelpful, unsubscribe. If you find it useful, keep reading. The experiment is small and reversible.

A few practical tips. Filter the mailing-list messages into a separate folder in your mail client, so they do not pollute your regular inbox. Read in batches rather than as individual messages arrive. Be willing to ignore threads that are not about topics you care about. The goal is sustainable attention, not exhaustive coverage.

### 38.9.7 6.7 Watching for Deprecation Notices

A particular skill worth cultivating is watching for deprecation notices. Deprecation is how the project signals that a specific API, subsystem, or behaviour is going to change or be removed in a future release. Developers

who miss deprecation notices find out about the removal later, when their code no longer builds, and the fix is often more painful at that point than it would have been during the deprecation period.

Deprecation notices appear in several places. UPDATING is the primary one. Release notes mention them. Commit messages for deprecation changes often contain the word “deprecate” (with the ambiguous spelling to catch both “deprecate” and “deprecated”). Mailing list discussions often precede deprecation changes and give early warning.

A practical habit is to grep the recent commit log for deprecation keywords once a month:

```
$ git log --oneline --since=1.month --grep=deprecate
```

The output is usually short and scannable, and it catches most deprecations before they become problems.

### 38.9.8 6.8 Engaging When You See a Change That Affects You

When you spot a change that affects code you maintain, you have several options. You can adapt your code immediately, rebuilding and testing against the new behaviour. You can comment on the commit or the associated review, asking about the migration path. You can engage with the author on the relevant mailing list. You can file a PR in Bugzilla if you encounter a problem you think needs tracking.

The engagement itself is valuable beyond the specific change. Each time you reach out about a change, you are both contributing to the project’s awareness of downstream effects and building a relationship with the author. Over time, those relationships are one of the most valuable aspects of being part of the community.

One specific pattern worth highlighting. If a change is about to break your driver, and you find the change while it is still under review, commenting on the review is dramatically more valuable than commenting after the change is committed. Reviewers actively want to know about downstream effects, and a well-formed comment at review time often leads to adjustments that make the change less disruptive. Waiting until after commit means the change lands, downstream users discover the breakage, and the fix becomes a second patch that somebody else has to coordinate.

### 38.9.9 6.9 A Curated Reading List for Continued Study

The single most common request from readers who finish a book like this one is “what should I read next?” The honest answer is that the best reading depends on where you want to go. A reader headed toward filesystem work will have a very different list from one headed toward network drivers or toward embedded development. What follows is a curated starting list, organised by area, with one or two recommendations in each direction. The list is deliberately short. A long list would overwhelm. A short list lets you finish what you start.

For general FreeBSD kernel background, the most useful single source is still the FreeBSD source tree itself. Read `/usr/src/sys/kern/kern_synch.c` for an example of careful kernel engineering, read `/usr/src/sys/kern/vfs_subr.c` for a sense of how a large subsystem organises its internal interfaces, and read `/usr/src/sys/dev/null/null.c` once more at the end of the book to see how much richer the file reads now than it did in Chapter 1. The manual pages in section 9 remain the most authoritative reference for kernel interfaces; skim the list of section-9 pages with `apropos -s 9` and read anything that catches your eye.

For filesystem work, the classic reference is Kirk McKusick's "The Design and Implementation of the FreeBSD Operating System." Pay attention to the VFS chapters. Then read `/usr/src/sys/ufs/ffs/` carefully, choosing one file at a time and tracing how its functions are called from the layers above and below. If you prefer talks to books, the BSDCan archive contains several filesystem-focused talks from Kirk McKusick, Chuck Silvers, and others; watch them in chronological order to see how the system evolved.

For networking, start with the `netmap(4)` manual page and the sources under `/usr/src/sys/dev/netmap/`, then widen out to the stack proper under `/usr/src/sys/net/` and `/usr/src/sys/netinet/`. Read `if.c` first, then `if_ether.subr.c`, then pick a specific protocol family and follow its packets through. The `iflib(9)` manual page is essential for modern Ethernet drivers. For deeper coverage, the TCP/IP Illustrated series by Stevens remains the canonical reference; chapters 2 and 3 translate almost directly to FreeBSD's implementation.

For embedded and ARM work, read through the Device Tree sources under `/usr/src/sys/contrib/device-tree/src/` for your board, then study `/usr/src/sys/dev/fdt/` to see how FreeBSD consumes those trees. Warner Losh's conference talks on arm64 and RISC-V support are excellent complements. The `fdt(4)` manual page is short but dense; reread it after a month's practice.

For debugging and profiling, read the `kgdb(1)`, `ddb(4)`, `dtrace(1)`, and `hwpmc(4)` manual pages in sequence. Then pick a DTrace provider (io, vfs, sched) and write one meaningful script using it. Brendan Gregg's DTrace book remains relevant; most of its examples still apply to FreeBSD directly.

For security and hardening, read the `capsicum(4)`, `mac(4)`, and `jail(8)` manual pages carefully. The Capsicum papers from Watson and others are worth reading in full. `/usr/src/sys/kern/sys_capability.c` and `/usr/src/sys/kern/subr_capability.c` are the implementation worth tracing. The FreeBSD security advisory archive on the project website is a useful record of how real bugs have played out.

For USB work, start with the USB controller drivers in `/usr/src/sys/dev/usb/controller/` and the core files under `/usr/src/sys/dev/usb/` such as `usb_process.c` and `usb_request.c`. The USB specification is large but accessible; read only the chapters you need when you need them, and use the existing FreeBSD drivers as worked examples.

For virtualisation and bhyve, the bhyve manual page is the starting point, followed by the bhyve source under `/usr/src/usr.sbin/bhyve/`. John Baldwin's BSDCan talks provide excellent context. If you plan to use bhyve as a test harness for driver development, focus on the PCI passthrough and virtio sections.

For general craft, three books are worth owning: Kernighan and Ritchie's "The C Programming Language" for the language itself, Kernighan and Pike's "The Practice of Programming" for the habits, and McKusick and Neville-Neil's "The Design and Implementation of the FreeBSD Operating System" for the system. If you already know C well, the first book can be skimmed. If you have ever felt that your code "works but could be better," the second book will address that feeling directly.

For community culture and history, read the FreeBSD Handbook's introduction and then the archives of the FreeBSD project on the Internet Archive. The project has existed for decades and its culture is documented in its correspondence as much as in its code. Understanding how the project thinks about itself will help you contribute in a way the community welcomes.

One final recommendation: take the list above and mark, with a pencil, the one item in each area that you are most likely to actually complete in the next three months. Then close this book and start on those items, not all of them, just the marked ones. A small finished list beats a long unfinished list every time.



### 38.9.10 6.10 Wrapping Up Section 6

This section has walked through how to stay connected to the FreeBSD kernel’s ongoing development: where to watch for changes, which conferences to follow, how to track API drift, and what habits turn “staying current” from a vague aspiration into a sustainable practice.

The overarching idea is that staying current is not a task you do once. It is a rhythm you develop. The rhythm does not need to be intense; it needs to be regular. A weekly mailing-list glance, a monthly build test, a quarterly review of your code, and an annual walk through the release notes are together enough to keep most driver maintainers in sync with the project.

We have now covered the six main sections of the chapter: celebrating what you have accomplished, understanding where you stand, exploring advanced topics, building your toolkit, giving back to the community, and staying current. The remaining material of the chapter is hands-on: labs to apply what you have read, challenges to push your practice further, and planning artifacts you can keep as a record of the reflection you did here.

## 38.10 7. Hands-On Reflection Labs

The labs in this chapter are reflection labs rather than coding labs. They ask you to apply what the chapter has discussed to your own circumstances, using the companion files as templates where helpful. Each lab produces a concrete artifact you can keep, and the artifacts together form a record of where you were at the moment you finished this book.

Treat the labs as time well spent. A reader who does the labs will leave the book with a set of personal documents that shape the next few months of their work. A reader who skips the labs will have read the chapter but not applied it, and the difference shows up three months later when the one reader has a plan and the other is drifting.

### 38.10.1 7.1 Lab 1: Complete a Self-Assessment Worksheet

**Goal.** Produce a written self-assessment that captures where you stand at the end of the book.

**Time.** Two to three hours, spread across one or two sittings.

**Materials.** The `self-assessment.md` template in `examples/part-07/ch38-final-thoughts/`. A notebook or text editor. Access to the book’s chapters as a reference.

**Steps.**

1. Copy the self-assessment template into a directory you control, such as a personal Git repository or a folder in your home directory.
2. Date the file. Use ISO 8601 format (YYYY-MM-DD) in the filename or in the header, so you can sort multiple assessments chronologically when you redo the exercise later.
3. For each topic in the template, give yourself a confidence rating on a one-to-five scale. The topics are drawn from the book’s coverage: C for kernels, debugging and profiling, device tree integration, reverse engineering, userland interfaces, concurrency and synchronisation, DMA and interrupts, upstream submission, and the specific core APIs like `bus_space(9)`, `sysctl(9)`, `dtrace(1)`, `devfs(5)`, `poll(2)`, and `kqueue(2)`.

4. For each topic, write one sentence explaining why you chose that rating. A rating without a reason is just a number. A rating with a reason is a diagnosis.
5. At the end, identify the three topics where you most want to invest more practice. Explain why for each.
6. Save the file. Back it up. Put a calendar reminder for six months from now to redo the assessment and compare.

**Deliverable.** The completed self-assessment worksheet, saved with the current date.

**What to notice.** The act of forcing a specific confidence rating on each topic will reveal surprising distinctions. You may find that you rate yourself higher than you expected on some topics and lower on others. The surprises are where the diagnostic value lives.

### 38.10.2 7.2 Lab 2: Draft a Personal Learning Roadmap

**Goal.** Produce a written plan for the next three to six months of your FreeBSD learning.

**Time.** Two hours, in one focused sitting.

**Materials.** The `learning-roadmap.md` template in `examples/part-07/ch38-final-thoughts/`. The output of Lab 1. The advanced-topics section of this chapter.

**Steps.**

1. Copy the learning-roadmap template into your working directory.
2. Based on your self-assessment from Lab 1, choose the one technical area you most want to deepen over the next three months. The choice should be specific: not “the network stack,” but “the `ifnet` layer and its interaction with `iflib`.”
3. Break the three-month goal into monthly milestones. Each milestone should be concrete enough to be recognisable when it is done. Examples: “By end of month one, I have read `/usr/src/sys/net/if.c` in full and can explain the lifecycle of an `ifnet`.” “By end of month two, I have written a minimal pseudo-interface that implements the `ifnet` API.” “By end of month three, I have submitted the pseudo-interface for review.”
4. Identify the resources you will use. Manual pages. Specific source files. Specific conference talks. Specific people whose code you will read. Be concrete enough that you can start immediately, without additional research.
5. Identify the practice cadence. Will you work on this every day for an hour? Every weekend for a few hours? Twice a week? The cadence matters more than the intensity, and a sustainable cadence is better than an ambitious one.
6. Add secondary goals. A toolkit improvement. A community engagement goal (join a mailing list, answer N questions on the forums). A currency goal (watch M conference talks, follow commit logs for a specific subsystem).
7. Commit the roadmap to a Git repository or save it somewhere stable. Put a calendar reminder at the end of each month to review progress and adjust.

**Deliverable.** The completed learning roadmap, saved with the current date.

**What to notice.** A good roadmap feels slightly uncomfortable. If it feels easy and safe, it is probably not ambitious enough. If it feels impossible, it is probably not sustainable. The sweet spot is a roadmap that is visibly more than you can do in a weekend but visibly within reach over three months of consistent practice.

### 38.10.3 7.3 Lab 3: Set Up Your Driver Template

**Goal.** Produce a personal driver template that reflects your style choices.

**Time.** Two to four hours, in one or two sittings.

**Materials.** The `template-driver/` directory in `examples/part-07/ch38-final-thoughts/`. The book's chapters on Newbus, `bsd.kmod.mk`, and driver anatomy as reference. Your text editor.

**Steps.**

1. Copy the `template-driver` directory into a personal Git repository. Rename the files to something generic, such as `skeleton.c`, `skeleton.h`, `skeletonreg.h`, and adjust the Makefile accordingly.
2. Adjust the copyright header to match your name, email, and preferred license.
3. Adjust the include ordering to match your preferences, while still conforming to `style(9)`.
4. Add any standard patterns you reach for reflexively: a debug `sysctl`, a locking macro, a module event handler skeleton, a probe-attach-detach pattern you always use.
5. Remove any patterns you do not use. The goal is a minimal template, not a comprehensive one.
6. Write a short README explaining what the template is for and how to start a new driver from it. A paragraph or two is enough.
7. Commit the template to your Git repository. Tag the commit with a version number such as `template-v1.0`.

**Deliverable.** A personal, versioned driver template in a Git repository you control.

**What to notice.** The first version of your template will feel over- or under-specified in particular ways. That is normal. The template will evolve over the next several projects as you notice patterns you always add or patterns you always remove. Version it. Let it grow.

### 38.10.4 7.4 Lab 4: Subscribe to a Mailing List

**Goal.** Establish a connection to the FreeBSD community through a specific mailing list.

**Time.** Twenty minutes to subscribe; one hour per week for four weeks to read.

**Materials.** A working email account. Internet access.

**Steps.**

1. Choose a mailing list. For driver-focused readers, `freebsd-drivers@` is a natural choice. For readers who want a broader view of kernel development, `freebsd-hackers@` is appropriate. For readers who want to track the current branch specifically, `freebsd-current@` is the list.
2. Visit the project's mailing-list page at <https://lists.freebsd.org/> and follow the subscription instructions for the list you chose.

3. Set up a mail filter that moves the list's messages to a dedicated folder, so they do not overwhelm your inbox.
4. For each of the next four weeks, block out an hour to read the list. Read at least one full thread per week. Take notes on any threads that touch areas you care about or surprise you.
5. If you have something to contribute to a thread (a correction, a data point, an answer to a question), reply. If not, just read. Lurking is fine.
6. At the end of four weeks, decide whether to keep the subscription. If the list is not useful for your purposes, unsubscribe. If it is useful, keep it and let it become part of your routine.

**Deliverable.** A record of the subscription, a mail filter set up, and four weeks of at least one thread read per week.

**What to notice.** Mailing lists are a slow medium. Nothing happens quickly, and it can take several weeks for the value to become clear. Give the exercise the full four weeks before judging it.

### 38.10.5 7.5 Lab 5: Review a Manual Page You Have Not Read

**Goal.** Deepen your familiarity with FreeBSD's reference documentation by reading a manual page you have not yet read.

**Time.** One hour.

**Materials.** Access to the FreeBSD manual pages, either through `man(1)` on a FreeBSD system or through the web at <https://man.freebsd.org/>. A notebook or text editor.

**Steps.**

1. Pick a manual page you have not read. It should be relevant to areas you might want to study further. Good candidates: `vnode(9)`, `crypto(9)`, `iflib(9)`, `audit(4)`, `mac(9)`, `kproc(9)`.
2. Read the manual page in full. Expect it to take twenty to forty minutes if the page is substantive.
3. After reading, close the manual page. On a blank piece of paper or in a text file, write a one-paragraph summary in your own words. Cover: what the interface is for, what its main data structures and functions are, and what a hypothetical driver would use it for.
4. Reopen the manual page and compare your paragraph to the reference. Where is your paragraph vague or inaccurate? Revise it.
5. Save the paragraph in your personal knowledge directory, labelled by the manual page name and the date.

**Deliverable.** A written paragraph summarising a manual page you have newly read, saved for future reference.

**What to notice.** Writing your own summary forces understanding at a level that passive reading does not. A paragraph you can write well is a topic you understand. A paragraph you struggle with is a topic that needs more study. The paragraphs accumulate over time into a personal glossary of the kernel.

### 38.10.6 7.6 Lab 6: Build a Regression Test for a Driver You Wrote

**Goal.** Produce a regression test suite for one of your drivers from the book.

**Time.** Three to four hours.

**Materials.** A driver you wrote during the book. The `regression-test.sh` skeleton in `examples/part-07/ch38-final-thoughts/scripts/`. A FreeBSD development VM.

**Steps.**

1. Pick a driver. The async I/O driver from Chapter 35 is a good candidate because it has enough complexity to make regression testing worthwhile.
2. Copy the regression-test skeleton into a tests directory alongside the driver.
3. Identify three behaviours that the driver should reliably exhibit. Example for the async driver: “loads without warnings,” “accepts a write and allows it to be read back,” “unloads cleanly after being loaded and exercised.”
4. Translate each behaviour into an automated check. A check is a shell command that produces a predictable output or exit code if the behaviour holds, and an unpredictable one if it does not. Use `kldload`, `kldunload`, `dd`, `sysctl`, and similar tools as building blocks.
5. Wire the checks together in the script. A good script runs each check, reports the result, and exits with a non-zero status if any check failed.
6. Run the script. Verify it passes against the current driver. Verify it fails when you deliberately break the driver (add a line that aborts in the module init, for example) and then remove the deliberate break.
7. Commit the script alongside the driver in your Git repository.

**Deliverable.** A working regression-test script for a driver you wrote, committed to your repository.

**What to notice.** Regression tests are often harder to write than the code they test, especially for kernel modules. The difficulty is exactly why they are valuable: they force you to articulate what “correct behaviour” means in a way that can be checked automatically. A driver without regression tests is a driver whose correctness claims are verbal; a driver with regression tests has those claims encoded in something reproducible.

### 38.10.7 7.7 Lab 7: File or Triage a Bugzilla PR

**Goal.** Interact with the FreeBSD Bugzilla in a concrete, small way.

**Time.** One to two hours.

**Materials.** A FreeBSD Bugzilla account (free to create). Your FreeBSD 14.3 development system.

**Steps.**

1. Create a Bugzilla account at <https://bugs.freebsd.org/bugzilla/> if you do not have one.
2. Option A (triage path): Browse the open PRs in the `kern` product or a driver component you care about. Find a PR that has not been updated in at least six months and whose status is unclear. Attempt to reproduce the issue on your system. Add a comment to the PR with your findings: what you tried, what you observed, whether you can reproduce the issue.

3. Option B (reporting path): If you have encountered a bug in a driver or in the kernel during the book, and you have not yet reported it, file a PR. Include the FreeBSD version, the hardware, clear reproduction steps, and any relevant output from `dmesg`, `uname -a`, or the driver's own diagnostics.
4. Whatever option you choose, write the PR or comment in the clear, specific style that Bugzilla interactions reward. Avoid vague statements. Be specific. Include details someone else could act on.
5. Save the URL of the PR or comment. Add it to your contribution log if you keep one.

**Deliverable.** A PR you filed or a comment you added on an existing PR, in the FreeBSD Bugzilla.

**What to notice.** Bugzilla interactions have a specific style that is different from mailing-list posts and different from code-review comments. Clarity, reproducibility, and specificity are the main virtues. A well-formed PR gets attention; a poorly-formed one does not.

### 38.10.8 7.8 Lab 8: Set Up Your Virtual Lab

**Goal.** Produce a reproducible virtual lab you can boot quickly for future driver work.

**Time.** Half a day.

**Materials.** A host system with bhyve (on FreeBSD) or QEMU (on any host). A FreeBSD 14.3 installation image.

**Steps.**

1. Create a working directory for your lab. Initialise a Git repository in it for the scripts and configuration files.
2. Write a shell script that creates a disk image of at least 20 GB. Use `truncate(1)` or an equivalent to create a sparse file.
3. Write a shell script that runs the chosen hypervisor with the appropriate command-line arguments to boot from the FreeBSD installation image and install FreeBSD into the disk image. Document every flag.
4. After installation, take a snapshot of the disk image (copy it, or use the hypervisor's snapshot mechanism). Label the snapshot with the FreeBSD version and the date.
5. Write a shell script that boots the installed system. It should use a serial console, mount a shared directory for source code if your hypervisor supports it, and configure networking so that you can SSH into the guest.
6. Boot the guest. SSH into it. Verify you can build a trivial kernel module (a “hello world” module is fine) in the guest.
7. Commit all the scripts and the documentation to your Git repository. Tag the commit with the FreeBSD version and the date.

**Deliverable.** A Git repository containing your lab's configuration, along with a known-good snapshot of an installed system.

**What to notice.** The lab setup is a one-time investment that pays off on every subsequent project. If the lab takes ten minutes to boot, you will reach for it only for serious work. If it takes thirty seconds to boot, you will reach for it constantly, and the pace of your driver work will be much faster.

### 38.10.9 7.9 Wrapping Up the Reflection Labs

These eight labs have walked you through producing a set of personal artifacts: a self-assessment, a learning roadmap, a driver template, a mailing-list subscription, a manual-page summary, a regression-test script, a Bugzilla interaction, and a virtual lab. Together they constitute a snapshot of your current state and a starting position for the next phase of your work.

None of the artifacts is especially elaborate. Each one could be produced by someone who had finished this book. But together, they represent a surprisingly large amount of infrastructure that supports ongoing FreeBSD driver work. Without them, you start from zero every time. With them, you start from ground already cleared.

Keep the artifacts. Revisit them. Evolve them. Over the course of the next year, they will shape how your work proceeds in ways that are easier to recognise in retrospect than in prospect.

## 38.11 8. Challenge Exercises

The challenges in this section are longer, more open-ended than the labs, and designed for readers who want to go further than the main chapter material. There is no single correct way to complete any of them, and you may find that your approach differs substantially from another reader's. That is fine. The challenges are invitations, not puzzles.

### 38.11.1 8.1 Challenge 1: Write a Real Driver for Real Hardware

**Description.** Pick a piece of hardware you own, verify that FreeBSD does not already support it adequately, and write a driver for it. Go through the full cycle: acquisition, investigation, implementation, documentation, testing, and consideration for upstream submission.

**Scope.** This is a multi-week or multi-month project. A reasonable target is a relatively simple device: a USB serial adapter using an unsupported chipset, a PCI card for a niche application, a GPIO-connected sensor. Avoid ambitious targets for a first independent project; graphics cards and wireless chipsets are notoriously hard.

**Milestones.**

1. Identify the hardware. Confirm that FreeBSD does not support it, or supports it incompletely.
2. Gather documentation. Vendor datasheets, Linux or NetBSD drivers, chipset specs, whatever is available.
3. Set up a development environment. Configure your virtual lab to talk to the real hardware if your hypervisor supports passthrough, or plan to build and test on a physical FreeBSD system.
4. Write an initial probe and attach sequence. Confirm that FreeBSD can at least identify and attach to the hardware.
5. Implement the main driver functionality, one feature at a time, with tests for each.
6. Write a manual page.
7. Consider submission. Follow the Chapter 37 workflow if you decide to submit.

**What you will learn.** Independent driver development at a level the book's exercises did not require. The experience of a real, non-trivial project with its own friction and its own surprises.

### 38.11.2 8.2 Challenge 2: Fix a Real Bug in a Real Driver

**Description.** Find an open bug in a driver in the FreeBSD tree, reproduce it, understand it, fix it, and submit the fix upstream.

**Scope.** A couple of weekends, depending on the bug. Some bugs are one-line fixes; others require substantial investigation.

**Milestones.**

1. Browse the FreeBSD Bugzilla for open driver bugs. Filter by “kern” or by specific drivers. Look for bugs that have clear reproduction steps and are not obviously abandoned.
2. Pick a bug that matches your skill level. A good first bug is one where the reporter has already done some investigation and where the fix is likely to be small.
3. Reproduce the bug on your system. This step is essential and often takes longer than expected.
4. Investigate. Read the relevant driver source. Add diagnostic printouts or KTR traces if needed. Form a hypothesis about the cause.
5. Write a fix. Test it. Verify that the bug is resolved.
6. Check that you have not introduced new bugs.
7. Submit the fix, following the Chapter 37 workflow.

**What you will learn.** Reading and understanding code written by other people, working within existing conventions, and experiencing the full upstream workflow with a real, accepted contribution.

### 38.11.3 8.3 Challenge 3: Port a Linux or NetBSD Driver to FreeBSD

**Description.** Pick a driver that exists in Linux or NetBSD but not in FreeBSD (or is incomplete in FreeBSD). Study the existing driver, understand the hardware interface, and write an equivalent FreeBSD driver.

**Scope.** A serious multi-month project. Only attempt this if you have a specific piece of hardware in mind and enough interest to sustain the work.

**Milestones.**

1. Choose the target hardware and the source driver. Verify the source driver’s license is compatible with your intended license for the FreeBSD driver.
2. Read the source driver in full. Understand what it does at the interface level: how it probes, how it handles interrupts, what data structures it uses, how it handles DMA.
3. Design your FreeBSD driver architecture. It will not be a line-by-line port. FreeBSD’s conventions differ from Linux’s, and a well-ported driver respects FreeBSD’s conventions.
4. Write the driver, using the source driver as a reference for what the hardware does but writing the code from scratch in FreeBSD style.
5. Test against the real hardware.
6. Document the driver and submit.



**What you will learn.** Cross-system driver translation is a serious craft. You will learn FreeBSD idioms more deeply by contrasting them with Linux or NetBSD idioms. You will also learn the discipline of writing fresh code from an understanding of a specification rather than from cut-and-paste.

#### 38.11.4 8.4 Challenge 4: Write a Deep Technical Blog Post

**Description.** Write a thorough, carefully researched blog post about a FreeBSD kernel topic you want to understand better. The goal is not to produce a perfect post for a large audience; it is to use the act of writing to deepen your own understanding.

**Scope.** A weekend or two. Longer if you have a particularly complex topic.

**Milestones.**

1. Pick a topic. It should be something you partly understand but want to understand better. Good examples: “How FreeBSD’s epoch interface works,” “The lifecycle of an interrupt in FreeBSD,” “How `iflib` abstracts hardware for network drivers.”
2. Read everything you can find on the topic. Manual pages, source code, previous blog posts, conference talks.
3. Draft the post. Aim for three to five thousand words.
4. Revise. Cut the parts that are wrong or unclear. Sharpen the parts that are strong.
5. Ask a friend or colleague to read the draft. If they are a FreeBSD developer, all the better. If they are not, their questions will reveal gaps in your explanation.
6. Publish the post. Your own blog, the FreeBSD forum, or dev.to are all reasonable venues.

**What you will learn.** Teaching is one of the deepest forms of learning. The act of writing a careful explanation forces you to confront everything you do not yet understand, and resolving those gaps is how understanding grows.

#### 38.11.5 8.5 Challenge 5: Become a Reviewer

**Description.** Pick a specific area of the FreeBSD tree (driver-related for readers of this book) and commit to reviewing every Phabricator review that touches that area for one month.

**Scope.** Depending on the area’s activity, this could be a few reviews a week or several per day. A month of regular reviewing will give you experience at the craft of review without overwhelming other obligations.

**Milestones.**

1. Choose an area. A specific driver or subsystem is ideal.
2. Set up notifications for Phabricator reviews in that area. The Phabricator interface supports this.
3. For each review, read the proposed change carefully. Understand what it is trying to do. Test it locally if the change is small enough.
4. Post your review comments. Be specific. Be constructive. Ask questions when you are uncertain rather than asserting things you are not sure of.

5. Respond to replies. Engage in the review discussion until the review is resolved.
6. At the end of the month, reflect on what you learned.

**What you will learn.** Code review is a different skill from code writing. It requires reading carefully, understanding intent, and articulating suggestions productively. Developers who review well are among the most valuable members of any project, and the skill is one you can develop with practice.

### 38.11.6 8.6 Challenge 6: Run a Study Group

**Description.** Organise a small group of peers to work through a challenging topic together. The topic should be something none of you feel confident about individually, where working together will produce a shared understanding faster than working alone.

**Scope.** A group of three to six people, meeting weekly for two to three months.

**Milestones.**

1. Find the peers. They can be former colleagues, members of a local BSD user group, participants in an online forum. Three to six is the right size.
2. Choose a topic together. The VFS layer, `iflib`, the ACPI subsystem, or `netgraph` are all reasonable candidates.
3. Agree on a cadence and a format. A weekly video call with a shared reading list is a standard format. Each member prepares a section each week and presents it.
4. Meet regularly. Take notes. Share what you learn between meetings.
5. Produce a group artifact at the end. A shared notes document, a blog post summarising the study, or a presentation to a local user group.

**What you will learn.** Collaborative study is among the most effective ways to learn complex material. You will also learn how to organise and sustain a group, which is a leadership skill in its own right.

### 38.11.7 8.7 Challenge 7: Contribute a Non-Code Artifact

**Description.** Contribute something to the FreeBSD Project that is not code and not driver-specific. A documentation patch, a translation, a manual-page improvement, a test-case addition, a diagram for the handbook, whatever has a clear value.

**Scope.** A weekend.

**Milestones.**

1. Find a non-code artifact that needs improvement. Outdated manual pages, incomplete handbook sections, missing translations, or gaps in the test suite are common candidates.
2. Improve it. Write the new version carefully. Test it if it is testable.
3. Submit it through the appropriate workflow. For documentation, this means the documentation Git repository. For manual pages, the main source tree. For translations, the translation system.
4. Respond to review feedback until the contribution is accepted.

**What you will learn.** Non-code contributions have their own submission flows and their own community expectations. Going through one end-to-end broadens your sense of what it means to contribute.

### 38.11.8 8.8 Challenge 8: Sustain a Personal Practice for Six Months

**Description.** The most difficult challenge in this chapter is not technical. It is the challenge of sustaining your practice after the book is closed. Many readers finish technical books with enthusiasm and then let the momentum fade within weeks. This challenge is to resist that pattern deliberately, by committing to a specific, sustained practice for six months.

**Scope.** Commit to a small, regular practice. Not a heroic one. Examples of a sustainable cadence: two hours of kernel work every Saturday morning, or one hour every weekday evening, or every second weekend dedicated to a single driver project. Whatever you choose, write it down and stick to it.

**Milestones.**

1. Define the practice in a single sentence you can quote from memory. If you cannot state it cleanly, it is not specific enough.
2. Name a concrete project that will run through the full six months. The project must be small enough to complete but meaningful enough to sustain your interest.
3. Track your sessions in a plain log. Date, duration, one sentence on what you did. The log's existence matters more than its length.
4. At month three, review the log honestly. If you are behind, adjust the cadence downward rather than giving up. A smaller sustained practice is worth more than a larger abandoned one.
5. At month six, write a short reflection on what you produced, what you learned, and what you will do next.

**What you will learn.** Sustaining a practice is itself a skill, and it compounds across every area of a long career. Readers who learn to sustain their own practice in the months after this book will, in the long run, go much further than readers whose enthusiasm was bright but brief. This challenge exists specifically to help you build that habit on the other side of the book's last page.

### 38.11.9 8.9 Wrapping Up the Challenges

The challenges above range from small to substantial. None of them is required. A reader who completes one will have deepened their practice in a specific direction. A reader who completes several will be approaching the level of a working junior FreeBSD developer. A reader who completes all of them will be approaching the level at which commit rights become a meaningful possibility.

Pick the one or two that align with your interests and your available time. Execute them carefully. Let them inform the next iteration of your learning roadmap.

## 38.12 9. Personal Planning and Checklists

This section presents a few checklists and planning artifacts that you can use as templates. They are deliberately short and tactical, not elaborate frameworks. Each of them answers a specific practical question.

### 38.12.1 9.1 A Contribution Checklist

Use this checklist the first time you send a patch to the FreeBSD Project, and every time after that.

- ☐ My patch addresses a specific, well-defined issue or adds a specific, well-defined feature.
- ☐ I have verified that the patch applies cleanly to the current tip of the relevant branch.
- ☐ I have built the patch on amd64 and verified it compiles without warnings.
- ☐ I have built the patch on at least one other architecture (typically arm64) and verified it compiles without warnings.
- ☐ I have run the pre-submission test script from Chapter 37 and all steps pass.
- ☐ I have written or updated the relevant manual page.
- ☐ I have run `mandoc -Tlint` on the manual page and it is silent.
- ☐ I have written a commit message that explains what the change does and why.
- ☐ I have verified that no proprietary information, internal company URLs, or private correspondence appear in the patch.
- ☐ I have identified the reviewer(s) I will request.
- ☐ I have chosen the appropriate submission channel: Phabricator for structured review, GitHub or mailing list otherwise.
- ☐ I have a plan for responding to review feedback within a reasonable time.
- ☐ I have subscribed to notifications for the review so I will see comments quickly.
- ☐ I understand that merging the patch is a beginning, not an end, and I am prepared to maintain the code.

### 38.12.2 9.2 A Stay-Current Checklist

Use this checklist on a monthly cadence to keep yourself current with FreeBSD development.

- ☐ I have pulled the latest FreeBSD source tree.
- ☐ I have built my personal drivers against the latest source and investigated any warnings or errors.
- ☐ I have run my regression tests against the latest source and investigated any failures.
- ☐ I have read the commit log for the subsystems I care about since the last time I looked.
- ☐ I have checked UPDATING for new notices.
- ☐ I have read at least one thread on `freebsd-hackers@` or `freebsd-drivers@` since the last check.
- ☐ I have watched at least one conference talk or read at least one blog post about FreeBSD development.
- ☐ I have thought about whether any of the changes I saw affect my ongoing work.
- ☐ I have noted any TODO items for next month's review.

### 38.12.3 9.3 A Self-Assessment Worksheet

Use this worksheet every three to six months to track your growth.

**Date:** \_\_\_\_\_

**FreeBSD version last built:** \_\_\_\_\_

**Confidence rating (1-5) and one-sentence rationale:**

- C for kernel programming: \_\_\_\_\_
- Kernel debugging and profiling: \_\_\_\_\_

- Device tree integration: \_\_\_\_\_
- Reverse engineering and embedded development: \_\_\_\_\_
- Userland interfaces (ioctl, sysctl, poll, kqueue): \_\_\_\_\_
- Concurrency and synchronisation: \_\_\_\_\_
- DMA, interrupts, and hardware interaction: \_\_\_\_\_
- Upstream submission and review readiness: \_\_\_\_\_
- bus\_space(9): \_\_\_\_\_
- sysctl(9): \_\_\_\_\_
- dtrace(1): \_\_\_\_\_
- devfs(5): \_\_\_\_\_
- Newbus (device\_t, DEVMETHOD, DRIVER\_MODULE): \_\_\_\_\_
- Locking (mutexes, sx, rmlocks, condition variables): \_\_\_\_\_

**Three topics I most want to deepen in the next six months:**

1. \_\_\_\_\_
2. \_\_\_\_\_
3. \_\_\_\_\_

**One concrete project I will start this month:**

\_\_\_\_\_

**One community action I will take this month:**

\_\_\_\_\_

### 38.12.4 9.4 A Learning Roadmap Template

Use this template to plan each three-to-six-month learning cycle.

**Period:** \_\_\_\_\_ to \_\_\_\_\_

**Primary focus area:** \_\_\_\_\_

**Why this area, now:** \_\_\_\_\_

**Monthly milestones:**

- Month 1: \_\_\_\_\_
- Month 2: \_\_\_\_\_
- Month 3: \_\_\_\_\_

**Resources I will use:**

- Manual pages: \_\_\_\_\_
- Source files: \_\_\_\_\_
- Conference talks: \_\_\_\_\_
- Books and papers: \_\_\_\_\_

- People I will learn from: \_\_\_\_\_

**Practice cadence:**

- Frequency: \_\_\_\_\_
- Duration per session: \_\_\_\_\_

**Secondary goals:**

- Toolkit improvement: \_\_\_\_\_
- Community engagement: \_\_\_\_\_
- Currency: \_\_\_\_\_

**Review date:** \_\_\_\_\_

### 38.12.5 9.5 A Code Review Checklist

Sooner or later, whether you are reviewing another developer's patch or reviewing your own code before submitting it, you will benefit from a consistent mental pass over the same set of concerns. Use this checklist when you sit down to review a change. It applies equally to your own work and to work you review for others.

- ☐ I understand what the change is trying to accomplish and can restate its purpose in one sentence.
- ☐ The change is scoped appropriately: not too large to review carefully, not so small that the context is missing.
- ☐ The commit message explains the "why" clearly and will still make sense in five years.
- ☐ The change does not mix unrelated concerns (cleanup plus feature, refactor plus bug fix) without a strong reason.
- ☐ Every function added or modified has a clear responsibility and a clear contract.
- ☐ Error paths are handled, not silently swallowed.
- ☐ Resources allocated in `attach()` are released in `detach()` in reverse order.
- ☐ Any new locks are named clearly, their scope is explicit, and their lock order is consistent with existing code.
- ☐ The change does not introduce potential use-after-free, leaked references, or double-free paths.
- ☐ The change passes `checkstyle9.pl` without warnings.
- ☐ The change compiles on at least two architectures with no new warnings.
- ☐ Any manual pages or documentation affected by the change have been updated.
- ☐ Any `sysctl` or `dev.` names introduced are named consistently with existing conventions.
- ☐ The change behaves correctly under load and unload cycles.
- ☐ The change does not break any in-tree test.
- ☐ The change has a test case or a clear reason why no test is possible.

When reviewing other people's code, add one more item: have I been respectful, specific, and helpful, rather than abrupt or dismissive? Code review is one of the most visible forms of community participation, and the tone matters as much as the substance.

### 38.12.6 9.6 A Debugging Journal Entry Template

When you run into a hard bug, write it down before, during, and after the debugging session. The journal serves two purposes: it helps you think clearly while the bug is fresh, and it gives you something to return to the next time you hit a similar problem. Over time, these entries become a personal reference that saves you hours.

Use this template for each bug you spend more than a single short session on.

**Date:** \_\_\_\_\_

**Module or driver:** \_\_\_\_\_

**Kernel version:** \_\_\_\_\_

**One-line description of the symptom:**

\_\_\_\_\_

**What I observed first:**

\_\_\_\_\_

**First hypothesis:**

\_\_\_\_\_

**How I tested the first hypothesis:**

\_\_\_\_\_

**What happened instead:**

\_\_\_\_\_

**Second hypothesis (if applicable):**

\_\_\_\_\_

**What the root cause turned out to be:**

\_\_\_\_\_

**What specifically fixed it:**

\_\_\_\_\_

**What I learned that I did not know before:**

---

**What I would look at sooner next time:**

---

**Relevant source paths and line numbers:**

---

**Relevant mailing-list threads, commits, or bug IDs:**

---

The act of filling in the last two fields (“what I learned” and “what I would look at sooner next time”) is the most valuable part. Those fields compound across entries and gradually turn a developer who solves each bug individually into one who recognises patterns across bugs.

### 38.12.7 9.7 A Driver Handoff Checklist

A driver is rarely owned by the same person forever. Whether you are handing a driver off to a new maintainer, leaving a company, or simply transferring stewardship to a colleague, there is a short list of things that make the handoff successful. Use this checklist when the handoff is planned, and use it as a self-review when you suspect a handoff may become necessary.

- ☐ The driver has a current manual page that reflects its real behaviour.
- ☐ The driver has a short design document that explains the non-obvious decisions and the hardware-specific quirks.
- ☐ The driver has a regression test suite that the new maintainer can run on their own hardware.
- ☐ The driver has a working build from a clean tree, with no local modifications required.
- ☐ The driver has been tested against at least two FreeBSD releases, and the compatibility range is documented.
- ☐ All TODOs, known bugs, and planned work are recorded in an issue tracker or a text file in the repository.
- ☐ The `sysctl` variables and `dev.` names are documented with their meaning and their expected range of values.
- ☐ Any vendor-specific documentation, datasheets, or NDAs that affect the driver are recorded, either with the driver or in a known location.
- ☐ The hardware required to test the driver is described in enough detail that the new maintainer can acquire or borrow it.
- ☐ The new maintainer has been introduced to the relevant mailing list discussions and to any community members who have contributed.
- ☐ Any outstanding patches, reviews, or conversations have been surfaced to the new maintainer.
- ☐ The commit history does not contain any proprietary information that would prevent further public development.

If a driver cannot cleanly meet this checklist, the handoff will leave debt for the next maintainer. Spend the time before the handoff closing the gaps. The cleaner the handoff, the more likely the driver continues to receive care after you step away.



### 38.12.8 9.8 A Quarterly Review Template

Every three months, take an hour or two to review your broader trajectory, not just your immediate projects. The quarterly review is different from the monthly stay-current rhythm. The monthly rhythm keeps you in touch with the kernel as a living system. The quarterly review keeps you in touch with your own growth as a developer.

Use this template for each quarterly review. Keep the filled copies together in your `freebsd-learning` repository so you can read them in sequence.

**Quarter:** Q\_\_ of \_\_

**Start date:** \_\_\_\_\_

**End date:** \_\_\_\_\_

**What I worked on this quarter, at a glance:**

\_\_\_\_\_

**What I consider the most significant thing I learned:**

\_\_\_\_\_

**A specific moment I am proud of:**

\_\_\_\_\_

**A specific moment that was harder than I expected:**

\_\_\_\_\_

**Progress against my learning roadmap:**

- Milestones I completed: \_\_\_\_\_
- Milestones I missed: \_\_\_\_\_
- Milestones I changed or abandoned, and why: \_\_\_\_\_

**Community engagement this quarter:**

- Patches submitted: \_\_\_\_\_
- Reviews performed: \_\_\_\_\_
- Mailing-list replies: \_\_\_\_\_
- Conference talks watched: \_\_\_\_\_
- Bug reports filed or triaged: \_\_\_\_\_

**Toolkit improvements this quarter:**

- New scripts or templates I added: \_\_\_\_\_
- Existing scripts I improved: \_\_\_\_\_

- Scripts I retired: \_\_\_\_\_

**What I want to prioritise next quarter:**

1. \_\_\_\_\_
2. \_\_\_\_\_
3. \_\_\_\_\_

**What I want to stop doing, or do less of, next quarter:**

\_\_\_\_\_

**Who I would like to learn from next quarter:**

\_\_\_\_\_

**One small, concrete commitment for the coming month:**

\_\_\_\_\_

Filling in this template takes real time. It is tempting to skip it when you are busy with other work, but skipping it is a false economy. A quarterly review is one of the cheapest forms of long-term self-correction available to a working engineer, and an hour of honest reflection prevents many hours of drifted effort later.

### 38.12.9 9.9 Where to Keep These Artifacts

A practical question: where should these artifacts live? The answer depends on your working style, but a few options work well.

A personal Git repository is ideal. Create a repository called something like `freebsd-learning` or `kernel-work` and commit each artifact as you produce it. The version history becomes its own record of your growth. If you use a hosted Git service, make the repository private unless you are comfortable sharing your progress publicly; both choices are legitimate.

A folder in your home directory works if you prefer not to use Git. Organise it by date so that you can see the chronology. Back it up regularly.

A paper notebook works for some readers. The constraint it imposes (one-way revision, no text search) is a feature for some kinds of thinking, and paper notebooks have their own durability advantages. The downside is that you cannot easily share the contents with others or sync them across devices.

Whatever you choose, choose consistently. Scattered artifacts across multiple systems tend to become unreachable over time. A single, stable location where your learning record lives is much more useful.

### 38.12.10 9.10 Wrapping Up Section 9

The checklists and templates in this section are small tactical tools. They will not change your life individually, but used regularly, they help structure the ongoing work of professional growth. Copy them, adapt them to your circumstances, and treat them as living documents that evolve as you learn.

The chapter is now approaching its end. What remains is the wrapping-up material of the chapter itself, and the final closing words of the book.

## 38.13 Wrapping Up

This chapter has walked through the full reflective arc of the book's ending: a recognition of what you accomplished, an honest assessment of where you stand, a map of the advanced topics that remain beyond the book, a practical toolkit for continued work, a careful treatment of community engagement, a rhythm for staying current with kernel development, hands-on reflection labs, challenge exercises, and planning artifacts. Each section has had its own closing bridge, and the reflection labs have produced concrete artifacts that you will keep beyond the book.

A few themes run through the chapter and deserve a final explicit summary.

The first theme is that finishing the book is a milestone, not an endpoint. The milestone is real. The journey from no kernel experience to competent driver author is a substantial one, and you have completed it. The endpoint is something different: mastery is never reached in kernel work, because the kernel keeps changing and the depth of understanding keeps deepening. A finished book is a useful checkpoint, not a final destination.

The second theme is that independence grows from practice, not from reading. You have the knowledge to write drivers. The knowledge is necessary but not sufficient. What turns knowledge into independent capability is repeated practice on real problems, and the book can only gesture toward that practice; it cannot substitute for it. The chapter has suggested many specific paths for that practice, and the right one for you is whichever you are most likely to follow through on.

The third theme is that the FreeBSD community is a resource, a destination, and a teacher. You can finish this book and continue as a solo developer, and that is a legitimate path. But if you engage with the community, you will grow faster, reach further, and encounter ideas you could not have found alone. The community is not an obligation. It is an invitation, and the chapter has shown you many specific doors through which you can enter.

The fourth theme is that staying current is a rhythm, not a task. The FreeBSD kernel changes constantly, and a developer who stops paying attention will find their knowledge slowly rotting. A developer who maintains a weekly glance, a monthly rebuild, and a quarterly review will stay in step with the project over decades. The rhythm is sustainable, the benefits compound, and the chapter has shown you exactly what that rhythm looks like in practice.

The fifth theme is that contribution takes many forms. A reader who thinks of contribution only as committing driver code will miss most of the ways to be useful to the project. Mailing list help, documentation work, translations, mentoring, bug triage, code review, and test-case contributions are all real contributions, and all of them are valued. The chapter has named each of them and pointed you toward the channels where they happen.

The sixth theme is that a development toolkit pays for itself. The driver template, the virtual lab, the regression tests, the CI scripts, and the packaging habits are not glamorous. Each of them takes a few hours to set up and a few more hours to maintain. But together they turn each new driver project from a clean-slate effort into a refinement of an existing pipeline, and the compounding gain is large. The chapter has given you starter artifacts for each part of the toolkit, and the companion directory contains the templates.

The seventh and perhaps most important theme is that you are, now, in the practical sense, a FreeBSD device-driver developer. Not a senior one. Not yet a committer. But a working developer who can write, debug, test, and submit drivers. That identification is not a small one. It took work to earn it, and it deserves to be acknowledged.

Take a moment to appreciate what has changed in your toolkit over the course of the book. Before Chapter 1, the FreeBSD kernel may have been an opaque system that ran below your applications. Now it is a readable piece of software, with familiar patterns, known subsystems, and a predictable layout. Before Chapter 1, writing a driver may have been a distant aspiration. Now it is a concrete activity with a known workflow. Before Chapter 1, the FreeBSD community may have been an abstraction. Now it is a set of specific channels where specific kinds of work happen.

That transformation is the quiet kind of change that long books produce. It is not the dramatic transformation of a single chapter, but the slow accumulation of many chapters over many hours. Its value is measured in the difference between the person who started and the person who finished. By that measure, you are not the same reader who opened Chapter 1, and neither is the book.

Before the book closes, the final words are directed to the larger journey of which this book has been only a part.

## 38.14 Part 7 Checkpoint

Part 7 spent ten chapters turning a capable driver author into one who can ship work into the wider FreeBSD world. Before the book's final pages, it is worth pausing to confirm that the mastery topics have settled, because nothing after Chapter 38 will remind you of them again.

By the end of Part 7 you should be comfortable refactoring a driver so that its hardware-facing code sits behind a small backend interface, compiling it against both a simulation backend and a real one, and running the resulting module under bhyve, inside a VNET jail, or behind VirtIO without changing its core. You should be comfortable hardening a driver against hostile or careless input, which means bounds-checking every copy, zeroing buffers before copyout, applying privilege checks through `priv_check`, rate-limiting logs, and carrying a safe detach all the way through `MOD_QUIESCE` and resource teardown. You should be able to measure a driver's behaviour with the right instrument for the question at hand: DTrace and SDT probes for function-level tracing, `pmcstat` and `hwpmc` for CPU-level events, per-CPU counters and cache-aligned `softc` fields for contention relief, and `kgdb` with a core dump or the GDB stub for bugs that survive `INVARIANTS` and `WITNESS`. And you should be able to extend a driver's user-space contract with `poll(2)`, `kqueue(2)`, and `SIGIO` support, approach an undocumented device through safe probing and reverse-engineering discipline, and prepare a submission that reviewers will actually accept, from KNF-clean source through the manual page to the cover letter and the Phabricator review.

If any of those still feels soft, the labs to revisit are:

- Portability and backend separation: Lab 3 (Extract the Backend Interface) and Lab 5 (Add a Simulation Backend) in Chapter 29.
- Virtualisation and jails: Lab 3 (A Minimal Character Device Driver Inside a Jail) and Lab 6 (Building and Loading the vtedu Driver) in Chapter 30.
- Security discipline: Lab 2 (Bounds-Check the Buffer), Lab 4 (Add Privilege Checks), and Lab 6 (Safe Detach) in Chapter 31.
- Embedded and Device Tree work: Lab 2 (Build and Deploy an Overlay) and Lab 4 (Build the edled Driver End to End) in Chapter 32.
- Performance and profiling: Lab 2 (DTrace perfdemo) and Lab 4 (Cache Alignment and Per-CPU Counters) in Chapter 33.
- Advanced debugging: Lab 2 (Capturing and Analyzing a Panic With kgdb) and Lab 5 (Catching a Use-After-Free With memguard) in Chapter 34.
- Asynchronous I/O: Lab 2 (Adding poll() Support), Lab 3 (Adding kqueue Support), and Lab 6 (The Combined v2.5-async Driver) in Chapter 35.
- Reverse engineering: Lab 3 (Building a Safe Register Probing Module) and Lab 5 (Building a Safe Probing Wrapper) in Chapter 36.
- Upstreaming: Lab 1 (Prepare the File Layout), Lab 4 (Draft the Manual Page), and Lab 6 (Generate a Submission Patch) in Chapter 37.

Part 7 has no successor part. What it does have is the practice that follows the book: real hardware, real review cycles, real bugs, and the long-running rhythm of staying current that Chapter 38 has already named. The baseline the rest of your career assumes is exactly what these chapters taught, a driver that is portable, hardened, measured, debuggable, and submittable, written by an author who knows when to reach for each tool and when to step away from the keyboard. If the ideas above feel like habits rather than lookups, the book's work is done. What remains is yours.

## 38.15 Final Closing Words for the Book

This book began with a story about a chemistry student in Brazil in 1995, who found FreeBSD in a university library and slowly, over years, turned that curiosity into a career. That story was not offered as a model to follow; it was offered as evidence that curiosity is enough to start, even when the starting conditions are difficult.

Your starting conditions, whatever they were, brought you to the last pages of a technical book about an operating system that is now more than thirty years old. The operating system is still being developed, still being used in places that matter, and still welcoming new contributors. The ability to write device drivers for it is a skill that remains valuable and will continue to be valuable for as long as the project continues. By finishing this book, you have acquired that skill to a working level. What you do with it from here is entirely yours to decide.

There is a specific hope I have for the readers of this book, and I want to name it clearly. The hope is that at least some of you will find, in the months after finishing, that the book unlocked something you did not know was possible. That you will write a driver for a device you care about, or fix a bug that has been bothering you, or answer a question on a mailing list that helps a stranger, or begin a conversation with the project that leads somewhere you could not have anticipated. That the skills this book has given you become the beginning of a longer relationship with FreeBSD than the book's pages can contain.

The kernel is not magic. Over the course of this book, you have come to see that more clearly. A kernel is software, written by people, available for anyone to read, and modifiable by anyone with the patience to

understand it. The difference between a reader who finds kernels mysterious and a reader who finds them approachable is, in the end, whether they have actually sat down and looked. You have looked. You have looked carefully, across many chapters and many hours, and the mystery has receded.

That receding is permanent. You cannot un-know the shape of a driver now. You cannot un-know what a `softc` is, or what a `device_method_t` array does, or what the `_IOWR` macro means. Those pieces of knowledge will be with you for the rest of your career. They will help you read other systems, write other software, and reason about problems that are nothing like drivers. The investment you made in learning them has a return that extends well beyond FreeBSD.

There is also something the book has tried to convey without saying it directly. Systems engineering is a craft, and crafts are learned in a particular way: through long practice with specific materials, under the guidance of people who have done the work before. Books are one part of that guidance, but they are not the whole of it. The rest of the guidance comes from source code read carefully, from bugs debugged patiently, from mailing-list threads followed through to their conclusions, from conference talks watched with attention, and from the slow accumulation of intuitions that cannot be explicitly written down. This book has pointed you toward that larger guidance. The larger guidance is what will carry you forward from here.

If you choose to engage with FreeBSD after this book, you will find a community that has been remarkably stable across decades of technology change. The faces change, but the culture persists: an emphasis on careful engineering, on clear documentation, on long-term thinking, on the value of doing things well rather than quickly. Joining that culture is a privilege, and the culture welcomes newcomers who approach it with respect for its traditions.

If you do not choose to engage further, that is also fine. Many readers of this book will use what they learned for their own purposes, inside their own companies or on their own hardware, and never contribute publicly. Those readers have not wasted their time. They have acquired a skill, and the skill is useful in many contexts. The FreeBSD community is large enough to welcome those who engage and patient enough to let others benefit from the work without engaging. Both relationships are legitimate, and the book has tried not to pressure you toward either one.

A few last specific things deserve to be said.

If you submit a driver upstream, and it takes three rounds of review to land, that is normal. Do not be discouraged. Most first submissions go through several rounds, and the reviewers are helping you rather than judging you. Patience is the discipline that completes a submission; impatience is the discipline that abandons one. You have both disciplines available to you. Choose the first.

If you hit a bug you cannot figure out, and you have looked at it for an hour without progress, stop. Step away. Come back in a few hours or the next day. Fresh eyes see what tired eyes miss. The kernel is patient. Your bug will still be there tomorrow, and you will likely see its cause within the first few minutes of the next session. This is one of the most consistent patterns in debugging work, and it applies to kernel work as fully as to any other kind.

If you get discouraged, and at some point you probably will, remember why you started. Most people who finish a book like this finished it because something about FreeBSD caught their imagination. Return to that something. Reread the chapter that excited you most, or return to the driver you wrote that worked first, or watch a conference talk about a topic that made you curious. The return to the source of excitement is how you sustain a long career, and the sustenance matters more than the speed.

If you teach someone else what you learned, your own understanding will deepen in ways that solitary practice cannot produce. The slower, less efficient-feeling work of explaining to a newcomer what a device-method table does, or how `copyin` protects the kernel from userland pointers, or why locking order matters, is one of the most potent forms of learning there is. If you ever have the chance to help a future reader through the material you just worked through, take it. You will be doing them a favour, but you will be doing yourself a larger one.

This book has been a long journey, and every long one deserves a formal ending. Thank you for reading it. Thank you for working through the labs, the challenges, and the reflections. Thank you for caring enough about your own learning to see the book through. Your time is the most limited resource you have, and you spent a substantial portion of it on the pages of this book. I hope the return on that investment is large, and I hope you carry forward both the technical knowledge and the habits of thought that the book tried to convey.

The FreeBSD kernel is not going anywhere. It will still be here next year, and the year after, and ten years from now. It will still have drivers that need writing, bugs that need fixing, and documentation that needs updating. Whenever you are ready to return to the kernel, whether that is tomorrow or a decade from now, it will be waiting for you, and the community around it will still welcome the careful, curious work that you have learned to do.

Close the book now. Take a moment to notice that you have done something substantial. Then, when you are ready, open a terminal, type `cd /usr/src/sys/dev/`, and look around. You know what you are looking at. You know how to read it. You know how to change it. You know how to make it your own.

Good luck, and welcome to the community.

The kernel was never magic.

You just learned how to work with it.

## 39 Appendix A: FreeBSD Kernel API Reference

### 39.1 How to Use This Appendix

This appendix is the companion lookup table for everything the book has taught you to use inside a FreeBSD driver. The main chapters build each API carefully, show it in a working driver, and explain the mental model behind it. This appendix is the short, scannable counterpart you keep open while coding, debugging, or reading someone else's driver.

It is deliberately written to be a reference and not a tutorial. It does not try to teach any subsystem from scratch. Each entry assumes you have already met the API somewhere in the book, or that you are willing to read the manual page before you use it. What the entry gives you is the vocabulary to navigate: what the API is for, the small set of names that actually matter, the mistakes you are likely to make, where it typically fits in the driver lifecycle, and which chapter teaches it in full. If the entry is doing its job, you can answer four questions in under a minute:

1. Which API family do I need for the problem in front of me?
2. What is the exact name of the function, macro, or type I want?
3. What caveat should I check before I trust it?
4. Which manual page or chapter should I open next?

Nothing more is promised. Every detail below was verified against the FreeBSD 14.3 source tree and the corresponding manual pages in `man 9`. Where a distinction is important but deferred to another part of the book, the entry points forward rather than pretending to resolve it here.

#### 39.1.1 How the Entries Are Organised

The appendix groups APIs by the problem they solve, not alphabetically. A driver rarely reaches for a name in isolation. It reaches for a whole family: memory with its flags, a lock with its sleep rules, a callout with its cancellation story. Keeping those families together makes the appendix more useful for real lookup tasks.

Within each family, every entry follows the same short pattern:

- **Purpose.** What the API is for, in one or two sentences.
- **Typical use in drivers.** When a driver reaches for it.
- **Key names.** The functions, macros, flags, or types you actually call or declare.
- **Header(s).** Where the declarations live.
- **Caveats.** The handful of mistakes that cause real bugs.
- **Lifecycle phase.** Where in probe, attach, normal operation, or detach the API usually appears.
- **Manual pages.** The `man 9` entries to read next.
- **Where the book teaches this.** Chapter references for full context.



Keep this pattern in mind when you scan. If you only need the name of a flag, look at **Key names**. If you only need the manual page, look at **Manual pages**. If you have forgotten why the API exists at all, read **Purpose** and stop there.

### 39.1.2 What This Appendix Is Not

This appendix is not a replacement for `man 9`, not a replacement for the book's teaching chapters, and not a replacement for reading real drivers under `/usr/src/sys/dev/`. It is short on purpose. The canonical reference remains the manual page; the canonical mental model remains the chapter that introduced the API; the canonical truth remains the source tree. This appendix helps you find all three quickly.

It also does not cover every kernel interface. The kernel is large, and a full reference would repeat material that belongs in Appendix E (FreeBSD Internals and Kernel Reference) or in focused chapters such as Chapter 16 (Accessing Hardware), Chapter 19 (Handling Interrupts), and Chapter 20 (Advanced Interrupt Handling). The goal here is coverage of the APIs that a driver author actually uses in day-to-day work, at the level of detail a driver author actually needs.

## 39.2 Reader Guidance

You can use this appendix in three different ways, and they each call for a different reading strategy.

If you are **writing new code**, treat it as a checklist. Pick the family that matches your problem, skim the entries, note the key names, and jump to the manual page or the chapter for the details. Time investment: one or two minutes per lookup.

If you are **debugging**, treat it as a map of assumptions. When a driver misbehaves, the bug is almost always in a caveat that the author overlooked: a mutex held across a sleepable copy, a callout stopped but not drained, a resource freed before the interrupt was torn down. The **Caveats** line of each entry is where those assumptions live. Read them in order and ask whether your driver honours each one.

If you are **reading an unfamiliar driver**, treat it as a translator. When you see a function or macro you do not recognise, find its family in this appendix, read the **Purpose**, and move on. The full understanding can come later from the chapter or the manual page. The goal during exploration is to keep moving and to form an initial mental model of what the driver is doing.

A few conventions used throughout:

- All source paths are shown in book-facing form, `/usr/src/sys/...`, matching the layout on a standard FreeBSD system.
- Manual pages are cited in the usual FreeBSD style: `mtx(9)` means section 9 of the manual. You can read any of them with, for example, `man 9 mtx`.
- When a family has no dedicated manual page, the entry says so and points to the closest available documentation.
- When the book defers a topic to a later chapter or to Appendix E, the entry points forward rather than fabricating detail here.

With that in mind, we can open the appendix proper. The first family is memory: where drivers get the bytes they need, how they give them back, and which flags control the behaviour along the way.

## 39.3 Memory Allocation

Every driver allocates memory, and every allocation carries rules about when you may block, where the memory is physically located, and how it is returned. The kernel provides three main allocators: `malloc(9)` for general-purpose allocation, `uma(9)` for high-frequency fixed-size objects, and `contigmalloc(9)` for physically contiguous ranges that hardware can address. Below you will find each one at a glance, plus the small vocabulary of flags they share.

### 39.3.1 `malloc(9)` / `free(9)` / `realloc(9)`

**Purpose.** General-purpose kernel memory allocator. Gives you a byte buffer of any size, tagged by a `malloc_type` so you can account for it later with `vmstat -m`.

**Typical use in drivers.** Softc allocation, small variable-size buffers, temporary scratch space, anything where a fixed-size zone would be overkill.

**Key names.**

- `void *malloc(size_t size, struct malloc_type *type, int flags);`
- `void free(void *addr, struct malloc_type *type);`
- `void *realloc(void *addr, size_t size, struct malloc_type *type, int flags);`
- `MALLOC_DEFINE(M_FOO, "foo", "description for vmstat -m");`
- `MALLOC_DECLARE(M_FOO);` for use in headers.

**Header.** `/usr/src/sys/sys/malloc.h`.

**Flags that matter.**

- `M_WAITOK`: the caller may block until memory is available. The allocation will succeed or the kernel will panic.
- `M_NOWAIT`: the caller must not block. The allocation may return `NULL`. Always check for `NULL` with `M_NOWAIT`.
- `M_ZERO`: zero the returned memory before handing it back. Combine with either wait flag.
- `M_NODUMP`: exclude the allocation from crash dumps.

**Caveats.**

- `M_WAITOK` must not be used while holding a spin mutex, in an interrupt filter, or in any context that cannot sleep.
- `M_NOWAIT` callers must check the return value. Failing to handle `NULL` is one of the most common driver crashes in review.
- Never mix allocator families. Memory returned by `malloc(9)` must be freed by `free(9)`; `uma_zfree(9)` and `contigfree(9)` are not interchangeable.
- The `struct malloc_type` pointer must match between `malloc` and the corresponding `free`.

**Lifecycle phase.** Most commonly in `attach` (softc, buffers) and `detach` (release). Smaller allocations can appear in normal I/O paths as long as the context permits the chosen flag.

**Manual page.** `malloc(9)`.

**Where the book teaches this.** Introduced in Chapter 5 alongside kernel-specific C idioms; used in Chapter 7 when your first driver allocates its softc; revisited in Chapter 10 when I/O buffers become real, and again in Chapter 11 when allocation flags must obey locking rules.

### 39.3.2 uma(9) Zones

**Purpose.** Fixed-size object cache, highly optimised for allocations that are frequent, uniform, and performance-sensitive. Reuses objects instead of repeatedly hitting the general allocator.

**Typical use in drivers.** Network mbuf-like structures, per-packet state, per-request descriptors, anything where you allocate and free millions of identical small objects per second.

**Key names.**

- `uma_zone_t uma_zcreate(const char *name, size_t size, uma_ctor, uma_dtor, uma_init, uma_fini, int align, uint32_t flags);`
- `void uma_zdestroy(uma_zone_t zone);`
- `void *uma_zalloc(uma_zone_t zone, int flags);`
- `void uma_zfree(uma_zone_t zone, void *item);`

**Header.** `/usr/src/sys/vm/uma.h`.

**Flags.**

- `M_WAITOK`, `M_NOWAIT`, `M_ZERO` on allocation, identical in meaning to `malloc(9)`.
- Creation-time flags such as `UMA_ZONE_ZINIT`, `UMA_ZONE_NOFREE`, `UMA_ZONE_CONTIG`, and alignment hints (`UMA_ALIGN_CACHE`, `UMA_ALIGN_PTR`, and so on) tune behaviour for specific workloads.

**Caveats.**

- Zones must be created before they can be used and must be destroyed before the module unloads. Forgetting `uma_zdestroy` in `detach` leaks the entire zone.
- Constructors and destructors run on allocation and free respectively, not on zone creation and destruction; use the `init` and `fini` callbacks for once-per-slab work.
- A zone is expensive to create. Create one per object type per module, not per instance.
- There is no dedicated `uma(9)` manual page. The authoritative reference is the header and the existing users under `/usr/src/sys/`.

**Lifecycle phase.** `uma_zcreate` in module load or early attach; `uma_zalloc` and `uma_zfree` in I/O paths; `uma_zdestroy` in module unload.

**Manual page.** None dedicated. Read `/usr/src/sys/vm/uma.h` and look at `/usr/src/sys/kern/kern_mbuf.c` and `/usr/src/sys/net/netisr.c` for realistic use.

**Where the book teaches this.** Mentioned briefly in Chapter 7 as an alternative to `malloc(9)`; revisited when high-rate drivers need it in Chapter 28 (networking) and Chapter 33 (performance tuning).

### 39.3.3 contigmalloc(9) / contigfree(9)

**Purpose.** Allocate a physically contiguous range of memory within a specified address window. Required when hardware must DMA into memory without an IOMMU and therefore needs contiguous physical pages.

**Typical use in drivers.** DMA buffers for devices that cannot scatter-gather, and only after confirming that `bus_dma(9)` is not a better fit.

**Key names.**

- `void *contigmalloc(unsigned long size, struct malloc_type *type, int flags, vm_paddr_t low, vm_paddr_t high, unsigned long alignment, vm_paddr_t boundary);`
- `void contigfree(void *addr, unsigned long size, struct malloc_type *type);`

**Header.** `/usr/src/sys/sys/malloc.h`.

**Caveats.**

- Fragmentation after boot makes large contiguous allocations fail. Do not assume success.
- For almost all modern hardware, prefer the `bus_dma(9)` framework. It handles tags, maps, bouncing, and alignment in a portable way.
- `contigmalloc` allocations are a scarce system resource; free them as soon as possible.

**Lifecycle phase.** Typically in `attach`; freed in `detach`.

**Manual page.** `contigmalloc(9)`.

**Where the book teaches this.** Mentioned alongside `bus_dma(9)` in Chapter 21 when DMA first becomes a real concern.

### 39.3.4 Allocation Flag Cheat Sheet

| Flag                  | Meaning                                                      |
|-----------------------|--------------------------------------------------------------|
| <code>M_WAITOK</code> | Caller may block until memory is available.                  |
| <code>M_NOWAIT</code> | Caller must not block; returns <code>NULL</code> on failure. |
| <code>M_ZERO</code>   | Zero the allocation before returning.                        |
| <code>M_NODUMP</code> | Exclude the allocation from crash dumps.                     |

Use `M_WAITOK` only where sleeping is permitted. When in doubt, the safe answer is `M_NOWAIT` plus a `NULL` check.

## 39.4 Synchronisation Primitives

If memory is the raw material of a driver, synchronisation is the discipline that prevents two execution contexts from corrupting it at the same time. FreeBSD gives you a small, well-designed toolkit. The names below are the ones you will meet most often. Full teaching lives in Chapters 11, 12, and 15, with interrupt-context nuances in Chapters 19 and 20; this appendix collects the vocabulary.

### 39.4.1 mtx(9): Mutexes

**Purpose.** The default kernel mutual-exclusion primitive. A thread acquires the lock, enters the critical section, and releases the lock.

**Typical use in drivers.** Protecting softc fields, circular buffers, reference counts, and any shared state whose critical section is short and does not sleep.

**Key names.**

- `void mtx_init(struct mtx *m, const char *name, const char *type, int opts);`
- `void mtx_destroy(struct mtx *m);`
- `mtx_lock(m), mtx_unlock(m), mtx_trylock(m).`
- `mtx_assert(m, MA_OWNED | MA_NOTOWNED | MA_RECURSED | MA_NOTRECURSED);` for invariants.
- Sleep-on-mutex helpers: `msleep(9)` and `mtx_sleep(9)`.

**Header.** `/usr/src/sys/sys/mutex.h`.

**Options.**

- `MTX_DEF`: the default sleepable-on-contention mutex. Use this for almost everything.
- `MTX_SPIN`: pure spinlock. Required for interrupt-filter context and other places where blocking is impossible. Rules are stricter.
- `MTX_RECURSE`: allow the same thread to acquire the lock multiple times. Use with caution; it often hides design mistakes.
- `MTX_NEW`: force `mtx_init` to treat the lock as newly created. Useful with `WITNESS`.

**Caveats.**

- Never sleep while holding an `MTX_DEF` or `MTX_SPIN` mutex. `uiomove(9)`, `copyin(9)`, `copyout(9)`, `malloc(9, M_WAITOK)`, and most bus primitives can sleep. Audit carefully.
- Always pair `mtx_init` with `mtx_destroy`. Forgetting the destroy leaks internal state and annoys `WITNESS`.
- Lock order matters. Once the kernel has seen you acquire lock A before lock B, it will warn if you ever reverse the pair. Plan your lock hierarchy in advance.
- `MTX_SPIN` disables preemption; hold it for as short a time as possible.

**Lifecycle phase.** `mtx_init` in attach; `mtx_destroy` in detach. Lock and unlock operations everywhere in between.

**Manual pages.** `mutex(9)`, `mtx_pool(9)`, `msleep(9)`.

**Where the book teaches this.** First treatment in Chapter 11, deepened in Chapter 12 with lock-order and `WITNESS` discipline, and revisited in Chapter 19 for interrupt-safe variants (`MTX_SPIN`).

### 39.4.2 sx(9): Sleepable Shared-Exclusive Locks

**Purpose.** A reader-writer lock where either the reader or the writer may block. Use when multiple readers are common, writers are rare, and the critical section may sleep.

**Typical use in drivers.** Configuration state read by many paths and modified infrequently. Not for fast-path data.

**Key names.**

- `void sx_init(struct sx *sx, const char *desc);`
- `void sx_destroy(struct sx *sx);`
- `sx_slock(sx), sx_sunlock(sx)` for shared access.
- `sx_xlock(sx), sx_xunlock(sx)` for exclusive access.
- `sx_try_slock, sx_try_xlock, sx_upgrade, sx_downgrade.`
- `sx_assert(sx, SA_SLOCKED | SA_XLOCKED | SA_LOCKED | SA_UNLOCKED);`

**Header.** `/usr/src/sys/sys/sx.h`.

**Caveats.**

- `sx` allows sleeping inside the critical section, unlike `mtx`. That flexibility is the whole point; make sure you actually need it.
- `sx` locks are more expensive than mutexes. Do not use them as a default.
- Avoid mixing `sx` and `mtx` in the same lock order without thinking through the implications.

**Lifecycle phase.** `sx_init` in `attach`; `sx_destroy` in `detach`.

**Manual page.** `sx(9)`.

**Where the book teaches this.** Chapter 12.

### 39.4.3 `rmlock(9)`: Read-Mostly Locks

**Purpose.** Extremely fast reader path, slower writer path. Readers do not contend with each other. Designed for data that is read on every operation but written only rarely.

**Typical use in drivers.** Routing-like tables, configuration state used in fast paths, structures where writer overhead is acceptable because writes are rare.

**Key names.**

- `void rm_init(struct rmlock *rm, const char *name);`
- `void rm_destroy(struct rmlock *rm);`
- `rm_rlock(rm, tracker), rm_runlock(rm, tracker).`
- `rm_wlock(rm), rm_wunlock(rm).`

**Header.** `/usr/src/sys/sys/rmlock.h`.

**Caveats.**

- Each reader needs its own struct `rm_priotracker`, typically on the stack. Do not share one.
- Readers may not sleep unless the lock was initialised with `RM_SLEEPABLE`.
- The writer path is heavy; if writes are frequent, `sx` or `mtx` is a better choice.

**Lifecycle phase.** `rm_init` in `attach`; `rm_destroy` in `detach`.

**Manual page.** `rmlock(9)`.

**Where the book teaches this.** Introduced briefly in Chapter 12 and used in later chapters where read-mostly patterns arise.

### 39.4.4 `cv(9)` / `condvar(9)`: Condition Variables

**Purpose.** A named wait channel. One or more threads sleep until another thread signals that the condition they are waiting for has become true.

**Typical use in drivers.** Waiting for a buffer to drain, for hardware to finish a command, or for a specific state transition. Use instead of bare `wakeup(9)` channels when you want the wait reason to be explicit.

**Key names.**

- `void cv_init(struct cv *cv, const char *desc);`
- `void cv_destroy(struct cv *cv);`
- `cv_wait(cv, mtx), cv_wait_sig(cv, mtx), cv_wait_unlock(cv, mtx).`
- `cv_timedwait(cv, mtx, timo), cv_timedwait_sig(cv, mtx, timo).`
- `cv_signal(cv), cv_broadcast(cv), cv_broadcastpri(cv, pri).`

**Header.** `/usr/src/sys/sys/condvar.h`.

**Caveats.**

- The mutex passed to `cv_wait` must be held by the caller; `cv_wait` drops it while sleeping and reacquires it on return.
- Always recheck the predicate after `cv_wait` returns. Spurious wakeups and signals are possible.
- `cv_signal` wakes one waiter; `cv_broadcast` wakes all. Pick based on the design, not on instinct.

**Lifecycle phase.** `cv_init` in `attach`; `cv_destroy` in `detach`.

**Manual page.** `condvar(9)`.

**Where the book teaches this.** Chapter 12, with interruptible and timed waits revisited in Chapter 15.

### 39.4.5 `sema(9)`: Counting Semaphores

**Purpose.** A counting semaphore with `wait` and `post` operations. Less common than mutexes or condition variables.

**Typical use in drivers.** Producer-consumer patterns where a counted resource must be tracked, such as a fixed pool of command slots.

**Key names.**

- `void sema_init(struct sema *sema, int value, const char *desc);`
- `void sema_destroy(struct sema *sema);`
- `sema_wait(sema), sema_trywait(sema), sema_timedwait(sema, timo).`
- `sema_post(sema).`

**Header.** `/usr/src/sys/sys/sema.h`.

**Caveats.**

- Semaphores are appropriate for counting. For one-thread-in-critical-section patterns, use `mtx` instead.
- `sema_wait` may return early on signal; check return values.

**Manual page.** sema(9).

**Where the book teaches this.** Chapter 15, as part of the advanced synchronisation toolkit.

### 39.4.6 atomic(9): Atomic Operations

**Purpose.** Single-word, interruption-free read-modify-write operations. Faster than any lock, and strictly limited in expressiveness.

**Typical use in drivers.** Counters, flags, and compare-and-swap patterns where the critical section fits in one integer.

**Key names.**

- atomic\_add\_int, atomic\_subtract\_int, atomic\_set\_int, atomic\_clear\_int.
- atomic\_load\_int, atomic\_store\_int, with acquire and release variants.
- atomic\_cmpset\_int, atomic\_fcmpset\_int for compare-and-swap.
- Width variants: \_8, \_16, \_32, \_64, and pointer-sized \_ptr.
- Barrier helpers: atomic\_thread\_fence\_acq(), atomic\_thread\_fence\_rel(), atomic\_thread\_fence\_acq\_rel()

**Header.** /usr/src/sys/sys/atomic\_common.h plus machine/atomic.h for architecture-specific pieces.

**Caveats.**

- Atomic operations give you one word of mutual exclusion. Any invariant that spans two fields still needs a lock.
- Memory ordering matters. The plain operations are relaxed; use the \_acq, \_rel, and \_acq\_rel variants when one access must become visible before or after another.
- For per-CPU counters that are read rarely, counter(9) scales better.

**Lifecycle phase.** Any. Cheap enough to use in interrupt filters.

**Manual page.** atomic(9).

**Where the book teaches this.** Chapter 11, with counter(9) introduced alongside for per-CPU patterns.

### 39.4.7 epoch(9): Read-Mostly Lock-Free Sections

**Purpose.** Lightweight reader protection for data structures where readers vastly outnumber writers and latency must be minimal. Writers wait for all current readers to leave before freeing memory.

**Typical use in drivers.** Network stack fast paths, read-mostly lookup tables in high-performance drivers. Not a general-purpose primitive.

**Key names.**

- epoch\_t epoch\_alloc(const char \*name, int flags);
- void epoch\_free(epoch\_t epoch);
- epoch\_enter(epoch), epoch\_exit(epoch).
- epoch\_wait(epoch) for writers to block until readers drain.
- NET\_EPOCH\_ENTER(et) and NET\_EPOCH\_EXIT(et) wrappers for the network stack.



**Header.** `/usr/src/sys/sys/epoch.h`.

**Caveats.**

- Readers must not block, sleep, or call any function that does, while inside an epoch section.
- Freeing protected memory must be deferred until `epoch_wait` returns.
- Epoch sections are a tool of last resort, not a default primitive. Choose locks first.

**Manual page.** `epoch(9)`.

**Where the book teaches this.** Introduced briefly in Chapter 12; used in depth only where real drivers in later chapters require it.

### 39.4.8 Lock Decision Cheat Sheet

| You want to...                                        | Reach for                                      |
|-------------------------------------------------------|------------------------------------------------|
| Protect a short, non-sleeping critical section        | <code>mtx(9)</code> with <code>MTX_DEF</code>  |
| Protect state in an interrupt filter                  | <code>mtx(9)</code> with <code>MTX_SPIN</code> |
| Allow many readers, rare writers, may sleep           | <code>sx(9)</code>                             |
| Allow many readers, rare writers, no sleep in readers | <code>rmlock(9)</code>                         |
| Sleep until a named condition holds                   | <code>cv(9)</code> with a mutex                |
| Increment or compare-and-swap a single word           | <code>atomic(9)</code>                         |
| Lock-free reader path for read-mostly data            | <code>epoch(9)</code>                          |

When a row does not obviously match the problem, the full discussion in Chapters 11, 12, and 15 is the place to resolve it.

## 39.5 Deferred Execution and Timers

Drivers often need to run work later, periodically, or from a context that can sleep. The kernel gives you three tools for that: `callout(9)` for one-shot and periodic timers, `taskqueue(9)` for deferred work that may sleep, and `kthread(9)` or `kproc(9)` for long-running background threads. They overlap in some situations; the rule of thumb is that callouts run from the timer interrupt context (fast, no sleeping), taskqueues run in a worker thread (can sleep, can grab sleepable locks), and kthreads are whole threads you own.

### 39.5.1 `callout(9)`: Kernel Timers

**Purpose.** Schedule a function to run after a time delay. The callback runs in soft-interrupt context by default and must not sleep.

**Typical use in drivers.** Watchdog timers, polling intervals, retry delays, idle timeouts.

**Key names.**

- `void callout_init(struct callout *c, int mpsafe);` plus `callout_init_mtx` and `callout_init_rm`.
- `int callout_reset(struct callout *c, int ticks, void (*func)(void *), void *arg);`
- `int callout_stop(struct callout *c);`
- `int callout_drain(struct callout *c);`
- `int callout_pending(struct callout *c);`, `callout_active(struct callout *c);`

**Header.** `/usr/src/sys/sys/callout.h`.

**Caveats.**

- `callout_stop` does not wait for a running callback. Use `callout_drain` before freeing the softc in detach.
- A callout can fire even after you think you cancelled it if the timer was already dispatched. Guard the callback with a flag or use the `_mtx` and `_rm` variants to integrate cancellation with your lock.
- Running tickless kernels means ticks are abstract. Convert real time with `hz` or use `callout_reset_sbt` for subsecond precision.

**Lifecycle phase.** `callout_init` in attach; `callout_drain` in detach; `callout_reset` whenever the next firing time needs to be set.

**Manual page.** `callout(9)`.

**Where the book teaches this.** Chapter 13.

### 39.5.2 taskqueue(9): Deferred Work in a Worker Thread

**Purpose.** Hand off work from a context that cannot sleep, or that should not hold a lock for long, to a worker thread. Tasks queued on the same taskqueue run in order.

**Typical use in drivers.** Interrupt post-processing, hardware command completion handlers, reset and recovery paths that may need to allocate memory or grab sleepable locks.

**Key names.**

- `struct taskqueue *taskqueue_create(const char *name, int mflags, taskqueue_enqueue_fn, void *context);`
- `void taskqueue_free(struct taskqueue *queue);`
- `TASK_INIT(struct task *t, int priority, task_fn_t *func, void *context);`
- `int taskqueue_enqueue(struct taskqueue *queue, struct task *task);`
- `void taskqueue_drain(struct taskqueue *queue, struct task *task);`
- `void taskqueue_drain_all(struct taskqueue *queue);`
- Global queues such as `taskqueue_thread`, `taskqueue_swi`, `taskqueue_fast`.

**Header.** `/usr/src/sys/sys/taskqueue.h`.

**Caveats.**

- Enqueuing the same task twice before it runs is a no-op by design. If you need a new request every time, that is fine; if you expect two runs, use distinct tasks.
- `taskqueue_drain` waits for the task to finish; call it before freeing anything the task uses.

- Private taskqueues are cheap but not free. Reuse global taskqueues (`taskqueue_thread`, `taskqueue_fast`) unless you have a reason to own one.

**Lifecycle phase.** `taskqueue_create` (if private) and `TASK_INIT` in `attach`; `taskqueue_drain` and `taskqueue_free` in `detach`.

**Manual page.** `taskqueue(9)`.

**Where the book teaches this.** Chapter 14.

### 39.5.3 `kthread(9)` and `kproc(9)`: Kernel Threads and Processes

**Purpose.** Create a dedicated kernel thread or process that runs your function. Useful when the workload is long-lived, needs its own scheduling policy, or needs to be explicitly addressable.

**Typical use in drivers.** Rare. Most driver work is served better by a taskqueue or a callout. Kernel threads appear in subsystems with genuine long-running loops, such as housekeeping daemons.

**Key names.**

- `int kthread_add(void (*func)(void *), void *arg, struct proc *p, struct thread **td, int flags, int pages, const char *fmt, ...);`
- `int kproc_create(void (*func)(void *), void *arg, struct proc **procp, int flags, int pages, const char *fmt, ...);`
- `void kthread_exit(void);`
- `kproc_exit`, `kproc_suspend_check`.

**Header.** `/usr/src/sys/sys/kthread.h`.

**Caveats.**

- Creating a thread is heavier than queuing a task. Prefer `taskqueue(9)` unless the workload is genuinely long-running.
- Cleanly shutting down a `kthread` requires cooperation: set a stop flag, wake the thread, and wait for it to exit. Forgetting any step leaks the thread across module unloads.
- A `kthread` must exit by calling `kthread_exit`, not by returning.

**Manual pages.** `kthread(9)`, `kproc(9)`.

**Where the book teaches this.** Mentioned in Chapter 14 as the heavier alternative to taskqueues.

### 39.5.4 Deferred Work Decision Cheat Sheet

| You need to...                                      | Reach for                 |
|-----------------------------------------------------|---------------------------|
| Fire a function after a delay, briefly, no sleeping | <code>callout(9)</code>   |
| Defer work that may sleep or grab sleepable locks   | <code>taskqueue(9)</code> |
| Run a persistent background loop                    | <code>kthread(9)</code>   |
| Convert short periodic polling into real interrupts | See Chapter 19            |

## 39.6 Bus and Resource Management

The bus layer is where a driver meets hardware. Newbus introduces the driver to the kernel; `rman(9)` hands out the resources that represent MMIO regions, I/O ports, and interrupts; `bus_space(9)` accesses them portably; `bus_dma(9)` lets devices DMA safely.

### 39.6.1 Newbus: `DRIVER_MODULE`, `DEVMETHOD`, and Related Macros

**Purpose.** Register a driver with the kernel, bind it to a device class, declare the entry points the kernel should call, and publish version and dependency information.

**Typical use in drivers.** Every kernel module that owns a device. This is the scaffolding that turns a pile of C code into something `kldload` can attach to hardware.

**Key names.**

- `DRIVER_MODULE(name, bus, driver, devclass, evh, evharg);`
- `MODULE_VERSION(name, version);`
- `MODULE_DEPEND(name, busname, vmin, vpref, vmax);`
- `DEVMETHOD(method, function)` and `DEVMETHOD_END` for the method table.
- `device_method_t` entries such as `device_probe`, `device_attach`, `device_detach`, `device_shutdown`, `device_suspend`, `device_resume`.
- Types: `device_t`, `devclass_t`, `driver_t`.

**Header.** `/usr/src/sys/sys/module.h` and `/usr/src/sys/sys/bus.h`.

**Caveats.**

- `DRIVER_MODULE` expands into a module event handler; do not declare your own `module_event_t` table by hand unless you know exactly why.
- `MODULE_DEPEND` is how you make the loader bring in your prerequisites. Forgetting it produces ugly symbol-resolution failures at load time.
- `DEVMETHOD_END` terminates the method table. Without it the kernel will walk past the end.
- `device_t` is opaque; use accessors such as `device_get_softc`, `device_get_parent`, `device_get_name`, and `device_printf`.

**Lifecycle phase.** Declaration only. The macros expand into `module-init` and `module-fini` glue that is run on `kldload` and `kldunload`.

**Manual pages.** `DRIVER_MODULE(9)`, `MODULE_VERSION(9)`, `MODULE_DEPEND(9)`, `module(9)`, `DEVICE_PROBE(9)`, `DEVICE_ATTACH(9)`, `DEVICE_DETACH(9)`.

**Where the book teaches this.** Full treatment in Chapter 7, with the anatomy first sketched in Chapter 6.

### 39.6.2 `devclass(9)` and Device Accessors

**Purpose.** A `devclass_t` groups instances of the same driver so the kernel can find them, number them, and iterate over them. In drivers, you mostly use the accessors, not the `devclass` directly.

**Key names.**

- `device_t device_get_parent(device_t dev);`
- `void *device_get_softc(device_t dev);`
- `int device_get_unit(device_t dev);`
- `const char *device_get_nameunit(device_t dev);`
- `int device_printf(device_t dev, const char *fmt, ...);`
- `devclass_find`, `devclass_get_device`, `devclass_get_devices`, `devclass_get_count` when you truly need to walk a class.

**Header.** `/usr/src/sys/sys/bus.h`.

#### Caveats.

- `device_get_softc` assumes the `softc` was registered through the driver structure. Rolling your own `device_t`-to-state mapping is almost always wrong.
- Direct `devclass` manipulation is rare in drivers. If you find yourself reaching for it, check whether the question belongs in a bus-level interface instead.

**Manual pages.** `devclass(9)`, `device(9)`, `device_get_softc(9)`, `device_printf(9)`.

**Where the book teaches this.** Chapter 6 and Chapter 7.

### 39.6.3 `rman(9)`: Resource Manager

**Purpose.** A uniform view over MMIO regions, I/O ports, interrupt numbers, and DMA channels. Your driver asks for resources by type and RID and gets back a `struct resource *` with useful accessors.

#### Key names.

- `struct resource *bus_alloc_resource(device_t dev, int type, int *rid, rman_res_t start, rman_res_t end, rman_res_t count, u_int flags);`
- `struct resource *bus_alloc_resource_any(device_t dev, int type, int *rid, u_int flags);`
- `int bus_release_resource(device_t dev, int type, int rid, struct resource *r);`
- `int bus_activate_resource(device_t dev, int type, int rid, struct resource *r);`
- `int bus_deactivate_resource(device_t dev, int type, int rid, struct resource *r);`
- `rman_res_t rman_get_start(struct resource *r);`, `rman_get_end`, `rman_get_size`.
- `bus_space_tag_t rman_get_bustag(struct resource *r);`, `rman_get_bushandle`.
- Resource types: `SYS_RES_MEMORY`, `SYS_RES_IOPORT`, `SYS_RES_IRQ`, `SYS_RES_DRQ`.
- Flags: `RF_ACTIVE`, `RF_SHAREABLE`.

**Header.** `/usr/src/sys/sys/rman.h`.

#### Caveats.

- The `rid` parameter is a pointer and may be rewritten by the allocator. Pass the address of a real variable.
- Release every allocated resource in detach in reverse order of allocation. Leaking a resource almost always corrupts the next attach.
- `RF_ACTIVE` is the common case. Do not forget it, or you will get a handle that cannot be used with `bus_space(9)`.
- Always check the return value. A failed allocation is common on hardware with quirks.

**Lifecycle phase.** Allocation in attach; release in detach. If the driver has exotic needs, `bus_activate_resource` and `bus_deactivate_resource` can manage activation separately.

**Manual pages.** `rman(9)`, `bus_alloc_resource(9)`, `bus_release_resource(9)`, `bus_activate_resource(9)`.

**Where the book teaches this.** Chapter 16.

### 39.6.4 `bus_space(9)`: Portable Register Access

**Purpose.** Read and write device registers through a (tag, handle, offset) triple that hides whether the underlying access is memory-mapped, port-based, big-endian, little-endian, or indexed.

**Typical use in drivers.** Every MMIO or I/O port access. Do not dereference `rman_get_virtual` yourself; use `bus_space`.

**Key names.**

- Types: `bus_space_tag_t`, `bus_space_handle_t`.
- Reads: `bus_space_read_1(tag, handle, offset), _2, _4, _8`.
- Writes: `bus_space_write_1(tag, handle, offset, value), _2, _4, _8`.
- Multi-register helpers: `bus_space_read_multi_N`, `bus_space_write_multi_N`, `bus_space_read_region_N`, `bus_space_write_region_N`.
- Barriers: `bus_space_barrier(tag, handle, offset, length, flags)` with `BUS_SPACE_BARRIER_READ` and `BUS_SPACE_BARRIER_WRITE`.

**Header.** `/usr/src/sys/sys/bus.h`, with machine-specific details in `machine/bus.h`.

**Caveats.**

- Never touch device registers through a raw pointer. Portability and debugging both depend on `bus_space`.
- Barriers are not automatic. When two writes must occur in order, insert `bus_space_barrier` between them.
- The width used in `bus_space_read_N` or `bus_space_write_N` must match the register's natural size. Mismatches cause silent corruption on some architectures.

**Lifecycle phase.** Any time the driver talks to the device.

**Manual page.** `bus_space(9)`.

**Where the book teaches this.** Chapter 16.

### 39.6.5 `bus_dma(9)`: Portable DMA

**Purpose.** Describe DMA constraints with a tag, load a buffer through a map, and let the framework handle alignment, bouncing, and coherency. Required for any serious device that moves data.

**Key names.**

- `int bus_dma_tag_create(bus_dma_tag_t parent, bus_size_t alignment, bus_addr_t boundary, bus_addr_t lowaddr, bus_addr_t highaddr, bus_dma_filter_t *filtfunc, void *filtfuncarg, bus_size_t maxsize, int nsegments, bus_size_t maxsegsz, int flags, bus_dma_lock_t *lockfunc, void *lockfuncarg, bus_dma_tag_t *dmat);`

- `int bus_dma_tag_destroy(bus_dma_tag_t dmat);`
- `int bus_dmamap_create(bus_dma_tag_t dmat, int flags, bus_dmamap_t *mapp);`
- `int bus_dmamap_destroy(bus_dma_tag_t dmat, bus_dmamap_t map);`
- `int bus_dmamap_load(bus_dma_tag_t dmat, bus_dmamap_t map, void *buf, bus_size_t buflen, bus_dmamap_callback_t *callback, void *arg, int flags);`
- `void bus_dmamap_unload(bus_dma_tag_t dmat, bus_dmamap_t map);`
- `void bus_dmamap_sync(bus_dma_tag_t dmat, bus_dmamap_t map, bus_dmasync_op_t op);`
- `int bus_dmamem_alloc(bus_dma_tag_t dmat, void **vaddr, int flags, bus_dmamap_t *mapp);`
- `void bus_dmamem_free(bus_dma_tag_t dmat, void *vaddr, bus_dmamap_t map);`
- Flags: `BUS_DMA_WAITOK`, `BUS_DMA_NOWAIT`, `BUS_DMA_ALLOCNOW`, `BUS_DMA_COHERENT`, `BUS_DMA_ZERO`.
- Sync operations: `BUS_DMASYNC_PREREAD`, `BUS_DMASYNC_POSTREAD`, `BUS_DMASYNC_PREWRITE`, `BUS_DMASYNC_POSTWRITE`.

**Header.** `/usr/src/sys/sys/bus_dma.h`.

#### Caveats.

- Tags form a tree. Child tags inherit parent constraints; create them in the right order.
- `bus_dmamap_load` may complete asynchronously. Always use the callback, even for synchronous buffers.
- `bus_dmamap_sync` is not decoration. Without the right sync direction, caches and device memory will disagree.
- On platforms with IOMMUs the framework does the right thing. Do not skip it just because your development hardware is coherent.

**Lifecycle phase.** Tag creation and mapping setup in `attach`; load and sync in I/O paths; unload and destroy in `detach`.

**Manual page.** `bus_dma(9)`.

**Where the book teaches this.** Chapter 21.

## 39.6.6 Interrupt Setup

**Purpose.** Attach a filter or handler to an IRQ resource so the kernel can deliver interrupts to the driver.

#### Key names.

- `int bus_setup_intr(device_t dev, struct resource *r, int flags, driver_filter_t *filter, driver_intr_t *handler, void *arg, void **cookiep);`
- `int bus_teardown_intr(device_t dev, struct resource *r, void *cookie);`
- Flags: `INTR_TYPE_NET`, `INTR_TYPE_BIO`, `INTR_TYPE_TTY`, `INTR_TYPE_MISC`, `INTR_MPSAFE`, `INTR_EXCL`.

#### Caveats.

- Provide a filter when the fast-path decision is cheap and the driver can honour filter-context restrictions (no sleeping, no sleepable locks). Provide a handler when the work needs a thread.

- `INTR_MPSAFE` is mandatory for new drivers. Without it, the kernel serialises the handler on the Giant lock, which is almost always wrong.
- Tear down before freeing the resource. The order is: `bus_tear_down_intr`, then `bus_release_resource`.

**Lifecycle phase.** `bus_setup_intr` at the end of `attach`, after the rest of the softc is ready; `bus_tear_down_intr` at the start of `detach`, before any resource is released.

**Manual page.** `BUS_SETUP_INTR(9)`, `bus_alloc_resource(9)` for the resource side.

**Where the book teaches this.** Chapter 19, with advanced patterns in Chapter 20.

## 39.7 Device Nodes and Character-Device I/O

Once hardware is bound, drivers most often expose themselves to userland through a device node in `/dev/`. The APIs below build and tear down those nodes, and the associated switch table declares how the kernel should dispatch `read`, `write`, `ioctl`, and `poll`.

### 39.7.1 `make_dev_s(9)` and `destroy_dev(9)`

**Purpose.** Create a new device node under `/dev/`, connected to a `cdevsw` that holds the function pointers the kernel will call.

**Key names.**

- `int make_dev_s(struct make_dev_args *args, struct cdev **cdev, const char *fmt, ...);`
- `void destroy_dev(struct cdev *cdev);`
- Fields in `struct make_dev_args`: `mda_si_drv1`, `mda_devsw`, `mda_uid`, `mda_gid`, `mda_mode`, `mda_flags`, `mda_unit`.
- Legacy helper `make_dev(struct cdevsw *, int unit, uid_t, gid_t, int mode, const char *fmt, ...)` still exists but `make_dev_s` is preferred in new code.

**Header.** `/usr/src/sys/sys/conf.h`.

**Caveats.**

- Always use `make_dev_s`. The older `make_dev` swallows errors and does not let you set all the arguments.
- Set `mda_si_drv1` to the softc so the `cdev` carries a pointer back to driver state without a separate lookup.
- `destroy_dev` waits for all active threads to leave the `cdev` before returning, making it safe to free the softc afterwards.

**Lifecycle phase.** `make_dev_s` at the end of `attach`; `destroy_dev` at the start of `detach`, before any backing state is torn down.

**Manual page.** `make_dev(9)`.

**Where the book teaches this.** Chapter 8.



### 39.7.2 cdevsw: Character-Device Switch Table

**Purpose.** Declare the entry points the kernel should call when a process opens, reads, writes, or otherwise interacts with the device node.

**Key names.**

- struct cdevsw fields: d\_version, d\_flags, d\_name, d\_open, d\_close, d\_read, d\_write, d\_ioctl, d\_poll, d\_kqfilter, d\_mmap, d\_mmap\_single.
- d\_version must be D\_VERSION.
- Common flags: D\_NEEDGIANT (legacy), D\_TRACKCLOSE, D\_MEM.

**Header.** /usr/src/sys/sys/conf.h.

**Caveats.**

- Always set d\_version = D\_VERSION. The kernel refuses to attach a switch table with a missing or stale version.
- d\_flags default of zero is fine for modern MPSAFE drivers. Do not add D\_NEEDGIANT unless you genuinely need it.
- Unused entries can be left as NULL; the kernel substitutes defaults. Do not point them at stubs that do nothing.

**Lifecycle phase.** Declared statically at module scope. Referenced by struct make\_dev\_args.

**Manual page.** make\_dev(9) covers the structure in the context of make\_dev\_s.

**Where the book teaches this.** Chapter 8.

### 39.7.3 ioctl(9) Dispatch

**Purpose.** Provide out-of-band commands to a device, addressed by a numeric command and an argument buffer.

**Key names.**

- Entry point: d\_ioctl\_t with signature `int (*)(struct cdev *, u_long cmd, caddr_t data, int fflag, struct thread *td);`
- Command encoding macros: \_IO, \_IOR, \_IOW, \_IOWR.
- Copy helpers: copyin(9) and copyout(9) for pointer-carrying ioctls.

**Caveats.**

- Use \_IOR, \_IOW, or \_IOWR to declare commands. They encode size and direction, which matters for cross-architecture compatibility.
- Validate command arguments before acting on them. An ioctl is a trust boundary.
- Never dereference a user pointer directly. Use copyin(9) and copyout(9).

**Lifecycle phase.** Normal operation.

**Manual page.** ioctl(9) (conceptual); the entry point is documented alongside cdevsw in make\_dev(9).

**Where the book teaches this.** Chapter 24 (Section 3), with further patterns in Chapter 25.

### 39.7.4 `devfs_set_cdevpriv(9)`: Per-Open State

**Purpose.** Attach a driver-private pointer to an open file descriptor. The pointer is freed by a callback when the last close occurs.

**Key names.**

- `int devfs_set_cdevpriv(void *priv, d_priv_dtor_t *dtor);`
- `int devfs_get_cdevpriv(void **datap);`
- `void devfs_clear_cdevpriv(void);`

**Header.** `/usr/src/sys/sys/conf.h`.

**Caveats.**

- Per-open state is the right tool for per-descriptor settings, cursors, or pending transactions. Do not store per-open state in the softc.
- The destructor runs in the context of the last close. Keep it short and non-blocking.

**Manual page.** `devfs_set_cdevpriv(9)`.

**Where the book teaches this.** Chapter 8.

## 39.8 Process and User-Space Interaction

Userland cannot be trusted with kernel addresses, and the kernel cannot be trusted to follow userland pointers without care. The APIs below cross the trust boundary safely.

### 39.8.1 `copyin(9)`, `copyout(9)`, `copyinstr(9)`

**Purpose.** Move bytes between kernel and user address spaces with address validation. These are the only safe way to touch a user pointer from kernel code.

**Key names.**

- `int copyin(const void *uaddr, void *kaddr, size_t len);`
- `int copyout(const void *kaddr, void *uaddr, size_t len);`
- `int copyinstr(const void *uaddr, void *kaddr, size_t len, size_t *done);`
- Related: `fuword`, `fuword`, `subyte`, `suword`, documented under `fetch(9)` and `store(9)`.

**Header.** `/usr/src/sys/sys/system.h`.

**Caveats.**

- All three may sleep. Do not call them while holding a non-sleepable mutex.
- They return `EFAULT` on a bad address, not zero. Always check the return value.
- `copyinstr` distinguishes truncation from success via its `done` argument; do not ignore it.

**Lifecycle phase.** `d_ioctl`, `d_read`, `d_write`, and anywhere else userland is the source or destination.

**Manual pages.** `copy(9)`, `fetch(9)`, `store(9)`.

**Where the book teaches this.** Chapter 9.

### 39.8.2 uio(9): I/O Descriptor for Read and Write

**Purpose.** The kernel's own description of an I/O request. Hides the difference between user and kernel buffers, between scatter-gather and contiguous transfers, and between read and write directions.

**Key names.**

- `int uiomove(void *cp, int n, struct uio *uio);`
- `int uiomove_nofault(void *cp, int n, struct uio *uio);`
- Fields in `struct uio`: `uio_iov`, `uio_iovcnt`, `uio_offset`, `uio_resid`, `uio_segflg`, `uio_rw`, `uio_td`.
- Segment flags: `UIO_USERSPACE`, `UIO_SYSSPACE`, `UIO_NOCOPY`.
- Direction: `UIO_READ`, `UIO_WRITE`.

**Header.** `/usr/src/sys/sys/uio.h`.

**Caveats.**

- Use `uiomove` in `d_read` and `d_write` entry points. It is the right tool even when the userland buffer is a simple contiguous region.
- `uiomove` may sleep. Drop non-sleepable mutexes before calling it.
- After `uiomove` returns, `uio_resid` has been updated. Do not maintain your own byte count in parallel; read it from `uio_resid`.

**Lifecycle phase.** Normal I/O.

**Manual page.** `uio(9)`.

**Where the book teaches this.** Chapter 9.

### 39.8.3 proc(9) and Thread Context for Drivers

**Purpose.** Access to the calling thread and its process, primarily for credential checks, signal state, and diagnostic printing.

**Key names.**

- `curthread`, `curproc`, `curthread->td_proc`.
- `struct ucred *cred = curthread->td_ucred;`
- `int priv_check(struct thread *td, int priv);`
- `pid_t pid = curproc->p_pid;`

**Header.** `/usr/src/sys/sys/proc.h`.

**Caveats.**

- Direct use of process internals is rare. When you need it, it is usually for a credential check, which should go through `priv_check(9)`.
- Do not store `curthread` across sleeps. The thread that re-enters the driver may be a different one.

**Manual page.** No single page; see `priv(9)` and `proc(9)`.

**Where the book teaches this.** Referenced in Chapter 9 and Chapter 24 when `ioctl` handlers need credentials.

## 39.9 Observability and Notification

A driver that cannot be observed is a driver that cannot be trusted. The kernel provides several ways for userland to peek at driver state, subscribe to events, and wait for readiness. The APIs below are the most common.

### 39.9.1 `sysctl(9)`: Read-and-Write Configuration Nodes

**Purpose.** Publish driver state and tunables under a hierarchical name so tools like `sysctl(8)` and monitoring scripts can read or modify them.

**Key names.**

- Static declarations: `SYSCTL_NODE`, `SYSCTL_INT`, `SYSCTL_LONG`, `SYSCTL_STRING`, `SYSCTL_PROC`, `SYSCTL_OPAQUE`.
- Dynamic context API: `sysctl_ctx_init`, `sysctl_ctx_free`, `SYSCTL_ADD_NODE`, `SYSCTL_ADD_INT`, `SYSCTL_ADD_PROC`.
- Handler helpers: `sysctl_handle_int`, `sysctl_handle_long`, `sysctl_handle_string`.
- Access flags: `CTLFLAG_RD`, `CTLFLAG_RW`, `CTLFLAG_TUN`, `CTLFLAG_STATS`, `CTLTYPE_INT`, `CTLTYPE_STRING`.

**Header.** `/usr/src/sys/sys/sysctl.h`.

**Caveats.**

- For anything tied to a specific device instance, use the dynamic API. `device_get_sysctl_ctx` and `device_get_sysctl_tree` give you the right context.
- Handlers run in user context. They may sleep and may fail.
- Publish tunables sparingly. Every knob is a contract with future users.

**Lifecycle phase.** Static declarations are module-wide. Dynamic declarations are created in attach and destroyed automatically in detach through the context.

**Manual pages.** `sysctl(9)`, `sysctl_add_oid(9)`, `sysctl_ctx_init(9)`.

**Where the book teaches this.** Introduced in Chapter 7, with deeper treatment in Chapter 24 (Section 4) when the driver starts exposing metrics to userland.

### 39.9.2 `eventhandler(9)`: In-Kernel Publish-Subscribe

**Purpose.** Register for kernel-wide events such as mount, unmount, low memory, and shutdown. The kernel invokes registered callbacks in response.

**Key names.**

- `EVENTHANDLER_DECLARE(name, type_t);`
- `eventhandler_tag EVENTHANDLER_REGISTER(name, func, arg, priority);`
- `void EVENTHANDLER_DEREGISTER(name, tag);`
- `void EVENTHANDLER_INVOKE(name, ...);`
- Priority constants: `EVENTHANDLER_PRI_FIRST`, `EVENTHANDLER_PRI_ANY`, `EVENTHANDLER_PRI_LAST`.

**Header.** `/usr/src/sys/sys/eventhandler.h`.

**Caveats.**

- Handlers run synchronously. Keep them short.
- Always deregister before module unload. A dangling handler will panic when the event fires.

**Manual page.** `EVENTHANDLER(9)`.

**Where the book teaches this.** Referenced in Chapter 24 when drivers integrate with kernel-wide notifications such as shutdown and low-memory events.

### 39.9.3 `poll(2)` and `kqueue(2)`: Readiness Notification

**Purpose.** Let userland wait for driver-owned readiness events. `poll(2)` is the older interface; `kqueue(2)` is the modern one with richer filters.

**Key names.**

- Entry point for `poll`: `int (*d_poll)(struct cdev *, int events, struct thread *)`;
- Entry point for `kqueue`: `int (*d_kqfilter)(struct cdev *, struct knote *)`;
- Wait-list management: `struct selinfo`, `selrecord(struct thread *td, struct selinfo *sip)`, `selwakeup(struct selinfo *sip)`.
- `kqueue` support: `struct knote`, `knote_enqueue`, `knlist_init_mtx`, `knlist_add`, `knlist_remove`.
- Event bits: `POLLIN`, `POLLOUT`, `POLLERR`, `POLLHUP` for `poll`; `EVFILT_READ`, `EVFILT_WRITE` for `kqueue`.

**Headers.** `/usr/src/sys/sys/selinfo.h`, `/usr/src/sys/sys/event.h`, `/usr/src/sys/sys/poll.h`.

**Caveats.**

- `d_poll` must call `selrecord` when no events are ready and report current readiness when they are.
- `selwakeup` must be called with no mutex held that might invert against the scheduler. This is a common lock-order bug.
- `kqueue` support is richer but also more code. When the driver already has a clean `poll` path, extending it to `kqueue` is often the right next step rather than a rewrite.

**Lifecycle phase.** Setup in `attach`; tear down in `detach`; actual dispatch in `d_poll` or `d_kqfilter`.

**Manual pages.** `selrecord(9)`, `kqueue(9)`, and the userland pages `poll(2)` and `kqueue(2)`.

**Where the book teaches this.** Chapter 10 introduces `poll(2)` integration in full; `kqueue(2)` is referenced there and explored in depth in Chapter 35.

## 39.10 Diagnostics, Logging, and Tracing

Driver correctness does not just live in the code. It lives in the ability to observe, assert, and trace. The APIs below are how you make a driver tell the truth about itself.

### 39.10.1 `log(9)` and `printf(9)`

**Purpose.** Emit messages to the kernel log so they appear in `dmesg` and in `/var/log/messages`.

**Key names.**

- `void log(int level, const char *fmt, ...);`
- Standard kernel `printf` family: `printf`, `vprintf`, `uprntf`, `tprintf`.
- Per-device helper: `device_printf(device_t dev, const char *fmt, ...);`
- Priority constants from `syslog.h`: `LOG_EMERG`, `LOG_ALERT`, `LOG_CRIT`, `LOG_ERR`, `LOG_WARNING`, `LOG_NOTICE`, `LOG_INFO`, `LOG_DEBUG`.

**Headers.** `/usr/src/sys/sys/systm.h`, `/usr/src/sys/sys/syslog.h`.

**Caveats.**

- Do not log at `LOG_INFO` on the I/O fast path. It floods the console and masks real issues.
- `device_printf` automatically prepends the device name, which makes logs easy to filter. Prefer it over bare `printf`.
- Log once for each distinct event class, not once per packet.

**Lifecycle phase.** Any.

**Manual pages.** `printf(9)`.

**Where the book teaches this.** Chapter 23.

### 39.10.2 `KASSERT(9)`: Kernel Assertions

**Purpose.** Declare invariants that must be true. When the kernel is built with `INVARIANTS`, a violated assertion panics with a descriptive message. Without `INVARIANTS`, the assertion compiles away.

**Key names.**

- `KASSERT(expression, (format, args...));`
- `MPASS(expression);` for simpler message-free assertions.
- `CTASSERT(expression);` for compile-time assertions on constants.

**Header.** `/usr/src/sys/sys/kassert.h`, included transitively by `/usr/src/sys/sys/systm.h`.

**Caveats.**

- The expression must be cheap and free of side effects. The compiler does not optimise it into place; you write the invariant.
- The message is a parenthesised `printf` argument list. Include enough context to diagnose a failure from the panic alone.
- Use `KASSERT` for conditions that indicate a programmer error, not for normal runtime conditions.

**Lifecycle phase.** Wherever an invariant must be documented and enforced.

**Manual page.** `KASSERT(9)`.

**Where the book teaches this.** Chapter 23 introduces `INVARIANTS` and assertion use; Chapter 34 Section 2 treats `KASSERT` and diagnostic macros in depth.

### 39.10.3 WITNESS: Lock-Order Verifier

**Purpose.** A kernel option that tracks the order in which each thread acquires locks and warns when a later thread reverses a previously observed order.

**Key names.**

- Built into `mtx(9)`, `sx(9)`, `rm(9)`, and the locking macros. No separate API call is needed.
- Kernel options: `WITNESS`, `WITNESS_SKIPSPIN`, `WITNESS_COUNT`.
- Assertions that cooperate with `WITNESS`: `mtx_assert`, `sx_assert`, `rm_assert`.

**Caveats.**

- `WITNESS` is a debugging option. Build a debug kernel to enable it; it is too expensive for production.
- Warnings are not noise. If `WITNESS` complains, there is a bug.
- Lock-order warnings reference the lock names passed to `mtx_init`, `sx_init`, and so on. Give each lock a meaningful name.

**Manual page.** No single page. See `lock(9)` and `locking(9)`.

**Where the book teaches this.** Chapter 12 (Section 6), with reinforcement in Chapter 23.

### 39.10.4 ktr(9): Kernel Trace Facility

**Purpose.** A low-overhead ring buffer for event tracing inside the kernel. `ktr` records are emitted by macros and can be dumped with `ktrdump(8)`.

**Key names.**

- `CTR0(class, fmt)`, `CTR1(class, fmt, a1)`, up to `CTR6` with increasing argument counts.
- Trace classes: `KTR_GEN`, `KTR_NET`, `KTR_DEV`, and many others in `sys/ktr_class.h`.
- Kernel option: `KTR` with per-class masks.

**Header.** `/usr/src/sys/sys/ktr.h`.

**Caveats.**

- `ktr` must be enabled at kernel build time; check for `KTR` in the configuration.
- Each record is small. Do not try to log entire structures.
- For user-facing diagnostics, `dttrace(1)` is often a better answer.

**Manual page.** `ktr(9)`.

**Where the book teaches this.** Chapter 23.

### 39.10.5 DTrace Static Probes and Major Providers

**Purpose.** Static and dynamic tracing infrastructure that lets userland attach to probe points in the running kernel without a recompile.

**Key names.**

- Statically defined tracing: `SDT_PROVIDER_DECLARE`, `SDT_PROVIDER_DEFINE`, `SDT_PROBE_DECLARE`, `SDT_PROBE_DEFINE`, `SDT_PROBE`.
- Common providers on FreeBSD: `sched`, `proc`, `io`, `vfs`, `fbt` (function boundary tracing), `sdt`.
- Headers: `/usr/src/sys/sdt.h`, `/usr/src/sys/cddl/dev/dtrace/...`

#### Caveats.

- `fbt` requires no change to the driver, but `sdt` probes give you named, stable points that survive future refactoring.
- A disabled probe has negligible cost. Do not worry about adding several of them.
- DTrace scripts themselves are userland code; the driver only defines the probe points that the scripts can attach to.

**Manual pages.** `SDT(9)`, `dtrace(1)`, `dtrace(8)`.

**Where the book teaches this.** Chapter 23.

## 39.11 Cross-Reference by Driver Lifecycle Phase

The same APIs show up in different phases of a driver's life. The table below is a quick reverse index: when you are writing a particular phase, here are the families that usually belong there.

### 39.11.1 Module Load

- `MODULE_VERSION`, `MODULE_DEPEND`, `DEV_MODULE` (if the module is a pure cdev).
- Static `MALLOC_DEFINE`, `SYSCTL_NODE`, `SDT_PROVIDER_DEFINE` declarations.
- Event-handler registration that must survive before any device attaches.

### 39.11.2 Probe

- `device_get_parent`, `device_get_nameunit`, `device_printf`.
- Return values: `BUS_PROBE_DEFAULT`, `BUS_PROBE_GENERIC`, `BUS_PROBE_SPECIFIC`, `BUS_PROBE_LOW_PRIORITY`, `ENXIO` on no match.

### 39.11.3 Attach

- `device_get_softc`, `malloc(9)` for `softc` fields, `MALLOC_DEFINE` tags.
- Locking initialisation: `mtx_init`, `sx_init`, `rm_init`, `cv_init`, `sema_init`.
- Resource allocation: `bus_alloc_resource` or `bus_alloc_resource_any`.
- `bus_space` setup through `rman_get_bustag` and `rman_get_bushandle`.
- DMA scaffolding: `bus_dma_tag_create`, `bus_dmamap_create`.
- `callout_init`, `TASK_INIT`, taskqueue creation when needed.
- Interrupt setup: `bus_setup_intr`.
- Device node creation: `make_dev_s`.
- `sysctl` tree: `device_get_sysctl_ctx`, `SYSCTL_ADD_*`.



- `uma_zcreate` for high-frequency objects.
- Eventhandler registration tied to this driver.

### 39.11.4 Normal Operation

- `d_open`, `d_close`, `d_read`, `d_write`, `d_ioctl`, `d_poll`, `d_kqfilter`.
- `uiomove`, `copyin`, `copyout`, `copyinstr`.
- `bus_space_read_*`, `bus_space_write_*`, `bus_space_barrier`.
- `bus_dmamap_load`, `bus_dmamap_sync`, `bus_dmamap_unload`.
- Locking: `mtx_lock`, `mtx_unlock`, `sx_slock`, `sx_xlock`, `cv_wait`, `cv_signal`, `atomic_*`.
- Deferred work: `callout_reset`, `taskqueue_enqueue`.
- Diagnostics: `device_printf`, `log`, `KASSERT`, `SDT_PROBE`.

### 39.11.5 Detach

- Tear-down in reverse attach order.
- `bus_tearardown_intr` before any resource is released.
- `destroy_dev` before the softc fields it references are torn down.
- `callout_drain` before freeing the callout structure.
- `taskqueue_drain_all` and `taskqueue_free` for private taskqueues.
- `bus_dmamap_unload`, `bus_dmamap_destroy`, `bus_dma_tag_destroy`.
- `bus_release_resource` for every resource allocated in attach.
- `cv_destroy`, `sx_destroy`, `mtx_destroy`, `rm_destroy`, `sema_destroy`.
- `uma_zdestroy` for every zone the driver owns.
- Eventhandler deregistration.
- Final `free` or `contigfree` of anything allocated.

### 39.11.6 Module Unload

- Verify that no device instance is still attached. Newbus usually handles this, but defensive `DRIVER_MODULE` event handlers should refuse unload if state remains.

## 39.12 Quick-Reference Checklists

These checklists are meant to be read in five minutes or less. They do not replace the teaching in the chapters; they remind you of the things that experienced driver authors no longer forget.

### 39.12.1 Locking Discipline Checklist

- Every shared field in the softc has exactly one lock that protects it, documented in a comment near the field.
- No mutex is held across `uiomove`, `copyin`, `copyout`, `malloc(9, M_WAITOK)`, or `bus_alloc_resource`.
- Lock order is declared in a comment at the top of the file and respected everywhere.

- `mtx_assert` or `sx_assert` appears on functions that require a particular lock to be held on entry.
- `WITNESS` is enabled in the development kernel and its warnings are treated as bugs.
- Every `mtx_init` has a matching `mtx_destroy`, and so on for every lock type.

### 39.12.2 Resource Lifetime Checklist

- `bus_setup_intr` is the last thing in `attach`; `bus_tearardown_intr` is the first thing in `detach`.
- Every allocated resource has a matching release, in reverse order, in `detach`.
- `callout_drain` is called before the structure it points at is freed.
- `taskqueue_drain_all` or `taskqueue_drain` is called before the task structures or their arguments are freed.
- `destroy_dev` is called before the `softc` fields referenced by `mda_si_drv1` are torn down.

### 39.12.3 User-Space Safety Checklist

- No user pointer is dereferenced directly. Every cross-boundary access goes through `copyin`, `copyout`, `copyinstr`, or `uiomove`.
- All return values from the copy helpers are checked. `EFAULT` is propagated rather than ignored.
- `_IOR`, `_IOW`, and `_IOWR` are used for `ioctl` command numbers.
- `Ioctl` handlers validate arguments before acting on them.
- Credentials are checked with `priv_check(9)` when the operation is privileged.

### 39.12.4 Diagnostic Coverage Checklist

- Every major branch that should never be taken carries a `KASSERT`.
- Logs use `device_printf` for instance context.
- At least one DTrace SDT probe marks entry to the main I/O paths.
- `sysctl` exposes the driver's counters in a stable, documented tree.
- The driver has been built and exercised under `INVARIANTS` and `WITNESS` before it is considered done.

## 39.13 Wrapping Up

This appendix is a reference, not a chapter. It becomes more useful the more you use it. Keep it within reach while you are writing, debugging, or reading driver code, and turn to it whenever you want a fast reminder about a flag, a manual page, or a caveat you almost remember.

Three suggestions for getting the most out of it over time.

First, treat the **Manual pages** line as the canonical next step for any API you only half remember. The manual pages in section 9 are maintained with the tree; they age well. Opening one of them costs nothing and pays back every time.

Second, treat the **Caveats** line as a debugging companion. Most driver bugs are not unknown unknowns. They are documented caveats that the author skipped under time pressure. When you are stuck, read the caveats for every API the problem area touches. It is unglamorous but effective.

Third, when you find a missing entry or a correction to make, write it down. This appendix improves as drivers improve. The FreeBSD kernel is alive and so is the reference. If a new primitive appears, or an old one is retired, the appendix that matches reality is the one you will actually trust.

From here you can jump in several directions. Appendix E covers FreeBSD internals and subsystem behaviour at a depth this reference deliberately avoids. Appendix B collects the algorithms and systems-programming patterns that recur throughout the kernel. Appendix C grounds the hardware concepts that the bus and DMA families depend on. And every chapter in the main book still has source code you can read, labs you can run, and questions you can answer by opening `/usr/src/` and looking at the real thing.

Good reference material is quiet. It stays out of the way while you work, and it is there when you need it. That is the role this appendix is meant to play for the rest of your driver-writing life with FreeBSD.

## 40 Appendix B: Algorithms and Logic for Systems Programming

### 40.1 How to Use This Appendix

The main chapters teach the primitives a driver author uses day to day. Alongside those primitives there is a second layer of knowledge that the book assumes without always naming it: the small catalogue of data structures, control-flow shapes, and reasoning patterns that show up in nearly every driver you will read. A red-black tree in a virtual-memory subsystem, a STAILQ of pending requests on the side of a softc, a goto out ladder that unwinds an attach routine, a switch on a state enum that drives a protocol handshake. None of these ideas is hard in isolation. What makes them feel hard is meeting them for the first time embedded in real kernel code, where the pattern is implicit and the name is nowhere on the page.

This appendix is that missing name. It is a short, practical field guide to the algorithmic patterns that recur across FreeBSD kernel and driver code, organised so that you can recognise a pattern when you see it and decide which pattern fits when you have to write one. It is not a computer-science textbook, and it is not a replacement for the chapters that already teach the specific pieces you will use most. What it provides is a middle layer: enough pattern recognition to let you read driver code confidently, enough mental modelling to pick the right structure when the blank page stares back at you, and enough caveat awareness to avoid the mistakes that every driver author eventually makes once.

#### 40.1.1 What You Will Find Here

The appendix is organised by problem family, not by abstract taxonomy. Each family collects a few related patterns with a consistent short structure:

- **What it is.** One or two sentences.
- **Why drivers use it.** The concrete role in kernel or driver code.
- **Use this when.** A compact decision hint.
- **Avoid this when.** The other side of the same decision.
- **Core operations.** The handful of names you need in muscle memory.
- **Common traps.** The mistakes that actually cost people time.
- **Where the book teaches it.** A pointer back to the main chapters.
- **What to read next.** A manual page or a realistic source-tree example.

When a pattern has a real FreeBSD implementation with a stable interface, the entry uses those names directly. When the pattern is generic, the entry still grounds itself in how FreeBSD drivers actually apply it.

### 40.1.2 What This Appendix Is Not

It is not a first course in algorithms. If you have never met a linked list or a state machine before, this appendix will feel too compressed; read Chapters 4 and 5 first, then come back. It is not a deep theoretical reference either. You will find almost no asymptotic analysis here, because driver authors rarely choose between  $O(\log n)$  and  $O(n)$  on paper. They choose between patterns whose trade-offs are mostly about locking, cache behaviour, and invariants. Finally, it is not a replacement for Appendix A (which covers the APIs), Appendix C (which covers hardware concepts), or Appendix E (which covers kernel subsystems). If the pattern you want points you there, go there and come back.

## 40.2 Reader Guidance

Three ways to read this appendix, each calling for a different strategy.

If you are **writing new code**, skim the problem family that matches your situation, read the one or two entries that apply, glance at the **Common traps** line, and close the appendix. Thirty seconds is enough. Start from the pattern, not from the code.

If you are **reading someone else's driver**, treat the appendix as a translator. When you see an unfamiliar idiom, find the family that names it, read **What it is** and **Core operations**, and keep moving. Full understanding can come later; right now the goal is to form a mental model of what the driver is doing.

If you are **debugging**, read the **Common traps** lines in the pattern family around the bug. Most driver bugs that look mysterious are ordinary caveats the author did not honour. A ring buffer with a stale full check, an unwind ladder that frees in the wrong order, a state machine that forgot a transient state. The traps in this appendix are not exhaustive, but they cover the ones that recur.

A few conventions apply throughout:

- Source paths are shown in the book-facing form, `/usr/src/sys/...`, matching a standard FreeBSD system.
- Manual pages are cited in the usual FreeBSD style. For the data-structure macros, the authoritative pages live in section 3: `queue(3)`, `tree(3)`, `bitstring(3)`. For kernel services such as `hashinit(9)`, `buf_ring(9)`, `refcount(9)`, the pages live in section 9. The distinction matters only when you type `man` at the prompt.
- When an entry points to real source as a reading example, it points to a file that a beginner can open and navigate. Larger subsystems exist that also use the pattern; those are mentioned only when they are the canonical place to see a pattern in practice.

With that in mind, we start with the structures a driver uses to hold state.

## 40.3 Data Structures in the Kernel

A driver without state is rare. Almost every non-trivial driver keeps a collection of something: pending requests, open handles, per-CPU statistics, per-channel configuration, per-client contexts. The question is which collection shape matches the access pattern. The kernel ships a small toolkit of header-file data structures that you

should reach for by default instead of rolling your own. Each of them solves a different problem; confusing them is one of the most common category errors in driver design.

### 40.3.1 The `<sys/queue.h>` Family

**What it is.** A family of intrusive linked-list macros. You define a head and an entry, embed the entry inside your element structure, and the macros give you insertion, removal, and traversal without heap allocation for list nodes.

The header defines four flavours:

- `SLIST` is a singly linked list. Cheapest, forward only,  $O(n)$  removal of arbitrary elements.
- `LIST` is a doubly linked list. Forward and backward traversal,  $O(1)$  removal of an arbitrary element.
- `STAILQ` is a singly linked tail queue. Forward only, but with fast insertion at the tail. Classic FIFO.
- `TAILQ` is a doubly linked tail queue. Every operation you can reasonably want, at the cost of two pointers per element.

**Why drivers use them.** Almost every per-driver collection in the tree is one of these. Pending commands, open file handles, registered callbacks, child devices. They are predictable, allocation-free at the list level, and the macros make the code read almost like pseudocode.

**Use this when** you have a collection of objects you already allocate yourself, and you want standard list operations without inventing them. The intrusive design means an object can live in several lists at once; that is a feature, not a bug.

**Avoid this when** you need ordered lookup by key. A list is  $O(n)$  to search. Reach for a tree or a hash table instead.

#### Choosing among the four.

- Start with `TAILQ` unless you have a reason not to. It is the most common and the most flexible.
- Drop to `STAILQ` when you know you only ever push and pop like a queue. You save one pointer per element.
- Use `LIST` when you want doubly linked behaviour but do not need a tail pointer. Unordered sets of independent elements, for example.
- Use `SLIST` when every byte counts and the list is short or traversal is linear anyway. Fast paths in hot structures sometimes justify it.

**Core operations.** The macros are uniform across flavours. Insertion: `TAILQ_INSERT_HEAD`, `TAILQ_INSERT_TAIL`, `TAILQ_INSERT_BEFORE`, `TAILQ_INSERT_AFTER`. Removal: `TAILQ_REMOVE`. Traversal: `TAILQ_FOREACH`, plus the `_SAFE` variant when you may remove the current element. The same operations exist with `LIST_`, `STAILQ_`, and `SLIST_` prefixes. Field storage is an embedded `TAILQ_ENTRY(type)` in your struct.

#### Common traps.

- Using the non-`_SAFE` foreach while removing elements. Plain `TAILQ_FOREACH(var, head, field)` expands to a for loop that reads `TAILQ_NEXT(var, field)` at the end of each iteration. If the body freed `var`, the next step dereferences freed memory. `TAILQ_FOREACH_SAFE(var, head, field, tmp)` caches the next pointer into `tmp` before the body runs, and is always the right answer when the loop body may remove `var`.

- Forgetting that the list is intrusive. The same element cannot be in two TAILQs through the same entry field at once. If you need that, define two TAILQ\_ENTRY fields with different names.
- Mixing flavours on the same head. SLIST\_INSERT\_HEAD on a TAILQ\_HEAD compiles but produces silent corruption. Keep the macros consistent with the head type.

**Where the book teaches this.** Introduced briefly in Chapter 5 as part of the kernel dialect of C and revisited whenever a driver in later parts of the book needs an internal collection.

**What to read next.** `queue(3)` for the full macro catalogue, and any real driver that keeps pending work. `/usr/src/sys/net/if_tuntap.c` uses these idioms extensively.

### 40.3.2 Red-Black Trees via `<sys/tree.h>`

**What it is.** A set of macros that generate a self-balancing binary search tree inlined into your types. FreeBSD's implementation is rank-balanced, but it exposes the historical RB\_ prefix and behaves like a red-black tree for all practical purposes.

**Why drivers use them.** When you need  $O(\log n)$  ordered lookup by a key and the key is not a small integer, a red-black tree is the default. Virtual-memory maps, scheduler run-queues in some subsystems, and per-subsystem registries of named objects all use this header.

**Use this when** you have a collection of objects indexed by a key, the set grows to hundreds or thousands of entries, and you care about ordered traversal as well as lookup.

**Avoid this when** the set is small (a dozen elements) or the key space is small and dense. A linear scan on a cache-hot array usually beats a tree in both cases, because the constant factor of a pointer chase is much higher than a sequential read.

**Core operations.** `RB_HEAD(name, type)` defines the head type, `RB_ENTRY(type)` defines the embedded node field, `RB_PROTOTYPE` and `RB_GENERATE` instantiate the function family (insert, remove, find, min, max, next, prev, foreach). You also write a comparison function that returns negative, zero, or positive the way `strcmp(3)` does. A variant `SPLAY_` family in the same header gives splay trees when locality of reference is important.

#### Common traps.

- Calling `RB_GENERATE` in a header. That produces multiple definitions at link time. Keep `RB_GENERATE` in exactly one .c file and `RB_PROTOTYPE` in the header that declares the type.
- Forgetting that the tree is intrusive. An element cannot live in two trees through the same entry, and removing from the wrong tree silently corrupts both.
- Holding a pointer across an insert. Balance rotations do not move nodes, but the comparator must be consistent; if your key changes while the node is in the tree, you have invalidated the search structure and every subsequent lookup is undefined.

**Where the book teaches this.** Not taught as a dedicated pattern in the book. The reader meets it through example when Appendix E walks the virtual-memory subsystem and when advanced drivers in Part 7 reach for ordered lookup.

**What to read next.** `tree(3)` for the macro catalogue, and any subsystem that keeps an ordered set of named things.

### 40.3.3 Radix Trees via `<net/radix.h>`

**What it is.** A kernel-specific radix (or Patricia) tree, primarily used by the routing table to match addresses against prefixes. A radix tree differs from a red-black tree in that it keys on prefix length rather than total order, which is precisely what longest-prefix-match needs. The header lives at `/usr/src/sys/net/radix.h`, alongside the rest of the networking code, rather than under `/usr/src/sys/sys/`.

**Why drivers use them.** Rarely, directly. The radix tree exists because the networking stack needs it, and the machinery is specialised to that use. A driver almost never creates its own radix tree. You may meet one as a caller in network-adjacent code, or you may see the related LinuxKPI radix-tree shim in compatibility layers.

**Use this when** you genuinely need prefix matching over keys of varying length. Routing, ACL tables, prefix-based policy.

**Avoid this when** what you actually need is ordered lookup by exact key. Use `<sys/tree.h>` instead.

**Core operations.** `rn_inithead`, `rn_addroute`, `rn_delete`, `rn_lookup`, `rn_match`, `rn_walktree`. These are specialised enough that the routing subsystem is the canonical reference.

**Common traps.** The header and the routines are tightly coupled to the routing-table abstraction. Attempting to reuse the radix tree for non-routing data structures is almost always a mistake; the abstraction will leak in uncomfortable ways. The LinuxKPI radix tree in `/usr/src/sys/compat/linuxkpi/common/include/linux/radix-tree.h` is a different structure and should not be confused with the native one.

**Where the book teaches this.** Only mentioned in Appendix E alongside the routing table. This appendix names it so you can recognise it in passing.

**What to read next.** The routing-table code under `/usr/src/sys/net/`.

### 40.3.4 Bitmaps and Bit-Strings via `<sys/bitstring.h>`

**What it is.** A compact set-of-bits abstraction backed by an array of machine words. You allocate a bit-string of *N* bits, then set, clear, test, and scan individual bits.

**Why drivers use them.** Any time you have a dense finite universe and want to track boolean state for each element. Free-slot bitmaps for a fixed pool of hardware descriptors, interrupt allocations, minor numbers, per-channel enable flags. A bitmap uses one bit per element and supports efficient first-free searches.

**Use this when** the universe is dense, known in advance, and you track boolean state per element. Bitmaps are particularly effective when you need “find the first free slot” or “how many are set”.

**Avoid this when** the universe is sparse. A set of a few thousand used slot numbers in a 32-bit key space does not belong in a bitmap; use a hash table or a tree.

**Core operations.** `bit_alloc` (kernel allocator variant) and `bit_decl` (stack), `bit_set`, `bit_clear`, `bit_test`, `bit_nset`, `bit_nclear` for ranges, `bit_ffs` for first-set, `bit_ffc` for first-clear, `bit_foreach` for iteration. Sizes are computed with `bitstr_size(nbits)`.

**Common traps.**

- Forgetting that bit indices are zero-based and that the `bit_ffs` / `bit_ffc` macros do not return a value; they write the found index into `*_resultp` and set it to `-1` when nothing matched. Always check `if (result < 0)` before using the index.



- Race conditions under concurrent modification. `bitstring(3)` is not atomic; if two contexts can set or clear bits concurrently, they need a lock or a coordinated handoff.
- Allocating the wrong size. Use `bitstr_size(nbits)` rather than doing the division by hand.

**Where the book teaches this.** Not taught directly. A driver that needs a free-slot bitmap will reach for this header, and this entry is the pointer.

**What to read next.** `bitstring(3)`.

### 40.3.5 Hash Tables with `hashinit(9)`

**What it is.** A small helper pattern for building hash tables out of `<sys/queue.h>` lists. `hashinit(9)` allocates a power-of-two array of `LIST_HEAD` buckets and gives you back the array and a mask. Your code hashes the key to a 32-bit value, ANDs it with the mask, and walks the resulting bucket.

**Why drivers use them.** When you need unordered lookup by key and the set is large enough that a linear scan is too slow. File-system name caches, open-file tables per-process, and any driver that maintains a registry keyed by hashable identifier all use this shape.

**Use this when** the set is large, lookups are frequent, and ordering does not matter. A hash table beats a tree on average-case lookup and is easier to lock coarsely (one lock per bucket, or one lock for the whole table if writes are rare).

**Avoid this when** the set is small (a tree or a list is simpler) or you need ordered traversal (`RB_FOREACH` is the answer, not hashing).

**Core operations.** `void *hashinit(int elements, struct malloc_type *type, u_long *hashmask);` returns the bucket array and writes the mask to `*hashmask`. Each bucket is a `LIST_HEAD`, so you insert and iterate with the usual `LIST_` macros. `hashdestroy` tears it down. `hashinit_flags` takes an extra flags word (`HASH_WAITOK` or `HASH_NOWAIT`) when you need the non-sleeping variant; `hashinit` itself is equivalent to `HASH_WAITOK`. `phashinit` and `phashinit_flags` give you a prime-sized table when you prefer modulo arithmetic to masking, and write the chosen size (not a mask) through the out-pointer. A small `<sys/hash.h>` header provides basic hash functions such as `hash32_buf` and `hash32_str`, along with `jenkins_hash` and `murmur3_32_hash`; the Fowler-Noll-Vo family lives in `<sys/fnv_hash.h>`.

#### Common traps.

- Hashing a pointer directly. Pointer low bits are usually aligned and produce poor bucket distribution. Use a proper hash function on the contents or on the shifted pointer.
- Assuming `hashinit` uses your requested size. It rounds down to the nearest power of two and returns the mask for that size. A request for 100 elements produces a 64-bucket table, not 100.
- Forgetting the lock story. A bare hash table is not thread-safe. Decide up front whether you want a single lock, per-bucket locks, or something fancier, and document it in the softc.

**Where the book teaches this.** Briefly referenced in Part 7 drivers that maintain a per-client registry. This entry gives the scaffolding for when you need to build one yourself.

**What to read next.** `hashinit(9)`, and the implementation in `/usr/src/sys/kern/subr_hash.c`.

## 40.4 Circular and Ring Buffers

Rings appear everywhere in a driver that mediates between hardware and software. A NIC hands out a ring of descriptors for packet transmission. A UART buffers characters between the interrupt that receives them and the process that consumes them. A command driver batches requests into a ring for the hardware engine to pick up. The shape is always the same: a fixed-size array plus a producer index and a consumer index that wrap around. What varies is who is concurrent with whom, which primitives you use, and whether the ring needs to be lock-free.

### 40.4.1 Single-Producer Single-Consumer Ring Buffers

**What it is.** A fixed-size circular buffer where exactly one context produces and exactly one context consumes. The producer advances a head (or tail) index after writing; the consumer advances a tail (or head) index after reading; neither ever writes the other's index.

**Why drivers use them.** The interrupt handler produces data (a received character, a completion event), and a thread consumes it. That is the prototypical SPSC shape. It is lock-free by construction, because no two contexts modify the same variable.

**Use this when** you have exactly one producer and exactly one consumer, and you want to avoid a lock. Character device ring buffers are the classic case.

**Avoid this when** more than one thread can produce, or more than one thread can consume. Adding a second producer breaks the invariant; you need a lock, or you need `buf_ring(9)`.

**Core operations and invariants.** The usual invariants are:

- head advances only by the producer; tail advances only by the consumer.
- $(\text{head} - \text{tail}) \bmod \text{capacity}$  is the number of occupied slots.
- $\text{capacity} - (\text{head} - \text{tail}) \bmod \text{capacity}$  is the number of free slots (minus one, depending on the encoding).
- The buffer is empty when `head == tail`.
- The buffer is full when  $(\text{head} + 1) \bmod \text{capacity} == \text{tail}$ . The one sacrificed slot is how you distinguish full from empty.

The alternative encoding uses a separate count variable or uses free-running indices modulo the capacity with a mask (`head & mask`, `tail & mask`). Free-running indices make full-versus-empty a signed comparison and avoid the sacrificed slot.

#### Common traps.

- Mixing up full and empty without the one-slot convention. Every ring design must have a clear answer; pick one and write it at the top of the file.
- Not ordering memory properly. On modern CPUs, writes can become visible out of order. The producer must make sure the buffer contents are visible to the consumer before the updated head is. In the kernel that means an explicit memory barrier (`atomic_thread_fence_rel` on the producer, `atomic_thread_fence_acq` on the consumer), or using the atomic store-release and load-acquire variants.

- Assuming capacity is a power of two when the code does not enforce it. Masking requires a power of two; modulo does not. The code and the invariant must agree.

**Where the book teaches this.** Chapter 10 builds a circular buffer for the `myfirst` driver's character device and teaches the head/tail/full/empty discipline from first principles. Revisited in Chapters 11 and 12 when the ring meets synchronisation.

**What to read next.** The circular-buffer example from Chapter 10 in the companion `examples/` tree, and any small character-device driver that uses a ring to bridge interrupt and process context.

#### 40.4.2 `buf_ring(9)` for MPMC Rings

**What it is.** FreeBSD's kernel ring-buffer library. `buf_ring` provides a multi-producer multi-consumer ring with lock-free enqueue and both single-consumer and multi-consumer dequeue paths. It is used heavily by `iflib`, networking drivers, and elsewhere in the tree.

See Chapter 11 for the introduction and the `buf_ring(9)` man page for the authoritative concurrency contract.

**Why drivers use them.** When more than one context can enqueue (several CPUs pushing packets onto the same transmit queue) or more than one context can dequeue (several worker threads draining the same work queue), you cannot get away with the SPSC invariants. `buf_ring` does the atomic compare-and-swap work for you and hides the memory-ordering details behind a stable interface.

**Use this when** you have multiple producers or multiple consumers and you want a lock-free path. Network-adjacent drivers are the canonical case.

**Avoid this when** you truly have one producer and one consumer. The SPSC ring is simpler, cheaper, and easier to reason about.

**Core operations.** `buf_ring_alloc(count, type, flags, lock)` and `buf_ring_free` at setup and teardown. `buf_ring_enqueue(br, buf)` enqueues, returning 0 on success or `ENOBUFFS` when full. `buf_ring_dequeue_mc(br)` dequeues safely under multiple consumers; `buf_ring_dequeue_sc(br)` is faster when the caller guarantees single-consumer semantics. `buf_ring_peek`, `buf_ring_count`, `buf_ring_empty`, `buf_ring_full` give you inspection without removing.

##### Common traps.

- Calling `buf_ring_dequeue_sc` from multiple consumers at once. The SC path assumes only one consumer is running; violating that corrupts the tail index silently.
- Treating `ENOBUFFS` as a hard error. `buf_ring_enqueue` returning `ENOBUFFS` is an ordinary back-pressure signal; the caller must either retry, drop, or queue on a slower path.
- Forgetting that `buf_ring_count` and `buf_ring_empty` are advisory. Another CPU may enqueue or dequeue the moment after you check. Design the caller around eventual consistency, not instantaneous truth.

**Where the book teaches this.** Named in Chapter 10 as the production-grade counterpart to the pedagogical SPSC ring. Used in Part 5 networking chapters when driver examples need real ring semantics.

**What to read next.** `buf_ring(9)`, and the header itself at `/usr/src/sys/sys/buf_ring.h`. For a real user, browse `iflib` or any high-performance NIC driver under `/usr/src/sys/dev/`.

### 40.4.3 Wrap-Around Arithmetic and the Full-vs-Empty Problem

Every ring has to distinguish full from empty. The two states satisfy `head == tail` or are one slot apart, and the encoding you pick decides which comparison to use. The two common conventions are worth naming so you can spot them in code.

- **Sacrificed slot.** The ring holds `capacity - 1` usable slots. Empty is `head == tail`. Full is `(head + 1) mod capacity == tail`. Simple, cheap, works with any capacity.
- **Free-running indices.** `head` and `tail` count every operation since the ring was created and never wrap. The modular index for array access is `head & mask` (which requires `capacity` to be a power of two). Occupied count is `head - tail` as unsigned arithmetic; full is `occupied == capacity`; empty is `occupied == 0`. The advantage is that you never sacrifice a slot; the cost is that you must write `(uint32_t)(head - tail)` carefully.

Both conventions are correct. Pick one and make it a one-sentence comment at the top of the file. Readers who know ring buffers will know what to look for; readers who do not will have the comment.

## 40.5 Sorting and Searching

Drivers sort and search less often than they list, ring, or tree things. But when they do, a small toolkit covers almost every case, and knowing which tool is appropriate saves time.

### 40.5.1 In-Kernel `qsort` and `bsearch`

**What it is.** The kernel provides the familiar `qsort` and `bsearch` from the standard library, declared in `/usr/src/sys/sys/libkern.h`. A thread-safe variant `qsort_r` passes a user context pointer through to the comparator.

**Use this when** you need to sort an array in place once or occasionally, or when you need binary search over a sorted array. Device ID tables, compatibility matching, and sorted snapshots all fit here.

**Avoid this when** the collection is long-lived and changes frequently. In that case you want a tree, not a sort-and-search pair. Also avoid `qsort` on hot paths: the function-call overhead of the comparator dominates for small arrays.

**Core operations.** `qsort(base, nmemb, size, cmp)` and `qsort_r(base, nmemb, size, cmp, thunk)` for sorting; `bsearch(key, base, nmemb, size, cmp)` for searching. The comparator returns negative, zero, or positive in the `strcmp(3)` style. All three are declared in `libkern.h` and have section-3 manual pages (`qsort(3)`, `bsearch(3)`).

#### Common traps.

- Non-stable comparisons. `qsort` is not guaranteed stable. If ordering among equal keys matters, add a tiebreaker to the comparator.
- Passing a comparator that tests with subtraction (`a->x - b->x`) when the fields are wide. Integer overflow can make a negative look positive. Use explicit comparisons (`a->x < b->x ? -1 : a->x > b->x ? 1 : 0`).

**Where the book teaches this.** Referenced in Chapter 4 (the C tour) and in Part 7 where specific drivers sort a table once at attach.

**What to read next.** `qsort(3)`, `bsearch(3)`.

### 40.5.2 Binary Search Over Device ID Tables

**What it is.** A specialised application of `bsearch`. A driver maintains a sorted table of (vendor, device) or compatible-string keys, and the probe path does a binary search to decide whether the hardware is one it supports.

**Why drivers use it.** A PCI driver may match dozens of device IDs. A scan through the table is fine at attach time; doing it linearly works. Doing it with `bsearch` is a small, readable, fast improvement when the table grows.

**Use this when** the ID table is longer than a handful of entries and ordering by key is easy. You can sort at compile time by declaring the table statically sorted, or once at module load with `qsort`.

**Avoid this when** the match is not a simple key comparison. PCI matching sometimes involves subclass and programming interface fields; express the matching logic directly rather than contorting it into a comparator.

**Common traps.** Sorting at runtime and then relying on the sort for ABI compatibility. If another piece of code reads the table by index, changing the order breaks it. Declare the table `const` and sort at build time when you can.

**Where the book teaches this.** Referenced in Chapter 18 (PCI) and in the device-matching discussion in Chapter 6.

**What to read next.** Any PCI driver's ID table and probe path. `/usr/src/sys/dev/uart/uart_bus_pci.c` is a readable example.

### 40.5.3 When Linear Search Wins

**What it is.** A reminder that for small collections, linear search on a cache-hot array is the fastest lookup you can write. CPU caches reward sequential reads and punish pointer chasing.

**Use this when** the collection has fewer than a few dozen entries. In practice, a linear scan on twenty items is faster than any tree lookup, because the tree walk takes several cache misses while the linear scan takes one.

**Avoid this when** the collection grows without bound, or when the hot-path latency requirement is tight and the worst case matters more than the average.

**Common traps.** Optimising the wrong side. Do not introduce a tree to cure “linear search is slow” before measuring. The real cost is usually elsewhere.

## 40.6 State Machines and Protocol Handling

Many drivers are state machines in disguise. A USB device probes, attaches, goes idle, resumes, suspends, detaches. A network driver goes through link-down, link-up, negotiation, steady-state, error recovery. When the state transitions are complicated, making the state explicit pays off immediately. When they are simple, an enum and a switch often do the job.

### 40.6.1 Explicit State Enum vs Implicit Flag Bits

**What it is.** Two ways to represent state. An enum says “this device is in exactly one of N states”. A flag word says “this device has any combination of N independent booleans”.

**Use the enum when** the conditions are mutually exclusive and naming the transitions matters. A link is one of `LINK_DOWN`, `LINK_NEGOTIATING`, or `LINK_UP`. A command is one of `CMD_IDLE`, `CMD_PENDING`, `CMD_RUNNING`, or `CMD_DONE`. An enum forces every transition to be explicit; you cannot accidentally be in two states at once.

**Use flag bits when** the conditions are independent and can be combined. “Interrupt enabled”, “autonegotiation allowed”, “in promiscuous mode”. Mixing mutually exclusive and independent conditions in the same flag word is the usual mistake; separate them into an enum for the first and a flag word for the second.

**Common traps.**

- Encoding a state machine as bit flags and then having to decide what “both `CMD_PENDING` and `CMD_DONE` set” means. That never means anything useful. Switch to an enum.
- Encoding independent booleans as a single packed state. You will reinvent bitwise OR, badly.

### 40.6.2 Switch-Based FSMs

**What it is.** The simplest explicit state machine: an enum for the state, a switch in a function that takes an event, and a body per case that updates the state and performs the action.

**Use this when** the state space is small (perhaps ten states) and the transitions are shallow. A switch is easy to read, easy to extend, and the compiler catches missing cases if you enable the right warning flag.

**Avoid this when** the switch grows to hundreds of lines or the same transition logic is duplicated across several states. That is the signal to move to a table-driven FSM.

**Common traps.**

- Forgetting a state. Compile with `-switch-enum` so the compiler tells you.
- Doing non-trivial work inside the switch. Keep the switch as a dispatcher; delegate the real work to named helper functions.

**Where the book teaches this.** The pattern appears naturally in many driver examples. Chapter 5 introduces it in its simplest form.

### 40.6.3 Table-Driven FSMs

**What it is.** The state machine expressed as a two-dimensional table indexed by (state, event). Each cell holds the next state and, typically, a function pointer for the action to run on that transition.

**Use this when** the FSM has many states and many events and you want the transition logic to be data, not code. Visualising the matrix is often easier than chasing a giant switch. Protocol stacks, bus enumeration logic, and suspend-resume state machines are classic uses.

**Avoid this when** the FSM is small. A five-state switch is easier to read than a five-by-five matrix.

**Core operations.** Define the state enum, the event enum, and a transition struct containing `next_state` and `action_fn`. Declare the table as a static constant two-dimensional array. The driver steps the machine with `table[current_state][event]`.

**Common traps.**

- Null action pointers without a check. Either the table must be dense (every cell is a valid transition) or the dispatcher must treat a null action as “ignore”, and the convention must be consistent.
- Embedding locks inside the action functions without a clear discipline. The state transition is a critical section; decide whether the caller holds the lock or the dispatcher takes it, and stick to that.

#### 40.6.4 Function-Pointer Dispatch

**What it is.** A lighter alternative to a full state table. The current “mode” of the driver is a pointer to a set of handler functions; switching state is as simple as reassigning the pointer.

**Use this when** the “state” is really a set of methods the driver performs differently depending on mode. A link that comes up has a different receive path than a link that is still negotiating. A device that is booting its firmware dispatches commands differently from one that is already running.

**Avoid this when** the state transitions are ad-hoc or numerous. A function-pointer swap is a coarse operation; reserving it for a few well-named modes keeps the code honest.

**Common traps.**

- Racing with the swap. If one CPU calls through the pointer while another is replacing it, the call may dispatch to stale code. The swap needs ordering. In the kernel, that usually means a lock or an RCU-style deferred free via `epoch(9)` for any state structure the old pointer referenced.
- Orphaned resources. When you swap the function-pointer table, make sure the resources the old mode owned are handed off or released cleanly. The switch itself does not free anything.

#### 40.6.5 Reentrancy and Partial Completion

**What it is.** A state-machine design concern rather than a data structure. Real drivers receive events in interleaved order: an interrupt fires while the driver is halfway through processing an `ioctl`; a completion arrives while the command that triggered it is still being posted; a detach races with an open. The state machine has to tolerate this.

**Design principles.**

- Make every transition a single, atomic update. Read current state, compute next state, commit with a lock or an atomic. Do not perform the action before committing the state; if you crash or return early, the state must match reality.
- Separate “what to do” from “who runs it”. The state transition decides what needs to happen; a taskqueue or a worker thread actually does the work. That way the transition can finish before the expensive work starts, and a second caller sees consistent state.
- Have a “transient” state for every transition that takes time. A command that is in the act of being sent is neither `IDLE` nor `RUNNING`; it is `SENDING`. A link that is bringing itself down is neither `UP` nor `DOWN`; it is `GOING_DOWN`. Transient states give concurrent callers something to wait for or retry on.

**Common traps.** Assuming events are serialised. They are not. An interrupt can arrive at any time, a detach can race with an open, a userspace close can race with a device's internal teardown. Build the state machine so that every event is handled from every state, even if the handling is just "reject" or "defer".

**Where the book teaches this.** Revisited in Chapter 11 (synchronisation), Chapter 14 (interrupts in depth), and Part 5 when real networking drivers hit these cases hard.

## 40.7 Error-Handling Patterns

A driver that gets the happy path right but gets the error path wrong will corrupt memory, leak resources, or panic the kernel. Good error handling in driver code is almost always about structure, not cleverness. The kernel has a strong, uniform idiom for it, and this section names that idiom.

### 40.7.1 The `goto out` Idiom

**What it is.** The canonical kernel cleanup pattern. At the top of the function, declare every resource to `NULL`. At each acquisition, check the result; on failure, `goto fail_N` where `fail_N` is a label that releases exactly the resources acquired so far. At the end of the function, the success path returns; the cleanup ladder runs in reverse order of acquisition and falls through every label down to the final return.

**Why drivers use it.** Because it works. The ladder makes it impossible to forget a release, and the order-of-release is visually identical to the reverse of the order of acquisition. The pattern is pervasive in the FreeBSD tree.

**Use this when** a function acquires more than one resource. Three is usually the threshold where a ladder is cleaner than nested `if`.

**Avoid this when** the function is simple. A single acquisition, a single release, and a single error path do not need a ladder; a straight-line function with two returns is clearer.

**The shape.**

```
int
my_attach(device_t dev)
{
 struct my_softc *sc;
 struct resource *mem = NULL;
 void *ih = NULL;
 int error, rid;

 sc = device_get_softc(dev);
 sc->dev = dev;

 rid = 0;
 mem = bus_alloc_resource_any(dev, SYS_RES_MEMORY, &rid, RF_ACTIVE);
 if (mem == NULL) {
 device_printf(dev, "cannot allocate memory window\n");
 }
}
```



```
 error = ENXIO;
 goto fail_mem;
 }

 error = bus_setup_intr(dev, sc->irq, INTR_TYPE_MISC | INTR_MPSAFE,
 NULL, my_intr, sc, &ih);
 if (error != 0) {
 device_printf(dev, "cannot setup interrupt: %d\n", error);
 goto fail_intr;
 }

 sc->mem = mem;
 sc->ih = ih;
 return (0);

fail_intr:
 bus_release_resource(dev, SYS_RES_MEMORY, rid, mem);
fail_mem:
 return (error);
}
```

The shape is what matters. Each label releases exactly the resources acquired between it and the start of the function. Falling through is intentional; the labels are stacked in the order a human would read them from the top of the function.

### Variants.

- A single out: label with conditional checks (`if (mem != NULL) bus_release_resource(...)`) works for short functions. For long ones, the numbered-label variant is easier to audit because it encodes the order directly.
- Some code names the labels `err_mem`, `err_intr` instead of `fail_mem`, `fail_intr`. The prefix does not matter; consistency within a file does.

### Common traps.

- Freeing in the wrong order. Always free in reverse order of allocation. The ladder's visual order is a reminder, not a proof; review the labels when you add a new resource.
- Initialising pointers to something other than `NULL`. If a ladder conditionally frees, the condition has to be reliable. Uninitialised pointers are undefined; free of uninitialised memory is a panic.
- Returning on the success path with a partial state. The success path at the end should either hand full ownership to the caller or return an error and unwind. There is no third option.

**Where the book teaches this.** Chapter 5 introduces the pattern explicitly and the book uses it consistently from Chapter 7 onward.

**What to read next.** Real attach functions under `/usr/src/sys/dev/`. Almost any driver shows the pattern in action.

### 40.7.2 Return-Code Conventions

**What it is.** The kernel convention: integer return codes where 0 means success and a positive errno value means failure. 1 is not an error; it is a truthy return that would be confusing next to EPERM (which is also 1). Always return 0 on success.

**Core conventions.**

- 0 on success, a positive errno (EINVAL, ENOMEM, ENXIO, EIO, EBUSY, EAGAIN) on failure.
- Never return a negative error code. That is a Linux convention and does not match the rest of FreeBSD.
- Propagate error codes upwards unchanged unless you can make them more specific.
- When a function returns a pointer, convention is to return NULL on failure and set a separate int if the caller needs to distinguish why.
- Probe routines follow a different rule: they return BUS\_PROBE\_DEFAULT and friends on success and ENXIO on no-match. See Appendix A for the full list.

**Common traps.**

- Returning -1 from a kernel function. That is a userspace idiom; do not do it in kernel code.
- Overloading a single return value to mean both “an error” and “a resource count”. Use a separate output pointer parameter if you need both.
- Swallowing errors. An error code from a lower-level call is a contract; either handle it or propagate it. Returning 0 after a failed copyin is how kernel panics are born.

**Where the book teaches this.** Chapter 5 again, alongside the cleanup ladder.

### 40.7.3 Resource Acquisition and Release Ordering

**What it is.** The discipline that says: every resource you acquire in attach must be released in detach, in reverse order. Every lock you take must be released along every exit path. Every reference you add must be released before you drop the last one. The cleanup ladder above is a local expression of this principle; the whole driver is a global one.

**Core principles.**

- Attach acquires in order A, B, C. Detach releases in order C, B, A. The two are mirror images.
- Every error path inside attach unwinds what has been acquired so far, in reverse. That is the ladder.
- Locks are held for as short a time as possible and released along every exit, including the error paths. A goto ladder that unwinds memory but forgets the mutex is still wrong.
- Ownership should be local. One function acquires; the same function releases, or it hands ownership explicitly to the caller. A function that “leaks” ownership to a parameter side-effect is almost always a bug waiting to happen.

**Where the book teaches this.** Chapter 7 introduces the mirror-image discipline for a first driver, and every driver chapter afterwards repeats it.

## 40.8 Concurrency Patterns

Two threads touching the same state is where kernel bugs are born. The synchronisation primitives in Appendix A are the tools; this section collects the patterns that use those tools well. The goal here is pattern recognition, not a full concurrency course.

### 40.8.1 Producer-Consumer with Condition Variables

**What it is.** One context produces data (fills a buffer, posts a command, receives a character), and another consumes it. A shared state protected by a mutex, and a condition variable that the consumer waits on when the state is empty.

**Mental model.**

- The mutex protects the shared state. That is non-negotiable.
- The consumer checks the state under the mutex. If there is nothing to do, it sleeps on the condition variable via `cv_wait`. `cv_wait` atomically drops the mutex while sleeping and reacquires it on return.
- The producer also holds the mutex while it modifies the state, then signals the condition variable.
- The consumer re-checks the state after waking, because spurious wakeups and shared signals are possible. The check is always a loop, not an `if`.

**The shape.**

```
/* Producer. */
mtx_lock(&sc->lock);
put_into_buffer(sc, item);
cv_signal(&sc->cv);
mtx_unlock(&sc->lock);

/* Consumer. */
mtx_lock(&sc->lock);
while (buffer_is_empty(sc))
 cv_wait(&sc->cv, &sc->lock);
item = take_from_buffer(sc);
mtx_unlock(&sc->lock);
```

**Common traps.**

- Using `if` instead of `while` around `cv_wait`. Spurious wakeups are real; always re-check.
- Signalling without holding the lock. FreeBSD permits it, but it is almost always easier to reason about if you signal while holding the lock.
- `cv_signal` when you meant `cv_broadcast`. `cv_signal` wakes exactly one waiter; if several waiters are waiting on different preconditions, only one of them will be unblocked, possibly the wrong one.
- Sleeping under a spinlock. `cv_wait` is sleepable; you need a normal mutex (`MTX_DEF`), not a spin mutex (`MTX_SPIN`).

**Where the book teaches this.** Chapter 10 builds the pattern implicitly around the circular buffer; Chapters 11 and 12 formalise it.

### 40.8.2 Reader-Writer: `rmlock(9)` vs `sx(9)`

**What it is.** Two different read-write locks, with different cost profiles.

- `sx(9)` is a sleepable shared-exclusive lock. Readers block writers; writers block readers. Both sides may sleep inside the critical section.
- `rmlock(9)` is a read-mostly lock with an extremely cheap reader path (essentially a per-CPU indicator) and a much more expensive writer path (the writer must wait for every reader to drain).

**Choosing between them.**

- If reads are frequent, writes are rare, and the reader path must not sleep, choose `rmlock`.
- If reads are frequent, writes are occasional, and the reader path needs to sleep (for example, it calls `copyout`), choose `sx`.
- If reads and writes are roughly balanced or if the critical section is short, neither is right. Use a plain mutex.

**Core operations.**

- `sx`: `sx_slock`, `sx_sunlock` for shared; `sx_xlock`, `sx_xunlock` for exclusive; `sx_try_slock`, `sx_try_xlock`.
- `rmlock`: `rm_rlock(rm, tracker)`, `rm_runlock(rm, tracker)` for readers (each reader needs its own struct `rm_priotracker`, typically on the stack); `rm_wlock(rm)`, `rm_wunlock(rm)` for writers.

**Common traps.**

- Using `rmlock` for a balanced workload. The writer path is genuinely slow; use it only when reads dwarf writes.
- Letting an `rmlock` reader sleep without initialising with `RM_SLEEPABLE`. The default `rmlock` forbids sleeping in the read path.
- Upgrading a shared lock to an exclusive one. Neither `sx` nor `rmlock` supports a lock-free upgrade. Drop the shared lock, acquire the exclusive one, and re-check state.

**Where the book teaches this.** Chapter 12.

### 40.8.3 Reference Counting with `refcount(9)`

**What it is.** A kernel-blessed pattern for counting how many contexts still use an object. An atomic counter, `refcount_acquire` to bump it, `refcount_release` to drop it and return `true` when the count reaches zero (so the caller can free).

**Why drivers use it.** Whenever an object can outlive the operation that produced it. A `cdev` that is being closed while a read is in flight; a `softc` whose detach races an `ioctl`; a buffer passed to hardware that must not be freed until the hardware is done.

**Use this when** you have shared ownership and no single place that can reliably free the object. If one piece of code unambiguously owns the object, skip the reference count and free it there.

**Core operations.** `refcount_init(count, initial)`, `refcount_acquire(count)` returning the previous value, `refcount_release(count)` returning `true` if the count reached zero (caller must free). There are

`_if_gt` and `_if_not_last` variants for conditional acquisition, and `refcount_load` for read-only inspection.

### Common traps.

- Using `refcount` for resources with a real ownership hierarchy. Refcounts are for genuinely shared ownership, not for laziness about who frees what.
- Freeing before the release. `refcount_release` returning `true` is your permission to free; do it inside that branch, not before.
- Mixing refcounts with locks. Refcount operations are lock-free, but “increment the refcount then dereference” is a race if another thread may drop the last reference between the two steps. The usual fix is to hold a lock while you decide to acquire.

**Where the book teaches this.** Mentioned in Appendix A, with the concurrency context built up through Chapters 11 and 12.

**What to read next.** `refcount(9)`, and any subsystem that manages shared lifetime.

## 40.9 Decision Aids

The compact tables below are meant to be glanceable. They are not decision oracles; they are reminders of which family to look inside. The real decision is always local to the problem.

### 40.9.1 Collection Shape

| You have...                                | Reach for                                                                       |
|--------------------------------------------|---------------------------------------------------------------------------------|
| A small collection, no ordered lookup      | <code>TAILQ_</code> or <code>LIST_</code> from <code>&lt;sys/queue.h&gt;</code> |
| A FIFO queue with no arbitrary removal     | <code>STAILQ_</code>                                                            |
| A collection ordered by a key, grows large | <code>RB_</code> from <code>&lt;sys/tree.h&gt;</code>                           |
| A dense universe of boolean flags          | <code>bit_*</code> from <code>&lt;sys/bitstring.h&gt;</code>                    |
| Unordered lookup on a large set            | <code>hashinit(9)</code> plus <code>LIST_</code>                                |
| Prefix-matching networking keys            | <code>radix.h</code> (you usually do not write this yourself)                   |

### 40.9.2 Ring Buffer Shape

| You have...                                    | Reach for                                                   |
|------------------------------------------------|-------------------------------------------------------------|
| One producer, one consumer, lock-free          | SPSC ring with head/tail indices                            |
| Many producers, many consumers                 | <code>buf_ring(9)</code>                                    |
| Producer is an interrupt, consumer is a thread | SPSC ring plus condition variable or <code>selrecord</code> |

---

| You have...                                | Reach for                                                                 |
|--------------------------------------------|---------------------------------------------------------------------------|
| Soft queue with back-pressure to userspace | SPSC ring plus <code>poll(2)</code> or <code>kqueue(2)</code> integration |

---

### 40.9.3 State Representation

---

| Your state is...                              | Reach for                 |
|-----------------------------------------------|---------------------------|
| A handful of mutually exclusive modes         | enum plus switch          |
| Many states, many events, complex transitions | Table-driven FSM          |
| Coarse modes that change the driver's methods | Function-pointer dispatch |
| Independent boolean facts                     | Flag bits in an integer   |

---

### 40.9.4 Cleanup Strategy

---

| Your function...                   | Reach for                                                             |
|------------------------------------|-----------------------------------------------------------------------|
| Acquires one resource              | Single early return, single release                                   |
| Acquires two or three              | <code>goto out:</code> with conditional <code>if (ptr != NULL)</code> |
| Acquires many, order-sensitive     | Numbered <code>goto fail_N:</code> ladder                             |
| Has to roll back a mid-way success | Same ladder with explicit “rollback” label                            |

---

### 40.9.5 Concurrency Pattern

---

| You have...                                        | Reach for                                      |
|----------------------------------------------------|------------------------------------------------|
| Short critical section, no sleeping                | <code>mtx(9)</code> with <code>MTX_DEF</code>  |
| Short critical section in an interrupt filter      | <code>mtx(9)</code> with <code>MTX_SPIN</code> |
| Many readers, occasional writer, may sleep         | <code>sx(9)</code>                             |
| Many readers, rare writer, no sleep in readers     | <code>rmlock(9)</code>                         |
| One producer, one consumer, waiting on an event    | mutex plus <code>cv(9)</code>                  |
| Shared ownership of an object with no single freer | <code>refcount(9)</code>                       |
| A counted resource (pool of slots)                 | <code>sema(9)</code>                           |
| One-word counter or flag                           | <code>atomic(9)</code>                         |

---

## 40.10 Wrapping Up: How to Recognise These Patterns in Real Code

Recognising a pattern in someone else's driver is a different skill from writing the pattern yourself. The easy part is spotting the API: `TAILQ_FOREACH`, `buf_ring_enqueue`, `goto fail_mem`. The harder part is spotting the pattern when the author has not used an API at all, because the pattern is pure control flow.

Three habits help.

The first is asking what invariant the author is trying to preserve. A ring buffer preserves “the interrupt and the thread never write the same index”. A state machine preserves “we are always in exactly one state”. A cleanup ladder preserves “every resource is released exactly once, in reverse order”. When you read new code, naming the invariant first gives the rest of the code somewhere to attach to.

The second is reading the comment at the top of the file. Most FreeBSD drivers include a short block comment explaining the locking discipline, the state machine, or the ring conventions they use. That comment exists precisely because a reader cannot recover it from the code alone. Read it before you read the code, and read it again when the code surprises you.

The third is letting the patterns in this appendix colonise your vocabulary. When you find yourself writing “the ring buffer”, be able to say whether it is SPSC or MPMC, whether the indices are sacrificed-slot or free-running, where the barrier belongs. When you find yourself writing “the state machine”, be able to name the states, the events, and the transient states. When you find yourself writing “the cleanup path”, be able to list the resources in reverse order of acquisition. The appendix becomes useful once these words come to mind automatically.

From here you can move in several directions. Appendix A has the detailed API entries for every primitive named above. Appendix C grounds the hardware-side patterns (DMA rings, descriptor arrays, and the discipline of coherency) that sit just below these software ones. Appendix E covers the kernel subsystems that many of these patterns were born inside. And every chapter of the book has labs where these patterns appear in a working driver, so that the next time you meet one in the wild you recognise it without slowing down. That is all pattern recognition really is: the confidence to keep reading.

# 41 Appendix C: Hardware Concepts for Driver Developers

## 41.1 How to Use This Appendix

The main chapters teach the driver-facing side of FreeBSD: how to write a softc, how to declare a cdev, how to set up an interrupt, how to walk a PCI BAR into a bus\_space handle, how to tame a DMA engine with bus\_dma(9). Underneath every one of those topics there is a quiet layer of hardware knowledge that the book keeps using without always naming. What a physical address really is. Why the CPU cannot simply read a device register through an ordinary pointer. Why PCI and I2C feel like completely different worlds even though both carry bytes between a CPU and a peripheral. Why an interrupt is not a function call. Why DMA changes how you write a driver from top to bottom.

This appendix is that quiet layer, written down. It is a concept guide, not an electronics textbook. It is shaped by what a driver author actually needs to believe about the machine in order to write correct code, and deliberately stops short of full computer-architecture teaching. If you have ever written a bus\_space\_read\_4 call and wondered what the tag and the handle really represent, or set up an MSI-X vector and wondered what is being signalled across which wires, or debugged a DMA transfer that worked on your desk and broke on a different board, you are in the right place.

### 41.1.1 What You Will Find Here

The appendix is organised by driver relevance, not by abstract hardware taxonomy. Each topic follows the same compact rhythm:

- **What it is.** One or two sentences of plain definition.
- **Why this matters to driver writers.** The concrete place where the concept shows up in your code.
- **How this shows up in FreeBSD.** The API, header, or convention that names the idea.
- **Common trap.** The misunderstanding that actually costs people time.
- **Where the book teaches this.** A pointer back to the chapter that uses it in context.
- **What to read next.** A manual page, a header, or a real driver you can open.

Not every topic uses every label; the shape is a guide, not a template.

### 41.1.2 What This Appendix Is Not

It is not a first course in electronics. It assumes you can accept “a wire carries a signal” and move on. It is not a computer-architecture textbook either; there is no discussion of pipelines, caches beyond what a driver author must know, or branch prediction. It is not a replacement for the focused chapters: Chapter 16 teaches



`bus_space(9)`, Chapter 18 teaches PCI, Chapter 19 teaches interrupts, Chapter 20 teaches MSI and MSI-X, and Chapter 21 teaches DMA in depth. This appendix is what you keep open beside those chapters when you want a mental-model anchor instead of a full walkthrough.

It also does not overlap with the other appendices. Appendix A is the API reference; Appendix B is the algorithmic-pattern field guide; Appendix D is the operating-systems-concepts companion; Appendix E is the kernel-subsystem reference. If the question you want answered is “what does this macro call do” or “what algorithm fits this problem” or “how does the scheduler schedule”, you want a different appendix.

## 41.2 Reader Guidance

Three ways to use this appendix, each calling for a different strategy.

If you are **learning the main chapters**, open the appendix alongside them. When Chapter 16 mentions memory barriers, flip to the memory section here. When Chapter 18 talks about BARs, flip to the PCI section. When Chapter 19 introduces edge versus level triggering, flip to the interrupt section. The appendix is designed to be a mental-model companion, not a linear read. Thirty minutes from start to finish is realistic for a first pass; a few minutes at a time is more common in daily use.

If you are **reading real driver code**, use the appendix as a translator for unfamiliar terminology. When the header comment says “coherent DMA memory” or “message-signalled interrupt” or “BAR at offset 0x10”, find the concept here and keep going. Full understanding can come later; the first job is to form a mental picture of what the driver is doing against what kind of hardware.

If you are **designing a new driver**, read the sections that match your hardware. A PCI NIC will touch the memory model, PCI, interrupts (probably MSI-X), and DMA, so those four sections are your warm-up. An I2C sensor will touch the I2C section and the interrupt section and almost nothing else. An embedded driver on a system-on-chip will touch everything except PCI. Match the hardware to the sections, then skim only those.

A few conventions apply throughout:

- Source paths are shown in the book-facing form, `/usr/src/sys/...`, matching a standard FreeBSD system. You can open any of them on your lab machine.
- Manual pages are cited in the usual FreeBSD style. The kernel-facing pages live in section 9: `bus_space(9)`, `bus_dma(9)`, `pci(9)`, `malloc(9)`, `intr_event(9)`. Device overviews usually live in section 4: `pci(4)`, `usb(4)`, `iicbus(4)`.
- When an entry points to real source as reading material, the file is one a beginner can navigate in one sitting. Larger subsystems exist that also use the pattern; those are mentioned only when they are the canonical example.

With that in mind, we start with memory: the single concept that separates a working driver from a puzzling one.

## 41.3 Memory in the Kernel

A driver that misunderstands memory will write perfect-looking code that crashes on real hardware. Memory in the kernel is not the same as memory in a C tutorial. There are at least two address spaces in play (physical

and virtual), there are constraints the hardware enforces (DMA addressing limits, page boundaries), and there are regions where even an ordinary read has side effects (device registers). This section builds the mental model that every later topic in the appendix depends on.

### 41.3.1 Physical vs Virtual Memory

**What it is.** A physical address is the number the memory controller uses to select a real location in RAM, or the number the I/O fabric uses to select a register on a peripheral. A virtual address is the number a running thread sees when it dereferences a pointer; the memory-management unit (MMU) translates it to a physical address on the fly, using the page tables that the kernel maintains.

**Why this matters to driver writers.** Every pointer in your driver is virtual. The CPU never sees a physical address when it executes your C code. The kernel and the `pmap(9)` layer keep the page tables so that kernel virtual addresses map to wherever in physical memory the kernel has decided to put the corresponding pages. A driver that “knows” a physical address cannot simply cast it to `(void *)` and dereference; the translation might not exist, the permissions might forbid it, or the attributes might be wrong for the kind of access you need.

Hardware is on the opposite side of the mirror. A peripheral device lives somewhere in a physical address space (its registers are physical addresses, and when it does DMA it emits physical addresses on the bus). A device never dereferences a virtual pointer. If your driver hands the device a kernel virtual address and expects it to read the data there, the device will read some unrelated physical location and you will spend a long afternoon figuring out why.

**How this shows up in FreeBSD.** Two places. First, `bus_space(9)` encodes the translation for register access: a tag describes the kind of physical address space, a handle describes the virtual mapping the kernel has set up, and the `bus_space_read_*` and `bus_space_write_*` accessors use the handle to reach the device through that mapping. Second, `bus_dma(9)` mediates DMA: the `bus_dmamap_load` call walks a buffer and produces the list of bus addresses the device should use, which are not necessarily the same as either the virtual addresses or the physical addresses, because an IOMMU may sit in between.

**Common trap.** Casting a physical address returned by firmware or a BAR register to a pointer and reading from it directly. On some architectures and some builds this compiles and may even return something, but it is never correct. Always map through `bus_space`, `pmap_mapdev`, or an established FreeBSD API.

**Where the book teaches this.** Chapter 16 establishes the distinction and shows why raw pointer arithmetic is not enough for device memory. Chapter 21 returns to it when DMA makes the physical side visible.

**What to read next.** `bus_space(9)`, `pmap(9)`, and the early parts of `/usr/src/sys/kern/subr_bus_dma.c`.

### 41.3.2 The MMU, Page Tables, and Device Mappings

**What it is.** The MMU is the hardware unit that translates virtual addresses into physical addresses using a tree of page tables. Each page-table entry holds a physical page number and a small set of attribute bits: readable, writable, executable, cacheable, user-accessible. The same physical page can be mapped at several virtual addresses with different attributes; the same virtual address can point at different physical pages at different times.

**Why this matters to driver writers.** Device memory is not ordinary RAM, and the CPU must know that. A register may change on its own (a status bit that sets when the device finishes a transfer), it may have side effects

on read (a read-to-clear flag), and it may require that writes become visible in a specific order. If the CPU caches a device register the way it caches an integer in RAM, the code reads a stale value forever. The MMU solves this by letting the kernel map device regions with different attributes: typically uncached, strongly ordered, and sometimes write-combining. A driver does not usually choose those attributes by hand. `bus_space` and `bus_alloc_resource` do it on your behalf.

**How this shows up in FreeBSD.** When `bus_alloc_resource_any(dev, SYS_RES_MEMORY, &rid, RF_ACTIVE)` returns, the kernel has already mapped the device region with the right attributes for MMIO. The struct `resource` \* you get back carries a `bus_space_tag_t` and `bus_space_handle_t` pair that the accessors use. You should never manipulate the underlying page table yourself. In practice the access is two lines:

```
uint32_t val = bus_space_read_4(rman_get_bustag(res),
 rman_get_bushandle(res), STATUS_REG_OFFSET);
```

Each `bus_space_read_4` / `bus_space_write_4` call goes through the MMIO mapping the kernel set up for you; no pointer dereference, no cached load.

**Common trap.** Mixing cacheable and uncacheable views of the same physical region. A DMA buffer allocated with `bus_dmamem_alloc` and then cast to a different virtual mapping will give you two incoherent views of the same bytes. Use the mapping the allocator returned, not a mapping you build on the side.

**Where the book teaches this.** Chapter 16 touches the MMU indirectly when it explains why the `bus_space` abstraction exists. Chapter 21 makes it explicit when DMA coherence becomes the topic.

**What to read next.** `pmap(9)`, `bus_space(9)`, and `/usr/src/sys/vm/vm_page.h` for the kernel's representation of a physical page (`vm_page_t`).

### 41.3.3 Why Hardware Cannot Simply See Kernel Pointers

**What it is.** A short way of stating the most important consequence of the previous two subsections. The kernel lives in virtual memory. Peripherals live in physical (and sometimes I/O-virtual) memory. A kernel pointer has no meaning to a device.

**Why this matters to driver writers.** Every time you give a device a memory address, you must give it a bus address, not a virtual pointer. The bus address is what the device sees; it may be the physical address, or it may be an I/O-virtual address remapped by an IOMMU. You do not compute it yourself. The DMA infrastructure computes it for you when you load a map.

**How this shows up in FreeBSD.** `bus_dmamap_load` takes a virtual buffer and produces `bus_dma_segment_t` entries whose `ds_addr` is the address the device must use. You copy that value into the descriptor you are about to hand the hardware. The moment you find yourself wanting to pass the output of `vtophys` or a raw BAR value to the device, stop and reach for `bus_dma` instead.

**Common trap.** Writing `(uint64_t)buffer` into a device descriptor because it works in a toy test on one machine. It will fail on any machine where the kernel address is not the same as the bus address, which is most real machines.

**Where the book teaches this.** Chapter 21. The whole chapter is about honouring this separation.

**What to read next.** `bus_dma(9)`, and the DMA section below.

### 41.3.4 Kernel Memory Allocation in One Page

The kernel provides three main ways to get memory, each with a different purpose. Drivers reach for them constantly, and picking the wrong one is a classic beginner mistake. The point here is the conceptual difference; the full API reference is in Appendix A.

- **malloc(9)** is the general-purpose allocator. It returns kernel-virtual memory that is perfectly fine for softc structures, lists, command buffers for your own bookkeeping, and anything else that stays inside the driver's data. It is not automatically physically contiguous, and it is not automatically suitable for DMA. Use it for control data.
- **uma(9)** is the zone allocator. It is a fast cache for fixed-size objects that the driver allocates and frees often, such as per-request structures or per-packet state. It is a specialisation of general-purpose allocation; it does not change the physical-vs-virtual story.
- **contigmalloc(9)** returns a physically contiguous range, with options for alignment and an upper address limit. A driver calls it directly only when it truly needs a single contiguous physical block and the `bus_dma` layer is not a better fit. In modern drivers, `bus_dma` almost always is the better fit. The call shape is telling on its own:

```
void *buf = contigmalloc(size, M_DEVBUF, M_WAITOK | M_ZERO,
 0, BUS_SPACE_MAXADDR, PAGE_SIZE, 0);
```

The `low`, `high`, `alignment`, and `boundary` arguments are exactly the DMA constraints a `bus_dma_tag_create` call would express, which is why `bus_dma(9)` is almost always the better place to encode them.

- **bus\_dmamem\_alloc(9)** is the DMA-aware allocator. It returns a buffer that is usable for DMA under a given tag, properly aligned, optionally coherent, and paired with a `bus_dmamap_t` you can load. This is the call you want in almost every DMA scenario.

**Why this matters to driver writers.** The question to ask yourself is always “who is going to read or write this memory?”. If only the CPU ever touches it, `malloc(9)` is fine. If the device needs to read or write it, you want `bus_dmamem_alloc(9)` under a `bus_dma(9)` tag that expresses the device's addressing constraints. Mixing the two is the source of an entire genus of driver bugs.

**Common trap.** Allocating a buffer with `malloc(9)`, handing its virtual address to the device, and wondering why the device reads garbage. The fix is rarely to patch around the symptoms; it is to rewrite the allocation through `bus_dma(9)`.

**Where the book teaches this.** Chapter 5 introduces `malloc(9)` in a kernel-C context, and the driver chapters from Chapter 7 onward use it repeatedly for softc and bookkeeping memory. Chapter 21 introduces `bus_dmamem_alloc`. Appendix A lists the full API and flags.

**What to read next.** `malloc(9)`, `contigmalloc(9)`, `uma(9)`, `bus_dma(9)`.

### 41.3.5 IOMMUs and Bus Addresses

**What it is.** An IOMMU is a hardware unit (sometimes the `amdvi` or `intel-iommu` unit; on ARM, the SMMU) that sits between devices and main memory and translates the addresses devices emit into physical addresses. It is the device-side analogue of the MMU.

**Why this matters to driver writers.** When an IOMMU is present, the address the device uses on the bus (a bus address or I/O-virtual address) is not the same as the physical address of the underlying memory. The IOMMU lets the kernel give each device a restricted view of memory, which is good for safety and isolation, and it lets the kernel use scatter-gather where the device would otherwise demand contiguous physical pages.

**How this shows up in FreeBSD.** Transparently, for most drivers. When the kernel is built with the IOMMU option and one of the architecture backends (the Intel and AMD drivers under `/usr/src/sys/x86/iommu/` on amd64, the SMMU backend under `/usr/src/sys/arm64/iommu/` on arm64), the `busdma_iommu` layer in `/usr/src/sys/dev/iommu/` bridges `bus_dma(9)` and the IOMMU automatically, without changing the driver's code. You still allocate a tag, create a map, load the map, and read the `ds_addr` of each segment; the number you get is the address the device should use, IOMMU or not. You do not need to know whether an IOMMU is present.

**Common trap.** Assuming `ds_addr` is a physical address and logging it that way. On IOMMU systems the number is an I/O-virtual address and will not match anything you see in `/dev/mem`. The right mental model is “the address the device uses”, not “the physical address”.

**Where the book teaches this.** Chapter 21 mentions IOMMU integration briefly and sticks to the common case where `bus_dma` hides it.

**What to read next.** `bus_dma(9)`, `/usr/src/sys/dev/iommu/busdma_iommu.c` for the bridge code, and `/usr/src/sys/x86/iommu/` or `/usr/src/sys/arm64/iommu/` for the per-architecture backends.

### 41.3.6 Memory Barriers in One Paragraph

On modern CPUs, memory operations may become visible to other agents (other CPUs, other devices) in an order different from the one in which the code issued them. For ordinary RAM the kernel's locking primitives hide this from you. For device registers, though, ordering matters: writing to a “start” register before writing to the “data” register produces a different outcome from the reverse. The kernel provides explicit barriers for both sides. For register access, `bus_space_barrier(tag, handle, offset, length, flags)` is the right tool; for DMA buffers, `bus_dmamap_sync(tag, map, op)` is the right tool. You will meet both again later in this appendix. For now, internalise the rule: when the order of two accesses matters to the hardware, say so with a barrier. Do not rely on the code order alone.

## 41.4 Buses and Interfaces

A bus is a physical and logical pathway that carries commands, addresses, and data between a CPU and peripherals. FreeBSD treats every bus as a specialised tree in the Newbus framework: the bus has a driver that enumerates its children, each child is a `device_t`, and each child can in turn be a bus for further children. The concept of “a bus” is uniform at the Newbus level. The concepts of “the electrical, protocol, and enumeration rules of a particular bus” are not uniform at all, and a driver author must understand which one they are writing for.

This section gives the mental models for the four buses the book cares about: PCI (and PCIe), USB, I2C, and SPI. Their chapters are later; this is the shared vocabulary you need first.

### 41.4.1 PCI and PCIe at a Driver-Relevant Level

**What it is.** PCI is the canonical bus for attaching peripherals to a CPU complex. Classical PCI was parallel and shared; modern PCI Express is a set of point-to-point serial lanes that a switch fabric organises into a tree. From a driver's perspective, the electrical difference is mostly invisible: you still see a configuration space, you still see Base Address Registers, you still see interrupts, and you still write probe-and-attach code. Unless stated otherwise, "PCI" in this book means "PCI and PCIe together".

A PCI device is identified by a Bus:Device:Function (B:D:F) tuple and by a configuration space that the firmware and the kernel populate. The configuration space holds the vendor and device identifiers, the class code that describes what kind of peripheral this is, the Base Address Registers (BARs) that describe the device's memory and I/O windows, and a linked list of capability structures that advertise optional features like MSI, MSI-X, and power management.

**Why this matters to driver writers.** Every PCI driver does roughly the same dance. Read `pci_get_vendor(dev)` and `pci_get_device(dev)` during probe, compare them to a table of devices the driver supports, return a `BUS_PROBE_*` value on a match. In attach, allocate each BAR with `bus_alloc_resource_any(dev, SYS_RES_MEMORY, &rid, RF_ACTIVE)`, pull the `bus_space_tag_t` and `bus_space_handle_t` out with `rman_get_bustag` and `rman_get_bushandle`, allocate the interrupt with `bus_alloc_resource_any(dev, SYS_RES_IRQ, ...)` (or `pci_alloc_msix/pci_alloc_msi` for message-signalled interrupts), wire the handler with `bus_setup_intr`, and you are in business.

Capabilities are where the PCI driver meets the features of the specific device family. `pci_find_cap(dev, PCIY_MSIX, &offset)` finds the MSI-X capability; `pci_find_extcap(dev, PCIZ_AER, &offset)` finds the extended capability for Advanced Error Reporting. A driver usually only walks the capabilities it cares about.

**How this shows up in FreeBSD.** The PCI bus driver lives under `/usr/src/sys/dev/pci/`. The key header is `/usr/src/sys/dev/pci/pci.h` for register constants and `/usr/src/sys/dev/pci/pcivar.h` for functions a client driver calls. Every PCI driver you will write starts with a `device_probe_t` and a `device_attach_t`, with `DRIVER_MODULE(name, pci, driver, 0, 0)` at the bottom of the file. The last two arguments are the per-module event handler and its argument, which are normally left as `0, 0` (or `NULL, NULL`); the separate `devclass_t` slot that older books mention was removed from the macro some releases ago.

**Common trap.** Forgetting to call `pci_enable_busmaster(dev)` before expecting the device to DMA. The chip can live in a quiet state where MMIO works and DMA silently does nothing. Enabling bus-mastering in attach is part of the usual sequence.

**Where the book teaches this.** Chapter 18 is the full chapter on PCI driver writing. Chapter 20 extends it with MSI-X.

**What to read next.** `pci(4)`, `pci(9)`, and a compact real example such as `/usr/src/sys/dev/uart/uart_bus_pci.c`. The PCI bus itself is implemented in `/usr/src/sys/dev/pci/pci.c`.

### 41.4.2 USB at a Driver-Relevant Level

**What it is.** USB is a host-controlled tiered bus, in which a host controller talks to devices through a tree of hubs. Unlike PCI, the bus is strictly hierarchical and polling-like: the host schedules every transaction. A device

does not “speak” on the bus without being asked. USB speed classes (low, full, high, super) share the same conceptual model; the protocol details change.

A USB device exposes a set of endpoints. Each endpoint is a one-way data channel with a specific transfer type: control, bulk, interrupt, or isochronous. Control is the setup channel every device has. Bulk is high-throughput with no timing guarantees. Interrupt is small, periodic, low-latency (mice, keyboards, sensors). Isochronous is real-time streaming (audio, video) that trades reliability for timing.

**Why this matters to driver writers.** A FreeBSD USB driver attaches to a `device_t` whose parent is a USB bus, not a PCI bus. The identification model is different: instead of vendor/device IDs and class codes in configuration space, a USB driver matches on a USB Vendor ID (VID), Product ID (PID), device class, subclass, and protocol. The FreeBSD USB stack gives you a structured probe against those fields. At run time, a USB driver opens specific endpoints through the USB framework and issues transfers whose lifetime is managed by the host controller; you do not write to registers the way a PCI driver does.

**How this shows up in FreeBSD.** The USB stack lives under `/usr/src/sys/dev/usb/`. A class driver uses `usbd_transfer_setup` to describe the endpoints it wants, `usbd_transfer_start` to initiate transfers, and callback functions to handle completion. Transfer-type constants live in `/usr/src/sys/dev/usb/usb.h` (`UE_CONTROL`, `UE_ISOCHRONOUS`, `UE_BULK`, `UE_INTERRUPT`).

**Common trap.** Thinking in terms of registers and interrupts. A USB device driver is event-driven around transfer completion callbacks, not around device-register reads. A register-oriented mindset leads to fighting the framework instead of using it.

**Where the book teaches this.** Chapter 26 is the full chapter on USB and serial drivers. This appendix is the concept-level primer; the chapter is where your first USB driver lives.

**What to read next.** `usb(4)`, `usbdi(9)`, and the headers under `/usr/src/sys/dev/usb/`.

### 41.4.3 I2C at a Driver-Relevant Level

**What it is.** I2C (Inter-Integrated Circuit) is a slow, two-wire, master-slave bus designed to connect low-bandwidth peripherals on a board: temperature sensors, power management controllers, EEPROMs, small displays. Each slave has a seven-bit or ten-bit address. A master initiates every transaction by asserting a start condition, sending the address, sending or receiving bytes, and asserting a stop condition.

Speeds are typically 100 kHz (standard), 400 kHz (fast), 1 MHz (fast-plus), and higher variants on newer devices. The bus supports multiple masters, but most embedded systems use a single master; in that case the driver only has to worry about the master-to-slave interaction.

**Why this matters to driver writers.** An I2C driver in FreeBSD comes in two flavours. An I2C *controller* driver implements the master side of a specific controller chip and hangs under the Newbus tree as an I2C bus. An I2C *device* driver is a client that uses the `iicbus` layer to talk to a slave device without knowing the controller details. The second form is what most driver authors write. You ask `iicbus_transfer` to perform a short sequence of `struct iic_msg` operations, and the `iicbus` layer takes care of arbitration and timing.

There are no interrupts or BARs to allocate in the client driver. There is no DMA. Each transfer is short and blocking, typically a few bytes; long reads are broken into a series of transfers.

**How this shows up in FreeBSD.** The framework lives under `/usr/src/sys/dev/iicbus/`. The client interface centres on `iicbus_transfer(dev, msgs, nmsgs)` and the `struct iic_msg` type. FreeBSD discovers

I2C buses through FDT on embedded platforms and ACPI on x86 laptops that expose an I2C controller.

**Common trap.** Writing an I2C driver that busy-waits. Even at 1 MHz, a transfer is several tens of microseconds; a driver that pins a CPU through a polling loop wastes resources. The `iicbus` layer handles the waiting for you.

**Where the book teaches this.** The book does not dedicate a full chapter to I2C; this appendix is the primary concept treatment, and the embedded chapter (Chapter 32) revisits it in the context of device-tree bindings. For a concrete client you can read front-to-back, open `/usr/src/sys/dev/iicbus/icee.c`.

**What to read next.** `iicbus(4)`, `/usr/src/sys/dev/iicbus/iicbus.h`, and an example client such as `/usr/src/sys/dev/iicbus/icee.c` (an EEPROM driver).

#### 41.4.4 SPI at a Driver-Relevant Level

**What it is.** SPI (Serial Peripheral Interface) is a simple, fast, full-duplex, master-slave bus. It has no addressing in the bus itself; each slave has a dedicated chip-select line, and the master asserts that line to begin a transaction. Four wires are typical: `SCLK` (clock), `MOSI` (master-out slave-in), `MISO` (master-in slave-out), and `SS/CS` (slave-select). Speeds commonly range from a few MHz to tens of MHz.

Unlike I2C, SPI has no protocol beyond the electrical one: the master shifts bits out on `MOSI` while simultaneously shifting bits in on `MISO`. What those bits mean is entirely up to the device.

**Why this matters to driver writers.** As with I2C, FreeBSD separates the controller from the client. A controller driver knows how the SPI master in a particular chip is programmed; a client driver uses the `spibus` interface to send and receive bytes without knowing the controller. The client calls `spibus_transfer` or `spibus_transfer_ext` with a `struct spi_command` that describes a single transaction, including the chip-select to assert and the buffers to shift.

**How this shows up in FreeBSD.** The framework lives under `/usr/src/sys/dev/spibus/`. The typical client driver hangs under an `spibus` parent in Newbus and uses the same `probe/attach` shape as any other device. `/usr/src/sys/dev/spibus/spigen.c` is a generic character-device client that exposes raw SPI to userland; reading it is a good way to see the client side of the framework.

**Common trap.** Expecting SPI to carry data with any built-in framing. It does not. The device datasheet tells you what bytes mean; the bus tells you nothing.

**Where the book teaches this.** The book does not dedicate a full chapter to SPI; this appendix is the primary concept treatment, and the embedded chapter (Chapter 32) revisits it in the context of device-tree bindings. For a concrete example you can read front-to-back, open `/usr/src/sys/dev/spibus/spigen.c`.

**What to read next.** `/usr/src/sys/dev/spibus/spibus.c`, `/usr/src/sys/dev/spibus/spigen.c`, and the device datasheet for whatever SPI peripheral you are driving.

#### 41.4.5 A Compact Comparison of the Four Buses

A small reference is often easier than a long description. This table compares the buses along the axes that actually affect driver design. It is not a full electrical comparison.



| Aspect           | PCI / PCIe            | USB                        | I2C                        | SPI                     |
|------------------|-----------------------|----------------------------|----------------------------|-------------------------|
| Topology         | Tree (PCIe switches)  | Strict host/hub tree       | Multi-drop, addressed      | Multi-drop, chip-select |
| Typical speed    | Gigabytes/s per lane  | Kilobytes/s to gigabytes/s | 100 kHz - 1 MHz            | 1 MHz - tens of MHz     |
| Identification   | Vendor/Device + class | VID/PID + class            | 7- or 10-bit slave address | None (chip-select)      |
| Protocol framing | PCI TLPs, invisible   | Packets + transfer types   | Start/address/data/stop    | Raw bits                |
| Interrupts       | INTx or MSI / MSI-X   | Transfer callbacks         | Rare; bus is polled        | Rare; bus is polled     |
| DMA              | Yes, device-driven    | Host controller does it    | No                         | No                      |
| Driver role      | Register-level        | Framework client           | iicbus client              | spibus client           |
| Enumeration      | Firmware + pci(4)     | Hub/device descriptors     | FDT / ACPI / hand-wired    | FDT / ACPI / hand-wired |

The pattern you should take away is that PCI and USB are complex enough to need their own enumeration stacks and transaction machinery, while I2C and SPI are simple enough that a short transaction struct is the whole API. A driver author's workload looks different on each side of that split.

## 41.5 Interrupts

An interrupt is how a device tells the CPU that something has happened. Without interrupts, drivers would have to poll, burning CPU cycles to discover events the hardware already knows about. With interrupts, the CPU does other work until the hardware signals; then the kernel dispatches a handler in a narrow, well-defined context. This section explains what that signal actually is, what distinguishes the common forms, and why interrupt discipline is one of the most demanding skills in driver work.

### 41.5.1 What an Interrupt Actually Is

**What it is.** At the hardware level, an interrupt is a signal the device raises to request CPU attention. On classical parallel buses, the signal is a line that the device pulls high or low. On modern PCIe, the signal is a memory-write transaction carrying a small payload that the interrupt controller recognises. In both cases, the interrupt controller maps the signal to a CPU vector, the CPU saves just enough state to switch context, and the kernel's low-level entry code decides which driver to call.

What the driver sees is a function call: the interrupt handler runs with a small stack, in a special scheduling context, usually with preemption disabled or highly constrained. The hardware model is “a device raised a line”; the software model is “the kernel ran your callback”.

**Why this matters to driver writers.** Interrupts are not ordinary function calls. They happen asynchronously, possibly on a different CPU from the one your driver is running on, and they can arrive in the middle of almost any code path. That is why interrupt handlers must be short, must avoid sleeping, and must not touch shared state without a lock that is safe in that context. Most of the discipline a driver learns about locking, ordering, and deferred work exists because of this one fact.

**How this shows up in FreeBSD.** Interrupts reach drivers through `intr_event(9)`. A driver calls `bus_setup_intr(dev, irq_resource, flags, filter, ithread, arg, &cookie)` to register up to two callbacks: a *filter* that runs first, in interrupt context, with only spin locks available, and optionally an *ithread* (interrupt thread) that runs as a kernel thread and may use sleep locks. The filter's return value decides what happens next: `FILTER_HANDLED` (fully handled, acknowledge at the controller), `FILTER_STRAY` (not mine, do nothing), or `FILTER_SCHEDULE_THREAD` (the ithread should run now). Those constants live in `/usr/src/sys/sys/bus.h`.

**Common trap.** Doing real work in the filter. The filter is there to decide ownership, acknowledge the device quickly if possible, and either handle a trivially small amount of work or schedule the thread. Long processing belongs in the ithread or on a taskqueue, not in the filter.

**Where the book teaches this.** Chapter 19 is the dedicated chapter on interrupts. Chapter 20 adds MSI and MSI-X. Chapter 14 (taskqueues) explains how to defer heavy work out of the interrupt path.

**What to read next.** `intr_event(9)`, `bus_setup_intr(9)`, and the filter/ithread documentation in `/usr/src/sys/sys/bus.h` around the `FILTER_` and `INTR_` constants.

## 41.5.2 Edge-Triggered vs Level-Triggered

**What it is.** Two ways the hardware signal can mean “there is something new”.

- **Edge-triggered.** The device asserts a transition (a rising or falling edge) to signal an event. If the driver does not record that an interrupt happened, the event is lost. The edge is a moment, not a state.
- **Level-triggered.** The device holds the line (or the signal) active as long as the condition persists. The interrupt controller keeps firing until the driver acknowledges the device and the line deasserts. The level is a state, not a moment.

**Why this matters to driver writers.** Edge-triggered interrupts are unforgiving: if the driver has a race and a second edge arrives while the first is being processed without being recorded, the second event is forgotten. Level-triggered interrupts are self-healing: if the line is still asserted when you return, the controller will fire again. Legacy PCI INTx lines are level-triggered and shared, which is why driver handlers must read the device's status register and check whether “this interrupt” is really theirs before claiming it. MSI and MSI-X are effectively edge-like: each message is a discrete event.

**How this shows up in FreeBSD.** The distinction is mostly handled below your driver. Legacy INTx registration with `SYS_RES_IRQ` and `rid 0` gets you level-triggered, shared interrupts; `pci_alloc_msi` and `pci_alloc_msix` give you message-signalled interrupts that are per-vector and not shared with unrelated drivers. Your filter still has to behave well; but the sharing concern goes away.

**Common trap.** Writing a filter that returns `FILTER_HANDLED` without actually checking the device. On level-triggered INTx, this hangs the system because the line stays asserted and the controller fires forever.

**Where the book teaches this.** Chapter 19 teaches the classical case; Chapter 20 teaches the message-signalled case.

**What to read next.** The legacy interrupt discussion in `/usr/src/sys/dev/pci/pci.c` around `pci_alloc_msi`, and the filter examples in Chapter 19.

### 41.5.3 MSI and MSI-X Without the Acronyms

**What it is.** Message-Signalled Interrupts. Instead of asserting a shared interrupt line, a PCI Express device sends a small memory-write transaction to a special address that the interrupt controller watches. The payload of the write identifies which interrupt has fired. MSI supports a handful of vectors per device (typically up to 32); MSI-X supports thousands, each with its own address and data, and per-vector CPU affinity.

**Why this matters to driver writers.** Two things change when a driver uses MSI or MSI-X instead of legacy INTx. First, the interrupts are no longer shared, so the filter does not need to coexist with unrelated drivers (it still has to confirm the event came from its own hardware). Second, you can have more than one interrupt per device. A NIC can have one interrupt per receive queue, another per transmit queue, another for the management path, and you can bind each one to a different CPU. That changes how you structure the driver: each queue has its own softc substructure, its own lock, and its own handler.

**How this shows up in FreeBSD.** `pci_msi_count(dev)` and `pci_msix_count(dev)` tell you how many vectors are available. `pci_alloc_msi(dev, &count)` and `pci_alloc_msix(dev, &count)` reserve them; the call updates `count` with the number actually allocated. You then allocate each vector with `bus_alloc_resource_any(dev, SYS_RES_IRQ, &rid, RF_ACTIVE)` using `rid = 1, 2, ..., count`. CPU affinity is set with `bus_bind_intr(dev, res, cpu_id)`. Teardown mirrors the sequence: per-vector `bus_teardown_intr` and `bus_release_resource`, then a single `pci_release_msi(dev)`.

**Common trap.** Requesting too many vectors and assuming the system will give them all. `pci_alloc_msix` may return fewer than requested, even on hardware that advertises plenty; the driver must adapt its queue structure to the number actually allocated. A fallback ladder (MSI-X, then MSI, then INTx) is the standard shape.

**Where the book teaches this.** Chapter 20.

**What to read next.** `pci(9)` for the function list, and `/usr/src/sys/dev/pci/pci.c` for the implementation.

### 41.5.4 Why Interrupt Discipline Matters

**What it is.** The set of rules that keep a driver correct under interrupts. Most of it follows from three facts: the filter runs with very few primitives available, the filter can race with every other path in the driver, and late acknowledgement produces wedged systems.

#### Core rules.

- Keep the filter short and allocation-free. No `malloc`, no sleeping, no calls that might sleep, no complex logic.
- Acknowledge at the device before returning `FILTER_HANDLED`, or make sure the device continues to assert until your `ithread` runs.

- Use spin locks (MTX\_SPIN) to protect state the filter and a CPU path share. Sleep locks cannot be taken in the filter.
- Defer real work to an ithread, a taskqueue, or a callout. The filter's job is to move data off the critical path.
- Count your interrupts. A `sysctl` counter per vector is cheap and often the first clue when the driver misbehaves.

**Common trap.** Holding a sleep lock across a path that could be re-entered from the filter. A thread that owns a sleep mutex cannot be preempted by an interrupt on the same CPU; the kernel's `witness(9)` diagnostic will catch many of these before production, but only if the scenario is actually exercised.

**Where the book teaches this.** Chapter 11 introduces the discipline in general; Chapter 19 makes it concrete for real hardware.

**What to read next.** The locking summary in Appendix A and the filter/ithread split in `/usr/src/sys/sys/bus.h`.

## 41.6 DMA

DMA (Direct Memory Access) is how a peripheral reads or writes memory without the CPU copying every byte. It exists because the alternative (the CPU reading a register, placing the byte in RAM, and looping) is too slow for any modern device. DMA is also the most hardware-aware part of a driver, because it is where the driver must speak the bus's address language instead of the kernel's virtual-pointer language. A driver that is not disciplined about DMA will produce data corruption that is almost impossible to diagnose from user space.

### 41.6.1 What DMA Is and Why It Exists

**What it is.** A transfer in which the device reads from or writes to system memory under its own control, using bus addresses the driver has handed it in advance. The CPU sets up the transfer and goes on with other work; the device signals completion, usually by raising an interrupt. There is no per-byte CPU involvement during the data movement itself.

**Why this matters to driver writers.** Everything about how you allocate, load, synchronise, and release memory changes the moment DMA enters the picture. You no longer own the buffer exclusively; the device owns it during the transfer. You must tell the kernel which parts of the buffer are being read or written and when, so that cache coherence, alignment, and address remapping can be handled correctly.

**How this shows up in FreeBSD.** `bus_dma(9)` is the whole machinery. The driver author's workflow is consistent across every DMA-capable device:

1. Create a `bus_dma_tag_t` in `attach`, describing the device's addressing constraints (alignment, boundary, `lowaddr`, `highaddr`, maximum segment size, number of segments, and so on). The tag captures the device's truth.
2. Allocate a DMA-capable buffer with `bus_dmamem_alloc(tag, &vaddr, flags, &map)`, or allocate an ordinary buffer and load it later.
3. Load the buffer with `bus_dmamap_load(tag, map, vaddr, length, callback, arg, flags)`. The callback receives one or more `bus_dma_segment_t` entries whose `ds_addr` fields are the bus addresses to hand the device.

4. Before the device reads, `bus_dmamap_sync(tag, map, BUS_DMASYNC_PREWRITE)`. Before the CPU reads what the device wrote, `bus_dmamap_sync(tag, map, BUS_DMASYNC_POSTREAD)`. The flags are defined in `/usr/src/sys/sys/bus_dma.h`.
5. When the transfer is complete, `bus_dmamap_unload(tag, map)` releases the mapping; `bus_dmamem_free(tag, vaddr, map)` releases the buffer; `bus_dma_tag_destroy(tag)` releases the tag (usually in detach).

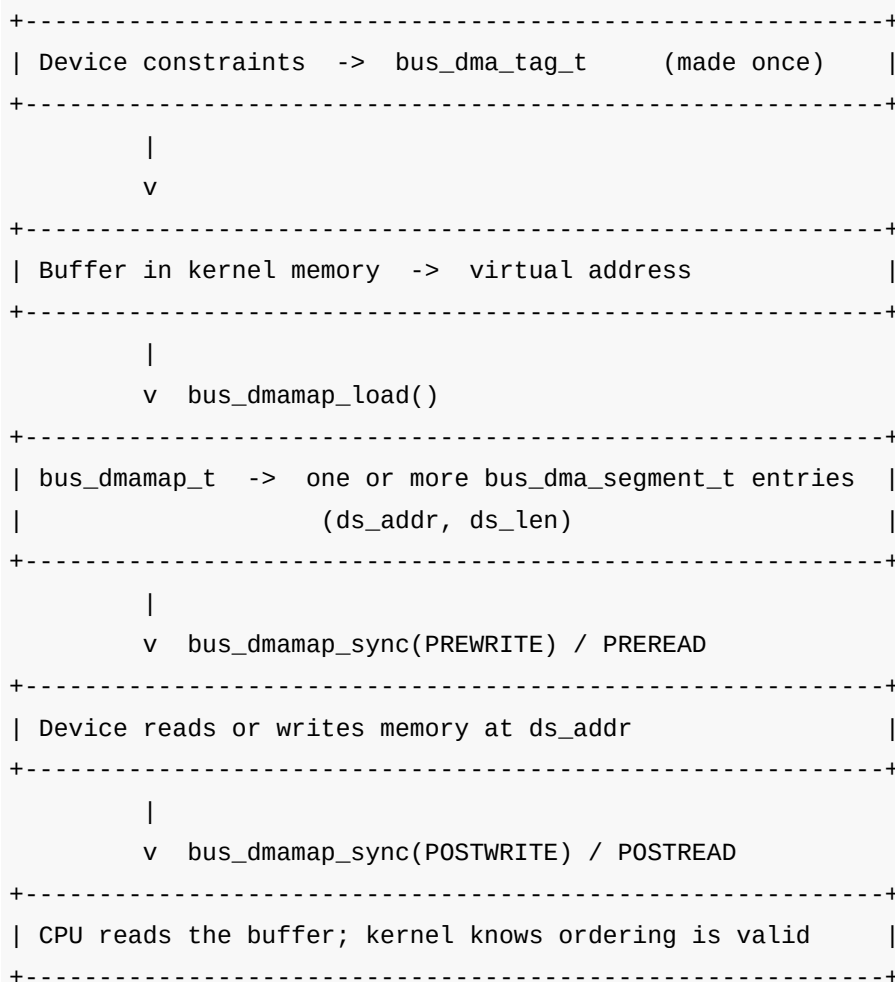
**Common trap.** Forgetting the PREWRITE or POSTREAD sync. On coherent architectures (amd64 with IOMMU, for example), the driver often works without it; on non-coherent architectures (some ARM systems), it silently corrupts data. Write the sync calls every time, even when you are developing on a forgiving platform.

**Where the book teaches this.** Chapter 21 is the full-length dedicated treatment.

**What to read next.** `bus_dma(9)` and `/usr/src/sys/kern/subr_bus_dma.c`.

### 41.6.2 The Mental Model: Tags, Maps, Segments, Sync

A short refresher in the shape of a diagram. The pieces fit together like this:



Each arrow is a responsibility. The tag expresses what the device can tolerate. The map expresses one particular set of segments the device will actually touch. The sync calls bracket the device's use of the buffer. Miss any one of them and the transfer is undefined.

### 41.6.3 Why Synchronisation and Mapping Matter

**What it is.** Two concerns that together explain why `bus_dma` is not just an address translator.

- **Mapping.** The device needs a bus address, not a virtual one. On IOMMU systems, the mapping is a real I/O-virtual range; on non-IOMMU systems, it is the physical address, possibly through a bounce buffer if the device cannot reach where the memory actually lives. The driver does not choose which; `bus_dma` chooses based on the tag.
- **Synchronisation.** The CPU has caches. The device may have its own. The kernel has ordering rules. The sync calls translate the driver's intent ("I am about to hand this buffer to the device for reading") into whatever cache flushes, invalidations, or memory barriers the architecture requires.

**Why this matters to driver writers.** Skipping synchronisation is the single most common source of subtle data corruption in a DMA driver. The code appears to work because the CPU caches happen to be flushed by other activity; then a kernel update changes cache behaviour slightly and the driver begins to fail. Writing every sync call every time is the only reliable habit.

**How this shows up in FreeBSD.** The four sync flags are `BUS_DMASYNC_PREREAD`, `BUS_DMASYNC_PREWRITE`, `BUS_DMASYNC_POSTREAD`, and `BUS_DMASYNC_POSTWRITE` (values 1, 4, 2, 8 in `/usr/src/sys/sys/bus_dma.h`). "PRE" runs before the device touches the buffer; "POST" runs after. The "READ" and "WRITE" halves follow the wording in `bus_dma(9)`: `PREREAD` is "the device is about to update host memory, and the CPU will later read what was written"; `PREWRITE` is "the CPU has updated host memory, and the device is about to access it". In the pair of directions a driver usually thinks about, `PREWRITE` covers the CPU-writes-buffer-then-device-reads case, and `PREREAD` covers the device-writes-buffer-then-CPU-reads case.

**Common trap.** Reading the flag names as if they described the device's own action. If the device is about to write to memory, and the CPU will later read what the device wrote, you call `BUS_DMASYNC_PREREAD` first (because the CPU will later read this memory), not `BUS_DMASYNC_PREWRITE`. The flag pairs track the memory operation the CPU is part of, not the direction the device happens to move data on the wire.

**Where the book teaches this.** Chapter 21 spends substantial space on the sync semantics because they are easy to get wrong.

**What to read next.** `bus_dma(9)` and the header comments in `/usr/src/sys/sys/bus_dma.h`.

### 41.6.4 DMA Buffers vs Control Buffers

**What it is.** A final distinction that keeps the memory section and the DMA section in step. Not every buffer in a DMA-capable driver is a DMA buffer.

- **DMA buffers** are the ones the device reads from or writes to. They go through `bus_dma(9)`: allocated with `bus_dmamem_alloc` or `malloc` plus `bus_dmamap_load`, loaded, synchronised, unloaded.
- **Control buffers** are the ones only the CPU touches: ring indices, per-request bookkeeping, command structures the driver inspects but the hardware does not. They live in ordinary `malloc(9)` memory.

**Why this matters to driver writers.** Both kinds of buffer usually exist in the same driver, often side by side inside the same softc. Keeping them straight makes the code much easier to review. A pointer that the device reads should be labelled and allocated unambiguously; a pointer the device never sees should be labelled and allocated just as unambiguously. Confusing the two usually produces hard-to-reproduce bugs.

**Where the book teaches this.** Chapter 21 revisits the split when it walks a complete descriptor-ring driver.

## 41.7 Quick-Reference Tables

The compact tables below are meant for scanning. They do not replace the sections above; they help you point at the right section fast.

### 41.7.1 Address Spaces at a Glance

| You have...                | It lives in...                         | How you get it                                                                               | Who can dereference it                           |
|----------------------------|----------------------------------------|----------------------------------------------------------------------------------------------|--------------------------------------------------|
| A C pointer in kernel code | Kernel virtual memory                  | <code>malloc</code> , <code>uma_zalloc</code> ,<br>stack, <code>softc</code>                 | CPU                                              |
| A device register          | Physical I/O space                     | <code>bus_alloc_resource(SYS_CHS_MEMORY, space_*</code><br><code>+ bus_space_handle_t</code> | CPU                                              |
| A DMA buffer               | Kernel virtual, plus a bus view        | <code>bus_dmamem_alloc</code>                                                                | CPU via virtual, device via <code>ds_addr</code> |
| A device-side bus address  | Bus address space (maybe IOMMU-mapped) | <code>bus_dmamap_load</code><br>callback                                                     | Device only                                      |

### 41.7.2 When to Use Which Allocator

| Purpose                                        | Call                                                              |
|------------------------------------------------|-------------------------------------------------------------------|
| <code>Softc</code> , lists, control structures | <code>malloc(9)</code> with <code>M_DRIVER_NAME</code>            |
| Frequent fixed-size objects                    | <code>uma(9)</code> zone                                          |
| DMA buffer with tag constraints                | <code>bus_dmamem_alloc(9)</code> under a <code>bus_dma</code> tag |
| Physically contiguous block, no DMA tag        | <code>contigmalloc(9)</code> (rarely)                             |
| Stack temporary, small and bounded             | Plain C stack                                                     |

### 41.7.3 Interrupt vs Polling

| You should use...                      | When...                                                   |
|----------------------------------------|-----------------------------------------------------------|
| A hardware interrupt                   | The event rate is moderate and latency matters            |
| Polling inside an <code>ithread</code> | The event rate is very high and interrupts would dominate |

| You should use...                            | When...                                        |
|----------------------------------------------|------------------------------------------------|
| A taskqueue triggered by interrupts          | The work is heavy and cannot run in the filter |
| A timer (callout) plus occasional interrupts | The device is slow and state is simple         |

#### 41.7.4 Bus at a Glance (Driver-Writer View)

| Bus        | Driver form                       | Identification                       | Main API call                                                          |
|------------|-----------------------------------|--------------------------------------|------------------------------------------------------------------------|
| PCI / PCIe | Full Newbus driver with registers | Vendor/Device ID + class             | <code>bus_alloc_resource_any</code> ,<br><code>bus_setup_intr</code>   |
| USB        | Framework class driver            | VID/PID +<br>class/subclass/protocol | <code>usbd_transfer_setup</code> ,<br><code>usbd_transfer_start</code> |
| I2C        | <code>iicbus</code> client        | Slave address                        | <code>iicbus_transfer</code>                                           |
| SPI        | <code>spibus</code> client        | Chip-select line                     | <code>spibus_transfer</code>                                           |

#### 41.7.5 DMA Sync Quick Check

| You are about to...                                        | Call this                                                         |
|------------------------------------------------------------|-------------------------------------------------------------------|
| Hand a buffer to the device for the device to <i>read</i>  | <code>bus_dmamap_sync( . . . , BUS_DMASYNC_PREWRITE )</code>      |
| Hand a buffer to the device for the device to <i>write</i> | <code>bus_dmamap_sync( . . . , BUS_DMASYNC_PREREAD )</code>       |
| Read from a buffer the device just wrote                   | <code>bus_dmamap_sync( . . . , BUS_DMASYNC_POSTREAD )</code>      |
| Reuse a buffer after the device read it                    | <code>bus_dmamap_sync( . . . ,<br/>BUS_DMASYNC_POSTWRITE )</code> |

The “PRE” calls fence the CPU writes or cache lines before the device looks at the buffer. The “POST” calls fence the device’s completion before the CPU looks at what changed.

### 41.8 Wrapping Up: How to Keep Relating Hardware Concepts to Driver Code

The hardware concepts in this appendix are not a separate subject from driver writing. Every one of them shows up in code. The sequence the reader will use, over and over, is always the same:

1. The device exists in physical address space. Its registers are at physical addresses.
2. The kernel gives the driver a `bus_space` handle that maps those registers with the right attributes.
3. The device raises interrupts on events, and the driver handles them through a filter (fast, small, spin-lock safe) and optionally an `ithread` (slower, sleep-lock safe).



4. When the device needs to move data, the driver uses `bus_dma(9)` to produce bus addresses that the device can use, and it brackets every transfer with sync calls.
5. The driver never hands a kernel virtual pointer to the device, and never treats a `ds_addr` as if it were a physical address the CPU can read directly.

If you keep that sequence in mind, the hardware concepts here stop being a separate subject and become a running commentary on the code you are writing. When you are about to cast a pointer and hand it to hardware, stop and translate the action into the sequence: in which address space is this, who is going to read it, what attributes does that view need. The sequence never changes. The details change with every device family.

Three habits that reinforce the model.

The first is reading the header comment of a real driver before reading its code. Most FreeBSD drivers open with a block that explains the locking discipline, the MSI-X structure, or the ring layout. That comment is there because the code alone cannot communicate it. When the comment says “the receive ring uses the per-queue lock; the filter acknowledges the device and schedules a taskqueue”, read it carefully. It is a condensed map of the hardware-to-software bridge for that driver.

The second is naming the bus every time. When you read an unfamiliar driver, find the `DRIVER_MODULE` line. The bus name in that line (`pci`, `usb`, `iicbus`, `spibus`, `acpi`, `simplebus`) tells you which hardware model applies before you read a single function. A driver attached to `pci` is register-oriented and interrupt-driven; a driver attached to `iicbus` is transaction-oriented and polled at the bus level. The same C code reads differently when you know which bus is behind it.

The third is keeping a short companion sheet with your driver notes. The files under `examples/appendices/appendix-c-hardware-concepts-for-driver-developers/` are meant for exactly this. A bus-comparison cheatsheet you can glance at when you open a new datasheet. A DMA mental-model checklist you can walk when you suspect a sync is missing. A physical-vs-virtual diagram you can annotate with the particular tag and handle pair you are working with today. The teaching is in this appendix; the application is in those sheets.

With that, the hardware side of the book has a consolidated home. The chapters keep teaching; the appendix keeps naming; the examples keep reminding. When a reader closes this appendix and opens a real driver, the vocabulary is ready.

## 42 Appendix D: Operating System Concepts

### 42.1 How to Use This Appendix

The main chapters teach the reader how to write a FreeBSD device driver: how to declare a softc, how to probe and attach to a bus, how to register a character device, how to set up an interrupt, how to do DMA. Beneath all of that there is a running assumption that the reader has a working mental model of what an operating system kernel actually is, how it differs from userland, what a driver is *inside* that kernel, and how the machine gets from power-on to a kernel that is ready to load and run that driver. The book introduces each piece of this model in the chapter where it first becomes relevant, but the pieces never appear in one place together.

This appendix is that one place. It is a concept companion for the operating-system ideas the rest of the book keeps reusing: the kernel/user boundary, the definition of a driver, the handful of data structures you will see again and again in FreeBSD source, and the boot-to-init sequence that sets the stage for everything a driver can do. It is deliberately short. The goal is to consolidate the mental model, not to teach operating-systems theory, and not to repeat what the chapters already teach at length.

#### 42.1.1 What You Will Find Here

The appendix is organised by driver relevance, not by textbook taxonomy. Four sections, each with a small set of entries. Each entry follows the same compact rhythm:

- **What it is.** One or two sentences of plain definition.
- **Why this matters to driver writers.** The concrete place where the concept shows up in your code.
- **How this shows up in FreeBSD.** The API, header, subsystem, or convention that names the idea.
- **Common trap.** The misunderstanding that actually costs people time.
- **Where the book teaches this.** A pointer back to the chapter that uses it in context.
- **What to read next.** A manual page, a header, or a real source file you can open.

Not every entry uses every label; the shape is a guide, not a template.

#### 42.1.2 What This Appendix Is Not

It is not an introduction to operating-systems theory. If you have never met the idea of a process or an address space before, pick up a general operating-systems textbook first and come back. It is not a systems-programming tutorial either; the early chapters of the book already handle that. And it does not overlap with the other appendices. Appendix A is the API lookup. Appendix B is the algorithmic-pattern field guide. Appendix C is the hardware-concepts companion. Appendix E is the kernel-subsystem reference. If the question you want answered is “what does this macro call do”, “what data structure should I use”, “what is a BAR”, or “how does the scheduler work”, you want a different appendix. This one keeps its focus on the operating-system mental model that makes driver work make sense.

## 42.2 Reader Guidance

Three ways to use this appendix, each with a different reading strategy.

If you are **learning the main chapters**, keep this appendix open as a companion. When Chapter 3 discusses the UNIX division between the kernel and userland, glance at Section 1 here for the driver-relevant summary. When Chapter 6 introduces Newbus and the driver lifecycle, Section 2 compresses the same story into a one-page mental model. When Chapter 11 walks you through your first mutex, Section 3 names the family. A first pass through the whole appendix should take about twenty-five minutes; daily use looks more like two or three minutes at a time.

If you are **reading unfamiliar kernel code**, treat the appendix as a translator. Kernel source assumes the reader already knows why `copyin(9)` exists, what a `softc` is, when a `TAILQ` is better than a `LIST`, and where `SYSINIT` chains come from. If any of those words feel fuzzy when you meet them in code, the entry here names the concept in one page and points to the chapter that teaches it in full.

If you are **coming back to the book after time away**, read the appendix as a consolidation. The concepts here are the recurring spine of driver work. Re-reading them is a low-cost way to reload the mental model before you open an unfamiliar driver.

A few conventions apply throughout:

- Source paths are shown in the book-facing form, `/usr/src/sys/...`, matching a standard FreeBSD system. You can open any of them on your lab machine.
- Manual pages are cited in the usual FreeBSD style. Kernel-facing pages live in section 9 (`intr_event(9)`, `mtx(9)`, `kld(9)`). Userland system calls live in section 2 (`open(2)`, `ioctl(2)`). Device overviews live in section 4 (`devfs(4)`, `pci(4)`).
- When an entry points to real source as reading material, the file is one a beginner can navigate in a single sitting.

With that framing in place, we start where every driver must start: by understanding which side of the kernel/user line it lives on.

## 42.3 Section 1: What Is a Kernel, and What Does It Do?

A kernel is the program that owns the machine. It is the only program that runs in the CPU's most privileged mode, the only program that may touch physical memory directly, the only program that may talk to hardware, and the only program that may decide which other programs run and when. Everything else (shells, compilers, web browsers, daemons, your own programs) runs in a less privileged mode and asks the kernel for the things it cannot do itself. A driver is kernel code. That single fact changes the rules the code must follow, and the rest of this section explains why.

### 42.3.1 Kernel Responsibilities in One Page

**What it is.** The FreeBSD kernel is responsible for a small, well-defined set of jobs: managing the CPU (processes, threads, scheduling, interrupts), managing memory (virtual address spaces, page tables, allocation), managing I/O (talking to devices through drivers), managing files and filesystems (through VFS and GEOM), and

managing the network stack. It also mediates security boundaries, handles signals, and serves as the only entry point for every system call.

**Why this matters to driver writers.** A driver is not a free-standing program. It is a participant in each of those jobs. A network driver participates in the networking subsystem. A storage driver participates in the I/O and filesystem subsystems. A sensor driver participates in the I/O subsystem and sometimes in the event-handler subsystem. When you write a driver, you are extending the kernel, not sitting beside it. The kernel's invariants become your invariants, and every convention the kernel relies on (locking discipline, memory discipline, error-return conventions, cleanup ordering) becomes a convention you must honour.

**How this shows up in FreeBSD.** The kernel source tree under `/usr/src/sys/` is organised by responsibility: `/usr/src/sys/kern/` holds the machine-independent core (process management, scheduling, locking, VFS), `/usr/src/sys/vm/` holds the virtual-memory system, `/usr/src/sys/dev/` holds the device drivers, `/usr/src/sys/net/` and `/usr/src/sys/netinet/` hold the network stack, and `/usr/src/sys/amd64/` (or `arm64/`, `i386/`, `riscv/`) holds the machine-dependent code. When you read a driver, you are reading code that ties together pieces from several of these directories.

**Common trap.** Treating the kernel as a passive library that drivers “call into”. The kernel is the running program; your driver lives inside it. A crash, a memory leak, or a held lock in your driver is a crash, a memory leak, or a held lock for the whole system.

**Where the book teaches this.** Chapter 3 introduces the UNIX and FreeBSD picture. Chapter 5 sharpens the distinction at the level of kernel-space C. Chapter 6 grounds it in driver anatomy.

**What to read next.** `intro(9)` for the top-level kernel overview, and the directory listing of `/usr/src/sys/` itself. Spend five minutes reading folder names and you will already have a rough map.

### 42.3.2 Kernel Space vs User Space

**What it is.** Two distinct execution environments inside the same machine. Kernel space runs with full CPU privileges, shares one address space among all kernel threads, and has direct access to hardware. User space runs with reduced privileges, lives in per-process virtual address spaces that the kernel sets up and protects, and has no direct access to hardware. A single machine is always running code from both environments; the CPU switches between them many times per second.

**Why this matters to driver writers.** Almost every rule that makes kernel programming feel different from user programming comes from this separation. A kernel pointer is not addressable from user space, and a user pointer is not safely addressable from kernel space. User memory may be swapped, paged, or unmapped at any moment; kernel memory is stable within its allocation lifetime. The kernel must never blindly dereference a pointer that came from userland; it must always use the dedicated copy primitives. And because one kernel address space is shared by every thread in the kernel, any kernel bug that stomps memory can corrupt every subsystem at once.

**How this shows up in FreeBSD.** Three places in driver code are visible every time the boundary is crossed:

- `copyin(9)` and `copyout(9)` move fixed-size data between a user buffer and a kernel buffer, returning an error on a bad user address instead of panicking.
- `copyinstr(9)` does the same for NUL-terminated strings with a caller-supplied length cap.
- `uiomove(9)` walks a `struct uio` (user I/O descriptor) through either direction of a read or write system call, handling the buffer list and the direction flag for you.

Under the hood, each of these routines uses machine-dependent helpers that know how to catch a fault from a bad user address and convert it into an `EFAULT` return rather than a kernel panic.

**Common trap.** Dereferencing a user pointer directly inside a syscall handler or an `ioctl` implementation. The code may appear to work when the user passes a valid pointer and the page happens to be resident. It will panic or silently misbehave the moment the user passes something bad (either by mistake or deliberately), and that is one of the easier bugs to turn into a security problem.

**Where the book teaches this.** Chapter 3 sets up the distinction. Chapter 5 covers it in kernel C. Chapter 9 walks the boundary in character-driver code.

**What to read next.** `copyin(9)`, `copyout(9)`, `uiomove(9)`, and the header `/usr/src/sys/sys/uio.h` for `struct uio`.

### 42.3.3 System Calls: The Door Between the Two Sides

**What it is.** A system call is the disciplined mechanism a user-space program uses to ask the kernel to do something on its behalf. The user program invokes a special instruction (`syscall` on amd64, `svc` on arm64, with equivalent traps on every other architecture FreeBSD supports), which causes the CPU to trap into kernel mode, switch to the kernel stack of the current thread, and dispatch to the implementation the kernel has registered for that system-call number.

**Why this matters to driver writers.** Most driver interaction with userland happens because a userland program issued a system call. When the user runs `cat /dev/mydevice`, `cat(1)` invokes `open(2)`, `read(2)`, and `close(2)` against that device node. Each of those system calls eventually reaches your driver through the `cdev` switch table you registered with `make_dev_s(9)`: `open(2)` becomes `d_open`, `read(2)` becomes `d_read`, `ioctl(2)` becomes `d_ioctl`. Your driver's entry points are therefore system-call entry points in disguise, which is exactly why the copy primitives above matter so much.

**How this shows up in FreeBSD.** Two pieces. First, the system-call table itself and the trap-handling path, which live in the machine-independent core and in the architecture-specific backends. Second, the driver-visible structures that a system call hands you: a `struct thread *` for the calling thread, a `struct uio *` for the buffer list on `read/write`, an argument pointer for `ioctl`, and a set of flags that describe the mode of the call. The driver does not see the trap; it sees a function call that arrives already in kernel context.

**Common trap.** Assuming the calling thread can be treated as an ordinary kernel thread. A thread that entered the kernel through a system call is temporarily running kernel code on behalf of a user process. It may be killed if the process is killed, its priority may be different from an internal kernel thread, and it is usually the wrong thread to own long-lived work. Long work belongs on a taskqueue or a kernel thread the driver owns.

**Where the book teaches this.** Chapter 3 names the mechanism. Chapter 9 exercises it in a working `read/write` path.

**What to read next.** `syscall(2)` for the userland view, `uiomove(9)` for the kernel entry shape, and `/usr/src/sys/kern/syscalls.master` if you are curious about the actual system-call numbering table.

### 42.3.4 Why Kernel Mistakes Have System-Wide Consequences

**What it is.** A short way of stating what the previous entries imply. A kernel thread can read and write any kernel memory; there is no per-module address space inside the kernel. A kernel bug is therefore free to damage any

data structure, including data structures that belong to other subsystems. A kernel crash takes the whole machine down. A kernel deadlock can wedge threads across every driver at once.

**Why this matters to driver writers.** The blast radius of a kernel bug is the whole system. That is why the book repeats the same disciplines in every chapter: balanced allocate/free pairs, balanced lock acquire/release pairs, consistent cleanup ladders on error, defensive validation of any pointer that comes from userland, and careful ordering at boundaries (interrupts, DMA, device registers). None of these disciplines are about heroics; they are about acknowledging that the cost of a kernel mistake is paid by every process on the machine, not only by the driver that made it.

**How this shows up in FreeBSD.** Directly, in the tools the kernel offers you to catch mistakes before they escape. `witness(9)` verifies lock ordering. `KASSERT(9)` lets you encode invariants that compile away in production but fire loudly in a development build. `INVARIANTS` and `INVARIANT_SUPPORT` in the kernel config turn on extra self-checks. `DTrace` lets you watch driver behaviour under real load. Your development lab should always be running a kernel with these diagnostics enabled; your production target should not.

**Common trap.** Testing only on a release-build kernel and discovering the bugs in production. The development kernel is the one that finds the problem cheaply. Use it.

**Where the book teaches this.** Chapter 2 walks you through building a diagnostics-enabled lab kernel. Chapter 5 makes the discipline concrete at the C level. Every chapter from Chapter 11 onward depends on these habits.

**What to read next.** `witness(9)`, `kassert(9)`, `ddb(4)`, and the kernel-debugging options in `/usr/src/sys/conf/NOTES`.

### 42.3.5 How Drivers Participate in Kernel Responsibilities

The previous entries describe the kernel as an abstract entity. The role a driver plays inside that kernel is concrete and limited. A driver is the component that makes a specific piece of hardware, or a specific pseudo-device, usable through the kernel's existing subsystems. A storage driver exposes a disk through GEOM and the buffer cache so that filesystems can use it. A network driver exposes a NIC through `ifnet` so that the network stack can use it. A character driver exposes a device node in `/dev` so that userland programs can use it through ordinary file system calls. Drivers do not invent interfaces; they implement the ones the kernel already publishes.

That framing is worth keeping close while you read a new driver. Finding the two ends, the hardware end (registers, interrupts, DMA) and the subsystem end (the `ifnet`, the `cdev`, the GEOM provider), is the single best way to orient yourself in an unfamiliar driver. Section 2 puts more shape on the subsystem end.

## 42.4 Section 2: What Is a Device Driver?

A device driver is the piece of kernel code that makes a particular device, real or virtual, usable through the operating system's regular interfaces. It lives on two fronts at once. On one side it speaks the hardware's language: register accesses, interrupts, DMA descriptors, protocol sequences. On the other side it speaks one of FreeBSD's standard subsystem interfaces: `cdevsw` for character devices, `ifnet` for network interfaces, GEOM for storage. A driver's job is to translate between those two sides, correctly and safely, for the lifetime of the device.

This section names the driver types the book cares about, the role the driver plays in the hardware-to-userland path, and the mechanics by which FreeBSD loads a driver and binds it to a device.

### 42.4.1 Driver Types: Character, Block, Network, Pseudo

**What it is.** A classification by the subsystem interface the driver publishes. The vocabulary is older than FreeBSD and the historical category names still appear, but on modern FreeBSD the exact boundaries differ from the textbook picture.

- A **character device driver** exposes the hardware as a stream through a `cdev` node in `/dev`. It registers a `struct cdevsw` with entry points such as `d_open`, `d_close`, `d_read`, `d_write`, and `d_ioctl`, and the kernel routes corresponding system calls to those entry points. Pseudo-devices (drivers with no real hardware behind them, like `/dev/null` or `/dev/random`) are character devices too.
- A **block device driver** in the historical UNIX sense exposed a fixed-size disk through a block-oriented cache. In modern FreeBSD, disk-like devices are presented through GEOM, not through a separate block-`cdevsw`; the entries in `/dev` for a disk (`/dev/ada0`, `/dev/da0`) are still `cdev` nodes, but their substance is a GEOM provider, and filesystems talk to them through the BIO layer rather than through raw read/write.
- A **network driver** exposes a network interface through `ifnet`. It does not use `cdevsw` at all; it registers with the network stack, receives packets as `struct mbuf` chains, sends packets through the stack's transmit queues, and participates in link-state events.
- A **pseudo-device driver** is any driver with no real hardware behind it. It may be a character device (`/dev/null`, `/dev/random`), a network device (`tun`, `tap`, `lo`), or a storage provider (`md`, `gmirror`). The word “pseudo” refers only to the absence of hardware; the subsystem integration is the same as for a real device.

**Why this matters to driver writers.** The type of driver you are writing determines the interface you must implement. A character-device driver spends its time in `d_open`, `d_close`, `d_read`, `d_write`, `d_ioctl`. A network driver spends its time in `if_transmit`, `if_input`, and interrupt-driven receive paths. A storage driver spends its time in GEOM start routines and BIO completion. Picking the right type early saves a rewrite; picking the wrong one is an expensive mistake.

**How this shows up in FreeBSD.** Three separate sets of headers and chapters of the tree handle the three main types:

- Character drivers register through `/usr/src/sys/sys/conf.h` (`struct cdevsw`, `make_dev_s`) and are implemented all across `/usr/src/sys/dev/`.
- Network drivers register through `/usr/src/sys/net/if.h` (`struct ifnet`) and have their stack integration in `/usr/src/sys/net/` and `/usr/src/sys/netinet/`.
- Storage drivers register through the GEOM framework under `/usr/src/sys/geom/` and use `struct bio` for I/O requests.

**Common trap.** Reaching for `cdevsw` to expose a disk. The “everything is a file” intuition is half right, but a modern FreeBSD disk is a GEOM provider; a raw `cdevsw` interface skips the cache, the partitioning layer, and the transforms the rest of the system expects.

**Where the book teaches this.** Chapter 6 names the categories and positions them inside Newbus. Chapter 7 writes the first character driver from scratch. Chapter 27 covers storage and GEOM. Chapter 28 covers network drivers.

**What to read next.** `cdevsw(9)`, `ifnet(9)`, and a compact pseudo-device such as `/usr/src/sys/dev/null/null.c`. Read the `null` driver in one sitting; it is the cleanest example of a complete character driver in the tree.

### 42.4.2 The Role of a Driver in Hardware Communication

**What it is.** The driver is the only kernel code that talks to its hardware. Nothing else in the kernel accesses the device's registers, arms its interrupts, or queues its DMA transfers. All other subsystems reach the device through the driver.

**Why this matters to driver writers.** This exclusivity is why the driver must be trustworthy. The network stack cannot double-check whether a packet was really transmitted; it trusts the network driver. The filesystem cannot double-check whether a block really reached the disk; it trusts the storage driver. When the driver lies (by completing a transfer before the hardware is finished, for example), the rest of the system believes the lie and carries on. Most catastrophic failures in real systems come from this class of subtle driver dishonesty, not from outright crashes.

**How this shows up in FreeBSD.** In the shape of the driver's public functions. A character driver's `d_read` returns only after the data is really in the user buffer or an error has occurred. A network driver's `if_transmit` tells the stack whether the packet is owned by the driver now or still by the stack. A storage driver's `g_start` takes responsibility for completing the `bio` through `biodone(9)` eventually. Every one of those contracts is a promise the driver is making on behalf of the hardware.

**Common trap.** Returning early from a completion path because "the hardware usually works". On a good day, nothing fails. On a bad day, the stack thinks a packet was sent that was not, a filesystem thinks a block was written that was not, and the resulting corruption is impossible to track back to its source.

**Where the book teaches this.** Chapter 6 sets up the contract. Chapter 19 makes it rigorous for interrupt-driven completion. Chapter 21 does the same for DMA.

**What to read next.** `biodone(9)`, and the transmit/receive path of a small NIC driver.

### 42.4.3 How Drivers Are Loaded and Bound to Devices

**What it is.** FreeBSD drivers reach a running kernel in one of two ways. Either they are compiled into the kernel binary (built-in drivers, activated during boot) or they are built as kernel-loadable modules (`.ko` files) that the administrator loads at run time with `kldload(8)`. Once a driver is present in the kernel, Newbus offers it the devices it claims.

The binding dance is the *probe/attach* cycle. Every bus enumerates its children at boot or on hot-plug and asks each registered driver whether it can handle each child. The driver's `probe` function examines the child (vendor/device ID for PCI, VID/PID for USB, FDT compatibility string for embedded systems, a class register for generic buses) and returns a priority value saying how confident it is. The bus picks the highest-priority match and calls that driver's `attach` function. From that point on, the device belongs to the driver, and the driver's `softc` holds the device's state.

**Why this matters to driver writers.** Every FreeBSD driver ends with a short block of boilerplate that registers it for this dance. The boilerplate is the contract between the driver and the rest of the kernel. Get it wrong and the driver never attaches; get it right and the kernel will attach, detach, suspend, resume, and unload your driver at the moments it chooses.

**How this shows up in FreeBSD.** Five pieces, all compact:

- A `device_method_t` array that lists the driver's implementations of the standard device methods, ending with `DEVMETHOD_END`.



- A `driver_t` structure that binds a driver name, the method table, and a softc size.
- A `DRIVER_MODULE(name, busname, driver, evh, arg)` invocation at the bottom of the file, which registers the driver with the named bus. The `evh` and `arg` fields pass an optional event handler; most drivers leave them as `0, 0`.
- A `MODULE_VERSION(name, version)` invocation to advertise the module's ABI version.
- Optional `MODULE_DEPEND(name, other, min, pref, max)` lines if the driver depends on another module.

These live in `/usr/src/sys/sys/module.h` and `/usr/src/sys/sys/bus.h`. The `DEVMETHOD` macro hooks the driver's `probe`, `attach`, `detach`, and optional `suspend/resume/shutdown` functions into the `kobj` method dispatcher that Newbus uses internally.

**Common trap.** Forgetting `MODULE_VERSION`. The module may load and work on your machine, and then fail to load on someone else's because the kernel has no way to check ABI compatibility. The version declaration is not ceremonial; it is how the module loader keeps pieces compatible.

**Where the book teaches this.** Chapter 6 introduces the Newbus tree. Chapter 7 writes the full boilerplate end to end and uses `kldload(8)` to exercise it.

**What to read next.** `DRIVER_MODULE(9)`, `MODULE_VERSION(9)`, `kld(9)`, `kldload(8)`, and the compact `/usr/src/sys/dev/null/null.c` as a complete example.

#### 42.4.4 Connecting Hardware, Kernel Subsystems, and Userland

**What it is.** A short way of stating what a driver does viewed from the top. Hardware is on one side. Userland is on the other. In between are the kernel subsystems (`VFS`, `GEOM`, `ifnet`, `cdev`). The driver is the component that lets bytes flow through that sandwich.

A rough picture, in one direction:

| userland | kernel subsystems                  | driver                         | hardware   |
|----------|------------------------------------|--------------------------------|------------|
| -----    | -----                              | -----                          | -----      |
| cat(1)   | -> open(2) / read(2)               |                                |            |
|          | -> cdev switch                     |                                | ADC, NIC,  |
|          | (d_read)                           | -> driver entry point          | disk, etc. |
|          |                                    | -> register read, MMIO, DMA    |            |
|          |                                    |                                | -> bytes   |
|          |                                    | <- sync, interrupt, completion |            |
|          | <- mbuf chain / bio / uiomove back |                                |            |
|          | <- read(2) returns                 |                                |            |

The picture is not literal code. It is the shape of responsibility. Every driver fits some variant of it.

**Why this matters to driver writers.** When you are unsure where a piece of code belongs, ask which layer of the sandwich is responsible for the current concern. A user-pointer validation belongs at the subsystem boundary. A register access belongs in the driver. A per-device state update belongs in the softc. Once the layer is clear, the names and primitives follow.

**Where the book teaches this.** Chapter 6 draws the picture for character drivers. Chapter 27 redraws it for GEOM. Chapter 28 redraws it for `ifnet`. The shape is always recognisable.

**What to read next.** The top of `/usr/src/sys/dev/null/null.c` for a minimal character variant, and the top of `/usr/src/sys/net/if_tuntap.c` for a minimal network variant.

## 42.5 Section 3: Kernel Data Structures You Will See Often

Certain shapes appear in nearly every FreeBSD driver. A linked list of pending requests. A buffer that crosses the kernel/user boundary. A lock around the softc. A condition variable waited on from one thread and signalled from another. None of these is complicated in isolation, but meeting them for the first time inside real code is where the mental model often breaks down. This section names the families, explains what role they play, and points you back to the chapter that teaches them in full. Appendix A has the detailed APIs; Appendix B has the algorithmic patterns. This section is the one-page orientation.

### 42.5.1 Lists and Queues from `<sys/queue.h>`

**What it is.** A header-only family of intrusive-list macros. You embed a `TAILQ_ENTRY`, `LIST_ENTRY`, or `STAILQ_ENTRY` inside your element structure, define a head, and the macros give you insertion, removal, and traversal without any heap allocation for list nodes. Four flavours exist: `SLIST` (singly linked), `LIST` (doubly linked, insertion at head), `STAILQ` (singly linked with fast tail insertion), and `TAILQ` (doubly linked with tail insertion and  $O(1)$  arbitrary removal).

**Why this matters to driver writers.** Almost every per-driver collection in the tree is one of these. Pending commands, open file handles, registered callbacks, queued bio requests. Picking the right flavour for the access pattern produces code that is short, predictable, and lockable with a single mutex. Picking the wrong one produces needless pointer bookkeeping or surprise  $O(n)$  removal.

**How this shows up in FreeBSD.** The macros live in `/usr/src/sys/sys/queue.h`. A softc often holds its queue heads as ordinary fields. Every traversal is wrapped by the driver's lock. The safe-against-removal variants (`TAILQ_FOREACH_SAFE`, `LIST_FOREACH_SAFE`) are the ones you want any time the body of the loop might unlink the current element.

**Common trap.** Using a plain `LIST_FOREACH` while freeing elements inside the loop body. The iterator dereferences the element's link pointer after you have already freed it. Switch to the `_SAFE` variant or rearrange the loop.

**Where the book teaches this.** Chapter 5 introduces the macros in kernel C. Chapter 11 uses them with locking. Appendix B has the full pattern-oriented treatment.

**What to read next.** `queue(3)`, `/usr/src/sys/sys/queue.h`, and any driver softc that holds an in-flight request list.

### 42.5.2 Kernel Buffers: mbuf, buf, and bio

**What it is.** Three buffer representations that appear in specific subsystems. `struct mbuf` is the networking packet-representation type, a linked list of small fixed-size units that together hold a packet and its metadata. `struct buf` is the storage buffer-cache unit, used by the VFS and the legacy block layer. `struct bio` is the GEOM I/O-request structure, the modern unit of storage I/O that a storage driver completes with `biodone(9)`.

**Why this matters to driver writers.** You will not meet all three at once. A network driver thinks in mbuf chains. A storage driver thinks in bio requests and, at the VFS layer, in buf entries. A character driver rarely sees any of them; it uses `struct uio` instead. Knowing which buffer type your subsystem uses tells you which completion convention applies and which headers to include. Mixing them up is a signal that the driver type is wrong or that a layer boundary is being violated.

**How this shows up in FreeBSD.** `struct mbuf` is defined in `/usr/src/sys/sys/mbuf.h`. `struct buf` is defined in `/usr/src/sys/sys/buf.h`. `struct bio` is defined in `/usr/src/sys/sys/bio.h`. Each has its own allocation, reference-counting, and completion rituals; the driver never invents a substitute.

**Common trap.** Allocating a plain byte buffer for network transmit because it looks easier. The network stack hands you an mbuf; the network stack expects an mbuf back. Converting in and out of the right type is almost always a mistake, and it breaks zero-copy fast paths that the stack relies on.

**Where the book teaches this.** Chapter 27 introduces bio in the storage context. Chapter 28 introduces mbuf in the network context. Appendix E will have deeper entries for each.

**What to read next.** `mbuf(9)`, `bio(9)`, `buf(9)`.

### 42.5.3 Locking: Mutex, Spin Mutex, Sleep Lock, Condition Variable

**What it is.** A family of kernel primitives for ordering access to shared data. Four kinds matter at the appendix level:

- A **default mutex** (`mtx(9)` with `MTX_DEF`) is the everyday sleep-capable mutex. A thread that cannot immediately acquire it goes to sleep until the owner releases. It is the right default for almost every softc field.
- A **spin mutex** (`mtx(9)` with `MTX_SPIN`) is an interrupt-safe busy-wait lock. A thread that cannot immediately acquire it spins. Spin mutexes are used in the tiny set of contexts where sleeping is forbidden (interrupt filters and a few scheduler paths). They are not a performance optimisation; they are a correctness tool for a narrow case.
- A **sleepable shared/exclusive lock** (`sx(9)`) is the right choice for long read-mostly critical sections that may sleep while holding the lock. It is used by code that performs operations the kernel might block on (for example, allocating memory with `M_WAITOK` while holding the lock).
- A **condition variable** (`cv(9)`) is a synchronisation primitive for waiting on a predicate. A thread holds a mutex, checks a condition, and if the condition is false, calls `cv_wait` which atomically drops the mutex, sleeps, and reacquires the mutex on wakeup. Another thread changes the predicate under the same mutex and calls `cv_signal` or `cv_broadcast`. A condition variable never exists without an accompanying mutex.

**Why this matters to driver writers.** Every non-trivial driver keeps shared state: the softc itself, an in-flight request list, a pending I/O counter, a reference count. Protecting that state correctly is the difference between a

driver that works under load and a driver that fails intermittently in ways nobody can reproduce. The FreeBSD locking primitives are the tools you reach for every time.

**How this shows up in FreeBSD.** The headers are `/usr/src/sys/sys/mutex.h` for the regular and spin mutexes, `/usr/src/sys/sys/sx.h` for the shared/exclusive lock, and `/usr/src/sys/sys/condvar.h` for condition variables. The convention is to store the lock inside the `softc` and name it after the `softc` field: a driver that protects its work queue usually has something like `mtx_init(&sc->sc_lock, device_get_nameunit(dev), NULL, MTX_DEF)` in `attach` and a matching `mtx_destroy` in `detach`.

**Common trap.** Two, both common. The first is taking a sleep mutex from an interrupt filter. A filter runs in a context that cannot sleep; a sleep mutex may block; the kernel detects this with `witness(9)` in a development build. The second is reading the state the condition variable guards *outside* the mutex. The mutex-predicate-wait pattern relies on the predicate and the wait being atomic with respect to the signaller; breaking that atomicity reintroduces the race the condition variable was supposed to eliminate.

**Where the book teaches this.** Chapter 11 introduces mutexes and spin mutexes. Chapter 12 adds condition variables and shared/exclusive locks. Appendix B collects the reasoning patterns.

**What to read next.** `mtx(9)`, `sx(9)`, `cv(9)`, `witness(9)`, and the top of a real driver's `softc` structure to see the lock and its users in one place.

#### 42.5.4 `softc`: The Per-Device Driver Structure

**What it is.** The per-device state structure that the driver defines and the kernel allocates on its behalf. “Softc” is short for “software context”. It holds everything the driver needs to know about one specific instance of the device it drives: the `device_t` back-pointer, the lock, the `bus_space` tag and handle for MMIO, the allocated resources, any in-flight state, counters, and `sysctl` nodes. When FreeBSD attaches a driver to a device, it allocates a `softc` of the size the driver declared in its `driver_t` and gives the driver a pointer to it through `device_get_softc(9)`.

**Why this matters to driver writers.** Every function in the driver reaches the `softc` the same way: `struct mydev_softc *sc = device_get_softc(dev)`. Everything specific to this particular device lives behind that pointer. A driver that attaches to three devices gets three independent `softcs`, each with its own lock, its own state, its own hardware view. Keeping the `softc` the single owner of per-device state is what makes the code reentrant across multiple devices.

**How this shows up in FreeBSD.** The pattern is uniform across the tree. In the driver:

- A `struct mydev_softc` definition at the top of the file.
- `.size = sizeof(struct mydev_softc)` in the `driver_t` initialiser.
- `struct mydev_softc *sc = device_get_softc(dev)` at the top of every function that receives a `device_t`.
- A `sc->sc_lock` field initialised in `attach` and destroyed in `detach`.

The kernel zeroes the `softc` before handing it to the driver, so a fresh `attach` always starts in a known state.

**Common trap.** Stashing state in file-scope globals because “there is only one device”. Then someone attaches two, or builds a VM with two, and the second device silently corrupts the first. Every driver, no matter how simple it looks, keeps state in the `softc`.

**Where the book teaches this.** Chapter 6 introduces the concept. Chapter 7 writes a full softc and uses it end to end. Every chapter from Chapter 8 onward assumes the pattern.

**What to read next.** `device_get_softc(9)`, the `softc` block at the top of `/usr/src/sys/dev/null/null.c`, and the `softc` of any small PCI driver.

### 42.5.5 Why These Structures Keep Reappearing

The families above (queues, buffers, locks, and the `softc`) recur because they correspond to the four questions every driver eventually has to answer. What pending work do I hold? What bytes am I moving? How do I serialise access to my own state? Where do I keep the state of one instance of my device? Once you recognise that framing, driver code starts to read very differently. The unfamiliar parts are the details of a specific device or bus; the familiar parts are the four answers, in four standard shapes, used by every driver in the tree.

## 42.6 Section 4: Kernel Compilation and Booting

A driver runs inside a kernel. That kernel did not appear from nothing; something had to compile it, something had to load it into memory, something had to call its entry point, and something had to set up the environment in which your driver's `attach` function eventually runs. This section traces that chain at concept level, so that when a bug happens early in the boot, you have a mental picture of what the system was trying to do.

### 42.6.1 The Boot Process Overview

**What it is.** The sequence of events from power-on to a running FreeBSD kernel with userland processes. On a typical x86 machine it is roughly: firmware (BIOS or UEFI), first-stage boot (`boot0` on BIOS, the UEFI boot manager on UEFI), the FreeBSD loader (`loader(8)`), the architecture-specific kernel entry (`hammer_time` on amd64, called from assembly), the machine-independent startup (`mi_startup`), the `SYSINIT` chain, and finally the first userland process (`/sbin/init`). On ARM64 and other architectures, the firmware and machine-dependent parts differ; the machine-independent kernel startup is the same.

**Why this matters to driver writers.** Two reasons. First, many driver failures happen during `SYSINIT` processing, before there is a console to print to, and you need to know where you are when that happens. Second, the order in which subsystems initialise determines when your driver's bus has enumerators, when memory allocators are available, when the root filesystem is mounted, and when it is safe to talk to userland. A driver that tries to allocate memory too early will fail; a driver that tries to open a file before the root filesystem is mounted will also fail. The `SYSINIT` mechanism names the stages explicitly so that ordering is visible in the code.

**How this shows up in FreeBSD.** Each stage has a file you can open. The boot loader lives under `/usr/src/stand/`; the amd64 early C entry is `hammer_time` in `/usr/src/sys/amd64/amd64/machdep.c` (called from `/usr/src/sys/amd64/amd64/locore.S`); the machine-independent startup is `mi_startup` in `/usr/src/sys/kern/init_main.c`; and `start_init` in the same file is what the `init` process runs when it is finally scheduled, execing `/sbin/init` as PID 1.

**Common trap.** Treating boot as opaque. The `SYSINIT` mechanism makes the stages explicit and inspectable, and a driver that understands its place in them is much easier to debug.

**Where the book teaches this.** Chapter 2 walks through boot in lab context. Chapter 6 connects the driver-loading sequence to the kernel's startup picture.

**What to read next.** `boot(8)`, `loader(8)`, `/usr/src/sys/kern/init_main.c`, and the `SYSINIT` section of `/usr/src/sys/sys/kernel.h`.

### 42.6.2 From Power-On to init: A Compact Timeline

A thumbnail picture of the whole sequence. The names on the right are the functions or manual pages you would open to look closer.

|                                                                                   |                                                                                     |  |
|-----------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|--|
| Stage                                                                             | Where to look                                                                       |  |
| Firmware: BIOS / UEFI starts executing                                            | (hardware / firmware)                                                               |  |
| First-stage boot: <code>boot0</code> / UEFI boot manager                          | <code>/usr/src/stand/</code>                                                        |  |
| FreeBSD loader: selects kernel and modules                                        | <code>loader(8)</code> , <code>loader.conf(5)</code>                                |  |
| Kernel entry (amd64): <code>hammer_time()</code> + <code>locore</code>            | <code>sys/amd64/amd64/machdep.c</code> ,<br>  <code>sys/amd64/amd64/locore.S</code> |  |
| MI startup: <code>mi_startup()</code> drives <code>SYSINITs</code>                | <code>sys/kern/init_main.c</code>                                                   |  |
| <code>SYSINIT</code> chain: subsystems init in order                              | <code>sys/sys/kernel.h</code>                                                       |  |
| <code>SI_SUB_VM</code> , <code>SI_SUB_LOCK</code> , <code>SI_SUB_KLD</code> , ... |                                                                                     |  |
| <code>SI_SUB_DRIVERS</code> , <code>SI_SUB_CONFIGURE</code> , ...                 |                                                                                     |  |
| Bus probe and attach: drivers bind to devices                                     | <code>sys/kern/subr_bus.c</code>                                                    |  |
| <code>SI_SUB_CREATE_INIT</code> / <code>SI_SUB_KTHREAD_INIT</code> :              | <code>sys/kern/init_main.c</code>                                                   |  |
| <code>init</code> proc forked, then made runnable                                 |                                                                                     |  |
| <code>start_init()</code> mounts root, execs <code>/sbin/init</code>              | <code>sys/kern/init_main.c</code> ,                                                 |  |
| (PID 1)                                                                           | <code>sys/kern/vfs_mountroot.c</code>                                               |  |
| Userland startup: <code>rc(8)</code> , daemons, logins                            | <code>rc(8)</code> , <code>init(8)</code>                                           |  |

The table is a map, not a contract. Many details differ per architecture, and the boundaries between stages shift as the kernel evolves. What does not change is the direction: small, simple environment at the top, increasingly functional environment at the bottom, userland at the end.

### 42.6.3 Boot Loaders and Kernel Modules at Boot Time

**What it is.** The FreeBSD loader (`/boot/loader`) is a small program that the first-stage boot code hands control to. Its job is to read `/boot/loader.conf`, load the kernel (`/boot/kernel/kernel`), load any modules requested by configuration, hand tunables and hints to the kernel, and transfer control to the kernel's entry point. It also provides the boot menu, the ability to boot a different kernel for recovery, and the facility to load a module before the kernel starts.

**Why this matters to driver writers.** Some drivers must be present before the root filesystem is mounted (storage controllers, for example). Those drivers are either compiled into the kernel or loaded by the loader from `/boot/loader.conf`. A driver that is loaded after root mount (through `kldload(8)`) cannot drive the disk that holds `/`. Knowing the difference is part of planning when a driver should attach.

**How this shows up in FreeBSD.** `/boot/loader.conf` is the text file that lists modules to load at boot. `loader(8)` and `loader.conf(5)` describe the mechanism. The loader also provides facilities such as `kenv(2)` tunables that the kernel can read in early `SYSINIT` stages.

**Common trap.** Relying on boot-time loader variables from kernel code that runs after root mount. Those variables are available; they are just not the right place for runtime configuration. Use `sysctl(9)` tunables for runtime, `kenv` only for early init decisions.

**Where the book teaches this.** Chapter 2 uses the loader while setting up the lab. Chapter 6 introduces `kldload(8)` for run-time driver loading.

**What to read next.** `loader(8)`, `loader.conf(5)`, `kenv(2)`.

### 42.6.4 Kernel Modules and Driver Attachment

**What it is.** A kernel module is a compiled `.ko` file that the kernel can load at runtime (or that the loader can load before kernel entry). Modules are how most drivers reach the kernel during normal operation. The module framework handles symbol resolution, versioning (`MODULE_VERSION`), dependencies (`MODULE_DEPEND`), and the registration of any `DRIVER_MODULE` declarations the file contains.

**Why this matters to driver writers.** Most of your development cycle will look like this: edit source, run `make`, `kldload ./mydriver.ko`, test, `kldunload mydriver`, repeat. The module framework makes that cycle fast and safe, and it also makes the production deployment story clean (the same `.ko` that you test is what ships). The loader and the run-time `kldload(8)` use the same module format.

**How this shows up in FreeBSD.** Every driver file ends with `DRIVER_MODULE`, `MODULE_VERSION`, and optionally `MODULE_DEPEND`. The declarations register the driver's bus-attachment metadata and module-level metadata with the kernel. When the module is loaded, the kernel runs the module's event handler for `MOD_LOAD`, which (among other things) registers the driver with Newbus. Newbus then offers the driver each device of the right bus, calling `probe` on each.

**Common trap.** Forgetting to call `bus_generic_detach`, `bus_release_resource`, or `mtx_destroy` from `detach`. The module unload succeeds, the driver is gone, but the resources leak. Subsequent reloads accumulate leaks until something obvious breaks.

**Where the book teaches this.** Chapter 6 introduces the module/driver relationship. Chapter 7 exercises the full edit-build-load-test-unload cycle.

**What to read next.** `kld(9)`, `kldload(8)`, `kldstat(8)`, `DRIVER_MODULE(9)`, `MODULE_VERSION(9)`.

### 42.6.5 `init_main` and the `SYSINIT` Chain

**What it is.** `mi_startup` is the machine-independent entry point that the architecture-specific boot code calls after early assembly (on amd64, `hammer_time` returns to `locore.S`, which calls `mi_startup`). Its body is almost entirely a loop that walks a sorted list of `SYSINIT` records and calls each one. Each record is registered at compile time by a `SYSINIT(...)` macro and carries a subsystem identifier (`SI_SUB_*`) and an order within the subsystem (`SI_ORDER_*`). The linker collects them all into a single section; `mi_startup` sorts and runs them. The init process itself is created in a stopped state at `SI_SUB_CREATE_INIT`, made runnable at `SI_SUB_KTHREAD_INIT`, and once it is scheduled it runs `start_init`, which mounts the root filesystem and execs `/sbin/init` as PID 1.

**Why this matters to driver writers.** Most drivers never touch `SYSINIT` directly; `DRIVER_MODULE` wraps the right registration for them. But a few pieces of infrastructure (kernel threads that must exist before drivers attach, subsystem-wide initialisers, early event handlers) use `SYSINIT` directly, and when you read such code it helps to know what the declaration is registering. The `SI_SUB_*` names make the ordering auditable, and the numeric spacing between them leaves room for future stages.

**How this shows up in FreeBSD.** The `SYSINIT` macro and the `SI_SUB_*` constants are in `/usr/src/sys/sys/kernel.h`. The driver-relevant stages include `SI_SUB_VM` (virtual memory up), `SI_SUB_LOCK` (lock initialisation), `SI_SUB_KLD` (module system), `SI_SUB_DRIVERS` (early driver-subsystem init), `SI_SUB_CONFIGURE` (Newbus probe and attach), `SI_SUB_ROOT_CONF` (candidate root devices identified), `SI_SUB_CREATE_INIT` (init process forked), and `SI_SUB_KTHREAD_INIT` (init made runnable). `mi_startup` is in `/usr/src/sys/kern/init_main.c`, and `start_init` is in the same file.

**Common trap.** Assuming your driver's `attach` runs very early. It runs at `SI_SUB_CONFIGURE`, which is after locks, memory, and most subsystems are up. If you need to coordinate with something earlier, either use the event-handler system (`eventhandler(9)`) to hook a well-defined event, or place the work at the right `SYSINIT` stage.

**Where the book teaches this.** Chapter 2 introduces the picture at boot-time. Chapter 6 connects `DRIVER_MODULE` to the `SYSINIT` chain conceptually. Appendix E goes into the subsystem-stage detail for readers who need it.

**What to read next.** The `SYSINIT` definitions in `/usr/src/sys/sys/kernel.h`, the loop in `/usr/src/sys/kern/init_main.c`, and `/usr/src/sys/kern/subr_bus.c` for the Newbus side.

### 42.6.6 Kernel Compilation in One Paragraph

A FreeBSD kernel is built from source by the ordinary `make` system invoked from `/usr/src`. The build takes a kernel configuration file (typically under `/usr/src/sys/amd64/conf/GENERIC` or a customised file you keep alongside it), compiles the selected options and device entries, and links the result into `kernel` plus a set of `.ko` files for devices not statically included. The standard commands are `make buildkernel KERNCONF=MYCONF` and `make installkernel KERNCONF=MYCONF`, run from `/usr/src`. For out-of-tree modules, the build uses `bsd.kmod.mk`, a short Makefile fragment that knows how to compile one or more `.c` files into a `.ko` file with the right kernel flags. Chapter 2 walks through both paths in lab context; the details are not repeated here.



## 42.7 Quick-Reference Tables

The compact tables below are meant for scanning. They do not replace the sections above; they help you point at the right section fast.

### 42.7.1 Kernel Space vs User Space at a Glance

| Property                   | Kernel space                                      | User space                                      |
|----------------------------|---------------------------------------------------|-------------------------------------------------|
| Privilege                  | Full CPU privilege                                | Reduced, CPU-enforced                           |
| Address space              | One, shared by all kernel threads                 | One per process, isolated                       |
| Memory stability           | Stable while allocated                            | May be paged, unmapped, moved                   |
| Direct hardware access     | Yes                                               | No                                              |
| Failure blast radius       | Whole machine                                     | One process                                     |
| How to cross over          | System call trap                                  | <code>syscall</code> instruction (architecture) |
| Data copy primitive (into) | <code>copyin(9)</code> , <code>uiomove(9)</code>  | (no equivalent; kernel does the copy)           |
| Data copy primitive (out)  | <code>copyout(9)</code> , <code>uiomove(9)</code> | (no equivalent; kernel does the copy)           |

### 42.7.2 Driver Type Comparison

| Driver type        | Registers with                            | Main entry points                                                | Main buffer type         | Typical devices                                       |
|--------------------|-------------------------------------------|------------------------------------------------------------------|--------------------------|-------------------------------------------------------|
| Character          | <code>cdevsw</code> in <code>devfs</code> | <code>d_open</code> , <code>d_read</code> , <code>d_ioctl</code> | <code>struct uio</code>  | Serial, sensors, <code>/dev/*</code>                  |
| Network            | <code>ifnet</code>                        | <code>if_transmit</code> , receive path                          | <code>struct mbuf</code> | NICs, <code>tun</code> , <code>tap</code>             |
| Storage (GEOM)     | GEOM class/provider                       | <code>g_start</code> , <code>biodone(9)</code>                   | <code>struct bio</code>  | Disks, <code>md</code> , <code>geli</code>            |
| Pseudo (character) | <code>cdevsw</code> in <code>devfs</code> | Same as character                                                | <code>struct uio</code>  | <code>/dev/null</code> ,<br><code>/dev/random</code>  |
| Pseudo (network)   | <code>ifnet</code>                        | Same as network                                                  | <code>struct mbuf</code> | <code>tun</code> , <code>tap</code> , <code>lo</code> |

### 42.7.3 Locking Primitive Selection

| You need...                                | Use                                            |
|--------------------------------------------|------------------------------------------------|
| A default lock around softc fields         | <code>mtx(9)</code> with <code>MTX_DEF</code>  |
| A lock safe to hold in an interrupt filter | <code>mtx(9)</code> with <code>MTX_SPIN</code> |

| You need...                                | Use                                        |
|--------------------------------------------|--------------------------------------------|
| Many readers, occasional writer, may sleep | <code>sx(9)</code>                         |
| Wait for a predicate under a mutex         | <code>cv(9)</code> together with the mutex |
| Atomic single-word read-modify-write       | <code>atomic(9)</code>                     |

#### 42.7.4 softc Lifecycle Checkpoint

| Phase   | Typical softc actions                                                       |
|---------|-----------------------------------------------------------------------------|
| probe   | Read IDs, return <code>BUS_PROBE_*</code> . No softc allocation yet.        |
| attach  | <code>device_get_softc</code> , init lock, allocate resources, setup state. |
| Runtime | All operations dereference <code>sc</code> , all state updates under lock.  |
| suspend | Save volatile hardware state, stop pending work.                            |
| resume  | Restore state, restart work.                                                |
| detach  | Stop work, free resources in reverse order, destroy lock.                   |

#### 42.7.5 Boot-to-Init Quick Map

| Phase                                                              | Driver-relevant fact                                               |
|--------------------------------------------------------------------|--------------------------------------------------------------------|
| Firmware + loader                                                  | Kernel and preloaded modules reach memory.                         |
| Kernel entry ( <code>hammer_time</code> on amd64)                  | Machine-dependent early setup; no drivers yet.                     |
| <code>mi_startup</code> / <code>SYSINIT</code> chain               | Subsystems initialise in <code>SI_SUB_*</code> order.              |
| <code>SI_SUB_DRIVERS</code> / <code>SI_SUB_CONFIGURE</code>        | Buses enumerate; compiled-in and preloaded drivers probe / attach. |
| <code>SI_SUB_ROOT_CONF</code>                                      | Candidate root devices identified.                                 |
| <code>SI_SUB_CREATE_INIT</code> / <code>SI_SUB_KTHREAD_INIT</code> | Init process forked, then made runnable.                           |
| <code>start_init</code> runs                                       | Root filesystem mounted; <code>/sbin/init</code> execed as PID 1.  |
| Runtime <code>kldload(8)</code>                                    | Additional drivers attach after this point.                        |

### 42.8 Wrapping Up: How These OS Concepts Support Driver Development

Each of the concepts in this appendix is already in use somewhere in the book. The appendix only assembles the pieces so that the reader can glance at the whole picture when one part feels out of focus.

The kernel/user boundary is the reason driver code looks different from ordinary C. Every unusual rule, from `copyin(9)` to the short spin-safe interrupt filter, exists because the kernel runs in a privileged world that cannot afford to trust the less privileged one. Keep that framing and the rest follows.

The driver types name the contracts with the rest of the kernel. Character, network, and GEOM are not ornaments; they are the standard interfaces that let userland and other subsystems reach your device without learning anything about its hardware. When you begin a new driver, naming the type is the first architectural decision you make.

The recurring data structures name the shapes inside the driver. A queue for pending work. A buffer for bytes moving through the subsystem. A mutex around shared state. A `softc` that owns everything about one device. The shapes are small; the discipline they encode is what keeps the driver correct.

The boot sequence names the stage on which your driver performs. The kernel did not spring into existence; it was assembled by a chain of well-defined steps, and your driver enters during one specific step at `SI_SUB_CONFIGURE`. Understanding the order of the steps turns early-boot bugs from mysteries into ordinary kernel diagnostics.

Three habits keep these concepts active instead of dormant.

The first is asking which side of the boundary you are on. Before you touch a pointer, ask whether it is a user pointer, a kernel pointer, or a bus address; the correct primitive follows immediately.

The second is asking which subsystem you are speaking to. Before you reach for a buffer type, ask whether the code is in the character, network, or GEOM path; the right buffer follows immediately.

The third is keeping a short companion sheet for the concepts you use most. The files under `examples/appendices/appendix-d-operating-system-concepts/` are meant for exactly that. A kernel-vs-user-space cheatsheet, a driver-type comparison, a `softc` lifecycle checklist, a boot-to-init timeline, and a “where drivers fit in the OS” diagram. The teaching is in this appendix; the application is in those sheets.

With that, the operating-system side of the book has a consolidated home. The chapters keep teaching; this appendix keeps naming; the examples keep reminding. When a reader closes this appendix and opens an unfamiliar driver, the mental model is ready, and every line of kernel code has a place to attach to.

## 43 Appendix E: Navigating FreeBSD Kernel Internals

### 43.1 How to Use This Appendix

The main chapters teach you to build a FreeBSD device driver from the first `printf("hello")` module to a working PCI driver with DMA and interrupts. Underneath that progression is a large kernel with many moving parts, and the book cannot teach each of those parts from scratch without losing the thread of what you are actually trying to do. Most of the time you do not need to know every corner of the kernel. You just need to know where you are, which subsystem your current line of code is touching, which structure gives you the answer when you stop to look, and where in `/usr/src` the evidence lives.

This appendix is that map. It does not try to teach each subsystem from first principles. It takes the seven subsystems a driver author meets most often and, for each one, gives you the short version: what it is for, which structures matter, which APIs your driver is likely to cross, where to open a file and look, and what to read next. You can think of it as the field guide you keep next to the chapters, not a replacement for them.

#### 43.1.1 What You Will Find Here

Each subsystem is covered with the same small pattern, so that you can skim one and know where to look in the next.

- **What the subsystem is for.** A one-paragraph statement of the subsystem's responsibility.
- **Why a driver author should care.** The concrete reason your code meets this subsystem.
- **Key structures, interfaces, or concepts.** The short list of names that actually matter.
- **Typical driver touchpoints.** The specific places where a driver calls in, registers with, or receives a callback from the subsystem.
- **Where to look in `/usr/src`.** The two or three files that are worth opening first.
- **Manual pages and files to read next.** The next step when you want more depth.
- **Common beginner confusion.** The misunderstanding that costs people time.
- **Where the book teaches this.** Back-references into the chapters that use the subsystem in context.

Not every entry needs every label, and no entry tries to be exhaustive. The aim is pattern recognition, not a full subsystem manual.

#### 43.1.2 What This Appendix Is Not

It is not an API reference. Appendix A is the API reference and goes deeper into flags, lifecycle phases, and caveats for each call. When the question is *what does this function do* or *which flag is right*, Appendix A is the place to look.

It is not a concept tutorial either. Appendix D covers the operating-system mental model (kernel vs userland, driver types, the boot-to-init path), Appendix C covers the hardware model (physical memory, MMIO, interrupts, DMA), and Appendix B covers the algorithmic patterns (`<sys/queue.h>`, ring buffers, state machines, unwind ladders). If the question you want answered is “what is a process”, “what is a BAR”, or “which list macro should I use”, one of those appendices is the right destination.

It is also not a complete subsystem reference. A full tour of the VFS, the VM, or the network stack would be its own book. What you get here is the ten per cent of each subsystem a driver author actually meets, in the order in which a driver author meets it.

## 43.2 Reader Guidance

Three ways to use this appendix, each calling for a different reading strategy.

If you are **reading the main chapters**, keep the appendix open in a second window. When Chapter 5 introduces the kernel’s memory allocators, glance at the Memory Subsystem section here to see where those allocators sit relative to UMA and the VM system. When Chapter 6 walks through `device_t`, `softc`, and the probe/attach lifecycle, the Driver Infrastructure section shows where those types fit into the Newbus layer. When Chapter 24 discusses `SYSINIT`, `eventhandler(9)`, and taskqueues, the Boot and Module System and Kernel Services sections give you the surrounding context in one page each.

If you are **reading unfamiliar kernel code**, treat the appendix as a translator. When you see `struct mbuf` in a function signature, jump to the Network Subsystem section. When you see `struct bio`, jump to File and VFS. When you see `kobj_class_t` or `device_method_t`, jump to Driver Infrastructure. The goal during exploration is not to master the subsystem, just to name it.

If you are **designing a new driver**, scan the subsystems your driver will touch before you start. A character driver for a small peripheral will lean on Driver Infrastructure and File and VFS. A network driver will add the Network Subsystem. A storage driver will add the GEOM and buffer cache layers. An early-boot driver on an embedded board will add the Boot and Module System section. Knowing which subsystems you will touch helps you guess the right headers and the right chapters before you write a line of code.

A few conventions apply throughout:

- Source paths are shown in the book-facing form, `/usr/src/sys/...`, matching the layout on a standard FreeBSD system. You can open any of them on your lab machine.
- Manual pages are cited in the usual FreeBSD style. Kernel-facing pages live in section 9: `kthread(9)`, `malloc(9)`, `uma(9)`, `bus_space(9)`, `eventhandler(9)`. Userland interfaces live in section 2 or 3 and are mentioned where they matter.
- When an entry points at a reading example, the file is one a beginner can read in one sitting. Larger files exist that also use each pattern; those are mentioned only when they are the canonical reference.

With that in mind, we start with a one-page orientation to the whole kernel before drilling into subsystems one at a time.

### 43.3 How This Appendix Differs From Appendix A

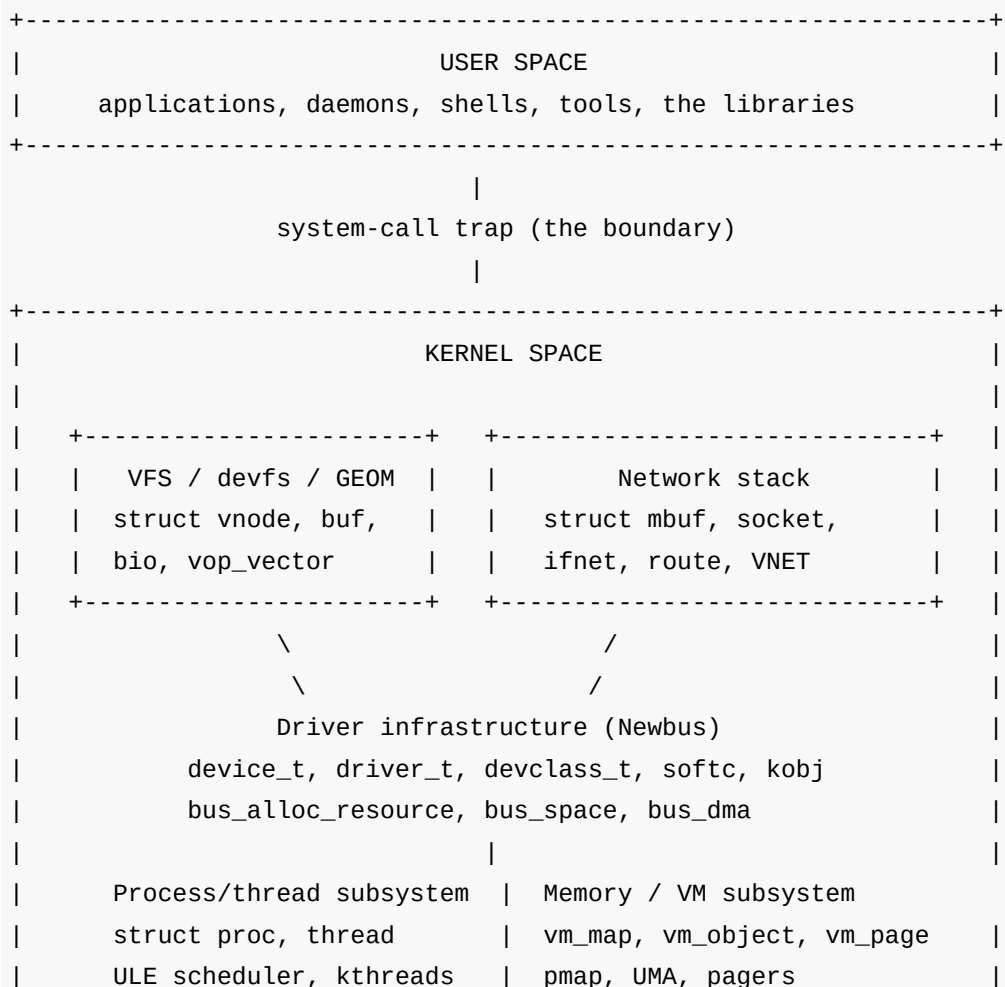
A driver author ends up consulting two very different kinds of reference while working. One kind answers the question *what is the exact name of the function or flag I need*. That is Appendix A. The other kind answers the question *which subsystem am I in, and where does this piece fit*. That is this appendix.

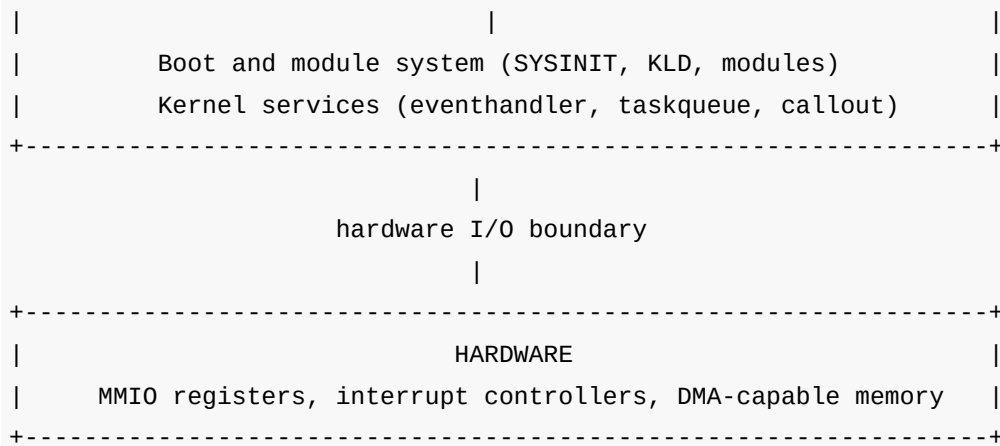
Concretely, the difference shows up like this. When you want to know the signature of `malloc(9)`, the meaning of `M_WAITOK` versus `M_NOWAIT`, and which manual page to open, that is Appendix A. When you want to know that `malloc(9)` is a thin convenience on top of `UMA`, which in turn builds on the `vm_page_t` layer, which in turn depends on the per-architecture `pmap(9)`, that is this appendix.

Both appendices cite real source paths and real manual pages. The split is deliberate. Keeping the API lookup separate from the subsystem map makes each one short enough to actually use. If an entry here starts to look like Appendix A, it has drifted from its role, and the right move is to read Appendix A instead.

### 43.4 A Map of the Major Subsystems

Before you walk into any one subsystem, the shape of the whole kernel is worth naming. The FreeBSD kernel is large, but the pieces a driver author meets fit into a small set of families. The diagram below is the simplest honest picture.





Each of the labelled boxes above has a section below. The driver-infrastructure box in the middle is where every driver starts. The two boxes at the top are the subsystem entry points that a driver publishes to the rest of the kernel (character or storage on the left, network on the right). The two boxes in the middle rows are the horizontal services every driver relies on. The bottom boxes are the plumbing that gets the kernel running in the first place.

Most drivers touch only three or four of these boxes in detail. The appendix is organised so that you can read only the ones your driver actually uses.

## 43.5 Process and Thread Subsystem

### 43.5.1 What the subsystem is for

The process and thread subsystem manages every unit of execution inside FreeBSD. It owns the data structures that describe a running program, the scheduler that decides which thread runs next on which CPU, the machinery that creates and destroys kernel threads, and the rules that govern how a thread may block, sleep, or be preempted. Every line of kernel code, including your driver, is being executed by some thread, and the discipline the subsystem enforces is a direct constraint on what your driver is allowed to do.

### 43.5.2 Why a driver author should care

Three practical reasons. First, the context your code runs in (interrupt filter, `ithread`, taskqueue worker, syscall thread from userland, dedicated kernel thread you spawned) decides whether you may sleep, may allocate memory with `M_WAITOK`, or may hold a sleep lock. Second, any driver that needs background work (a polling loop, a recovery watchdog, a deferred command handler) will create a kernel thread or a `kproc` to host it. Third, any driver that looks at the caller's process credentials (for security checks in `d_ioctl`, for example) reaches into the process structure.

### 43.5.3 Key structures, interfaces, or concepts

- **struct proc** is the per-process descriptor. It records the process id, credentials, file-descriptor table, signal state, address space, and the list of threads that belong to the process.

- **struct thread** is the per-thread descriptor. It records the thread id, priority, run state, saved register context, the pointer to its owning struct proc, and the locks it currently holds. A FreeBSD kernel thread is also described by a struct thread; it simply has no userland side.
- **The ULE scheduler** is FreeBSD's default multi-processor scheduler. It assigns threads to CPUs, implements priority classes (real-time, time-sharing, idle), and honours interactivity and affinity hints. From a driver author's perspective, the most important fact about ULE is that it runs whichever thread is next, whenever a lock is released, a sleep ends, or an interrupt finishes; you cannot assume control of the CPU across such events.
- **kthread\_add(9)** creates a new kernel thread inside an existing kernel process. Use it when you want a lightweight worker that shares state with an existing kproc (for example, extra worker threads inside a driver-specific kproc).
- **kproc\_create(9)** creates a new kernel process, which comes with its own struct proc and one initial thread. Use it when you want an independent, top-level worker that ps -axH will show as a distinct name (for example, g\_event, usb, bufdaemon).

#### 43.5.4 Typical driver touchpoints

- Interrupt handlers and bus\_setup\_intr(9) callbacks run in kernel-thread context created by the interrupt framework.
- A driver that needs long-running background work calls kproc\_create(9) or kthread\_add(9) from its attach path and joins the thread from detach.
- A driver that takes action on behalf of a user process reads curthread or td->td\_proc to look at credentials, process id, or root-directory for validation.
- A driver that sleeps on a condition uses the sleep primitives, which record the sleeping thread and yield to the scheduler until woken.

#### 43.5.5 Where to look in /usr/src

- /usr/src/sys/sys/proc.h defines struct proc and struct thread and the macros that navigate between them.
- /usr/src/sys/sys/kthread.h declares the kernel-thread creation API.
- /usr/src/sys/kern/kern\_kthread.c contains the implementation.
- /usr/src/sys/kern/sched\_ule.c is the ULE scheduler source.

#### 43.5.6 Manual pages and files to read next

kthread(9), kproc(9), curthread(9), proc(9), and the header /usr/src/sys/sys/proc.h. If you want to see a driver that owns a kernel thread, /usr/src/sys/dev/random/random\_harvestq.c is a readable example.

#### 43.5.7 Common beginner confusion

The most frequent trap is assuming that the thread running your driver code is a driver-owned thread. It is not. Most of the time it is a user thread that entered the kernel through a syscall, or an ithread that the interrupt



framework created for you. Your driver only owns the threads it explicitly creates. Another recurring trap is reaching into `curthread->td_proc` from a context where `curthread` has nothing to do with the device (an interrupt `ithread`, for example); the process you find is not the process that asked for the operation.

### 43.5.8 Where the book teaches this

Chapter 5 introduces kernel execution context and the sleep-versus-atomic distinction. Chapter 11 returns to it when concurrency becomes real. Chapter 14 uses taskqueues as a way to offload work into a safely sleepable context. Chapter 24 shows a full `kproc` lifecycle inside a driver.

### 43.5.9 Further reading

- **In this book:** Chapter 11 (Concurrency in Drivers), Chapter 14 (Taskqueues and Deferred Work), Chapter 24 (Integrating with the Kernel).
- **Man pages:** `kthread(9)`, `kproc(9)`, `scheduler(9)`.
- **External:** McKusick, Neville-Neil, and Watson, *The Design and Implementation of the FreeBSD Operating System* (2nd ed.), chapters on process and thread management.

## 43.6 Memory Subsystem

### 43.6.1 What the subsystem is for

The virtual-memory (VM) subsystem manages every byte of memory the kernel can address. It owns the mapping from virtual addresses to physical pages, the allocation of pages to processes and to the kernel, the paging policy that reclaims pages under pressure, and the page-backing stores that hand pages in from disk, from devices, or from zero. A driver that allocates memory, exposes memory to userland through `mmap`, or does DMA is interacting with the VM subsystem whether it names it or not.

### 43.6.2 Why a driver author should care

Four practical reasons. First, every kernel allocation goes through this subsystem, directly or indirectly. Second, any driver that exports a memory-mapped view of a device (or of a software buffer) does so through a VM pager. Third, DMA involves physical addresses, and only the VM subsystem knows how virtual kernel addresses translate to them. Fourth, the subsystem defines the sleep rules for allocation: `M_WAITOK` may walk the VM's page-reclamation path, which is something you cannot do from an interrupt filter.

### 43.6.3 Key structures, interfaces, or concepts

- **`vm_map_t`** represents a contiguous collection of virtual address mappings belonging to one address space. The kernel has its own `vm_map_t`, and each user process has one. Drivers almost never walk a `vm_map_t` directly; higher-level APIs do the walking for them.
- **`vm_object_t`** represents a backing store: a set of pages that can be mapped into a `vm_map_t`. Objects are typed by the pager that produces their pages (anonymous, vnode-backed, swap-backed, device-backed).

- **vm\_page\_t** represents one physical page of RAM, together with its current state (wired, active, inactive, free) and the object it currently belongs to. All physical memory in the system is tracked by an array of **vm\_page\_t** records.
- **The pager layer** is the set of pluggable strategies that fill pages with data. The three most important ones for driver authors are the swap pager (anonymous memory), the vnode pager (file-backed memory), and the device pager (memory whose content is produced by a driver). When a driver implements **d\_mmap** or **d\_mmap\_single**, it is publishing a slice of the device pager.
- **pmap(9)** is the machine-dependent page-table manager. It knows how to translate virtual addresses into physical ones for the current CPU architecture. Drivers rarely call **pmap** directly. The portable way to see a physical view is through **bus\_dma(9)** (for DMA) or **bus\_space(9)** (for MMIO registers).
- **UMA** is FreeBSD's slab allocator for fixed-size objects, with per-CPU caches to avoid locking in the fast path. **malloc(9)** itself is implemented on top of UMA for common sizes. Drivers that allocate and free millions of identical small objects per second (network descriptors, per-request contexts) create their own UMA zone with **uma\_zcreate** and reuse objects instead of walking the general allocator.

#### 43.6.4 Typical driver touchpoints

- **malloc(9)**, **free(9)**, **contigmalloc(9)** for general control-plane memory.
- **uma\_zcreate(9)**, **uma\_zalloc(9)**, **uma\_zfree(9)** for high-rate fixed-size objects.
- **bus\_dmamem\_alloc(9)** and the rest of the **bus\_dma(9)** interface for DMA-capable memory; this is the driver-facing wrapper around the physical side of the VM.
- **d\_mmap(9)** or **d\_mmap\_single(9)** in a **cdevsw** method table to publish a device pager view of hardware memory to userland.
- **vm\_page\_wire(9)** and **vm\_page\_unwire(9)** only in the rare cases where the driver needs to pin pages of a user buffer across a long-running I/O.

#### 43.6.5 Where to look in /usr/src

- **/usr/src/sys/vm/vm.h** declares the **vm\_map\_t**, **vm\_object\_t**, and **vm\_page\_t** typedefs.
- **/usr/src/sys/vm/vm\_map.h**, **/usr/src/sys/vm/vm\_object.h**, and **/usr/src/sys/vm/vm\_page.h** hold the full type definitions.
- **/usr/src/sys/vm/swap\_pager.c**, **/usr/src/sys/vm/vnode\_pager.c**, and **/usr/src/sys/vm/device\_pager.c** are the three pagers most relevant to drivers.
- **/usr/src/sys/vm/uma.h** is the UMA public interface; **/usr/src/sys/vm/uma\_core.c** is the implementation.
- **/usr/src/sys/vm/pmap.h** is the machine-independent **pmap** interface; the machine-specific side lives under **/usr/src/sys/amd64/amd64/pmap.c**, **/usr/src/sys/arm64/arm64/pmap.c**, and similar files for each architecture.

#### 43.6.6 Manual pages and files to read next

**malloc(9)**, **uma(9)**, **contigmalloc(9)**, **bus\_dma(9)**, **pmap(9)**, and the header **/usr/src/sys/vm/uma.h**. For a readable driver that publishes a device pager, inspect **/usr/src/sys/dev/drm2/** or the framebuffer code under **/usr/src/sys/dev/fb/**.

### 43.6.7 Common beginner confusion

Two traps. First, conflating the three pointer flavours a DMA-capable driver sees: the kernel virtual address (what your pointer dereferences), the physical address (what the memory controller sees), and the bus address (what the device sees, which may go through an IOMMU). `bus_dma(9)` exists precisely to keep these separate. Second, assuming that a `bus_dmamem_alloc(9)` allocation is a generic memory allocation; it is a specialised allocation with stricter alignment, boundary, and segment rules dictated by the tag you passed in.

### 43.6.8 Where the book teaches this

Chapter 5 introduces kernel memory and the allocator flags. Chapter 10 revisits buffers in read/write paths. Chapter 17 introduces `bus_space` for MMIO access. Chapter 21 is the full DMA chapter and where the bus-versus-physical distinction becomes concrete.

### 43.6.9 Further reading

- **In this book:** Chapter 5 (Understanding C for FreeBSD Kernel Programming), Chapter 21 (DMA and High-Speed Data Transfer).
- **Man pages:** `malloc(9)`, `uma(9)`, `contigmalloc(9)`, `bus_dma(9)`, `pmap(9)`.
- **External:** McKusick, Neville-Neil, and Watson, *The Design and Implementation of the FreeBSD Operating System* (2nd ed.), chapter on memory management.

## 43.7 File and VFS Subsystem

### 43.7.1 What the subsystem is for

The file and VFS (Virtual File System) subsystem owns everything that userland sees through `open(2)`, `read(2)`, `write(2)`, `ioctl(2)`, `mmap(2)`, and the filesystem hierarchy in general. It dispatches operations to the correct filesystem through the vnode-operation vector, manages the buffer cache that sits between filesystems and storage drivers, and hosts the GEOM framework that lets storage drivers compose into stacks. For a driver author, this subsystem is either the main entry point (if you write a character or storage driver) or a quiet middle layer you are glad to let someone else worry about (if you write a network or embedded driver).

### 43.7.2 Why a driver author should care

Three practical reasons. First, every character driver publishes itself to VFS through a `cdevsw` and a `/dev` node created by `devfs`. Second, every storage driver plugs into the bottom of a stack that the VFS and GEOM layers assemble on top, and the unit of work you receive is a `struct bio`, not a user pointer. Third, even a non-storage driver may need to understand vnodes and `mmap` if it publishes its memory to userland.

### 43.7.3 Key structures, interfaces, or concepts

- **struct vnode** is the kernel's abstract file or device. It carries a type (regular file, directory, character device, block device, named pipe, socket, symbolic link), a pointer to its filesystem's `vop_vector`, the

mount it belongs to, a reference count, and a lock. Every file descriptor in userland ultimately resolves to a vnode.

- **struct vop\_vector** is the vnode-operation dispatch table: a pointer per operation (VOP\_LOOKUP, VOP\_READ, VOP\_WRITE, VOP\_IOCTL, and dozens more) that the filesystem or devfs implements. The vector is declared conceptually in `/usr/src/sys/sys/vnode.h` and generated from the operation list in `/usr/src/sys/kern/vnode_if.src`.
- **The GEOM framework** is FreeBSD's stackable storage layer. A GEOM *provider* is a storage surface; a *consumer* is something attaching to a provider. Drivers for storage hardware register as providers; classes such as `g_part`, `g_mirror`, or filesystems attach as consumers. The topology graph is dynamic and visible at runtime through `gpart show`, `geom disk list`, and `sysctl kern.geom`.
- **devfs** is the pseudo-filesystem that populates `/dev`. When your character driver calls `make_dev_s(9)`, devfs allocates a vnode-backed entry that forwards VFS operations into your `cdevsw` callbacks. devfs is the single layer between `open(2)` on a `/dev` path and `d_open` in your driver.
- **struct buf** is the traditional buffer-cache descriptor used by the old block-device path and by filesystems that layer on top of the buffer cache. It is still important because many filesystems drive I/O through `buf` objects before `buf_strategy()` funnels them into GEOM.
- **struct bio** is the modern per-operation descriptor that flows through GEOM. Every block read or write in GEOM is a `bio` with a command (`BIO_READ`, `BIO_WRITE`, `BIO_FLUSH`, `BIO_DELETE`), a range, a buffer pointer, and a completion callback. Your storage driver receives `bios` on its start routine and calls `biodone()` (or the GEOM equivalent) when it finishes them.

#### 43.7.4 Typical driver touchpoints

- A character driver fills a `struct cdevsw` with callbacks (`d_open`, `d_close`, `d_read`, `d_write`, `d_ioctl`, optionally `d_poll`, `d_mmap`) and calls `make_dev_s(9)` to attach it to `/dev`.
- A storage driver registers a GEOM class, implements a start routine that accepts `bios`, and calls `g_io_deliver()` when it finishes them.
- A driver that wants to be visible as a file (for reading telemetry, for example) may expose a character device whose `d_read` copies driver data out.
- A driver that publishes device memory to userland implements `d_mmap` or `d_mmap_single` to hand back a device pager object.

#### 43.7.5 Where to look in `/usr/src`

- `/usr/src/sys/sys/vnode.h` declares `struct vnode` and the vnode-operation plumbing.
- `/usr/src/sys/kern/vnode_if.src` is the source of truth for every VOP in the kernel; read it to see the operation list and the locking protocol.
- `/usr/src/sys/fs/devfs/` holds the devfs implementation; `devfs_devs.c` and `devfs_vnops.c` are the readable entry points.
- `/usr/src/sys/geom/geom.h` declares providers, consumers, and the GEOM class interface.
- `/usr/src/sys/sys/buf.h` and `/usr/src/sys/sys/bio.h` declare the block-I/O structures.
- `/usr/src/sys/dev/null/null.c` is the simplest character driver in the tree and the right first read.

### 43.7.6 Manual pages and files to read next

`vnode(9)`, `VOP_LOOKUP(9)` and the rest of the VOP family, `devfs(4)`, `devfs(5)`, `cdev(9)`, `make_dev(9)`, `g_attach(9)`, `geom(4)`, and the header `/usr/src/sys/sys/bio.h`.

### 43.7.7 Common beginner confusion

Three traps. First, expecting a character driver to deal with `struct buf` or `struct bio`. It does not; those live in the storage path. A character driver sees `struct uio` on its `d_read` and `d_write` callbacks and nothing more. Second, expecting a storage driver to create a `/dev` node itself. In modern FreeBSD the GEOM layer creates the `/dev` entries for block devices; your storage driver registers with GEOM, and `devfs` does the rest on the other side of GEOM. Third, assuming the `vnode` and the `cdev` are the same object. They are not. The `vnode` is the VFS-side handle to the open file; the `cdev` is the driver-side identity. `open(2)` on `/dev/foo` produces a `vnode` whose operations forward into your `cdevsw`.

### 43.7.8 Where the book teaches this

Chapter 7 writes the first character driver and the first `cdevsw`. Chapter 8 walks through `make_dev_s(9)` and `devfs` node creation. Chapter 9 connects `d_read` and `d_write` to `uio`. Chapter 27 is the storage chapter and introduces `struct bio`, GEOM providers and consumers, and the buffer cache.

### 43.7.9 Further reading

- **In this book:** Chapter 7 (Writing Your First Driver), Chapter 8 (Working with Device Files), Chapter 27 (Working with Storage Devices and the VFS Layer).
- **Man pages:** `vnode(9)`, `make_dev(9)`, `devfs(5)`, `geom(4)`, `g_bio(9)`.
- **External:** McKusick, Neville-Neil, and Watson, *The Design and Implementation of the FreeBSD Operating System* (2nd ed.), chapters on the I/O system and local filesystems.

## 43.8 Network Subsystem

### 43.8.1 What the subsystem is for

The network subsystem moves packets. It owns the data structures that represent a packet in flight (`mbuf` and friends), the per-interface state that represents a network device to the rest of the stack (`ifnet`), the routing tables that decide where a packet should go, the socket layer that userland sees, and the VNET infrastructure that lets multiple independent network stacks coexist in a single kernel. A network driver is the bottom layer of this stack: it hands packets up into the stack on receive, and the stack hands packets down into the driver on transmit.

### 43.8.2 Why a driver author should care

Two reasons. If you write a network driver, nearly every byte you touch is a field of one of the structures named below, and the shape of your code is set by the protocols they enforce. If you write any other kind of driver, you

still benefit from recognising `struct mbuf` and `struct ifnet` when you see them in code, because they turn up in many adjacent subsystems (packet filters, load-balancing helpers, virtual interfaces).

### 43.8.3 Key structures, interfaces, or concepts

- **struct mbuf** is the packet fragment. Packets are represented as chains of mbufs, linked by `m_next` for a single packet and by `m_nextpkt` for consecutive packets in a queue. An mbuf carries a small header and either a small inline data area or a pointer into an external storage cluster. The design is optimised for cheap prepending of headers.
- **struct m\_tag** is an extensible metadata tag attached to an mbuf. It lets the stack and drivers attach typed information to a packet (for example, hardware transmit-checksum offload, receive-side-scaling hash, filter decision) without enlarging the mbuf itself.
- **ifnet** (spelled `if_t` in modern APIs) is the per-interface descriptor. It carries the interface name and index, the flags, the MTU, a transmit function (`if_transmit`), the counters the stack increments, and the hooks that let higher layers deliver packets to the driver.
- **VNET** is the per-virtual-network-stack container. When `VIMAGE` is compiled in, every jail that enables VNET has its own routing table, its own set of interfaces, and its own protocol control blocks. Network drivers must be VNET-aware: they use `VNET_DEFINE` and `VNET_FOREACH` so that per-VNET state lives in the right place.
- **Routing** is the subsystem that selects a next-hop for an outbound packet. It owns the forwarding information base (FIB), a per-VNET radix tree of routes. Drivers rarely interact with routing directly; the stack has already chosen the interface before it reaches the driver.
- **The socket layer** is the kernel side of the socket(2) family of syscalls. For driver authors, the relevant fact is that a socket ultimately produces calls into `ifnet`, which produce calls into your driver. You do not implement sockets yourself.

### 43.8.4 Typical driver touchpoints

- The driver allocates and populates an `ifnet` in `attach`, registers a transmit function, and calls `ether_ifattach(9)` or `if_attach(9)` to announce itself to the stack.
- The driver's transmit function receives an mbuf chain, writes descriptors, kicks the hardware, and returns.
- The receive path handles an interrupt or a poll, wraps received bytes in mbufs, and calls `if_input(9)` on the interface to push them up into the stack.
- On detach, the driver calls `ether_ifdetach(9)` or `if_detach(9)` before freeing its resources.
- The driver registers for `ifnet_arrival_event` or `ifnet_departure_event` if it needs to react when peers appear or disappear (see `/usr/src/sys/net/if_var.h` for the declarations).

### 43.8.5 Where to look in `/usr/src`

- `/usr/src/sys/sys/mbuf.h` declares `struct mbuf` and `struct m_tag`.
- `/usr/src/sys/net/if.h` declares `if_t` and the public interface API.
- `/usr/src/sys/net/if_var.h` declares the interface-event eventhandlers and internal state.
- `/usr/src/sys/net/if_private.h` contains the full `struct ifnet` definition used inside the stack.
- `/usr/src/sys/net/vnet.h` declares the VNET infrastructure.

- `/usr/src/sys/net/route.h` and `/usr/src/sys/net/route/` hold the routing tables.
- `/usr/src/sys/sys/socketvar.h` declares `struct socket`.

### 43.8.6 Manual pages and files to read next

`mbuf(9)`, `ifnet(9)`, `ether_ifattach(9)`, `vnet(9)`, `route(4)`, and `socket(9)`. For a small, readable real network driver, `/usr/src/sys/net/if_tuntap.c` is the canonical reading example.

### 43.8.7 Common beginner confusion

Two traps. First, expecting a network driver to publish itself through `/dev`. It does not; it publishes itself through `ifnet` and becomes visible as `bge0`, `em0`, `igb0`, and so on, not through `devfs`. Second, holding onto an `mbuf` after handing it to the stack. Once you call `if_input` or return from `if_transmit`, the `mbuf` is no longer yours; using it afterwards corrupts the stack silently.

### 43.8.8 Where the book teaches this

Chapter 28 is the full network-driver chapter and the right place for the details. Chapters 11 and 14 provide the locking and deferred-work discipline the receive path needs. Chapter 24 covers `ifnet_arrival_event` and related event hooks at a driver-integration level.

### 43.8.9 Further reading

- **In this book:** Chapter 28 (Writing a Network Driver), Chapter 11 (Concurrency in Drivers), Chapter 14 (Taskqueues and Deferred Work).
- **Man pages:** `mbuf(9)`, `ifnet(9)`, `vnet(9)`, `socket(9)`.
- **External:** McKusick, Neville-Neil, and Watson, *The Design and Implementation of the FreeBSD Operating System* (2nd ed.), chapters on the network subsystem.

## 43.9 Driver Infrastructure (Newbus)

### 43.9.1 What the subsystem is for

Newbus is FreeBSD's driver framework. It owns the tree of devices in the system, matches drivers to devices through probe, manages the lifecycle of each attachment, routes resource allocations to the right bus, and provides the object-oriented dispatch that lets buses override and extend each other's behaviour. Every character driver, storage driver, network driver, and embedded driver in the tree is a Newbus participant. If the other subsystems in this appendix are the rooms, Newbus is the hallway that connects them.

### 43.9.2 Why a driver author should care

There is essentially no FreeBSD driver without Newbus. The types you meet first (`device_t`, `softc`, `driver_t`, `devclass_t`), the APIs you reach for in `attach` (`bus_alloc_resource_any`, `bus_setup_intr`), the macros

you wrap the whole driver in (`DRIVER_MODULE`, `DEVMETHOD`, `DEVMETHOD_END`) all belong here. Learning to navigate Newbus is the same thing as learning to navigate FreeBSD driver source.

### 43.9.3 Key structures, interfaces, or concepts

- **device\_t** is an opaque handle to a node in the Newbus device tree. You receive one in probe and attach, pass it into almost every bus API, and use it to fetch the softc with `device_get_softc(9)`.
- **driver\_t** is the descriptor for a driver: its name, its method table, and the size of its softc. You construct one for your driver and hand it to `DRIVER_MODULE(9)`, which registers it under a parent bus name.
- **devclass\_t** is the per-driver-class registry: the collection of `device_t` instances a driver has attached. It is how the kernel gives each instance a unit number.
- **kobj(9)** is the object-oriented machinery underneath Newbus. Method tables, method dispatch, and the ability for a bus to inherit another bus's methods are all kobj features. As a driver author you use the `DEVMETHOD` macros, which expand into kobj metadata; you rarely call kobj primitives directly.
- **softc** is the per-instance state of your driver, allocated by the kernel when it allocates the `device_t`. The kernel knows how large your softc is because you told it in the `driver_t`. `device_get_softc(9)` gives you back a pointer to it.
- **bus\_alloc\_resource(9)** and friends allocate memory windows, I/O ports, and interrupt lines from the parent bus on behalf of your driver. They are the portable way to get at a device's resources without caring which bus it sits on.

### 43.9.4 Typical driver touchpoints

- Declare a `device_method_t` array with `DEVMETHOD(device_probe, ...)`, `DEVMETHOD(device_attach, ...)`, `DEVMETHOD(device_detach, ...)`, terminated with `DEVMETHOD_END`.
- Declare a `driver_t` with your driver name, the methods, and `sizeof(struct mydev_softc)`.
- Use `DRIVER_MODULE(mydev, pci, mydev_driver, ...)` (or `usb`, `iicbus`, `spibus`, `simplebus`, `acpi`, `nexus`) to register the driver under its parent bus.
- In probe, decide whether this device is yours and, if so, return `BUS_PROBE_DEFAULT` (or a weaker/stronger value) and a description.
- In attach, allocate resources with `bus_alloc_resource_any(9)`, map registers with `bus_space(9)`, set up interrupts with `bus_setup_intr(9)`, and only then expose yourself to the rest of the kernel.
- In detach, undo everything in reverse order.

### 43.9.5 Where to look in /usr/src

- `/usr/src/sys/sys/bus.h` declares `device_t`, `driver_t`, `devclass_t`, `DEVMETHOD`, `DEVMETHOD_END`, `DRIVER_MODULE`, `bus_alloc_resource_any`, `bus_setup_intr`, and most of the rest.
- `/usr/src/sys/sys/kobj.h` declares the method-dispatch machinery.
- `/usr/src/sys/kern/subr_bus.c` holds the Newbus implementation.
- `/usr/src/sys/kern/subr_kobj.c` holds the kobj implementation.
- `/usr/src/sys/dev/null/null.c` and `/usr/src/sys/dev/led/led.c` are very small real drivers you can read in a sitting.



### 43.9.6 Manual pages and files to read next

`device(9)`, `driver(9)`, `DEVMETHOD(9)`, `DRIVER_MODULE(9)`, `bus_alloc_resource(9)`, `bus_setup_intr(9)`, `kobj(9)`, and `devinfo(8)` to see the running Newbus tree.

### 43.9.7 Common beginner confusion

Two traps. First, thinking that `device_t` and `softc` are the same object. `device_t` is the Newbus handle; `softc` is your driver's private state. You fetch `softc` from `device_t` with `device_get_softc(9)`. Second, forgetting that `DRIVER_MODULE`'s second argument is the parent bus name. A driver declared with `DRIVER_MODULE( . . . , pci, . . . )` can only attach under a PCI bus, no matter how many PCI-like boards exist elsewhere. If a driver must attach under multiple buses (for example, a chip that appears both as PCI and as ACPI), you register it twice.

### 43.9.8 Where the book teaches this

Chapter 6 is the full anatomy of a driver chapter and the canonical teaching location for everything above. Chapter 7 writes the first working driver against these APIs. Chapter 18 extends the picture for PCI. Chapter 24 returns to `DRIVER_MODULE`, `MODULE_VERSION`, and `MODULE_DEPEND` when kernel integration becomes the subject.

### 43.9.9 Further reading

- **In this book:** Chapter 6 (The Anatomy of a FreeBSD Driver), Chapter 7 (Writing Your First Driver), Chapter 18 (Writing a PCI Driver).
- **Man pages:** `device(9)`, `driver(9)`, `DRIVER_MODULE(9)`, `bus_alloc_resource(9)`, `bus_setup_intr(9)`, `kobj(9)`, `rman(9)`.

## 43.10 Boot and Module System

### 43.10.1 What the subsystem is for

The boot and module system is how the kernel gets into memory, how it initialises the hundreds of subsystems it depends on before anything can run, and how code that is not compiled into the kernel (loadable modules) is brought in, wired up, and eventually removed. From a driver author's perspective, the subsystem defines when your initialisation code runs relative to the rest of the kernel, and how your module-level `MOD_LOAD` and `MOD_UNLOAD` events interact with the kernel's internal initialisation order.

### 43.10.2 Why a driver author should care

Three reasons. First, if your driver can be loaded as a module, it may run on a kernel whose subsystems have been ordered differently than you expect, and you need to declare what you depend on. Second, if your driver must run early (for example, a console driver or a boot-time storage driver), you must understand `SYSINIT(9)`

subsystem IDs so that your code runs in the right slot. Third, even a plain driver relies on the module system to register itself, to declare ABI compatibility, and to fail cleanly if a dependency is missing.

### 43.10.3 Key structures, interfaces, or concepts

- **The boot sequence** follows a fixed arc: the loader reads the kernel from disk, hands control to the kernel entry point, which sets up the early CPU state and then calls `mi_startup()`. `mi_startup()` walks a sorted list of `SYSINIT` entries, invoking each one in order. When the list is exhausted, the kernel has enough services up to start `init(8)` as user process 1.
- **`SYSINIT(9)`** is the macro that registers a function to be called at a specific phase of kernel initialisation. Each entry has a subsystem ID (`SI_SUB_*`, coarse ordering) and an order-within-subsystem (`SI_ORDER_*`, fine ordering). The full list of legal subsystem IDs is in `/usr/src/sys/sys/kernel.h` and is worth skimming once. `SYSUNINIT(9)` is the matching teardown.
- **Module loading** is driven by the KLD framework. `kldload(8)` invokes the linker, which relocates the module, resolves its symbols against the running kernel, and invokes the module's event handler with `MOD_LOAD`. A matching `MOD_UNLOAD` runs when the module is removed. Drivers rarely write module event handlers by hand; `DRIVER_MODULE(9)` produces one for you.
- **`MODULE_DEPEND(9)`** declares that your module requires another module (`usb`, `miibus`, `pci`, `iflib`) to be present, and at which version range. The kernel refuses to load your module if the dependency is missing.
- **`MODULE_VERSION(9)`** declares the ABI version your module exports, so that other modules can depend on it with `MODULE_DEPEND`.

### 43.10.4 Typical driver touchpoints

- `DRIVER_MODULE(mydev, pci, mydev_driver, ...)` emits a module event handler that registers the driver on `MOD_LOAD` and deregisters it on `MOD_UNLOAD`.
- `MODULE_VERSION(mydev, 1)`; announces your module's ABI version.
- `MODULE_DEPEND(mydev, pci, 1, 1, 1)`; declares a `pci` dependency.
- A driver that must run before Newbus is available uses `SYSINIT(9)` to register a one-off setup hook at an early subsystem ID.
- A driver that wires a teardown hook at the last possible moment uses `SYSUNINIT(9)` with the matching ordering.

### 43.10.5 Where to look in `/usr/src`

- `/usr/src/sys/sys/kernel.h` defines `SYSINIT`, `SYSUNINIT`, `SI_SUB_*`, and `SI_ORDER_*`.
- `/usr/src/sys/kern/init_main.c` contains `mi_startup()` and the walk through the `SYSINIT` list.
- `/usr/src/sys/sys/module.h` declares `MODULE_VERSION` and `MODULE_DEPEND`.
- `/usr/src/sys/sys/linker.h` and `/usr/src/sys/kern/kern_linker.c` implement the KLD linker.
- `/usr/src/stand/` holds the loader and the boot-time code. (On older FreeBSD releases this lived under `/usr/src/sys/boot/`; FreeBSD 14 hosts it entirely under `/usr/src/stand/`.)

### 43.10.6 Manual pages and files to read next

`SYSINIT(9)`, `kld(9)`, `kldload(9)`, `kldload(8)`, `kldstat(8)`, `module(9)`, `MODULE_VERSION(9)`, and `MODULE_DEPEND(9)`. For a short real `SYSINIT` example, look near the top of `/usr/src/sys/dev/random/random_harvestq.c`.

### 43.10.7 Common beginner confusion

Two traps. First, assuming that `MOD_LOAD` is the moment your `attach` function runs. It is not. `MOD_LOAD` is the moment your *driver* is registered with Newbus; `attach` runs later, per device, whenever a bus offers a matching child. Second, using `SYSINIT` levels as if they were arbitrary. Each `SI_SUB_*` corresponds to a well-defined phase of kernel startup, and registering your hook at the wrong phase either makes it run too early (with half the kernel missing) or too late (after the event you cared about has passed).

### 43.10.8 Where the book teaches this

Chapter 6 introduces `DRIVER_MODULE`, `MODULE_VERSION`, and `MODULE_DEPEND` as part of driver anatomy. Chapter 24 covers kernel-integration topics including `SYSINIT`, subsystem IDs, and module teardown ordering. Chapter 32 returns to boot-time concerns on embedded platforms.

### 43.10.9 Further reading

- **In this book:** Chapter 24 (Integrating with the Kernel), Chapter 32 (Device Tree and Embedded Development).
- **Man pages:** `SYSINIT(9)`, `module(9)`, `MODULE_VERSION(9)`, `MODULE_DEPEND(9)`, `kldload(8)`, `kldstat(8)`.

## 43.11 Kernel Services

### 43.11.1 What the subsystem is for

The kernel ships a small set of general-purpose services that are not tied to any one subsystem but show up repeatedly in drivers: event notifications, deferred-work queues, timed callbacks, and subscription hooks. None of them teaches you how to write a driver, but all of them turn up in real driver code, and recognising them speeds up every code-reading session. This section collects the handful you are likely to meet.

### 43.11.2 Why a driver author should care

Drivers often need to react to system-wide events (shutdown, low memory, interface arrival, root filesystem mount), or to do work outside the context that delivered the event (away from an interrupt filter, away from a spin-locked critical section). The kernel services below are the standard FreeBSD answers to both needs. Using them means your driver integrates cleanly with the rest of the system; reimplementing them means you will eventually collide with a subsystem that expects your hooks to exist.

### 43.11.3 Key structures, interfaces, or concepts

- **eventhandler(9)** is the publish/subscribe system for kernel events. A publisher declares an event with `EVENTHANDLER_DECLARE`, a subscriber registers with `EVENTHANDLER_REGISTER`, and invocation with `EVENTHANDLER_INVOKE` fans out to every subscriber. Standard event tags defined in `/usr/src/sys/sys/eventhandler.h` include `shutdown_pre_sync`, `shutdown_post_sync`, `shutdown_final`, `vm_lowmem`, and `mountrout`; interface events (`ifnet_arrival_event`, `ifnet_departure_event`) are declared in `/usr/src/sys/net/if_var.h`. Drivers use these to clean up, to release memory, to react when a sibling interface appears, or to delay early work until the root filesystem is available.
- **taskqueue(9)** is a queue of deferred work items. A driver enqueues a task from a context that cannot sleep (for example, an interrupt filter) and the task later runs on a dedicated worker thread where sleeping and blocking are allowed. The kernel ships a small set of system-wide taskqueues (`taskqueue_swi`, `taskqueue_thread`, `taskqueue_fast`) and lets you create your own.
- **Grouped taskqueues (gtaskqueue)** extend taskqueue with CPU affinity and rebalancing; they are heavily used in `iflib` and in the high-rate network stack. The declarations live in `/usr/src/sys/sys/gtaskqueue.h`.
- **callout(9)** is the kernel's one-shot and periodic timer. A driver arms a callout with a future deadline and receives a callback when the deadline arrives. `callout(9)` replaces almost every ad-hoc “sleep for N ticks” loop a driver might otherwise write.
- **hooks(9)-style subsystem extension points.** A number of FreeBSD subsystems publish registration APIs that behave like eventhandlers but are specific to the subsystem (for example, packet filters register with `pfil(9)`; disk drivers can register with `disk(9)` events). These are not one unified interface, but the pattern is the same: a list of callbacks that a subsystem invokes at a well-defined moment.

### 43.11.4 Typical driver touchpoints

- `EVENTHANDLER_REGISTER(shutdown_pre_sync, mydev_shutdown, softc, SHUTDOWN_PRI_DEFAULT);` in `attach` so the driver flushes hardware before a reboot; `EVENTHANDLER_DEREGISTER` in `detach`. (The three standard priority constants for shutdown hooks are `SHUTDOWN_PRI_FIRST`, `SHUTDOWN_PRI_DEFAULT`, and `SHUTDOWN_PRI_LAST`, declared in `/usr/src/sys/sys/eventhandler.h`.)
- `taskqueue_create("mydev", M_WAITOK, ...); taskqueue_start_threads(...);` in `attach` to create a per-device worker; `taskqueue_drain_all` and `taskqueue_free` in `detach`.
- `callout_init_mtx(&sc->sc_watchdog, &sc->sc_mtx, 0)` in `attach` to arm a watchdog; `callout_drain` in `detach`.
- Grouped taskqueues are most visible inside `iflib`-based network drivers; a typical standalone driver rarely reaches for them directly.

### 43.11.5 Where to look in /usr/src

- `/usr/src/sys/sys/eventhandler.h` and `/usr/src/sys/kern/subr_eventhandler.c` for event handlers.
- `/usr/src/sys/sys/taskqueue.h` and `/usr/src/sys/kern/subr_taskqueue.c` for taskqueues.
- `/usr/src/sys/sys/gtaskqueue.h` and `/usr/src/sys/kern/subr_gtaskqueue.c` for grouped taskqueues.
- `/usr/src/sys/sys/callout.h` and `/usr/src/sys/kern/kern_timeout.c` for callouts.

### 43.11.6 Manual pages and files to read next

eventhandler(9), taskqueue(9), callout(9), and the header `/usr/src/sys/sys/eventhandler.h`. Look at `/usr/src/sys/dev/random/random_harvestq.c` for a driver that uses SYSINIT and a dedicated kproc cleanly; it is a good companion when reading about kernel services even though it does not itself exercise taskqueue(9) or callout(9).

### 43.11.7 Common beginner confusion

One important trap: forgetting that registration is half of the contract. Every `EVENTHANDLER_REGISTER` needs an `EVENTHANDLER_DEREGISTER` at the matching lifecycle moment, every `taskqueue_create` needs a `taskqueue_free`, and every armed callout needs a `callout_drain` before its memory is freed. A leaked registration keeps a dangling pointer into freed memory; the next invocation of the event will then crash the kernel in a subsystem that has nothing to do with your driver.

### 43.11.8 Where the book teaches this

Chapter 13 introduces callout(9). Chapter 14 is the taskqueue chapter. Chapter 24 is the kernel-integration chapter and covers eventhandler(9) and the SYSINIT/module cooperation in context.

### 43.11.9 Further reading

- **In this book:** Chapter 13 (Timers and Delayed Work), Chapter 14 (Taskqueues and Deferred Work), Chapter 24 (Integrating with the Kernel).
- **Man pages:** eventhandler(9), taskqueue(9), callout(9).

## 43.12 Cross-References: Structures and Their Subsystems

The table below is the fastest way to turn an unfamiliar type into a known subsystem. Use it when you are reading driver source, hit a struct name you do not recognise, and want to know which section of this appendix to open.

| Structure or type   | Subsystem          | Where declared                                                                    |
|---------------------|--------------------|-----------------------------------------------------------------------------------|
| struct proc, thread | Process and Thread | <code>/usr/src/sys/sys/proc.h</code>                                              |
| vm_map_t            | Memory (VM)        | <code>/usr/src/sys/vm/vm.h</code> and<br><code>/usr/src/sys/vm/vm_map.h</code>    |
| vm_object_t         | Memory (VM)        | <code>/usr/src/sys/vm/vm.h</code> and<br><code>/usr/src/sys/vm/vm_object.h</code> |
| vm_page_t           | Memory (VM)        | <code>/usr/src/sys/vm/vm.h</code> and<br><code>/usr/src/sys/vm/vm_page.h</code>   |
| uma_zone_t          | Memory (VM)        | <code>/usr/src/sys/vm/uma.h</code>                                                |

| Structure or type                                          | Subsystem                    | Where declared                                                                     |
|------------------------------------------------------------|------------------------------|------------------------------------------------------------------------------------|
| <code>struct vnode</code>                                  | File and VFS                 | <code>/usr/src/sys/sys/vnode.h</code>                                              |
| <code>struct vop_vector</code>                             | File and VFS                 | generated from<br><code>/usr/src/sys/kern/vnode_if.src</code>                      |
| <code>struct buf</code>                                    | File and VFS                 | <code>/usr/src/sys/sys/buf.h</code>                                                |
| <code>struct bio</code>                                    | File and VFS (GEOM)          | <code>/usr/src/sys/sys/bio.h</code>                                                |
| <code>struct g_provider</code>                             | File and VFS (GEOM)          | <code>/usr/src/sys/geom/geom.h</code>                                              |
| <code>struct cdev</code>                                   | File and VFS (devfs)         | <code>/usr/src/sys/sys/conf.h</code>                                               |
| <code>struct cdevsw</code>                                 | File and VFS (devfs)         | <code>/usr/src/sys/sys/conf.h</code>                                               |
| <code>struct mbuf, m_tag</code>                            | Network                      | <code>/usr/src/sys/sys/mbuf.h</code>                                               |
| <code>if_t, struct ifnet</code>                            | Network                      | <code>/usr/src/sys/net/if.h</code> ,<br><code>/usr/src/sys/net/if_private.h</code> |
| <code>struct socket</code>                                 | Network                      | <code>/usr/src/sys/sys/socketvar.h</code>                                          |
| <code>device_t</code>                                      | Driver Infrastructure        | <code>/usr/src/sys/sys/bus.h</code>                                                |
| <code>driver_t, devclass_t</code>                          | Driver Infrastructure        | <code>/usr/src/sys/sys/bus.h</code>                                                |
| <code>device_method_t</code>                               | Driver Infrastructure (kobj) | <code>/usr/src/sys/sys/bus.h</code> (kobj in<br><code>sys/kobj.h</code> )          |
| <code>struct resource</code>                               | Driver Infrastructure        | <code>/usr/src/sys/sys/rman.h</code>                                               |
| <code>SYSINIT, SI_SUB_*</code>                             | Boot and Module              | <code>/usr/src/sys/sys/kernel.h</code>                                             |
| <code>MODULE_VERSION,</code><br><code>MODULE_DEPEND</code> | Boot and Module              | <code>/usr/src/sys/sys/module.h</code>                                             |
| <code>eventhandler_tag</code>                              | Kernel Services              | <code>/usr/src/sys/sys/eventhandler.h</code>                                       |
| <code>struct taskqueue</code>                              | Kernel Services              | <code>/usr/src/sys/sys/taskqueue.h</code>                                          |
| <code>struct callout</code>                                | Kernel Services              | <code>/usr/src/sys/sys/callout.h</code>                                            |

When a type is not in the table, search `/usr/src/sys/sys/` or `/usr/src/sys/<subsystem>/` for its declaration; the comment near the definition usually names the subsystem outright.

### 43.13 Source-Tree Navigation Checklists

The FreeBSD source tree is organised by responsibility, and once you know the pattern you can guess where almost anything lives. The lists below are the five quick questions that turn “where in the tree” into “open this file”.

### 43.13.1 When you have a structure name

1. Is it a low-level primitive (proc, thread, vnode, buf, bio, mbuf, callout, taskqueue, eventhandler)? Look in /usr/src/sys/sys/ first.
2. Is it a VM type (vm\_\*, uma\_\*)? Look in /usr/src/sys/vm/.
3. Is it a network type (ifnet, if\_\*, m\_tag, route, socket, vnet)? Look in /usr/src/sys/net/, /usr/src/sys/netinet/, or /usr/src/sys/netinet6/.
4. Is it a device or bus type (device\_t, driver\_t, resource, rman, pci\_\*, usb\_\*)? Look in /usr/src/sys/sys/bus.h, /usr/src/sys/sys/rman.h, or the matching bus directory under /usr/src/sys/dev/.
5. Is it something else entirely? `grep -r 'struct NAME {' /usr/src/sys/sys/ /usr/src/sys/kern/ /usr/src/sys/vm/ /usr/src/sys/net/` usually finds it in one pass.

### 43.13.2 When you have a function name

1. If the name starts with vm\_, it lives under /usr/src/sys/vm/.
2. If it starts with bus\_, device\_, driver\_, devclass\_, resource\_, it lives in /usr/src/sys/kern/subr\_bus.c, /usr/src/sys/kern/subr\_rman.c, or one of the bus-specific directories.
3. If it starts with vfs\_, vn\_, or a VOP\_ prefix, it lives under /usr/src/sys/kern/vfs\_\*.c or one of the filesystems under /usr/src/sys/fs/.
4. If it starts with g\_, it is GEOM; look in /usr/src/sys/geom/.
5. If it starts with if\_, ether\_, or in\_, it is networking; look in /usr/src/sys/net/ or /usr/src/sys/netinet/.
6. If it starts with kthread\_, kproc\_, sched\_, or proc\_, it is the process/thread subsystem under /usr/src/sys/kern/.
7. If it starts with uma\_ or malloc, it is memory; look in /usr/src/sys/vm/uma\_core.c or /usr/src/sys/kern/kern\_malloc.c.
8. When nothing matches, `grep -rI '\bFUNC_NAME\s*( ' /usr/src/sys/` is slower but exhaustive.

### 43.13.3 When you have a macro name

1. SYSINIT, SYSUNINIT, SI\_SUB\_\*, SI\_ORDER\_\*: /usr/src/sys/sys/kernel.h.
2. DRIVER\_MODULE, DEVMETHOD, DEVMETHOD\_END, MODULE\_VERSION, MODULE\_DEPEND: /usr/src/sys/sys/bus.h and /usr/src/sys/sys/module.h.
3. EVENTHANDLER\_\*: /usr/src/sys/sys/eventhandler.h.
4. VNET\_\*, CURVNET\_\*: /usr/src/sys/net/vnet.h.
5. TAILQ\_\*, LIST\_\*, STAILQ\_\*, SLIST\_\*: /usr/src/sys/sys/queue.h.
6. VOP\_\*: generated from /usr/src/sys/kern/vnode\_if.src, visible in sys/vnode\_if.h once the kernel is built.

### 43.13.4 When you have a subsystem question

1. What initialises the kernel and in which order? /usr/src/sys/kern/init\_main.c.
2. What drivers does the tree contain? `ls /usr/src/sys/dev/` and its subdirectories.

3. Where are network-stack entry points? `/usr/src/sys/net/if.c`, `/usr/src/sys/netinet/`, and their siblings.
4. How does a specific syscall reach a driver? start in `/usr/src/sys/kern/syscalls.master`, follow the dispatcher into the relevant VFS or socket code, and keep reading until the dispatch lands in a `cdevsw`, a `vop_vector`, or an `ifnet`.

## 43.14 Manual Pages and Source Reading Itinerary

Pattern recognition across the kernel comes from reading it, not only from reading about it. A self-study plan that covers the subsystems in this appendix might look like this:

1. `intro(9)` plus a walk of `/usr/src/sys/sys/` file names, fifteen minutes total.
2. `kthread(9)`, `kproc(9)`, and `/usr/src/sys/sys/proc.h`.
3. `malloc(9)`, `uma(9)`, `bus_dma(9)`, and `/usr/src/sys/vm/uma.h`.
4. `vnode(9)`, `cdev(9)`, `make_dev(9)`, `devfs(4)`, and `/usr/src/sys/dev/null/null.c`.
5. `mbuf(9)`, `ifnet(9)`, `ether_ifattach(9)`, and `/usr/src/sys/net/if_tuntap.c`.
6. `device(9)`, `DRIVER_MODULE(9)`, `bus_alloc_resource(9)`, and `/usr/src/sys/dev/led/led.c`.
7. `SYSINIT(9)`, `kld(9)`, `module(9)`, and the top of `/usr/src/sys/kern/init_main.c`.
8. `eventhandler(9)`, `taskqueue(9)`, `callout(9)`, and `/usr/src/sys/dev/random/random_harvestq.c`.

The companion files in `examples/appendices/appendix-e-navigating-freebsd-kernel-internals/` collect the same itinerary in a form you can print, annotate, and keep next to the machine.

## 43.15 Wrapping Up: How to Keep Exploring the Kernel Safely

Exploring a kernel source tree can feel endless, and it is easy to lose a weekend chasing one interesting thread through ten subsystems. A small set of habits keeps the exploration productive.

Read in short sessions with a specific question. “What does `bus_setup_intr` actually do under the hood” is a good session. “Read the VM” is not.

Keep the map in view. When you jump from a driver into the VFS, remind yourself that you are now in the VFS and that the rules of the VFS apply. When you return to the driver, remind yourself that the VFS stopped at the function boundary. Each subsystem has its own invariants and its own locking discipline, and they rarely carry over.

Write down what you find. A short note like “`bus_alloc_resource_any` in `subr_bus.c` calls into `BUS_ALLOC_RESOURCE` via `kobj` dispatch, which the PCI bus method implements in `pci.c`” is worth more than an afternoon of passive reading. The appendix and its companion files are there to give you anchor points for exactly this kind of note-taking.

Use the safety rails. `/usr/src/sys/dev/null/null.c` and `/usr/src/sys/dev/led/led.c` are tiny. `/usr/src/sys/net/if_tuntap.c` is small enough to read in a sitting. `/usr/src/sys/dev/random/random_harvestq.c` uses real kernel services without hiding behind layers of abstraction. Start from these whenever a subsystem feels too large to approach directly.

And remember that the goal is not to memorise the kernel. It is to build enough pattern recognition that, the next time you open an unfamiliar driver or a new subsystem, the structures, the functions, and the source paths feel



like neighbourhoods you have already walked through. This appendix, together with Appendices A through D and the chapters that teach the pieces in context, is designed to make that feeling arrive sooner.

When the map here is not enough, the book is. When the book is not enough, the source is. And the source is already sitting on your FreeBSD machine, waiting to be read.

## 44 Appendix F: Benchmark Harness and Results

### 44.1 How to Use This Appendix

Several chapters in this book make performance claims. Chapter 15 gives order-of-magnitude timings for `mtx_lock`, `sx_slock`, condition variables, and semaphores. Chapter 28 notes that driver code typically shrinks by thirty to fifty percent when a network driver is converted to `iflib(9)`. Chapter 33 discusses the cost hierarchy of timecounter sources, with TSC at one end, HPET at the other, and ACPI-fast in the middle. Chapter 34 describes the run-time cost of debug kernels with INVARIANTS or WITNESS enabled, and of kernels with active DTrace scripts attached. Every one of those claims is qualified in the text with a phrase such as “on typical FreeBSD 14.3-amd64 hardware”, “in our lab environment”, or “order-of-magnitude”. The qualifiers exist because the absolute numbers depend on the specific machine, the specific workload, and the specific compiler that built the kernel.

This appendix exists so that those qualifiers stand on reproducible ground. For each class of claim, the companion tree under `examples/appendices/appendix-f-benchmarks/` contains a working harness that the reader can build, run, and extend. Where a harness is portable and does not require hardware access, this appendix also reports the measurement it produces on a known machine, so that readers have a concrete number to compare against. Where a harness requires a specific kernel configuration that cannot be assumed to exist on any given reader’s machine, only the harness itself is provided, together with clear instructions for reproducing it.

The aim is not to replace the Chapter 15 or Chapter 34 claims with authoritative figures. It is to let readers see how the claims were arrived at, to let them verify the claims on their own hardware, and to make the results honest about what varies with environment and what does not.

#### 44.1.1 How this appendix is organised

The appendix has five benchmark sections, each with the same internal structure.

- **What is being measured.** One paragraph describing the claim from the chapter and the quantity the harness measures.
- **The harness.** The filesystem location of the companion files, the programming language used, and a short description of the approach.
- **How to reproduce.** The exact command or command sequence the reader runs.
- **Representative result.** A measured value, or “harness only, no result captured” when the harness has not been run by the author.
- **Hardware envelope.** The range of machines over which the result is expected to generalise, and the range over which it is known not to.

The five sections in order are timecounter read cost, synchronization primitive latency, `iflib` driver code-size reduction, and DTrace-with-INVARIANTS-and-WITNESS overhead. A final section on scheduler wakeup

latency points at the existing Chapter 33 script rather than introducing a new one, since the script there is already the harness.

## 44.2 Hardware and Software Setup

Before presenting the benchmarks, a word on the envelope. The numbers in this appendix and the representative results quoted inside it come from two kinds of measurement.

The first kind is **portable measurement**: anything that counts lines of source code, reads the output of a deterministic tool, or otherwise depends only on the FreeBSD source tree and a working compiler. Those measurements produce the same result on any host that has the same source checkout. The iflib code-size comparison in Section 4 is the only portable measurement in the appendix, and its result can be reproduced exactly on any machine with `/usr/src` synchronised to a FreeBSD 14.3-RELEASE source tag.

The second kind is **hardware-dependent measurement**: anything that times a kernel path, a syscall, or a hardware register read. Those measurements depend on the CPU, the memory hierarchy, and the kernel configuration. For every hardware-dependent benchmark in this appendix, the harness is provided, the reproduction steps are precise, and a representative result is quoted only when the author has actually run the harness on a known machine. Where the author has not, the appendix says so explicitly and leaves the results table blank for the reader to fill in.

The fairer phrasing is that this appendix is a **runnable field guide**. The chapters quote qualified figures; the appendix shows you how to measure the same quantities on the hardware in front of you, and what to expect if your hardware belongs to the same broad family as the hardware the chapters had in mind (“modern amd64”, “Intel and AMD generations currently in service”).

### 44.2.1 Caveats that apply to every section

A few caveats apply throughout, and it is simpler to name them once than to repeat them in each section.

All the hardware-dependent harnesses measure averages over large loops. Averages hide tail behaviour. A P99 latency can be an order of magnitude higher than the mean on the same path, especially for anything that involves a scheduler wakeup. Any serious performance claim in production will need a distributional measurement, not a single number; this appendix is about the mean because that is what the chapter claims refer to.

Readers running these harnesses under virtualisation should expect significantly noisier results than on bare metal. A virtualised TSC, for example, may be synthesised by the hypervisor in a way that adds hundreds of nanoseconds to every read. The Chapter 33 cost hierarchy still holds qualitatively under virtualisation, but the absolute numbers will move.

Finally, none of these harnesses are intended for use in production kernels. The sync-primitive `kmod` in particular spawns kernel threads that do tight no-op loops; it is safe to load for a few seconds on a development machine and unload, but it should not be loaded on a busy server.

## 44.3 Timecounter Read Cost

### 44.3.1 What is being measured

Chapter 33 describes a three-way cost hierarchy for timecounter sources in FreeBSD: TSC is cheap to read, ACPI-fast is moderately expensive, and HPET is expensive. The claim is qualified with “on the Intel and AMD generations currently in service”, and separately with “on typical FreeBSD 14.3-amd64 hardware”. The harness in this section measures the average cost of one `clock_gettime(CLOCK_MONOTONIC)` call, which the kernel resolves through whichever source `kern.timecounter.hardware` currently selects, and separately the cost of a bare `rdtsc` instruction as a floor.

The quantity measured is nanoseconds per call, averaged over ten million iterations. The underlying kernel path is `sbinoptime()` in `/usr/src/sys/kern/kern_tc.c`, which reads the current timecounter’s `tc_get_timecount` method, scales it, and returns the result as an `sbintime_t`.

### 44.3.2 The harness

The harness lives under `examples/appendices/appendix-f-benchmarks/timecounter/` and is composed of three pieces.

`tc_bench.c` is a small userland program that calls `clock_gettime(CLOCK_MONOTONIC)` in a tight loop and reports the average nanoseconds per call. It reads `kern.timecounter.hardware` at startup and prints the current source name, so that each run is self-documenting.

`rdtsc_bench.c` is a companion userland program that reads the `rdtsc` instruction directly, using the inline assembly pattern that the kernel uses in its own `rdtsc()` wrapper in `/usr/src/sys/amd64/include/cpufunc.h`. Its output is the cost of the instruction itself, without any kernel overhead.

`run_tc_bench.sh` is a root-only shell wrapper that reads `kern.timecounter.choice` (the list of available sources for the current kernel), iterates over each entry, sets `kern.timecounter.hardware` to that source, runs `tc_bench`, and restores the original setting on exit. The result is a table with one row per timecounter source, ready to compare.

### 44.3.3 How to reproduce

Build the two userland programs:

```
$ cd examples/appendices/appendix-f-benchmarks/timecounter
$ make
```

Run the rotation (requires root to flip the `sysctl`):

```
sh run_tc_bench.sh
```

Or just the direct TSC floor:

```
$./rdtsc_bench
```

#### 44.3.4 Representative result

Harness only, no result captured. The harness has been compiled and its logic reviewed, but the author has not run it on the reference machine while writing this appendix. A reader running the harness on typical FreeBSD 14.3-amd64 hardware should expect the TSC column to report values in the low tens of nanoseconds, the ACPI-fast column to report values several times higher, and the HPET column (if available and not disabled in firmware) to report values an order of magnitude higher still. The absolute numbers will vary with CPU generation, power state, and whether `clock_gettime` is served by the fast-gettime path or falls through to the full syscall.

#### 44.3.5 Hardware envelope

The ordering of the three timecounter sources by cost has been stable across amd64 generations since invariant TSC became standard in the mid-2000s. Readers on different CPU vendors or different microarchitectural generations will see different absolute numbers but the same ordering. On ARM64, there is no HPET or ACPI-fast in the usual sense, and the relevant comparison is between the Generic Timer counter register and the software paths that wrap it; the harness will still run, but only one entry will appear in the table. If `kern.timecounter.choice` on the reader's machine shows a single source, that is itself a useful data point: the system's firmware has restricted the choice, and no rotation is possible.

See also Chapter 33 for the surrounding context, particularly the section on `sbinuptime()` and the discussion of why driver code should avoid reading `rdtsc()` directly.

### 44.4 Synchronization Primitive Latencies

#### 44.4.1 What is being measured

Chapter 15 presents a table of approximate per-operation costs for FreeBSD's synchronization primitives: atomics at a nanosecond or two, uncontended `mtx_lock` at tens of nanoseconds, uncontended `sx_slock` and `sx_xlock` slightly higher, and `cv_wait/sem_wait` at microseconds because they always involve a full scheduler wakeup. The numbers in the table are described as “order-of-magnitude estimates on typical FreeBSD 14.3 amd64 hardware”, with the caveat that they “can move by a factor of two or more across CPU generations”. The harness in this section measures each row of that table directly.

The quantities measured are:

- Nanoseconds per `mtx_lock / mtx_unlock` pair on an uncontended mutex.
- Nanoseconds per `sx_slock / sx_sunlock` pair on an uncontended `sx`.
- Nanoseconds per `sx_xlock / sx_xunlock` pair on an uncontended `sx`.
- Nanoseconds per one-shot `cv_signal / cv_wait` round-trip between two kernel threads.
- Nanoseconds per one-shot `sem_post / sem_wait` round-trip between two kernel threads.

### 44.4.2 The harness

The harness lives under `examples/appendices/appendix-f-benchmarks/sync/` and is a single loadable kernel module, `sync_bench.ko`. The module exposes five write-only sysctls under `debug.sync_bench.` (one per benchmark) and five read-only sysctls that report the most recent result for each benchmark.

Each benchmark times a fixed number of iterations using `sbinuptime()` in `/usr/src/sys/kern/kern_tc.c` for the timestamps. The `mutex`, `sx_slock`, and `sx_xlock` benchmarks run entirely in the calling thread and exercise only the uncontended fast path. The `cv` and `sema` benchmarks spawn a worker `kproc` that signs a ping/pong protocol with the main thread; every iteration therefore includes one wakeup and one context switch in each direction, which is precisely what Chapter 15’s “wakeup latency” column measures.

The module follows the `kmod` pattern used in other chapters of this book, declared with `DECLARE_MODULE` and initialised through `SI_SUB_KLD / SI_ORDER_ANY`. Sources are `/usr/src/sys/kern/kern_mutex.c` for `mtx_lock`, `/usr/src/sys/kern/kern_sx.c` for `sx_slock / sx_xlock`, `/usr/src/sys/kern/kern_condvar.c` for condition variables, and `/usr/src/sys/kern/kern_sema.c` for counting semaphores.

### 44.4.3 How to reproduce

Build the module:

```
$ cd examples/appendices/appendix-f-benchmarks/sync
$ make
```

Load and drive it:

```
kldload ./sync_bench.ko
sh run_sync_bench.sh
kldunload sync_bench
```

`run_sync_bench.sh` runs each benchmark in sequence and prints a small table; individual benchmarks can also be triggered directly by writing a 1 to the corresponding `debug.sync_bench.run_*` sysctl and then reading `debug.sync_bench.last_ns_*`.

### 44.4.4 Representative result

Harness only, no result captured. The `kmod` has been written against FreeBSD 14.3 headers and its logic reviewed against the Chapter 15 table, but the author has not loaded and run it on the reference machine while writing this appendix. A reader running the harness on typical FreeBSD 14.3-amd64 hardware should expect the uncontended `mutex` and `sx` numbers to land in the low tens of nanoseconds, and the `cv` and `sema` round-trip numbers to land in the low microseconds because those paths cross the scheduler twice. Any reader whose numbers are more than a factor of two higher should look at scheduler affinity, CPU frequency scaling, and whether the host is under additional load.

### 44.4.5 Hardware envelope

The Chapter 15 table’s “order-of-magnitude” qualifier is deliberate. Uncontended lock costs track the cost of one or two atomic compare-and-swap operations on the current cache line, which varies with CPU generation and cache topology. Round-trip wakeup costs track scheduler latency, which varies more: a dedicated-CPU server with `kern.sched.preempt_thresh` tuned for low latency can show sub-microsecond round-trips, while a busy multi-tenant machine can see tens of microseconds. The harness reports a single mean per benchmark; readers who need a distribution should extend the module to capture quantiles, or use DTrace `lockstat` probes on the running kernel instead.

See also Chapter 15 for the conceptual framework and the guidance on when each primitive is appropriate.

## 44.5 iflib Driver Code-Size Reduction

### 44.5.1 What is being measured

Chapter 28 claims that on the drivers that have been converted to `iflib(9)` to date, driver code typically shrinks by thirty to fifty percent compared with the equivalent plain-ifnet implementation. Unlike the other benchmarks in this appendix, this one is not a hardware measurement. It is a source-code measurement: how many lines of C source a modern NIC driver needs under `iflib(9)` versus how many it needed without it.

The quantity measured is the line count of a driver’s main source file, split into three figures: the raw `wc -l` total, the number of non-blank lines, and the number of non-blank non-comment lines (an approximation of “code lines” that is close enough to the truth for order-of-magnitude comparison).

### 44.5.2 The harness

The harness lives under `examples/appendices/appendix-f-benchmarks/iflib/` and is a set of portable shell scripts.

`count_driver_lines.sh` takes one driver source file and reports all three figures. The comment stripping is a simple `awk` pass that understands `/* ... */` (including multi-line) and `// ... EOL` forms; it is not a full C parser but is accurate enough to be useful.

`compare_iflib_corpus.sh` is the main driver. It walks two curated corpora and produces a comparison table:

- iflib corpus: `/usr/src/sys/dev/e1000/if_em.c`, `/usr/src/sys/dev/ixgbe/if_ix.c`, `/usr/src/sys/dev/igc/if_igc.c`, `/usr/src/sys/dev/vmware/vmxnet3/if_vmx.c`.
- Plain-ifnet corpus: `/usr/src/sys/dev/re/if_re.c`, `/usr/src/sys/dev/bge/if_bge.c`, `/usr/src/sys/dev/fxp/if_fxp.c`.

The iflib drivers were selected by grepping for `IFDI_` method callbacks, which are the characteristic iflib interface points; the plain-ifnet drivers were selected to span a comparable range of hardware classes. Both lists are variables at the top of the script that the reader can edit.

`git_conversion_delta.sh` is a third script for readers who have a full FreeBSD Git clone with history. It finds the commit that converted a named driver to iflib (by searching commits that touch the file and mention

“iflib” in the log) and reports the line-count delta at that commit. A before/after conversion diff is the only way to measure Chapter 28’s claim directly; the cross-driver comparison is a proxy that depends on driver complexity being roughly comparable, which is a strong assumption.

### 44.5.3 How to reproduce

On any FreeBSD 14.3 source checkout:

```
$ cd examples/appendices/appendix-f-benchmarks/iflib
$ sh compare_iflib_corpus.sh /usr/src
```

For the before/after measurement, on a full FreeBSD Git clone:

```
$ sh git_conversion_delta.sh /path/to/freebsd-src.git if_em.c
```

### 44.5.4 Representative result

Captured against the FreeBSD 14.3-RELEASE source tree, `compare_iflib_corpus.sh` produces the following summary:

```
=== iflib ===
 if_em.c raw=5694 nonblank=5044 code=4232
 if_ix.c raw=5168 nonblank=4519 code=3573
 if_igc.c raw=3305 nonblank=2835 code=2305
 if_vmx.c raw=2544 nonblank=2145 code=1832
 corpus=iflib drivers=4 avg_code=2985

=== plain-ifnet ===
 if_re.c raw=4151 nonblank=3693 code=3037
 if_bge.c raw=6839 nonblank=6055 code=4990
 if_fxp.c raw=3245 nonblank=2943 code=2228
 corpus=plain-ifnet drivers=3 avg_code=3418

=== summary ===
 iflib avg code lines: 2985
 plain-ifnet avg code lines: 3418
 delta: 433
 reduction: 12%
```

A cross-corpus reduction of roughly twelve percent is smaller than Chapter 28’s thirty-to-fifty-percent claim, which is exactly what the caveat at the bottom of the script warns about. The chapter claim is a per-driver before-and-after figure: the same hardware, converted to iflib, goes from N lines to (0.5-0.7) times N lines. A cross-driver comparison is a different thing: it compares different hardware with different feature sets and



different quirk counts. The cross-driver reduction sets a floor (there is *some* reduction in average size across the corpus), and the per-driver reduction quoted in Chapter 28 sets the ceiling (the individual conversion commits show the larger number). Readers with a Git clone can use `git_conversion_delta.sh` to verify the per-driver number directly.

#### 44.5.5 Pinned per-driver measurement

To anchor Chapter 28’s range to a concrete figure, we ran the per-commit comparison on 2026-04-21 against the `ixgbe` conversion commit `4fd3548cada3` (“`ixgbe(4)`: Convert driver to use `iflib`”, authored 2017-12-20 by `erj@FreeBSD.org`). That commit is the cleanest per-driver conversion in the tree: it neither consolidated drivers nor changed features on the same revision. The four driver-specific files that existed on both sides of the commit (`if_ix.c`, `if_ixv.c`, `ix_txrx.c`, and `ixgbe.h`) went from 10,606 raw lines (7,093 non-blank non-comment) down to 7,600 raw lines (5,074 non-blank non-comment) on the conversion commit itself, a twenty-eight percent reduction in code lines. Restricting the comparison to the core PF files (`if_ix.c` and `ix_txrx.c`) tightens the result to thirty-two percent, inside Chapter 28’s thirty-to-fifty-percent range. The twenty-eight-percent figure is the headline one to carry away, with the narrower thirty-two-percent figure showing how much of the residual is shared-header and VF-sibling code rather than framework-saved driver logic.

A larger but less representative data point is the earlier `em/e1000` conversion commit `efab05d61248` (“Migrate `e1000` to the `IFLIB` framework”, 2017-01-10), which achieves roughly a seventy percent reduction on the `e1000`-class driver code: the combined driver sources drop from 13,188 down to 3,920 non-blank non-comment lines. That commit folded `if_em.c`, `if_igb.c`, and `if_lem.c` into a single `iflib`-based `if_em.c` plus a new `em_txrx.c`, so the measured reduction mixes framework savings with the consolidation of three related drivers into one and should not be read as a typical per-driver figure. Taken together, the `ixgbe` and `e1000` data points bracket Chapter 28’s thirty-to-fifty-percent range from below and above: a clean single-driver conversion lands at or just below the lower edge, and a driver-consolidating conversion overshoots the upper edge.

#### 44.5.6 Hardware envelope

This benchmark does not depend on hardware. The result is a function of the FreeBSD source tree at a specific revision, and should be identical on any machine with the same checkout. Readers using a different FreeBSD branch (15-CURRENT, an older release) will see different absolute numbers because the tree evolves.

See also Chapter 28 for the surrounding discussion of `ifnet(9)` and how `iflib(9)` sits inside it.

### 44.6 DTrace, INVARIANTS, and WITNESS Overhead

#### 44.6.1 What is being measured

Chapter 34 makes two performance claims about debug kernels:

- A busy `INVARIANTS` kernel runs roughly five to twenty percent slower than a release kernel, sometimes more on allocation-heavy workloads, as a rough order-of-magnitude figure on typical FreeBSD 14.3-amd64 hardware.
- `WITNESS` adds bookkeeping to every lock acquisition and release; in our lab environment, on a busy kernel running a lock-heavy workload, the overhead can approach twenty percent.

It also mentions, in the context of DTrace, that active DTrace scripts add overhead proportional to how many probes they fire and how much work each probe does.

The quantity measured by the harness in this section is wall-clock time to complete a fixed workload. The harness does not attempt to compute percentages directly; it provides two well-defined workloads and a consistent output format, and it expects the reader to run the suite once per kernel condition and compute ratios between them.

### 44.6.2 The harness

The harness lives under `examples/appendices/appendix-f-benchmarks/dtrace/` and has four parts.

`workload_syscalls.c` is a userland program that performs one million iterations of a tight loop containing four cheap system calls (`getpid`, `getuid`, `gettimeofday`, `clock_gettime`). This workload exaggerates the cost of the syscall entry and exit path, where `INVARIANTS` assertions and `WITNESS` lock-tracking fire most often.

`workload_locks.c` is a userland program that spawns four threads and performs ten million `pthread_mutex_lock` / `pthread_mutex_unlock` pairs per thread, on a small rotating set of mutexes. Userland mutexes fall through to the kernel's `umtx(2)` path when they contend, so this workload exercises the lock-heavy contention path that `WITNESS` instruments.

`dtrace_overhead.d` is a minimal DTrace script that fires on every syscall entry and return without printing anything per probe. Attaching it during a `workload_syscalls` run measures the cost of having the DTrace probe framework actively instrumenting the syscall path.

`run_overhead_suite.sh` runs both workloads once, captures `uname` and `sysctl kern.conf.txt`, and writes a tagged report. The reader is expected to run the suite four times: on a base `GENERIC` kernel, on an `INVARIANTS` kernel, on a `WITNESS` kernel, and on any of those with `dtrace_overhead.d` attached in another terminal. Comparing the four reports gives the percentages Chapter 34 claims.

The surrounding kernel sources are `/usr/src/sys/kern/subr_witness.c` for `WITNESS`, `/usr/src/sys/sys/proc.h` and its siblings for the assertion macros enabled by `INVARIANTS`, and the DTrace provider sources under `/usr/src/sys/cddl/dev/` for the probe framework.

### 44.6.3 How to reproduce

On each kernel condition:

1. Boot into the kernel under test.
2. Build the workloads:

```
$ cd examples/appendices/appendix-f-benchmarks/dtrace
$ make
```

3. Run the suite:

```
sh run_overhead_suite.sh > result-<label>.txt
```

4. For the DTrace condition, launch the script in another terminal before running the suite:

```
dtrace -q -s dtrace_overhead.d
```

After all four runs, compare the `result-*.txt` files side by side. The ratios `INVARIANTS_ns / base_ns`, `WITNESS_ns / base_ns`, and `dtrace_ns / base_ns` are the overhead figures Chapter 34 refers to.

#### 44.6.4 Representative result

Harness only, no result captured. The workloads have been compiled on the reference system and their logic reviewed against the Chapter 34 claims, but the author has not built the three comparison kernels to capture end-to-end numbers while writing this appendix. Readers who run the four-kernel comparison on typical FreeBSD 14.3-amd64 hardware should expect `INVARIANTS` to land in the five-to-twenty-percent range on `workload_syscalls`, somewhat higher on `workload_locks` because lock paths are the hottest code under `INVARIANTS`. `WITNESS` should land lower than `INVARIANTS` on pure-syscall workloads (where few new lock orders are visited) and higher on `workload_locks`, approaching the twenty-percent figure the chapter names. The `DTrace` column should be small on `workload_locks` (no relevant probes fire in the hot path) and non-trivial on `workload_syscalls` (every iteration fires two probes).

#### 44.6.5 Hardware envelope

The ratios are more stable across hardware than the absolute numbers. Readers on different CPUs will see different wall times for the baseline workload, but the per-kernel percentage overhead should stay within a tight band because it is determined by how much extra work the debug kernel does per operation, not by how fast each operation is. The most common sources of surprise are virtualisation (where the syscall path has additional per-call hypervisor overhead that dilutes the percentage), aggressive power management (where CPU frequency varies across runs), and SMT (where per-core numbers differ from per-logical-CPU numbers). The harness does not try to control for any of these; a reader wanting production-quality numbers will need to pin the test to a single CPU, disable frequency scaling, and run the suite several times.

See also Chapter 34 for the surrounding treatment of debugging kernels and for the discussion of when each of `INVARIANTS`, `WITNESS`, and `DTrace` is worth enabling.

### 44.7 Scheduler Wakeup Latency

Chapter 33 contains a `DTrace` snippet that measures scheduler wakeup latency using `sched::wakeup` and `sched::on-cpu`. That snippet is already the harness; this section would only duplicate it. Readers interested in the sub-microsecond idle-system figure and the low-tens-of-microseconds contended-system figure that Chapter 33 names should use the script exactly as printed there, on the reader's hardware. The `DTrace` provider sources are under `/usr/src/sys/cddl/dev/dtrace/`.

If the reader wants to compare wakeup latency with and without `WITNESS`, the same script applies; just boot the two kernels and run it on each.

## 44.8 Wrapping Up: Using the Harness Without Being Misled

The harness in this appendix exists because every number in Chapters 15, 28, 33, and 34 is an order-of-magnitude claim, and order-of-magnitude claims deserve reproducibility. A few habits make the harness actually useful rather than a source of false confidence.

First, run each benchmark more than once. Single-run results are dominated by startup noise, warm-up effects, and interference from whatever else is running on the machine. The harness scripts report a single number; run them three to five times and take the median, or extend them to aggregate runs automatically before trusting the result.

Second, keep the conditions honest. If you want to compare `INVARIANTS` to a base kernel, build both kernels with the same compiler, the same `CFLAGS`, and the same `GENERIC` or `GENERIC-NODEBUG` baseline. If you want to compare two timecounter sources, run both comparisons on the same boot of the same kernel; a reboot between runs changes too many variables.

Third, resist the temptation to treat the numbers as universal. A measurement on one 4-core laptop in a quiet office is not a measurement on a 64-core production server in a noisy rack. The harness measures the machine in front of you; Chapter 15 through 34's qualified phrases ("on typical 14.3-amd64 hardware", "in our lab environment") are honest about the same limitation. The harness makes the qualified phrase testable without pretending it becomes universal.

Fourth, prefer ratios to absolutes. A claim that `WITNESS` costs twenty percent is much more stable than a claim that `WITNESS` costs four hundred nanoseconds per lock. The former survives CPU changes, compiler changes, and kernel-version changes that would wipe out the latter. When you run the harness on your hardware, the percentage overhead is what you should keep; the absolute number is just the arithmetic that produced it.

And finally, extend the harness rather than trusting it blindly. Every script here is small enough to read end to end; every `kmod` is under three hundred lines. If a chapter claim matters to you, open the harness, verify that it measures what you expect it to measure, modify it if your workload needs something different, and run the modified version. The harness is a starting point, not a finish line.

The qualified phrases in Chapters 15, 28, 33, and 34 and the runnable harness in this appendix are two halves of the same discipline. The chapters are honest about what is variable; the appendix shows you how to measure the variation yourself.